# Une approche pour la maintenance et la ré-ingénierie globale des logiciels

Jean-Marie Favre

# THESE

présentée par

**Jean Marie FAVRE**

pour obtenir le titre de

**Docteur de l'Université Joseph Fourier - Grenoble** I

(arrêtés ministeriels du 5 juillet 1984 et du 30 mars 1992)

Spécialité : **Informatique**

# Une approche pour la maintenance et la ré-ingénierie globale des logiciels

date de soutenance : 26 octobre 1995

Composition du Jury :

J.P. Verjus
K. Bennett
J. Estublier
G. Kahn
S. Krakowiak
A. Van Lamsweerde

Thèse préparée au sein
du Laboratoire de Génie Informatique - Institut Imag

à *Carolina* et à notre famille

Je tiens vivement à remercier les membres du jury pour m'avoir fait l'honneur d'y participer et pour le temps qu'ils ont consacrés à cette thèse malgré leurs emplois du temps souvent surchargés :

Monsieur Jean-Pierre Verjus, Professeur à l'Institut National Polytechnique de Grenoble, Directeur de l'Institut de Mathématique Appliquée de Grenoble (IMAG), Directeur de l'INRIA Rhône-Alpes, pour m'avoir fait l'honneur de présider le jury de cette thèse.

Monsieur Keith Bennett, Professeur à l'université de Durham, Directeur du Laboratoire pour la Maintenance à Durham, Editeur du journal international "Software Maintenance", pour son accueil à Durham le mois dernier, pour avoir lu en détails cette thèse bien qu'elle soit écrite dans la langue de Molière et pour être venu d'Angleterre malgré un calendrier bien chargé.

Monsieur Gilles Kahn, Directeur de recherche INRIA, Directeur Scientifique de l'INRIA France, et Monsieur Axel Van Lamsweerde, Professeur à l'Université Catholique de Louvain (Belgique), éditeur du journal international "Transactions on Software Engineering and Methodology" pour avoir consacré de leur temps à rapporter sur ce travail et pour les remarques qu'ils m'ont faites.

Monsieur Jacky Estublier, Directeur de cette thèse, Directeur de recherche CNRS, Directeur de l'équipe ADELE, pour m'avoir accueilli dans son équipe et pour m'avoir toujours fait confiance. Je tiens également à lui témoigner ma reconnaissance pour m'avoir permis de travailler dans d'aussi bonnes conditions.

Monsieur Sacha Krakowiak, Professeur à l'Université Joseph Fourier, Co-directeur du projet Guide et de l'unité de recherche Bull-Imag, pour m'avoir accueilli en stage, il y a déjà bien longtemps, dans le projet Guide, et pour l'honneur qu'il me fait aujourd'hui de participer à ce jury.

Je voudrais aussi remercier tous les membres de l'équipe ADELE qui, de deux personnes lors de mon arrivée, est passée à plus d'une dizaine aujourd'hui. Tous m'ont permis d'une manière ou d'une autre de travailler dans la joie et la bonne humeur...

Je tiens aussi à remercier les membres de la escuela de computaciòn qui m'ont si bien accueilli lors de mon séjour à l'Universidad Central de Venezuela en tant que coopérant. Je garde un très bon souvenir de ces deux années et de l'accueil exceptionnel qui m'a été fait.

Finalement, ou plutôt avant tout, je tiens à remercier ma famille pour sa patiente et son soutient inconditionnel ainsi que *Carolina* pour son courage et son amour ❤❤❤.

A tous, merci !!!

# Introduction

**Alors que l'informatique est résolument tournée vers l'avenir, cette thèse se concentre sur le passé ; non pas par nostalgie mais plutôt parce que le futur des logiciels âgés est une question d'actualité.**

## I.1 Contexte

Présentons tout d'abord le contexte général, puis le contexte spécifique.

### *I.1.1 Contexte général : maintenance, programmation globale et ré-ingénierie*

Trop souvent recherche et industrie s'opposent, tout comme théorie et pratique, comprendre et faire, développer et maintenir, etc.

Cette thèse se réclame comme faisant partie du Génie Logiciel et à ce titre ces oppositions devraient plutôt être vues comme des complémentarités. En effet, *le génie logiciel, comme toute discipline d'ingénierie, est la rencontre entre la science et l'industrie* [Shaw90]. Cette définition marque la direction à suivre, même si l'on a parfois tendance à l'oublier.

Le ***génie logiciel*** est en effet une discipline très large. Selon que l'on se place dans un milieu académique ou un milieu industriel, sa signification est susceptible de changer. Pour certains ce terme fait référence à des méthodes abstraites qu'il est aujourd'hui impossible d'appliquer en pratique. Pour d'autres, il fait référence à des pratiques éloignées de tout fondement scientifique. Quoi qu'il en soit ces deux visions ne sont que les extrêmes d'un spectre voué à être équilibré.

Cette thèse retient trois thèmes du génie logiciel : *la maintenance, la programmation globale* et la *ré-ingénierie*. Ces thèmes correspondent à des préoccupations industrielles particulièrement importantes et pourtant elles sont souvent regardées avec dédain dans le monde académique. Cette tendance est sans doute liée au fait que les notions et les concepts sous-jacents sont particulièrement flous. Mais n'est ce pas justement parce qu'il y a des problèmes que l'on doit chercher des solutions? En fait, ces trois thèmes correspondent en eux même à des oppositions.

La ***maintenance*** s'oppose au développement.
Bien que les problèmes qu'elle pose soient très importants, pendant longtemps ils ont tout simplement été ignorés. Bien qu'aujourd'hui la situation se soit améliorée, la maintenance reste l'une des principales zones d'ombre du génie logiciel. Maintenance et rapiéçage sont encore trop souvent associés dans la réalité.
En génie logiciel, presque tous les efforts pour diminuer les coûts de la maintenance consistent à améliorer la qualité des logiciels développés. Cette approche est indispensable. Pourtant elle est loin d'être suffisante : (1) Tout au long de la maintenance, les modifications successives du logiciel détériore sa qualité initiale. (2) La notion même de qualité évolue dans le temps, tout comme la technologie. (3) Améliorer la qualité des logiciels lors du développement n'est valable que pour le futur ; le problème de la maintenance consiste à considérer le passé et le présent.

La ***programmation globale*** s'oppose à la programmation détaillée.
Dès le début de l'informatique, ce sont les notions de la programmation détaillée qui ont été étudiées : les algorithmes et les structures de données. Après plus de 5 décennies, cette discipline a acquis un niveau de maturité important. Dans ce domaine des théories sont parfois utilisées. Il est par exemple possible de décrire formellement la sémantique d'un langage de programmation.
Au contraire la programmation globale est basée sur des notions floues. On parle de versions, de configurations et d'objets dérivés, sans pour autant que les concepts associés soient toujours les mêmes ; en tout cas ils ne sont que rarement définis formellement. Il est vrai que ce thème a reçu bien peu d'attention par rapport à la programmation détaillée : ce n'est qu'au cours de la dernière décennie que des recherches actives y ont été consacrées.
Aujourd'hui de nombreux systèmes de programmation globale voient le jour. Pourtant les fondements sur lesquels ceux-ci sont basés ne sont pas toujours très clairs. Dans ces conditions comparer et évaluer ces différents systèmes est difficile. Dans bien des cas, l'apparition d'un nouveau système correspond plus à des variations de surface qu'à des améliorations de fond. Cette situation pourrait être comparée, dans le domaine de la programmation détaillée, à la prolifération de nouveaux langages de programmation dans les années 60. Par contre, dans ce dernier cas, ces apparitions ont étés suivies par un processus de classification et de rationalisation.
Remarquons finalement que dans le domaine de la programmation globale, l'état de la pratique est encore dominé par l'utilisation de techniques rudimentaires conçues dans les années 70.

La ***ré-ingénierie*** est liée à la différence entre l'état de l'art et l'état de la pratique. Il s'agit d'une approche pour diminuer le coût de la maintenance. Comme son nom l'indique, l'idée sous jacente est de "refaire" en utilisant des méthodes et des techniques (plus) adaptées. Refaire, non seulement parce que quelque chose a été mal fait, mais aussi parce que la manière de faire a changé. En fait la ré-ingénierie est un support pour l'évolution des logiciels existants mais peut être vue aussi comme un moyen facilitant le transfert de technologie.
La rétro-ingénierie est l'une des composantes de la ré-ingénierie. Comme son nom le suggère, il s'agit de "faire les choses à l'envers". Une telle approche est contestable dans bien des cas ; mais elle peut aussi être utile. Les fonctionnalités offertes par les logiciels âgés ont souvent été affinées au cours de longues années et il peut être préférable d'examiner le logiciel existant que de faire une nouvelle analyse des besoins.

La ré-ingénierie s'est surtout intéressée à la programmation détaillée. Par contre son intersection avec la programmation globale est quasi-inexplorée.

### I.1.2 Contexte spécifique : maintenance en présence de préprocesseurs

Le point de départ de cette thèse est plus spécifique. Nous nous intéressons aux problèmes posés par l'utilisation de préprocesseurs dans le domaine de la programmation globale et de son rapport avec la maintenance et la ré-ingénierie. Le cas du préprocesseur CPP est plus particulièrement étudié.

CPP est le préprocesseur du langage C. Pour les chercheurs il s'agit d'un outil dépassé et pourtant il est cité dans de nombreux articles... Pour beaucoup de programmeurs il s'agit d'un outil du passé mais aussi du présent ; dans le futur, il risque d'être encore présent... L'utilisation extensive et excessive des préprocesseurs rend les programmes impossibles à lire ; pourtant ils sont utilisés pour écrire de grands volumes de code... L'utilisation des préprocesseurs rend la maintenance difficile et pourtant ils sont largement utilisés par les chargés de maintenance... La présence des préprocesseurs est un problème pour les programmeurs, les chargés de maintenance, les constructeurs d'outils et pourtant ils sont toujours là... *Malgré les problèmes occasionnés par les préprocesseurs, aucun support ne leur est dédié...*

Ces paradoxes forment le contexte spécifique de cette thèse. Ce contexte s'insère parfaitement dans le contexte général : les préprocesseurs sont typiquement utilisés pour résoudre les problèmes de programmation globale (problème d'architecture, de variation, etc.). C'est essentiellement dans le cadre de la maintenance qu'ils sont utilisés et qu'ils posent des difficultés. De nombreux logiciels âgés utilisent de tels outils et le problème de leur ré-ingénierie se pose.

*Bien que le cas particulier des préprocesseurs ait été le point de départ de ce travail, ce n'est aujourd'hui que l'illustration d'un discours bien plus général.*

## I.2 Objectifs

Les oppositions et les paradoxes présentés ci-dessus donnent lieu à différents objectifs.

### I.2.1 Objectifs généraux

Trois objectifs généraux sont poursuivis :

***O1*** ***Montrer que la maintenance, la programmation globale et la ré-ingénierie sont des thèmes dignes d'intérêt, autant pour la recherche que pour l'industrie.***

L'effort doit se porter surtout du côté de la recherche : tout au long de la préparation de la thèse, l'expérience nous a montré que les préoccupations qui nous animaient étaient souvent bien éloignées de celles d'autres chercheurs en génie logiciel. Insister sur la problématique et la justifier a été un passage obligé à chaque présentation de nos travaux. Bien souvent cela revient à combattre les préjugés négatifs associés à la maintenance ; car finalement, si depuis des décennies la maintenance est une zone sombre du génie logiciel, c'est entre autre parce qu'elle est accompagnée d'un manque d'intérêt généralisé.

***O2*** ***Clarifier les relations entre maintenance, programmation globale et ré-ingénierie.***

Expliquer clairement ce que recouvre chaque domaine et les relations qui les lient est important pour leur développement.

***O3*** ***Favoriser un processus de rationalisation dans le domaine de la programmation globale.***

Dans le domaine de la programmation détaillée, l'apparition d'un grand nombre de langages a été suivie par un processus de rationalisation. Aujourd'hui on dispose de certains critères pour comparer et classer les langages de programmation. Ceux-ci sont fondés sur des bases théoriques et concevoir un nouveau langage est une activité d'ingénierie dans le sens propre du terme.

Pourquoi ne pas essayer d'atteindre le même degré de maturité dans le domaine programmation globale ? Conceptualiser et formaliser devient un objectif primordial dans ce domaine ; même si c'est un objectif à long terme. Nous chercherons plus particulièrement à montrer que *bien que la pratique de la programmation globale se caractérise par des échafaudages de systèmes hétérogènes, ceux-ci sont basés sur un nombre limité de concepts*.

***O4*** ***Définir et explorer le thème de la ré-ingénierie globale.***

L'intersection entre la programmation globale et la ré-ingénierie est un thème peu étudié. Présenter les concepts sous-jacents est l'un de nos objectifs.

Ces objectifs sont aussi généraux que difficiles à atteindre. O1 est très subjectif. Bien qu'il ne corresponde pas à une démarche habituelle dans une thèse[1], il nous a semblé important qu'il y figure. O2 et O4 sont difficiles à atteindre car les domaines mis en jeu sont plutôt flous. Finalement O3 est un objectif à long terme et notre travail ne constitue qu'un pas dans cette direction. Les deux tiers de cette thèse sont consacrés à ces 4 objectifs.

### *I.2.2 Objectifs spécifiques*

Le reste de la thèse est consacré à deux objectifs particuliers :

***O5*** ***Etudier le rôle des préprocesseurs et montrer qu'il s'agit d'un thème digne d'intérêt.***

Si paradoxe il y a dans l'utilisation des préprocesseurs, il est naturel d'en chercher les causes. La deuxième partie de cet objectif est un sous-ensemble réduit de O1. Montrer qu'un préprocesseur peut être un sujet de recherche actuel représente l'un des défis de cette thèse.

***O6*** ***Faciliter la maintenance et la ré-ingénierie de programmes utilisant des préprocesseurs.***

Cet objectif est une réponse naturelle au dernier paradoxe présenté en I.1.2 (”Bien que les préprocesseurs posent des problèmes de maintenance, ils ne bénéficient d'aucun support...”)

Ces objectifs forment le point de départ de ce travail, ils ne sont que cependant l'argument d'une approche plus générale.

1. Souvent une thèse a pour but de résoudre un problème particulier dans un domaine donné. Justifier l'importance de la problématique du domaine est souvent moindre.

### I.2.3 “Comprendre” ou “Faire”?

Presque tous les buts que nous nous sommes fixés correspondent à une tâche d’*analyse de l’existant ou à une reformulation plutot qu’à la création d’un nouveau système*. Autrement dit, les objectifs de cette thèse se caractérisent plus par le besoin de “comprendre” que de “faire” ; plus par la nécessité d’”analyser” que de “produire”[1].

Revendiquer un tel objectif peut surprendre. En tout cas cela mérite d’être justifié.
Depuis le début du génie logiciel, de très nombreux systèmes ont été proposés. Pourtant bien peu sont restés ; bien peu ont eu un impact majeur dans l’industrie. La maturité de la programmation détaillée est elle liée à la profusion de langages plus ou moins voisins, ou aux travaux menés pour déterminer et formaliser les concepts sous jacents?
“Comprendre” est sans nul doute indispensable.

En génie logiciel, sous estimer l’importance du “faire” serait néanmoins une erreur fondamentale. Bon nombre de propositions apparaissent irréalistes à cause de problèmes d’échelles, notamment dans le domaine de la programmation globale.
“Comprendre” et “faire” sont donc complémentaires.

Si dans cette thèse “comprendre” est privilégié, c’est entre autre pour compenser la tendance actuelle consistant essentiellement à “faire”[2]. L’objectif réel est de “comprendre” pour “faire”, mais l’on ne peut ni *tous* faire, ni *tout* faire... Il s’agit alors d’un compromis. Ici nous avons choisi de “bien comprendre” pour “bien faire” (par la suite).

## I.3 Approches et idées directrices

Quelques idées directrices caractérisent l’approche poursuivie :

***A1 Transformer les oppositions en complémentarités.***

Dans ce document nous opposons différents points de vues. L’objectif est plus de provoquer des réactions et de défendre les alternatives défavorisées que de critiquer les approches existantes. Par exemple, la maintenance est montrée ici comme un problème fondamental. Il est clair à notre esprit que ce n’est pas le centre de l’informatique !

***A2 Etre aussi rigoureux et abstrait que possible.***

Dans l’état actuel des connaissances, l’utilisation de méthodes formelles pour la programmation globale n’est qu’un voeu pieu. A défaut d’être formelle, notre démarche tente d’introduire un peu de rigueur.

Réutilisation et ré-ingénierie sont à la base des trois points suivants. Ces concepts sont appliqués aux connaissances scientifiques et non pas au logiciel.

1. L’objectif spécifique O6 est un cas particulier.
2. Ici il est fait référence à la programmation globale, à la gestion de configurations et à la ré-ingénierie. Les premiers résultats obtenus dans ces domaines se sont conjugués aux besoins industriels ce qui a précipité l’apparition de marchés importants. “Faire” est devenu un enjeu économique qui n’est pas toujours propice à des avancées réelles dans le domaine.

### *A3 Réutilisation : Réutiliser autant que possible[1].*

Une masse colossale de travaux ont été entrepris en informatique. Certains domaines ont maintenant atteint un haut niveau de maturité, en tous cas par rapport à d'autres. Nous pensons plus particulièrement à la programmation détaillée par rapport à la programmation globale. Même si les problèmes que cherche à résoudre chaque discipline sont différents, certaines expériences, concepts et techniques peuvent être utilement réutilisés de part et d'autre. C'est en tout cas ce que nous nous efforçons de montrer tout au long de cette thèse. *Réutiliser plutot que ré-inventer a été une préoccupation permanente*. Cette tâche n'est pas facile car les travaux scientifiques et techniques ne sont pas toujours prévus pour être réutilisés.

### *A4 Réutilisation : Rendre aussi réutilisable que possible.*

Une thèse peut être utile pour résoudre un problème particulier dans un contexte donné, mais on tirera vraiment profit du travail effectué si celui-ci fournit des résultats réutilisables (ce peut être une démarche, une taxonomie, une bibliographie, un état de l'art, des techniques, etc.). Tout comme pour le logiciel, développer pour réutiliser coûte plus cher mais c'est un investissement pour le futur.

### *A5 Rétro-ingénierie : Reconsidérer l'existant sous un nouveau jour.*

Les avancées permanentes dans le domaine de la recherche font que les différents travaux deviennent peu à peu obsolètes. Cela ne signifie par pour autant qu'ils perdent toute leur valeur. Certains continuent à être utilisés longtemps dans l'industrie. D'autres restent importants tout simplement parce qu'ils font partie de l'expérience. L'idée de la retro-ingénierie est d'étudier des travaux existants (éventuellement obsolètes) pour (re)trouver les concepts sous-jacents (concepts éventuellement n'ayant jamais existé sous la forme recherchée). Cette approche est largement utilisée dans cette thèse. C'est le cas par exemple pour les préprocesseurs. Ils sont aussi obsolètes qu'utilisés mais nous prétendons que les étudier peut être instructif (O5)[2].

La préoccupation pour prendre en compte l'existant se retrouve autant dans le thème de cette thèse (la maintenance et la ré-ingénierie des logiciels) que dans l'approche utilisée (la réutilisation et la ré-ingénierie des connaissances (A3, A4, et A5)). Dans les deux cas, rigueur et abstraction sont nécessaires (A2).

1. Dans le domaine de la réutilisation de logiciels, on distingue la "réutilisation pour le développement" du "développement pour la réutilisation". Cette même distinction est faite ici. Par contre ce n'est pas le logiciel que l'on cherche à réutiliser, ce sont des travaux de recherches, des expériences, etc.
2. Plus généralement nous analysons les systèmes de programmation globale pour reformuler les concepts existants ou mettre à jour de nouveaux de nouveaux concepts.

## I.4 Contenu

Le corps de la thèse est composé de 4 chapitres de tailles relativement équilibrées. Par contre leur lecture est plus ou moins rapide et aisée (figure 1).

- **Chapitre I "Contexte"**
  Le contexte et la problématique sont exposés. *Maintenance*, *programmation-globale* et *ré-ingénierie* sont les trois thèmes abordés. On répond aux questions “*quoi ?*” et “*pourquoi ?*” Autrement dit on y trouve d’une part des *définitions* et d’autre part une *analyse* des *problèmes* et de leur *importance*. Par contre aucune information technique n’est donnée. A la fin du chapitre, le concept de *ré-ingénierie globale* est défini, mais en l’absence de références technologiques, celui-ci n’est pas approfondi.
- **Chapitre II "Un modèle abstrait pour la programmation globale"**
  C’est l’une des pièces maîtresses de cette thèse. Son but est de présenter un modèle abstrait qui sera utilisé comme toile de fond pour les chapitres suivants. Ce modèle est si abstrait qu’il ne fait référence ni au génie logiciel, ni même au logiciel. Grossièrement, la théorie des ensembles et des notions de programmation détaillée sont utilisées pour décrire des notions que l’on retrouve dans la programmation globale.
- **Chapitre III "Aspects concrets de la programmation globale"**
  Il se focalise sur le logiciel et présente la technologie de programmation globale en se basant sur le modèle abstrait présenté dans le Chapitre II. L’objectif est de montrer que bien que la pratique se caractérise par des échafaudages de systèmes hétérogènes, seul un nombre limité de concepts intervient. Les différences technologiques entre état de l’art et état de la pratique permettent en fin de chapitre de préciser la problématique liée à la *ré-ingénierie globale*.
- **Chapitre IV "Maintenance et Ré-ingénierie globale en présence de préprocesseurs"**. Contrairement aux autres chapitres qui proposaient une vision large du sujet, celui-ci se focalise sur un point précis : l’utilisation de préprocesseurs. Au point de vue du contenu, on retrouve des similarités avec les 3 chapitres précédents. Problématique, aspects concrets et abstraits se côtoient. Un prototype permettant de faciliter la maintenance et la ré-ingénierie des logiciels est présenté.
- **Conclusion et Perspectives.**
  Le long parcours que représente la lecture de cette thèse se termine naturellement par une conclusion qui résume et analyse les différents résultats obtenus. Des perspectives viennent ensuite car cette thèse ouvre de nouveaux horizons plus qu’elle ne ferme un domaine ; tout au moins elle soulève de nouvelles questions.

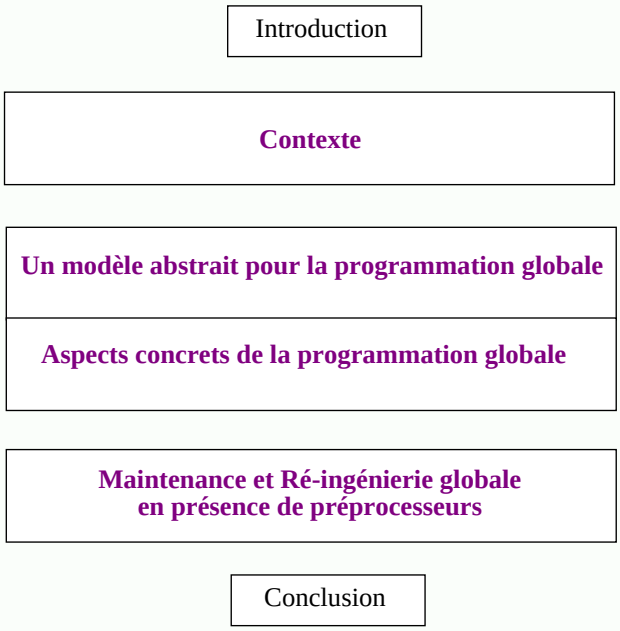
A la fin de ce document, le lecteur trouvera différentes annexes.

- ANNEXE A, "Fondements: Des ensembles aux programmes".
  Bien que cette thèse se concentre sur la programmation globale, les concepts utilisés proviennent de la théorie des ensembles et de la programmation détaillée. Cette annexe regroupe l’ensemble des connaissances qui sont réutilisées dans cette thèse. La plupart des chapitres font référence à cette annexe ; plus particulièrement le Chapitre II. Au point de vue du contenu il s’agit d’un passage progressif de la théorie des ensembles aux fonctions, puis des fonctions aux programmes, pour terminer finalement sur des techniques relatives aux programmes.

- ANNEXE B, "Le prototype APP".
  Cette annexe est un complément au Chapitre IV. Elle contient des exemples de programmes CPP et des exemples de résultats produits avec le prototype proposé.
- ANNEXE B, "Bibliographie".
  La bibliographie en fin de document rassemble de nombreuses références bibliographiques et reflête la variété des domaines qui ont dus être étudiés pour mener à bien ce travail.

Bonne lecture...[1]

*figure 1* Architecture globale du document

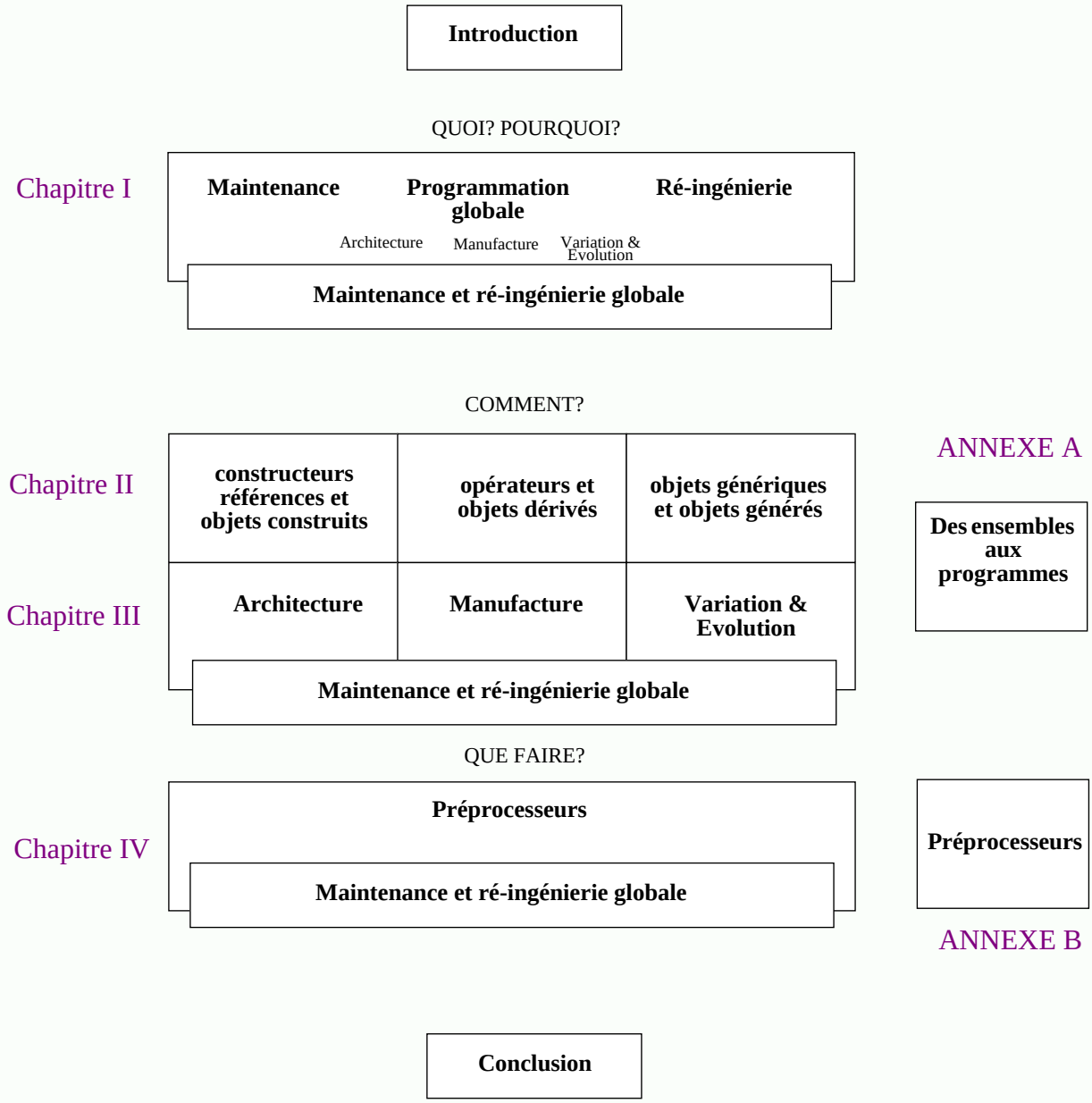



Les différentes figures ont pour but de faciliter la compréhension globale de cette thèse.

Ce document comporte 4 chapitres. La programmation globale est le thème des deux chapitres centraux. Il est abordé d'un point de vue abstrait puis concret.

1. Plus d'informations peuvent être consultées à l'adresse suivante : http://www-lsr.imag.fr/users/Jean-Marie.Favre

*figure 1* Architecture logique détaillée



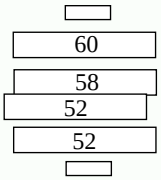

Dans tous les chapitres la préoccupation finale est d'étudier la maintenance et la ré-ingénierie globale des logiciels.

Le Chapitre I présente le contexte général, puis, à partir de là, présente la notion de ré-ingénierie globale. La programmation globale est décomposée en Architecture, Manufacture, Variation & Evolution. Cette décomposition est utilisée pour structurer les deux chapitres suivants.

Le Chapitre II et le Chapitre III s'intéressent aux aspects technologiques de la programmation globale mais respectivement d'un point de vue abstrait et concret. Le Chapitre II présente les fondements du Chapitre III. Ces deux chapitres sont grossièrement décomposés de la même manière. La lecture peut être faite en "largeur d'abord" ou "en profondeur d'abord". On conclue en précisant le concept de ré-ingénierie globale.

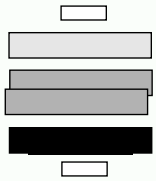
Le dernier chapitre se concentre sur la maintenance et la ré-ingénierie des logiciels utilisant des préprocesseurs et propose des solutions.

*figure 3* Quelques métriques



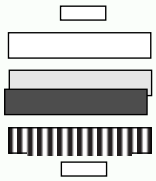

***(a)* Taille**

Tous les chapitres sont de tailles voisines.



***(b)* Général ■ vs. spécifique ■**

La description du contexte est très générale. Les chapitres centraux donnent une vision générale de la programmation globale. Par contre des aspects bien plus spécifiques sont abordés dans le dernier chapitre. On prend notamment en compte certaines spécificités du préprocesseur CPP.



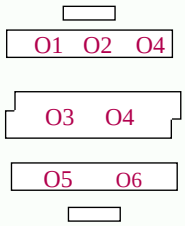
***(c)* Abstrait ■ vs. concret ■**

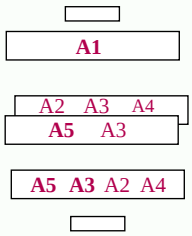
Le chapitre II présente un modèle abstrait. Au contraire le chapitre III décrit des aspects concrets de la programmation globale. Finalement dans le dernier chapitre le cas des préprocesseurs est abordé d'un point de vue très concret mais en utilisant des concepts bien plus abstraits.

**(d)**

***(d)* Répartitions des objectifs**

- O1 Montrer que la maintenance, la programmation globale et la ré-ingénierie sont des thèmes dignes d'intérêt, autant pour la recherche que pour l'industrie.
- O2 Clarifier les relations entre maintenance, programmation globale et ré-ingénierie.
- O3 Favoriser un processus de rationalisation dans le domaine de la programmation globale.
- O4 Définir et explorer le thème de la ré-ingénierie globale.
- O5 Etudier le rôle des préprocesseurs et montrer qu'il s'agit d'un thème digne d'intérêt.
- O6 Faciliter la maintenance et la ré-ingénierie de programmes utilisant des préprocesseurs.




***(e)* Influence des idées directrices dans le document**

- A1 "Transformer les oppositions en complémentarités."
- A2 "Etre aussi rigoureux et abstrait que possible."
- A3 "Réutilisation : Réutiliser autant que possible."
- A4 "Réutilisation : Rendre aussi réutilisable que possible."
- A5 "Rétro-ingénierie : Reconsidérer l'existant sous un nouveau jour."

# Chapitre I

# Contexte

# INDEX

# T



# V

# LISTE DES FIGURES

# CHAPITRE I

# Contexte

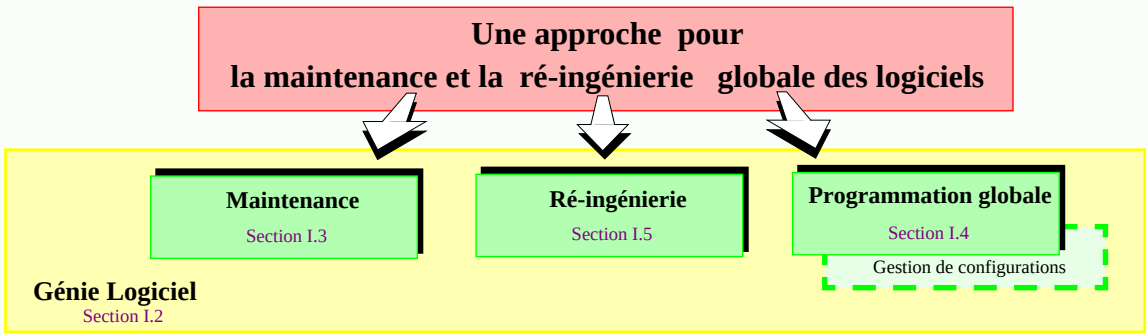
## I.1 Introduction

Ce chapitre a pour but de définir le contexte dans lequel nous nous plaçons et la problématique associée. Pour cela 4 thèmes sont successivement présentés :

- le ***Génie logiciel***,
- la ***Maintenance*** des logiciels,
- la ***Programmation globale***,
- la ***Ré-ingénierie*** des logiciels.

Le choix de ces thèmes est directement lié au titre de cette thèse (figure 4).

*figure 4* Les thèmes d'intérêts




Le *génie logiciel* constitue le cadre général de ce travail et englobe les autres thèmes. La *gestion de configurations* est également un thème important dans cette thèse. Nous étudierons son intersection avec la *programmation globale*.
Bien évidemment ces thèmes ne sont pas traités de manière exhaustive. Il s'agit seulement de faire ressortir leur importance et les points utiles dans le cadre de cette thèse (Les objectifs plus particulièrement visés sont O1 et O2[1]).

La première section est dédiée au *génie logiciel*. C'est essentiellement une introduction aux autres sections. Chaque thème est ensuite abordé en débutant systématiquement par les points suivants :

- **Terminologie**. Les définitions des termes propres au domaine sont présentées ; la définition du terme principal tout d'abord, comme par exemple *maintenance* ; puis la taxonomie associée, comme par exemple *maintenance adaptative*, *maintenance corrective*, etc. Les problèmes de terminologie sont abordés dans un contexte historique et comme nous le verrons ce sera l'occasion d'introduire des discussions parfois fondamentales.
- **Importance**. L'importance du domaine est ensuite soulignée ; premièrement en considérant les coûts liés à la problématique, puis en considérant l'importance des efforts menés dans différents secteurs d'activités : industrie, sociétés de conseils, recherche, etc. Des chiffres issus de différentes sources d'informations sont mentionnés dans l'intention de fournir des ordres de grandeurs et non pas des valeurs absolues.

Après avoir présenté chaque thème, ce chapitre se termine par une présentation de la problématique retenue dans cette thèse.

*figure 5* Architecture logique du Chapitre I.

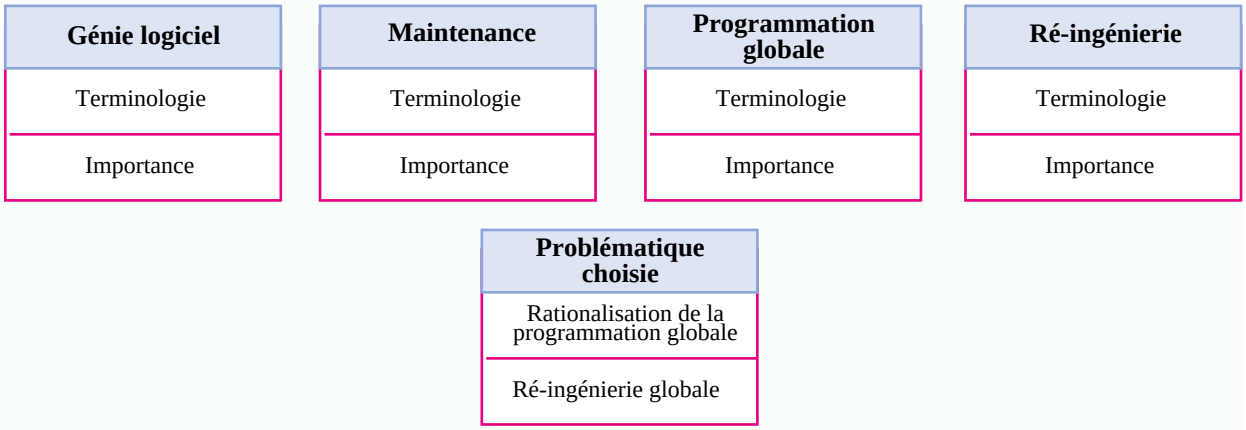
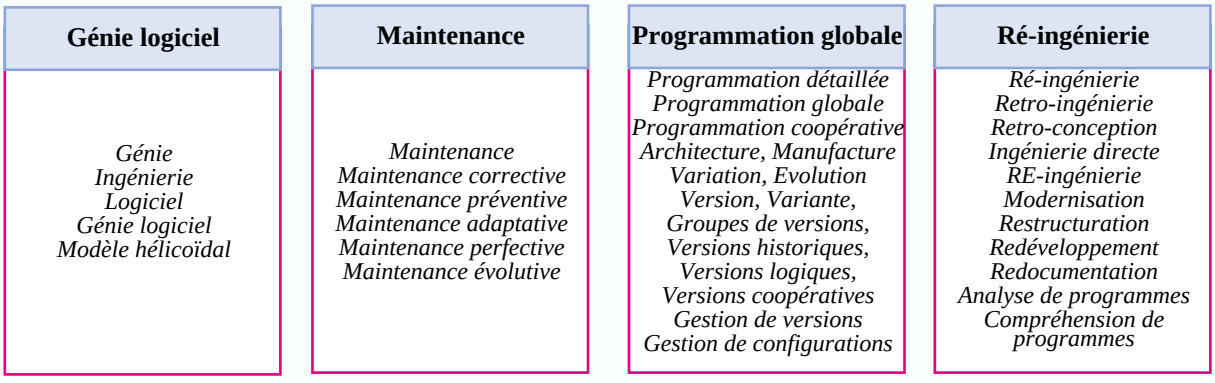



*figure 6* Répartition des concepts




1. O1 Montrer que la maintenance, la programmation globale et la ré-ingénierie sont des thèmes dignes d'intérêt, autant pour la recherche que pour l'industrie.
   O2 Clarifier les relations entre maintenance, programmation globale et ré-ingénierie.

# I.2 Génie logiciel

*“Engineering practice enables ordinary practitioners so they can create sophisticated systems that work - unspectacularly, perhaps, but reliably." (M. Shaw)*

*“Un mathématicien, un physicien et un informaticien montent dans une voiture pour se rendre à une conférence. La voiture démarre, roule pendant quelques kilomètres, puis tombe en panne... Les trois passagers s’inquiètent. Le physicien propose alors une théorie, mais il voudrait la vérifier expérimentalement. Le mathématicien suggère qu’il faudrait commencer par démontrer que la voiture est arrêtée. L’informaticien s’exclame : ‘on pourrait peut être descendre de la voiture et remonter, elle marchera peut être cette fois...’.”*

## I.2.1 Introduction

Au cours des dernières décennies les progrès réalisés dans le domaine des composants électroniques ont été spectaculaires ; tant au point de vue des coûts de production que des performances. Ceci a rendu possible la construction de systèmes informatiques de plus en plus complexes. Le développement de logiciels néanmoins n’a pas suivi cette croissance rapide. Après l’échec de nombreux projets importants il est apparu que l’élaboration de logiciels de grande taille ne pouvait être menée de manière empirique. Le “Génie logiciel” s’est alors dégagé comme un thème essentiel.

## I.2.2 Terminologie

Bien que “Génie logiciel” soit aujourd’hui un terme d’usage courant, son analyse est digne d’intérêt. Les termes “Génie”, “Ingénierie”, “Génie Logiciel” et “Logiciel” sont successivement présentés dans cette section.

### I.2.2.1 *“Génie” et “Ingénierie”*

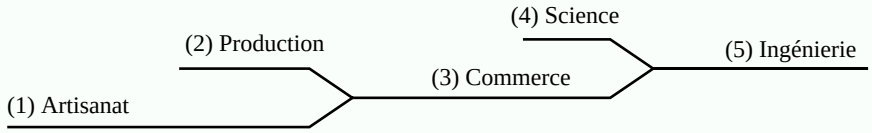
Le terme “Génie” a été utilisé originellement dans le jargon militaire[1]. Par la suite l’industrie de la construction a employé le terme “Génie civil” dans le cadre de l’élaboration d’oeuvres de grande échelle impliquant un travail intensif et des coûts élevés : ponts, routes, canaux, tunnels, etc. Dans le milieu des années 60, le terme “ingénierie” a été introduit en français comme traduction directe du terme anglais “Engineering” [Robe87]. Une définition possible est :

*Ingénierie* :
Proposer des solutions pratiques
en appliquant des connaissances scientifiques
pour construire des produits de qualité
à un coût et dans des délais déterminés.

1. pour désigner l’art d’attaquer et de défendre des places fortes mais aussi le corps des troupes affectées à cette tâche.

M. Shaw propose un ***modèle d'évolution des disciplines d'ingénierie*** [Shaw90], modèle auquel nous ferons souvent référence dans la suite de ce document (figure 7).

*figure 7* Evolution d'une discipline d'ingénierie. [Shaw90].




Ce modèle est composé de 5 phases :

- ***Artisanat***. Différentes personnes cherchent à résoudre des problèmes pragmatiques auxquels ils sont confrontés lors de la production de certains artefacts. Des solutions ad-hoc sont élaborées de manière empirique. Dans cette phase la puissance de travail, l'intuition et le talent des "artisans" sont des facteurs déterminants. Les progressions sont hasardeuses et le manque de communication fait que souvent différentes personnes "ré-inventent" des solutions déjà existantes. Ces solutions se caractérisent par un usage interne supportant une production personnelle.
- ***Production***. Peu à peu, quelques solutions donnant de bons résultats se dégagent. Leur utilisation s'étend et l'on passe à une phase de production où elles sont appliquées dans un contexte plus large.
- ***Commerce***. A un certain point la demande devient plus forte que la production. De nouveaux problèmes interviennent, comme par exemple la disponibilité des ressources et leur utilisation efficace. Il faut alors élaborer un certain nombre de règles. Des connaissances des gestions deviennent nécessaires. Cette évolution rend possible l'exploitation commerciale de ces solutions. Différentes personnes doivent être formées pour soutenir cette production à grande échelle. Elles sont formées de manière pragmatique pour pouvoir répéter des procédures déterminées. Au cours du temps ces procédures sont raffinées, généralement en suivant une méthode essai-erreur. Des considérations économiques plus que techniques mènent ce processus. Elles mettent à jour des points critiques dans l'utilisation des méthodes employées. La phase commerciale se traduit souvent par l'apparition de nombreuses méthodes et techniques "fonctionnant" toutes plus ou moins, sans pour autant que l'on sache déterminer pourquoi, ni comment comparer et évaluer celles-ci.
- ***Science***. Les problèmes pratiques rencontrés dans l'application d'une discipline donnée motivent souvent le développement d'une science associée. De fortes interactions sont bénéfiques entre recherche et industrie et les expérimentations permettent de confirmer les fondements de certaines théories.
- ***Ingénierie***. Lorsque la science est suffisamment mûre, elle est à même d'avoir des retombées pratiques importantes. Les différentes techniques industrielles jusqu'alors utilisées peuvent être revues à la lumière des modèles théoriques. Ceci permet une réduction de leur nombre et une ***rationalisation***. Des solutions unifiées et générales sont élaborées. **La jonction entre science et industrie marque le début de la pratique de l'ingénierie.** La transmission du savoir est essentielle dans cette phase, tout comme la formation d'ingénieurs qualifiés.

Ce modèle d'évolution s'applique à la réalité de différentes disciplines, comme par exemple le Génie Civil, le Génie Mécanique ou le Génie Electrique. De telles évolutions se sont faites au cours des siècles [Shaw90][Babe91]. Aujourd'hui la formation des ingénieurs dans ce domaine s'appuie sur des bases scientifiques solides. Différents ouvrages regroupent un ensemble de règles permettant de choisir la solution la plus adaptée à un problème particulier. Un ingénieur peut ainsi élaborer des constructions complexes et fiables tout en respectant un certain nombre de contraintes. *Dans ces domaines* la pratique de l'ingénierie est une réalité.

### I.2.2.2 *"Génie Logiciel"*

Revenons au "***Génie Logiciel***". Ce terme a été introduit en 1968 pour nommer une conférence traitant des problèmes liés au développement de logiciels[1]. Cette désignation a attiré l'attention et, dans les années 70, elle s'est popularisée.
Jusqu'alors les logiciels étaient essentiellement développés de manière "artisanale" (phase 1 du modèle). Des problèmes de production (phase 2) ont mis à jour le besoin de nouvelles solutions et la nécessité de s'organiser pour mettre en place une réelle industrie du logiciel (phase 3).

En 1976 Boehm propose une définition de Génie logiciel dont une traduction pourrait être : "*appliquer des connaissances scientifiques à la production de programmes et de la documentation nécessaire à leur développement, mise en oeuvre et maintenance*" [Boeh76]. Cette définition est cohérente avec le terme "Ingénierie". Cependant Boehm faisait immédiatement remarquer que peu de connaissances scientifiques étaient disponibles. Bien que des progrès très importants aient été faits depuis (l'utilisation de méthodes formelles par exemple), la construction de logiciel reste un savoir-faire. *Actuellement le terme ingénierie ne correspond pas à la réalité industrielle du logiciel* [Shaw90][Babe91][2].

### I.2.2.3 *"Logiciel"*

Traditionnellement le terme "Logiciel" désigne l'ensemble des programmes indispensables à l'exécution des systèmes informatiques.

Dans les années 70 il est apparu essentiel de ne pas se focaliser uniquement sur ce produit final. Au cours de l'élaboration d'un logiciel un grand nombre de documents doivent également être produits. Il ne s'agit pas de produire des "programmes" mais des "logiciels"!

Pour souligner cette idée, en 1976 la définition de Boehm inclue sous le terme Logiciel "*les programmes et la documentation nécessaire au développement, à la mise en oeuvre et à la maintenance de ceux-ci*".

---

1. Il s'agit d'une réunion de travail internationale à l'initiative de l'OTAN et ayant eu lieu à Garmisch (Allemagne). Pour beaucoup ce fut un point clé dans l'évolution du génie logiciel [Toma94]. Voir par exemple les actes de la conférence ICSE-11 dans lequel on trouve une analyse des résultats de 20 après [Shaw89].
2. M. Shaw conclue par le fait que le terme ingénierie est quelque peu usurpé. C'est également la conclusion du "Software Engineer's Reference Book" [Mcde91a]. Des siècles ont été nécessaires à l'apparition d'une pratique d'ingénierie dans les autres disciplines et il n'est donc pas étonnant que l'informatique, beaucoup plus jeune, n'ait pas encore atteint le même niveau de maturité. Dans le même ordre d'idée, remarquons aussi que les premiers informaticiens étaient des "codeurs". Par la suite, pour éliminer ce caractère mystérieux, on parlait de "programmeur" et aujourd'hui on se plaît à parler d'"ingénieur logiciel" ; même si au fond le rôle de la science dans leur activité professionnelle est bien limité. Le terme "ingénieur système" est sans doute bien plus contestable encore !

Dans les années 80, grâce à la disponibilité d'outils et de méthodes favorisant la production de tels documents, cette définition large de "***logiciel***" a été plus couramment employée [SmitOman90]. C'est ce sens large que nous utiliserons dans ce document. De manière équivalente, nous parlerons aussi de "***produit logiciel***".

### I.2.3 Importance du problème

#### *Des dépenses énormes...*

Selon certaines estimations les investissements faits dans les technologies d'informations (informatique, télé-communications, etc.) représentent plus de 40% des dépenses totales des Etats Unis [Kras91]. Comme nous l'avons mentionné, la part relative du logiciel dans les systèmes informatiques s'est accrue considérablement. Aujourd'hui dans la plupart des systèmes il surpasse le coût du matériel.

Le coût du logiciel n'est pas facile à évaluer au niveau mondial. Il apparaît cependant qu'il se chiffre en centaines de milliards de dollars. Par exemple l'utilisation des ressources humaines dans la production de logiciel consommerait plus de 250 milliards de dollars par an [Fugg93].
Si l'on considère l'ordre de grandeur de ces chiffres, il apparaît clairement que d'un point de vue économique, l'industrie du logiciel est un secteur majeur. Le Génie logiciel prend alors tout son intérêt : une amélioration de productivité, même faible, est susceptible d'entraîner des gains substantiels.

#### *Des résultats trop souvent insatisfaisants...*

Dans les années 70, on parlait de "crise du logiciel". Les projets de grandes envergures se caractérisaient par des retards, des coûts de production sous-estimés, une mauvaise qualité du produit final ou tout simplement des échecs. Aujourd'hui ces risques restent d'actualité. De nombreux projets se soldent par des résultats mitigés . Les logiciels produits ne correspondent pas nécessairement aux besoins des utilisateurs ou ne sont pas considérés suffisamment fiables pour être utilisés dans des situations critiques. Selon certaines estimations plus de 16 milliards de dollars seraient dépensés annuellement pour des systèmes qui ne seront jamais utilisés! [Ulri93].

Ces résultats ne signifient pas pour autant qu'aucun progrès n'ait été fait dans le domaine du génie logiciel. La complexité des logiciels que l'on souhaite construire aujourd'hui est bien plus grande que dans les années 50[1].

#### *Des logiciels de grandes tailles...*

Citons quelques caractéristiques concernant les logiciels de grande taille.

- **Taille des logiciels**. C'est en millions de lignes de code, que se mesure la taille de certains logiciels. Dans un milieu académique ces chiffres surprennent souvent. Faisons simplement remarquer à titre indicatif que X-window représente plus de 3 millions de lignes de code alors qu'il ne s'agit que de services de base. On comprendra aisément que la taille d'une application

1. Le développement du compilateur FORTRAN par IBM dans les années 50 a mobilisé 30 hommes années, ce qui à cette époque a été perçu comme un effort tout a fait remarquable [Gall89].

complète puisse être beaucoup plus importante.
Notons au passage que la complexité d'un logiciel ne dépend pas nécessairement de sa taille en nombre de lignes de code. Par exemple les logiciels temps réels sont souvent beaucoup plus compacts et complexes que les logiciels de gestion.

- **Longue durée de vie**. Il n'est pas rare que des logiciels soient utilisés pendant une décennie. La durée de vie de systèmes militaires peut même dépasser 30 ou 40 ans [Benn91][Benn95].
- **Durée de développement**. Le développement de logiciels complexes peut se faire au cours de plusieurs années.
- **Taille des équipes**. Pour mener à bien la production de tels logiciels, de nombreuses personnes doivent collaborer. La mobilité du personnel en informatique est importante et des dizaines, voire des centaines de personnes peuvent avoir participé au développement d'un logiciel de grande taille.
- **Répartition géographique**. Dans un certain nombre de cas, des équipes de développement sont réparties sur des sites distants. Souvent aussi un système informatique est installé sur des sites différents.
- **Multiples organisations**. Il est extrêmement rare qu'une organisation développe de manière interne la totalité d'un système informatique. En plus de la composante matérielle, il est nécessaire d'utiliser des services logiciels développés par d'autres organisations. C'est souvent le cas par exemple des systèmes d'exploitation, des systèmes de bases de données, etc.
- **Multiples installations**. Pour tirer le plus grand bénéfice d'un logiciel, il faut autant que possible l'installer sur un grand nombre de sites. Il est également nécessaire de l'adapter aux besoins des différents clients.

Ces caractéristiques montrent que l'industrie du logiciel a considérablement changé depuis la dite "crise du logiciel". Face à ces conditions difficiles, cette crise ne s'est pas vraiment dissipée. Simplement aujourd'hui il n'est plus surprenant que la complexité des logiciels soit un facteur limitant l'ambition des systèmes informatiques.

### I.2.4 Importance des efforts

L'importance de l'industrie du logiciel justifie des travaux dans le domaine du Génie logiciel. Les efforts menés sur ce thème ont suivi une progression relativement lente.

L'inéquation des méthodes et techniques ad-hoc s'est révélée dans les années 60. Comme nous l'avons dit en 68 se tenait la première conférence sur le thème du génie logiciel. Vers le milieu des années 70 de nombreux articles traitaient de ce thème et différentes procédures étaient utilisées de manière interne dans certaines organisations. Des outils supportant des méthodes de génie logiciel ont été utilisables dès la fin des années 70, mais ce n'est que vers le milieu des années 80 que la pratique du génie logiciel a commencé à se répandre [RedwRidd85] [Shaw86] [Shaw90]. Aux Etats Unis, le SEI (Software Engineering Institute) a été fondé en 1984 pour favoriser le développement du génie logiciel en améliorant la communication entre l'industrie, les organisations gouvernementales et académiques.

Aujourd'hui, l'importance du génie logiciel est reconnue dans différents secteurs d'activités.

- **Industrie**. Une section "génie logiciel" existe dans la plupart des organisations produisant du logiciel à une grande échelle et leur participation aux conférences sur ce thème est importante.
- **Producteurs d'outils et sociétés de service**. Dans la dernière décennie le marché du génie logiciel s'est considérablement développé et a provoqué l'apparition de nombreuses sociétés de services et producteurs d'outils. A titre d'exemple le marché des ateliers de génie logiciel était estimé à plus de 5 milliards de dollars en 1993 [Fugg93].
- **Recherche**. Dans le milieu académique le génie logiciel est maintenant enseigné et beaucoup d'universités mènent des activités de recherche dans ce domaine. De très nombreux livres sur ce thème sont disponibles. Des "livres de références" et des "encyclopédies" sont apparus récemment [Mcde91a][Marc94a]. Plusieurs revues et journaux internationaux y sont consacrés, tout comme différentes conférences .

## I.2.5 Le modèle hélicoïdal

Le modèle proposé par M. Shaw nous a permis de montrer que le développement des disciplines d'ingénierie était basé sur des interactions fortes entre la recherche et l'industrie. Pour mettre en évidence d'autres aspects de cette interaction, nous allons maintenant définir notre propre modèle que nous appellerons "***modèle hélicoïdal***". Ce modèle permet de représenter à la fois l'évolution de l'industrie et l'évolution de la recherche ; mais surtout, il fait ressortir leur complémentarité. Il sera utilisé et affiné tout au long de cette thèse.

### I.2.5.1 *L'interaction recherche-industrie*

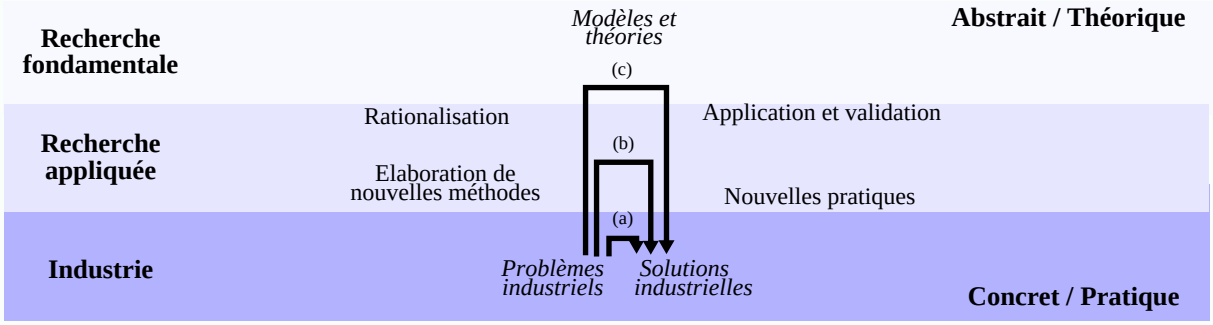
Trois acteurs principaux sont considérés : (1) la ***recherche fondamentale***, (2) la ***recherche appliquée*** et (3) l'***industrie***. La figure 8 représente ces trois acteurs comme différents niveaux manipulant des notions plus ou moins abstraites.
L'*industrie* a des objectifs précis ; les problèmes auxquels elle doit faire face sont spécifiques et concrets. A l'autre extrême, la *recherche fondamentale* se caractérise par la généralité et l'abstraction. De ce point de vue la recherche fondamentale et l'industrie s'opposent [Davi92] [Dene93]. C'est la *recherche appliquée* qui leur permet de communiquer ; elle joue en quelque sorte un rôle d'interface[1].

Ce besoin de communication peut tout d'abord être considéré du point de vue de l'industrie. Il s'agit alors de résoudre des problèmes industriels spécifiques. Plusieurs approches sont possibles selon les acteurs et le niveau d'abstraction mis en jeu (voir la figure 8).

- *(a) Industrie*. Face à un problème spécifique le plus simple est de proposer une solution ad-hoc. Cette approche est la plus courante dans l'industrie car c'est la plus directe.
- *(b) Recherche appliquée*. Des solutions moins directes font intervenir la recherche appliquée. Ses objectifs sont : (1) étudier les coutumes industrielles et les problèmes associés d'un point

1. La décomposition proposée est simplificatrice ; en pratique il s'agit plutôt d'un continuum. Par exemple on aurait aussi pu intégrer les sociétés de conseils s'occupant de transfert de technologie ; différents types d'industries auraient pu être distinguées : industrie de pointe, industrie lourde, etc.

*figure 8* L'interaction recherche-industrie




de vue plus abstrait, (2) élaborer de nouvelles méthodes de production, (3) permettre la mise en place de nouvelles pratiques industrielles.

- *(c) Recherche fondamentale*. Si l'état des connaissances le permet il est parfois possible d'utiliser des modèles et des théories développées dans le contexte de la recherche fondamentale. Le problème est alors traité à un niveau plus abstrait[1].

Ces approches ont des caractéristiques différentes. Alors que les solutions ad-hoc sont rapides, leur efficacité est incertaine. Les solutions basées sur l'utilisation de théories et de modèles sont plus sûres mais plus longues et plus coûteuse à élaborer. Il est donc nécessaire de faire un compromis entre le coût, la rapidité, l'efficacité et les risques. En fait ces approches sont complémentaires : alors que des solutions ad-hoc sont indispensables pour résoudre rapidement les problèmes, pour traiter les problèmes de fond des solutions à plus long terme sont nécessaires.

Le développement de l'industrie dépend des interactions qu'elle a avec la recherche. Inversement, le développement de la recherche fondamentale peut aussi bénéficier de telles interactions. Comme le souligne M. Shaw, la plupart des sciences devraient voir l'industrie comme une source de problèmes intéressants. De même dans la plupart des sciences, les expérimentations sont nécessaires pour confirmer ou infirmer les théories.

Dans un tel contexte, l'ingéniérie se présente comme l'utilisation de principes théoriques déjà connus pour résoudre un problème industriel particulier. C'est donc en quelque sorte la répétition d'une transformation (c) mais tout en conservant les avantages des transformation de type (a).

### I.2.5.2 *Le cycle problème-solution*

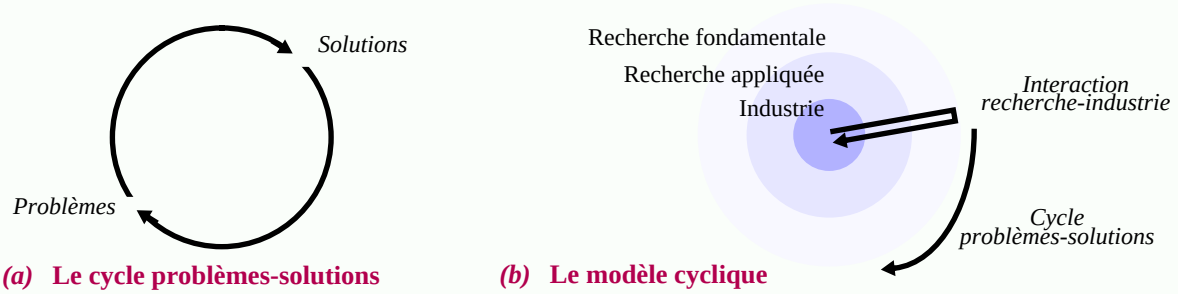
Trop souvent on est en quête de *la* solution. Pourtant il n'existe pas de solution définitive ; en tout cas le fait de trouver une solution n'implique pas la fin des problèmes [Broo87]. Les raisons en sont multiples :

1. A titre d'illustration, prenons le cas de la gestion de configurations (thème développé dans les chapitres suivants). A partir des années 70, l'évolution des logiciels de grande taille s'est peu à peu révélée être un problème industriel majeur. Il est devenu indispensable de gérer les multiples versions créées lors de cette évolution. Des solutions ad-hoc ont été (sont encore) utilisées dans l'industrie. Typiquement il s'agit d'organiser l'espace des fichiers en utilisant des conventions de nommage pour stocker les différentes versions, d'écrire des fichiers de commandes pour les manipuler, etc (approche (a)). Ce n'est qu'à partir des années 80 que la gestion de configurations est devenue un thème de recherche appliquée. Aujourd'hui des solutions sont disponibles et commencent à avoir un impact réel sur les pratiques industrielles (approche (b)). Par contre l'utilisation de théories et de techniques formelles est encore très limitée dans ce domaine. Autrement dit la connexion avec la recherche fondamentale n'est pas encore faite (approche (c)).

- La solution proposée peut être inadaptée, soit parce que le problème a été mal compris, soit parce qu'elle n'est pas applicable en pratique.
- L'application d'une nouvelle solution peut créer de nouveaux problèmes ou mettre à jour des problèmes déjà existants.
- Si une solution est satisfaisante, elle donnera lieu à de nouvelles applications faisant invariablement apparaître de nouveaux problèmes.

On parlera du cycle "problème-solution" pour faire référence à ce phénomène. Combiné à l'interaction recherche-industrie, on obtient le "modèle cyclique" (figure 9).

*figure 9* Le cycle problèmes-solutions et le modèle cyclique




***(a)*** **Le cycle problèmes-solutions**
L'apparition d'un problème donne normalement lieu à la recherche d'une solution. Une fois celle-ci trouvée de nouveaux problèmes apparaissent, d'autres subsistent, etc.

***(b)*** **Le modèle cyclique**
Ce modèle intègre les deux dimensions. La dimension circulaire suggère qu'au fil du temps on assiste a une alternance entre problèmes et solutions. Les différents niveaux d'abstractions de la figure 8 sont représentés circulairement : l'industrie est au centre et la recherche fondamentale à la périphérie.

### I.2.5.3 *Le modèle hélicoïdal*

Le modèle cyclique correspond à une vision plutôt pessimiste car il donne l'impression de "tourner en rond" plus que d'évoluer. En réalité à chaque tour des progrès sont faits. C'est ce qui assure "l'ascension" de ce processus et le rend hélicoïdal plutôt que cyclique.

Le "modèle hélicoïdal" résulte donc de l'intégration de trois dimensions : (1) La dimension circulaire correspond à la répétition des problèmes et des solutions. (2) La dimension latérale représente la communication entre recherche et industrie. (3) La dimension verticale symbolise la maturité acquise au cours du temps.

Grâce à ce modèle, on peut représenter la *trajectoire* d'un thème particulier en utilisant une métaphore physique.
L'industrie se *focalise* sur son *centre* d'intérêt alors que la recherche, *éloignée* de telles préoccupations, a beaucoup plus de *recul*. En fait, l'industrie est animée d'une *force centripète* qui l'*attire* invariablement vers des aspects *centraux*.
La *distance* prise par la recherche fondamentale lui laisse une plus grande *liberté d'action*. Elle est soumise à une *force centrifuge.* Ces caractéristiques se retrouvent d'ailleurs dans les clichés habituels opposant la recherche et l'industrie. Dans le monde de la recherche on a tendance à penser que "l'industrie tourne en rond" alors que dans l'industrie certains pensent que "les chercheurs planent" (figure 9).
Quant à la recherche appliquée, elle est tantôt *attirée* par un problème précis, tantôt par des

aspects plus théoriques. Pour qu'elle puisse jouer son rôle d'interface, elle doit d'une part se *détacher* des problèmes industriels spécifiques et d'autre part sélectionner les apports de la recherche fondamentale pouvant avoir des *retombées* sur l'industrie. Un tel *va et vient* ne peut se faire qu'en dépensant de l'*énergie*.

*figure 10* Le modèle hélicoïdal

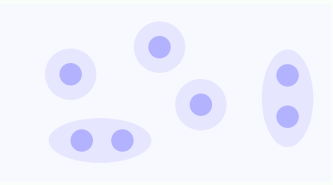

**(a) Vue de coté** **(b) Vue de dessus**

Graphiquement il apparaît que la distance à parcourir entre un problème et une solution est bien plus courte dans le cas de l'industrie que dans le cas de la recherche fondamentale. Ceci s'interprète par le fait que des solutions ad-hoc sont plus simples à obtenir mais font apparaître plus rapidement des problèmes. Inversement, utiliser des techniques formelles est plus long mais permet de traiter des problèmes de fond.

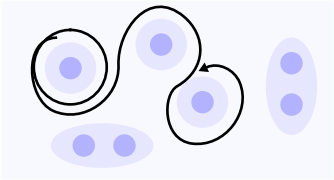
### I.2.5.4 *Echanges inter-disciplinaires*

En fait, jusqu'à maintenant nous avons supposé que l'industrie avait un seul centre d'intérêt. Si l'on prend en compte plusieurs secteurs d'activités l'espace est plus complexe (figure 11). Plus le un niveau d'abstraction est élevé, plus les centres d'intérêt semblent proches. Par exemple des activités de recherche appliquée peuvent regrouper différents secteurs d'activités. De même, des résultats de recherche fondamentale peuvent être réutilisés dans différents travaux de recherche appliquée. En fait, plus une trajectoire s'éloigne d'un centre de gravité, plus il est probable qu'elle puisse s'intégrer à une autre. Ce phénomène permet les échanges inter-disciplinaires et favorise une évolution rapide grâce à la réutilisation de connaissances[1] (figure 11).

*figure 11* Echanges inter-disciplinaires



Comme dans les autres figures l'industrie est représentée par les cercles les plus foncés. La recherche fondamentale est en fait l'espace englobant. La figure de droite montre que les trajectoires des travaux de recherche fondamentale permettent une communication inter-disciplinaire.



1. Par exemple la conception assistée par ordinateur et la gestion bancaire sont deux secteurs d'activités industriels distincts, pourtant la modélisation des données peut être un thème de recherche commun. Les modèles de données peuvent être basés sur la théorie des ensembles, théorie pouvant elle-même être réutilisée dans d'autres contextes.

#### I.2.5.5 *Evolution dans le temps*

L'espace présenté ci-dessus n'est pas statique. Tout au moins ses caractéristiques ne sont pas connues à priori : de nouveaux centres d'intérêt peuvent apparaître, les trajectoires possibles évoluent dans le temps...
Le modèle d'évolution de l'ingéniérie de M. Shaw est complémentaire : il permet de délimiter la zone d'attraction de laquelle une trajectoire ne peut s'échapper. Au cours du temps, cette zone s'agrandit progressivement et on peut donc visualiser cette enveloppe infranchissable comme un *cône* centré sur un sujet déterminé. Dans la phase artisanale, seules des solutions industrielles sont possibles car l'on n'est pas en mesure de se détacher des spécificités des problèmes. La maturité aidant, il devient peu à peu possible de lutter contre la force centripète et même d'atteindre un niveau où des connaissances scientifiques peuvent être réutilisées (phase d'ingéniérie). Parallèlement des connexions parfois insoupçonnées apparaissent et permettent la communication avec d'autres domaines (les cônes s'entrecroisent). Aujourd'hui, en informatique ce phénomène tend à s'amplifier. C'est sans nul doute une signe de maturité[1].

Il est important de souligner que les différents processus décrits ci-dessus sont beaucoup plus lents qu'on pourrait le penser à priori. Rappelons par exemple que le développement des différentes disciplines d'ingénierie s'est fait au cours des siècles [Shaw90][Babe91]. Il n'est donc pas surprenant de constater que, même après plus de trois décennies, le terme "génie logiciel" ne soit pas adapté à la réalité industrielle.

Il est également intéressant de donner quelques idées sur le temps que peut prendre un cycle problème - solution faisant intervenir la recherche. Selon une étude réalisée sur différents cas de figure, l'ordre de grandeur d'une telle période s'exprimerait en décennies [RedwRidd85]. Plus précisément entre 15 et 20 ans seraient nécessaires pour passer de la définition claire d'un problème à une évolution conséquente des pratiques industrielles .

### I.2.6 Conclusion

Dans cette section nous avons présenté ce que recouvrait le terme génie logiciel. Son inadéquation a été soulignée : les pratiques industrielles résultent encore trop souvent de démarches empiriques. En tout cas pour mériter le qualificatif d'*ingéniérie*, il faudra que la recherche et l'industrie collaborent étroitement.

Historiquement, la taille et la complexité croissantes des logiciels a été l'un des catalyseurs expliquant l'apparition de nouvelles préoccupations : schématiquement, la nécessité du génie logiciel est apparue au cours des années 60, le problème de la maintenance dans les années 70, l'intérêt pour la programmation globale dans les années 80 et finalement ré-ingénierie dans les années 90 . Ces thèmes sont successivement abordés dans les sections qui suivent.

---

1. Par exemple la gestion de versions est un thème commun à de nombreux domaines d'application. Pendant très longtemps les recherches ont été presque totalement déconnectées. Aujourd'hui des approches plus générales voient le jour et même si une unification des concepts n'a pas encore été atteinte il est clair que les échanges sont bénéfiques. Dans un autre ordre d'idée remarquons aussi que le paradigme objet est utilisé sous différentes formes dans de très nombreux contextes (bases de données, langages de programmation, méthodes de conception et de spécifications, représentation des connaissances, etc.).

# I.3 Maintenance

*“Programs, like people, get old. We can’t prevent aging, but we can understand its causes, take steps to limits its effects, temporarily reverse some of the damage it has caused, and prepare for the day when software is no longer viable. A sign that the Software Engineering profession has matured will be that we lose our preoccupation with the first release and focus on the long term health of our products. “(D.L. Parnas).*

*“Unfortunately, the nature of hardware and software errors differs in at least one fundamental characteristic -- hardware deteriorates because of lack of maintenance, whereas software deteriorates because of the presence of maintenance” (Fisher & Standish).*

## I.3.1 Introduction

Traditionnellement le génie logiciel s’est surtout tourné vers la production de logiciels, ou plus exactement le *développement* de logiciels. En fait, il apparaît que cette étape s’achève par la livraison de produits imparfaits. De plus, les besoins des clients évoluent tout comme l’environnement du logiciel. Il est donc nécessaire de “maintenir” les produits logiciels.
Dans cette thèse nous nous concentrons sur la maintenance des logiciels et non pas sur leur développement.

*figure 12* Développement ∪ Maintenance


## I.3.2 Terminologie

Comme nous allons le voir le terme “Maintenance” est facile à définir. Par contre l’utilisation de ce terme a été largement contestée au cours des années.

### I.3.2.1 *“Maintenance”*

La *maintenance* est “l’ensemble des activités effectuées sur un logiciel après sa livraison ou sa mise en opération” [Pari86b]

Dans son sens le plus général le terme “maintenance” correspond normalement au “maintien d’un matériel technique en état de fonctionnement”[1] [Robe87].

1. “maintenance” fait partie du vieux français et est synonyme de “confirmation” (XII$^{e}$ siècle). En fait, l’utilisation actuelle provient d’un un anglicisme récent dans le jargon militaire (XX$^{e}$ siècle) [Robe87].

Les concepts que recouvrent ces deux définitions ne sont pas équivalents :

- Déjà à sa mise en service le logiciel ne fonctionne pas correctement. Des erreurs sont toujours présentes. Il ne s'agit pas de préserver un fonctionnement correct mais plutôt d'éliminer les dysfonctionnements existants.
- Les systèmes physiques se détériorent quand ils sont utilisés ; leur maintenance a pour but de pallier à cette dégradation. Au contraire un logiciel ne s'use pas lors de son utilisation ; c'est l'application de la maintenance qui le détériore[1].
- Au cours de la durée de vie d'un logiciel les besoins des utilisateurs peuvent varier. C'est le cas aussi du matériel supportant le système informatique. La maintenance d'un logiciel consiste surtout à l'adapter et à l'étendre lorsque nécessaire. Dans la vie courante ajouter une nouvelle aile à un bâtiment n'est pas considéré comme un acte de "maintenance".

Comme l'ont fait remarquer de nombreux auteurs, le terme "maintenance" est bien mal choisi pour les logiciels [GlasNois81] [Wein83] [Pari86a] [Schn87], surtout que cette désignation a souvent une connotation péjorative. Pour beaucoup d'informaticiens la maintenance "c'est ce qui se fait après que le travail soit fait".
La définition de la maintenance est basée sur l'occurrence d'un événement, la livraison ou la mise en service. Cette barrière est quelque peu arbitraire car elle n'implique pas un changement radical d'activités.

Les cycles de vie du logiciel introduits dans les années 70 ont été bénéfiques du point de vue du génie logiciel, mais pas en ce qui concerne la maintenance [Schn87]. Les premiers cycles de vie ne mentionnaient d'ailleurs même pas la maintenance. Ce n'est que dans le milieu des années 70 que celle-ci est apparue, comme une petite case ajoutée en fin de cycle, pour dire "ici le logiciel est maintenu" [GlasNois81].

Différentes propositions ont été faites pour éviter d'utiliser le terme maintenance, par exemple "Software support" ou "Continued developpement" [Pari86b]. En pratique seul le terme "Software evolution" est utilisé [Lehm80], et, comme nous le verrons, il correspond effectivement à la réalité. Quoi qu'il en soit, le terme "Maintenance" étant le terme consacré, c'est celui que nous utiliserons dans cette thèse.

### I.3.2.2 *"Maintenance corrective, adaptative, perfective et évolutive"*

Pour clarifier les raisons amenant à modifier le logiciel au cours de la maintenance Swanson a proposé une taxonomie introduisant les termes "Maintenance corrective", "Maintenance Adaptative" et "Maintenance Perfective" [Swan76]. Par la suite d'autres termes ont été introduits comme par exemple "Maintenance évolutive" et "Maintenance préventive"[2].

1. Plus de détails seront donnés dans la Section I.3.3.
2. A la lecture d'un article, il est vivement recommandé de vérifier les définitions utilisées car des taxonomies contradictoires sont utilisées. Par exemple dans [Dart92] la maintenance préventive correspond à la maintenance perfective, la maintenance adaptative regroupe la maintenance évolutive et la maintenance adaptative. Dans [RamaPU84] la maintenance préventive n'est pas un sous-ensemble de la maintenance, etc.

#### *“Maintenance corrective”*

La ***maintenance corrective*** est l’ensemble des activités de maintenance déclenchées par la détection d’une erreur.

Notons tout d’abord que ce type de maintenance n’existerait pas si l’on était capable de développer des logiciels sans erreurs. Notons aussi que de la maintenance corrective ne génère pas de nouveaux profits. Elle est cependant indispensable.

#### *“Maintenance adaptative”*

Le logiciel n’est pas une entité isolée. Au contraire il dépend :

- *de son environnement matériel et logiciel* ; (on parle aussi de ***plate-forme*** matérielle et logicielle). Il s’agit par exemple du matériel supportant son exécution ou d’un système d’exploitation. Le logiciel peut dépendre également d’autres services logiciels, comme par exemple un gestionnaire de fenêtres, une base de données, etc.
- *de l’environnement propre à l’organisation dans lequel il est employé et des préférences de ses utilisateurs*. Par exemple la langue dans laquelle sont exprimées les interactions avec l’utilisateur est susceptible de varier tout comme le type d’interface utilisé.

La ***maintenance adaptative*** est l’ensemble des activités de maintenance déclenchées par une modification de l’environnement du logiciel.

Le but d’une activité de ***portage***[1] est d’adapter un logiciel à une plate-forme spécifique. La ***portabilité*** est une qualité du logiciel exprimant la facilité avec laquelle celui-ci peut être “porté” sur une nouvelle plate-forme.

#### *“Maintenance perfective”*

Tout en gardant les même fonctionnalités, il peut être important de modifier certaines qualités du logiciel ; par exemple sa rapidité d’exécution, ou encore sa facilité d’utilisation.

La ***maintenance perfective*** est l’ensemble des activités de maintenance menées pour améliorer une ou plusieurs qualités du logiciel.

Ce type de maintenance n’est pas toujours indispensable. Avant d’entreprendre une modification il est donc important de comparer le coût de l’opération avec les bénéfices attendus.

#### *“Maintenance évolutive”*

La ***maintenance évolutive*** est l’ensemble des activités de maintenance déclenchées par la modification des besoins des utilisateurs.

Il s’agit là de modifier ou d’étendre les fonctionnalités du logiciel. Les causes en sont multiples. Tout d’abord il n’est pas évident que le cahier des charges élaboré lors du développement du logiciel corresponde aux besoins réels des utilisateurs. Au cours de l’utilisation du système

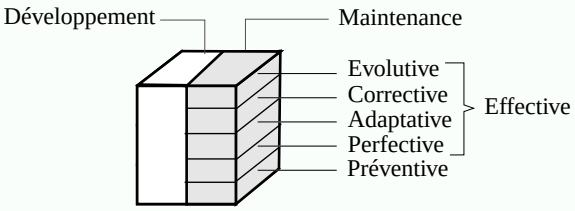
1. Portage est un anglicisme.

informatique, les besoins peuvent être affinés. De plus, il peut arriver aussi que d'autres logiciels offrant plus de fonctionnalités apparaissent sur un marché compétitif. Enfin, les besoins des utilisateurs peuvent tout simplement changer pour des raisons externes[1].

### I.3.2.3 *"Maintenance effective vs. maintenance préventive"*

En fait les activités de maintenance peuvent être découpées en maintenance effective et maintenance préventive (figure 13).

*figure 13* Taxonomie des activités de maintenance




#### *Maintenance effective*

La *maintenance effective* regroupe la maintenance corrective, adaptative, perfective et évolutive.

Ce terme est introduit uniquement dans le but de simplifier la suite de l'exposé. Il s'oppose à maintenance préventive et nous avons choisi ce terme pour souligner le fait qu'il regroupe les activités de maintenance ayant un but direct et des répercutions immédiates.

#### *Maintenance préventive*

La *maintenance préventive* est l'ensemble des activités de maintenance ayant pour but d'améliorer la facilité de maintenance des logiciels.

Notons tout d'abord qu'avec cette définition les activités de maintenance préventive se déroulent lors de la phase de maintenance. Ce n'est pas le cas avec la définition que l'on trouve dans [RamaPU84] ; elle désigne l'ensemble de toutes les activités menées tout au long du cycle de vie pour faciliter la maintenance.

La taxonomie originale de Swanson n'introduit pas cette classe [Swan76] et selon les définitions qu'il donne ce serait une sous classe de la maintenance perfective[2]. Si dans cette thèse nous avons choisi de la présenter à part c'est que, comme nous le verrons par la suite, il s'agit d'un sous-ensemble de la ré-ingénierie.

Ce type de maintenance ne génère aucun bénéfice direct. De telles activités ont uniquement pour but de faciliter, dans le futur, les autres activités de maintenance. La maintenance préventive n'est

1. Soulignons au passage que ce sont les activités de maintenance évolutive qui rendent le terme "maintenance" peu adapté. C'est d'ailleurs dans ces conditions que le terme "évolution" est parfois utilisé comme substitut. En fait certaines définitions du terme maintenance excluent tout simplement les activités de maintenance évolutive : "Maintenance: Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment", Norme ANSI/IEEE 729, 1983
2. La facilité de maintenance n'est qu'une qualité particulière du logiciel. La maintenance préventive pourrait alors être vue comme une spécialisation de la maintenance perfective. Ici au contraire la distinction est faite.

pas déclenchée par un événement externe mais plutôt par une décision prise dans les services de maintenance.

#### I.3.2.4 *Intérêt de la taxonomie*

En réalité les différentes classes présentées ne sont pas strictement disjointes. L'avantage majeur d'une telle taxonomie est qu'elle identifie différentes raisons poussant à modifier le logiciel lors de la maintenance. Ceci est important car dans beaucoup d'esprits, la maintenance est trop souvent assimilée à la maintenance corrective : "corriger les erreurs faites".
En fait, différentes études ont montré que la maintenance corrective représente moins de 20% des efforts totaux de maintenance. La part de la maintenance adaptative et perfective sont également du même ordre de grandeur. Par contre la maintenance évolutive représente environ 40%[1] .
Par ailleurs, pendant longtemps la maintenance préventive n'a représenté qu'une part mineure, voire négligeable face à la maintenance effective[2]. Plus récemment le concept de ré-ingénierie s'est popularisé. Schématiquement il s'agit d'effectuer des transformations majeures d'un logiciel en phase de maintenance, pour faciliter sa maintenance future. Cette démarche, typique de la maintenance préventive, tend à en faire augmenter l'importance relative [Arno93][Ulri93].

### I.3.3 Importance du problème

La maintenance est l'un des problèmes majeur en génie logiciel. Pourtant relativement peu d'informations sont diffusées quant à son amplitude réelle et quantitative. Les informations présentées ci-dessous sont la synthèse de nombreuses sources.

#### *Beaucoup de logiciels à maintenir...*

Au cours du temps de très nombreux logiciels ont été développés et beaucoup de secteurs d'activités dépendent aujourd'hui du fonctionnement de ceux-ci. Chaque année de nouveaux systèmes informatiques sont mis en place et il est donc nécessaire de maintenir de plus en plus de logiciels. Pour donner un ordre de grandeur citons simplement que le DoD[3] maintient actuellement des centaines de systèmes d'informations hétérogènes répartis géographiquement sur plus de 1700 sites, pour un total de 1.4 milliard de lignes de code [AikeMR94].

#### *Des logiciels à maintenir pendant longtemps...*

La durée de vie des logiciels est de plus en plus longue. Certains systèmes de grandes tailles ont déjà plus de 20 ans et l'on espère pouvoir les maintenir pendant encore 15 ou 20 ans [Benn91][Arno93]. Le temps passé à maintenir un logiciel donné est donc bien supérieur au temps passé à le développer.

1. Différents auteurs rapportent des chiffres de cet ordre de grandeur. Le lecteur pourra par exemple consulter [GlasNois81] et [LeluSalo86] pour des chiffres plus précis. Rappelons cependant qu'il est important de comparer la sémantique des taxonomies utilisées.
2. Ceci explique pourquoi elle n'est pas mentionnée dans beaucoup de taxonomies.
3. United State Departement of Defense

#### Des coûts directs élevés...

Dans les années 70 la répartition des coûts entre le développement et la maintenance a surpris bon nombre de chefs de projets. Il est en effet apparu que contrairement aux espérances, la maintenance consommait la majeure partie du budget dédié au logiciel. De plus cette proportion augmenterait . Selon le Centre pour la maintenance de Durham, au Royaume-Uni en 1986, la maintenance représentait deux tiers du budget total dédié au traitement d'informations.
Ce rapport peut être bien supérieur dans certains cas. Par exemple, il a été estimé que pour l'un des systèmes de l'US Air Force, 30 dollars ont été dépensés par instruction lors du développement, puis 4000 dollars tout au long de la maintenance [Boeh76].
Plus généralement on estime que pour un projet de grande taille le rapport des coûts entre maintenance et développement est entre 50% et 80%.

#### Des coûts indirects...

Les coûts mentionnés ci-dessus correspondent aux coûts directs de la maintenance. Cependant à ceux-ci, déjà fort élevés, s'ajoutent toute une série de coûts indirects liés à des répercussions défavorables de la maintenance. Ces coûts, plus difficilement évaluables, sont néanmoins réels et ils contribuent pour une large part au problème de la maintenance. Il s'agit entre autre des conséquences de la détérioration de la qualité des logiciels et de l'insatisfaction des clients.

#### La détérioration des logiciels...

La détérioration des logiciels est une conséquence de la maintenance [LehmBela85] [SchwStra93] [Parn94]. Tout d'abord cette détérioration peut, à terme, impliquer la mort du logiciel ; soit parce qu'il n'est plus possible de l'adapter, soit parce qu'il n'est plus suffisamment fiable pour rester en activité. Les bénéfices tirés de ce logiciel en sont alors à leur fin et il est nécessaire d'engager de nouvelles dépenses pour développer un nouveau logiciel.
La dégradation de la qualité du logiciel due à la maintenance peut aussi avoir des conséquences catastrophiques. Selon Weinberg, parmi la liste des erreurs de programmation les plus coûteuses, les 10 premières sont des erreurs de maintenance. La première aurait coûté plus d'un milliard de dollars [Wein83]. Contrairement au développement, lors de la maintenance c'est un système en activité que l'on modifie et les conséquences d'une erreur peuvent être immédiates...

#### Des clients insatisfaits...

L'insatisfaction des clients constitue un autre coût indirect. Les équipes chargées de maintenir un logiciel reçoivent généralement plus de demandes de modifications qu'elles ne sont capables d'en traiter. Souvent les clients ont du mal à comprendre pourquoi une modification jugée simple nécessite des mois à être incorporée, si elle l'est un jour. Après la livraison du logiciel, le client n'est pas nécessairement satisfait. Au cours des longues années de maintenance cette tendance risque de s'amplifier. Il est difficile de chiffrer les impacts de cette insatisfaction, mais dans un contexte compétitif, la facilité d'évolution d'un logiciel est primordiale.

#### Vers une nouvelle crise?

Les ressources affectées à la maintenance ne sont pas disponibles pour le développement.
Cette constatation semble bien banale. Néanmoins, pour une organisation donnée, l'ensemble des ressources disponibles est généralement fini. Les aspirations de certaines organisations de faible

taille pour le développement de nouvelles applications se voient alors contrariées par le besoin de maintenir les logiciels déjà produits.

Le problème peut se poser également pour des organisations plus importantes.
Selon [MIL-srah94] l'état des systèmes du DoD serait "proche d'une crise nationale". Actuellement seulement 30% du budget est dépensé pour le développement de nouveaux logiciels. 70% sont consommés par la maintenance[1].

L'US Air Force estime que si aucune mesure n'est prise pour changer cette situation, il sera nécessaire d'employer 25% de la population des Etats Unis entre 18 et 25 ans pour maintenir leurs logiciels en l'an 2000! [Your89].

### I.3.4 Importance des efforts

"Voulez-vous assurer la maintenance d'un logiciel ou participer au développement de nouveaux produits?". Assurément si l'on pose cette question à des ingénieurs sortant d'une école d'informatique la réponse sera unanime[2]. D'ailleurs certaines organisations peu conscientes de l'importance de la maintenance évitent justement de poser la question ; une affectation a un poste de maintenance étant parfois vue comme une pénalité [GlasNois81] [Pari86b] [DartCB93][3].
Si l'on pose la question "Quelle est la situation idéale pour la maintenance", il y aura toujours une personne pour répondre "la supprimer!". Hélas ce n'est pas possible.
Une enquête menée dans 35 départements de maintenance révèle que 90% des chefs d'équipe se plaignent du manque d'intérêt du personnel[4].

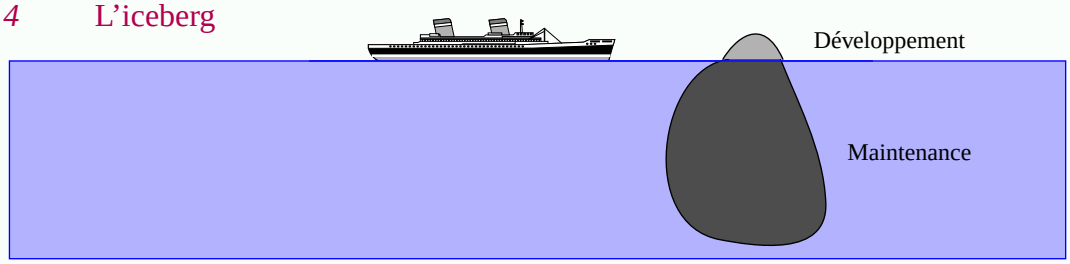
Dans le domaine de la recherche, la maintenance n'est pas non plus un sujet vraiment prisé. D'ailleurs, comme nous l'avons fait remarquer, dans les premiers cycles de vie la maintenance n'apparaissait même pas. Aujourd'hui certains voient même encore maintenance et recherche comme deux termes quelque peu contradictoires puisque l'un est tourné vers le passé et l'autre tourné vers l'avenir. Il est souvent dit "Plutôt que de maintenir des logiciels de mauvaises qualités il serait préférable d'en construire de nouveaux". Cette affirmation est tout à fait sensée dans un contexte de recherche. Hélas elle n'est pas réaliste dans l'industrie si l'on considère des logiciels de grandes tailles. C'est justement le propos du génie logiciel.

Pour combattre ce manque d'intérêt, des images fortes ont souvent été employées pour faire ressortir le problème de la maintenance. Par exemple la maintenance a été comparée à un "iceberg"[5]. Cette image suggère que le coût de développement n'est que la partie visible du problème. La maintenance est méconnue et la sous-estimer c'est s'exposer à certains risques...

---

1. Pour ne pas dire "70% se consument dans la maintenance".
2. Pour savoir quelle serait la réponse le lecteur est invité à se poser la question...
3. Cette dernière référence correspond à un rapport récent du SEI (Software Engineering Institute). Le point est fait sur l'état de la maintenance dans les organisations gouvernementales des Etats Unis. Il s'agit en fait d'une enquête réalisée dans différents départements de maintenance mais comme le suggère ses auteurs, les résultats de cette enquête sont applicables dans la plupart des organisations de grande ou de faible taille. La deuxième partie de ce rapport contient les recommandations du SEI. Ce document sera souvent cité dans la suite de cette thèse.
4. R.K. Boll, juin 1986.
5. EDP Analyzer, "That maintenance 'Iceberg'", Canning Publications, October 1972.

*figure 14* L'iceberg




Le manque de travaux de recherche dans le domaine de la maintenance est souvent souligné [GlasNois81] [Schn87]. Dans [HaleHawo88], il est fait une analyse statistique du nombre d'articles décrivant des recherches expérimentales dans ce domaine entre 1978 et 1987. On retiendra entre autre qu'avant 1982 aucun de ces travaux n'avait été entrepris dans un milieu universitaire.

### I.3.5 Pourquoi la maintenance est elle difficile et coûteuse ?

De nombreux facteurs techniques et non techniques s'ajoutent et rendent la maintenance particulièrement difficile et coûteuse.

#### *Gestion de projets et aspects humains...*

Au cours du développement d'un logiciel, de nombreuses personnes assurent le déroulement correct de chaque phase. Chaque personne tient un rôle particulier correspondant à ces compétences. Par la suite les effectifs affectés à la maintenance sont naturellement réduits car il s'agit d'un travail moins intensif.

Les tâches qui incombent aux chargés de maintenance sont les plus diverses, surtout dans les organisations de faible taille. Pour mener à bien toutes ces tâches le chargé de maintenance doit donc posséder de solides compétences dans différents domaines, surtout en l'absence de réelle planification comme c'est souvent le cas lors de la maintenance. La polyvalence, la patience, la flexibilité, la capacité à se motiver soi-même, la capacité d'organisation, la responsabilité sont des qualités nécessaires à un tel rôle. Le profil du chargé de maintenance s'avoisine à celui de la perle rare (Dans on parle de "mouton a 5 pattes" [Stei95]). Malgré cela, ce sont les personnes les moins qualifiées qui sont souvent affectées aux postes de maintenance [GlasNois81] [DartCB93] ! La maintenance se caractérise souvent par un manque d'intérêt flagrant. Il en résulte évidemment des coûts élevés mais aussi une détérioration de la qualité du logiciel.

En fait une partie des problèmes de la maintenance vient du fait que celle-ci est sous-estimée par les chefs de projets [GlasNois81] [Schn87]. Les équipes de maintenance travaillent souvent dans de mauvaises conditions et le personnel se sent lésé par rapport aux équipes de développement [DartCB93].

La maintenance n'est pas toujours prise au sérieux et les chefs de projets préfèrent parfois des solutions bâclées [Wein83]. Weinberg a déterminé expérimentalement la probabilité d'introduire une erreur en fonction du nombre de lignes de code modifiées. Il apparaît que les modifications

les plus simples sont celles ayant le plus de chances d'introduire des erreurs (tableau 1).

TABLEAU 1 Probabilité d'introduire une erreur par nombre de lignes modifiées [Wein83]

| Nombre de lignes de code modifiées | 1 | 2 | 3 | 4 | 5 | 10 | 20 |
|---|---|---|---|---|---|---|---|
| Probabilité d'introduire une erreur | 50% | 60% | 65% | 70% | 75% | 50% | 35% |

Cela est dû au fait qu'elles semblent si simples qu'aucune attention ne leur est consacrée. Ce phénomène est rapporté par différents auteurs [Buss&al94]. L'application de tous les tests de qualité est souvent omise après de telles modifications. Les 3 erreurs de programmation les plus coûteuses (respectivement $1.600.000.000, $900.000.000 et $245.000.000) sont dues à la modification d'une seule ligne de code! [Wein83]. De quoi faire réfléchir...

#### *Facteur temps : de courts délais...*

Le temps est une contrainte forte lors du développement de logiciels. La pression correspondant à des délais de livraisons courts font que les logiciels passent souvent en phase de maintenance avant qu'ils ne soient réellement achevés. Tout au long de la maintenance, on observe aussi une telle pression ; avec des délais parfois plus courts encore [DartCB93].
Ce peut être le cas lors du portage de logiciels. S'il existe sur le marché des logiciels concurrents, être le premier à proposer des implémentations pour une nouvelle plate-forme peut être important pour un producteur de logiciels [TilbCroo92].
Rappelons également que le système est en service lors de la maintenance. Des solutions immédiates doivent être apportées si l'on détecte certains dysfonctionnements. C'est le cas par exemple pour les systèmes embarqués. Ce peut être également le cas pour des applications plus "classiques". Par exemple, la détection d'une erreur dans un système de cartes bancaires peut impliquer des corrections urgentes.
Bien évidemment cette pression au niveau du temps rend la maintenance difficile. D'une part il faut élaborer des solutions en peu de temps, d'autre part les solutions apportées tendent à déstructurer le logiciel.

#### *Facteur temps : de longues périodes...*

Le facteur temps joue aussi un rôle inverse. Les durées de vie s'évaluent en années, voire en décennies. La longueur de ces périodes n'est pas sans conséquence.

La mobilité importante du personnel en informatique fait que tout au long de la durée de vie d'un logiciel un grand nombre de personnes différentes interviennent.
Au fur et à mesure que le temps passe, le nombre de personnes "connaissant" le logiciel diminue. Au bout de quelques années il est courant qu'aucune personne n'ayant participé au développement ne soit présente. La mauvaise réputation de la maintenance accentue aussi ce phénomène car une personne affectée à un poste de maintenance a tendance à en changer le plus tôt possible [DartCB93]. Le problème est qu'avec le départ de ces personnes "s'envole" aussi une masse importante de connaissances. Souvent seul le code source reste [Arno93].
La diversité des styles et des méthodes employés par chaque intervenant constitue également un problème important. Il en résulte un produit logiciel peu homogène et difficile à comprendre.

De longues durées de vies impliquent aussi de très nombreuses modifications. Chaque année différentes plates-formes matérielles et services logiciels apparaissent, d'autres disparaissent. Les besoins en terme d'application informatique changent également à cause de l'évolution de la société.

Notons aussi le caractère incrémental des modifications apportées lors de la maintenance.
Au début du développement, l'élaboration du cahier des charges a pour but de fixer un ensemble de besoins. Il est alors possible de construire un système de qualité répondant à ces besoins.
Par contre, au cours de la maintenance ce n'est que peu à peu que les demandes de modifications sont faites. En l'absence d'une vision globale de ce que va être l'évolution du logiciel il est difficile de proposer directement des solutions générales. Souvent chaque modification est implémentée pour résoudre un problème spécifique et ce n'est qu'avec le temps et l'expérience que l'on découvre *a posteriori* comment apporter des solutions générales [SchwStra93].

Finalement, pour conclure avec les conséquences néfastes du facteur temps, notons qu'au cours des années les techniques informatiques utilisées changent. Il en résulte des produits logiciels hétérogènes [AikeMR94], où l'on peut par exemple trouver simultanément des parties en assembleurs, en cobol ou utilisant des bases de données relationnelles. Une fois encore, le chargé de maintenance est supposé avoir des compétences polyvalentes...
Remarquons que la plate-forme sur lequel le logiciel est maintenu peut, elle aussi, être obsolète. Des ressources matérielles de mauvaise qualité ne simplifient évidemment pas les activités de maintenance. De plus les équipes de maintenance doivent aussi se contenter des outils périmés fonctionnant sur une plate-forme âgée. Face aux équipes de développement se dotant plus souvent d'outils correspondant à l'état de l'art, les chargés de maintenance se sentent souvent frustrés et abandonnés [DartCB93].

#### *Des logiciels complexes...*

La complexité des logiciels à maintenir est évidemment la raison principale expliquant les difficultés de la maintenance car bien entendu pour modifier un logiciel il est tout d'abord nécessaire de bien le comprendre. Obtenir une vision globale d'un logiciel complexe est une tâche très difficile. Comprendre le fonctionnement d'un composant peut l'être aussi, tout comme comprendre quelles sont les interactions entre les différents composants.

#### *Des logiciels de mauvaise qualité...*

Actuellement et depuis longtemps, les logiciels que l'on doit maintenir sont, non seulement complexes mais en plus de mauvaise qualité ; tout au moins en ce qui concerne la facilité de maintenance.
En fait la plupart des logiciels existant aujourd'hui ont été conçus sans prendre en compte la facilité de maintenance. Dans les premières décennies de l'informatique, qualité du logiciel et efficacité étaient d'ailleurs synonymes. Les ressources matérielles étaient chères et les logiciels étaient conçus pour en optimiser l'utilisation. Maintenant au contraire c'est l'utilisation des ressources humaines que l'on cherche à optimiser.
L'un des problèmes primordiaux et que la documentation des systèmes en phase de maintenance n'est pas à jour. D'ailleurs les statistiques montrent que les équipes de maintenance se plaignent avant tout de l'absence ou de la mauvaise qualité de la documentation [Schn87] [DartCB93].

## I.3.6 Approches

Après avoir fait ressortir les problèmes rencontrés au cours de la maintenance, il est naturel d'étudier les solutions potentielles. Nous distinguons trois grandes classes de solutions selon qu'elles s'attachent au futur, au présent ou au passé.

### I.3.6.1 *Le futur, Prévenir...*

*"Mieux vaux prévenir que guérir..."*

La maintenance doit être une préoccupation dès le développement d'un logiciel. En effet les méthodes et techniques employées au cours de cette phase déterminent sa facilité de maintenance, tout au moins initialement. Cette constatation est à la base de la première classe de solutions. Ici il s'agit de faciliter la maintenance *future* des logiciels. C'est une approche "a priori".

Comme nous l'avons dit, traditionnellement le génie logiciel s'est surtout attaché à améliorer la phase de développement. Lorsque les problèmes liés à la maintenance se sont révélés comme étant critiques, il est paru tout naturel de mettre l'accent sur le développement de logiciels plus facilement maintenables. Il était ainsi possible de mener essentiellement les même travaux qu'auparavant mais en incluant une préoccupation particulière pour la maintenance.
Incontestablement de gros progrès ont été faits. Maintenir un logiciel bien conçu, composé de documents de spécifications et de programmes ADA, est évidemment une tâche plus facile que de maintenir un logiciel écrit en assembleur sans standards ni règles de codification.

Néanmoins cette approche n'est pas suffisante.
Tout d'abord elle n'est valable que pour faciliter la maintenance des logiciels produits dans le futur. Elle ne s'applique donc pas à la masse importante des logiciels à maintenir aujourd'hui.
Notons aussi qu'il n'est pas toujours facile d'évaluer les impacts d'une solution de ce type car ceux-ci ne sont pas immédiats. Ce n'est que quelques années plus tard, lors de la maintenance de logiciels de grandes tailles, qu'il est possible de vérifier expérimentalement qu'une solution est meilleure qu'une autre[1]. C'est aussi après de longues périodes que les problèmes associés apparaissent. Enfin comme nous l'avons souligné, les coûts élevés de la maintenance ne sont pas uniquement liés à la qualité du produit, mais aussi à la manière de les maintenir et à d'autres difficultés pragmatiques.

### I.3.6.2 *Le présent, Simplifier et Préserver...*

Développer des logiciels faciles à maintenir est une chose, conserver cette qualité tout au long de la maintenance en est une autre. C'est justement l'un des buts de cette seconde approche où l'objectif est d'élaborer des méthodes et techniques permettant : (1) d'organiser et de simplifier les activités de maintenance, (2) de limiter la détérioration du logiciel lors de ces activités.

Par exemple, les travaux menés dans le domaine de la gestion de configurations répondent à ces critères (ce thème est présenté dans la Section I.4). C'est le cas aussi des travaux visant à définir et

1. Par exemple il n'existe pas aujourd'hui d'évidence montrant que le paradigme objet simplifie la maintenance des logiciels [CartShep95]

contrôler le processus de maintenance. La gestion des demandes de modifications est un exemple typique.

Le premier avantage de cette approche est que ses effets sont immédiats. On peut donc attendre des bénéfices directs. Il est envisageable de comparer différentes solutions alternatives. L'autre avantage est qu'elle est applicable aux logiciels actuellement en phase de maintenance.

#### I.3.6.3 *Le passé, Guérir...*

*"Mieux vaux prévenir que guérir, certes, mais mieux vaux guérir que laisser périr..."*

La troisième approche consiste à améliorer a posteriori la facilité de maintenance d'un logiciel déjà existant. C'est un sous-ensemble de la ré-ingénierie, thème présenté plus loin dans ce chapitre (Section I.5). Citons à titre d'exemple, la conversion de programmes FORTRAN en programmes ADA, où la redocumentation d'un système écrit en assembleur.

Cette approche, tournée vers le passé, s'adresse essentiellement à des logiciels âgés et de mauvaise qualité, ceux justement dont la maintenance est difficile et coûteuse...

### I.3.7 Conclusion

**La maintenance reste une zone sombre du génie logiciel**. Elle se caractérise autant par la présence de nombreux problèmes que par l'absence d'intérêt de ceux qui la pratique. C'est aussi un point de divergence entre recherche et industrie : alors que la recherche est résolument tournée vers l'avenir et est capable d'évoluer très rapidement, l'industrie est sans cesse confrontée à la maintenance des acquis développés dans le passé.

Pour faire face aux problèmes de maintenance le SEI recommande aux organisations de se concentrer sur les thèmes suivants triés par ordre d'importance [DartCB93] :

- *(1) la retro-ingénierie,*
- *(2) la ré-ingénierie,*
- (3) les tests,
- *(4) la gestion de configurations,*
- (5) les outils de conception,
- (6) les outils de documentation,
- (7) les outils d'intégration,
- (8) les métriques.

Dans cette thèse nous nous concentrons sur l'intersection entre la *gestion de configurations* (4) (plus généralement la programmation globale étudiée dans la Section I.4), *la rétro-ingénierie* et *la ré-ingénierie* (1) et (2) (thèmes étudiés dans la Section I.5).

# I.4 Programmation globale

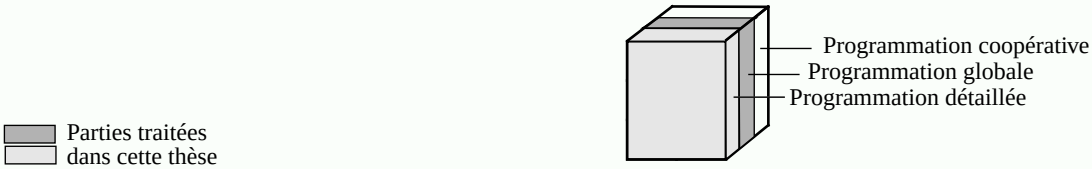
*“Structuring a large collection of modules to form a ‘system’ is an essentially different intellectual activity from that constructing the individual modules. That is, we must distinguish programming-in-the-large from programming-in-the-small” (D.DeRemer, H.Kron)*

*“Einstein argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity, forced without rhyme or reason by the many human institutions and systems to which his interfaces must conform. These differ from interface to interface, and from time to time, not because of necessity but only because they were designed by different people, rather than by God.” (J.F. Brooks)*

## I.4.1 Introduction

Pendant longtemps les recherches en informatique se sont focalisées sur l’élaboration de structures de données et d’algorithmes. Il est apparu par la suite que la réalisation de logiciels de grande taille impliquait bien d’autres activités et concepts. Ceci s’est traduit par une distinction entre ce que l’on appelle la *programmation détaillée* et la *programmation globale*. Par la suite le terme *programmation coopérative* est également apparu.

*figure 15* Programmation détaillée ∪ Progr. globale ∪ Progr. coopérative




Soulignons tout de suite que cette taxonomie n’est qu’une vision de l’esprit. Chaque classe se caractérise par quelques notions, mais n’est pas en soi un thème de recherche. En fait cette taxonomie permet de structurer un espace où chaque thème peut être situé en fonction des notions qu’il considère. Cet espace sera appelé ***espace logique*** (par opposition à l’espace technologique présenté dans le Chapitre III et qui lui contient des solutions concrètes : outils, techniques, etc.).

Nous décrivons plus particulièrement la programmation globale puisque les notions correspondantes sont fondamentales dans le cadre de cette thèse. Nous nous intéressons aussi aux interactions qu’elle a avec la programmation détaillée.

Cette section aborde également un thème de recherche particulier : la *gestion de configurations*. Ce thème trouve sa place dans cette thèse vu sa forte connexion avec la maintenance. Elle trouve sa place dans cette section de part le fait qu’elle considère essentiellement des notions de programmation globale.

## I.4.2 Terminologie : Programmation détaillée, globale et coopérative

Les trois termes *programmation détaillée*, *programmation globale* et *programmation coopérative* sont tout d'abord présentés. La notion de programmation globale est ensuite affinée et l'on introduit l'*architecture*, la *manufacture*, la *variation* et l'*évolution*.

### I.4.2.1 *"Programmation détaillée"*

La ***programmation détaillée*** regroupe les activités se focalisant sur chaque composant du logiciel pris individuellement. Deux notions sont essentielles : les algorithmes et les structures de données.

La programmation détaillée considère le logiciel à un niveau de granularité fin. Les différentes entités considérées sont les variables, les types, les opérateurs arithmétiques, les instructions, etc. Typiquement les compilateurs et les éditeurs syntaxiques sont des outils de programmation détaillée. En fait, les langages de programmation sont à la base de cette discipline.

### I.4.2.2 *"Programmation globale"*

La signification du terme "Programmation globale" a évolué au cours du temps [Tich92]. Tout d'abord nous présentons son origine, ensuite son évolution et enfin notre définition.

#### *L'origine du terme...*

Le volume et la complexité des logiciels sont des facteurs importants qui influent sur la manière de les développer et de les maintenir, non seulement d'un point de vue quantitatif mais également d'un point de vue qualitatif [Shaw86]. Le développement de logiciels de grandes tailles, implique bien sûr des activités de programmation détaillée, mais aussi des activités de natures différentes.

Cette constatation est à la base de la distinction entre programmation globale et programmation détaillée, idée introduite en 1976 par DeRemer et Kron dans l'article "Programming-in-the-large vs. Programming-in-the-small" [DereKron76]. L'objectif de DeRemer et de Kron était de montrer que les langages de programmation n'étaient pas aptes à résoudre seuls les problèmes posés par le développement de logiciels de grandes tailles. Ils définirent un ***Langage de Connexions de Modules*** (un ***M.I.L.***[1]) ayant pour but de décrire les interactions entre les composantes du logiciel à un niveau global. La comparaison de ce langage aux langages de programmation explique la présence du mot "programmation" dans le terme programmation globale.

#### *L'évolution du terme...*

Par la suite, il est apparu beaucoup plus clairement que la programmation ne représentait qu'une partie des activités en génie logiciel. Le concept de programmation globale, initialement représenté par l'utilisation de M.I.L, s'est progressivement élargi à d'autres activités du cycle de vie. Quoique devenu contestable, le terme programmation globale est néanmoins resté[2].

---

1. "Module Interconnection Language" en anglais. L'acronyme M.I.L. étant largement utilisé, c'est celui-ci que nous utiliserons dans la suite de ce document.
2. Pour contourner ce problème de terminologie le terme "Developement-in-the-large" a été utilisé [LiRama85]. Néanmoins cet usage ne s'est pas répandu. D'ailleurs il est trop restrictif car il exclut la maintenance.

En fait, bien qu'introduit en 1976, ce terme a surtout été en vogue vers le milieu des années 80, période pendant laquelle il a été utilisé par de nombreux auteurs, par exemple [Wegn84] [Mull86] [RamaGP86] [Shaw86] [WattWF87] [Belk88][1].

Ce terme n'en est pas moins flou. En réalité il n'existe pas de consensus sur ce qu'il recouvre exactement. Bien souvent il est uniquement défini par opposition à la programmation détaillée et est donc susceptible de designer de nombreuses notions. Par exemple dans l'article "Programming-in-the-large" [RamaGP86] cette désignation inclue des concepts allant de la réutilisation jusqu'à la mesure, en passant par des systèmes de bases de connaissances. Selon [Tich92] ce terme recouvre principalement (1) *la gestion de configurations*, (2) *la réutilisation* et (3) *la modélisation de processus*.

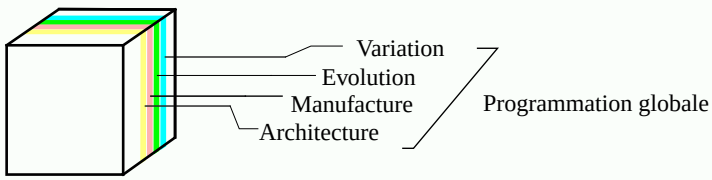
***Notre définition...***

Dans le cadre de cette thèse nous donnons une définition plus restrictive et plus précise. Cette définition sera utilisée tout au long de ce document.

La ***programmation globale*** regroupe les activités se focalisant sur les *produits* logiciels de grande taille en considérant leur ***architecture***, leur ***manufacture***, leur ***évolution*** et/ou ***variation***.

Les quatre termes "*Architecture*", "*Manufacture*", "*Evolution*" et "*Variation*" permettent de raffiner le domaine de la programmation globale.

*figure 16* Programmation globale = Architecture ∪ Manufacture ∪ Evolution ∪ Variation




- ***Architecture*** (ou ***structuration***[2]). Ici les entités de bases sont les modules, les interfaces, les sous-systèmes, etc. En ce qui concerne les relations d'intérêt citons par exemple la relation de dépendance entre modules ou encore la relation de composition entre systèmes et sous systèmes[3].
- ***Manufacture*** (ou ***dérivation***[4]). L'une des caractéristiques du logiciel est qu'il se compose d'objets dont le traitement peut être automatisé. Par exemple le texte source des programmes

1. Dans [CurtKSI87] on trouve même le terme "Programming-in-the-gargantuan". En 1985 un workshop international s'intitulait "Software Engineering Environment for Programming-in-the-large" [Chea85]. Notons aussi en 1986, la décomposition des actes du workshop [ConrDW86] en deux parties : "Programming-in-the-small" et "Programming-in-the-large".
2. Le terme "structuration" étant employé à de nombreuses occasions, comme certains auteurs, nous préférerons le terme "architecture".
3. Depuis quelques années le terme "architecture" est employé dans un contexte plus large que celui utilisé ici. Il intègre désormais les problèmes de communication entre outils et est souvent associé au problème d'intégration (on parle par exemple d'architecture dans le cadre des environnements de génie logiciel). Cette vision n'est pas contradictoire avec la vision plus ancienne (celle utilisée dans cette thèse). La différence réside dans le fait que les communications entre composants étaient essentiellement réalisées par des appels de procédures et des échanges de structure de données alors qu'aujourd'hui il s'agit de techniques réseaux plus diversifiées et complexes. Dans le cadre de la maintenance et de la ré-ingénierie il est suffisant de considérer la définition la plus restrictive.

peut être automatiquement transformé, ou dérivé, en un code objet à l'aide d'un compilateur. On est alors amené à distinguer les ***objets sources*** des ***objets dérivés***. La production des premiers nécessite une intervention humaine alors que les seconds peuvent être dérivés automatiquement à partir d'autres objets via l'application d'***outils de dérivation***. Pour un logiciel, on appelle ***manufacture*** la génération des objets dérivés à partir des objets sources.

- ***Evolution***. L'évolution des logiciels est un phénomène incontournable et intrinsèquement lié au temps. Dans le cadre de la programmation globale *seuls* les résultats de l'évolution sur le *produit* logiciel sont considérés, pas le *processus* d'évolution. En fait, l'évolution des logiciels est à la fois un problème de programmation globale et de programmation coopérative (voir la section suivante). Les versions, ou plus exactement les révisions successives, sont des exemples d'entités.
- ***Variation***. Dans cette thèse nous faisons une distinction entre les problèmes d'évolution et les problèmes de variations. Dans le dernier cas on pourrait également parler de problèmes d'adaptation. Pour un logiciel donné, il s'agit de répondre aux variations en termes de besoins des utilisateurs et des plates-formes. Quand une version unique et générale du logiciel n'est pas suffisante, et c'est souvent le cas, il est nécessaire de décrire et de maintenir plusieurs variantes[1]. Trop souvent ce problème est assimilé aux problèmes d'évolution. Pourtant conceptuellement il s'agit d'une autre dimension où le temps n'intervient pas. Par exemple dès la conception initiale du logiciel il peut être utile de proposer différentes variantes de celui-ci[2].

#### Discussion

Selon notre définition, la programmation globale se focalise sur le *produit logiciel* et ne prend pas en compte le *processus logiciel*. Ceci signifie par exemple que les problèmes de coordination et de coopération entre différents développeurs ne sont pas considérés.
Cette définition fait également ressortir le fait que l'on s'intéresse aux produits logiciels de *grande taille*. Dans ces conditions l'architecture, l'évolution, la variation et la manufacture sont des éléments essentiels et il est indispensable de les contrôler.

#### Granularité forte et granularité fine...

Il est souvent considéré que la distinction entre programmation globale et programmation détaillée implique un saut de granularité. Alors que la programmation détaillée considère des entités de faible grain (par exemple des variables et des instructions), la programmation globale fait abstraction de nombreux "détails" et considèrent des grains plus forts (des modules, des interfaces, etc.). C'est d'ailleurs cette idée qui a été mise en avant dans la traduction française puisque "globale" est opposée à "détaillée"[3].

---

4. Dans le domaine de la conception assistée par ordinateur et parfois même en génie logiciel, on dit qu'une version est *dérivée* d'une autre si celle-ci a été obtenue (manuellement) à partir d'une copie de cette dernière. Ici la *dérivation* est supposée être automatique et n'est pas liée à la notion de version. On préférera donc le terme manufacture, car plus spécifique et donnant lieu à moins de confusion.

1. Les termes "versions", "variantes", "variation" sont définis plus loin (cf Chapitre II).

2. Même si l'évolution et la variation sont des concepts différents, les techniques proposées sont souvent proches ou même identiques. Par exemple les techniques de versionnement sont généralement communes. Ceci explique l'amalgame souvent fait entre évolution et variation.
F.P. Brooks fait également la distinction entre ces deux notions qui sont appelées "conformity" et "changeability" [Broo87].

### I.4.2.3 *"Programmation coopérative"*

Avec la définition de Tichy, la programmation globale inclut les aspects processus logiciels [Tich92]. Au contraire dans cette thèse ces aspects sont regroupés sous le terme "Programmation coopérative".

#### *L'évolution du terme...*

C'est dans le début des années 80 que ce raffinement a été introduit dans le projet Gandalf sous le terme "Programming-in-the-many"[1]. Au cours de cette décennie peu d'auteurs ont repris cette distinction [Mull86]. Par contre au cours des dernières années la croissance de la gestion de processus a provoqué un regain d'intérêt pour celle-ci [Melo93] [Ahme94].

#### *Une définition...*

Deux décennies après l'introduction du terme "*programmation globale*" il n'existe pas de consensus sur ce qu'il recouvre. Le terme "*programmation coopérative*" est encore plus récent et il n'est donc pas étonnant que les limites du domaine qu'il désigne ne soient pas claires. Auparavant la programmation globale était utilisée pour désigner tout ce qui n'était pas du ressort de la programmation détaillée. Maintenant cette même logique s'applique à la programmation coopérative. Ce terme est introduit dans cette thèse uniquement pour situer la programmation globale et la programmation détaillée dans un contexte général. La définition suivante, même si elle n'indique pas de bornes précises, est néanmoins suffisante :

La ***programmation coopérative*** est l'ensemble des activités liées à la multiplicité des agents impliqués dans un projet logiciel [Melo93].

Clairement la présence de différents agents coopérant pour produire un logiciel de grande taille introduit de nouvelles notions. Par exemple la notion de rôle est importante, tout comme la notion d'espace de travail. Le contrôle des activités est également essentiel.

Remarquons aussi que la différence entre la programmation coopérative et la programmation globale n'est pas toujours très nette. Rappelons simplement que la programmation globale se focalise sur le produit logiciel alors que la programmation coopérative prend en compte les aspects processus.

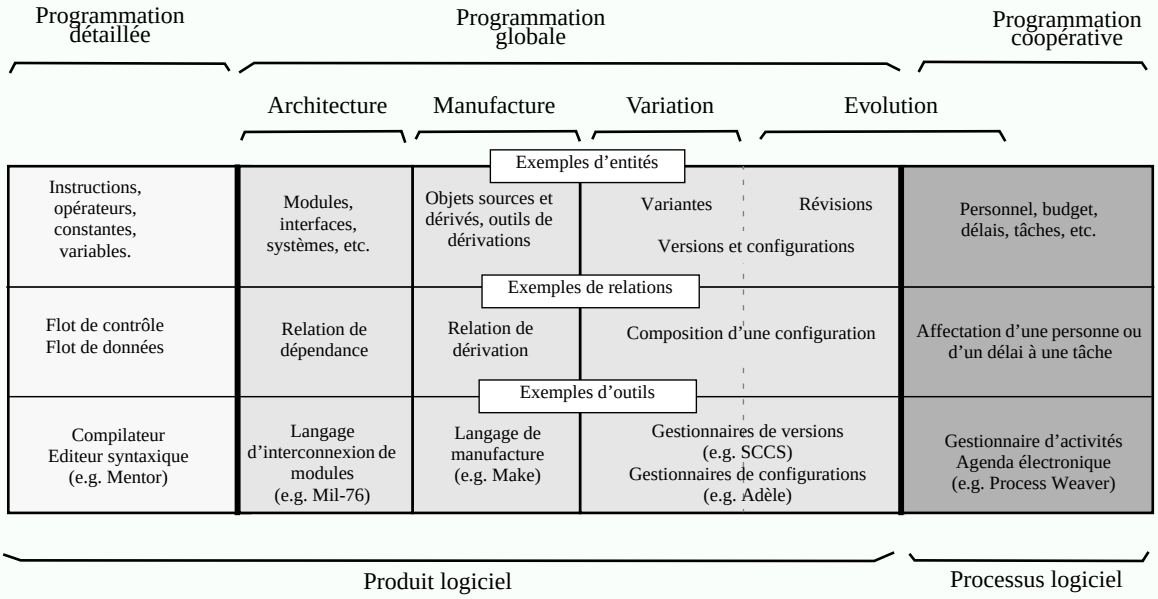
---

3. Cette idée est également reprise dans différents domaines et les suffixes "-in-the-large" et "-in-the-small" ont été utilisés à d'autres occasions. On trouve par exemple la distinction entre "Reuse-in-the-large" et "Reuse-in-the-small" [Luba86], [Mull86], "Documenting-in-the-small" et "Documenting-in-the-large" [Till93]. Nous avons introduit aussi la distinction entre "Reengineering-in-the-large" et "Reengineering-in-the-small" [Favr94a] que nous traduisons par ré-ingénierie détaillée et ré-ingénierie globale.

1. "Gandalf system development environments are characterized by (1) System version control support that aids in describing and manipulating the interfaces, composition, and dependencies of modules, subsystems, and systems. Such support is related to the notion of Programming-in-the-large. (2) Project management support that helps control the development process so that programmers make changes in an orderly fashion. Since this is essential to the cooperation, communication, and coordination of programmers in a project, we have christened this Programming-in-the-many" [HabeNotk86].

La figure 17 présente, dans une vision simplifiée, quelques caractéristiques de chaque domaine.

*figure 17* Programmation détaillée, globale et coopérative




## I.4.3 Terminologie : Gestion de versions et gestion de configurations

> *"There is difficulty associated with extracting concepts from Configuration Management systems since there is no commonality in terminology concerning CM functionality throughout the software engineering community and many CM systems implement variations on concepts. ... No single CM system provides all the functionality required by different kinds of users of CM systems". (S. Dart)*

Dans cette section nous présentons la terminologie associée à deux thèmes liés à la programmation globale : la *gestion de versions et la gestion de configurations.* Tout d'abord il est constaté que le terme "*version*" est aussi flou qu'utilisé. Une taxonomie faisant la distinction entre "*versions historiques*", "*versions coopératives*" et "*versions logiques*" est ensuite présentée. Cette terminologie se termine par les termes associés à la gestion de configurations.

### I.4.3.1 *Version et groupe de versions*

Les approches aux problèmes d'évolution et de variation se basent généralement sur la notion de *version*. L'évolution et les variations étant des phénomènes généraux il n'est pas étonnant de constater que dans différents domaines, des travaux ont été entrepris pour tenter de résoudre les problèmes de versionnement. C'est bien évidemment le cas du génie logiciel, mais aussi de la conception assistée par ordinateur, des bases de données et plus particulièrement des bases de données temporelles.

Pendant longtemps les recherches menées dans ces différents secteurs se sont effectuées en parallèle, sans échanges significatifs (Voir la Section I.2.5.4 relative au modèle hélicoïdal). Bien souvent la notion de version et les notions associées étaient définies en termes de techniques d'implémentation et non pas conceptuellement. Bien que cette situation n'ait pas radicalement

changée, depuis quelques années on assiste à des tentatives de rapprochement et d'unification [Katz90][Scio94].

En fait la *notion de version est intuitive mais floue*. Alors que tout le monde s'accorde sur ce qu'est plus ou moins une version, il n'existe pas de consensus sur ce dont il s'agit précisément [Katz90][Dart91][Mahl94]. Nous nous contenterons ici d'une description informelle et générale[1].

La notion de version n'est pas intrinsèque : on parle non pas d'une "version", mais plutôt d'une "version de quelque chose". Pour désigner ces deux concepts on utilisera respectivement les termes "***version***" et "***groupe de versions***"[2]. Dans un sens aussi large que vague, différentes versions sont regroupées dans un groupe de versions pour pouvoir être manipulées comme un tout.

### I.4.3.2 *Versions historiques, versions logiques et versions coopératives.*

L'utilisation d'un terme si général n'est pas toujours approprié et au cours du temps différentes taxonomies ont été proposées. Initialement celles-ci étaient souvent liées à des problèmes d'implémentation propres à chaque système. Peu à peu les notions de *révision* et de *variante* se sont dégagées [Wink86a].
Ici nous utilisons la taxonomie proposée récemment dans le cadre du système ADELE [EstuCasa94] [EstuCasa95]. Elle présente l'avantage d'être basée sur des aspects fonctionnels et non sur des aspects techniques. Trois classes sont distinguées : les *versions historiques*, les *versions logiques*, et les *versions coopératives*.

- ***Versions historiques*** (ou ***révisions***). L'existence de *versions historiques* est liée à l'*évolution* du logiciel dans le temps et au fait que l'on désire garder la *trace* de cette évolution. Si l'on était capable de produire directement un objet donné il n'y aurait pas de versions historiques. Conserver de telles versions est utile pour pouvoir revenir à un état antérieur du produit. La trace est également intéressante dans le cadre de la gestion de projets pour mesurer l'état d'avancement d'un projet donné. Le temps joue un rôle fondamental et l'évolution du logiciel est réellement le problème central. Cette notion correspond plus ou moins au concept de *révision*. Cette terminologie étant largement utilisée elle sera employée dans la suite de ce document de manière équivalente.
  Un groupe de révisions est construit incrémentallement. L'opération la plus importante est l'ajout d'une nouvelle révision.

- ***Versions coopératives***. L'existence de *versions coopératives* est due à l'évolution du logiciel dans un cadre coopératif. Sans activités parallèles il n'y aurait pas de versions coopératives. En fait celles-ci permettent le travail en parallèle et le déroulement d'activités de manière isolées, généralement suivie d'une étape d'intégration. De telles versions ont la caractéristique d'être créées puis détruites dynamiquement[3] en fonction du processus logiciel. Elles sont

1. Le lecteur trouvera une description plus précise des concepts relatifs au versionnement dans le Chapitre II. En fait la terminologie est souvent dirigée par la technologie sous-jacente. Rappelons que le but de ce chapitre n'est pas de présenter la technologie mais plutôt les problèmes.
2. La terminologie employée pour désigner un tel ensemble varie selon la discipline, les auteurs, les systèmes et quelques différences sémantiques ou techniques. On trouve par exemple "groupe de versions" [Tich88], "objet versionné", "objet générique" [Scio94], ou encore "famille" [Estu85] [Wink87].

temporaires et locales à une activité donnée. Cette notion est voisine de la notion de "*variante temporaire*" [Whit91].
Un groupe de versions coopératives évolue dynamiquement. L'opération la plus importante est la fusion de deux versions.

- ***Versions logiques***.
  L'existence de *versions logiques* est liée aux problèmes de *variations* et correspond plus ou moins au concept de *variantes*[1].
  Soulignons de nouveau que l'existence de versions logiques *n'est pas* liée à l'évolution du logiciel, mais plutôt à des aspects fonctionnels ou logiques (d'où le nom). Il est vrai que souvent c'est au cours de la maintenance adaptative que l'on découvre le besoin de différentes variantes. Cependant la caractéristique essentielle d'une version logique est sa fonctionnalité et non pas la manière et le moment auquel elle a été créée.
  L'opération la plus importante est la sélection de version.

#### *Discussion*

Les trois classes présentées ci-dessus correspondent chacune à trois objectifs différents :

- conserver une trace de l'histoire du logiciel (versions historiques),
- rendre possible le travail coopératif (versions coopératives),
- permettre l'utilisation d'un logiciel dans différents contextes (versions logiques).

Alors que la dernière classe correspond typiquement aux problèmes de *variation*, les deux premières correspondent à des problèmes d'*évolution*.

Les *versions logiques* ne sont pas liées à l'aspect coopératif et à l'évolution, il s'agit d'une notion de programmation globale.
Au contraire bien que les *versions coopératives* aient une influence sur le produit logiciel, celle-ci n'est que temporaire. Vu leur rapport étroit avec le processus logiciel, on aura naturellement tendance à classer les versions coopératives comme étant une notion de programmation coopérative.
La notion de *versions historiques* est à cheval sur les deux domaines. Leur impact sur le produit logiciel (par exemple la trace de l'évolution) est un concept de programmation globale alors que le processus de création de telles versions est un concept de programmation coopérative.
Ces différentes considérations sont reflétées dans la figure 18[2].

### I.4.3.3 *Variante*

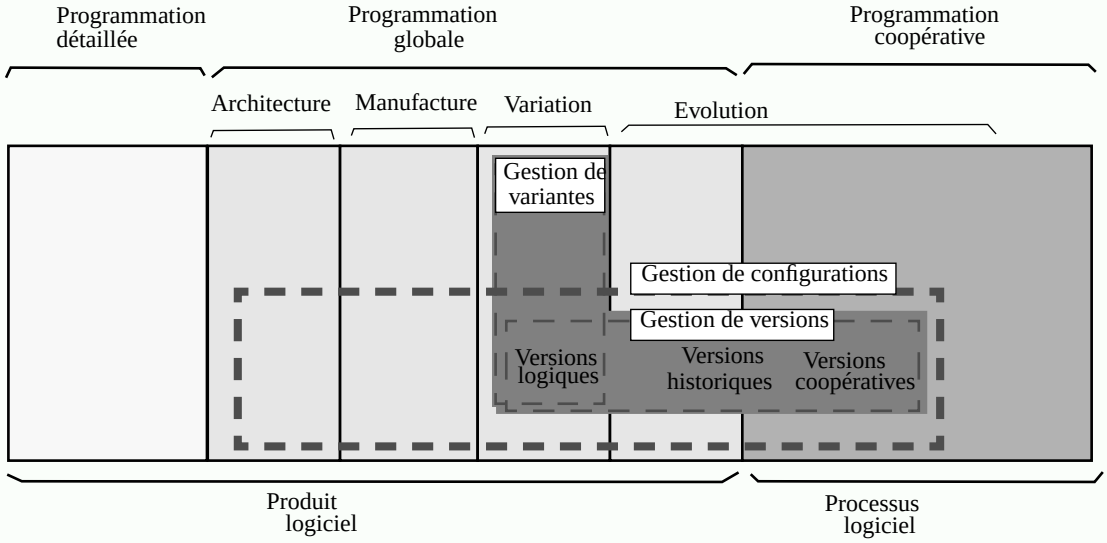
Dans le cadre de cette thèse le terme "variante" est utilisé de manière large. Il recouvre les moyens mis en oeuvre pour résoudre les problèmes de variations. Comme le suggère la figure 18 les variantes ne relèvent pas uniquement des problèmes de gestions de versions et de gestion de

---

3. Dans ce document nous utilisons le terme "version coopérative" mais en fait dans [EstuCasa94] c'est le terme "version dynamique" qui est utilisé. Notre choix s'explique par la cohérence que l'on obtient avec le terme "programmation coopérative", mais aussi par le fait que l'adjectif "dynamique" pourrait en un certain sens s'appliquer aux versions historiques.

1. Le terme alternative ou alternative fonctionnelle est parfois utilisé.
2. Dans ce schéma sont représentées aussi la gestion de configurations et de variantes. Ces termes sont expliqués par la suite.

*figure 18* Versions logiques, versions historiques et versions coopératives
Evolution et variation
Gestion de versions, gestion de variantes, et gestion de configurations




configurations. Au contraire ce concept est également présent dans le cadre de la réutilisation. Les versions logiques forment un sous-ensemble des variantes mais nous ne donnons pas de critère clair délimitant parfaitement ce sous-ensemble. Notons simplement que certaines techniques sont utilisées pour résoudre des problèmes de variations mais ne sont généralement pas prises en compte dans le cadre de la gestion de versions. C'est le cas par exemple de la généricité. Ces différents aspects sont étudiés dans le Chapitre II et le Chapitre III.

### I.4.3.4 *Gestion de configurations logicielles*

La gestion de configurations *logicielles* est une adaptation d'une discipline déjà établie : la gestion de configurations (matérielles), initialement développée dans les années 50 dans l'industrie de l'aviation et de l'espace [Tich92]. Le but de la *gestion de configurations* est de contrôler les changements dans des systèmes complexes et de grandes tailles[1]. Il s'agit d'éviter le chaos résultant de nombreuses corrections, adaptations et extensions apportées à un système tout au long de son existence. La gestion de configurations doit assurer un processus de modification systématique et dont on puisse garder la trace. Il est important de pouvoir connaître à chaque instant l'état du système pour pouvoir le comparer à ses spécifications.

Dans cette thèse nous considérons la ***gestion de configurations logicielles***, ou plus exactement la gestion de configuration de systèmes logiciels. Par souci de simplification, nous utiliserons dans ce document le terme ***gestion de configurations***.
L'un des nouveaux aspects lié à la nature même des logiciels est que les composantes logicielles changent plus rapidement que les composantes matérielles. Un support automatisé est donc nécessaire. Les objets logiciels étant électroniques et virtuels, leur manipulation se fait nécessairement via des outils logiciels et il est donc d'autant plus facile de contrôler leur évolution.

1. Dans son sens général, il s'agit donc de systèmes dont les composantes sont matérielles et *éventuellement* logicielles.

Comme le fait remarquer S. Dart la solution à la gestion de configurations est aussi complexe que le problème [Dart92]. Il est difficile de dégager les concepts clés dans ce domaine et chaque système propose des variations sur les concepts [Dart91]. La gestion de configurations n'a pas de bornes précises [Dart92]. Elle contient autant des aspects de programmation globale que de programmation coopérative (figure 18). Dans cette thèse nous allons nous intéresser uniquement à son intersection avec la programmation globale, c'est à dire aux conséquences de la gestion de configurations sur le produit logiciel.

## I.4.4 Importance du problème

> *"It is easy to ignore configuration management, or persuade yourself that you do not really need it, until one day disaster strikes. Like other preventive measures such as virus detection and network security, users are often driven to CM after the fact, by a calamity that might have been avoidable."*
> *(P.Ingram, C.Burrows, I.Wesley)*

Comme nous l'avons dit la programmation globale regroupe une collection de différentes notions, ce n'est pas un ensemble d'activités spécifiques. Il n'est donc pas possible de chiffrer directement les coûts associés, comme nous l'avons fait par exemple pour la maintenance. Ici au contraire nous essayons de faire ressortir l'influence des concepts sur différentes activités.
Quoi qu'il en soit, il est clair que l'élaboration de logiciel de grandes tailles reste un problème majeur. La dite "crise du logiciel" et les problèmes de la maintenance présentés précédemment en sont la preuve.

### I.4.4.1 *Importance des aspects quantitatifs...*

Revenons sur l'affirmation que nous avons faite dans la Section I.4.2.2 : "le volume et la complexité des logiciels sont des facteurs qui influent sur la manière de les développer et de les maintenir, non seulement d'un point de vue *quantitatif* mais également d'un point de vue *qualitatif*".

L'aspect *qualitatif* correspond à l'apparition de nouveaux concepts et comme nous l'avons vu, c'est l'essence même de la distinction entre la programmation détaillée et la programmation globale.

Reste l'aspect *quantitatif*. Bien souvent cet aspect est négligé dans les travaux de recherche. On a trop souvent tendance à penser que pour passer d'un problème de taille x à un problème de taille $n * x$, un facteur n est suffisant en ce qui concerne les solutions. Tout se passerait bien si les équations étaient effectivement linéaires. Hélas ce n'est généralement pas le cas. Considérons par exemple un logiciel composé de 3 modules, chacun existant en deux versions. Un gestionnaire de configurations ne constitue pas un besoin fondamental. Si au contraire on considère un logiciel de 300 modules, le nombre de configurations possible est de $2^{300}$, un chiffre n'ayant plus aucun sens. On assiste a une explosion combinatoire.

Dans le domaine de la programmation globale les quantités ou les performances peuvent devenir des problèmes critiques. C'est le cas par exemple en ce qui concerne la manufacture des logiciels.

### I.4.4.2 *Importance des problèmes de manufacture*

Tel que nous l'avons défini la manufacture d'un logiciel est la production des objets dérivés à partir des objets sources. Chaque dérivation est automatisée grâce à un outil. Reste néanmoins à assurer leur enchaînement et à fournir les paramètres nécessaires. Pour un logiciel de faible taille cette tâche peut être faite manuellement. Le risque d'oubli et d'erreur augmente dramatiquement avec la taille du logiciel. Pour les logiciels de grandes tailles il est indispensable d'automatiser la manufacture [Feld79].

Automatiser n'est pas suffisant, il faut optimiser. La pause café pendant la compilation est chère aux programmeurs et pour un système très complexe le temps de reconstruction est un problème très sérieux [Kame87]. Plusieurs heures sont nécessaires pour générer un système comme X-window sur une station de travail. Dans le cas de systèmes plus conséquents plusieurs jours sont parfois nécessaires même sur un ordinateur central[1]. Il n'est donc pas pensable de tout regénérer chaque fois qu'un objet source est modifié. Optimiser la manufacture est donc essentiel pour pouvoir faire évoluer les logiciels de grandes tailles.

### I.4.4.3 *Importance de l'architecture et des problèmes associés*

L'architecture est l'aspect le plus important du produit logiciel, en tout cas pour les logiciels de grandes tailles. Une mauvaise architecture accélère la mort du logiciel.

Ces affirmations sont dues au fait que presque toutes les activités de programmation globale et de programmation coopérative sont basées sur l'architecture du produit logiciel. La raison en est simple. Les logiciels de grandes tailles sont trop complexes pour être traités d'un seul bloc. L'application de la devise "Diviser pour mieux régner" recommande alors de décomposer le problème en sous problèmes aussi indépendants que possible. Une telle décomposition doit tirer pleinement profit de la décomposition du produit logiciel, c'est à dire de son architecture.

S'il était possible de décomposer un logiciel de grande taille en différents composants indépendants il n'y aurait pas de problème. Le problème de la production d'un logiciel de grande taille se réduirait à celui de la production de différents logiciels de faibles tailles. Tout réside donc dans la décomposition mais surtout dans les relations existant entre différents composants ; relations souvent regroupées sous le terme générique de "***relation de dépendance***".
Dans son sens le plus général, une relation de dépendance entre deux composants exprime le fait que ceux-ci ne sont pas indépendants ! Cette remarque est triviale mais le rôle de cette relation fondamentale. En effet les activités essayant de se focaliser sur un composant donné se voient néanmoins obligées de prendre en compte :

- *les composants dont il dépend*. Par exemple pour comprendre leur fonctionnalité puisque le composant considéré en dépend.
- *les composants qui dépendent de lui*. Pour contrôler les impacts d'une modification éventuelle.

1. L'une des contraintes imposées par un client lors de l'adoption du gestionnaire de configurations ADELE était que la manufacture de leur logiciel devait se faire en moins de 2 jours (afin de pouvoir être faite le week-end).

D'un point de vue *qualitatif* cette contrainte est importante : des activités se focalisant sur des composants différents doivent être coordonnées si ceux-ci sont reliés par une relation de dépendance. C'est d'ailleurs l'une des bases de la programmation coopérative.

L'aspect *quantitatif* est également un facteur à ne pas sous estimer. Dans des projets de faible taille les programmeurs ajoutent souvent quelques dépendances sans se préoccuper des conséquences. Dans des projets de grandes tailles ces "quelques dépendances" peuvent avoir des répercutions importantes.

En fait comme nous l'avons suggéré l'architecture d'un logiciel a une influence sur un très grand nombre d'activités, autant dans le cadre de la programmation globale que dans celui de la programmation coopérative.

- ***Influence de l'architecture sur la manufacture***. La modification d'un composant a un impact sur tous les objets dérivés des composants qui dépendent de lui (récursivement). Plus les dépendances sont nombreuses entre composants, plus les optimisations sont inefficaces pour réduire le temps de manufacture total. En pratique les programmeurs évitent d'introduire une nouvelle ressource dans un module fortement connecté, quitte à ranger celle-ci dans un autre module ne correspondant pas à sa fonctionnalité. C'est évidemment un facteur de déstructuration (I.3.3). Il est facile de blâmer cette pratique, mais trois heures de compilation après la modification d'une constante dans un module est un réel frein à l'évolution.
- ***Influence de l'architecture sur l'évolution***. La décomposition d'un logiciel peut se faire de différentes façons. Les travaux de Parnas dans les années 70 ont fait prendre conscience du fait que c'est la facilité d'évolution qui devait être pris comme critère [Parn72]. L'idée est d'encapsuler dans chaque composant tous les détails susceptibles d'évoluer pour limiter l'effet d'une modification à ce seul module.
  Remarquons que le nombre de dépendances influe sur le nombre de révisions : si à chaque modification d'un composant il est nécessaire de modifier de nombreux autres composants on assiste à une explosion combinatoire du nombre de révisions.
- ***Influence de l'architecture sur la variation***. Les problèmes d'architecture et de variation sont intimement liés. Sans une bonne structuration, proposer plusieurs variantes peut être quasi impossible. Au contraire, une décomposition adéquate du logiciel permet de localiser les variations et de les gérer beaucoup mieux. Encore une fois la relation de dépendance joue un rôle fondamental. De manière analogue à ce qui se passe pour l'évolution, il y a un risque d'explosion combinatoire du nombre de variantes nécessaires.
- ***Influence de l'architecture sur le travail coopératif***. La décomposition des tâches, des équipes, des responsabilités se fait souvent en fonction de la décomposition du produit logiciel. Plus les composants dépendent les uns des autres, plus les problèmes de coordinations entre équipes seront importants. Il est de plus en plus difficile d'organiser les différentes activités et de faire des planifications car un problème non planifié peut avoir des répercutions sur de nombreuses autres activités.

Même en connaissant parfaitement les interactions entre les différents composants du logiciel, les problèmes mentionnés ci-dessus sont susceptibles d'apparaître. Ils sont liés au fait que les composants ne sont pas indépendants.

Qu'arrive-t-il si les interactions entre les différents composants du logiciel ne sont pas décrites précisément? Le programmeur oublie des contraintes entre composants et implémente incorrectement certaines modifications. Certains objets dérivés ne sont plus mis à jour lors de la manufacture du logiciel. Des erreurs difficiles à détecter sont ainsi introduites. Le logiciel passe à un état incohérent. On découvre des problèmes de coordination non planifiés pendant le déroulement des activités. Sans vision globale de l'architecture du logiciel, celui-ci tend à se dégrader.

*Décrire et contrôler l'architecture du logiciel est donc un problème fondamental*[1]. Non seulement parce que c'est important en soi, mais aussi parce que la structuration du logiciel est la base sur laquelle reposent de nombreuses activités.

#### I.4.4.4 *Importance des variantes*

Pour amortir les coûts de développement d'un logiciel, beaucoup d'entreprises sont amenées à proposer un grand nombre de variantes pour pouvoir multiplier les sites d'installation [GentMSC89][TilbCroo92].

Dans un environnement compétitif ce phénomène s'accentue car il s'agit de prendre des parts de marché avant les concurrents. La production d'outils ou de plates-formes logicielles n'échappe pas à cette règle. Par exemple, la survie d'un système de fenêtrage ou d'une base de données est liée à sa disponibilité sur un grand nombre de plates-formes et à sa flexibilité. A terme, seuls quelques standards de facto subsistent.

En pratique, pour un certain nombre de petites et moyennes entreprises, les problèmes de variations sont *vitaux* [MahlLamp88][TilbCroo92][Mahl94].
Ce phénomène est moindre pour les grandes entreprises ou les organismes d'état [DartCB93]. Ceux-ci sont moins vulnérables ; la concurrence est moins importante ; le marché des projets de très grande envergure est réduit.

#### I.4.4.5 *Importance des problèmes d'évolution des variantes*

Même si la notion de variantes est indépendante de la notion de temps, les problèmes d'évolution s'ajoutent aux problèmes de variations. Au cours de la maintenance des logiciels il est nécessaire de créer de nouvelles variantes mais aussi de faire évoluer les variantes existantes. Le premier problème est lié aux problèmes de portage ; le deuxième au "*problème de la maintenance multiple*".

***Problèmes de portage***. Au cours de la durée de vie d'un logiciel il est impossible de prévoir dès le début dans quels contextes il sera utilisé. Dans bien des cas, un logiciel initialement prévu pour une plate-forme donnée devra être porté si son succès est important. Les activités de maintenance adaptative devront se dérouler au rythme de l'apparition de nouvelles plates-formes. Afin d'assurer la disponibilité d'un produit avant les concurrents, les délais prévus pour les portages sont particulièrement réduits. De surcroît il n'est pas rare que la plate-forme cible ne soit pas

1. Pour F.P. Brooks il s'agit d'un problème "essentiel", tout comme les problèmes d'évolution et de variations ("essentiel" dans le sens philosophique du terme et par opposition à "accidentel") [Broo87].

disponible sur le site de développement. Il s'agit alors de faire du développement croisé et/ou de détacher temporairement du personnel sur le site distant [Gent89]. En pratique, le portage est peut être l'une des activités de maintenance qui se déroule dans les conditions les plus difficiles. Le chargé de maintenance doit adapter dans les délais les plus courts un produit à un environnement qu'il ne connaît pas.

***Problème de la maintenance multiple***. Même si aucune variante n'est ajoutée au cours du temps, l'évolution du logiciel est nécessaire et les différentes variantes existantes doivent être maintenues. Il s'agit par exemple d'intégrer la correction d'une erreur ou l'ajout d'une fonctionnalité dans *toutes* les variantes concernées. Plusieurs méthodes peuvent être employées en fonction de la représentation des variantes. Considérons ici les deux cas extrêmes :

- Les variantes du logiciel sont représentées par des copies séparées. La modification doit être effectuée successivement sur chaque variante concernée. Sans une automatisation, même partielle, cette duplication des efforts est souvent rédhibitoire. Les problèmes sont accrus dans la mesure où une même modification doit parfois être implémentée de différentes manières selon les variantes.
- La représentation du logiciel est basée sur la description des parties communes et des différences entre variantes. Cette représentation permet d'éviter une duplication des efforts dans le cas où une modification peut être appliquée sur une partie commune. En contrepartie la complexité d'une telle représentation rend la tâche du chargé de maintenance très difficile.

Dans les deux cas, maintenir des variantes est un problème majeur. Il tourne souvent au cauchemar sans méthodes et outils associés [Mahl94]. L'enquête du SEI sur la maintenance révèle d'ailleurs qu'à cause des difficultés liées à la maintenance multiple, beaucoup de grandes entreprises évitent tout simplement de proposer plusieurs variantes, et ce au détriment des utilisateurs [DartCB93]. Alors que ces entreprises peuvent faire l'impasse sur ce problème, ce n'est toujours le cas pour d'autres producteurs du logiciel [Mahl94]

Les problèmes de la maintenance multiple et de la représentation des variantes font l'objet de discussions plus approfondies dans le Chapitre II et le Chapitre III.

#### I.4.4.6 *Importance des problèmes d'évolution et de gestion de configurations*

Vu son rapport avec les problèmes cités ci-dessus, la gestion de configurations est un thème particulièrement important. Ce problème peut même devenir critique et il est parfois l'une des raisons principales de l'échec de projets de grande taille. Par exemple cela aurait été le cas pour l'échec du projet Taurus en Angleterre, projet annulé après avoir dépensé plus de 80 millions de livres [IngrBW93].

Selon le modèle CMM la gestion de configurations est l'une des fonctions nécessaires pour qu'une organisation puisse sortir du stade "d'immaturité" [PaulCCW93]. Notons aussi que parmi les recommandations faites par le SEI aux organisations concernées par la maintenance de grands logiciels, la gestion de configurations vient en quatrième position [DartCB93]. Il est d'ailleurs signalé dans ce rapport que c'est l'une des priorités selon le personnel de ces organisations.

### I.4.5 Importance des efforts

A l'image de la maintenance, la programmation globale a été délaissée pendant bien longtemps. Au cours de la dernière décennie des travaux de recherche de plus en plus nombreux et ambitieux ont vu le jour. Actuellement l'industrie commence à tirer pleinement profit de ces apports. Néanmoins la programmation globale est encore bien loin d'atteindre le niveau de maturité de la programmation détaillée [Tich92].

***Recherche...***

Historiquement l'architecture est le premier thème de la programmation globale à avoir été abordé. Vers la fin des années 60, les premières idées concernant l'architecture des logiciels sont apparues dans le cadre des systèmes d'exploitations avec, entre autre, la notion de systèmes hiérarchiques et de systèmes en couches. Dans les années 70 ces idées ont été affinées avec la notion de masquage d'informations [Parn72] et l'orienté objet. La distinction entre la programmation détaillée et la programmation globale introduite en 1976 [DereKron76] se concentre sur la structure du logiciel. Vers la fin des années 70 différents mécanismes ont été introduits dans les langages de programmation, ce qui a donné par exemple naissance aux langages modulaires. Au cours de cette décade, seuls quelques outils ont été proposés pour la gestion de versions et la manufacture. Par exemple Make [Feld79] et SCCS [Roch74].

Ce n'est que dans les années 80 que la programmation globale à réellement pris son essor. Dans cette décade le contrôle de l'évolution du logiciel est peu à peu devenu un thème de recherche à part entière. Dans la première moitié des années 80, on a assisté à une prolifération de travaux visant à proposer des environnements de programmation combinant éditeurs syntaxiques, compilateurs incrémentaux, outils de mise au point, etc[1]. Mis à part quelques rares exceptions dont le projet Gandalf, les problèmes de programmation globale n'étaient pris en compte.
Convaincus de l'importance de ce thème, certains chercheurs ont tenté de développer des outils supportant ces activités. En 1985 le workshop intitulé "Software Environements for Programming-in-the-large"[2] témoigne de cette réaction. Par la suite c'est sous l'étiquette "gestion de configurations" que se sont organisés 6 workshops[3].

Aujourd'hui, à défaut d'être un thème à la mode, la programmation globale est sortie de l'ombre. Notons que l'absence de logiciels de grandes tailles dans le milieu universitaire contribue dans une très large part à la méconnaissance des problèmes de programmation globale dans ce milieu. Tout comme la maintenance, la programmation globale reste souvent un thème regardé avec suspicion par certains. Les concepts ne sont pas clairs [Dart91] et on est encore bien loin d'être à l'étape "science" du modèle présenté dans la Section I.2.2.1, (p.19).

1. Voir par exemple "Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Developement Environments" Software Engineering Notes, Vol. 9. N. 3, 1984".
2. Proc. of the Workshop on Software Engineering Environments for Programming-in-the-Large, Harwichport (Massachusetts), 1985
3. "International Workshop on Software Version and Configuration Control" à Grassau (Germany) en janvier 1988,
"International Workshop on Software Configuration Management" à Princeton (New Jersey) en octobre 1989 ; à Trondheim (Norway) en juin 1991 ; à Baltimore (Maryland) en mai 1993. ; à Seattle (Washington) en mai 1995.

Quoi qu'il en soit, grâce aux recherches menées dans la dernière décennie l'industrie des grands logiciels peut enfin commencer à pouvoir bénéficier d'un certain soutien.

#### *Industrie...*

Dans le passé quelques organisations conscientes des problèmes de la programmation globale ont confectionné pour un usage interne des systèmes de gestion de configurations. Cette phase correspond à l'étape artisanale du modèle. L'utilisation de ces systèmes n'était pas forcement aisé et leur maintenance est rapidement devenue un problème important [Dart92]. Les autres organisations n'avaient pas une vision très claire des problèmes ou n'étaient pas en mesure d'assumer les coûts de développements internes. Dans le passé la gestion de configurations a donc été un problème peu ou pas traité dans l'industrie.

Aujourd'hui cette situation est en passe de changer. Non seulement de plus en plus d'organisations sont conscientes des problèmes mais aussi depuis quelques années des systèmes commerciaux de gestion de configurations sont disponibles.
Au Etats Unis, le modèle de maturité CMM classant la gestion de configurations au deuxième niveau de maturité [PaulCCW93] à une influence particulière. Il incite toute entreprise susceptible d'avoir des contrats avec le gouvernement américain à automatiser la gestion de configurations. On assiste donc aujourd'hui à un démarrage assez net.
Il ne faudrait cependant pas croire qu'actuellement l'utilisation de tels systèmes soit déjà très répandue. Selon certaines estimations, moins de 10% des programmeurs utilisent un gestionnaire de configurations conséquent [IngrBW93]. Seuls des outils basiques comme SCCS et Make, conçus dans années 70, sont largement utilisés [Dart92][Tich92]. Une enquête du SEI dans les grandes organisations gouvernementales assurant la maintenance de logiciels montre que le personnel n'est pas au courant de l'état de l'art en ce qui concerne les systèmes de gestion de configurations [DartCB93].

Malgré ce retard, il est clair que l'adoption de nouveaux produits est amorcée et que cette tendance s'amplifie [IngrBW93].

#### *Sociétés de services...*

Les producteurs d'outils de génie logiciel sont, depuis le début des années 90, très sensibles à l'évolution du marché des gestionnaires de configurations. En 1991 déjà plus de 40 systèmes étaient disponibles sur le marché [Dart91]. Aujourd'hui de nombreux vendeurs d'outils CASE intègrent des fonctionnalités de gestion de configurations dans leurs produits [Dart92]. En fait le marché est très loin d'être saturé puisque comme nous l'avons vu seul 10% des programmeurs utilisent un gestionnaire de configurations. Les revenus générés par ce marché sont très nettement à la hausse depuis quelques années et devraient atteindre presque 1 milliard de dollars en 1997 .

### I.4.6 Conclusion

Le terme *programmation globale* n'est généralement défini que par opposition à la programmation détaillée, ce qui du coup le rend très flou. Nous proposons une définition plus précise se basant sur l'utilisation de notions plus que de techniques particulières : la *programmation détaillée* est basée sur la notion d'algorithme et de structures de données ; la

*programmation globale* sur les notions d'architecture, de manufacture, de variation et d'évolution ; la *programmation coopérative* sur les notions de rôle et d'activités. Une telle décomposition peut être utile pour situer des disciplines comme la gestion de configurations.

Parmi les 4 volets concernant la programmation globale, l'*architecture* est sans doute le thème le plus important car elle a des répercutions sur tout les autres thèmes. Une mauvaise architecture accélère inévitablement la mort du logiciel. Bien que l'évolution et la variation soient souvent confondues, ce sont des thèmes distincts. L'*évolution* est liée au temps alors qu'il existe de multiples facteurs rendant nécessaire des *variations* du logiciel (il s'agit typiquement de problèmes de plates-formes). La gestion de configurations se concentre essentiellement sur l'aspect évolution et les problèmes de variations sont parfois délaissés. Par contre il s'agit là d'une intersection avec le domaine de la réutilisation.

Bien qu'historiquement la programmation globale ait été définie dans les années 70, ce n'est que dans les années 80 que des efforts substantiels lui ont été dédiés et que des résultats significatifs ont été obtenus. Ce domaine est aujourd'hui en pleine expansion, notamment sous la forme de la gestion de configurations. Des systèmes de plus en plus nombreux voient le jour, même si les concepts manipulés dans le domaine restent flous. Malgré les avancées récentes en terme de technologie, l'état de la pratique reste très largement dominé par l'utilisation de systèmes ad-hoc issus des années 70.

Du point de vue de la technologie, le passé ne peut donc être ignoré. De plus, les logiciels de grande taille sont souvent des logiciels âgés. Ces facteurs poussent naturellement à l'étude de la ré-ingénierie.

# I.5 Ré-ingénierie

*“Reengineering is like looking at a Picasso and trying to come up with a photography of the subject” (V. Merlyn)*

*“Soothsayers and prophets will tell you re-engineering, restructuring, and reverse engineering can turn bad programs into good, cut software maintenance costs by a factor of ten, and improve sex lives of your maintenance programmers. Naysayers and skeptics will tell you it’s all bullshit and there simply is no mechanical way to derive a program’s functional specification from the source code. I say: “it doesn’t matter”. The RE-3 technologies, as, I call them, are less miraculous than claimed by soothsayers, yet more valuable than skeptics will admit.” (E. Yourdon)*

## I.5.1 Introduction

La ré-ingénierie est l’un des derniers chevaux de bataille sorti pour combattre les problèmes de maintenance dans l’industrie. Cette bataille, dont les enjeux sont considérables, s’accompagne parfois d’un caractère peu rationnel. Avant toute chose, la terminologie est présentée. Ensuite l’importance de la ré-ingénierie est soulignée en considérant d’une part l’importance du problème et d’autre part l’importance des efforts menés dans ce domaine.

## I.5.2 Terminologie

*“In the late 1980 the term software restructuring was giving way to reengineering: “We don’t just restructuring your code, we reengineer it!”. This started a trend of new names for software improvement, maintenance, and migration - a trend that continues today. With each new term came an implication of some magical, unique slant on solving the reengineering problem.” (R.S. Arnold)*

Certains disent que dans le domaine de la ré-ingénierie on trouve plus de mots que de solutions. De fait il existe des problèmes importants de terminologie [Ulri90][ChikCros90]. Dans les articles techniques ou dans les brochures commerciales de nouveaux termes sont progressivement apparus[1] , parfois comme arguments de marketing. Le problème de terminologie est réel car ce domaine recouvre des notions et des techniques très variées ; certaines sont validées, d’autre contestées ou jugées utopiques [WeidHH95]. La confusion entre les différents termes et concepts est un frein à l’acceptation et à l’évolution de ce domaine [Ulri90].

Pour tenter de pallier à ce problème, une taxonomie a été publiée en 1990 dans IEEE Software [ChikCros90]. L’usage des termes continue cependant à évoluer [Arno93].

### I.5.2.1 *Ré-ingénierie*

Lorsque l’on essaie de décrire un domaine, la première chose à faire est de lui donner un nom. Il faut alors un terme général mais fédérateur. Celui-ci doit être suffisamment précis pour borner le

1. Software renewal, renovation, reclamation, salvaging, software archeology, rescue engineering, inverse engineering, etc.

domaine mais suffisamment large pour recouvrir tous les concepts qui en font partie. Ici on retiendra le terme ***ré-ingénierie des logiciels***[1] ou plus simplement *ré-ingénierie*[2].

Pour avoir une idée intuitive de ce que recouvre ce terme remarquons qu'il contient d'une part l'idée de *re*-faire et d'autre part l'idée d'ingénierie[3]. Alors que l'ingénierie consiste à trouver une solution à partir d'un problème, dans le cas de la *ré-ingénierie* une solution insatisfaisante est déjà disponible. Souvent la qualité de cette solution est médiocre car elle a été construite sans respect des règles d'ingénierie. On pourrait alors parler d'ingénierie *a posteriori*.

*Ce qui caractérise le plus la ré-ingénierie est sans doute la prise en compte de l'existant dans le processus d'évolution d'un logiciel*. C'est une solution intermédiaire entre la maintenance (solution se caractérisant par une certaine stabilité) et un nouveau développement (approche ex nihilo, une révolution plutôt qu'une évolution). Une définition plus précise est nécessaire ; nous utiliserons celle proposée par Arnold [Arno93] .

La ***ré-ingénierie*** des logiciels est l'ensemble des activités :
- (A) qui améliorent la compréhension que l'on a d'un logiciel

ou
- (B) qui améliorent le logiciel lui-même, généralement dans le but
  - (B.1) d'accroître sa facilité de maintenance ou d'évolution,
  - (B.2) de faciliter la réutilisation de ses composants.

*Cette définition recouvre un large spectre d'activités*. Par exemple afficher un programme sous forme indentée peut être considéré comme une technique de ré-ingénierie, tout comme la production d'un programme Ada à partir de sources Fortran.
Arnold justifie une définition si large par le fait qu'elle correspond à l'usage actuel du terme . Bien que certains la voient comme une manière de galvauder le terme "ré-ingénierie", elle présente l'avantage d'être fédératrice et est cohérente dans la mesure où la problématique qu'elle recouvre peut être considérée comme un tout. Certes cette problématique n'est pas neuve, mais maintenant elle a un nom ! Avec une telle définition, ces dernières années bon nombre de chercheurs se sont trouvés, tel monsieur Jourdain, faisant de la ré-ingénierie sans le savoir...[4]

Revenons à des aspects plus cartésiens. Avec cette définition, *la ré-ingénierie est définie par un but et non pas par un procédé ou une technique*. Ainsi l'utilisation d'une technique particulière peut être considérée comme une activité de ré-ingénierie dans certains cas seulement. Par exemple l'analyse statique de programmes peut être une technique d'optimisation de programmes

1. Nous précisons qu'il s'agit de logiciels car dans le début des années 90, une mode bien plus forte est née dans le domaine de la gestion d'entreprise : la ***ré-ingénierie des processus*** ("business-process reengineering"). N'ayant pas de rapport direct avec la ré-ingénierie des logiciels il n'y sera plus fait mention par la suite.
2. Tout comme "maintenance", il s'agit d'un anglicisme. Il n'existe pas d'orthographe standard pour ce mot, ni en français, ni en anglais (*réingénierie* [Raul95], *ré-ingénierie*, *reengineering* [Arno93], *re-engineering* [Your89]). Certains auteurs utilisent directement *reengineering* en français (dans le cas de la ré-ingénierie des processus) [HammCham93] [Jaco94].
3. L'utilisation du terme ingénierie est abusive dans le cas du génie logiciel (Section I.2.2.2) et à fortiori bien plus encore dans le cas de la ré-ingénierie puisque le niveau de maturité de ce thème est bien inférieure.
4. Notre cas n'est pas un cas isolé. Voir par exemple le passage de [ChenRama89] à [Chen95]. En fait cette situation n'est pas étonnante dans la mesure où ce n'est souvent qu'après des années qu'une problématique et qu'une approche sont clairement identifiées. Mettre un terme sur un concept est un énorme avantage du point de vue de la communication car il permet de rallier différentes forces pour un but commun. Ce processus n'est pas propre à la ré-ingénierie ; par contre les intérêts commerciaux associés font que parfois ce terme est utilisé plus à la recherche de l'illusion que pour désigner une réalité.

(si l'on cherche à obtenir un code exécutable efficace) ou une technique de ré-ingénierie (si l'objectif est d'obtenir des informations pour faciliter la compréhension du programme).

Remarquons aussi que la définition est composée de deux parties. Bien qu'il n'existe pas de frontière nette entre ces deux parties, l'idée intuitive est que le logiciel n'est modifié que dans le deuxième cas[1].

- (A) **Compréhension du logiciel**. Il s'agit par exemple de prendre des notes, de discuter avec les chargés de maintenance pour collecter des informations sur le logiciel, de survoler les sources ou encore d'examiner son exécution, etc.
- (B) **Maintenance et réutilisation**. La définition met clairement à jour que la *ré-ingénierie* a un rapport avec la *maintenance* (B.1), mais aussi avec la *réutilisation* (B.2). Le premier aspect (B.1) correspond à ce qui a été appelé la *maintenance préventive* (Section I.3.2.3)[2]. La réutilisation est une thème périphérique dans cette thèse et aucune section ne lui est consacrée.

L'utilisation du terme "généralement" dans la partie (B) est importante. Ce mot ouvre la porte à d'autres utilisations éventuelles et rend la définition moins stricte (et plus floue...). L'idée n'est pas de dénaturer le concept de ré-ingénierie mais de reconnaître le fait que dans bien des cas ces activités sont mises en oeuvre pour réaliser des objectifs complexes. Par exemple, il est clair qu'une activité de ré-ingénierie consistant à passer d'un logiciel écrit en assembleur à un logiciel Ada est une tâche coûteuse. Lors d'une telle transformation, il est naturel de vouloir intégrer de nouvelles fonctionnalités, de rendre le programme plus portable, plus efficace, etc. Ces objectifs ne ternissent pourtant pas fondamentalement l'idée de ré-ingénierie.

Soulignons aussi que l'on parle de ré-ingénierie *de logiciels* plutôt que de *programmes*. La distinction entre logiciel et programme est importante (Section I.2.2.3). Ici, elle suggère que tout type de documents peut être pris en compte dans une activité de ré-ingénierie. D'ailleurs *la définition ne limite pas a priori les sources d'informations utilisées* (certaines activités peuvent avoir recourt aux souvenirs des personnes ayant participé au développement du logiciel, à des connaissances liées aux domaines d'applications, etc.). Dans l'état actuel de la pratique le code source des programmes joue cependant un rôle prédominant. Plus encore dans cette thèse.

*Cette définition ne fait nullement allusion à une automatisation des activités de ré-ingénierie*. Pourtant très souvent lorsque l'on débute un exposé sur la ré-ingénierie, la première objection qui vient est qu'il est utopique de vouloir passer d'un programme de qualité douteuse à des spécifications ou à des documents de conception. Clairement une telle transformation ne peut être réalisée automatiquement, mais après tout personne ne prétend cela[3]. La définition de la ré-ingénierie laisse une large place pour l'effort humain. L'*ingénierie* est une activité que très

1. En fait tout dépend de ce que l'on met derrière le terme logiciel. Comme nous allons le voir la rétro-ingénierie rentre dans le premier cas de figure et pourtant elle génère des documents ou des spécifications décrivant le logiciel. Si l'on admet que le logiciel n'est pas composé uniquement de son code exécutable, ces documents font partie du logiciel et il est donc modifié. Ces problèmes de terminologie ne sont que rarement pris en compte et la taxonomie qui suit n'est qu'approximative.
2. Arnold distingue la facilité de maintenance (B.1) et la facilité d'évolution (B.3) pour éviter les problèmes liés à la définition de maintenance (Section I.3.2.1).
3. L'intelligence artificielle dans ses débuts a donné lieu à de fortes expectatives, par la suite beaucoup se sont révélées irréalisables. Le phénomène est le même dans le cas de la ré-ingénierie et les enjeux commerciaux laissent parfois la parole à certains prophètes trop enthousiastes.

partiellement automatisée et pourtant fort pratiquée ; alors pourquoi exiger a priori qu'une activité de *ré-ingénierie* ne demande aucun effort ? La seule conséquence est que la supériorité de cette approche par rapport à une approche ex nihilo est beaucoup plus mitigée.

### I.5.2.2 *Un cadre pour une taxonomie*

"Ré-ingénierie" est un nom acceptable pour un cheval de bataille mais il n'indique pas quelles armes vont être utilisées concrètement. D'autres termes doivent être introduits pour pouvoir différencier les activités de ré-ingénierie. Même si une taxonomie n'est pas très précise, elle permet de toute façon de faire ressortir différents aspects dans un domaine[1]. Mais sur quels critères une taxonomie doit-elle s'appuyer ? Il n'existe pas de réponse générale à cette question. Cependant on peut affiner le terme ré-ingénierie en fonction (1) du fait que le logiciel soit modifié ou pas, (2) du but poursuivi (faciliter la maintenance, réutiliser des composants), (3) de la complexité de la tâche, (4) des techniques utilisées, (5) des sources d'informations utilisées, (6) du type de logiciel concerné, etc.

#### *Des vues du logiciel plus ou moins abstraites...*

Remarquons cependant que les taxonomies existantes sont principalement basées sur la notion de ***vue du logiciel***[2] et sur une caractérisation de ces *vues* [Arno93][ChikCros90][3]. Le cycle de vie peut être utilisé pour ordonner les différentes vues par niveaux d'abstractions. Par exemple, le flot de contrôle d'un programme ou son code source sont considérés comme des vues d'implémentation alors qu'un diagramme entité-relation est considéré comme une vue de conception. Soulignons qu'une telle classification est subjective[4] (figure 19.a).

#### *Des transformations de vues...*

On peut voir les activités de ré-ingénierie comme des transformations de vues. Elles sont classées en fonction des niveaux d'abstractions de la vue de départ et de celle d'arrivée. Considérer la différence de niveau d'abstraction à un sens. Par exemple il semble naturel d'utiliser des termes différents pour le passage d'un programme à une spécification et l'extraction du flot de contrôle à partir d'un texte source. La figure 19 présente différentes possibilités et les termes associés. Ceux-ci sont décrits dans les sections suivantes.

---

1. Par exemple le principal intérêt de la taxonomie de maintenance (Section I.3.2.2) est qu'elle identifie différents objectifs possibles, même si en pratique il est souvent difficile de classer précisément une activité donnée. Les taxonomies concernant la ré-ingénierie souffrent des mêmes problèmes. Les critères définis ci-dessous ne définissent pas des frontières nettes. Dans le cadre de cette thèse seul l'esprit est important.
2. "Software view" dans [Arno93].
3. La notion de vue intervient dans bien des domaines de l'informatique (bases de données, systèmes d'informations, représentation de connaissances, intelligence artificielle, environnement de génie logiciel, etc). Alors que tout le monde s'accorde sur ce qu'est plus ou moins une vue, il n'existe pas de consensus sur ce dont il s'agit précisément (tout comme pour la notion de version).
   Dans le cadre de cette thèse la notion de vue n'est pas affinée. On parlera de "vues du logiciel". Un logiciel est un ensemble d'informations très complexe et les activités de développement et de maintenance doivent pouvoir considérer des vues adaptées à leurs besoins.
4. Les différents auteurs la définissent d'ailleurs uniquement en donnant des exemples [ChikCros90] [Arno93]. En fait ce qu'il est important de retenir dans le cadre de cette thèse c'est que l'on peut distinguer différents niveaux d'abstractions ; leur nombre et leurs limites exactes importent peu.

### I.5.2.3 *Retro-ingénierie et retro-conception*

Le terme *rétro-ingénierie*[1] est un terme essentiel dans le domaine de la ré-ingénierie.
Son origine provient de l'ingénierie des systèmes matériels [Reko85]. Dans ce domaine, la rétro-ingénierie consiste à développer les spécifications d'un système matériel complexe produit à grande échelle en examinant des exemplaires de ce produit (par exemple des micro-processeurs). L'objectif est de construire des clones du système original ou d'en découvrir certains aspects techniques. Dans le domaine du génie logiciel les objectifs sont différents mais le principe est le même.

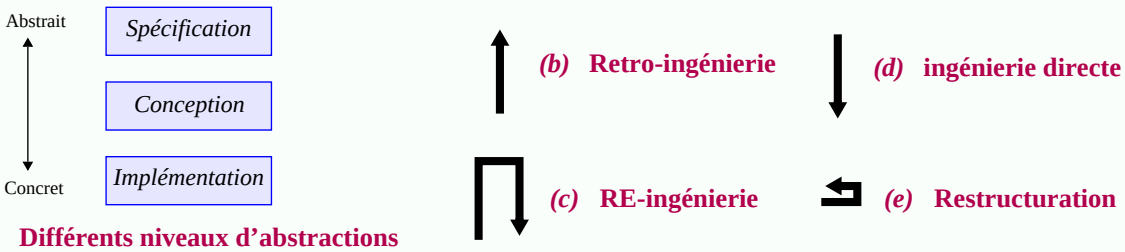
Une activité de ***retro-ingénierie*** consiste à analyser une représentation d'un logiciel pour générer des informations plus abstraites concernant ce logiciel.

Cette définition fait implicitement référence à une transformation d'une vue concrète à une vue plus abstraite (figure 19.b). Il s'agit par exemple de produire des documents de spécifications à partir d'un programme source. A première vue cette idée peut sembler aberrante. Elle est en tout cas contraire au principe général de l'ingénierie qui consiste à passer progressivement de l'abstrait au concret (figure 19.d)[2]. Non seulement la retro-ingénierie a un sens, mais en plus elle correspond potentiellement à des enjeux économiques importants (voir plus loin la Section I.5.3). Une fois l'idée de retro-ingénierie acceptée, tout de suite vient la question de la faisabilité et les controverses qui s'en suivent [WeidHH95]. En fait trop souvent les discussions manquent de discernement[3]. Il est clair que le terme rétro-ingénierie recouvre aussi bien des transformations irréalisables que des transformations plus modestes. Soulignons de nouveau que le degré d'automatisation et les informations utilisées pour assurer la transformation ne sont pas mentionnés.
Le terme *retro-conception*[4] correspond à un sous-ensemble ambitieux de la retro-ingénierie.

Une activité de ***retro-conception*** est une activité de retro-ingénierie visant à (re)générer des informations de conception en utilisant des connaissances externes au logiciel observé (domaine d'application, expérience, déductions et raisonnements flous, etc).

Dans tous les cas la retro-ingénierie est un processus d'analyse et non pas de modification (partie **(A)** de la définition de ré-ingénierie).

*figure 19* Une taxonomie basée sur les changements de niveau d'abstraction




1. Reverse-engineering
2. L'introduction du terme rétro-ingénierie est d'ailleurs souvent accompagnée par le terme ***ingénierie directe*** ("forward engineering" en anglais).
3. Selon certains la retro-ingénierie correspondrait à "reconstruire le cochon à partir des saucisses et du jambon...".
4. Design recovery

### I.5.2.4 *RE-ingénierie*

L'une des confusions les plus importantes qui régne dans le domaine de la *ré-ingénierie* concerne justement l'utilisation de ce terme. Dans cette thèse, c'est le sens proposé par Arnold qui est utilisé. Comme nous l'avons vu il s'agit d'une définition lâche, utile surtout pour donner un nom au domaine. Les autres taxonomies proposent un sens (un peu) plus précis [ChikCros90]. Dans cette thèse il sera associé au terme *RE-ingénierie* (RE en majuscules).

La ***RE-ingénierie*** est l'analyse et la modification d'un logiciel mettant en jeu une étape de retro-ingénierie puis une étape d'ingénierie directe. Au cours de cette transformation les fonctionnalités du logiciel peuvent éventuellement être modifiées *(figure 19.c)*.

Cette définition se distingue de celle d'Arnold, par le fait que le logiciel est nécessairement modifié et par le fait que les objectifs de cette modification ne sont pas définis[1].

### I.5.2.5 *Restructuration*

Une ***restructuration*** est une transformation d'une représentation en une représentation sémantiquement équivalente sans changement de niveau d'abstraction *(figure 19.e)*.

L'élimination des instructions goto est un exemple typique de restructuration. Dans ce cas précis le terme *restructuration* fait référence à la *structure* du programme ("structure" au sens *programmation structurée*). Une telle transformation n'est valide que si elle ne change pas la sémantique du programme.

### I.5.2.6 *Discussion*

Les trois termes principaux de la ré-ingénierie ont été présentés ci-dessus. Il s'agit de la *retro-ingénierie*, de la *RE-ingénierie* et de la *restructuration*. Les taxonomies introduisent en général plus de termes [Your89] [ChikCros90] [Yu91] [Arno93] [BourA94]. Hélas elles ne convergent pas nécessairement et les frontières entre les différentes classes sont souvent bien floues.

L'un des problèmes le plus ennuyeux est celui concernant l'utilisation du terme ré-ingénierie comme terme générique par certains et comme terme plus spécifique par d'autres (sens associé au terme RE-ingénierie dans cette thèse). Pour éviter ce problème, dans le deuxième cas certains auteurs utilisent le terme ***redéveloppement*** [Ulri90]. Les termes ***rénovation*** et ***modernisation*** supposent que l'on change de technologie, le terme ***migration*** que l'on change de plate-forme.

Plusieurs termes sont également associés aux activités facilitant la compréhension du logiciel mais ne modifiant pas celui-ci (partie **(A)** de la définition de ré-ingénierie). Une activité de ***redocumentation*** consiste à re(créer) la documentation d'un logiciel[2]. Le terme ***compréhension de programme***[3] est également fort utilisé [Corb89][RobsBCM91]. Il fait plus référence à une

1. La deuxième phrase correspond à une ouverture similaire à l'utilisation de "généralement" dans la définition d'Arnold. Elle correspond à une transformation horizontale dans la figure 19.c. Quelques fois elle est appelée ***ingénierie transversale*** (transversal engineering).
2. Il s'agit de documentation au sens large. Ce peut être par exemple des commentaires dans les programmes, des documents textuels ou graphiques, des descriptions du flot de contrôle, etc. Normalement le terme redocumentation est utilisé lorsque l'on ne change pas de niveau d'abstraction, sinon c'est le terme retro-ingénierie qui est utilisé.

activité mentale que la retro-ingénierie et surtout il correspond plus à un objectif qu'à un ensemble de technique. La ***visualisation de programmes*** ou ***visualisation de logiciels*** est un ensemble de techniques souvent employées dans ce cadre [ConsMR92] [EickSS92] [Gull92b] [BakeEick94] (bien qu'elles ne se limite pas à celui-ci [RomaCox93][Mend93][StasPatt91]). Le terme ***analyse de programme*** est parfois utilisé dans un sens large, plutôt pour désigner des techniques de retro-ingénierie.

D'autre termes sont introduits lorsque les activités se focalisent sur des aspects particuliers du logiciels. Par exemple dans le contexte des systèmes d'informations on parle de ***ré-ingénierie des données*** (il peut s'agir des données elles-mêmes ou des schémas de données) [AebiLarg94] [PremBlah94].

D'un point de vue historique, c'est sans doute le terme restructuration qui est le plus ancien. Son introduction correspond aux techniques de restructuration automatique du flot de contrôle développées au début des années 80. Ce terme a aussi été utilisé pour désigner la problématique sous-jacente, problématique aujourd'hui associée au terme ré-ingénierie[1].

## I.5.3 Importance du problème

L'importance de la ré-ingénierie ne se caractérise pas par la profusion de nouveaux termes, mais plutôt par les enjeux économiques et scientifiques correspondant à ce thème.

### I.5.3.1 *Importance de la compréhension, de la maintenance et de la réutilisation*

Reprenons la définition de la ré-ingénierie. Le premier objectif est de faciliter la *compréhension des logiciels*, le deuxième d'en faciliter la *maintenance*, le troisième de faciliter la *réutilisation* de ses composants. L'importance de la ré-ingénierie se justifie déjà par l'importance de ces différents objectifs.

- **Compréhension des logiciels**. Tout au long du cycle de vie, la compréhension du logiciel est une activité centrale. Certaines estimations indiquent que 50% du temps de la maintenance est consommé pour "comprendre" le logiciel à maintenir [PariZveg83][MIL-srah94]. Dans certains cas, il pourrait même s'agir de 90% [Stan84]. Lorsque l'on connaît les coûts de la maintenance on comprend pourquoi la compréhension des logiciels doit être assistée ! De surcroît cette activité n'est pas spécifique à la maintenance. Par exemple les équipes de validation de logiciels critiques sont amenées à faire des efforts importants pour comprendre un logiciel existant afin de modéliser son comportement dans un formalisme permettant certaines validations. En fait, dès que le logiciel est consulté par des équipes ne participant pas directement à son développement, il est essentiel de disposer d'outils facilitant sa compréhension. Selon certains auteurs *faciliter la compréhension des logiciels est un défi majeur pour la recherche dans cette décennie* [Corb89].
- **Maintenance**. Comme nous l'avons vu dans la Section I.3 simplifier la maintenance est un

3. Program understanding

1. Il est significatif par exemple de constater que les deux ouvrages de R..S. Arnold portent respectivement le nom "Tutorial on Software Restructuring" en 1986 [Arno86] et "Software Reengineering" en 1993 [Arno93]. Ce sont tous les deux des compilations d'articles sur le même sujet.

problème majeur dans le domaine du génie logiciel. Nous ne reviendrons pas sur l'importance de la maintenance en général (Section I.3.3). Remarquons cependant que dans les recommandations faites par le SEI pour combattre les problèmes de maintenance, la ré-ingénierie vient en première position [DartCB93].

- **Réutilisation**. La réutilisation du logiciel est un autre cheval de bataille dans le champ du Génie Logiciel. Puisque le développement de nouveaux logiciels est coûteux pourquoi pas ne réutiliser des composants déjà existants? Malgré les efforts entrepris dans cette direction, les résultats sont encore mitigés et tout comme la ré-ingénierie, la réutilisation est une approche controversée. Développer des composants réutilisables est l'un des problèmes importants dans ce domaine. Créer des composants à partir de zero est coûteux et dans ce contexte la ré-ingénierie peut être vue comme un moyen d'extraire d'un logiciel déjà existant de tels composants[1]. Cet aspect n'est pas abordé dans la suite de cette thèse mais il contribue largement à l'importance de la ré-ingénierie.

Ci-dessous nous présentons d'autres points justifiant l'importance de la ré-ingénierie. Tout d'abord l'importance des logiciels âgés est présentée, puis la ré-ingénierie est vue comme un moyen pouvant faciliter le transfert de technologie.

### I.5.3.2 *Importance des logiciels âgés*

> *"The us air force estimates that unless it does something about the maintenance "iceberg", it will require 25 percent of the country's 18 to 25 year old to maintain its software by the year 2000! Part of the problem is that applications portfolio in many large organizations is 10, 15, or even 20 years old - and obviously growing older each day. Even if the original programs had been perfectly designed and coded, 20 years of patching and modifying is almost certain to lead to a maintenance nightmare. But in fact, most programs developed in the sixties and seventies were NOT "perfect" in any sense of the word ; there were difficult to maintain from the day they were put into operation".Yourdon*

Bien que ce ne soit pas nécessairement le cas, la ré-ingénierie est souvent associée aux ***logiciels âgés***[2]. En tout cas la ré-ingénierie se justifie par l'importance des logiciels déjà existants.
Selon certaines estimations plus de 100 milliards de lignes de code seraient actuellement en service sur la planète [Lafu90][Ulri90][3]. Plus de 80% seraient non structurées et ne posséderaient pas de documentation à jour [Lafu90]. La grande majorité serait écrites en COBOL[4]. On trouve aussi beaucoup de langages ésotériques, entre autre des langages propriétaires fonctionnant sur des architectures matérielles très spécifiques. Les principes élémentaires du génie logiciel n'ont pas été appliqués dans le développement des logiciels âgés. De surcroît les années ou les décennies de maintenance n'ont fait que contribuer à leur détérioration. Généralement ces

1. On parle alors de la ***ré-ingénierie pour la réutilisation*** (reengineering for reuse) [ArnoFrak91].
2. "Legacy system" en anglais., soit d'après la définition de K. Bennett "Legacy systems may be defined informally as 'large software systems that we don't know how to cope with but that are vital to our organization' " [Benn95].
3. Bien évidemment ce ne sont que de grossières estimations mais l'ordre de grandeur est intéressant. Rappelons que le DoD maintient actuellement 1.4 milliard de lignes de code [AikeMR94].
4. Dans un workshop sur la ré-ingénierie le chiffre de plus de 80% a été avancé pour COBOL (1st SEI Workshop on Reengineering, may 1994).

logiciels ont largement “grandit” au fur et à mesure de leur maintenance et même si leurs objectifs initiaux étaient relativement limités, ce sont devenus des systèmes complexes.

Puisque ces logiciels sont de si mauvaise qualité pourquoi ne pas “les mettre à la poubelle” ? Autrement dit pourquoi ne pas choisir une approche ex nihilo et lancer de nouveaux développements ? Dans bien des cas se serait tout simplement *trop coûteux et trop risqué.* Il faut bien comprendre que pour certains problèmes complexes, ce n’est qu’au cours de nombreuses années et après de nombreux ajustements que les fonctionnalités souhaitées ont été obtenues. Les logiciels âgés ne sont pas seulement âgés, ils ont aussi acquis une grande maturité. Abandonner un tel acquis n’est pas toujours concevable car il peut être bien plus difficile de décrire le résultat souhaité que de se baser sur un logiciel existant. Autrement dit *le code source d’un logiciel âgé peut parfois se révéler la meilleure spécification du système à réaliser* (En plus ces spécifications sont exécutables!...). La retro-ingénierie prend alors tout son sens.

*Si ces situations semblent aberrantes du point de vue du Génie Logiciel, c’est aussi parce que l’on confond trop souvent la forme et le fond*. L’ingénieur logiciel, concentré sur des critères de qualité, aura naturellement tendance à voir avec respect un logiciel écrit dans un langage orienté-objet et avec dédain un langage écrit en Fortran ; même si ce dernier est le résultat d’années de recherche et implémente des algorithmes très sophistiqués. La tendance naturelle du génie logiciel à se concentrer sur la forme ne doit cependant pas faire oublier le fond. L’inverse est vrai pour l’industrie, souvent plus préoccupée par le fond que par la forme. Le développement de l’ingénierie correspond justement à une vision équilibrée de ces deux points de vue.

L’importance de la ré-ingénierie peut donc se justifier par l’importance des logiciels âgés. Certes les problèmes de maintenance auxquels l’industrie est confrontée aujourd’hui sont liés pour une large part aux erreurs commises dans le passé, mais le vieillissement des logiciels est un phénomène inévitable [Arno93] [SchwStra93] [Parn94].

### I.5.3.3 *Importance du transfert de technologie*

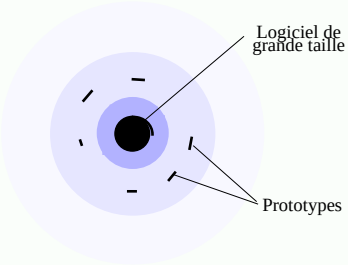
> *“The authors and owners of new software products often look at aging software with disdain. They believe that, if the product had been designed using today’s techniques, it wouldn’t be causing problems. Such remarks remind me of a young jogger scoffing at an 86 year old man (who, unknown to the jogger, was a champion swimmer into his 50’s) and saying that he should have had more exercise in his youth. Just as we will all (if we are lucky) get old, software aging can, and will occur in all* ***successful*** *products. We must recognise that it will happen to our products and prepare for it. When old age arrives, we must prepared to deal with it.” D.L.Parnas*

Si les logiciels vieillissent ce n’est pas seulement parce que le temps passe, c’est aussi parce que les critères de juvénilité changent. Autrement dit, l’évolution technologique est un facteur pouvant pousser à qualifier un logiciel du terme “dépassé”, même si sa mise en service est récente.

Dans le modèle hélicoïdal, les “traces” laissées par les trajectoires n’ont pas été prises en compte. Autrement dit nous n’avons pas étudié l’influence des artefacts produits au cours du temps. Cet aspect est pourtant fondamental car il change radicalement les possibilités d’évolution laissées à l’industrie. C’est aussi une différence de plus séparant la recherche et l’industrie : alors que la

recherche élabore de petits prototypes éphémères, l'industrie produit des logiciels de grandes tailles voués à rester longtemps (figure 20).

*figure 20* Modèle hélicoïdal et produit logiciel

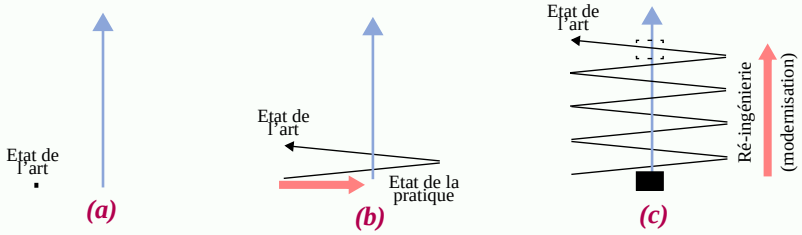



Dans le domaine de la recherche, de petits prototypes sont élaborés. Ils ne freinent en rien sa progression. Au contraire l'industrie produit des logiciels grandissant peu à peu. Ceux-ci sont de plus en plus *lourds* à gérer et à chaque fois leur *inertie* augmente. Si un logiciel a du succès, l'organisation risque de s'y attacher tellement qu'à terme ce logiciel va devenir un véritable boulet à traîner. En fait même si l'industrie voulait bien se projeter dans l'avenir la maintenance de ses logiciels âgés la retient vers le passé.

Le terme ***transfert de technologie*** fait généralement référence au passage d'une technologie de l'état de l'art à l'état de la pratique [Scac94]. Son importance est évidente pour le développement du génie logiciel. L'intention de cette section est de montrer l'importance de la ré-ingénierie comme moyen de transfert de technologie.

L'inertie des logiciels industriels pose un problème par rapport à l'évolution technologique. Une technologie qui pourrait être acceptée pour le développement de nouveaux logiciels ne l'est pas forcement si elle n'est pas applicable aux logiciels déjà existants. Dans ces conditions des outils de ré-ingénierie permettant de moderniser les logiciels sont forts utiles. On parle alors de ***modernisation*** ou de ***rénovation***. En fait quoi qu'il arrive, *il y aura toujours une différence entre l'état de l'art et l'état de la pratique* ; tout simplement parce que le transfert de technologie est particulièrement lent et hasardeux alors que la recherche ne cesse d'avancer[1] (figure 21).

*figure 21* Modèle hélicoïdal et transfert de technologie




Supposons qu'à l'instant $t_o$ une nouvelle technologie semble suffisamment au point (a). Le temps de transfert est généralement de plusieurs années, voire plus d'une décennie [RedwRidd85][Scac94]. Pendant ce temps l'état de l'art avance bien évidemment (b). Lors de sa maintenance, le logiciel est chaque fois plus pesant et la distance entre l'état de l'art et l'état de la pratique s'accroit (c). La ré-ingéniérie peut être vue comme un moyen de diminuer cette distance en se basant sur le logiciel existant.

Historiquement on peut remarquer par exemple que l'on est passé de la programmation ad-hoc, aux assembleurs, des assembleurs aux langages de haut niveau, puis à la programmation structurée, à la programmation modulaire et enfin à la programmation orientée objet. A chaque fois les standards de qualité augmentent. La préoccupation pour faciliter ces dernières étapes se retrouve dans les travaux de ré-ingéniérie. On trouve par exemple des travaux facilitant la compréhension ou modernisation des logiciels écrits en assembleurs [Clev89] [LakeBlan95], la

1. Par exemple actuellement les langages orientés objets font leur apparition dans l'industrie du logiciel, notamment avec le langage C++. Former tout le personnel à un nouveau paradigme n'est pas une mince affaire, surtout que les activités d'implémentations restent avant tout un savoir-faire qui s'acquiert au cours d'années d'expérience. Pendant des années on risque donc d'avoir à maintenir des logiciels orientés objets de qualité médiocre, en plus dans un contexte où l'on sait que la maintenance des logiciels orientés objets posent de nouveaux problèmes [LetjMR92][CartShep95]. Ce qui vaut la boutade suivante : "*La programmation en C++ chez les industriels est comme le sexe chez les adolescents : tout le monde en parle ; tout le monde croit que le voisin le fait ; presque personne le fait ; ceux qui le font le font mal ; ils pensent que la prochaine fois se sera mieux et ils ne prennent pas de précautions...*".

restructuration du flot de contrôle des programmes Cobol ou Fortran [Lyon81] [SneeJand87] [Arno89], la modularisation à posteriori des logiciels de grandes tailles [SchwPlat89] [SchwStra91] [MarkNBBK94], la découverte à posteriori d'objets dans le code [Jaco91] [LanoHaug92] [GallKlos93].
De même les applications de gestions et les systèmes d'informations sont passés de logiciels en assembleur à des logiciels en Cobol, puis aux bases de données relationnelles et peut être finalement orientées objet. Des travaux de ré-ingénierie supportent cette évolution.

L'idée de la ré-ingéniérie comme moyen de transfert de technologie est bien plus attirante en théorie qu'en pratique ; sa mise en oeuvre n'est jamais simple et les résultats sont souvent mitigés. Ceci dit il est clair que plus l'on attend, plus la différence entre l'état de l'art et l'état de la pratique est difficile à franchir. Au contraire, lorsque une nouvelle technologie apparaît, elle devrait être conçue pour évoluer et, si possible, des travaux de recherche en ré-ingénierie devraient être entrepris pour faciliter son application aux logiciels déjà existants.

## I.5.4 Importance des intérêts et des efforts

> *"Organizations have only three choices: (a) They can continue allowing their systems to deteriorate until they go out of business; (b) they can throw out their old systems and replace them with brandnew systems; or (c) they can turn to re-engineering and reverse engineering. Choice (a) is suicide; choice (b) is too expensive and too risky for most organizations. That leave choice (c) -- so, yes, the RE-3 technologies are going to be key technologies for the nineties*[1] *" (E. Yourdon).*

Dans cette section nous décrivons l'importance de la ré-ingénierie en considérant les intérêts et les efforts menés dans différents secteurs d'activités de l'informatique : l'industrie du logiciel, les producteurs d'outils et les sociétés de services, la recherche.

### I.5.4.1 *L'industrie du logiciel*

La ré-ingénierie des logiciels est un problème industriel plutôt qu'académique.
Du point de vue du *chef de projet* c'est un moyen de préserver les investissements faits dans le logiciel le plus longtemps possible, de diminuer les coûts futurs de la maintenance et d'éviter les risques et les délais de nouveaux développements. Bien évidemment une activité de ré-ingénierie peut être fort coûteuse et risquée [Arno92]. Cette solution n'est pas nécessairement la meilleure mais en tout cas c'est une alternative possible.

Du point de vue des *chargés de maintenance* deux réactions sont possibles face à la ré-ingénierie. Tout d'abord l'utilisation de nouveaux outils et de nouvelles méthodes peut parfois être vécue comme un changement trop radical. Il bouscule les habitudes et certains chargés de maintenance y sont fort réticents [Your89]. Inversement, selon l'enquête du SEI, s'équiper d'outils de ré-ingénierie est aujourd'hui un objectif prioritaire pour de nombreuses équipes de maintenance [DartCB93].

1. RE-3 : Re-engineering, Restructuring, Reverse Engineering

Les activités majeures de ré-ingénierie sont normalement déconnectées des activités de maintenance quotidienne. Le travail du *chargé de ré-ingénierie* peut être accompagné d'une saveur bien différente de celle qu'a un chargé de maintenance ou une personne effectuant un portage. Ces derniers travaillent dans l'ombre ; ils effectuent un travail neutre : le logiciel fonctionne déjà et ils ne doivent "que" maintenir celui-ci ou le porter. Personne ne risque de s'extasier devant leur travail quotidien. Au contraire le chargé de ré-ingénierie travaille dans un but noble et se voit parfois tel "Champollion" en train d'essayer de décrypter les mystères enfouis dans un logiciel d'un autre temps[1]. Faire du neuf avec du vieux est un acte positif. Un projet visant à passer d'un logiciel en assembleur à un logiciel en Ada sera, au début, regardé avec suspicion, mais s'il réussit, avec admiration. *Cette différence explique sans doute pourquoi la maintenance, après des décennies, n'est pas un sujet populaire et pourquoi la ré-ingénierie a pu le devenir en quelques années.*

#### *Evolution dans le temps*

Lorsque l'on décrit la ré-ingénierie comme étant un "nouveau cheval de bataille pour combattre la maintenance" c'est une exagération. Cela fait déjà longtemps que des activités de ré-ingénierie sont pratiquées dans l'industrie ; disons simplement qu'à l'époque il s'agissait d'un mulet plutôt que d'un cheval. Ces activités n'étaient ni automatisées, ni publiées. Pourtant depuis longtemps, la première mission d'une nouvelle recrue dans un projet de maintenance est de se "plonger" dans le code pendant les premiers mois. Certaines sociétés n'ont-elles pas employé des stagiaires pour mettre des commentaires du code, pour essayer de rédiger une nouvelle documentation pour un système, etc. Effectuer ces tâches manuellement peut être aussi coûteux que fastidieux. Par exemple dans [Phil83], il est rapporté que 6 personnes-année ont été nécessaires pour retrouver la fonctionnalité d'un système de simulation écrit en 20 000 lignes de Fortran.
C'est vers le milieu des années 80, que les premiers services de restructuration sont apparus. Au début considérée avec suspicion, la ré-ingénierie est peu à peu apparue comme une approche nécessaire face à la maintenance [Your89][Arno93]. Notons qu'actuellement les systèmes militaires et les systèmes d'informations de grandes tailles sont les principaux consommateurs de cette technologie[2].

### I.5.4.2 *Les producteurs d'outils*

Depuis que l'on sait que l'automatisation peut faire gagner du temps et de l'argent dans un processus de ré-ingénierie, la production d'outils de ré-ingénierie s'est considérablement développée. Plus de 300 outils seraient aujourd'hui disponibles sur le marché[3]. En 1990 certaines prévisions indiquaient une augmentation du marché de 40% par an [Hann91]. En fait cette tendance est positive non seulement pour ces outils mais aussi pour les environnements de génie

1. Certains auteurs utilisent d'ailleurs le terme "software archeology" [Gras92a].
2. Par exemple l'US navy est l'une des premières organisations a avoir organisé des workshop sur le thème de la ré-ingénierie. Depuis, ils ont lieu chaque année. Dans le domaine des systèmes d'informations, il existe une grande motivation pour passer de systèmes écrits dans des langages comme Cobol à des bases de données relationnelles, des langages plus modernes, des architectures réparties, etc.
3. C'est le chiffre annoncé par STSC (Software Technology Support Center; organisme dépendant de US Air Force et maintenant à jour un catalogue des outils de maintenance et de ré-ingénierie). Bien sûr parmi tous ces outils, nombreux sont ceux qui proposent des fonctionnalités identiques et bien modestes. Bien souvent afficher le graphe d'appel des procédures est suffisant pour que l'outil soit qualifié d'outil de retro-ingénierie.

logiciel en général. Disposer d'outils permettant de prendre en compte des logiciels existants est un argument de taille pour un tel environnement. La ré-ingénierie est aussi devenue la raison sociale de certaines sociétés autrefois tournées vers des activités moins lucratives ou moins porteuses[1].

#### *Evolution dans le temps*

Historiquement c'est à partir de 1984 que les premiers services de restructuration ont été proposés pour des applications réelles [Ulri90]. Dans les premières années les sociétés envoyaient leurs sources et récupéraient le résultat restructuré. Après quelques années la demande a été si forte que des outils plus robustes ont été développés et mis sur le marché. Aujourd'hui ceux-ci sont généralement intégrés à des environnements de génie logiciel beaucoup plus complets. Bien que les performances de ces outils constituent encore un problème majeur [SmitMulaSmit90] [DartCB93] les producteurs d'outils de ré-ingénierie ont un avenir prometteur.

### I.5.4.3 *Les sociétés de services*

L'apparition d'une nouvelle technologie, conjuguée au mysticisme entourant la ré-ingénierie, a donné lieu à une très forte demande en termes de sociétés de conseils et de services. Celles-ci sont chargées de classer et d'évaluer les outils, d'estimer les coûts et les risques, de définir des processus de ré-ingénierie, de créer des bases d'expériences ; elles peuvent prendre en charge les activités de ré-ingénierie. Le service est l'un des aspects les plus juteux de la ré-ingénierie. Certaines indications montrent qu'en 1990 les dépenses en services était 4 fois supérieures à celles faites pour l'achat d'outils [Arno93]. Le besoin en terme de conseils s'exprime aussi par le nombre important de tutoriaux sur ce thème dans les conférences consacrées au génie logiciel ou aux systèmes d'informations.

### I.5.4.4 *La recherche*

Pendant longtemps la recherche est restée en marge de cette tendance générale, mais aujourd'hui elle est vue comme :

- *un thème de recherche pouvant avoir un impact important sur l'industrie et donc une nouvelle source de financement possible*.
- *un nouveau champ d'application pour certaines techniques*. Certains travaux appliquent des techniques basées sur les réseaux neuronaux [ArseSpra94] où l'intelligence artificielle [KozaLN91] [NingEK92] [BiggMW94] [Quil94]. Dans bien des domaines, l'exécution symbolique s'est révélée être inapplicable et elle peut être utilisée dans le domaine de la ré-ingénierie ou de la compréhension de programmes (grâce à des contraintes différentes) [King81] [CowaInce91].
- *un catalyseur pour d'autre thème*. La ré-ingénierie peut être vue comme complémentaire par rapport à la réutilisation. De même, le fait que des informations soient générées à partir de

1. C'est le cas par exemple de Reasoning System qui commercialise aujourd'hui Software Refinery, l'un des systèmes les plus actifs dans le domaine de la ré-ingénierie [BursKM90][MarkNBBK94][MarkBK94]. Ce système n'est en fait que la consécration d'une technologie développée pour d'autres objectifs aux cours de nombreuses années.

logiciels âgés, permet d'envisager une plus grande automatisation des activités de maintenance et de relancer les recherches en ce sens [Arno93].

- *un moyen pour faire accepter une nouvelle technologie*. Certains chercheurs ont des difficultés pour valider leur travaux ou les faire accepter dans l'industrie. Par exemple dans le cadre du projet EPOS [Lie90] [Munc&al93], un gestionnaire de configurations a été développé, mais il n'a pas pu être validé en grandeur nature. Des recherches pouvant être qualifiée de "ré-ingénierie" ont été entreprises pour essayer d'intégrer des logiciels déjà existants sous contrôle de ce système [Munc93].

#### *Evolution dans le temps...*

Plusieurs auteurs débutent l'histoire de la ré-ingéniérie en 1966 [Your89][Arno94]. Cette année là Böhm et de Jacopini prouvaient qu'un flot de contrôle quelconque pouvait être restructuré en utilisant uniquement des instructions de séquence, des conditionnelles et des itérations [BohmJaco66]. En 1968, Djikstra faisait remarquer les problèmes liés à l'utilisation des instructions Goto [Dijk68]. Il faudra pourtant attendre le début des années 80 pour que ces techniques soient appliquées concrètement à la restructuration de programmes âgés [Lyon81]. Ceci est dû au manque d'intérêt pour la maintenance dans les années 70 (Section I.3.4).
Dans la deuxième moitié des années 80, l'attention des chercheurs s'est tournée vers la restructuration, puis avec le changement de terme vers la ré-ingénierie. Dans le début des années 90 ce phénomène s'est accentué. Aujourd'hui, suite à un nombre croissant de publications et de tutoriaux, la ré-ingénierie et la problématique associée se sont fait connaître par un large public et attirent de plus en plus de chercheurs [Arno94].

### I.5.5 Conclusion

Face aux logiciels âgés, deux approches radicalement opposées sont possibles : (1) *maintenir* ces logiciels, c'est à dire accepter la détérioration de leur qualité jusqu'à ce qu'ils deviennent inutilisables. C'est la solution de facilité, mais elle est plus que risquée... (2) *développer* de nouveaux logiciels. C'est long, coûteux et risqué.
La ré-ingénierie n'est pas la panacée; mais ce peut être *une* solution intermédiaire. Elle n'est pas uniquement liée à la maintenance ; elle l'est aussi à la ré-utilisation et à la compréhension des logiciels. La rétro-ingénierie n'est pas automatique ; tout comme l'ingénierie, elle peut être laborieuse. Ce n'est pas un problème nouveau ; c'est une préoccupation grandissante. Ce n'est pas un ensemble de techniques inédites ; c'est une approche face à un problème. Il ne s'agit pas uniquement de résoudre les problèmes liés à l'utilisation industrielle de techniques obsolètes ; c'est aussi l'occasion pour la recherche d'appliquer des techniques de pointe.

La tendance naturelle qu'a la recherche à "oublier" les logiciels âgés est dangereuse car elle risque de creuser encore plus le gouffre entre recherche et industrie. Ne faudrait-il pas éviter une industrie du logiciel à deux vitesses : d'une part une industrie pouvant suivre l'évolution technologique, d'autre part une industrie plus lourde s'enfonçant peu à peu ? *La ré-ingénierie comme moyen de transfert de technologie a un rôle à jouer dans la communication entre recherche et industrie. Autrement dit la ré-ingénierie est un problème* ***de*** *génie logiciel ; l'ignorer totalement pourrait devenir un problème de plus* ***pour*** *le génie logiciel.*

# I.6 Problématique choisie

Le contexte présenté dans ce chapitre est particulièrement riche en problèmes intéressants, mais dans le cadre de cette thèse nous avons choisi de faire ressortir seulement deux points particuliers :

- **Le besoin d'une rationalisation dans le domaine de la programmation globale**. Cet aspect correspond à l'objectif O3 de cette thèse.
- **La notion de ré-ingénierie globale**. Cet aspect correspond à l'objectif O4[1].

## I.6.1 Vers une rationalisation de la programmation globale

> *"Despite antiproliferation efforts, the number of programming languages and dialects began to rise. This prolonged a sad era in the history of software engineering: the time when programmers were shamans. ...*
> *Part of the problem with these languages is that they were produced before the scientific understanding of what software really was about." (FP Brooks)*

Si la programmation détaillée et la programmation globale ont été opposées, c'est surtout pour faire ressortir l'importance du second thème ; mais la comparaison de leur évolution respective peut également être riche d'enseignements.

#### *Le cas de la programmation détaillée...*

Si l'on se réfère au modèle d'évolution des disciplines d'ingénierie (Section I.2.2.1) le stade d'ingénierie peut être considéré comme atteint par *certaines* activités de programmation détaillée [Shaw90]. C'est le cas par exemple de la conception d'un langage de programmation et de la construction de compilateurs. L'une des conditions nécessaires pour atteindre ce niveau de maturité est d'utiliser des théories et des abstractions plutôt que de se focaliser uniquement sur des aspects concrets.

Par exemple si l'on veut définir des techniques de ré-ingénierie, plutôt que de considérer uniquement le texte source des programmes on utilisera des concepts plus abstraits ; par exemple son flot de contrôle ou son flot de données. Ces abstractions et leurs propriétés ont été largement étudiées et de nombreux résultats sont aujourd'hui disponibles. L'élimination des instructions goto, (application typique de restructuration), n'est qu'un résultat direct des travaux concernant le flot de contrôle [BohmJaco66].

L'un des avantages essentiels de telles abstractions est qu'elles permettent de raisonner sur des programmes sans pour autant se focaliser sur un langage de programmation spécifique. Dans un tel contexte les techniques de description de la sémantique des langages sont fort utiles. Grâce à elles, il est possible de faire ressortir les caractéristiques essentielles d'un langage mais aussi de comparer ces langages avec précision.

1. O3 Favoriser un processus de rationalisation dans le domaine de la programmation globale.
   O4 Définir et explorer le thème de la ré-ingénierie globale.

Il existe par exemple un nombre réduit de modes de passage de paramètres (passage par valeur, passage par adresse, passage par nom, etc.). Lors de la définition d'un nouveau langage de programmation ou de l'étude d'un langage existant, il est fort utile de se référer à ces différentes classes car leurs propriétés ont été étudiées en détail. La présence d'un mode ou d'un autre donne des indications précieuses concernant les techniques d'analyses utilisables, les différentes équivalences entre constructions, les problèmes possibles, etc. Les techniques de transformations de programmes sont basées sur une telle connaissance.

L'histoire de l'informatique a été ponctuée par l'apparition de nombreux langages de programmation dont les aspects techniques masquaient l'absence d'innovations réelles (phase "commerce" du modèle). Des variations "de surfaces" ont permis de justifier de trop nombreux langages impliquant une perte d'énergie considérable tant pour les concepteurs et réalisateurs de ces langages que pour les programmeurs[1]. Le problème se trouve maintenant répercuté sur la maintenance et la ré-ingénierie des logiciels.

Aujourd'hui par contre, grâce à la maturité acquise au cours de plusieurs décennies, la définition de nouveaux langages se fait heureusement dans des conditions bien différentes. Les concepts de syntaxe, de sémantique statique ou d'environnement d'exécution font partie de l'acquis. Il s'agit pour une large part de réutiliser des mécanismes déjà connus ce qui permet de se concentrer sur des aspects plus novateurs mais bien identifiés.

Dans le domaine de la programmation détaillée, l'utilisation d'abstraction et de techniques formelles a permis une certaine "*rationalisation*", tout au moins en ce qui concerne certains aspects liés aux langages de programmation. Même si le nombre de langages n'a pas nécessairement été réduit, il apparaît clairement aujourd'hui qu'il existe une base commune bien plus importante que les différences.

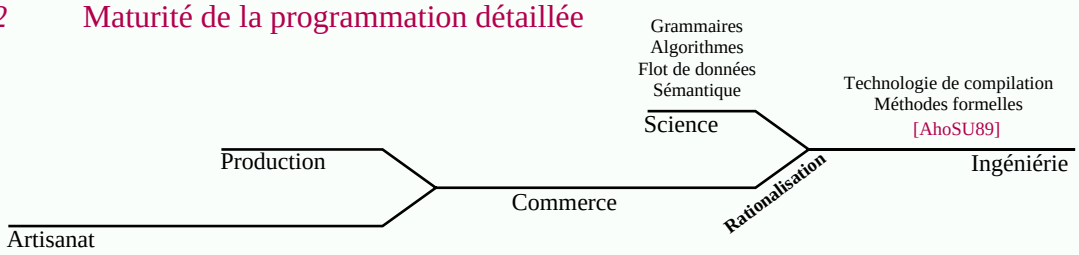
De très nombreux ouvrages offrent des comparaisons entre langages, les évaluent, dégagent des concepts communs. Des taxonomies plus ou moins précises existent, séparant par exemple les langages basés sur un paradigme impératif, fonctionnel, ou déclaratif. Même si les limites sont souvent floues ces taxonomies ont au moins le mérite d'exister. Les connaissances acquises ont été organisées, ce qui facilitent leur transmission et leur réutilisation (phase d'ingénierie). Par exemple, des ouvrages comme "Compilateurs : principes, techniques et outils" [AhoSU89] permettent à un ingénieur logiciel "standard" de choisir des solutions adaptées à la résolution d'un problème pratique, tout en respectant des contraintes données. C'est justement l'un des objectifs d'une discipline d'ingénierie (Section I.2.2.1).

#### *Le cas de la programmation globale...*

Dans le cas de la programmation globale, la situation est tout autre. L'un des problèmes principaux est que trop souvent on a tendance à se concentrer sur des aspects pragmatiques et syntaxiques plutôt qu'abstraits et sémantiques.

1. Dans les années 70, le DoD a pris conscience de la perte d'énergie due à l'hétérogénéité des langages utilisés par ses services. Il s'est donné pour mission de répertorier les différents langages utilisés et d'en choisir un seul pour les développements futurs. Il est ressortit que 470 langages étaient utilisés mais qu'aucun n'était satisfaisant! Le langage Ada a été conçu pour résoudre ce problème.

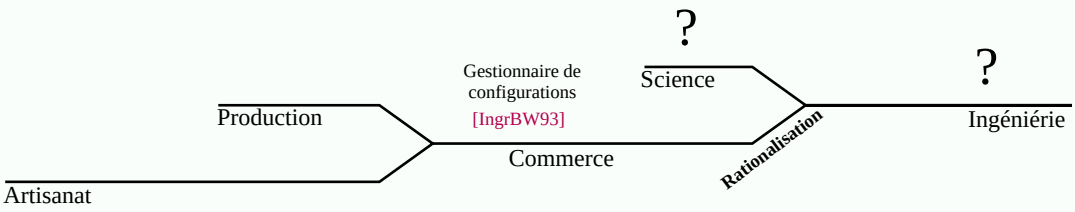
*figure 22* Maturité de la programmation détaillée




Ce phénomène est à l'origine normal car il s'agit en premier lieu de résoudre les problèmes pratiques rencontrés lors de la production de logiciels de grande taille. De ce point de vue il semble que le niveau de maturité de la programmation globale soit comparable à celle de la programmation détaillée dans les années 60. Un nombre croissant de systèmes sont proposés et commencent à avoir impact réel sur l'industrie. La phase "commerciale" fait peu à peu place à la phase "artisanale". Très peu de concepts se sont dégagés et l'utilisation de techniques formelles est encore très réduite. Il est difficile de comparer avec précision les différents systèmes et plus encore de trouver des équivalences entre eux.

A notre connaissance, il existe d'ailleurs très peu d'études comparatives basées sur des concepts précis. Les comparaisons existantes se limitent souvent à des critères fonctionnels plutot flous. Par exemple le rapport Ovum [IngrBW93] compare plus de 13 systèmes en fonction de leur "facilité" d'utilisation ou d'adaptation, de la "qualité" du support offert pour la gestion de versions, etc. Ces comparaisons bien qu'intéressantes d'un point de vue commercial ne donne pourtant pas d'indications précises sur les équivalences entre systèmes, les concepts sur lesquels ils sont basés, etc. Si l'on fait une transposition dans le domaine de la programmation détaillée, ceci correspond grossièrement à indiquer le type d'applications visées par les langages Fortran ou Cobol ou à comparer la puissance des opérations d'entrées sorties.

*figure 23* Maturité de la programmation globale




Alors vient une question : créer de nouveaux systèmes est certes important, mais *ne faudrait-il pas aussi favoriser un processus de rationalisation dans le domaine de la programmation globale?* Si l'on considère le domaine de la programmation détaillée, plus de progrès ont-ils été faits en proposant de nouveaux langages de programmation ou en essayant de décrire formellement leur sémantique ?

Dans cette thèse, nous proposons une modeste contribution en faveur d'une rationalisation (O3).

### I.6.2 Ré-ingénierie + programmation globale = ré-ingénierie globale

Abordons maintenant le deuxième point, c'est à dire celui correspondant à l'objectif O4.

L'une des originalités de cette thèse est d'aborder simultanément le thème de la programmation globale et celui de la ré-ingénierie. L'intersection de ces deux thèmes peut tout simplement être baptisée *ré-ingénierie globale* ! Autrement dit nous proposons d'utiliser la distinction entre programmation détaillée et programmation globale pour étendre la taxonomie de la ré-ingénierie [Favr94a].

#### *Re-ingénierie détaillée vs. ré-ingénierie globale...*[1]

La ***ré-ingénierie détaillée*** regroupe l'ensemble des activités de ré-ingénierie se focalisant sur les notions de programmation détaillée, i.e. sur les algorithmes et/ou les structures de données.

La ***ré-ingénierie globale*** regroupe l'ensemble des activités de ré-ingénierie se focalisant sur les notions de programmation globale, i.e sur l'architecture, la manufacture, l'évolution et/ou la variation.

Ces définitions sont basées sur des notions et non pas sur des techniques particulières.

#### *Intérêt des définitions...*

Introduire de nouveaux termes dans un domaine où les problèmes de taxonomies sont déjà si importants doit pour le moins être justifié. L'un des objectifs de cette thèse est justement de montrer que la distinction que nous proposons a un sens et qu'elle correspond à une réalité.

Notons tout d'abord que cette distinction a le mérite d'être simple et intuitive une fois la distinction entre programmation détaillée et programmation globale acceptée. Malgré tout, celle-ci n'est valable que si en pratique des différences existent effectivement. Sans rentrer dans les détails technologiques (étudiés dans les autres chapitres), remarquons qu'a priori :

- Des niveaux de granularité différents sont manipulés.
- Les activités de ré-ingénierie globale sont basées sur une connaissance incomplète du logiciel et par conséquent celui-ci n'est généralement pas modifié.
- La technologie de ré-ingénierie globale n'est pas nécessairement la même que celle de ré-ingénierie détaillée (par exemple stocker et manipuler un graphe d'appel et un arbre abstrait sont deux choses bien différentes).
- Certains outils se concentrent sur la ré-ingénierie globale alors que d'autres s'intéressent à la ré-ingénierie détaillée.

L'avantage principal de ces termes est d'affirmer l'importance de la programmation globale dans le domaine de la ré-ingénierie pour éviter que celle-ci ne soit une fois de plus occultée par la programmation détaillée.

1. Ce qui en anglais donne la distinction "Reengineering-in-the-large vs. Reengineering-in-the-small".

### *Une taxonomie plus précise...*

Bien entendu, les taxonomies utilisées respectivement pour la programmation globale et pour la ré-ingénierie peuvent être “croisées”. Par exemple on peut parler de restructuration de l’architecture (qui correspond grossièrement au terme (re)modularisation) ou de compréhension de l’évolution. Il n’est sans doute pas nécessaire d’introduire de nouveaux termes pour chaque intersection[1]. Par contre, dans le cadre de cette thèse nous utiliserons le terme ***compréhension de familles de programmes*** (par opposition au terme compréhension de programmes) lorsque l’on s’intéressera aux aspects de la programmation globale plus qu’aux algorithmes et aux structures de données.
Sans rentrer en détails, donnons une vue d’ensemble de ce que recouvre la ré-ingénierie globale.

- *Ré-ingénierie et architecture*. Ce thème est sans doute le plus étudié actuellement. Il s’agit typiquement de faciliter la compréhension de l’architecture des logiciels et éventuellement de modifier celle-ci. Cet aspect est particulièrement utile pour lutter contre la déstructuration naturelle du logiciel et pour essayer d’améliorer l’architecture de systèmes âgés.
- *Ré-ingénierie et manufacture*. Quasi inexploré, ce thème ne correspond pas à un problème majeur tout au moins actuellement. Comprendre le processus de manufacture d’un logiciel complexe peut tout de même présenter certaines difficultés et une aide peut être nécessaire. Changer de technologie de manufacture est l’autre aspect important.
- *Ré-ingénierie et variation*. Comprendre une variante d’un logiciel isolément est une chose, comprendre un ensemble de variantes enchevêtrées en est une autre... C’est à ce dernier problème que sont confrontés les chargés de maintenance dans le cas d’une maintenance multiple, notamment pendant et après les activités de portage. Celles-ci se font dans des conditions difficiles et sous la pression les “rapiéçages” sont monnaie courante. Postérieurement “refaire” avec calme devrait être la règle. En pratique le temps passe et les connaissances expliquant l’existence des différentes variantes et leur représentation disparaît. La ré-ingénierie prend alors tout son sens, d’une part pour faciliter la compréhension, d’autre part pour proposer des restructurations. Changer de technologie est un autre objectif possible. L’intersection entre la ré-ingénierie et la variation est quasiment inexplorée. C’est dans cette direction que s’oriente le reste de cette thèse.
- *Ré-ingénierie et évolution*. Puisque l’évolution est liée au temps qui passe, l’idée de ré-ingénierie peut sembler incompatible : ce qui est passé est passé ; il n’est pas question de changer le passé a posteriori. En fait, le centre d’intérêt ici sont les artefacts résultant de l’évolution du logiciel, c’est à dire grossièrement la “trace” de cette évolution. Dans ce contexte l’idée de ré-ingénierie correspond à faciliter la compréhension de cette trace, voir de restructurer celle-ci.

Pour affiner la notion de ré-ingénierie globale (Objectif O4) il faut décrire la technologie de la programmation globale et plus particulièrement la différence entre l’état de l’art et l’état de la pratique. C’est ce qui sera fait dans le Chapitre III. Mais tout d’abord il faut disposer d’abstractions. C’est ce que l’on cherche dans le Chapitre II.

1. Les adjectifs “globale” ou “détaillée” peuvent être utilisés lorsque nécessaires. Par exemple on pourrait parler de restructuration globale et de restructuration détaillée. En anglais on utilisera naturellement les suffixes “-in-the-large” et “-in-the-small”.

# Chapitre II

# Un modèle abstrait pour la programmation globale

# INDEX

# P



# R



# S



# T



# V

# LISTE DES FIGURES

# CHAPITRE II

# Un modèle abstrait pour la programmation globale

## II.1 Introduction

La programmation globale est au coeur de cette thèse. Dans le chapitre précédant les principaux concepts sous-jacents ont été présentés, mais sans considérer les aspects technologiques.

Nous avons présenté aussi l'intérêt des objectifs O3 et O4 consistant respectivement à

- (1) favoriser une rationalisation de la programmation globale et
- (2) à explorer le thème de la ré-ingénierie globale.

Dans le domaine de la programmation détaillée, de tels objectifs ont déjà été atteint (au moins partiellement). Ici il s'agit de le faire dans un domaine bien plus flou...

***Abstraire est indispensable*** :

- (1) Pour rationaliser[1] il faut "comprendre", utiliser autant que possible des théories et des techniques formelles, définir des *abstractions* (voir le modèle hélicoïdal et plus particulièrement la figure 8 (p.25)).
- (2) La ré-ingénierie d'un logiciel n'est possible que si l'on est capable de faire *abstraction* de sa représentation concrète (voir la figure 19 (p.62)).

Alors que les travaux concernant la programmation globale sont généralement tournés vers des aspects pragmatiques et concrets, ici une démarche inverse est prise : *un modèle abstrait est présenté.* En comparaison avec les approches classiques ce modèle pourrait être qualifié d'*abstrait*, de *minutieux* et d'*incomplet.*

- "*Abstrait*". Le modèle proposé est si abstrait qu'il ne fait d'ailleurs que rarement référence à la programmation globale et au logiciel... En fait il pourrait sans doute être appliqué à d'autres domaines, comme par exemple à la gestion de documents structurés. Utiliser des abstractions permet de réutiliser des concepts issus d'autres contextes[2]. Pour faciliter la lecture de ce chapitre, des exemples triviaux sont présentés. Ils n'utilisent que quelques chaînes de

1. Rationalisation : Organisation d'une activité, selon des principes rationnels d'efficacité, en soumettant tous ses éléments à une étude scientifique [Robe87].

caractères et n'ont rien à voir avec le logiciel (il s'agit de décomposer les mots "madame", "monsieur" et "mademoiselle"...).

- *"Minutieux"*. Les discussions qui vont suivre risquent de paraître parfois trop détaillées. C'est le prix à payer si l'on veut comparer une grande gamme de systèmes très proches sur certains aspects. Le fait qu'ils soient proches est d'ailleurs plutôt rassurant si l'on cherche à déterminer des transformations permettant de passer d'une représentation à une autre. Les "détails" sont alors plus importants qu'ils ne le paraissent. Par exemple, dans le domaine de la programmation détaillée, les travaux concernant la sémantique des langages de programmation sont basés sur des subtilités souvent fort utiles.
- *"Incomplet"*. La programmation globale est un domaine très large et il n'est pas possible de décrire en détail chaque aspect. Il est préférable de se concentrer sur des points particuliers aux détriments des autres (lors de la description de la sémantique des langages de programmation, on essaie aussi d'isoler les différents aspects : entrées sorties, exceptions, etc.). Ici nous nous intéresserons essentiellement aux problèmes de variations et d'évolution, et même plus précisément aux problèmes d'identification et de sélection de versions. Vu leurs fortes interactions, les aspects architecture et manufacture doivent aussi être pris en compte mais ce sont des thèmes périphériques dans cette thèse.
  Remarquons finalement que de toute façon le modèle proposé ne peut être complet : le domaine de la programmation globale est ponctué de différents problèmes pratiques difficiles à modéliser dans l'état actuel des connaissances. Bon nombre de ces problèmes n'apparaîtront donc pas explicitement dans ce modèle.

*C'est dans le chapitre suivant que seront décrits les aspects concrets de la programmation globale.* De nouvelles notions seront introduites. Certains concepts seront affinés.

Ce chapitre se décompose comme suit :

- **"Abstractions"**. Le modèle proposé est basé sur trois types d'abstractions principales : les objets structurés, les objets dérivés, et les objets générés (II.2). Nous nous intéressons surtout à cette dernière classe qui correspond aux problèmes de variations et d'évolution.
- **"Représentations"**. Les problèmes de représentations sont abordés (II.3). Le cas des objets génériques est traité plus en profondeur (II.4, II.5).
- **"Opérations"**. Différentes opérations sont présentées. On distingue les opérations concrètes des opérations abstraites (II.6).
- **"Un exemple"**. Un scénario permet d'illustrer les différentes notions présentées (II.7).

Tout au long de ce chapitre de nombreuses références sont faites aux annexes[1].

2. Cette démarche a été annoncée dans l'introduction : A2 "Etre aussi rigoureux et abstrait que possible.", A3 "Réutilisation : Réutiliser autant que possible." et A4 "Réutilisation : Rendre aussi réutilisable que possible.". Elle est compatible avec le modèle hélicoïdal qui suggère que s'éloigner d'un centre d'intérêt permet de faire des connexions avec d'autres thèmes.

*figure 24* Architecture logique du Chapitre II.

| Abstractions | Représentations | Opérations |
|---|---|---|
| Objets structurés | Objets structurés | Objets structurés |
| Objets dérivés | Objets dérivés | Objets dérivés |
| Objets générés | Objets générés | Objets générés |

**Exemple**

*figure 25* Répartitions des concepts

| Abstractions | Représentations | Opérations |
|---|---|---|
| *objet structuré*<br>*objet construit*<br>*objet composite*<br>*constructeur*<br>*référence*<br>*invariant*<br>*P-correction*<br><br>*objet source*<br>*objet dérivé*<br>*opérateur*<br><br>*contexte*<br>*choix*<br>*variation*<br>*variante*<br>*objet général*<br>*objet générique*<br>*G-P-correction* | *représentation en extension*<br>*représentation en compréhension*<br><br>*mécanisme de substitution*<br>*mécanisme de sélection*<br><br>*objet paramétré*<br>*objet générique en extension*<br>*objet générique fonction*<br>*objet générique ensemble*<br><br>*paramètre*<br>*expression de sélection*<br>*sélecteur*<br><br>*espace à n-dimension*<br>*estampille*<br>*fragment*<br>*structure de contrôle*<br>*variation détaillée*<br>*variation globale*<br>*synthèse des variantes*<br>*héritage des choix*<br><br>*domaine composé*<br>*domaine globale*<br>*domaine universel* | *opération abstraite*<br>*opération concrète*<br><br>*inclure*<br>*extraire*<br><br>*incrémentalité*<br>*découpage*<br>*mémorisation*<br><br>*génération de variantes*<br>*extension d'objets génériques*<br>*spécialisation d'objets génériques* |

1. Une utilisation extensive est faite des notions d'ensembles, de relations et de fonctions. Les relations sont vues comme des ensembles particuliers et les fonctions comme des relations particulières. On utilise des concepts provenant autant des langages de spécifications ensemblistes (comme Z ou VDM) que des modèles de données relationnels (imbriqués ou non). Tout au long de ce chapitre de nombreuses références sont donc faites à l'Annexe A.1 ("Des ensembles aux fonctions") qui rassemble et présente succinctement ces notions. Nous nous intéressons non seulement à des entités abstraites mais aussi à leur représentation. Une *fonction* est un objet mathématique abstrait, un *programme* est une représentation informatique d'une fonction. Ces notions sont présentées dans l'Annexe A.2 ("Des fonctions aux programmes"). Finalement l'Annexe A.3 ("Concepts et techniques relatifs aux programmes") présente un certain nombre de concepts liés à la notion de langage, d'interpréteur, etc. Ceux-ci proviennent de la programmation détaillée et vont être appliqués à la programmation globale.

# II.2 Abstractions

Dans cette section les aspects fondamentaux du modèle sont présentés d'un point de vue abstrait.

## II.2.1 Objets atomiques

Un ***objet atomique*** est un objet dont la valeur est atomique, c'est à dire dont la valeur ne peut pas être décomposée[1]. Les valeurs atomiques appartiennent à des ***domaines de base*** choisis en fonction de l'application (entiers, booléens, chaînes de caractères, etc.).

*figure 26* Objets et valeurs atomiques


***(a)*** **Valeurs atomiques** ***(b)*** **Objets atomiques**

## II.2.2 Trois grandes classes

Le modèle abstrait est organisé autour de trois thèmes : les *objets structurés*, les *objets dérivés* et les *objets générés*. Ces différents aspects peuvent être vus comme trois dimensions orthogonales (représentées dans l'espace dans la figure 27.a et dans le plan dans la figure 27.b).

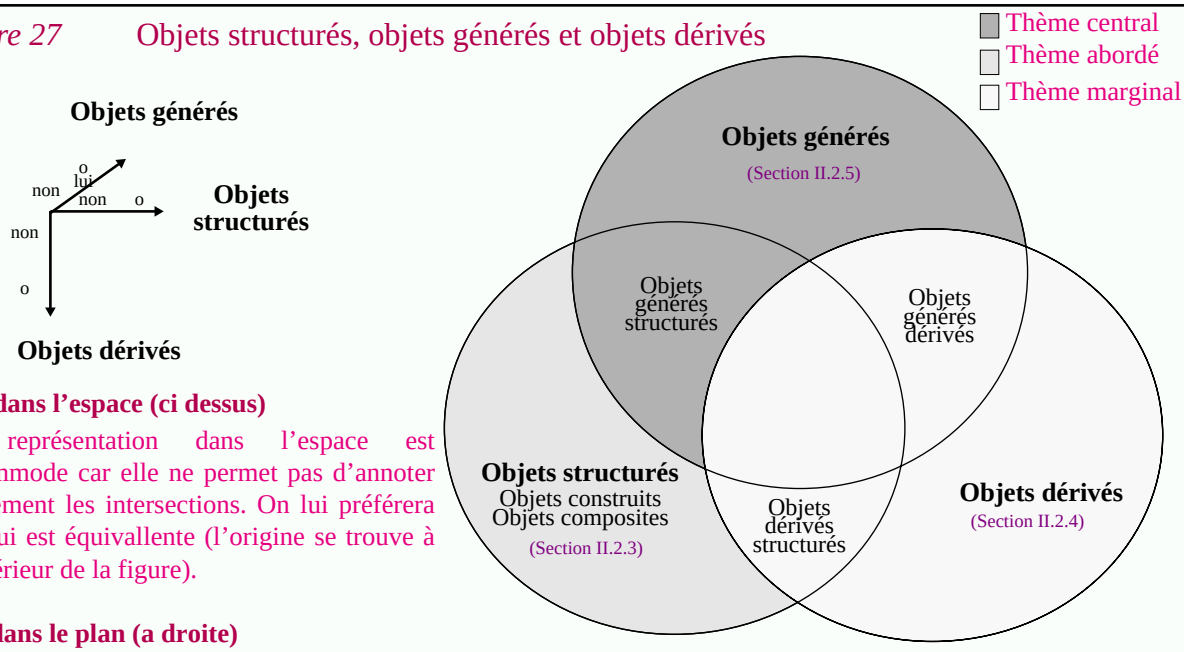
Nous verrons qu'en ce qui concerne la programmation globale, les objets structurés et les objets dérivés correspondent respectivement aux aspects architecture et manufacture ; les objets générés correspondent à la variation et à l'évolution (tableau 2). Comme il l'a été dit, ce chapitre est particulièrement abstrait. Malgré les exemples triviaux présentés tout au long de celui-ci, le lecteur s'intéressant surtout à la technologie du logiciel pourra utilement comparer la figure 27 à la figure 77 (p.149), consulter la figure 85 (p.162) et feuilleter le Chapitre III pour avoir une idée plus concrète de la manière dont vont être appliqués les concepts présentés ci-dessous.

TABLEAU 2 Relation simplifiée entre le modèle abstrait et la programmation globale

| Chapitre II | Modèle abstrait | Programmation globale | Chapitre III |
|---|---|---|---|
| Section II.2.3 | Objets structurés | Architecture | Section III.2 |
| Section II.2.4 | Objets dérivés | Manufacture | Section III.3 |
| Section II.2.5 | Objets générés | Variation et évolution | Section III.3 |

1. Dans cette thèse le terme "objet" est utilisé de manière très libre ; il ne fait pas référence aux modèles orientés objets. En particulier nous nous intéressons essentiellement à la structure de ces objets sans pour autant leur associer des méthodes. Ici un ***objet*** est une valeur identifiée, plus précisément un couple (identité, valeur). On aurait aussi pu parler d'entité. Nous avons préféré le terme objet dans la mesure où en génie logiciel on utilise fréquemment les termes objets sources, objets dérivés, etc... Bien souvent on aura tendance à ne pas faire la différence entre une valeur et un objet. C'est bien souvent le cas en génie logiciel. Par exemple on parlera d'un programme parfois en faisant référence à un objet parfois en faisant référence à une valeur. Presque tout ce qui est dit dans la suite s'applique autant à des valeurs qu'à des objets.

*figure 27* Objets structurés, objets générés et objets dérivés




***(a)* dans l'espace (ci dessus)**

La représentation dans l'espace est incommode car elle ne permet pas d'annoter facilement les intersections. On lui préférera (b) qui est équivallente (l'origine se trouve à l'extérieur de la figure).

***(b)* dans le plan (a droite)**

Ce schéma sera utilisé et raffiné tout au long de ce chapitre (figure 28 (p.88), figure 29 (p.89), figure 27 (p.87), figure 32 (p.92)). Chaque partie va être abordée successivement en entrant plus ou moins dans les détails.

## II.2.3 Objets structurés, constructeurs et références

La notion d'***objets structurés*** est liée aux problèmes de structuration. Pour résoudre ces problèmes deux techniques complémentaires peuvent être utilisées : (1) les constructeurs et (2) les références. Les constructeurs donnent lieu à la notion d'objet construit alors que les références donnent lieu à la notion d'objet composite. L'adjectif "structuré" sera utilisé pour désigner l'une ou l'autre de ces solutions. Par opposition on parlera d'*objet simple*.

### II.2.3.1 *Valeurs construites et constructeurs*

Une ***valeur construite*** est un élément d'un ***domaine construit***, c'est à dire d'un domaine défini à partir d'un ou plusieurs ***constructeurs***. Ici nous considérons les constructeurs d'ensembles, de listes, de tuples et de fonctions[1]. Ces constructeurs sont voisins de ceux définis en VDM [Jone90] ou Z [Word92] mais les notations diffèrent quelque peu. Le tableau 3 présente quelques exemples[2]. Seules des structures arborescentes peuvent être définies à partir de constructeurs.

TABLEAU 3 Constructeurs

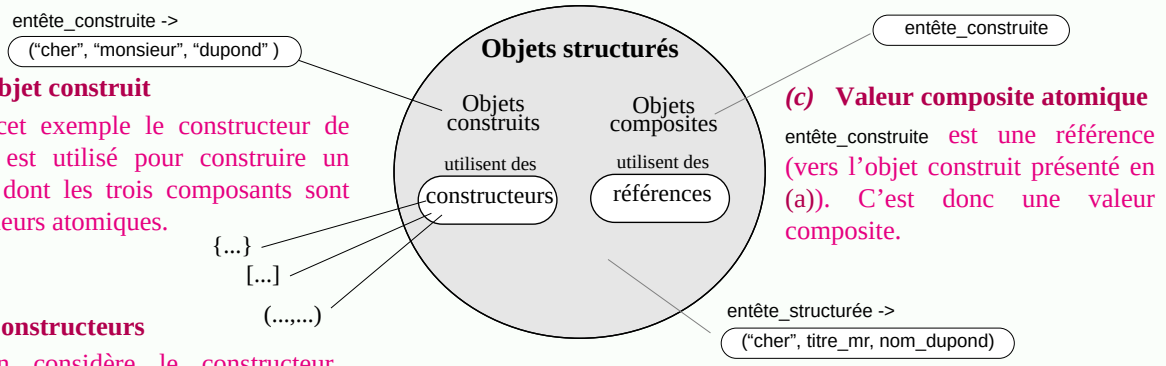
| Nom | Constructeur de types | Exemple de valeurs | Annexe |
|---|---|---|---|
| Ensemble | {Int} | {3;2;4;5} | A.1.1 |
| Liste | [Int] | [3;4;2;2;2;4] | A.1.9 |
| Tuple | (Int x Str) | (10,"dupond") | A.1.4 |
| Enregistrement | (numsecu->Int, nom->Str) | (numsecu->10, nom->"dupond" ) | A.1.4 |
| Fonction | Int -> Int | {2->3; 4->3; 5->10} | A.1.7 |

1. Dans la suite de ce document on utilisera la λ-notation pour dénoter des fonctions. Par exemple λx•x+1 dénote la fonction traditionnellement définie par f(x) = x+1.

### II.2.3.2 *Valeurs composites et références*

La notion de ***valeur composite*** correspond à l'utilisation de ***références***. Au sens large, une référence est une valeur d'un domaine particulier dont l'interprétation correspond à un "pointeur" vers une valeur. Le but est de pouvoir partager des informations et de décrire des structures pouvant être interprétées comme des graphes. Une valeur composite n'a de sens que dans un *environnement* donné. Un environnement est un ensemble de couples (identificateur, valeur).

*figure 28* Constructeurs et références ; Objets structurés




***(a)*** **Objet construit**
Dans cet exemple le constructeur de tuples est utilisé pour construire un triplet dont les trois composants sont des valeurs atomiques.

***(b)*** **Constructeurs**
Ici on considère le constructeur d'ensembles, de listes et de tuples respectivement.

***(c)*** **Valeur composite atomique**
entête_construite est une référence (vers l'objet construit présenté en (a)). C'est donc une valeur composite.

***(d)*** **Objet construit et composite**
Cet objet est à la fois construit et composite car d'une part il contient une référence et d'autre part c'est un couple.

Sans approfondir, notons qu'à partir de la notion de référence peut venir la notion de relation, que les références peuvent être "typées" et qu'une sémantique particulière peut leur être associée. L'ensemble des références peut former un graphe. Il peut être important de contrôler la complexité de ce graphe, par exemple en construisant une partition de l'ensemble des objets et en interdisant les relations entre objets de partitions différentes (nous verrons que ce genre de technique est utilisé dans le cadre de la programmation globale).

### II.2.3.3 *Invariants (contraintes d'intégrité)*

Définir un domaine à partir de constructeurs ou de références n'est pas toujours suffisant. Parfois il est nécessaire de définir un ensemble d'objets par une propriété que doit vérifier ses éléments. Schématiquement dans le domaine des langages de spécifications on parle d'***invariants*** alors que dans le domaine des bases de données on parle de ***contraintes d'intégrité***. D'un point de vue pratique il n'est pas toujours possible d'assurer toutes les propriétés. Pour une propriété donnée P, on dira qu'un objet est ***P-correct*** s'il vérifie P. Par exemple "monsieur" est orthographiquement-correct.

### II.2.3.4 *Discussion*

En pratique l'utilisation de constructeurs et de références se fait simultanément car elle correspond au même besoin de représenter des données structurées. Ceci justifie le fait de regrouper ces deux notions et d'utiliser le terme "valeur structurée" (objet structuré)[1].

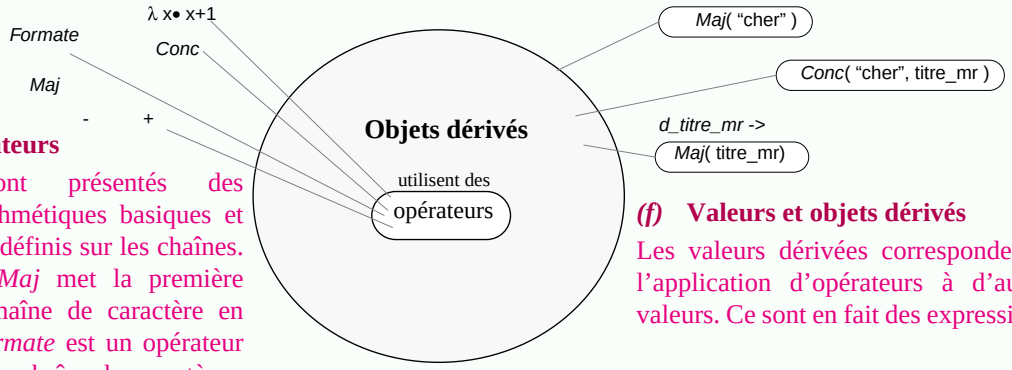
2. Ces constructeurs sont définis plus précisément en Annexe A.1 ("Des ensembles aux fonctions").

1. Le lecteur désireux de connaître directement l'application de ces concepts dans le domaine de la programmation globale pourra se reporter à la Section "Architecture" (p. 157).

## II.2.4 Objets dérivés et opérateurs

Un ***objet dérivé*** est un objet dont la valeur est une valeur dérivée. Alors qu'une valeur construite est définie à partir d'un constructeur, une ***valeur dérivée*** est définie à partir de l'application d'un ***opérateur***[1] à une valeur. On parlera aussi de "valeur calculée". Par opposition on parlera de ***valeur source***[2]. Une valeur dérivée est définie par une expression.

*figure 29* Opérateurs ; Objets dérivés




***(e)*** **Des opérateurs**

Ci-dessus sont présentés des opérateurs arithmétiques basiques et des opérateurs définis sur les chaînes. Par exemple *Maj* met la première lettre d'une chaîne de caractère en majuscule. *Formate* est un opérateur "formatant" une chaîne de caractères. *Conc* est l'opération de concaténation.

***(f)*** **Valeurs et objets dérivés**

Les valeurs dérivées correspondent à l'application d'opérateurs à d'autres valeurs. Ce sont en fait des expressions.

NOTE: Dans la suite de ce document les références à des objets dérivés ou à des opérateurs seront notées en *italiques*.

Un opérateur est une fonction. Le domaine et le codomaine des opérateurs peuvent être structurés. Autrement dit un opérateur peut prendre en entrée plusieurs paramètres (ou un paramètre structuré) et retourner un résultat structuré (donc en particulier retourner plusieurs résultats).

Si le domaine et le codomaine sont différents il peut être intéressant dans certains cas de distinguer alors les ***domaines sources*** et les ***domaines dérivés***.

Remarquons qu'un opérateur peut lui aussi être vu comme une valeur. On a alors des ***opérateurs atomiques***, des ***opérateurs construits***, des ***opérateurs composites***, et par la suite des ***opérateurs structurés***. Bien évidemment les constructeurs ne sont pas les mêmes que ceux présentés dans la section antérieure pour les objets structurés : puisque les opérateurs sont des fonctions il s'agit d'opérations définies sur les fonctions (e.g. la composition de fonctions (o) ou l'application parallèle (#). Voir l'Annexe A.1.7 ("Fonctions")).

Dans les exemples présentés ci-dessus *Formate*, *Conc*, *Maj* sont en fait des références à des opérateurs et non pas des opérateurs basiques. *Formate o Maj* est un opérateur structuré[3].

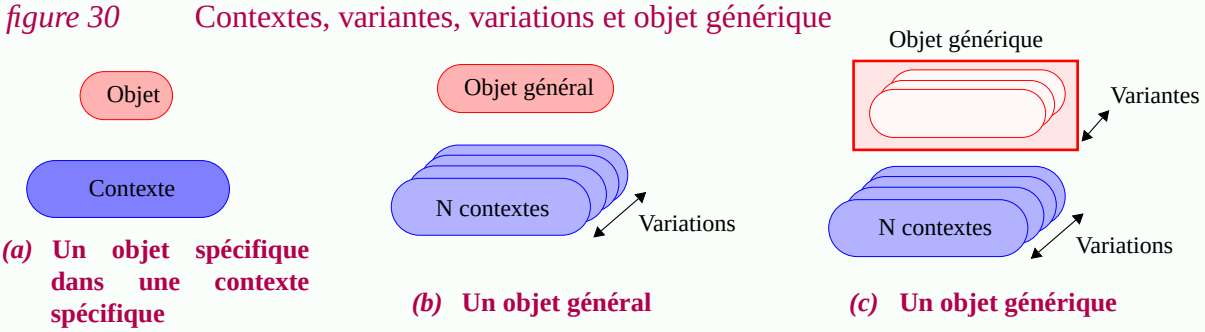
1. Dans ce document le terme "opérateur" est utilisé pour désigner ce concept précis. Par contre le mot opération est utilisé dans son sens général.
2. Ce terme a été choisi pour rester cohérent avec la terminologie objet source et objet dérivé souvent utilisée dans le domaine de la programmation globale.
3. C'est dans la Section III.3, ("Manufacture") que ces concepts sont appliqués dans le domaine de la programmation globale. Schématiquement les opérateurs atomiques correspondent aux outils de dérivation (compilateur, éditeur de liens, etc.). Les opérateurs structurés correspondent à l'enchaînement d'outils de dérivation (par exemple sous la forme de fichiers de commandes). Les objets dérivés sont désignés de la même manière.

## II.2.5 Objets générés et objets génériques

L'utilisation d'un objet se fait toujours dans un ***contexte*** donné (figure 30.a)[1]. La variété des contextes existants rend nécessaire de pouvoir (ré)utiliser l'objet aussi souvent que possible et d'adapter celui-ci à de telles ***variations***. A défaut d'un ***objet général*** (figure 30.b), c'est à dire d'une solution unique et générale, il s'agit de proposer différentes ***variantes*** de cet objet.

Remarquons l'emploi du terme "variation" qui correspond au problème venant d'un phénomène extérieur et celui du terme "variante" pour désigner une solution à ce problème. Autrement dit on crée des variantes pour répondre aux variations du contexte.

*figure 30* Contextes, variantes, variations et objet générique




***(a)*** **Un objet spécifique dans une contexte spécifique**

***(b)*** **Un objet général**

***(c)*** **Un objet générique**

D'un point de vue abstrait l'ensemble de ces variantes est regroupé dans ce que l'on appellera un ***objet générique***[2] (figure 30.c).
Dans "objet générique", le qualificatif "générique" vient du fait qu'il est possible de ***générer*** une variante à partir d'un ***choix*** particulier. Dans cette thèse le terme ***objet généré*** sera synonyme de variante[3]. Par opposition aux objets génériques on parlera d'***objets spécifiques***.

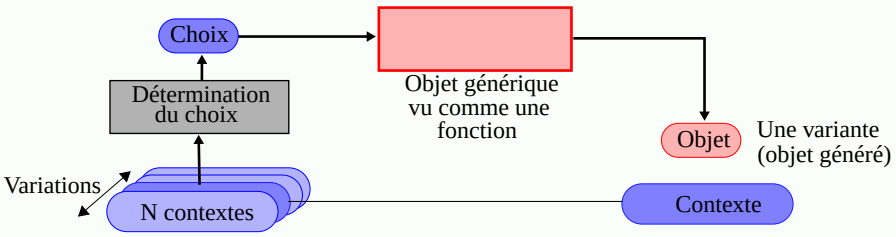
***D'un point de vue abstrait un objet générique peut être vu comme une fonction***

Nous allons voir que ce postulat constitue l'une des pierres angulaires de cette thèse. Elle va en effet permettre d'utiliser la théorie ensembliste dans le cadre des problèmes de variations, de rapprocher des concepts de bases de données et de ceux de génie logiciel, et finalement de réutiliser des concepts et techniques de programmation détaillée dans le domaine de la programmation globale.

1. Soulignons que contrairement au terme "environnement", dans cette thèse le terme "contexte" est pris au sens large du français et correspond à un ensemble d'informations et/ou de contraintes n'étant pas nécessairement formalisées.
   Rappelons qu'au contraire le terme "environnement" a été défini comme étant une structure formelle ayant une représentation informatique : une fonction d'un ensemble d'identificateur vers une valeur.
2. Le terme "objet générique" est utilisé par certains auteurs dans le contexte de modèles orientés objets (voir par exemple [Scio94]). Le concept qu'il désigne est voisin de celui présenté ici. Rappelons cependant que nous ne faisons pas d'hypothèse sur le modèle de données employé.
3. Le terme "variante" est utilisé de manière libre, qu'il s'agisse d'une valeur ou d'un objet.

La raison principale expliquant le choix de cette modélisation correspond à voir *la génération d'une variante comme l'application d'une fonction* (figure 30).

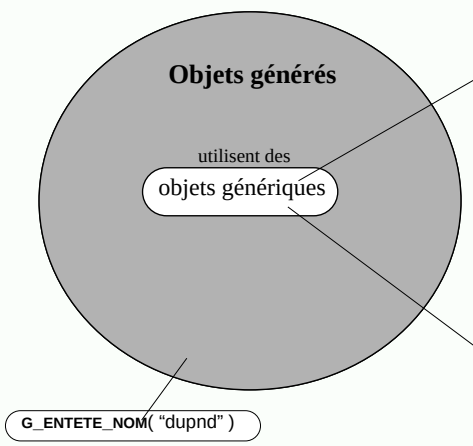
*figure 30* Un objet générique vu comme une fonction




Il sera montré dans la Section II.6, ("Opérations") que presque toutes les opérations traditionnellement définies sur les fonctions peuvent être interprétées dans le cas des objets génériques (ce qui prouve la pertinence de la modélisation choisie pour les objets génériques).

Le ***domaine de l'objet générique*** (i.e. l'ensemble des choix possibles) détermine l'ensemble des contextes dans lequel celui-ci est utilisable. L'ensemble des variantes qu'il est possible de générer sera naturellement appelé ***codomaine de l'objet générique***. Ces désignations sont cohérentes avec le fait de voir un tel objet comme une fonction. Des exemples sont donnés dans la figure 31.

*figure 31* Objets génériques -> Objets générés (variantes)




{ 1 -> "monsieur" ;
2 -> "madame" ;
3 -> "mademoiselle" }

***(a)*** **Valeur générique définie en extension**

Cette fonction finie est représentée en extension. Son domaine (i.e. le domaine des choix) est {1,2,3} alors que son codomaine est un {"monsieur", "madame", "mademoiselle"}. Selon le contexte (i.e le genre de la personne et le fait qu'elle soit mariée ou non), la variante appropriée sera choisie.

G_ENTETE_NOM ->

λ nom . ["cher","monsieur",nom]

***(b)*** **Objet générique défini via un mécanisme de substitution**

Cet objet générique est une fonction identifiée. C'est le mécanisme de substitution qui est utilisé. Le domaine est l'ensemble des chaînes de caractères. Il s'agit d'une fonction infinie. Nous verrons par la suite que ce type d'objet générique sera appelé objet paramétré.

***(c)*** **Valeur générée (variante)**

Il s'agit d'une variante. Plus précisément c'est l'application du choix "dupnd" à l'objet générique **G_ENTETE_NOM** (application de fonctions)

NOTE: Dans la suite de ce document les références à des objets génériques seront écrites en **PETITES CAPITALES**.

Il est important de bien faire la distinction entre un objet générique et un objet général. Un objet général est un objet pouvant être utilisé tel quel dans différents contextes. Par exemple "Madame, Monsieur" est un objet général couramment utilisé dans une lettre. Au contraire un objet générique est utilisé pour générer une variante particulière. Cette variante est un objet utilisable dans un contexte donné. La phase de génération est indispensable.

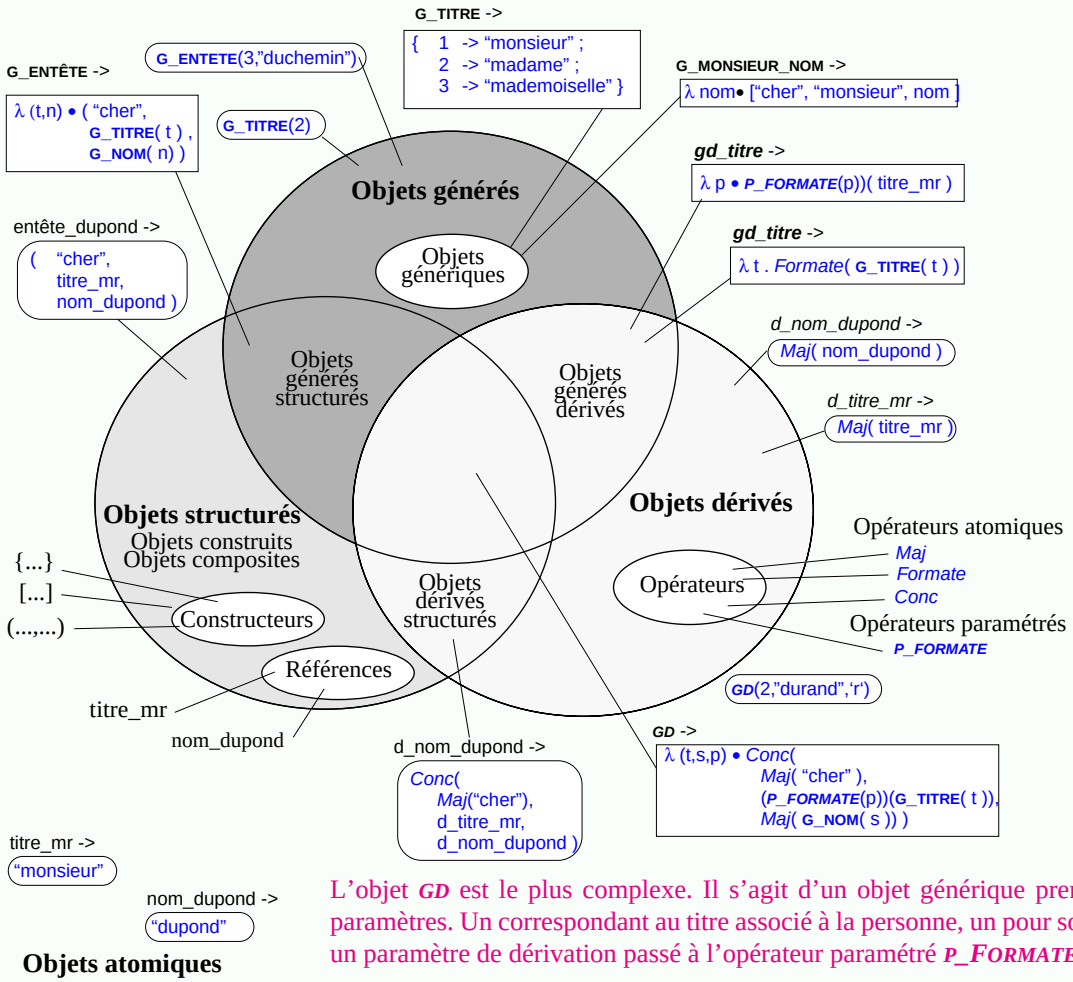
Finalement remarquons que même si un invariant *P* est défini sur le codomaine de l'objet générique, en pratique il n'est pas toujours possible d'assurer que toutes les variantes soient *P*-

correctes. Dans le cas ou cette condition est vérifiée on dira que l'objet générique est ***G-P-correct***. Par exemple la valeur générique (a) est G-orthographiquement-correcte. Ce n'est pas le cas de l'objet générique (b) car comme le montre l'exemple (c) des variantes orthographiquement incorrectes peuvent être générées. L'un des problèmes classiques rencontrés lors de la définition d'un objet générique est de pouvoir déterminer l'invariant qui doit être associé à son domaine pour que l'objet générique soit G-P-correct[1].

## II.2.6 Synthèse

Dans cette section nous avons présenté quelques abstractions utiles dans le domaine de la programmation globale : objets structurés, objets dérivés, objets générés. Les différentes combinaisons sont regroupées dans un même schéma (figure 32). Le lecteur pourra utilement comparer cette figure à la figure 77 (p.149). De nombreux points restent à décrire, notamment en ce qui concerne les objets générés et génériques. C'est ce qui est fait dans les sections suivantes.

*figure 32* Un exemple plus complet




L'objet ***GD*** est le plus complexe. Il s'agit d'un objet générique prenant trois paramètres. Un correspondant au titre associé à la personne, un pour son nom et un paramètre de dérivation passé à l'opérateur paramétré ***P_FORMATE***.

1. Pour avoir une vision plus concrète de ce que recouvrent les concepts présentés ci-dessus, le lecteur pourra survoler la Section "Variation : concepts" (p. 169) et la Section "Variation : approches" (p. 188). Grossièrement dans le domaine de la gestion de configurations un objet générique correspond à la notion de groupes de versions (Section I.4.3.1) ; la génération d'une variante correspond à la sélection d'une version ; un choix correspond à l'identification d'une version. Dans le domaine de la réutilisation on parlera plutot de composant générique, d'instanciation et de paramètres d'instanciation.

Le tableau 4, quand à lui, présente une synthèse des rapports entre les différents types objets.

TABLEAU 4 Objets atomiques, construits, composites, structurés, dérivés, générés

| Exemples | Un objet... | est défini à partir de... | et de... | Est opposé à |
|---|---|---|---|---|
| “dupond” | atomique | - | | structuré |
| [“mr.”;”dupond”] | construit | constructeur(s) | valeur(s) | |
| nom_dupond | composite | référence(s) | | |
| [“mr.”;nom_dupond] | structuré | constructeur(s) et/ou référence(s) | | |
| *Maj*( “dupond” ) | dérivé | opérateur(s) | valeurs(s) | source |
| **G_TITRE**( 3 ) | généré (variante) | objet générique | choix | spécifique |

D’un point de vue abstrait il y a une similitude entre les objets génériques et les opérateurs. Ce sont des fonctions. Dans le premier cas un objet généré est défini à partir d’un objet générique. Dans le deuxième cas un objet dérivé est défini à partir d’un opérateur. Malgré ces analogies, nous verrons qu’en pratique il est important de différencier ces deux concepts. Les opérateurs atomiques sont des données du problème alors que les objets génériques doivent être construits et maintenus tout comme n’importe quel autre objet[1].

1. D’un point de vue concret les opérateurs sont des outils de dérivation (par exemple un compilateur) alors que les objets génériques sont par exemple des groupes de versions.

# II.3 Représentations

Jusque là les différentes classes d'objets ont été considérées d'un point de vue *abstrait*, le but étant de pouvoir regrouper un grand nombre de systèmes existants (par exemple en ce qui concerne les objets génériques, nous verrons que le modèle proposé rassemble des systèmes allant des bases de données temporelles aux préprocesseurs, en passant par des gestionnaires de configurations!). En pratique des *représentations* très variées sont utilisées ; presque chaque système est basé sur une représentation particulière... Le but de cette section est de montrer que malgré tout, elles sont souvent très proches et qu'elles reposent sur un nombre limité de concepts. Il s'agit non seulement de déterminer quels sont les points communs et les différences mais aussi de déterminer s'il existe des transformations permettant de passer de l'une à l'autre. En réalité nous nous intéressons presque exclusivement à la représentation des objets génériques.

## II.3.1 Représentation des objets structurés

Les objets structurés sont définis à partir d'objets atomiques mais aussi de constructeurs et de références. La représentation des objets atomiques dépend de l'application considérée. Quant au problème de la représentation des objets structurés, elle se ramène en fait au problème de raffinement de données abstraites (problème typique des langages de spécifications comme VDM ou Z). Par exemple, un ensemble est une abstraction pouvant être représentée de multiples manières (liste, arbre binaire, etc.). Nous n'étudierons pas plus en détail ces aspects.

## II.3.2 Représentation des objets dérivés et des opérateurs

Les objets dérivés sont définis à partir d'opérateurs. Comme nous l'avons vu parmi ceux-ci on peut distinguer les opérateurs atomiques des opérateurs structurés. Bien que d'un point de vue abstrait ce soient des fonctions, ils doivent en pratique avoir une représentation informatique concrète.

***Puisque d'un point de vue abstrait un opérateur est une fonction sa représentation est un programme***

Ce raisonnement est en fait le même que celui tenu dans la section suivante à propos de la représentation des objets génériques (ce sont aussi des fonctions). Il n'est pas développé plus longuement ici car l'importance des objets dérivés et des opérateurs est moindre dans cette thèse.

## II.3.3 Représentation des objets générés et génériques

Les objets générés sont définis à partir d'objets génériques. Nous avons postulé que ces derniers étaient des fonctions. Représenter un objet générique revient donc à représenter une fonction. Les fonctions peuvent être définies en extension ou en intention (Annexe A.1.8, "Fonction en extension ou en compréhension"). Il en est de même pour leur représentation :

- *Représentation d'une fonction en extension*. C'est essentiellement dans le domaine des bases de données que ce thème a été étudié. Par exemple une fonction est un cas particulier de relation ; le modèle de données relationnel est justement basé sur la manipulation de l'extension de tels objets. Grossièrement il s'agit dans notre contexte de décrire des fonctions "tabulées". Remarquons aussi que les bases de données temporelles ont pour but de représenter et de manipuler des fonctions du temps.
- *Représentation d'une fonction en compréhension.* Dans ce cas ce sont plutôt des techniques de programmation détaillée qui sont utilisées. Ce cas est plus général que le premier et l'englobe car il est toujours possible de définir des fonctions en extension à partir d'un formalisme permettant de les définir en compréhension.

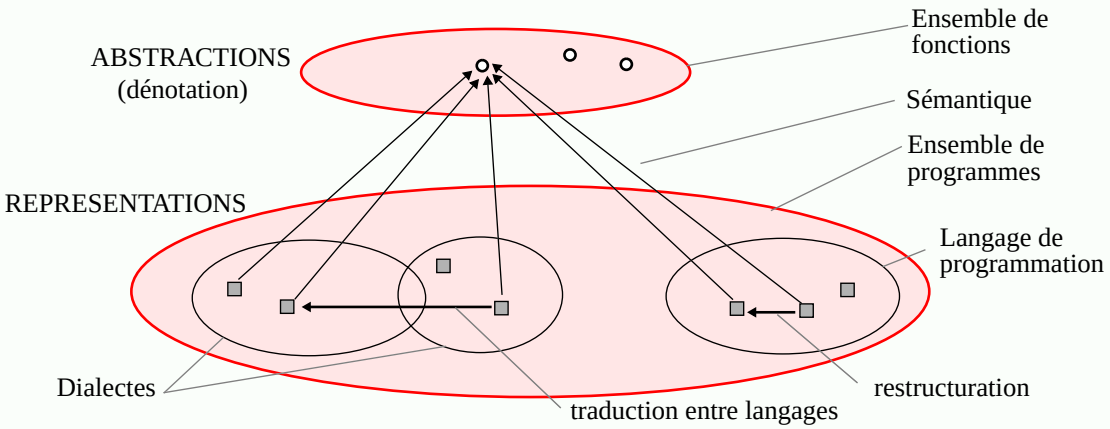
Ces remarques sont essentielles pour ce chapitre. Elles suggèrent que le problème de représentation auquel l'on est confronté met en jeu des notions et des techniques provenant de sources diverses. Dans cette thèse nous axerons la présentation surtout sur les concepts de programmation détaillée car ceux-ci sont plus nombreux et plus généraux. Les autres aspects ne seront pourtant pas négligés. C'est justement ce qui fait la richesse de ce chapitre.

***Du point de vue de la programmation détaillée...***

***Puisque d'un point de vue abstrait un objet générique est une fonction sa représentation est un programme***

Cette assertion correspond au thème développé dans l'Annexe A.2 ("Des fonctions aux programmes")[1] mais à la place d'être appliquée à la programmation détaillée, elle l'est à la programmation globale[2]. La figure 33, extraite de cette annexe, illustre cette idée.

*figure 33* Fonctions, programmes et sémantique




1. Une fonction est un objet mathématique abstrait. Un programme est la représentation informatique d'une fonction.
2. Par la suite lorsque l'on parlera d'objets génériques on utilisera les termes "représentations" et "programmes" de manière interchangeable ; il en sera de même pour "abstraction", "dénotation" et "fonction". Dans le cas où l'on voudra distinguer explicitement la représentation d'un objet générique d'un programme "standard" de programmation détaillée on utilisera respectivement les termes ***g-programme*** et ***p-programme*** (g pour générique et p pour programmation (détaillée)). On fera de même pour la représentation des opérateurs (les ***o-programmes***)

Le but d'une telle comparaison est bien évidemment de pouvoir réutiliser des concepts de programmation détaillée dans un contexte de programmation globale : la notion de sémantique dénotationnelle, de syntaxe, de transformation de programmes, de restructuration, etc.

#### *Objets génériques et langages...*

Comme le montre la figure 33 différentes représentations (programmes) correspondent à une même abstraction (à une même fonction). Ces représentations sont exprimées dans un langage. Un langage est caractérisé par un ensemble de règles déterminant, d'une part quels sont les programmes valides et, d'autre part, quelle est la sémantique associée à chaque programme (i.e. quelle est l'abstraction correspondante à chaque représentation).
Pour représenter une fonction, non seulement il y a le choix du langage mais, même pour un langage donné, plusieurs (voir une infinité de) programmes sont possibles.

En théorie un langage de programmation général pourrait être utilisé à cette fin. En pratique, ce n'est que rarement le cas : vue l'utilisation faite des objets génériques ce ne sont que des classes particulières de fonctions que l'on cherche à représenter. L'aspect algorithmique est souvent limité et dans bien des cas on se contente de langages proposant un nombre de constructions extrêmement réduit[1]. C'est d'ailleurs pour cela que des concepts de base de données peuvent être utilement rapprochés de ceux de programmation détaillée.

Comme dans le cas des langages de programmation détaillée, différents paradigmes peuvent être utilisés : langages impératifs, fonctionnels, déclaratifs (Annexe A.2.2, "Classes de langages de programmation"). Dans ce chapitre nous nous intéressons essentiellement au paradigme fonctionnel. CPP est un langage impératif et sera étudié dans le Chapitre IV. L'approche déclarative est aussi retenue en pratique (Chapitre III).

#### *Objets génériques et interpréteurs...*

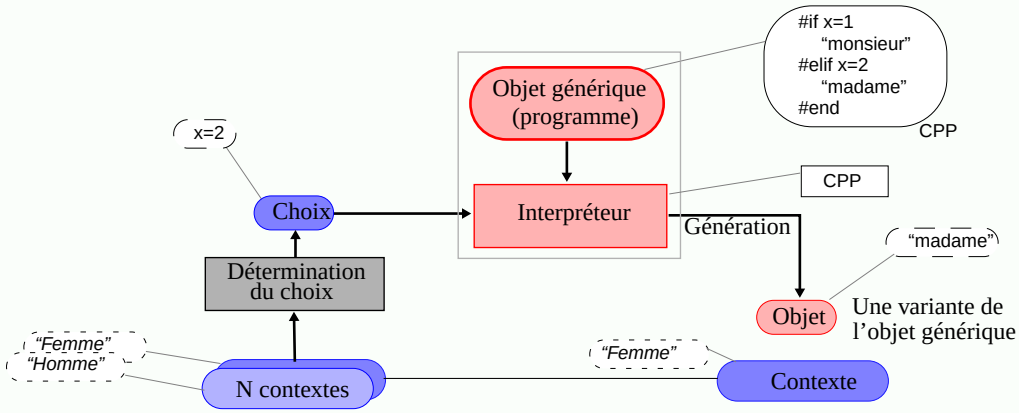
D'un point de vu concret un programme exprimé dans un langage n'a d'intérêt que si l'on dispose d'un interpréteur pour ce langage (Annexe A.3, "Concepts et techniques relatifs aux programmes"). Cette notion est présentée dans la figure 34.

Cette vision n'est en rien contradictoire avec celle présentée dans la figure 30 (p.91). Il s'agit simplement d'une vision moins abstraite. Elle correspond à une transformation communément faite lorsque l'on manipule des programmes et des interpréteurs (figure 143 (p.297)).

Etudier les représentations possibles pour les objets génériques revient donc à étudier les différents langages et leurs interpréteurs. Plutot que d'étudier ceux-ci un à un, il est préférable de faire ressortir les principaux concepts sur lesquels ceux-ci sont basés. C'est ce qui est fait dans les sections suivantes où l'on présente respectivement le *mécanisme de substitution* (typique de la programmation détaillée) et le *mécanisme de sélection* (on le retrouve autant dans le domaine de la programmation détaillée que dans le domaine des bases de données).

1. Le préprocesseur CPP est un cas relativement complexe et pourtant les constructions offertes par ce langage sont : l'affectation, la séquence d'instructions, l'instruction conditionnelle, l'appel de procédure et la substitution (Chapitre IV).

*figure 34* Un objet générique vu comme un programme interprété



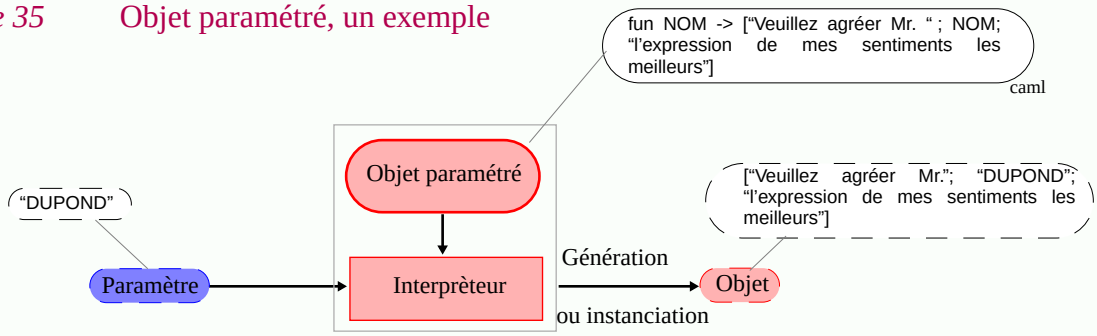

Dans cet exemple un texte source contenant des instructions CPP joue le rôle de la représentation d'un objet générique et est donc vu comme un programme. Une telle représentation n'a pas de sens sans l'interpréteur CPP. A partir d'un choix donné (ici sous la forme d'une valeur affectée à une macro) l'interpréteur génère le résultat (ici une chaîne de caractères).

## II.3.4 Objets paramétrés et mécanisme de substitution

Dans certains cas les variations entre les différents contextes peuvent s'exprimer par le changement d'un paramètre. Dans ce cas on parlera de ***paramètre*** plutôt que de choix et d'***objet paramétré*** plutôt que d'objet générique. On pourrait aussi parler d'***instanciation*** plutôt que de génération. Certains auteurs utilisent d'ailleurs le terme "valeur paramétrée" [GadiNair93]. C'est encore un autre terme pour dire "fonction", mais dans le contexte des problèmes de variation.

Le ***mécanisme de substitution*** est généralement à la base des objets paramétrés (figure 35). Il fait intervenir la notion de ***variable*** et d'***occurrence de variable***. Un ***mécanisme de liaison*** permet d'associer une valeur à une variable. La notion de ***portée*** intervient alors[1].

*figure 35* Objet paramétré, un exemple




Notons que le domaine et le codomaine d'un objet paramétré peuvent être infinis. C'est le cas dans l'exemple ci-dessus.

1. De telles notions sont à la base du lambda calcul. A titre d'exemple plus concret, les macro-processeurs proposent le mécanisme de macro-substitution. Les macros correspondent à la notion de variable. La définition de macro est un mécanisme de liaison. Leur portée est en général globale.

## II.3.5 Objets génériques en extension et mécanisme de sélection

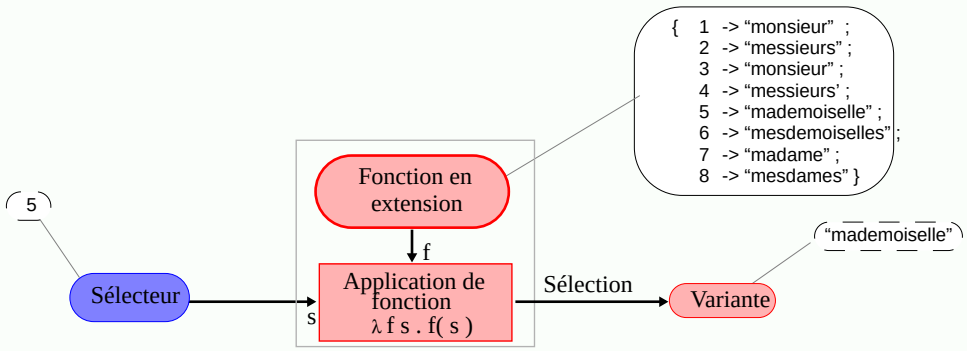
Un objet générique, tout comme une fonction, peut être représenté en extension (Annexe A.1.2, "Extension vs. compréhension: énumération vs. formulation"). Il s'agit alors d'énumérer toutes les variantes disponibles et d'utiliser un mécanisme de ***sélection*** plutôt que de substitution. On parlera alors d'***objet générique en extension***[1]. Dans ce cas on aura tendance à utiliser le terme ***sélecteur*** pour désigner le choix et le terme ***sélection*** pour désigner la génération d'une variante. Il n'est possible de représenter que des objets génériques dont le codomaine est fini.

Parmi les objets génériques en extension, on distingue les objets génériques fonctions et les objets génériques ensembles.

### II.3.5.1 *Objet générique fonction*

Un ***objet générique fonction*** est un objet générique représenté par une fonction décrite en extension[2] (Annexe A.1.8, "Fonction en extension ou en compréhension"). Les éléments du domaine et du codomaine sont représentés explicitement et il existe une dépendance fonctionnelle entre les éléments du domaine et du codomaine ; autrement dit à un choix donné correspond une seule variante. Dans l'exemple de la figure 36 le domaine est un sous-ensemble des entiers et le codomaine est un ensemble de chaînes de caractères.

*figure 36* Objet générique fonction ; représentation abstraite

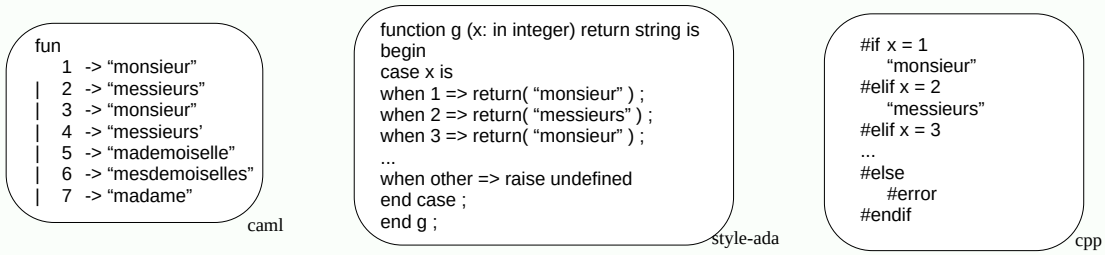



La fonction est définie en extension. La dépendance fonctionnelle entre le choix et la variante est respectée (à un choix donné il ne correspond qu'une seule variante).

D'un point de vue concret, cet objet générique fonction peut être représenté aussi bien sous la forme de programmes (figure 37.a) que de structures de données (figure 37.b).

1. Ce terme bien peu élégant a été préféré à "objet versionné" et à "groupes de versions". Ceux-ci font trop référence au domaine de la gestion de configurations alors que le concept que nous voulons désigner est également valide dans le domaine de la réutilisation ou dans le cas des bases de données temporelles. En fait le but est d'éviter d'utiliser la connotation particulière associée au terme "version".
2. On pourrait parler de "fonction tabulée" ou de "correspondance" (pour "map" en anglais).

*figure 37* Objet générique fonction ; représentations concrètes




***(a)*** **Représentations concrètes sous la forme de programmes**

En utilisant les concepts de programmation détaillée, différentes représentations concrètes sont possibles pour la représentation abstraite présentée dans la figure 36. Ci-dessus différents langages de programmation ont été utilisés pour représenter la fonction. Des aspects lexicaux et syntaxiques interviennent alors. Chaque représentation, (i.e. chaque programme) doit être interprétée avec un interpréteur adapté.

***(b)*** **Représentations concrètes sous la forme de structures de données**

Des structures de données classiques peuvent aussi servir de représentation concrète. Par exemple la fonction peut être représentée sous forme de table, de liste chaînée, d’arbre de recherche, etc. Dans ce cas ce sont les algorithmes des fonctions d’accès qui permettent d’*interpréter* ces structures.

### II.3.5.2 *Objet générique ensemble*

Un ***objet générique ensemble*** est un objet générique dont la représentation est un ensemble en extension. Il s’agit bien souvent d’une relation (i.e. les éléments de cet ensemble sont des tuples)[1]. Comme nous le verrons dans le Chapitre III, ce concept correspond plus ou moins à l’idée de “groupes de versions” dans le domaine de la gestion de configurations. En fait le problème est le même si ce n’est que l’on a affaire à une collection (i.e. à un ensemble) de variantes et qu’aucune dépendance fonctionnelle n’est assurée.

Le problème consiste à sélectionner une variante et ceci se fait à partir d’un sélecteur, que l’on pourrait appeler ***expression de sélection***. Généralement c’est un prédicat ou une construction à base de prédicats. Cette expression de sélection est utilisée comme un filtre et seuls les éléments vérifiant la propriété spécifiée sont retenus. L’interpréteur a donc pour rôle de calculer l’extension du prédicat sur l’ensemble spécifié. Dans le cas du modèle relationnel il s’agit de l’opération de sélection traditionnellement notée $\sigma$ (Annexe A.1.5, "Relations").

Puisque nous avons postulé que d’un point de vue abstrait un objet générique était une fonction, il ne doit retourner qu’un seul élément (i.e. une seule variante).

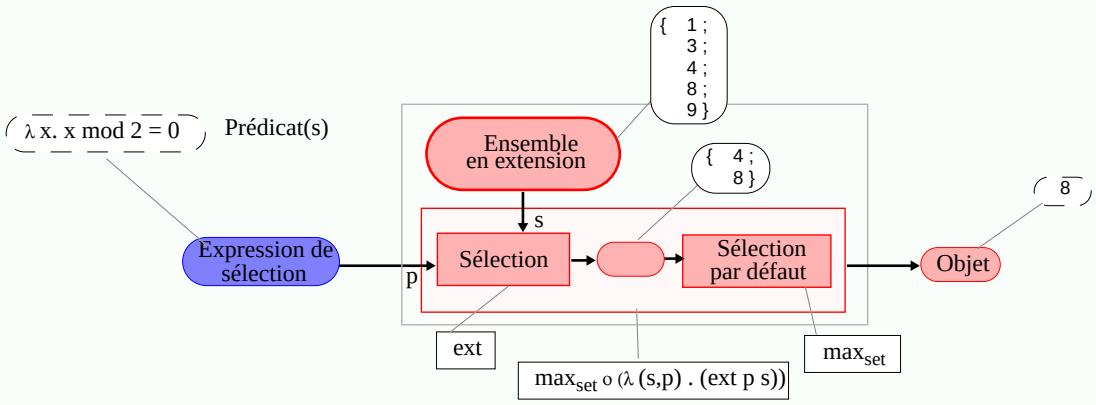
Pour assurer l’unicité du résultat plusieurs solutions ont été proposées :

- *(1) Echec*. La première solution est la plus draconienne. Elle consiste à ne retourner une variante que si c’est la seule sélectionnée par l’expression de sélection[2]. Le système Lemur est basé sur cette solution [PlaiWadg93a].

1. Dans le cas d’un objet générique fonction l’ensemble des composants est partitionné en deux pour former le domaine et le codomaine de la fonction. Au contraire dans le cas d’objets génériques ensembles cette distinction n’est pas faite et aucune hypothèse n’est faite sur l’existence de clés dans la relation.
2. L’interpréteur est donc une fonction de la forme “one o ($\lambda$ (s,p) . ext p s)”. Ici la fonction “one” assure l’unicité du résultat (Annexe A.1.1).

- *(2) Sélection par défaut câblée*. En pratique on préfère plutôt rajouter une sélection par défaut, c'est à dire une fonction prenant en entrée un ensemble de variantes et retournant une seule valeur. Typiquement si l'ensemble est totalement ordonné on peut choisir le maximum. Une telle sélection par défaut est câblée si elle est intégrée dans l'interpréteur (figure 38)[1]. Ce genre de solution est retenue par la plupart des systèmes élémentaires de versionnement. Généralement la dernière révision est retournée. SCCS fait partie de cette classe [Roch74].

*figure 38* Objet générique ensemble, sélection par défaut câblée




- *(3) Sélection par défaut externe*. L'inconvénient d'une sélection par défaut câblée est que celle-ci est imposée par l'interpréteur, normalement unique. La règle par défaut choisie n'est pas nécessairement adaptée à chaque application. Certains systèmes donnent à l'utilisateur la possibilité de spécifier de telles règles dans l'expression de sélection (c'est le cas par exemple des systèmes Adele [Estu85] ou Shape [MahlLamp88]).

Quelque soit le type de sélection par défaut, il s'agit d'interpréter un objet générique ensemble de manière à ce qu'il ait un comportement conforme à un objet générique, c'est à dire que pour un choix donné, une seule variante soit sélectionnée.

## II.3.6 Synthèse

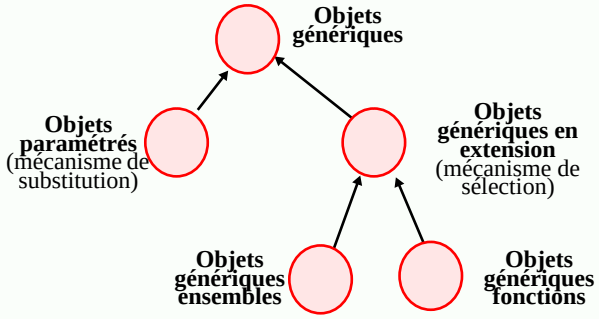
Trois types de représentations ont été présentées : les objets paramétrés, les objets génériques fonctions et les objets génériques ensemble. Alors que les objets paramétrés sont basés sur un mécanisme de *substitution*, les objets génériques fonction et les objets génériques ensemble sont basés sur l'utilisation de mécanisme de *sélection* (tableau 5). Ils sont regroupés sous l'appellation "objets génériques en extension" (figure 39).

Ces différents types ne sont pas incompatibles. Par exemple on imagine facilement une combinaison entre les exemples présentés dans la figure 35 (p.97) et la figure 36 (p.98).

Les problèmes de représentation sont détaillés dans la suite, tout d'abord les aspects concernant la structuration du domaine des objets génériques, puis ceux concernant la structuration du codomaine. En fait nous nous intéresserons plus particulièrement aux objets génériques fonctions.

1. Celui-ci est donc de la forme "défaut o (λ (s,p) . ext p s) La fonction "défaut" est par exemple "$max_{set}$" c'est à dire la fonction retournant l'élément maximum d'un ensemble.

*figure 39* Principales classes d'objets génériques




La hiérarchie ci-contre représente les différents raffinements concernant la représentation des objets génériques. Lorsque l'on parlera "d'objet générique" on fera référence au concept abstrait. Par contre plus l'on descend vers les feuilles plus l'on prend en compte des détails de représentation.

Les objets génériques ensemble ne sont pas très différents et peu de raffinements concernent les objets paramétrés.

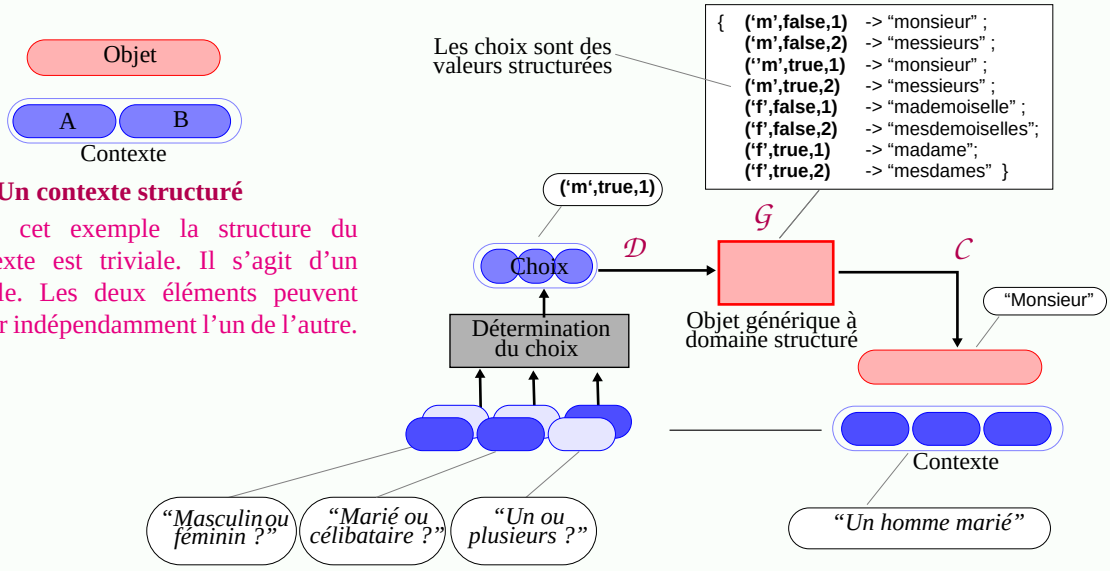
TABLEAU 5 Objets paramétrés, objet génériques fonction et ensemble

| Type d'objet générique | représentation | mécanisme d'interprétation | terme pour "choix" | terme pour "génération" | exemple |
|---|---|---|---|---|---|
| objet paramétré | utilise des variables | substitution | paramètre | instanciation | λn • "Cher" ^ n |
| objet générique fonction | fonction en extension | sélection | sélecteur | sélection | { 1 -> "monsieur" ; 2 -> "madame" } |
| objet générique ensemble | ensemble en extension | sélection + sélection par défaut (câblée ou non) | expression de sélection | sélection | { (1,"monsieur") ; (1,"Mr") ; (2,"madame") } |

# II.4 Structuration du domaine d'un objet générique

Le contexte d'utilisation d'un objet est souvent complexe et différents éléments de ce contexte varient indépendamment les uns des autres. L'objet doit être adapté à ces variations même si celui-ci est atomique. Il est alors utile de structurer le domaine de l'objet générique pour refléter la structuration du contexte (figure 41). Dans la suite, pour faciliter l'exposé nous noterons $\mathcal{G}$ l'ensemble des objets génériques, $\mathcal{D}$ le domaine et $\mathcal{C}$ le codomaine d'un objet générique. Ces notations seront utilisées de manière informelle. Dans cette section il s'agit de déterminer $\mathcal{D}$.

*figure 41* Contexte structuré => domaine structuré




***(a)* Un contexte structuré**

Dans cet exemple la structure du contexte est triviale. Il s'agit d'un couple. Les deux éléments peuvent varier indépendamment l'un de l'autre.

***(b)* Un objet générique à domaine structuré**

Ici, le contexte est logiquement composé de trois aspects pouvant varier indépendamment les uns des autres. Pour s'adapter à cette situation le domaine de l'objet générique proposé est structuré : c'est le produit cartésien de trois ensembles basiques : {'m','f'} × Bool × {1..2}. Ce domaine peut être vu comme un espace à trois dimensions.

## II.4.1 Problématique

Structurer le domaine d'un objet générique détermine la facilité d'expression des choix. Les opérations basiques, comme par exemple la génération d'une variante, se font en effet à partir d'éléments de ce domaine. Pour l'utilisateur il est donc particulièrement important que ceux-ci soient significatifs et compréhensibles. Si ce n'est pas le cas l'objet générique risque d'être inutilisable. Pour les éléments du domaine il est nécessaire de choisir : (1) une notation utilisable par l'utilisateur, (2) une représentation interne facilement manipulable et stockable[1].

1. Si l'on se réfère à la programmation globale, le problème traité dans cette section est *grossièrement* celui de l'identification des versions. Notons que c'est dans le domaine de la gestion de versions que nous nous situons et non pas dans la gestion de configurations (Pour l'instant on suppose que les objets manipulés sont atomiques). Dans le domaine de la réutilisation, le problème traité pourrait être rapproché du problème de classification des composants (les solutions retenues ici sont bien plus simples. Elles sont par contre utilisées pour une sélection entièrement automatique).

Lors de la conception d'un objet générique différentes questions surgissent : quel domaine $\mathcal{D}$ choisir ? comment désigner un élément particulier du domaine $\mathcal{D}$ (explicitement ou en intention) ? comment décrire un ensemble d'éléments de $\mathcal{D}$ (en extension $\{\mathcal{D}\}_e$ ou en intention $\{\mathcal{D}\}_i$ )[1]? comment relier les éléments du domaine $\mathcal{D}$ et ceux du codomaine $\mathcal{C}$ ?

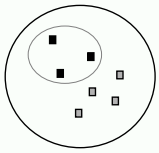
## II.4.2 Domaines basiques

Dans certaines applications des domaines basiques peuvent être suffisants pour le domaine $\mathcal{D}$. Quelques exemples sont représentés graphiquement dans la figure 42.

Un domaine peut être infini (e.g. $\mathcal{D}$=Int) (figure 42.b) ou infini (e.g. "Bool" ou "Char"). On peut être amené à définir des ensembles énumérés. Evidemment ces décisions dépendent de l'application et des possibilités offertes par l'interpréteur dont on dispose. Par exemple l'interpréteur CPP ne propose que des domaines entiers et il est alors nécessaire de "coder" les différents choix (cf Chapitre IV).

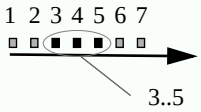
Lorsque c'est possible il est utile de définir une relation d'ordre sur un domaine. S'il s'agit d'un ordre total on pourra utiliser des intervalles pour désigner des ensembles. L'ensemble des entiers est typiquement utilisé pour représenter des variations temporelles et la notion d'intervalle est utilisée de manière extensive dans les bases de données temporelles [Tans&al93] [Lore93].

*figure 42* Exemples de domaines ($\mathcal{D}$=X)



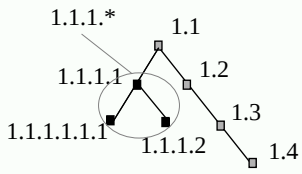
***(a)* Ensemble quelconque**

Les éléments sont identifiés d'une manière ou d'une autre. Un ensemble doit être spécifié en extension.




***(b)* Ensemble totalement ordonné**

On peut désigner un ensemble en compréhension à partir d'une notation d'intervalle par exemple (Base de données temporelles par exemple).




***(c)* Arbre à tronc**

Chaque noeud de l'arbre est un élément de l'ensemble. Parmi les fils d'un noeud, un est distingué. On peut désigner des sous arbres (appelées branches). (SCCS par exemple)

On peut imaginer toute une série d'autres structures pour $\mathcal{D}$ : partition, treillis (ICE [Zell94] [ZellSnel94], Lemur [PlaiWadg93a]), arbre (SCCS [Roch74], RCS [Tich85]), etc. Il est également possible de les combiner (Adele, cf [Estu85]). Il s'agit à chaque fois de trouver une manière de désigner chaque élément et mais aussi des ensembles d'éléments. Les propriétés du domaine interviennent alors. Par exemple pour un arbre, il sera possible de désigner un sous arbre. Pour un treillis on pourra désigner la borne supérieure ou inférieure d'un sous-ensemble, etc.

1. En plus de la notation {X} qui désigne l'ensemble des parties de X, dans cette thèse la notation $\{X\}_e$ (resp. $\{X\}_i$) désigne l'ensemble des parties de X représentées en extension (resp. en intention).

## II.4.3 Espace à n dimensions

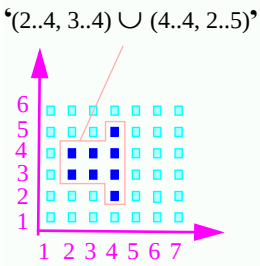
Dans le cas de plusieurs variations indépendantes il est naturel de prendre comme domaine le produit cartésien d'autres domaines (i.e. $\mathcal{D}=X_1 x\ X_2 x\ ...X_n$). Chaque ensemble $X_i$ intervenant dans le produit cartésien est généralement appelé ***dimension*** et l'on parlera d'***espace à n dimensions*** pour le domaine [Krus83] [Lie90] [Scio91] [Sing92] [Munc&al93].

Ce concept est présent dans différents domaines de recherche et plusieurs auteurs suggèrent qu'il s'agit d'un point de rencontre possible pour les problèmes de variations entre les bases de données temporelles, le génie logiciel, la conception assistée par ordinateur et les bases de données géographiques [GadiNair93] [Scio94].

L'avantage d'une telle structuration est qu'elle est relativement "parlante" car on peut l'interpréter géométriquement. Un élément de cet ensemble peut être vu comme un *point* de cet espace. Ce point est repéré dans l'espace par ses *coordonnées*. Un ensemble d'éléments (i.e. un ensemble de points) peut être vu comme un ensemble de *volumes* [Lie90] [Munc93]. Un espace à deux dimensions est naturellement appelé *plan* et les sous-ensembles sont des ensembles de *surfaces*.

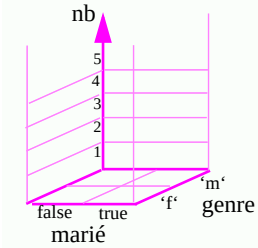
En fait, si l'on veut faire abstraction de ce vocabulaire, on remarquera qu'un point est tout simplement un tuple[1], que ses coordonnées sont les composantes de ce tuple, et qu'un sous-ensemble de l'espace est une relation (i.e. un ensemble de tuples). Dans la suite nous utiliserons ces termes de manière interchangeables. La figure 43 présente des exemples d'espaces.

*figure 43* Espaces à n dimensions ($\mathcal{D}=X_1 x\ X_2 x\ ...X_n$)




***(a)*** **Espace infini à 2 dimensions**

Ici, le produit cartésien à été fait entre deux ensembles ordonnés. Le domaine est un quart de plan. Les notations intervalles peuvent être utilisées pour désigner un ensemble, c'est à dire une surface. Dans l'exemple la surface a été décomposée en deux rectangles : le premier correspond au produit cartésien des intervalles 2..4 et 3..4 et le second à celui de 4..4 et 2..5. Bien d'autre solutions sont possibles. Nous verrons dans les sections suivantes que l'un des problèmes consiste à trouver des représentations compactes. Ce genre de constructions est utilisée dans le cadre des bases de données bi-temporelles (les deux dimensions sont des dimensions temporelles [Tans&al93]).




***(b)*** **Ensemble d'enregistrements (ou de tuples)**

Il s'agit d'un espace à trois dimensions. Contrairement à l'exemple précédent les dimensions sont nommées et de type différent : le domaine est un ensemble d'enregistrement plutôt que de triplets (Annexe A.1.4, "Tuples et enregistrements"). Plus précisément le domaine est : (genre->{'m','f'}, marié->Bool, nb->Int) . Les deux premières dimensions sont finies, la troisième non. L'enregistrement (genre->'m', marié->true, nb->1) est un point de cet espace.

Il est possible de représenter des ensembles de choix en extension, en énumérant chaque choix, où en compréhension à l'aide d'une formule (voir la section suivante). Par exemple le prédicat ‘genre='m' ∧ nb2’ est un prédicat dont l'extension est un ensemble infini de choix (un volume).

Le modèle CoV est basé sur ce type de domaine si ce n'est qu'il impose que chaque dimension soit booléenne [Lie90]. Le modèle correspondant au système EXTRA-V est également très proche [Scio94]

1. ou plutôt un enregistrement si les dimensions sont nommées.

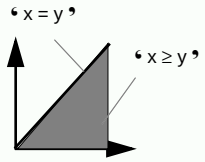
## II.4.4 Représentation d'un ensemble de choix

Il est utile de pouvoir désigner en intention un ensemble de choix (il s'agit de savoir représenter les éléments de $\{\mathcal{D}\}_i$ ). Pour cela il est nécessaire de définir un langage ; typiquement un langage de prédicats. Plus généralement on utilisera plutôt le terme "formule" (pour plus de détails voir l'Annexe A.1.2, "Extension vs. compréhension: énumération vs. formulation"). Les constructions présentes dans ce langage dépendent bien évidemment du type des différentes dimensions. Par exemple si l'ensemble est ordonné on pourra utiliser des opérateurs de comparaisons, des notations d'intervalles, etc. Plus le langage est sophistiqué et expressif, plus il est possible de décrire de manière compacte des ensembles de choix. En contre-partie la manipulation et les transformations de formules peuvent devenir particulièrement complexes et ce coût additionnel ne se justifie que pour certaines applications (par exemple dans le cas de bases de données géographiques). Cette notion est illustrée dans la figure 44.

*figure 44* Représentation d'un ensemble de choix $\{\mathcal{D}\}_i$

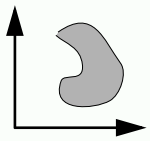
***(a)* Expressions mettant en jeu plusieurs dimensions**

Ici on a un espace à deux dimensions. Les prédicats ' $x \geq y$ ' et ' $x = y$ ' mettent en jeu plusieurs dimensions simultanément. Si les dimensions sont infinies leur extension l'est aussi. De telles surfaces ne peuvent pas être représentées si l'on se limite à des prédicats de la forme : <dimension> <opérateur> <constante>.




***(b)* Surfaces quelconques**

Dans le cas des bases de données géographiques les dimensions sont les dimensions géographiques [GadiNair93]. Par exemple le domaine représente un terrain et l'on cherche à représenter l'altitude. Les courbes de niveaux délimitent des zones quelconques. La représentation compacte de tels ensembles reste un thème de recherche, tout comme l'implantation efficace des opérations d'intersection, d'union, etc.



## II.4.5 Trois types d'estampilles

Les objets génériques fonctions peuvent en fait être représentés de nombreuses manières en utilisant la notion d'estampille définie ci-dessous. Celle-ci est elle-même liée à des opérations définies dans des modèles de données classiques.

Les modèles relationnels imbriqués introduisent l'opérateur d'***imbrication*** (noté traditionnellement $\mu$) et l'opération inverse, la "***désimbrication***" (notée $\nu$)[1]. Ces opérateurs permettent d'éviter la duplication d'informations en regroupant tous les tuples ayant la même valeur pour un attribut donné [Vand93]. Ces opérations sont typiquement définies sur des ensembles. Dans cette thèse on s'intéresse aussi aux autres constructeurs, par exemple aux listes (un fichier est une liste de lignes). Dans le cas général on parlera alors de ***factorisation*** et de ***développement*** (Annexe A.1.10, "Factorisation vs. développement"). Comme nous allons le voir dans cette section, ces techniques peuvent être utilisées pour représenter des objets génériques fonctions.

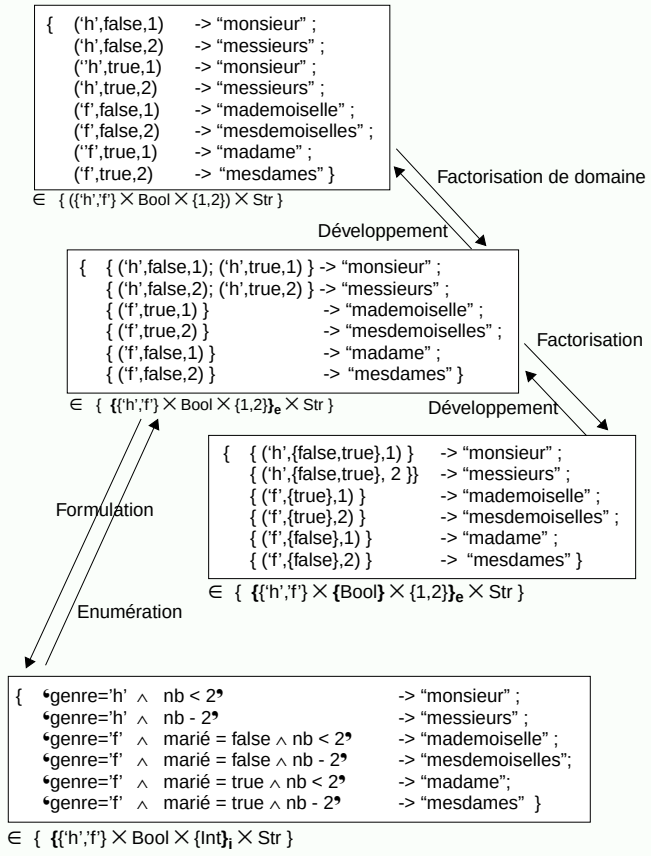
1. "nest" et "unest" en anglais, ou "fold" et "unfold".

### II.4.5.1 *Factorisation de domaine*

Les objets génériques fonctions étant des ensembles de couples il peut être intéressant de factoriser leur domaine ; sinon certaines variantes (i.e. certains éléments du codomaine) peuvent être inutilement dupliquées[1]. Selon la représentation choisie on peut alors distinguer deux sous classes d'objets génériques fonctions :

- ***objet générique par cas***. A *chaque choix* on associe une variante. La représentation de l'objet générique est donc une fonction de la forme $\mathcal{G} = \mathcal{D} \rightarrow \mathcal{C}$, plus précisément de la forme $\mathcal{G} = \{ \mathcal{D} \times \mathcal{C} \}$ (figure 45.a).
- ***objet générique par morceaux***[2]. A *un ensemble de choix* on associe une variante. Si chaque ensemble est décrit en extension on parlera d'***objet générique par morceaux en extension***. La représentation de l'objet générique est alors de la forme $\mathcal{G} = \{ \{\mathcal{D}\}_e \times \mathcal{C} \}$ (figure 45.b).

*figure 45* Factorisation et formulation de domaine, un exemple




***(a)* Objet générique par cas**

A chaque choix est associé une variante. Les variantes "messieurs" et "monsieur" sont dupliquées.

***(b)* Objets génériques par morceaux en extension**

A chaque ensemble de choix est associée une variante. Le domaine de la fonction a été factorisé. Les variantes ne sont donc pas dupliquées.

***(c)* Objet générique par morceaux en intention**

A la place de représenter l'ensemble des choix en extension celui-ci est représenté en compréhension (i.e. en intention) via l'utilisation d'une "formule". Ici la fonction représentée est infinie. Elle est plus générale que celles des exemples (a) et (b) puisque la variable "nb" correspondant au nombre de personnes est un entier positif quelconque.

1. Dans cette section les objets manipulés sont excessivement simples et le problème posé par la duplication d'éléments du codomaine n'est pas mis en évidence. Par contre si l'on fait l'analogie avec le génie logiciel, un texte pourrait être vu comme un système, chaque phrase comme un sous système, chaque mot comme un module et chaque caractère comme une procédure. Clairement représenter plusieurs fois la même procédure pose des problèmes de maintenance multiple.
2. Une terminologie analogue est parfois utilisée en mathématique lorsqu'une fonction a une définition différente sur différents intervalles. Par exemple la fonction suivante est une fonction définie par morceaux. [Waru66]

$$f(x) = \begin{cases} x+1 \text{ si } x<1 \\ x+4 \text{ si } x>=1 \text{ et } x<99 \\ x+2 \text{ si } x>100 \end{cases}$$

### II.4.5.2 *Formulation de domaine*

Les techniques de formulation présentées en Annexe A.1.2 ("Extension vs. compréhension: énumération vs. formulation") peuvent être appliquées aux objets génériques par morceaux. On parlera alors d'***objets génériques par morceaux en intention*** puisque une formulation permet de représenter chaque sous-ensemble du domaine en intention. Autrement dit on obtient un objet générique de la forme $\mathcal{G} = \{ \{\mathcal{D}\}_i \times \mathcal{C} \}$ (figure 45.c).

L'avantage majeur de cette représentation c'est qu'elle permet de représenter des fonctions infinies. Cette solution est d'ailleurs retenue dans la plupart des systèmes proposant des objets génériques fonctions. C'est le cas entre des bases de données temporelles, du préprocesseur CPP et du modèle CoV [Lie90].

### II.4.5.3 *Estampilles*

D'un point de vue abstrait, un objet générique fonction est un ensemble de couples (choix, variante). Comme nous venons de le voir la représentation précise peut varier mais à chaque fois le principe est le même : à chaque variante est associée ce que l'on appellera une "***estampille***"[1].

Si l'on note $\mathcal{E}$ l'ensemble des estampilles, un objet générique fonction est de la forme $\{ \mathcal{E} \times \mathcal{C} \}$ avec $\mathcal{E}=\mathcal{D}$ ou $\mathcal{E}=\{\mathcal{D}\}_e$ ou encore $\mathcal{E}=\{\mathcal{D}\}_i$. Le tableau 6 montre les diverses possibilités que l'on peut obtenir à partir de la factorisation et de la formulation du domaine.

TABLEAU 6 3 types d'estampilles

| $\mathcal{E}$ = | Estampille | Objet générique fonction | $\mathcal{G}$ = | Exemple |
|---|---|---|---|---|
| $\mathcal{D}$ | un choix particulier | objet générique par cas | $\{ \mathcal{D} \times \mathcal{C} \}$ | {***1***->a; ***2***->a ; ***3***->a ;***4*** -> b} |
| $\{\mathcal{D}\}_e$ | un ensemble de choix en extension | objet générique par morceaux en extension | $\{ \{\mathcal{D}\}_e \times \mathcal{C} \}$ | { ***{1;2;3}*** -> a ; ***{4}*** -> b } |
| $\{\mathcal{D}\}_i$ | un ensemble de choix en intention | objet générique par morceaux en intention | $\{ \{\mathcal{D}\}_i \times \mathcal{C} \}$ | {•1≤***x***≤3• -> a ; ***{x=4}*** -> b } |

## II.4.6 Trois types de structures de contrôle

Associer une estampille à une variante est une chose mais encore faut il organiser cette "collection" d'associations. Jusque là nous avons laissé supposer qu'il s'agissait toujours d'un ensemble mais ce n'est pas nécessairement le cas. Si l'on se place du point de vue de la programmation détaillée on parlera de ***structures de contrôle*** ; mais on peut aussi garder une vision centrée sur le modèle de données et considérer le constructeur utilisé pour structurer cette "collection". Trois possibilités peuvent être distinguées :

- ***Construction alternative*** (Ensemble). Les programmes présentés dans figure 37.a (page 99) utilisent des constructions alternatives (typiquement l'instruction "case"). La structure de donnée équivalente est un *ensemble* de couples estampille-variante où les estampilles forment une partition du domaine (il n'y a pas de recouvrement et le domaine est décrit en totalité).
- ***Construction conditionnelle*** (Liste). Il s'agit traditionnellement de l'instruction "if ... elseif

1. Une terminologie similaire est utilisée dans le domaine des bases de données temporelles ("timestamp"). D'autres auteurs parlent aussi de "tag" [Krus83] [Bers&al88].

... fi"). Dans ce cas la représentation est une *liste* de couples et évidemment l'interpréteur change : les comparaisons avec les estampilles sont faites séquentiellement et c'est la première expression qui retourne vrai qui est sélectionnée (figure 46.a)[1].

- ***Construction conditionnelle imbriquée*** (Arbre). Les constructions conditionnelles peuvent être imbriquées (figure 46.b). La structure correspondante est un *arbre* de décision. Le préprocesseur CPP offre cette construction. En pratique son usage est important car il permet de simplifier considérablement les estampilles.

*figure 46* Constructions conditionnelles

```
fun (genre,marie,nb) ->
if       genre='m' & nb<2          then "monsieur"
else if  genre='m' & nb>=2         then "messieurs"
else if  genre='f' & marie=false & nb<2   then "mademoiselle"
else if  genre='f' & marie=false & nb>=2  then "mesdemoiselles"
else if  genre='f' & marie=true & nb<2    then "madame"
else if  genre='f' & marie=true & nb>=2   then "mesdames"
else                               raise undefined
```

```
fun (genre,marie,nb) ->
if       genre='m' & nb<2          then "monsieur"
else if  genre='m'                 then "messieurs"
else if  genre='f' & marie=false & nb<2   then "mademoiselle"
else if  genre='f' & marie=false          then "mesdemoiselles"
else if  genre='f' & nb<2          then "madame"
else if  genre='f'                 then "mesdames"
else                               raise undefined
```

***(a)*** **Constructions conditionnelles (Liste)**

Les deux représentations ci-dessus sont équivalentes. La deuxième utilise le fait que les conditions sont évaluées séquentiellement. Ce n'est pas le cas de la première qui peut être directement obtenue à partir d'une représentation par morceaux en intention (comparer avec la figure 45.c (page 106). Mis à part les différences syntaxiques, l'un est une ensemble de couples, l'autre une liste de couples).

***(b)*** **Constructions conditionnelles imbriquées (Arbre)**

Les langages offrant des constructions conditionnelles permettent normalement leur imbrication. Cette possibilité permet de simplifier notablement les estampilles.

```
fun (genre,marie,nb) ->
   if      genre='m' then
           if  nb<2 then "monsieur"
               else      "messieurs"
   else if genre='f' then
               if   marie=false then
                    if   nb<2then   "mademoiselle"
                         else       "mesdemoiselles"
               else
                    if   nb<2 then  "madame"
                         else       "mesdames"
   else   raise undefined
```

Les trois possibilités sont résumées dans le tableau 7.

TABLEAU 7 3 types de structures de contrôle

| Structure de contrôle | Constructeur | Objet générique G | Exemple |
|---|---|---|---|
| alternative | ***Ensemble*** | ***{*** $\mathcal{E} \times \mathcal{C}$ ***}*** | figure 37.a (page 99) |
| conditionnelle | ***Liste*** | ***[*** $\mathcal{E} \times \mathcal{C}$ ***]*** | figure 46.a |
| conditionnelle imbriquée | ***Arbre*** | ***Tree(*** $\mathcal{E} \times \mathcal{C}$ ***)*** | figure 46.b |

La compréhension des instructions conditionnelles est généralement plus complexe (comparer par exemple la figure 46.b et la figure 41 (p.102)). Le fait d'introduire un ordre fait que de nombreuses représentations sont possibles pour une même fonction. Certaines représentations sont sans doute plus naturelles que d'autres. Des règles de *restructuration* préservant la sémantique peuvent alors être utiles.

Ces considérations ne sont pas gratuites. En effet dans cette thèse nous nous intéressons plus particulièrement à la compréhension et ré-ingénierie des programmes utilisant le préprocesseur CPP[2]. Ces activités sont bien évidemment basées sur la connaissance des propriétés des constructions offertes par ce langage.

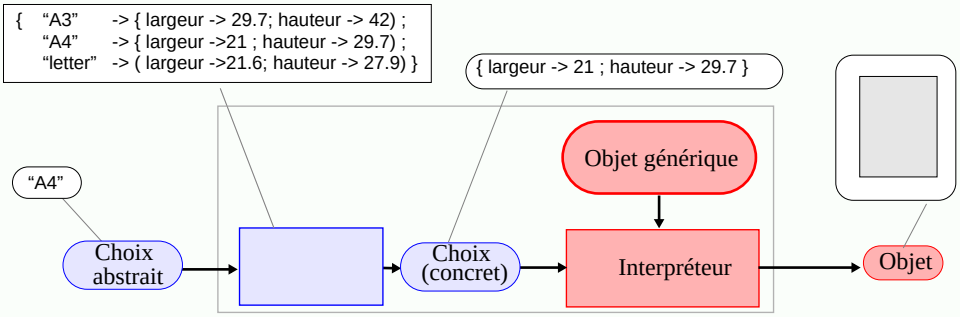
1. Il s'agit là d'une simplification ne prenant pas en compte la clause "else". Si celle-ci est présente la structure correspondante est un couple (liste,variante), la variante étant sélectionnée si aucune association de la liste n'est sélectionnée.
2. *"Conditional compilation constructs (#ifdef), especially nested ones, lead to programs that are virtually unreadable by humans."* [WeinCG92].

## II.4.7 Abstraction, réduction et simplification de domaine

En pratique le domaine d'un objet générique peut être très complexe. Dans le cas d'un logiciel portable et paramétré, des centaines de dimensions peuvent être mises en jeu (cf Chapitre IV). Il est nécessaire d'assister l'utilisateur de l'objet générique ; autant pour déterminer un choix que pour comprendre le domaine de celui-ci.

Il peut alors être utile de nommer certains sous-ensembles ou éléments du domaine et même laisser cette possibilité à l'utilisateur. Il pourra ainsi associer des noms significatifs et mnémotechniques à certaines combinaisons utiles et les réutiliser par la suite. Cette technique, qui sera appelée *abstraction de domaine*, peut être mise en oeuvre en ajoutant une fonction préalable à l'objet générique (figure 47). Le domaine de cette fonction sera appelé ***domaine abstrait*** par opposition au ***domaine concret*** qui est le domaine de l'objet générique.

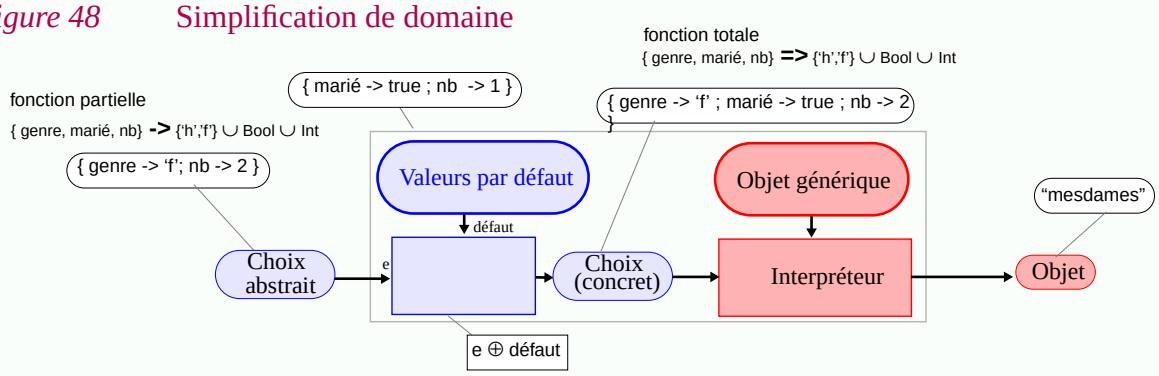
*figure 47* Abstraction et réduction de domaine




L'objet générique est paramétré par ses dimensions. Quelques choix spécifiques sont nommés.

Accepter des choix non totalement spécifiés et compléter ceux-ci avec des valeurs par défaut est une technique de *simplification de domaine*. Elle peut être intégrée à cette fonction (figure 48).

*figure 48* Simplification de domaine




L'objet générique est ici précédé d'une fonction de simplification de domaine : à la place d'avoir à spécifier systématiquement une valeur pour chaque dimension, des valeurs par défaut sont proposées. Le domaine concret est un ensemble de fonctions totales alors que le domaine abstrait est un ensemble de fonctions partielles. Dans l'exemple ci-dessus la dimension 'marié' peut être omise par l'utilisateur de l'objet générique. Ceci correspond à l'usage courant : par défaut dans une lettre "madame" est utilisé plutôt que "mademoiselle".
De multiples implantations sont possibles pour la fonction de simplification de domaine. Celle-ci peut être intégrée à la représentation de l'objet générique ; par exemple via l'introduction de valeurs par défaut.

```ada
function g ( genre : Char ; marie: Bool := true ; nb : Integer := 1 ) return Str is
...
```

**(a) Représentation de l'objet générique en ada**

```ada
g ( genre => 'f' ; nb => 2 )
```

**(b) Variante**
Le paramètre "marié" n'est pas spécifié.

Cette fonction peut être une bijection ou non. Autrement dit certains choix concrets peuvent être inaccessibles. Dans ce cas on parlera de *réduction de domaine*. Eventuellement certaines variantes ne pourront pas être générées. Une telle réduction est parfois faite expressément car l'on sait que seules certaines parties du domaine ont un sens. Dans d'autres cas réduire le domaine est une manière de le simplifier. On remarque en effet que pour beaucoup d'applications, l'utilisateur se limite généralement à faire des choix "standards". Il est alors préférable de lui présenter un petit nombre de possibilités bien identifiées plutôt que de le noyer dans une foule de détails (figure 47). Il est de toute façon possible d'utiliser directement l'objet générique si la fonction de simplification ne convient pas.
De telles considérations peuvent sembler superflues mais nous verrons qu'en pratique les contextes sont si complexes que ce genre de technique est fondamentale.

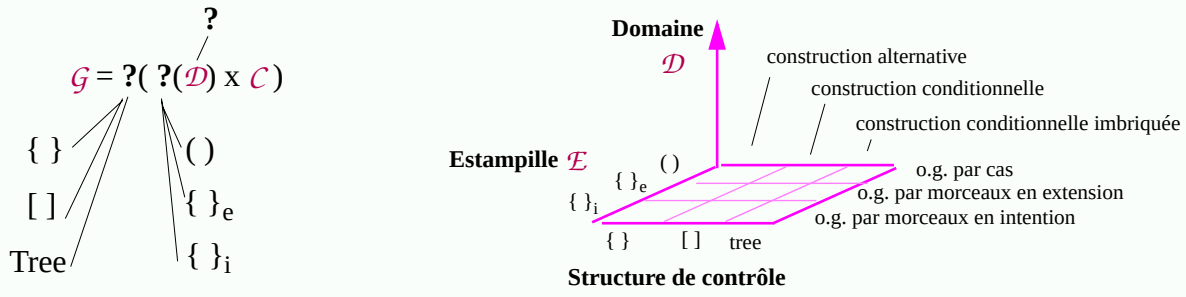
### II.4.8 Synthèse

Dans cette section, nous avons vu quels étaient les problèmes relatifs à la structuration du domaine des objets génériques. Une telle structuration a pour but de rendre plus facilement utilisable un objet générique. Elle peut également être utilisée afin d'obtenir des représentations plus compactes et surtout d'éviter les duplications.

En ce qui concerne les objets génériques fonctions et leur représentation, 3 caractéristiques doivent être choisies (figure 49) :

- **La structure du domaine** ($\mathcal{D}$= X ou $X_1$x $X_2$ x ...$X_n$) (II.4.2, II.4.3).
  Les propriétés que possède le domaine sont essentielles pour pouvoir désigner en intention des éléments ou des sous-ensembles de ce domaine.
- **Le type d'estampille** ( $\mathcal{E}$= $\mathcal{D}$ ou $\{\mathcal{D}\}_e$ ou $\{\mathcal{D}\}_i$ ) (II.4.5)**.**
  Les objets génériques fonctions sont représentés comme un ensemble de couples estampilles-variantes. Nous avons distingué (1) les objets génériques par cas, (2) les objets génériques par morceaux en extension et (3) les objets génériques par morceaux en intention.
- **La structure de contrôle** ( $\mathcal{G}$ = {$\mathcal{E}$ x $\mathcal{C}$} ou [$\mathcal{E}$ x $\mathcal{C}$] ou Tree($\mathcal{E}$ x $\mathcal{C}$) ) (II.4.6).
  Dans le cas où les estampilles sont des ensembles, la structure de contrôle choisie est importante car elle permet de simplifier celles-ci. Par contre la facilité de compréhension peut s'en ressentir.

*figure 49* Taxonomie pour les objets génériques fonctions

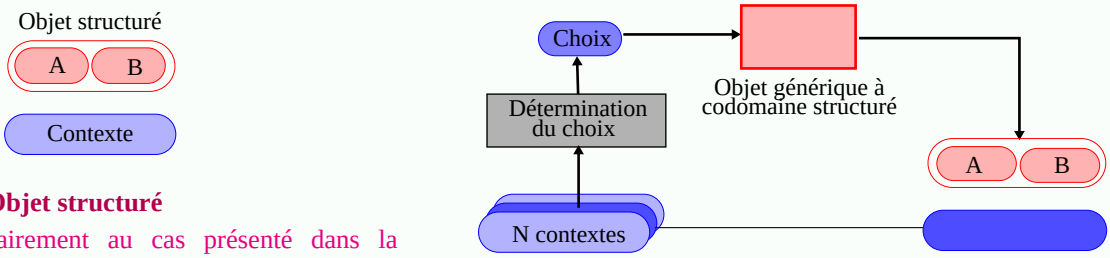



Il reste à prendre en considération la structure du codomaine $\mathcal{C}$.

# II.5 Structuration du codomaine d'un objet générique

Dans la section précédente nous avons vu le cas où un objet atomique était utilisé dans un contexte structuré. Il s'agissait de structurer le domaine $\mathcal{D}$ d'un objet générique. Ici nous allons étudier le cas inverse : celui où l'on manipule des objets structurés, c'est à dire la cas où le codomaine $\mathcal{C}$ est structuré. La figure 50 présente cette idée (à comparer avec la figure 41 (p.102))[1].

*figure 50* Objet structuré => Codomaine structuré




***(a)* Objet structuré**
Contrairement au cas présenté dans la figure 41.a (page 102), ici il s'agit d'un objet structuré dans un contexte simple.

***(b)* Un objet générique à codomaine structuré**
L'objet générique retourne un objet structuré (i.e. son codomaine est structuré).

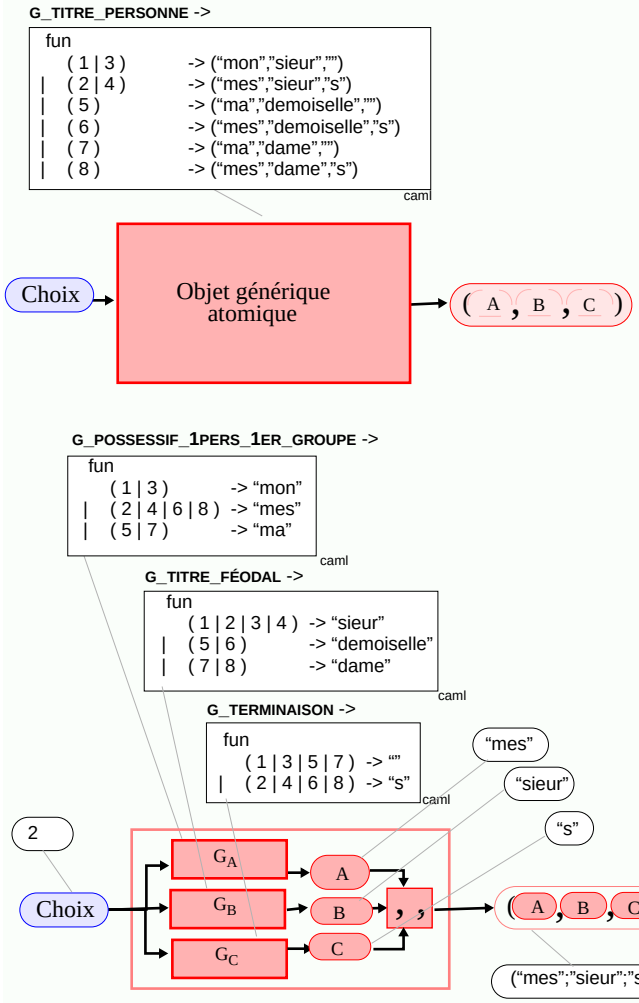
## II.5.1 Variation détaillée vs. variation globale

Lorsque l'on cherche à construire des objets génériques pouvant générer des objets structurés, deux solutions sont possibles selon la granularité choisie :

- ***Variation globale***[2]. La structure des objets n'est pas utilisée ; ceux-ci sont considérés comme étant atomiques. Les méthodes décrites dans les sections antérieures sont alors utilisables telles quelles. Cette solution triviale est toujours possible. Les problèmes de variations sont traités à un niveau de granularité forte dans la mesure où le contenu des objets n'intervient pas. On parlera alors de *variation globale* (figure 51.a).
- ***Variation détaillée***. Ici au contraire il s'agit d'une solution considérant une granularité fine et l'on utilisera le terme *variation détaillée*. La structure des objets est exploitée et l'objet générique est lui même structuré. Celui-ci est typiquement construit à partir d'autres objets génériques. D'un point de vue abstrait ce problème revient alors à définir une fonction à partir d'autres fonctions (figure 51.b). Il est clair que dans un langage de programmation de très nombreuses solutions sont possibles pour réaliser cette tâche.

1. Bien qu'il s'agisse de problèmes différents, la structuration du domaine $\mathcal{D}$ et du codomaine $\mathcal{C}$ vont souvent de paire. Afin de les rendre plus naturels, les exemples choisis dans cette section utiliseront souvent des domaines structurés.
Schématiquement, du point de vue de la programmation détaillée, nous nous intéressons maintenant au problème de la gestion de configurations (plus exactement au versionnement d'objets structurés. La section antérieure était plus proche de la gestion de versions car les objets étaient supposés être atomiques).
2. L'utilisation du terme variation est en fait un abus de langage puisque ici on s'intéresse à la manière dont sont représentés les objets génériques et non pas à la source du problème impliquant l'utilisation de ceux-ci.

*figure 51* Variation globale vs. variation détaillée, un problème de granularité




***(a)* Variation globale, objet générique atomique**

La solution la plus simple consiste à considérer les objets structurés comme des valeurs atomiques. Dans l’exemple ci-contre il s’agit d’un triplet de chaînes de caractères. Bien que la fonction soit représentée par morceaux (i.e. le domaine est factorisé), certaines composantes des triplets sont dupliquées de nombreuses fois. Cet objet est de la forme $\mathcal{G} = \{ \mathcal{E} \text{ x (Str x Str x Str)} \}$ où $\mathcal{E} = \{1..8\}_e$

***(b)* Variation détaillée, objet générique construit**

Un objet générique construit permet de générer des objets structurés. Il est lui même construit à partir d’autres objets génériques. Dans l’exemple il s’agit d’un triplet d’objets génériques, c’est à dire un élément de $\{ \mathcal{E} \text{ x Str} \} \text{ x } \{ \mathcal{E} \text{ x Str} \} \text{ x } \{ \mathcal{E} \text{ x Str} \}$ Remarquez qu’aucun élément n’est dupliqué. La structure exacte de l’objet générique est étudiée par la suite.

Par souci d’abstraction, les objets génériques $G_A$ , $G_B$ et $G_C$ ont été représentés comme étant des fonctions et non pas des programmes interprétés.

La distinction entre variation globale et variation détaillée joue une rôle important dans cette thèse et il y sera souvent fait référence par la suite. Mentionnons simplement que la variation globale est plus simple à mettre en oeuvre que la variation détaillée mais qu’en contrepartie elle impose souvent une duplication d’informations et donc des problèmes de maintenance multiple (Section I.4.4.5).

## II.5.2 Fragments = variantes ou différences

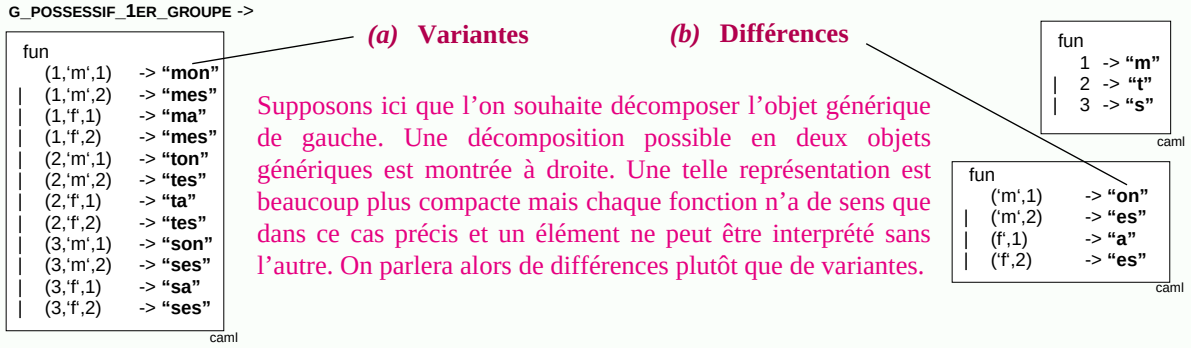
Deux raisons différentes peuvent mener à étudier les problèmes de granularité et de variations :

- *Approche ascendante*. On dispose de plusieurs objets génériques et il apparaît nécessaire de les composer pour définir un objet structuré. Il s’agit d’un problème de gestion de configurations.
- *Approche descendante*. On dispose d’un objet générique atomique générant des objets structurés (i.e. variation globale) et l’on est à la recherche d’une solution technique pour pouvoir faire varier chaque élément indépendamment les uns des autres, par exemple pour minimiser la duplication d’informations.

Dans une approche ascendante, les objets sont construits à partir de briques plus petites, mais ayant un sens intrinsèque. Au contraire l'approche descendante peut parfois amener à une fragmentation des informations dans laquelle il n'est plus possible d'associer de sémantique à chaque élément de base. On parlera alors de ***différences*** plutôt que de variantes. Ce phénomène est illustré dans la figure 52.

*figure 52* Variantes vs. différences

Les exemples de la figure 51 (p.112) correspondent à une approche descendante : on cherche à décomposer un objet générique pour éviter la duplication d'informations. Cependant la décomposition du titre d'une personne est faite selon des frontières logiques. Chaque objet générique composant a un sens en soit et pourrait être réutilisé dans un autre contexte. Cela n'aurait pas été le cas si la décomposition avait été différente.




Plus généralement, un objet générique structuré peut être G-*P*-correct sans pour autant que les objets génériques qui le compose le soit.

Par mesure de simplification on parlera de "***fragment***" qu'il s'agisse d'une variante ou d'un différence. Une telle terminologie est utilisée dans les modèles CoV [Lie90] et P-Edit [Krus83].

Le terme "fragment" sera généralement utilisé avec le terme "estampille". On pourra dire qu'une estampille est associée à un fragment sans s'attacher ni à la représentation de l'estampille, ni à la sémantique du fragment. Si l'on note $\mathcal{F}$ l'ensemble des fragments, un objet générique fonction sera de la forme $\mathcal{G} = \{\mathcal{E} \times \mathcal{F}\}$.

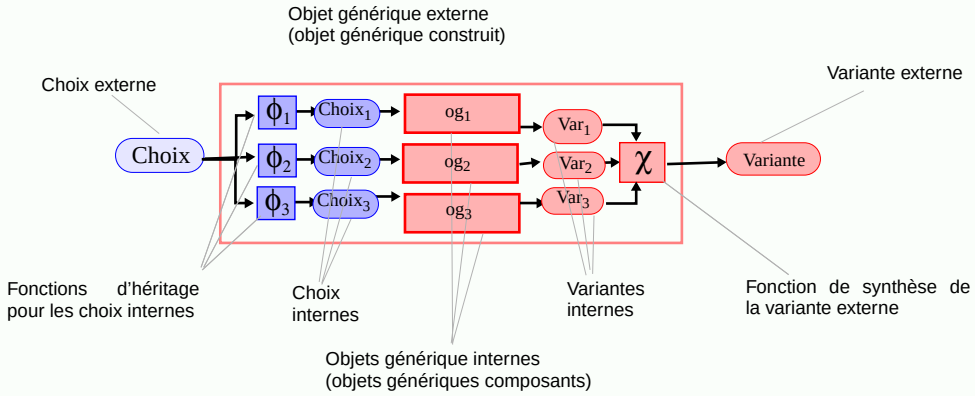
## II.5.3 Cohérence entre objets génériques

Un objet générique peut être construit à partir de différents autres objets génériques. Par facilité d'expression, l'objet générique construit sera qualifié d'*externe* et les objets génériques composants seront qualifiés d'*internes*. Cette terminologie sera appliquée également aux choix et aux variantes. Quelles sont les relations entre le domaine externe et les domaines internes? (resp. entre le choix externe et les choix internes?) Qu'en est il pour les codomaines et les variantes ?

Ces questions sont importantes pour pouvoir assurer une certaine "cohérence" entre les différentes variantes. Nous étudions ici plus particulièrement le paradigme fonctionnel. C'est un mode de calcul particulier se caractérisant par l'héritage des choix internes et la synthèse de la variante externe[1]. Le paradigme impératif, utilisé entre autre par CPP, sera étudié dans le Chapitre IV (Section IV.3.5). Pour simplifier l'exposé, un cas particulier est étudié : le cas où l'objet générique est construit à partir d'un triplet d'objets génériques (figure 53).

1. Le terme héritage ne fait pas ici référence au concept d'héritage des modèles orientés objets. Tout comme le terme synthèse il provient plutôt des définitions dirigées par la syntaxe (grammaires attribuées).

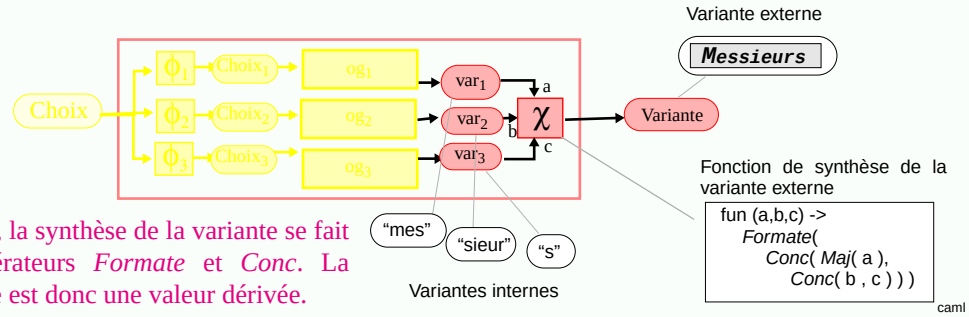
*figure 53* Héritage pour les choix internes et synthèse de la variante externe




### II.5.3.1 *Synthèse de la variante externe*

La variante externe est synthétisée si elle est définie à partir des différentes variantes internes. Ceci se fait normalement à partir de constructeurs. Par exemple dans la figure 51.b (page 112) les trois variantes internes étaient regroupées dans un triplet via le constructeur de tuples. Il est également possible d'appliquer des opérateurs. On obtient alors un objet générique structuré dérivé. La ***fonction de synthèse*** peut être complexe comme le montre la figure 54.

*figure 54* Synthèse de la variante externe, un exemple




Dans cet exemple, la synthèse de la variante se fait à partir des opérateurs *Formate* et *Conc*. La variante résultante est donc une valeur dérivée.

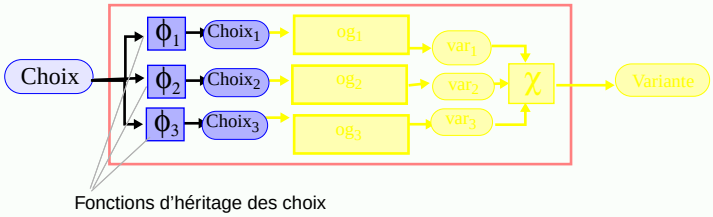
### II.5.3.2 *Héritage pour les choix internes*

La solution la plus simple et la plus intuitive pour déterminer les choix internes est de calculer ceux-ci à partir du choix externe. En pratique les fonctions permettant de faire cela, appelées ***fonctions d'héritages***, sont souvent très simples. La figure 55 présente diverses possibilités.

- ***Domaine composé***. Le domaine de l'objet générique composé est le produit cartésien des domaines des composants, i.e. chaque composant est choisi de manière indépendante. Cette solution permet de générer n'importe quelle combinaison de variantes. Aucune cohérence n'est assurée (figure 55.c).
- ***Domaine global***. Le même choix est appliqué à tous les composants. Le domaine est donc le même autant pour les composants que pour l'objet générique construit. On parlera de domaine global. Le qualificatif "global" fait ici référence à la "portée" du choix par rapport à l'objet générique construit. Cette solution permet de "synchroniser" le choix des différentes variantes (figure 55.b). Elle suppose cependant que les objets génériques composants aient été conçus de manière cohérente (ils doivent tous avoir le même domaine de définition). C'était le cas dans la figure 51.b (page 112).

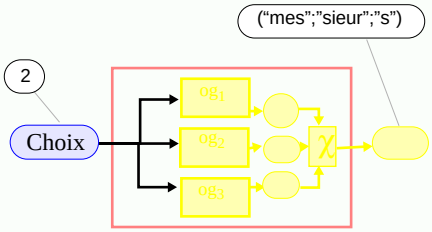
- ***Domaine universel***. On parlera de domaine universel, si un choix unique permet de synchroniser non pas les composants d'un objet générique construit, mais l'ensemble de tout les objets génériques utilisés dans une application donnée. Cette solution extrême est adoptée par différents systèmes de versionnement car elle est conceptuellement très simple et est plus facile à mettre en oeuvre [Lie90] [Scio94]. En contrepartie le domaine universel peut devenir très complexe (des centaines de dimensions voir plus).

*figure 55* Héritage des choix internes



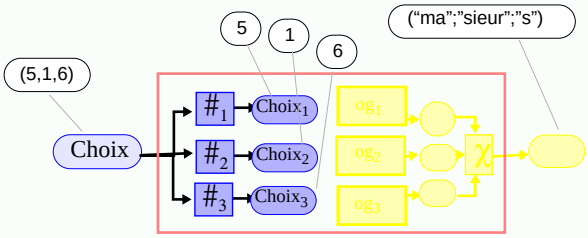

***(a)*** **Fonctions d'héritage**

Les fonctions d'héritage $\phi_\iota$ ont pour but de calculer les choix des objets génériques composants à partir du choix exprimé pour l'objet générique structuré. En pratique ces fonctions sont souvent triviales.




***(b)*** **Domaine global**

Le choix est transmis à tous les objets génériques composants. On a $\phi_1 = \phi_2 = \phi_3 = Id$. Ceci permet de "synchroniser" les différents variantes. Dans cet exemple l'objet générique construit est en fait ($og_1$ # $og_2$ # $og_3$) où # est l'opérateur d'application duale de fonction (Annexe A.1.7).




***(c)*** **Domaine composé**

Le domaine externe est le produit cartésien des domaines internes. Les fonctions d'héritages des choix permettent d'extraire les choix respectifs. Toutes les combinaisons peuvent donc être générées. Dans cet exemple l'objet générique construit est ($og_1$ || $og_2$ || $og_3$ ) où || est l'opérateur d'application parallèle (Annexe A.1.7).

Cette liste présente seulement quelques grandes idées.

### II.5.3.3 *Interprétation des fonctions d'héritage ou de synthèse*

En ce qui concerne la représentation des fonctions d'héritages ou de synthèse deux solutions sont possibles :

- Ces fonctions sont "câblées" dans l'interpréteur et sont figées. C'est le cas entre autre pour les systèmes proposant un domaine universel (dans ce cas ces fonctions sont triviales puisque elles correspondent tout simplement à la fonction identité).
- Leur spécification est intégrée à la représentation de l'objet générique (figure 56).

*figure 56* Un exemple complet d'héritage et de synthèse

L'objet générique **G_TITRE_PERSONNE** a été construit à partir de 3 autres objets génériques déjà existants. Il a donc été nécessaire de s'adapter aux conventions prises par ceux-ci en ce qui concerne leur domaine. Les fonctions d'héritage des choix sont décrites dans la représentation de l'objet générique. Elles permettent de faire les conversions de domaines nécessaires.

L'objet générique **G_POSSESSIF** est structuré selon 4 dimensions : (1) le numéro du sujet de 1 à 3 (je,tu,il), (2) le numéro du groupe (1er groupe ou 2ème groupe), (3) le genre de l'objet (masculin ou féminin), (4) le nombre d'objets (2 permet de coder le pluriel).

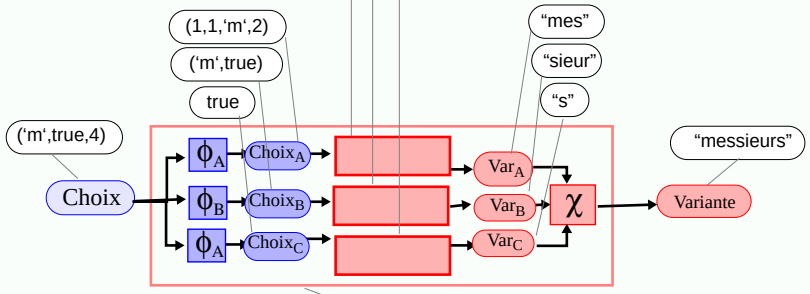
L'objet générique **G_TITRE_FEODAL** est structuré selon deux dimensions : le genre de la personne, le fait qu'elle soit mariée ou non.

L'objet générique **G_TERMINAISON** est structuré selon une seule dimension. Le booléen indique s'il s'agit d'un pluriel ou non.

```
let G_POSSESSIF = fun
    |  (1,1,`m`,1)  -> "mon"
    |  (1,1,`m`,2)  -> "mes"
    |  (1,1,`f`,1)  -> "ma"
    |  (1,1,`f`,2)  -> "mes"
    |  (2,1,`m`,1)  -> "ton"
    |  (2,1,`m`,2)  -> "tes"
    |  (2,1,`f`,1)  -> "ta"
    |  (2,1,`f`,2)  -> "tes"
    |  (3,1,`m`,1)  -> "son"
    |  (3,1,`m`,2)  -> "ses"
    |  (3,1,`f`,1)  -> "sa"
    |  (3,1,`f`,2)  -> "ses"
    |  (1,2,`m`,1)  -> "notre"
    |  (1,2,`m`,2)  -> "nos"
    |  (1,2,`f`,1)  -> "notre"
    |  (1,2,`f`,2)  -> "nos"
    |  (2,2,`m`,1)  -> "votre"
    |  (2,2,`m`,2)  -> "vos"
    |  (2,2,`f`,1)  -> "votre"
    |  (2,2,`f`,2)  -> "vos"
    |  (3,2,`m`,1)  -> "leur"
    |  (3,2,`m`,2)  -> "leurs"
    |  (3,2,`f`,1)  -> "leur"
    |  (3,2,`f`,2)  -> "leurs"
```

```
let G_TITRE_FÉODAL = fun
      ('m',false)  -> "sieur"
    | ('m',true)   -> "sieur"
    | ('f",false)  -> "demoiselle"
    | ('f',true)   -> "dame"
```

```
let G_TERMINAISON = fun
      false  -> ""
    | true   -> "s"
```




Dans cette représentation les fonctions d'héritage et de synthèse n'apparaissent pas clairement.

```
let G_TITRE_PERSONNE (genre,marie,nombre) =
    G_POSSESSIF (1,1,genre,(if nombre>1) then 2 else 1)
    ^ G_TITRE_FEODAL(genre,marie)
    ^ G_TERMINAISON(nombre>1)
```

```
let G_TITRE_PERSONNE = fun e ->
    χ ( G_POSSESSIF( ϕA(e), G_TITRE_FEODAL( ϕB(e)) , G_TERMINAISON(ϕC(e)) )
where   ϕA = fun (g,m,n) -> (1,1,g,(if (n>1) then 2 else 1))
and     ϕB = fun (g,m,n) -> (g,m)
and     ϕC = fun (g,m,n) -> (n>1)
and     χ  = fun (s1,s2,s3) = s1 ^ s2 ^ s3
```

caml

fonction de synthèse de la variante

fonctions d'héritage des choix

La représentation ci-contre est équivalente mais fait ressortir chaque fonction. (^ est l'opérateur permettant de concaténer deux chaînes de caractères). Il s'agit donc d'un objet générique dérivé.

Remarquons au passage que les objets génériques définis ci-dessus sont des objets génériques par cas utilisant des constructions alternatives (ils sont de la forme $\mathcal{G} = \{ \mathcal{D} \times \mathcal{C} \}$)

## II.5.4 Synthèse

Dans cette section les problèmes relatifs aux objets génériques structurés ont été présentés. Tout d'abord, la distinction entre *variation détaillée* et *variation globale* a été faite. Dans le premier cas la structure des objets n'est pas utilisée et ceux-ci sont versionnés "en bloc". Cette solution plus simple à mettre en oeuvre présente néanmoins l'inconvénient de dupliquer l'information, ce qui donne lieu à des problèmes de maintenance multiple. Inversement un grain plus fin peut être choisi pour les variations ; ce qui rend les représentations plus difficiles à comprendre dans le cas

de nombreuses variations. Plus le niveau de granularité est fin plus ce phénomène s'accentue. On risque d'arriver à un point où les différents entités manipulées ne sont plus des entités logiques. On parle alors de *différences* plutot que de *variantes*, le terme *fragment* désignant l'une ou l'autre de ces deux possibilités.

Dans le cas de la variation détaillée, il s'agit de construire des objets génériques à partir de la composition d'autres objets génériques. Le problème consiste alors à assurer la *cohérence* entre les variantes générées pour chaque composant. Nous avons présenté succinctement le paradigme fonctionnel : le choix de chaque composant est déterminé à partir du choix appliqué à l'objet générique (*héritage des choix* et *synthèse de la variante*). Certains cas particuliers sont souvent retenus en pratique. Notamment dans le cas d'un *domaine universel*, tous les objets génériques ont le même domaine ; ceci permet de "synchroniser" le choix des variantes. Dans le cas le plus général les fonctions d'héritages sont intégrées à la représentation des objets génériques.

Le modèle présenté n'est qu'un mode de calcul particulier. Il a été retenu pour sa simplicité et car il est proche du paradigme impératif utilisé par les préprocesseurs (Chapitre IV).

# II.6 Opérations

Dans ce chapitre nous avons présenté tout d'abord des *abstractions*, *puis* des *représentations*. Dans la pratique informatique la démarche est plutôt inverse : on manipule des représentations et par la suite on essaie de découvrir quelles sont les abstractions correspondantes. Remarquons aussi que jusqu'à maintenant il n'a pas été fait référence explicitement aux *opérations*[1]. Pour comprendre pourquoi, il faut distinguer deux problèmes :

- *Conception* (approche descendante). Les méthodes de conception orientées objets proposent tout d'abord de définir avec précision des abstractions à partir des opérations applicables. Ensuite un processus (éventuellement incrémental) permet d'arriver à des représentations concrètes offrant les mêmes opérations[2]. Par exemple on peut choisir un ensemble comme abstraction, le définir à partir d'opérations, puis le représenter à partir d'un arbre binaire ou d'une liste, tout en veillant que les opérations réalisées soient compatibles avec celles annoncées.
- *Modélisation* (approche ascendante). La démarche est inverse. Des représentations concrètes complexes sont disponibles et de multiples opérations leur sont appliquées de manière plus ou moins empirique. Il s'agit alors de trouver a posteriori des abstractions et de déterminer quelles sont les opérations ayant un sens sur ces abstractions. La facilité avec laquelle les opérations peuvent être définies d'un point de vue abstrait indique la qualité de l'abstraction retenue.

Ici c'est la deuxième démarche qui est appliquée. Comme de nombreuses opérations sont définies sur les représentations, de multiples abstractions sont possibles. Généralement l'abstraction retenue permet de modéliser facilement l'usage courant qu'il est fait des représentations. Remarquons toutefois qu'*utiliser* des objets est une chose mais qu'en pratique il est aussi nécessaire de les *créer* et de les *maintenir*. Pour cela il faut pouvoir les *comprendre*, les *éditer*, les *analyser*, les *restructurer*, etc. Nous utiliserons le terme *manipuler* pour désigner l'une quelconque de ces activités.

Par la suite nous utiliserons les termes ***manipulation*** et ***opération*** pour faire ressortir l'idée suivante : la manipulation désigne une activité plus ou moins complexe mais celle-ci n'est finalement que la composition de différentes opérations (élémentaires).

Prenons par exemple le cas de la programmation détaillée. Une fonction est l'abstraction communément choisie pour un programme. Celle-ci est retenue car l'on s'intéresse essentiellement à l'exécution de ce programme, plus exactement aux résultats qu'il produit. D'autres abstractions sont possibles. Par exemple si l'on s'intéresse à l'analyse ou à la restructuration des programmes, les modéliser comme des graphes est utile. Quelque soit la modélisation choisie, une activité de manipulation se réduit finalement à l'application d'opérations.

1. Rappelons que dans cette thèse le terme "opération" est utilisé dans un sens général alors que le terme "opérateur" est une classe précise du modèle abstrait.
2. Une approche similaire est utilisée également avec les langages de spécifications.

## II.6.1 Opérations abstraites vs. opérations concrètes

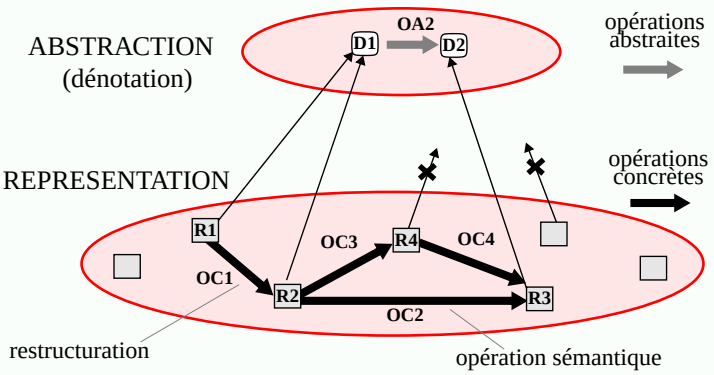
La dualité entre abstraction et représentation s'applique aussi aux opérations (figure 57). On distinguera :

- les ***opérations abstraites*** définies sur les abstractions (typiquement sur des fonctions).
- les ***opérations concrètes*** définies sur les représentations (typiquement sur des programmes).

Dans le cas d'un langage de programmation, plusieurs niveaux peuvent d'ailleurs être considérés et l'on peut distinguer les opérations textuelles, lexicales, syntaxiques et sémantiques. Ces niveaux correspondent aux propriétés vérifiées par les entités (figure 140 (p.293)). Une ***P-opération*** est une opération conservant la propriété P. Par exemple les éditeurs syntaxiques offrent des opérations syntaxiques.

*figure 57* Opérations abstraites et opérations concrètes

Cette figure est à comparer avec la figure 33 (p.95). D1 et D2 sont des dénotations, les $R_i$ sont des représentations. L'opération concrète OC1 est une opération de restructuration : elle ne change pas la dénotation. L'opération OC2 est une opération concrète sémantique correspondant à l'opération abstraite OA2. Sa sémantique est donc parfaitement définie. Par contre ce n'est pas le cas des opérations concrètes OC3 et OC4 qui considèrent une représentation sémantiquement incorrecte (bien que leur composée soit sémantique).




Dans le cas de la programmation détaillée les dénotations sont des fonctions et les représentations des programmes. L'opération OA2 pourrait être la restriction du domaine de la fonction D1. L'opération concrète correspondante OC2 est alors une opération de spécialisation de programme (Annexe A.3.9).

La relation entre les opérations concrètes et les opérations abstraites est loin d'être triviale. Des opérations abstraites simples peuvent correspondre à des opérations concrètes complexes. L'inverse est vrai aussi[1]. En fait la distance entre les abstractions et les représentations est directement liée à la simplicité de la sémantique du langage considéré.

Ci-dessous nous décrivons l'application de ses notions aux objets structurés, aux objets dérivés, et objets génériques. Ce dernier cas est étudié plus longuement.

## II.6.2 Opération et manipulation d'objets structurés

Les opérations définies sur les objets structurés dépendent des constructeurs. Il s'agit des différents constructeurs eux-mêmes, mais aussi des fonctions d'accès (par exemple les constructeurs de listes sont *[]* et *::* alors que *hd* et *tl* sont des fonctions d'accès) (Annexe A.1.9). Ces opérations dépendent des structures de données. Leur étude n'a pas lieu dans le cadre de cette

1. Le premier problème apparaît typiquement dans le cadre de la maintenance : les utilisateurs ne comprennent pas toujours pourquoi une modification mineure des *fonctionnalités* implique des travaux très importants sur la *représentation* du logiciel. Inversement le chargé de maintenance est souvent surpris de constater qu'une *opération concrète* aussi simple que la suppression d'un appel de procédure puisse avoir des répercutions aussi importantes sur la *sémantique* du programme.

thèse. Citons simplement deux opérations auxquelles il sera fait référence par la suite. Elles concernent justement les références (donc les objets composites) :

- ***Inclure***. Cette opération consiste à remplacer une référence par la valeur de l'objet référencé.
- ***Extraire***. Le passage inverse est appelé extraction. Il s'agit, pour une valeur donnée, de définir un nouvel objet ayant cette valeur et de la remplacer par la référence correspondante.

Bien que ces opérations ne soient définies formellement nous verrons qu'elles sont utiles dans un contexte de restructuration. Finalement remarquons que puisque les références amènent à la notion de relation, les différentes opérations définies sur cette abstraction sont utiles. Typiquement il s'agit de la fermeture transitive d'une relation (pour déterminer les objets dont dépend un objet) ou de l'inverse d'une relation (les objets qui dépendent directement de lui). Toutes ces opérations sont nécessaires pour faciliter la compréhension et plus généralement la manipulation d'objets structurés complexes. Elles sont à la base des langages d'interrogation et de manipulation d'objets structurés (l'exemple le plus typique correspond aux langages de manipulation associés au modèle relationnel).

### II.6.3 Opérations et manipulation d'objets dérivés et d'opérateurs

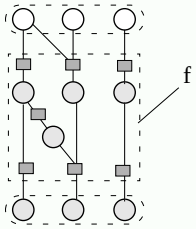
Les objets dérivés sont définis à partir d'opérateurs des fonctions. Leur représentation sont des (o)programmes. Bien évidemment l'application de fonction et l'interprétation d'un programme sont respectivement les opérations abstraites et concrètes les plus utilisées.

Bien que d'un point de vue abstrait les problèmes de performance n'existent pas, en pratique ils se révèlent essentiels. Le temps de calcul nécessaire à l'application d'un opérateur atomique ne peut pas être réduit (car par hypothèse celui-ci est atomique). Par contre dans le cas d'opérateurs construits on cherche souvent à limiter les calculs effectués au strict nécessaire[1]. C'est le cas lorsque le domaine ou le codomaine de l'opérateur est lui même construit et que les objets sources ou dérivés sont très complexes. Ce peut être aussi le cas lorsque la fonction réalisée est très complexe. Nous regrouperons ces problématiques sous le nom de ***calcul incrémental*** :

- ***Incrémentalité horizontale en entrée***. Considérons un opérateur prenant *en entrée* un objet *construit*. Normalement la manipulation d'un tel objet se fait de manière incrémentalle : seuls certains de ses composants sont modifiés à chaque étape. Après une modification partielle de l'entrée, seuls les objets dérivés "impactés" doivent être dérivés de nouveau (figure 58.a).
- ***Incrémentalité horizontale en sortie***. Certains opérateurs délivrent *en sortie* des objets structurés (ou plusieurs objets puisque c'est un cas particulier). Parfois on souhaite générer cette structure incrémentalement ; par exemple si l'on désire tester les composantes du résultat à des moments différents. On cherche alors à effectuer uniquement les calculs indispensables à la génération de ce résultat partiel (figure 58.b).
- ***Incrémentalité verticale***. Les opérateurs structurés sont souvent construits en composant séquentiellement d'autres opérateurs. Parfois il est souhaitable d'appliquer progressivement chaque étape ; par exemple pour tester ou utiliser les résultats intermédiaires.

1. Pour fixer les idées indiquons simplement à titre d'exemple que les opérateurs atomiques dans le cas de la programmation globale correspondent aux outils de dérivations alors que les opérateurs structurés correspondent aux makefiles ou fichiers de de commandes. Nous avons vu dans le Chapitre I à quel point il était important d'optimiser la manufacture.

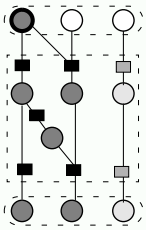
*figure 58* Incrémentalité horizontale en entrée et en sortie




Cette figure représente un graphe de dérivation. Les noeuds sont soit des opérateurs ( ), soit des objets dérivés ( ), soit des objets sources ( ). Les arcs représentent le flot de données Ils sont orientés (de haut en bas) mais les flèches ne sont pas représentées pour alléger la figure. La fonction f est un opérateur construit dont le domaine et le codomaine sont structurés.

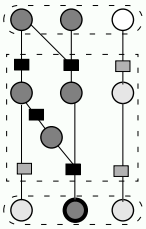
***(a)* Incrémentalité en entrée**

Si l'on suppose que seul le premier paramètre ( ) est modifié il suffit d'appliquer les opérateurs marqués ( ) pour calculer le nouveau résultat structuré.



***(b)* Incrémentalité en sortie**

Si l'on ne désire que le deuxième composant du résultat ( ) une optimisation consiste à n'appliquer que les opérateurs ( ).



Assurer l'incrémentalité en entrée ou en sortie peut se faire en utilisant des techniques de *découpage avant* ou *arrière*[1] (Annexe A.3.10, "Découpage"). Il s'agit grossièrement de calculer la fermeture transitive contenant le composant considéré en parcourant le graphe en avant ou en arrière selon le cas (d'où la terminologie). Ces techniques seront décrites plus longuement par la suite.

Pour l'incrémentalité en entrée on peut aussi utiliser la *mémorisation*, une technique développée dans le domaine des langages fonctionnels (Annexe A.3.6, "Mémorisation"). Schématiquement l'idée est d'éviter d'appliquer plusieurs fois la même fonction à un même paramètre en mémorisant les résultats déjà calculés dans une mémoire cache et en les réutilisant lorsque nécessaire. Dans le cas présent, si cette méthode est appliquée à chaque opérateur atomique, l'application d'un opérateur construit provoquera uniquement l'application des opérateurs nécessaires.

D'un point de vue concret l'incrémentalité en entrée est importante. Avec le modèle fonctionnel présenté ci-dessus cette opération concrète peut sembler simple. Pourtant elle est bien plus complexe à mettre en oeuvre avec d'autres représentations, notamment si les opérateurs sont représentés dans un langage impératif. Dans ce cas les effets de bords rendent impossible l'utilisation des techniques de mémorisation.

Quelque soit la complexité de l'opération concrète, l'opération abstraite est toujours la même : l'application de fonction. Il ne s'agit en effet que d'une opération d'optimisation.

L'opération abstraite permettant l'incrémentalité en sortie est en fait la composition séquentielle d'une opération de projection (typiquement notée $\pi$ dans les algèbres relationnelles) et de l'opérateur construit (il s'agit d'appliquer $\pi_2$ o f dans l'exemple de la figure 58.b). Calculer le résultat dans son intégralité et de ne sélectionner que la composante désirée est une opération concrète possible. Il ne s'agit pas là d'une optimisation! Ce n'est bien évidemment pas ainsi que l'opération concrète sera réalisée.

Une opération très légèrement différente est également utile. Plutot que d'optimiser l'*application* de la fonction $\pi_x$ o f, on peut vouloir générer un nouvel opérateur réalisant cette fonction. Bien

1. "Backward-slicing" et "forward-slicing" en anglais. Ces techniques sont regroupées sous le terme plus général "slicing".

évidemment on désire obtenir une représentation plus simple que la représentation originale de f. Cette opération, correspondant à la technique de découpage arrière, est largement utilisée dans le domaine de la programmation détaillée sur des p-programmes. Elle facilite les activités de tests, de parallèlisation, de mise-au-point, de compréhension, de maintenance et de rétro-ingéniérie car elle permet d'isoler les portions de programmes calculant chaque composante du résultat (chaque variable) (Annexe A.3.10, "Découpage")[1].

Plus généralement, bien que l'application des opérateurs soit l'opération primordiale dans notre cas, de très nombreuses autres opérations sont utiles pour développer et maintenir de tels objets. Une grande partie des concepts et outils valides pour les p-programmes pourrait être appliqués.

## II.6.4 Opérations abstraites pour les d'objets génériques

Tout comme pour les opérateurs, les fonctions sont les abstractions retenues pour les objets génériques (Section II.2.5). De la même manière, l'application de fonction est une opération indispensable (elle permet de générer une variante) pourtant elle n'est pas suffisante. Les besoins sont néanmoins quelque peu différents[2]. Cette section a pour but : (1) d'étudier quelles sont les opérations abstraites nécessaires ; (2) de montrer que celles-ci correspondent aux opérations habituellement définies sur les fonctions (inverse, ensemble image, surcharge, restriction, etc. (Annexe A.1.7, "Fonctions")) ; ce qui prouve l'adéquation de la modélisation choisie.
On se limitera dans cette section aux opérations abstraites ; les opérations concrètes ne seront décrites que dans la section suivante.

### II.6.4.1 *Génération de variante(s)*

L'application de fonction correspond à la génération d'une variante. L'image d'un ensemble par une fonction (notée (| |) ou img) correspond naturellement à la génération d'un ensemble de variantes. Le domaine et le codomaine (dom et rng) permettent d'obtenir respectivement l'ensemble des choix auxquels correspond une variante et l'ensemble de toutes les variantes qu'il est possible de générer. Ces ensembles peuvent être infinis.

TABLEAU 8 Génération de variante(s)

| Op. | Concept mathématique | Interprétation pour les objets génériques | |
|---|---|---|---|
| | fonction | Objet générique | $\mathcal{G} = \mathcal{D} \rightarrow \mathcal{C}$ |
| dom _ | domaine | Domaine de l'objet générique (choix) | dom $\in \mathcal{G} \rightarrow \mathcal{D}$ |
| rng _ | codomaine | Codomaine de l'objet générique (variantes) | rng $\in \mathcal{G} \rightarrow \mathcal{C}$ |
| _ _ | application de fonction | Génération d'une variante | _ _ $\in \mathcal{G} \times \mathcal{D} \rightarrow \mathcal{C}$ |
| img _ _ | ensemble image | Génération d'un ensemble de variantes | img $\in \mathcal{G} \times \{\mathcal{D}\} \rightarrow \{\mathcal{C}\}$ |
| $\_^{-1}$ | Association inverse | Ensemble des choix correspondant à une variante donnée | $^{-1} \in \mathcal{G} \rightarrow (\mathcal{D} \rightarrow \{\mathcal{C}\})$ |

1. Dans le chapitre suivant nous proposerons d'appliquer ces mêmes techniques dans le contexte de la manufacture du logiciel.
2. Généralement il n'est pas nécessaire d'optimiser l'interprétation d'un g-programme car comme nous l'avons vu souvent les objets génériques sont représentés en extension ou en tout cas n'utilisent pas des mécanismes complexes.

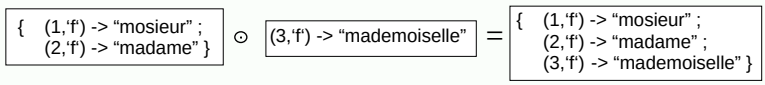
### II.6.4.2 *Extension ou modification d'objets génériques*

L'opération de base pour étendre un objet générique est l'insertion d'un couple (choix, variante). Cet opérateur est noté ⊙. Le résultat de cette opération n'est une fonction que si aucune variante ne correspond au choix spécifié (figure 59.a).

L'union de fonctions (∪) permet de regrouper deux objets génériques pour en former un nouveau (figure 59.b). Elle n'est définie que pour des fonctions ayant un domaine disjoint.

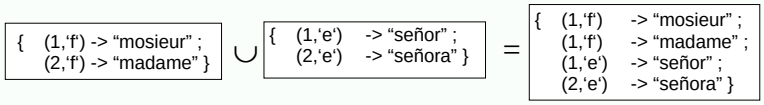
L'opérateur de surcharge, ⊕, permet de lever cette restriction : les "anciennes valeurs" sont écrasées. Cette opération est toujours définie et permet d'insérer de nouvelles variantes ou de modifier des variantes existantes (figure 59.c).

*figure 59* Extension et modification d'objets génériques



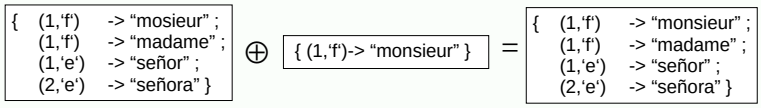

***a)* Insertion d'une variante pour un choix donné**

ans cet exemple une variante "mademoiselle" a été ajoutée pour le choix (3,'f'). On aurait pu utiliser l'union 'objet générique avec un singleton.




***(b)* Union d'objets génériques**

Les domaines des objets génériques doit être disjoints. Cette opération ne doit pas être confondue avec l'opération de fusion. Ici le résultat est un objet générique, pas une variante.




***(c)* Surcharge d'objets génériques**

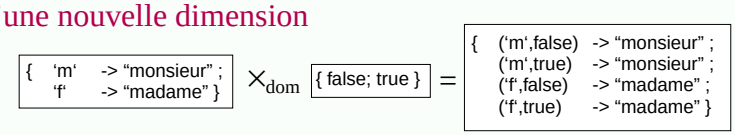
Le deuxième objet générique surcharge le premier. Dans l'exemple c'est une variante qui a été corrigée. Cet opérateur peut aussi être utilisé pour ajouter de nouvelles variantes

Le domaine d'un objet générique doit pouvoir évoluer. Il est souvent utile d'ajouter une nouvelle dimension pour modéliser une caractéristique qui n'avait pas été prise en compte auparavant (figure 60).

TABLEAU 9 Extension et modification d'un objet générique

| Op. | Concept mathématique | Interprétation pour les objets génériques | |
|---|---|---|---|
| _ ⊙ _ | insertion d'un élément dans une fonction | Insertion d'une (nouvelle) variante dans un objet générique | $\odot \in \mathcal{G} \times (\mathcal{D} \times \mathcal{C}) \rightarrow \mathcal{G}$ |
| _ ∪ _ | union de fonctions | Union d'objets génériques | $\cup \in \mathcal{G} \times \mathcal{G} \rightarrow \mathcal{G}$ |
| _ ⊕ _ | surcharge de fonction | Remplacement de variantes | $\oplus \in \mathcal{G} \times \mathcal{G} \rightarrow \mathcal{G}$ |
| _ $\times_{dom}$ | Produit cartésien sur le domaine | Ajout d'une dimension | $\times_{dom} \in (\mathcal{D} \rightarrow \mathcal{C}) \times \{X\} \rightarrow (\mathcal{D} \times X \rightarrow \mathcal{C})$<br>$(f \times_{dom} s)(x,x_1) = \text{if } x_1 \in s \text{ then } f(x) \text{ else undef}$ |

*figure 60* Ajout d'une nouvelle dimension

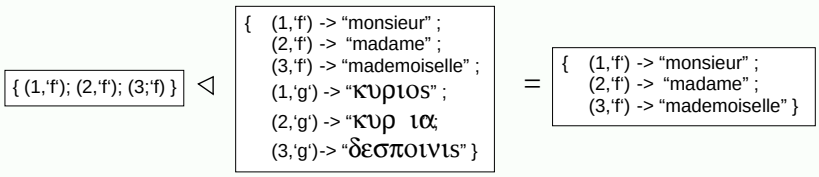



Dans cet exemple une dimension booléenne a été ajoutée. L'objet générique est "invariant" selon cette nouvelle dimension. Une telle opération est normalement suivie d'une opération de surcharge pour adapter les variantes à la nouvelle dimension. Ici l'intention est de créer une nouvelle dimension pour indiquer si la personne est mariée ou non, puis de surcharger l'objet générique avec la fonction {('m',false)->"mademoiselle"}. Cette technique est à la base du modèle de versionnement CoV qui propose de créer une dimension pour chaque modification [Munc93].

### II.6.4.3 *Spécialisation d'objets génériques*

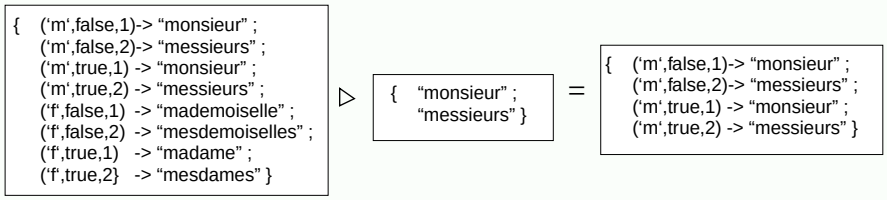
Dans cette section nous nous intéressons à quatre opérateurs définis sur les fonctions : les restrictions et suppressions de domaine et de codomaine (respectivement ◁, ▷, ◃ et ▹). En mathématique on parle de la restriction d'une fonction, qui correspond à la restriction de domaine. Lorsque ces techniques sont appliquées à des programmes plutôt qu'à des fonctions (i.e. à la représentation plutôt qu'à sa dénotation) on parle de spécialisation. Dans cette thèse nous utiliserons le terme "***spécialisation d'objet générique***" pour l'une quelconque de ces quatre opérations.

*figure 61* Spécialisation d'objets génériques




***(a)* Spécialisation du domaine d'un objet générique**

Il s'agit de restreindre l'objet générique aux choix spécifiés. Le même objet générique aurait pu être obtenu en donnant l'ensemble des choix à éliminer, ici {(1,'g'); (2,'g'); (3,'g')} et en utilisant l'opérateur de suppression de domaine (◃). La spécialisation de l'exemple peut avoir pour but de réutiliser l'objet générique en conjonction avec un opérateur (par exemple l'opérateur "Formate") ne fonctionnant que pour des caractères ascii.




***(b)* Spécialisation du codomaine d'un objet générique**

L'objet générique est restreint aux variantes spécifiées. On aurait pu donner l'ensemble {"madame"; "mademoiselle"} et utiliser l'opérateur de suppression de domaine (▹). Les opérateurs de spécialisations ne doivent pas être confondus avec l'image de la fonction et son inverse. Ici ce sont des objets génériques qui sont obtenus pas des variantes ou des ensembles de choix.

Cet exemple correspond au cas où l'on sait par une source ou par une autre que l'objet générique sera destiné à un ou à plusieurs hommes. Le choix de leur nombre est repoussé à une étape postérieure.

La spécialisation peut être utilisée pour :

- *Faciliter la réutilisation des objets génériques.*
  La généricité s'oppose généralement à l'efficacité et des compromis sont alors à faire. Dans le cadre de la réutilisation ce problème est bien connu. Pouvoir spécialiser un objet générique pour ne garder que les variantes réellement utilisées est un aspect important (figure 61.a).
- *Eliminer des variantes obsolètes.*

Tout au long du développement et de la maintenance d'un objet générique de nombreuses variantes sont insérées. Elles ne sont que rarement éliminées ; même si au cours du temps, certaines deviennent inutiles. C'est le cas par exemple si un événement extérieur rend un contexte obsolète (une langue n'est plus utilisée, l'usage d'un mot est obsolète, etc.). La spécialisation permet d'éliminer de telles variantes.

- *Générer incrémentalement une variante*.
  Dans certains contextes il est intéressant de déterminer le choix d'un variante, non pas en une seule étape mais plutôt incrémentalement. C'est le cas par exemple si la suite de décisions menant à un choix doit être prise par différentes personnes ou à différents moments (figure 61.b). Cet aspect est très important dans le domaine de la programmation globale (Section "Spécialisation incrémentale du logiciel" (p. 178)).
- *Simplifier la manipulation des objets génériques*.
  Un objet générique spécialisé est plus simple à comprendre que l'objet générique original. Sa manipulation en est d'autant simplifiée. C'est le cas par exemple de son édition. Par la suite il est possible d'intégrer ces modifications en revenant à l'objet générique original grâce aux opérations d'extension.

Certaines dimensions peuvent devenir inutiles, notamment après la spécialisation d'un objet générique. Une dimension inutile est une dimension selon laquelle l'objet générique est invariant. Une telle dimension peut être éliminée via la projection du domaine sur les autres dimensions. Le résultat de cette projection est alors une fonction (sinon c'est une association et, à au moins un choix, correspond plusieurs variantes). A l'extrême un objet générique peut être transformé en un objet si celui-ci est invariant pour toutes ses dimensions (i.e. s'il s'agit d'une fonction constante sur son domaine) (figure 62).

*figure 62* Suppression d'une dimension

$\pi_{dom,3}$ { ('m',false,1)-> "monsieur" ; ('m',false,2)-> "messieurs" ; ('m',true,1) -> "monsieur" ; ('m',true,2) -> "messieurs" } = { 1 -> "monsieur" ; 2 -> "messieurs" }

Après la spécialisation de l'objet générique présentée en (b) deux dimensions deviennent inutiles (la 1 et la 2). Une projection du domaine sur la troisième dimension permet de les éliminer. Le résultat est une fonction (un objet générique).

TABLEAU 10 Spécialisation d'objets génériques

| Op. | Concept mathématique | Interprétation pour les objets génériques | |
|---|---|---|---|
| $\_ \triangleleft \_$ | restriction de domaine | Spécialisation d'un objet générique via la sélection d'un ensemble de choix | $\triangleleft \in \{\mathcal{D}\} \times \mathcal{G} \rightarrow \mathcal{G}$ |
| $\_ \blacktriangleleft \_$ | suppression de domaine | Spécialisation d'un objet générique via l'élimination d'un ensemble de choix | $\blacktriangleleft \in \{\mathcal{D}\} \times \mathcal{G} \rightarrow \mathcal{G}$ |
| $\_ \triangleright \_$ | restriction de codomaine | Spécialisation d'un objet générique via la sélection d'un ensemble de variantes | $\triangleright \in \mathcal{G} \times \{\mathcal{C}\} \rightarrow \mathcal{G}$ |
| $\_ \blacktriangleright \_$ | suppression de codomaine | Spécialisation d'un objet générique via l'élimination d'un ensemble de variantes | $\blacktriangleright \in \mathcal{G} \times \{\mathcal{C}\} \rightarrow \mathcal{G}$ |
| $\pi_{dom,1}$ | projection du domaine | Suppression d'une dimension | $\pi_{dom,1} \in (\mathcal{D}_1 \times \mathcal{D}_2 \rightarrow \mathcal{C}) \rightarrow (\mathcal{D}_2 \rightarrow \mathcal{C})$ |

### II.6.4.4 *Construction d'objets génériques structurés*

Dans les sections antérieures nous avons vu différentes manières de construire des objets génériques structurés. Soulignons seulement ici le fait que les opérateurs d'application duale # et parallèle || peuvent être utilisés pour construire des objets génériques tuples à domaine global et à domaine composé respectivement (Voir les exemples de la figure 55.b (page 115) et figure 55.c (page 115)). Les itérateurs traditionnellement définis dans les langages fonctionnels sont également utilisables pour des listes (map, it_list, etc.).

TABLEAU 11 Constructions d'objets génériques structurés

| Op. | Concept mathématique | Interprétation pour les objets génériques | |
|---|---|---|---|
| _ \|\| _ | Application parallèle | Objet générique construit (couple) à domaine composé | $\|\| \in \mathcal{G} \times \mathcal{G} \rightarrow \mathcal{G}$ |
| _ # _ | Application duale | Objet générique construit (couple) à domaine global | $\# \in \mathcal{G} \times \mathcal{G} \rightarrow \mathcal{G}$ |

## II.6.5 Opérations concrètes pour les objets génériques

Etudions maintenant les opérations concrètes ; celles définies sur des représentations et non pas des abstractions.
Les exemples ci-dessus peuvent laisser croire que les opérations concrètes sont simples ; il n'en est rien. Cette impression n'est due qu'à la simplicité des représentations choisies dans les exemples (des objets génériques par cas). D'autres représentations, par exemple celles utilisées avec les préprocesseurs, peuvent rendre les opérations si complexes qu'elles ne sont pas réalisées en pratique (Chapitre IV).
Vu la multitude des représentations envisageables, il n'est pas possible de présenter les opérations concrètes correspondant à chaque cas. Nous nous contenterons donc ci-dessous de donner un exemple représentatif.

### II.6.5.1 *Exemples d'opérations concrètes*

Supposons que l'on dispose d'un objet générique **G** comportant deux variantes "madame" et "monsieur" et que l'on souhaite associer la variante "mademoiselle" au cas d'une jeune fille non mariée. Soit $O^a$ cette opération abstraite et **G'** l'objet générique résultant. On a :

$$\mathbf{G'} = \mathbf{G} \oplus \{ (\text{'f'},\text{false}) \text{ -> "mademoiselle" } \}$$

Le problème est de trouver une opération concrète $O_c$ pour une représentation donnée. En fait nous allons étudier ce problème pour 4 représentations notées $\mathbf{G}_1$, $\mathbf{G}_2$, $\mathbf{G}_3$ et $\mathbf{G}_4$ (figure 63).

La surcharge $\oplus$ est équivallente à la suppression de domaine $\triangleleft$ puis à l'union $\cup$. Autrement dit remplacer une variante consiste à l'effacer puis à l'insérer. Plus formellement on a :

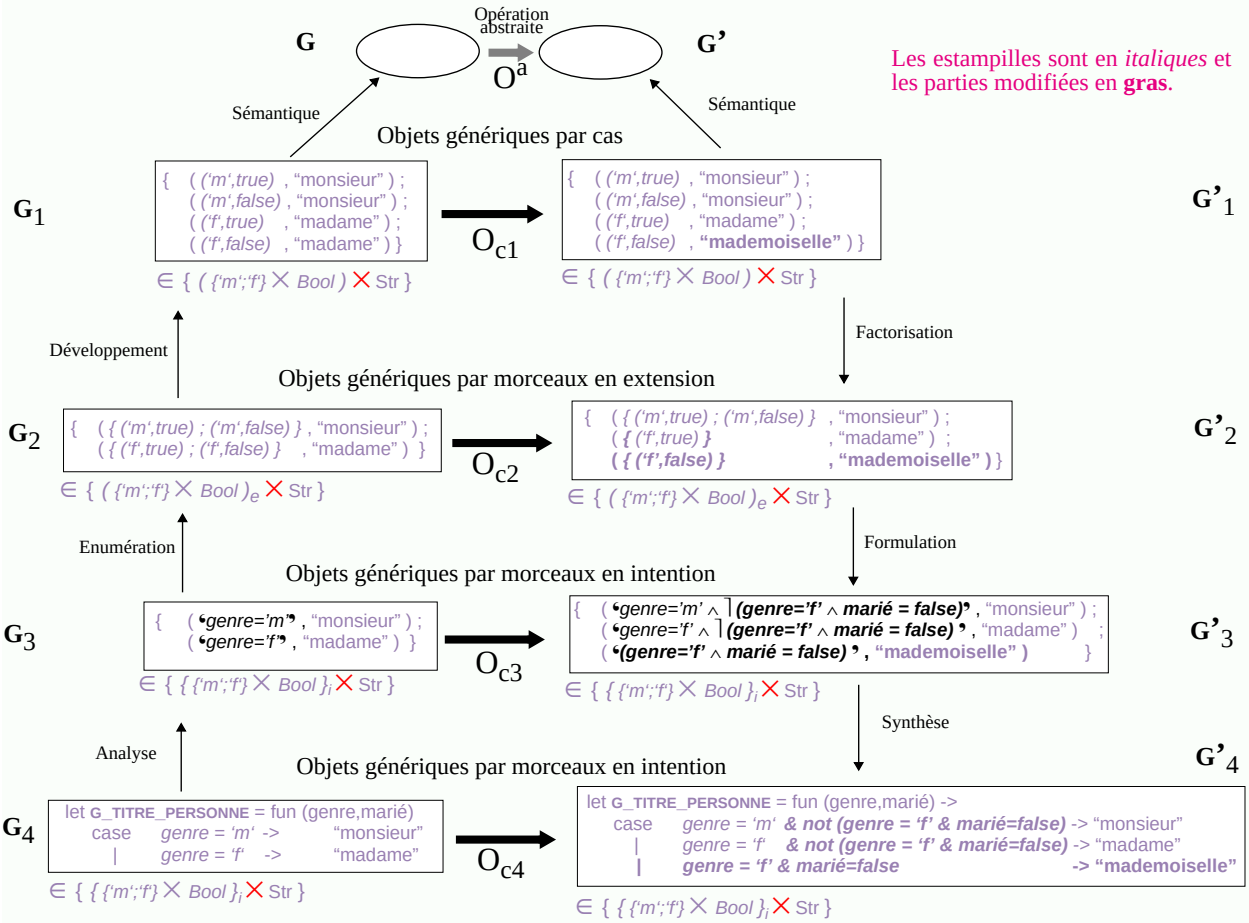
$$f \oplus g = ((\text{dom } g) \triangleleft f) \cup g$$

Ce résultat théorique peut être appliqué en pratique pour décomposer les opérations concrètes en des opérations plus simples. Il s'agit alors de déterminer les opérations concrètes pour (a) la suppression $\triangleleft$ et (b) l'union. Plus précisément on cherche à appliquer la relation :

$$\mathbf{G'}_i = \underbrace{(\{(\text{'f'},\text{false})\} \triangleleft \mathbf{G}_i)}_{(a)} \underbrace{\cup \{ (\text{'f'},\text{false}) \text{ -> "mademoiselle" } \}}_{(b)}$$

*figure 63* Exemples d'opérations concrètes de surcharge (⊕)

Les différentes représentations $G_i$ correspondent toutes à la même dénotation. On peut supposer par exemple que $G_1$, $G_2$ et $G_3$ sont représentées sous forme de structures de données alors que $G_4$ est le texte source d'un programme interprété (i.e. une suite de caractères). Plus précisément $G_1$ pourrait utiliser une base de donnée relationnelle. Un modèle relationnel imbriqué est nécessaire pour $G_2$.




Les opérations permettant de "passer" d'une représentation à l'autre ont déjà été étudiées (analyse, énumération, développement, etc). $G_1$ est la représentation la plus "proche" de la dénotation, $G_4$ la plus éloignée. Cette notion de proximité est liée à la complexité des opérations $O_{ci}$ par rapport à $O^a$.

La figure 63 présente la transformation dans son intégralité. Ceci dit, pour simplifier, commentons uniquement l'opération abstraite (a) qui consiste supprimer la variante associée à ('f',false). L'opération concrète correspondante dépend évidemment de la représentation choisie. En simplifiant :

- $G_1$ , *Objet générique par cas.*
  Il suffit de supprimer le couple dont l'estampille est ('f',false).
- $G_2$ , *Objet générique par morceaux en extension.*
  Chaque estampille $e_i$ est remplacée par $e_i \setminus \{('f',false)\}$.
- $G_3$ , *Objet générique par morceaux en intention.*
  Chaque estampille $e_i$ est remplacée par $e_i \wedge \rceil(genre='f' \wedge marié = false)$.

Il serait trop long de rentrer dans les détails. Signalons simplement que *les différentes opérations concrètes peuvent être dérivées systématiquement en utilisant les équivalences entre représentations*. Leur validité peut d'ailleurs être montrée par les même moyens.

Ces équivalences n'ont pas seulement une importance théorique, elles peuvent aussi servir pour implémenter les opérations concrètes. C'est d'ailleurs ce qui est fait dans le domaine des bases de données temporelles (voir par exemple [Lore93]).

#### II.6.5.2 *Opérations de restructuration*

Les opérations de restructuration n'ont pas encore été étudiées car sémantiquement elle correspondent à l'identité. Leur importance pratique est par contre très importante. Par exemple simplifier les estampilles de $\mathbf{G'_4}$ est une opération fort utile : genre = 'm' & not (genre = 'f' & marié=false) peut être changé en genre = 'm' . De telles restructurations sont bien évidemment basées sur la connaissance précise des propriétés des représentations manipulées.

### II.6.6 Synthèse

Dans cette section nous nous sommes intéressés aux *opérations* définies sur les objets, tant d'un point de vue *abstrait* que d'un point de vue *concret*. La démarche poursuivie est une démarche de *modélisation* et il s'agit donc de définir a posteriori quelles sont les opérations élémentaires traditionnellement appliquées aux représentations. La sémantique de ces opérations a été donnée. *La facilité avec laquelle ces opérations abstraites ont été décrites témoigne de la qualité des abstractions choisies*.

Cette démarche a été principalement appliquée au cas des objets génériques. Modéliser de tels objets par des fonctions apparaît satisfaisant non seulement pour décrire la génération d'une variante, mais aussi pour toutes les autres opérations présentées. Il s'avère en effet qu'en énumérant les opérations classiques sur les fonctions, on retrouve les différentes opérations de manipulation d'objet générique.

Finalement nous avons vu que pour une opération abstraite donnée, il existait autant d'opérations concrètes que de types de représentations. *Connaître les équivalences entre ces représentations permet de déduire les opérations concrètes, ce qui est important autant d'un point de vue théorique que pratique.*

# II.7 Un exemple

Dans cette section, un exemple complet est présenté. Il s'agit en fait d'un scénario mettant en jeu la conception, le développement et la maintenance de quelques objets génériques.

Le but de ce scénario est de montrer :

- la déstructuration liée à la maintenance,
- l'utilité d'outils permettant de limiter a priori ou a posteriori cette déstructuration.

Cet exemple n'est pas réaliste. Seuls quelques objets génériques et chaînes de caractères sont utilisés. Néanmoins sa simplicité le rend particulièrement facile à comprendre et nous pensons qu'il est représentatif de bon nombre de problèmes rencontrés dans le cadre de la maintenance du logiciel. Ce même exemple est d'ailleurs repris dans le Chapitre IV en utilisant le préprocesseur CPP.

Ce scénario n'a pas de sens en soit et il ne faut pas l'interpréter "à la lettre"... Au contraire le lecteur est invité à imaginer que chaque lettre est une procédure complexe et qu'elle représente des dizaines voir des centaines de lignes. Dans ces conditions la complexité du problème est plus évidente, tout comme le besoin d'éviter la duplication d'informations.

Ci-dessous les différentes étapes du scénario sont présentées.
Le développement et la réutilisation de quelques objets génériques sont illustrés dans la Section II.7.1, "Développement". Les sections suivantes traitent de l'évolution de ce système. Un objet composant est tout d'abord adapté à un nouveau contexte (Section II.7.2), puis intégré (Section II.7.3). Un défaut est détecté lors des tests de non régression. Ce défaut remet en cause la structure initiale du système. Plusieurs corrections possibles sont analysées dans la Section II.7.5, "Correction de l'anomalie". Celles-ci vont du rapiéçage "vite fait mal fait" à des solutions plus élaborées. Finalement, une autre modification est apportée au système. Celle-ci remet en cause celle faite auparavant (Section II.7.6). Dans tous les cas le système tend à se déstructurer.

## II.7.1 Développement

Bien souvent le développement d'un objet générique n'intervient qu'après avoir constaté qu'un objet spécifique existant n'est pas suffisamment général pour être utilisé dans de nouveaux contextes. Nous présentons ici (1) le problème spécifique initial accompagné d'une solution spécifique, (2) l'apparition de nouveaux contextes, et finalement (3) la conception et la réalisation d'objets génériques permettant de répondre à ces variations.

### II.7.1.1 *Un problème et une solution spécifique*

Le problème initial est le suivant. Une entreprise doit élaborer une lettre pour un client donné, monsieur Dupond. Cette lettre est le résultat de l'application d'un opérateur, disons "Formate" à une liste de chaînes de caractères. Il s'agit donc d'un objet dérivé, disons "*d_lettre_dupond*". L'objet source est "lettre_dupond" ; il a été réalisé manuellement.

### II.7.1.2 *Variation du contexte d'utilisation*

Après un certain temps, de nouveaux clients potentiels apparaissent. Pour cette entreprise il est vital de pouvoir adapter l'objet existant à ces variations. Le destinataire de la lettre est maintenant un paramètre du problème. Il est décidé de développer un objet générique dérivé ***GD_LETTRE***.

### II.7.1.3 *Conception d'un objet générique*

#### *Analyse des besoins...*

La première étape consiste à analyser les besoins. Evidemment on ne se limite pas aux besoins des clients actuels mais l'on essaie au contraire de prévoir les évolutions futures afin d'anticiper les problèmes d'adaptation et de pouvoir étendre le marché potentiel. Il est décidé de paramétrer la lettre par le nom du destinataire. Il ressort également que le titre "monsieur" figurant explicitement dans la lettre originale n'est pas adapté aux clientes. On préfère donc l'extraire et développer un objet générique **G_TITRE_PERSONNE** pouvant être réutilisé dans d'autres applications.

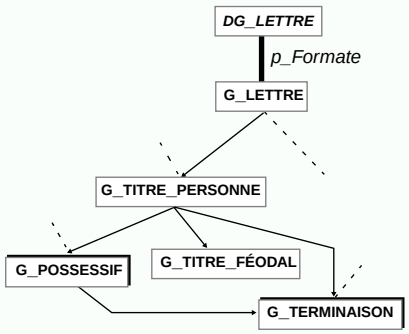
#### *Architecture générale...*

En réalité tout au long du projet réutiliser et rendre réutilisable sont des préoccupations. Lors de la conception il apparaît qu'une bonne solution est de décomposer **G_TITRE_PERSONNE** en trois objets génériques[1]. D'autant plus que *grâce à des outils facilitant la réutilisation, on s'est aperçu que deux de ces objets existaient déjà et pouvaient être réutilisés*. Un seul doit donc être développé.

- **G_POSSESSIF.** ("mon","ma","mes", "ton", "ta", "tes", etc.). Cet objet générique a déjà été développé dans un autre projet. Comme nous le verrons par la suite il utilise **G_TERMINAISON**.
- **G_TITRE_FÉODAL.** ("sieur","dame","demoiselle"). Cet objet doit être développé mais il est déjà prévu de le réutiliser dans une autre application.
- **G_TERMINAISON**. ("","s"). Cet objet générique a été développé il y a longtemps. Il s'agit de la terminaison de mots au singulier ou au pluriel. Même si l'on sait qu'il n'est pas adapté à tous les contextes[2] il est utilisé dans de nombreux autres projets.

La figure 64 présente l'architecture du système proposé.

*figure 64* Architecture




Seul un fragment de l'architecture est présenté. Les lignes en pointillés indiquent la présence de relations allant vers des noeuds non représentés dans la figure.

**G_TITRE_PERSONNE** et **G_LETTRE** sont des objets génériques composites car ils contiennent des références. C'est le cas également de **G_POSSESSIF** qui utilise la terminaison (c'est en tout cas le cas de l'implémentation existante).

La relation existant entre ***DG_LETTRE*** et **G_LETTRE** est une relation de dérivation. Elle est décorée par les opérateurs utilisés, ici l'opérateur paramétré *p_Formate* (voir la section suivante).

Les objets réutilisés sont dessinés en reliefs.

1. Cette conception a déjà été présentée dans la figure 56 (p.116) mais sans être justifiée.
2. Un cheval, des chevaux...

#### *Variation des objets dérivés...*

Les variations ne sont pas traitées uniquement au niveau des objets sources. C'est d'un objet générique dérivé dont on a besoin, disons de ***GD_LETTRE***. Plutot que définir celui-ci à partir de l'application directe de l'opérateur *Formate*, il est apparu préférable de paramétrer cet opérateur afin d'obtenir différents formats (par exemple la taille du papier). L'objectif bien évidemment est encore de pouvoir satisfaire une plus grande gamme de clients.

#### *Détermination des domaines...*

Lors de la conception, choisir le domaine de chaque objet générique est fondamental. Il s'agit de déterminer le nombre et la signification de chaque dimension. La manière de les "coder" est laissée à l'étape d'implémentation puisque ceci dépend des possibilités offertes par le langage utilisé. Remarquons que le domaine des objets réutilisés a déjà été déterminé antérieurement ; éventuellement avec des conventions quelque peu différentes.

- *Domaine de* **G_POSSESSIF**. Un possessif dépend du numéro de la personne et de son groupe ; du genre et du nombre d'objets désignés. Par exemple la variante "ta" correspond à la $2^{ème}$ personne du $1^{er}$ groupe et à un seul objet de genre féminin.
- *Domaine de* **G_TITRE_FÉODAL**. Deux dimensions sont associées au titre féodal : le genre de la personne et le fait qu'elle soit mariée ou non.
- *Domaine de* **G_TERMINAISON**. La solution simplificatrice retenue lors de la conception de "g_terminaison" était de n'indiquer que s'il s'agissait d'un pluriel ou non. Cette solution étant suffisante pour l'application visée, cet objet est réutilisé tel quel[1].

Finalement, et c'est le plus important, il faut définir le domaine de "g_titre_personne". Les conventions prises doivent être adaptées à l'application visée.

- *Domaine de* **G_TITRE_PERSONNE**. Le titre d'une ou plusieurs personnes dépend du nombre de personnes, du genre de ces personnes, et du fait qu'elles soient mariées ou non. Remarquons que la dimension correspondant au sujet n'apparaît pas car seule la première personne du premier groupe est utilisée (i.e. "mon", "ma" et "mes").

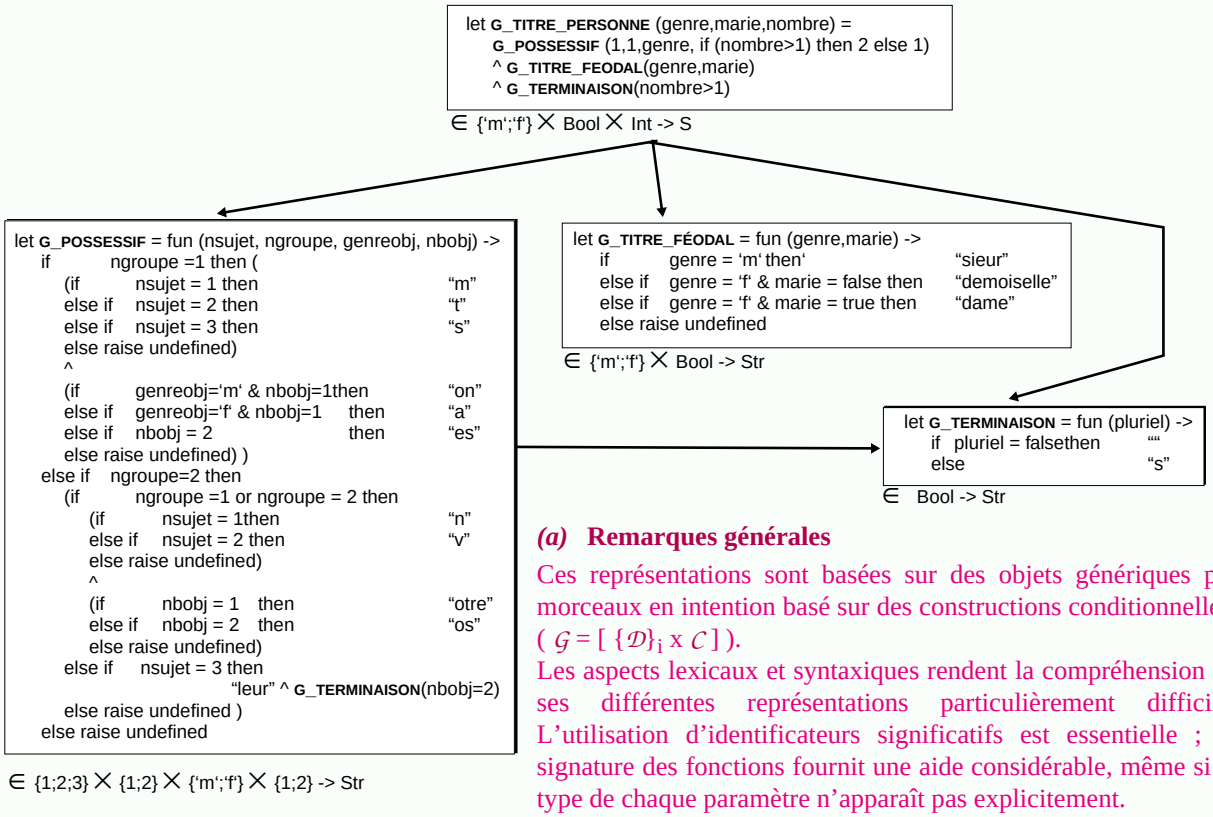
### II.7.1.4 *Réalisation de l'objet générique*

Lors de la phase d'implémentation il est nécessaire de choisir une représentation précise pour les objets génériques et les éléments de leurs domaines. Le choix de cette représentation est fondamental car il détermine la facilité de manipulation de ces objets ; plus particulièrement la complexité des opérations concrètes. Il faut non seulement veiller à la simplicité des solutions, mais aussi éviter la duplication d'information pour éviter les problèmes de maintenance multiple.

Les représentations de **G_POSSESSIF** et de **G_TERMINAISON** sont disponibles. Pour les réutiliser, il est essentiel de comprendre leur fonctionnement ou tout au moins les conventions prises pour structurer leur domaine. *Des outils facilitant la compréhension d'objets générique peuvent alors être fort utiles*.
**G_TITRE_FÉODAL** et **G_TITRE_PERSONNE** sont développés dans le même langage (figure 65).

1. Dans le cas du logiciel, on trouve souvent des bibliothèques de ce style : vielles, très utilisées et très difficile à changer.

*figure 65* Implémentation




### *(a)* Remarques générales

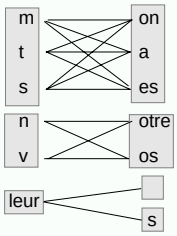
Ces représentations sont basées sur des objets génériques par morceaux en intention basé sur des constructions conditionnelles. ( $\mathcal{G} = [\ \{\mathcal{D}\}_i \times \mathcal{C}\ ]$ ).

Les aspects lexicaux et syntaxiques rendent la compréhension de ses différentes représentations particulièrement difficile. L'utilisation d'identificateurs significatifs est essentielle ; la signature des fonctions fournit une aide considérable, même si le type de chaque paramètre n'apparaît pas explicitement.

Dans le cas d'un langage comme ML les types des paramètres peuvent être inférés automatiquement via une analyse. Si ce service n'est pas assuré par le langage (c'est le cas des préprocesseurs (Chapitre IV)) des *outils sont nécessaires*. Notons finalement que si les différentes représentations n'étaient pas reliées par des flèches, connaître l'architecture de ce système simple constituerait aussi une difficulté. Là aussi une aide serait la bienvenue.

### *(b)* G_POSSESSIF

La représentation de **G_POSSESSIF** minimise la duplication d'informations (la comparer avec la figure 56 (p.116) sémantiquement équivallente). Cette représentation est basée sur une variation détaillée ; les différents fragments sont des différences et non pas des variantes (aucun sens n'est associé à "m" ou à "n" isolément).




Comprendre l'imbrication des instructions conditionnelles n'est pas chose facile sans documents de conception. Le graphe présenté à droite permet de visualiser la structure de cet objet générique. Hélas un tel document n'est pas forcément disponible lors du développement du système. *Des outils de rétro-ingénierie et des outils de visualisation sont importants.*

### *(c)* Une représentation plus explicite pour G_POSSESSIF

Cette représentation est équivallente mais elle explicite le fait que l'objet générique **G_POSSESSIF** est construit à partir de différentes valeurs génériques (représentées dans des cadres gris). Comme a priori celles-ci ne peuvent pas être réutilisées pour d'autres applications le concepteur a préféré les inclure directement. Il aurait pu au contraire définir 5 objets génériques et construire **G_POSSESSIF** à partir de références (c'est encore possible grâce à l'opération *extraire*). Cette solution aurait rendu inutilement complexe l'architecture générale du système. Il s'agit ici typiquement d'un problème de portée et d'encapsulation.

Les fonctions de transfert se déduisent directement des informations écrites en gras.

```
let G_POSSESSIF = fun (nsujet, ngroupe, genreobj, nbobj) ->
  ( fun 1 -> (
      ( fun 1 -> "m"
        | 2 -> "t"
        | 3 -> "s" ) (nsujet)
      ^
      ( fun ('m',1)          -> "on"
        | ('f',1)            -> "a"
        | (('m',1) | ('f',2)) -> "es" ) (genreobj,nbobj) )
    | 2 -> (
      ( fun (1|2) -> (
          ( fun 1 -> "n"
            | 2   -> "v" ) nsujet
          ^
          ( fun 1 -> "otre"
            | 2   -> "os" ) nbobj ) nsujet )
    | 3 -> "leur" ^ G_TERMINAISON (nbobj=2)
  ) (ngroupe)
∈ {1;2;3} X {1;2} X {'m';'f'} X {1;2} -> Str
```

Après avoir été développé et testé, le système donne entière satisfaction à l'entreprise. L'objet générique "**G_TITRE_FÉODAL**" a été réutilisé comme prévu dans un autre projet.

## II.7.2 Extension

Quelques années plus tard, dans un autre projet, il est nécessaire d'étendre **G_TITRE_FÉODAL** pour prendre en compte le fait qu'une personne soit noble ou non. Cette possibilité est aussi devenue indispensable pour pouvoir adresser des lettres à de nouveaux clients nobles. La dimension "noble" est rajoutée et le titre féodal "sire" est associé aux hommes nobles, mariés ou non. D'un point de vue sémantique la nouvelle valeur devient :

$$\underbrace{(\textbf{G\_TITRE\_FÉODAL} \times_{dom} \text{Bool})}_{\text{Ajout de la dimension}} \oplus \underbrace{\{ \text{❛ genre = 'm' } \wedge \text{ noble = true ❜ -> "sire" } \}}_{\text{Surcharge pour les hommes nobles}}$$

*L'opération concrète pourrait être exécutée automatiquement à l'aide d'un outil d'édition sémantique*. Dans la figure 66, cette opération a été réalisée en deux étapes.

*figure 66* Extension de l'objet générique **G_TITRE_FÉODAL**.

```
let G_TITRE_FÉODAL = fun (genre,marie, noble <- false ) ->
    if        genre = 'm' then          "sieur"
    else if   genre = 'f' & marie = false then   "demoiselle"
    else if   genre = 'f' & marie = true then    "dame"
    else raise undefined
```

∈ {'m';'f'} × Bool × **Bool** -> Str

***(d)* Ajout de la dimension "noble"**

Dans le langage utilisé ajouter une dimension consiste seulement à changer la signature de la fonction. Dans le cas du préprocesseur CPP aucune modification n'est nécessaire (Chapitre IV)! On suppose ici qu'une valeur par défaut peut être associée à un paramètre (c'est aussi le cas par exemple dans le langage ADA). Dans cet exemple, par défaut, la personne est supposée ne pas être noble.

```
let G_TITRE_FÉODAL = fun (genre,marie, noble<-false) ->
    if        genre = 'm' & noble = false then  "sieur"
    else if   genre ='m' & noble = true then    "sire"
    else if   genre = 'f' & marie = false then  "demoiselle"
    else if   genre = 'f' & marie = true then   "dame"
    else raise undefined
```

∈ {'m';'f'} × Bool × Bool -> Str

***(e)* Surcharge de l'objet générique**

L'objet générique est ensuite surchargé. On utilise donc une opération concrète de surcharge (Section II.6.5.1). Automatiser cette tâche consiste à :
(1) remplacer les estampilles $e_i$ correspondant à chaque fragment $f_i$ par "$e_i$ & not P" où P est le prédicat genre='m' ∧noble = true.
(2) ajouter le couple (P,"sire").
Les estampilles ont ensuite été simplifiées.

L'objet **G_TITRE_FÉODAL** est testé en *isolation* et aucun défaut n'est révélé.

## II.7.3 Intégration et tests

Après avoir modifié **G_TITRE_FÉODAL** isolément, il faut l'intégrer à tous les systèmes qui l'utilisent, puis tester ceux-ci.

### *Intégration...*

Deux solutions sont possibles pour intégrer **G_TITRE_FÉODAL** dans un objet générique l'utilisant :

- Il n'est pas intéressant de tirer profit de la nouvelle dimension.
  Autrement dit le domaine de l'objet générique reste le même. Si la valeur "faux" a été définie comme paramètre par défaut, aucune modification ne doit être faite. Sinon pour que la sémantique soit inchangée il faut rajouter cette valeur comme paramètre à chaque appel de **G_TITRE_FÉODAL**.

- On souhaite tirer profit de cette nouvelle possibilité.
  La dimension noble est alors rajoutée à l'objet générique client. C'est la solution qui a été retenue pour ***DG_LETTRE*** et a posteriori **G_LETTRE** puis **G_TITRE_PERSONNE**. La figure 67 présente le résultat obtenu dans ce dernier cas.

*figure 67* Mise à jour de **G_TITRE_PERSONNE**.

```
let G_TITRE_PERSONNE = fun (genre,marie,nombre, noble)->
    G_POSSESSIF (1,1,genre,(if nombre>1 then 2 else 1)
    ^ G_TITRE_FEODAL(genre,marie, noble)
    ^ G_TERMINAISON(nombre>1)
∈ {'m';'f'} × Bool × Int × Bool
```

La dimension "noble" est ajoutée à l'objet générique "g_titre_personne". Ici la fonction d'héritage se réduit à la fonction identité.

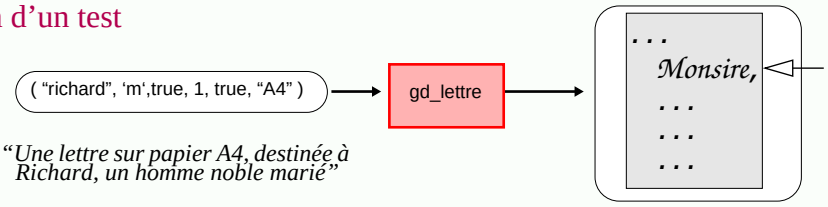
Remarquons que pour ne pas introduire d'incohérence, il est indispensable de connaître les objets qui dépendent de **G_TITRE_FÉODAL**. Si l'on dispose d'un grand nombre d'objets il n'est pas possible de tous les examiner. Un document d'architecture est alors fort utile. Hélas celui-ci n'existe pas toujours ou n'a pas été systématiquement mis à jour au cours d'années de maintenance. *Un outil permettant d'extraire la relation de dépendance à partir des différents objets est alors indispensable*. D'ailleurs avant toute modification, un chargé de maintenance averti aurait utilisé un *outil d'analyse d'impact* pour connaître l'ampleur des modifications à apporter.

#### *Tests...*

Des tests de non régression doivent ensuite être effectués sur chaque système impacté. Connaître la structure du domaine de chaque objet générique est essentiel pour élaborer des jeux des tests, calculer le taux de couverture, etc. *L'utilisation d'outils de tests peut apporter de nettes améliorations*.

***DG_LETTRE*** est testé directement. Concrètement cette tâche consiste à examiner les lettres produites par le système pour différents choix. La figure 68 présente l'application de cet objet générique à un jeu de tests particulier.

*figure 68* L'exécution d'un test




Une anomalie est détectée car la lettre contient le mot "monsire" ! Celui-ci n'existe pas en français, ce devrait être "messire"...

### II.7.4 Localisation de l'anomalie

Il est alors nécessaire d'identifier la source de l'anomalie. L'erreur ayant été détectée sur un objet dérivé, il peut s'agir d'un dysfonctionnement de l'opérateur ***P_FORMATE***. Cette possibilité est rejetée et l'on remet directement en cause l'objet générique **G_LETTRE**.
Reste tout de même à localiser plus précisément l'anomalie au sein de cet objet générique. Un composant de la lettre est anormal (le mot "monsire" ou plus précisément les lettres "on").

Pour corriger l'erreur il faut remonter à l'objet générique source. Cette tâche est grandement simplifiée si, lors de la génération puis de la manufacture, une ***trace*** a été automatiquement conservée et si des outils permettent de l'exploiter. Plus généralement *des outils d'analyse dynamique peuvent se révéler utile*s.
Le chargé de maintenance découvre que le problème vient du fait que **G_POSSESSIF** génère la variante "mon" à la place de la variante "mes".

## II.7.5 Correction de l'anomalie

De nombreuses solutions sont possibles pour corriger cette anomalie. Elles diffèrent sur la qualité de l'analyse effectuée, sur leur facilité de mise-en-oeuvre et sur la déstructuration qu'elles impliquent.

### II.7.5.1 *Solution (1) : correction via un rapiéçage de G_POSSESSIF*

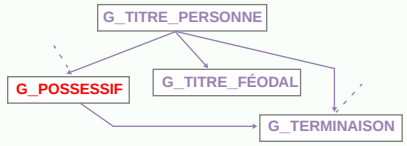
La solution la plus "brutale" consiste à modifier directement la ligne provoquant l'erreur. On ne cherche pas à comprendre dans le détail la logique de l'objet générique, ni à préserver une conception particulière (figure 69).

*figure 69* Rapiéçage de **G_POSSESSIF**.

```
let G_POSSESSIF = fun (nsujet, ngroupe, genreobj, nbobj, noble) ->
    if          ngroupe =1 then (
        (if         nsujet = 1 then                         "m"
        else if     nsujet = 2 then                         "t"
        else if     nsujet = 3 then                         "s"
        else raise undefined)
        ^
        (if         genreobj='m' & nbobj=1   then
              (if noble=true then                           "es"
                          else                              "on" )   <-
        else if     genreobj='f' & nbobj=1    then          "a"
        else if     nbobj>1                   then          "es"
        else raise undefined) )
    else if     ngroupe > 1 then
        (if         ngroupe =1 or ngroupe = 2 then
            (if         nsujet = 1then                      "n"
            else if     nsujet = 2 then                     "v"
            else raise undefined)
            ^
            (if         nbobj = 1    then                   "otre"
            else if     nbobj >1     then                   "os"
            else raise undefined)
        else if      nsujet = 3 then
            (if         nbobj = 1    then                   "leur"
            else if     nbobj > 1    then                   "leurs"
            else raise undefined)
        else raise undefined )
```

∈ {1;2;3} × {1;2} × {'m';'f'} × {1;2} × **Bool**

Le problème a été localisé à la ligne indiquée par le symbole <- (éventuellement automatiquement grâce à la trace).
La chaîne "on" a été identifiée comme étant celle causant le problème et devant être remplacée par "es" dans le cas d'une personne noble. Cette modification ne nécessite pas de comprendre la logique de l'objet générique.




La solution présentée ci-dessous est analogue mais la duplication de la ligne "es" a été évitée en modifiant les conditions. Pour cela la logique régissant le choix des différentes variantes doit être comprise.

```
let g_possessif = fun (nsujet, ngroupe, genreobj, nbobj, noble) ->
    ...
        ...
        (if     genreobj='m' & nbobj=1& not noble then "on"
        else if genreobj='f' & nbobj=1                 then  "a"
        else if genreobj>1 or noble=true               then  "es")
    ...
```

∈ {1;2;3} × {1;2} × {'m';'f'} × {1;2} × **Bool**

Bien sûr cette méthode est critiquable. Remarquons cependant qu'après quelques années de maintenance la "logique" sous-jacente n'est plus nécessairement très claire et que "comprendre" peut être particulièrement coûteux. Une telle solution est souvent choisie en pratique, soit par ignorance, soit sous la pression des délais de livraison. On préfère livrer un système qui fonctionne (c'est le cas), plutôt que d'annuler une livraison. *Cette solution est une alternative acceptable si l'on dispose d'outils permettant une restructuration a posteriori.*
Dans notre exemple, elle est particulièrement médiocre car **G_POSSESSIF** est un objet générique réutilisé dans d'autres systèmes. Lui ajouter la dimension "noble" le dénature complètement. Par

contre si le langage admet la définition de paramètres par défaut il n'est pas nécessaire de modifier les objets clients (ici le paramètre noble serait initialisé à faux).

### II.7.5.2 *Solution (2) : Correction après une restructuration de G_TITRE_PERSONNE*

Si on analyse plus en profondeur le problème, il ressort que la conception originale ne peut être respectée. La règle indiquant que le premier composant correspond au possessif est remise en cause. Un chargé maintenance n'ayant jamais été convaincu par les vertus de cette conception initiale se propose alors de restructurer le système et de le simplifier avant d'intégrer le cas des personnes nobles.

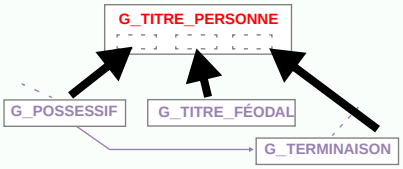
Les premières opérations effectuées consistent à *inclure* **G_POSSESSIF** et **G_TERMINAISON** dans **G_TITRE_PERSONNE**. Le résultat obtenu est ensuite simplifié en *spécialisant* la partie correspondant à **G_POSSESSIF**[1]. Cette spécialisation est intéressante car seules 3 variantes sur 15 sont utilisées ("mon","ma","mes"). La figure 70.a montre le résultat de cette transformation.
Grâce aux opérations de *développement*, de *reformulation* et de *restructuration d'objets par morceaux en intention*, on arrive finalement à une représentation 4 fois plus compacte que la représentation initiale (en nombre de ligne de code) (figure 70.b)[2]. *Clairement il est important de définir des critères de qualité et de pouvoir mesurer la complexité des représentations à l'aide d'outils* (on peut chercher à simplifier les conditions, à limiter la duplication, etc.)

*figure 70* Restructuration de **G_TITRE_PERSONNE.**

```
let G_TITRE_PERSONNE (genre,marie,nombre) =
        "m"
        ^
        (if       genre='m' & (nombre=1)  then      "on"
        else if   genre='f' & (nombre=1)  then      "a"
        else if   (nombre>1)              then      "es"
        else raise undefined)
        ^
        (if       genre = 'm' then'                 "sieur"
        else if   genre = 'f' & marie = false then  "demoiselle"
        else if   genre = 'f' & marie = true then   "dame"
        else raise undefined)
        ^
        (if  (nombre<=1)                  then      ""
        else                                        "s")
```

***(a)* Inclusions et spécialisation**

Cette représentation de **G_TITRE_PERSONNE** est sémantiquement équivalente à celle de la figure 65 (p.132). On retrouve les mêmes éléments mais l'objet a été "mis à plat" et spécialisé.




```
let G_TITRE_PERSONNE (genre,marie,nombre) =
        if        genre='m' & nombre=1            then  "monsieur"
        else if   genre='f' & nombre=1 & marie    then  "madame"
        else if   genre='f' & nombre=1            then  "mademoiselle"
        else if   nombre>1  & genre = 'm'         then  "messieurs"
        else if   nombre>1 & genre = 'f' & marie  then  "mesdames"
        else if   (nombre>1) & genre = 'f'        then  "mesdemoiselles"
        else raise undefined
```

***(b)* Développements, reformulations, restructurations**

Cette représentation est obtenue à partir de (a). La lettre "m" a été développée et donc dupliquée. Cette représentation est cependant bien plus simple à comprendre que la représentation originale.

La restructuration effectuée n'est pas directe. *En l'absence d'outil le chargé de maintenance risque de ne pas s'apercevoir des simplifications possibles et d'introduire des erreurs. De plus rien n'assurera la validité du résultat.* Par contre s'il dispose d'un éditeur sémantique quelques opérations seulement seront nécessaires ; l'insertion de "messire" est ensuite immédiate.

Le résultat obtenu n'est pas un "rapiéçage". Par contre cette solution correspond à une scission par rapport aux objets génériques déjà existants. La valeur de **G_POSSESSIF** et de **G_TERMINAISON** a été dupliquée ; les améliorations futures réalisées sur ces objets ne pourront être intégrées dans **G_TITRE_PERSONNE** qu'au prix d'une maintenance multiple. Malgré cela,

1. L'opération abstraite *inclure* a été introduite dans la Section II.6.2., la spécialisation dans la Section II.6.4.3.
2. Pour plus de détails consulter l'Annexe A.1.10, l'Annexe A.1.2 et la Section II.4.6.

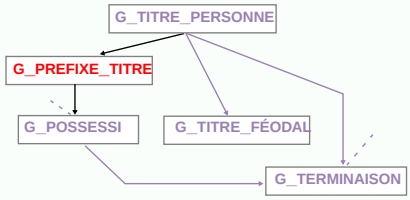
face à la complexité d'un objet générique qu'il ne savait plus gérer, le chargé de maintenance a choisi la discontinuité en simplifiant l'objet générique au maximum[1].

### II.7.5.3 *Solution (3) : Correction via une surcharge de G_POSSESSIF*

Des solutions moins radicales sont possibles. L'analyse de l'anomalie peut amener un chargé de maintenance à la conclusion suivante : "la décomposition du titre d'une personne en trois composantes n'est pas mauvaise en soi ; le seul problème c'est que le préfixe ne correspond pas *exactement* au possessif". Une solution possible est alors de surcharger l'objet générique en question. Concrètement cela consiste à définir un nouvel objet générique **G_PREFIXE_TITRE** à partir de **G_POSSESSIF** (figure 71).

*figure 71* Définition de **G_PREFIXE_TITRE.**

Ces représentations de **G_PREFIXE_TITRE** sont équivalentes à **G_POSSESSIF** sauf dans le cas d'une personne masculine noble. La représentation de gauche a été obtenue par un chargé de maintenance conscient du fait que "mes" était une variante déjà existante. Seule la fonction de transfert a été modifiée. A gauche par contre la variante à été dupliquée.




```
let G_PREFIXE_TITRE = fun( genre, marie, nombre, noble ) ->
    if genre='m' & noble=true
        then "mes"
        else G_POSSESSIF (1,1,genre, (if nombre>1 then 2 else 1)
```

```
let G_PREFIXE_TITRE = fun( genre, marie, nombre, noble ) ->
    G_POSSESSIF(1,1,genre,
        (if nombre>1 or (genre='m' & noble=true) then 2 else 1) )
```

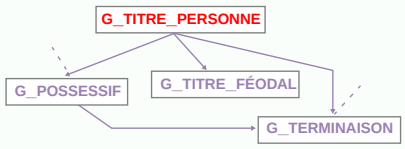
Cette solution est intellectuellement satisfaisante. Elle fait clairement ressortir le cas particulier tout en exprimant le fait que le préfixe peut être obtenu à partir du possessif[2]. En contrepartie le système comporte un objet de plus. Sa compréhension en est d'autant plus fragmentée. *Des outils de visualisation basés sur les opérations inclure et exclure permettent de pallier à ce problème.*

### II.7.5.4 *Solution (4) : Correction de G_TITRE_PERSONNE*

Le chargé de maintenance peut aussi faire un rapiéçage dans **G_TITRE_PERSONNE** (figure 72)

*figure 72* Correction de **G_TITRE_PERSONNE.**

Ces deux solutions sont plus naturelles. Elles consistent respectivement à choisir le possessif dans tous les cas sauf le cas particulier (à gauche) ou à changer la fonction de transfert (à droite).




```
let g_titre_personne = fun (genre,marie,nombre,noble) ->
    (if genre='m' & noble=true)
        then "mes"
        else G_POSSESSIF (1,1,genre, (if nombre>1) then 2 else 1))
    ^ G_TITRE_FEODAL(genre,marie,noble)
    ^ G_TERMINAISON(nombre>1)
```

```
let g_titre_personne = fun (genre,marie,nombre, noble) ->
    G_POSSESSIF (1,1,genre,
        (if nombre>1 or (genre='m' & noble=true) then 2 else 1))
    ^ G_TITRE_FEODAL(genre,marie, noble)
    ^ G_TERMINAISON(nombre>1)
```

En fait, ces deux solutions reviennent tout simplement à *inclure* **G_PREFIXE_TITRE**

1. Solution courante en informatique lorsque une organisation dispose des codes sources d'un produit qu'elle souhaite modifier.
2. Un chargé de maintenance familiarisé aux concepts orientés objet aurait certainement retenu cette solution proche de l'héritage.

## II.7.6 Une autre modification

Quelques années plus tard...
Suite à une évolution culturelle, l'usage du mot "seigneur" est préféré comme titre féodal de noblesse. L'entreprise répond à cette variation en surchargeant la variante "sire" par la variante "seigneur". Cette usage s'étend au titre d'une personne et "monseigneur" doit surcharger "messire".

Après une intégration rapide de **G_TITRE_FÉODAL** dans **G_TITRE_PERSONNE** on s'aperçoit que la variante "messeigneur" est incorrecte ! La conception initiale basée sur l'utilisation du possessif redevient valide. Autrement dit la déstructuration introduite par l'incohérence du français n'était peut être pas indispensable !

Si l'on dispose encore de l'ancienne version du système (celle sans le rapiéçage pour les nobles), le plus simple est peut être de repartir de celle-ci puis d'y intégrer la nouvelle version de **G_TITRE_FÉODAL**. Si des modifications ont été introduites entre temps (par exemple pour pouvoir prendre en compte des distinctions religieuses), celles-ci doivent être intégrées à la version originale. *Tout cela n'est possible que si l'on dispose d'outils de gestion de versions et d'outils permettant d'exploiter les différences entre les versions successives*[1].

Dans notre exemple, les années se sont écoulées et les raisons de la déstructuration pour la noblesse se sont perdues dans le temps. Le (nouveau) chargé de maintenance doit donc introduire la nouvelle modification en "aveugle". Selon sa compréhension du système, il risque de rapiécer le rapiéçage ou de proposer des solutions plus satisfaisantes.
Le lecteur est invité à étudier dans chaque cas quelles sont les solutions les plus probables.
Les rapiéçages les plus directs (solution (1) et (4)) sont en fait les plus faciles à défaire.
Inversement, la solution (3), la plus élégante et la plus "sophistiquée", n'est facilement modifiable que si le chargé de maintenance en comprend la logique. Sinon, des rapiéçages inutiles risquent d'être introduits (figure 73).

*figure 73* Modification de **G_PREFIXE_TITRE**.

```
let G_PREFIXE_TITRE = fun( genre, marie, nombre, noble ) ->
    if genre='m' & noble=true
        then "mon"
        else G_POSSESSIF (1,1,genre, (if nombre>1 then 2 else 1)
```

Le chargé de maintenance, après avoir bien cherché, estime finalement que la ligne ◁— est fautive. Le manque de visibilité, fait qu'il remplace directement le fragment "mes" par "mon". Ce faisant il ne s'est pas aperçu que le test est devenu inutile (dans les deux cas la variante générée est la même) ; **G_PREFIXE_TITRE** aussi... (la solution correcte pour ce problème est évidemment de supprimer tout simplement cet objet). Le plus grave, dans la solution implémentée est que le rôle de **G_PREFIXE_TITRE** risque d'être réellement obscur pour les futurs chargés de maintenance (évidemment, cet objet n'a aucun rôle!). Le système sera modifié avec bien des précautions et bien des craintes. Les solutions les plus directes seront retenues car ce sont celles qui semblent les moins risquées.

Suite à cette modification certaines parties du logiciel sont devenues obscures pour le chargé de maintenance. Eventuellement ce dernier, soupçonne l'existence de "code mort", mais en cas de doutes aucune modification ne sera faite. Le chargé de maintenance a de toute façon bien d'autres

1. Ces outils sont liés aux problèmes d'évolution plutôt que de variation et nous ne nous attarderons pas plus sur ce sujet. Remarquons simplement que l'évolution aurait pu être prise en compte avec les même mécanismes. Il s'agit de rajouter une dimension particulière : le temps. Ainsi, si la modification pour prendre en compte le cas des nobles s'est fait au temps $t_x$, les estampilles auraient été modifiées avec l'expression genre='m' & noble=true & temps >= $t_x$

préoccupations : après tout **G_TITRE_PERSONNE** n'est qu'un composant particulier ; le contenu de la lettre entière doit être régulièrement modifié...
*Cette situation fait ressortir l'importance d'outils de détection de code mort, plus généralement d'outils d'analyse statique et de restructuration. Des outils mesurant la complexité permettent par la suite de détecter les modules à risques, candidats potentiels pour une ré-ingéniérie.*

### II.7.7 Synthèse

Récapitulons. Au départ l'entreprise n'avait qu'un seul client et le système était relativement simple. Le développement de cette entreprise a été subordonnée à l'augmentation du nombre de clients. Il a bien fallu accepter leur variété. Un système faisant face à ces variations a été ingénieusement conçu et développé. Ce système étant utile, il a fallu le maintenir pendant longtemps et mais aussi l'adapter à des évolutions imprévues. Pour adapter le système à son environnement extérieur des entorses ont dues être faites à la conception initiale. Peu à peu le système s'est déstructuré, devenant à chaque fois plus difficile à comprendre. Aujourd'hui sa maintenance est fort coûteuse, et pourtant il est indispensable à l'entreprise. On est ainsi passé d'un système bien conçu à un cauchemar pour la maintenance.

Quelles étaient donc les autres opportunités ?

- il était possible de ne pas prendre en compte les personnes nobles, mais en sacrifiant le marché correspondant.
- l'entreprise pouvait aussi écrire à l'académie française pour protester contre les irrégularités de la langue... Il est sans doute plus pragmatique de se plier aux exigences extérieures.

Non, en réalité les problèmes de *variations* et d'*évolution* sont intrinsèques à la complexité de ce système. Le monde est varié et le monde change. Le système doit se plier à ces contraintes s'il veut continuer à être utile. Reste donc à essayer de limiter la déstructuration.

Nous avons vu aussi que pour un même problème, des solutions ayant des caractéristiques différentes étaient possibles. Les raisons menant le chargé de maintenance à une solution plutôt qu'à une autre sont souvent ténues ; *il est utile de proposer des outils facilitant la compréhension du système, l'évaluation de différentes solutions et la restructuration a posteriori si nécessaire.*

La modification maladroite de la solution la plus sophistiquée (la 3) s'est avérée être la plus désastreuse. En fait il apparaît plus généralement que *le changement de style des chargés de maintenance est un facteur supplémentaire de déstructuration*. Clairement les qualités de ces derniers sont déterminantes ; pourtant bien souvent ce sont les gens les moins qualifiés qui sont affectés à la maintenance !

Finalement remarquons que ce scénario élémentaire a permis de souligner le besoin d'outils supportant : la réutilisation, la compréhension, la visualisation, l'analyse d'impact, l'édition sémantique, la restructuration, l'analyse statique et dynamique, les tests, les mesures de complexité, la gestion de versions et l'analyse des différences.

La conception et la réalisation de tels services sont bien évidemment liées aux abstractions, représentations et opérations présentées dans ce chapitre.

# II.8 Conclusion

Dans ce chapitre un modèle abstrait a été présenté pour la programmation globale. Nous avons introduit les notions d'objets structurés, d'objets dérivés et d'objets générés ainsi que leur interactions. Pour chaque classe, il a été présenté :

- *des abstractions* définies à partir de notions issues de la théorie des ensembles ;
- *des représentations* très variées, mais aussi les équivalences les reliant ;
- *des opérations* définies autant sur les abstractions que sur les représentations ;
- *un exemple de manipulation* permettant d'apprécier de manière concrète les problèmes.

L'attention s'est surtout portée sur les problèmes de variations et les objets génériques. D'un point de vue abstrait ceux-ci ont été décrits comme étant des fonctions. Cette modélisation, pouvant sembler au départ quelque peu déconcertante, s'est révélée adaptée ; entre autre pour décrire facilement les différentes opérations définies sur les objets génériques.

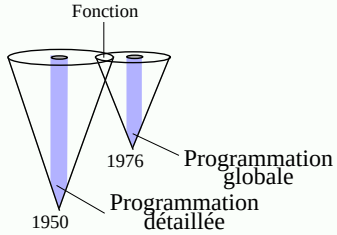
Les notions présentées dans ce chapitre ne sont pas spécifiques au logiciel. D'ailleurs comme le suggèrent les exemples, les concepts pourraient sans doute être appliqués à d'autres artefacts ; par exemple aux documents structurés et aux hypertextes. Certaines connexions ont également été faites avec le domaine des bases de données, notamment des bases de données temporelles. L'un des intérêts de ce chapitre est justement de proposer des abstractions permettant de franchir les limites d'un domaine particulier (voir les cônes du modèle hélicoïdal dans la Section I.2.5.5).

Dans le cadre de cette thèse nous nous concentrerons cependant sur le logiciel. Soulignons alors que la problématique traitée dans ce chapitre correspond à la *programmation globale* et non pas à la *programmation détaillée*. Nous avons longuement insisté dans le Chapitre I sur le fait qu'il s'agissait de deux aspects différents.

Cela n'empêche pas bout autant que des notions soient communes. Ici c'est le cas notamment de la notion de fonction et par la suite de celle de programme (figure 74)

*figure 74* Echanges programmation détaillée et programmation globale

L'approche poursuivie peut être schématisée à l'aide du modèle hélicoïdal. Ici nous avons représenté deux centres d'intérêts distincts et leurs cônes respectifs. La programmation détaillée, bien plus ancienne, est également plus ouverte à utiliser des abstractions. L'objectif de ce chapitre a été de trouver des abstractions pour la programmation globale. L'une de ces abstractions se révèle suffisamment proche de la programmation détaillée pour que l'on puisse espérer des échanges fructueux.




Reste tout de même à montrer que cette idée n'est pas qu'une vision de l'esprit, propre à l'utilisation d'abstractions ! Entre autre il faut montrer (1) qu'en pratique la technologie de la programmation globale est compatible avec le modèle présenté, tout au moins en partie ; (2) que non seulement des notions peuvent être réutilisées, mais aussi des techniques.
C'est ce que nous tentons de faire dans le reste de cette thèse.

Soulignons finalement que le modèle présenté n'est en aucune manière une solution définitive ! C'est ***un*** modèle pour la programmation globale et non pas ***le*** modèle.

# Chapitre III
# Aspects concrets de la programmation globale

# INDEX

## P



## R



## S



## T



## V

# LISTE DES FIGURES

# CHAPITRE III

# Aspects concrets de la programmation globale

## III.1 Introduction

*"La distance entre la théorie et la pratique est plus grande en pratique qu'en théorie..."*

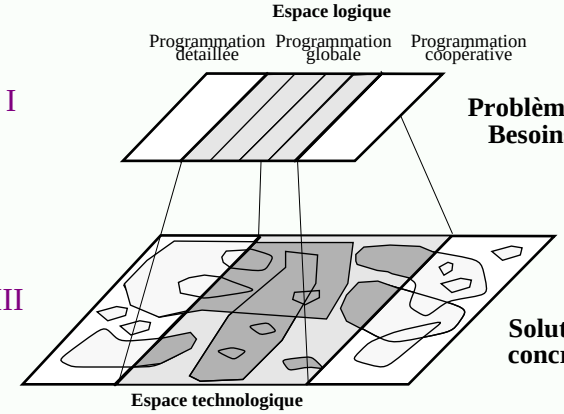
Ce chapitre s'intéresse à la ***technologie de programmation globale.*** Les objectifs sont : (1) de décrire les aspects concrets de la programmation globale ; (2) d'étudier quelles sont les techniques utilisées en pratique ; (3) de les classer ; (4) d'étudier leurs interactions.

***Espace logique vs. espace technologique...***

Lorsque l'on souhaite atteindre de tels objectifs, on s'aperçoit rapidement qu'il faut faire face à un problème majeur : de très nombreuses solutions techniques ont été proposées ; pour faire une synthèse *il est indispensable de structurer cet* ***espace technologique***. Ce n'est pas une tâche facile car les notions et les techniques utilisées par chaque outil forment une structure largement enchevêtrée.

Comme le suggère la figure 75, il est cependant possible de se baser sur la structuration de l'*espace logique* proposée dans le Chapitre I. Nous avions insisté sur le fait que cette structuration correspondait à une vision abstraite ; c'est bien ce que confirme la figure : la plupart des systèmes et outils chevauchent les frontières décrites.

*figure 75* Espace logique et espace technologique




Ce plan correspond à l'espace logique du génie logiciel. Sa structuration est basée sur les *notions* utilisées (Section I.4.2). Son principal avantage est qu'elle est indépendante de toutes considérations techniques.

Ce plan par contre correspond à l'espace technologique du génie logiciel. Il représente informellement les différents systèmes et outils aux frontières hasardeuses.

L'utilité de la structuration logique n'est pas remise en cause. Simplement il faut admettre que l'on ne peut pas se référer aux "outils de programmation détaillée" ou aux "outils de programmation globale" sans faire d'abus de langage ; ces classes ne sont pas indépendantes.

***Rationalisation : une réduction de l'espace technologique via l'utilisation d'abstractions...***

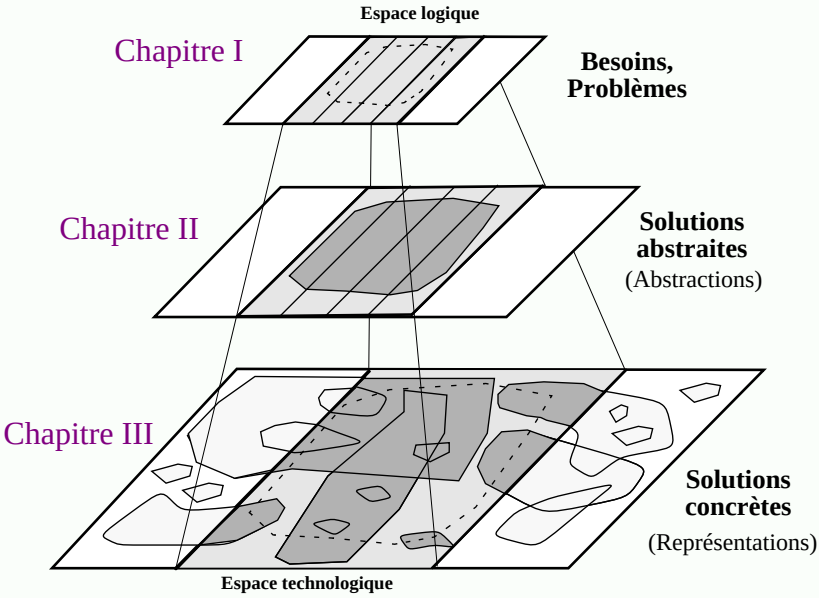
On pourrait se contenter de mettre en relation les outils aux besoins qu'ils couvrent (l'espace technologique à l'espace logique).
Dans cette thèse nous poursuivons un objectif bien plus ambitieux : O3 Favoriser un processus de rationalisation dans le domaine de la programmation globale. Autrement dit, il s'agit de montrer que l'ensemble des solutions concrètes, même s'il est très étendu, correspond à un ensemble d'abstractions beaucoup plus réduit. C'est bien évidemment là qu'intervient le modèle abstrait proposé dans le chapitre précédant (figure 76).

*figure 76* Programmation globale : problèmes, abstractions et représentations

a programmation globale se aractérise par un ensemble de roblèmes et de notions.

Le modèle abstrait fournit une solution abstraite. Comme nous l'avons déjà dit ce modèle n'est pas *complet* ; il ne couvre qu'une partie des problèmes. Ce n'est pas *le* modèle ; il ne permet de modéliser correctement qu'une partie des solutions concrètes.

De multiples solutions concrètes (techniques, outils, etc.) peuvent être interprétées à partir du modèle.




Le passage d'un problème de génie logiciel à une solution concrète peut être vu comme une exécution d'un cycle de vie. Le niveau intermédiaire correspond à une solution abstraite. Bien évidemment celle-ci devrait être élaborée avant la solution concrète, mais ce n'est pas toujours le cas. Dans cette thèse nous nous proposons au contraire de modéliser a posteriori les solutions concrètes (l'approche A5 "Rétro-ingénierie : Reconsidérer l'existant sous un nouveau jour.").

En fait, lors de la définition du modèle abstrait, nous n'avons fait référence au logiciel que de manière occasionnelle, et ce dans l'intention de le rendre plus général. Ce chapitre vise à montrer qu'il est adapté à la programmation globale. Pour fixer les idées donnons tout de suite la correspondance informelle (voir la figure 77 et plus précisément le tableau 12).

TABLEAU 12 Relation informelle entre le modèle abstrait et les notions de programmation globale

| Modèle abstrait (Chapitre II) | Programmation globale (Chapitre I) |
|---|---|
| constructeurs | composition de systèmes en sous-systèmes, de sous-systèmes en modules, etc. |
| références | e.g. relation de dépendance entre modules |
| objets structurés | e.g. modules, sous-systèmes, systèmes |
| opérateurs | outils de dérivation (e.g. compilateur, éditeur de liens) |

*gure 77* Architecture, manufacture, variation, évolution le modèle abstrait

Cette figure explicite la relation existant entre le modèle abstrait et les notions de programmation globale. La structure générale est celle utilisée tout au long du Chapitre II. Rappelons que cette représentation graphique correspond à un espace à trois dimensions (revoir la figure 27 (p.87)). L'association entre les classes d'objets et les thèmes de la programmation globale est la suivante :

**objets structurés** <--> **architecture**
**objets dérivés** <--> **manufacture**
**objets générés** <--> **variation et évolution**

Le modèle abstrait permet de souligner que les différents thèmes ne sont pas indépendants. C'est bien ce qui apparaissait aussi dans le Chapitre I, notamment dans la Section I.4.4.3. Si l'on utilise des structures classiques pour les représenter ce n'est que par mesure de simplification.

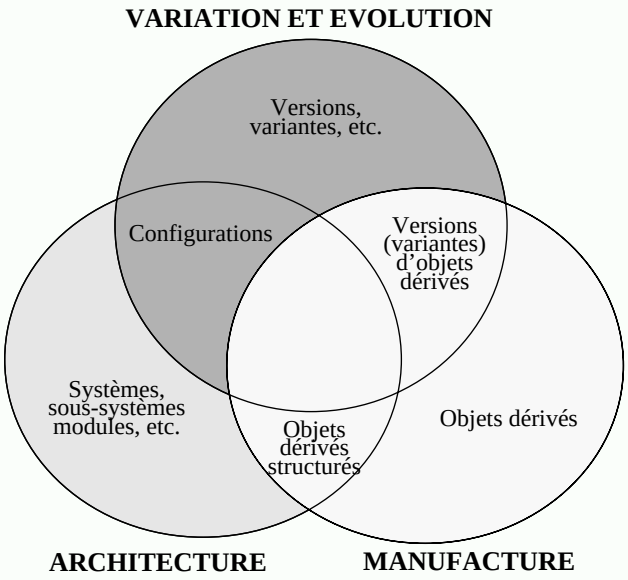



TABLEAU 12 Relation informelle entre le modèle abstrait et les notions de programmation globale

| Modèle abstrait (Chapitre II) | Programmation globale (Chapitre I) |
|---|---|
| objets dérivés | objets dérivés (e.g. code objet, binaire exécutable) |
| objets générés | versions, variantes |
| objets génériques | méta-programmes, groupes de versions |

### *Complexité de l'espace technologique du génie logiciel...*

Revenons aux caractéristiques de l'espace des solutions concrètes. La question est de pouvoir expliquer l'existence des nombreuses variantes et surtout de les classer.
Dans cette section les "solutions concrètes"[1] font l'*objet* de la discussion. Si l'on considère un tel objet on retrouve les notions de *variation*, d'*évolution* et d'*architecture* (non pas appliquées aux logiciels mais aux solutions de génie logiciel!). Ces aspects expliquent la complexité de l'espace technologique.

- *Variation*. Une solution est développée dans un certain *contexte* ; elle est souvent basée sur une technologie déjà existante. La *variété* des contextes (des technologies pouvant être utilisées) donne ainsi lieu à différentes *variantes*.
- *Evolution*. De surcroît, l'ensemble des technologies disponibles *évolue* au cours du temps.
- *Architecture*. L'ensemble des besoins en génie logiciel est si complexe qu'aucun outil ne cherche à les satisfaire tous. Par contre certaines solutions sont *construites* à partir de solutions *atomiques*. Dans le cas des outils, on arrive alors à la notion d'***environnement de génie logiciel***[2].

1. Un outil, un système, une technique, la manière d'utiliser un outil, etc...
2. Dans cette thèse nous utiliserons ce terme dans son sens le plus large et ce indépendamment des mécanismes d'intégrations (des *constructeurs*) utilisés ou de tout autre caractérisation. On fera parfois la distinction entre les ***environnements de programmation détaillée*** et les ***environnements de programmation globale***, mais par abus de langage. Le lecteur trouvera dans [DartEFH87] [SmitOman90] [NormC92] [Meye93] [Fugg93] des taxonomies beaucoup plus précises.

En fait dans un tel contexte versionné, un environnement de génie logiciel est une *configuration* d'outils particuliers. Plus généralement les solutions concrètes utilisées sont des configurations de solutions plus simples, ce qui explique leur grande variété.

Cette variété peut aussi être inférée à partir de la figure 76 présentée ci-dessus. Elle montre le passage d'un problème à une solution concrète comme un cycle de vie et est implicitement basée sur le fait qu'il existe différentes représentations possibles pour une même abstraction.

#### *Intégration et cohérence dans un contexte multi-technologique...*

Il est important de noter que dans une configuration donnée on assiste parfois à des recouvrements de fonctionnalités. Autrement dit différentes technologies peuvent être utilisées simultanément pour résoudre un même problème ; soit parce qu'elles offrent des avantages exploités différemment selon les situations ; soit pour des raisons d'héritage historique (une technologie largement utilisée ne peut être abandonnée) ; soit parce que les fonctionnalités offertes par les deux technologies se recouvrent partiellement. La gestion de technologies multiples pose bien évidemment des problèmes d'*intégration* et de *cohérence*.
Ce discours s'applique aussi bien à la programmation détaillée qu'à la programmation globale.

#### *Le cas de la programmation détaillée...*

Aujourd'hui l'espace technologique de la programmation détaillée est *relativement* clair. Des abstractions sont disponibles dans ce domaine pour modéliser algorithmes et structures de données. La technologie des langages de programmation est la plus développée en informatique. Elle est presque entièrement dédiée à la programmation détaillée[1]. Cette technologie est l'une des mieux structurée ; autant du point de vue de son évolution que de ses variations (le spectre allant des langages impératifs aux langages déclaratifs est la dimension principale de variation [PetrWind94]).

Les environnements de programmation détaillée sont classiquement des configurations composées d'un éditeur (syntaxique), d'un compilateur et d'un outil de mise au point.

Dans une même configuration peuvent apparaître plusieurs langages différents. En effet de nombreux logiciels de grande taille sont multi-langages [Favr88] ; soit pour des raisons d'héritages historiques (des parties sont écrites en assembleurs) ; soit pour des raisons fonctionnelles (un langage comme C peut être utilisé pour les aspects systèmes, un langage comme Prolog pour les aspects déductifs).

Un algorithme n'est généralement représenté qu'une seule fois, dans un langage unique. Le *problème d'intégration* se pose entre des modules écrits dans des langages différents.
Par contre *une même* structure de données peut avoir des *représentations multiples* dans des langages différents. Ceci pose alors un *problème de cohérence* : il faut s'assurer que l'information représentée est bien la même.

1. Elle chevauche dans certains cas le domaine de la programmation globale (voir ci-dessous).

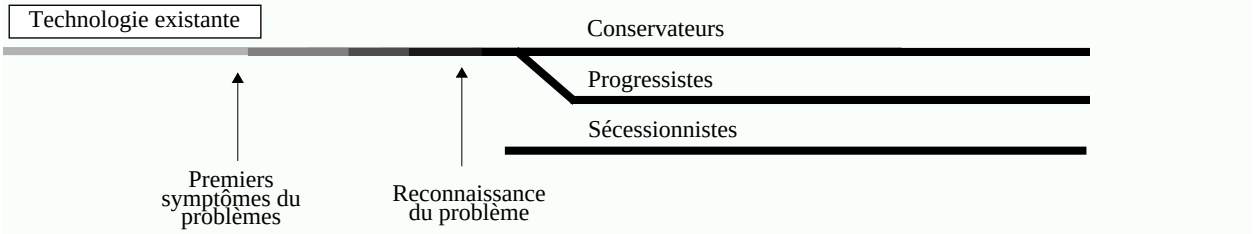
### *La cas de la programmation globale...*

L'espace technologique de la programmation globale est moins clair. Ceci peut s'expliquer par son manque de maturité ; mais aussi dans une certaine mesure par le fait que des technologies déjà existantes étaient disponibles lors de son apparition dans les années 70.

Le modèle hélicoïdal ne modélise pas cette situation explicitement, mais on peut l'étendre.

- *Continuité vs. changement de technologie*. Après la manifestation des premiers symptômes d'un problème, les premières solutions apparaissant correspondent à de nouvelles utilisations des technologies existantes ; utilisations plus ou moins ésotériques car détournant la technologie de ses objectifs. Après un certain temps le problème est reconnu comme tel. C'est souvent un point de divergence : (1) les *conservateurs* continuent à utiliser la même technologie, (2) les *progressistes* lui intègrent de nouveaux concepts, (3) les *sécessionnistes* cherchent à définir une nouvelle technologie adaptée au problème.

*figure 78* Divergence d'approches face à l'apparition progressive d'un problème



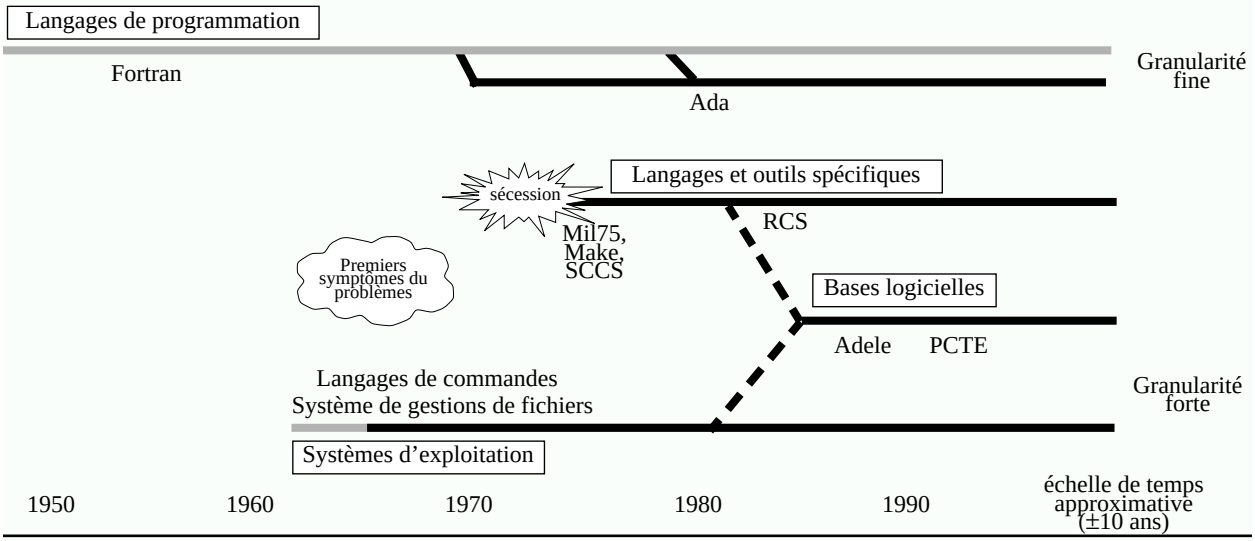

- *Solutions empiriques vs. élaborées*. De manière plus ou moins orthogonale les solutions peuvent être basées sur des recherches plus où moins longues. Les solutions empiriques sont directement utilisables. Par contre les retombées pratiques des solutions conceptuelles sont loin d'être immédiates. La technologie résultant, disponible et après des années, intègre souvent les progrès technologiques ou conceptuels faits pendant ce temps dans d'autres domaines.
- *Complétude des solutions*. Souvent les premiers symptômes d'un problème ne révèlent que certains aspects. Au départ ceux-ci sont souvent traités individuellement, ce qui donne lieu à un ensemble de techniques spécifiques. Après la clarification de leurs interactions des solutions plus globales peuvent être proposées.

La combinaison de ces différentes dimensions provoque, pour résoudre un problème donné, l'apparition et la fusion de technologies d'une manière relativement chaotique. En tout cas à un moment donné, les solutions pratiques sont souvent multi-technologiques.

Ce modèle s'applique assez bien à la programmation globale. L'origine des premiers symptômes peut être située aux alentours des années 60-70 avec l'apparition de problèmes sérieux pour développer des logiciels de grandes tailles. A cette époque les deux technologies principales étaient la technologie des langages de programmation et celle des systèmes d'exploitation. Comme nous l'avons vu ce n'est qu'au milieu des années 70 que le problème de la programmation globale est explicitement apparu sous la forme d'un problème d'architecture. DeRemer et Kron sont les instigateurs du mouvement sécessionniste[1].

Pour simplifier on considérera dans la suite quatre approches principales (figure 79) :

*figure 79* Quatre approches technologiques pour la programmation globale




- *"Systèmes d'exploitation"*. Cette approche essentiellement conservatrice est basée sur l'utilisation de la technologie des *systèmes de fichiers* et des *langages de commandes*[1] pour résoudre les problèmes de programmation globale. Par exemple l'architecture ou les variations du logiciel sont représentées grâce à des conventions de nommage et des structures en répertoires. En dépit de la qualité médiocre de cette solution, aujourd'hui cette technologie provenant des années 70 est encore utilisée telle quelle. C'est un standard de facto ; omniprésente elle peut être appliquée à un moindre coût. Certaines extensions ont également été proposées mais en partant de cette base fixe [ChiuLevi93] [Kris95].
- *"Langages de programmation"*. Cette approche progressiste consiste à intégrer des concepts de programmation globale aux langages de programmation ; par exemple le concept de module dans les langages de programmation modulaire (Modula, Ada, etc.).
- *"Langages et outils spécifiques"*. L'approche sécessionniste ne s'est pas limitée au langage d'interconnexion de modules défini par DeRemer et Kron (MIL75 [DereKron76]) : plus ou moins à la même époque des outils comme Make [Feld79] ou SCCS [Roch74] faisaient leur apparition. Par la suite bien d'autres langages ou outils ont été proposés mais à chaque fois pour résoudre des problèmes précis et spécifiques ; la plupart du temps de manière isolée.
- *"Bases logicielles"*[2]. Cette approche, la plus ambitieuse, est le résultat de recherches plus approfondies (ce qui explique son apparition bien postérieure). Sa principale caractéristique est qu'elle essaie de faire face à tous les problèmes de programmation globale. Elle est basée

1. *"Structuring a large collection of modules to form a 'system' is an essentially different intellectual activity from that constructing the individual modules. That is, we must distinguish programming-in-the-large from programming-in-the-small. Correspondingly, we believe that essentially distinct and different languages should be used for the two activities"* [DereKron76].

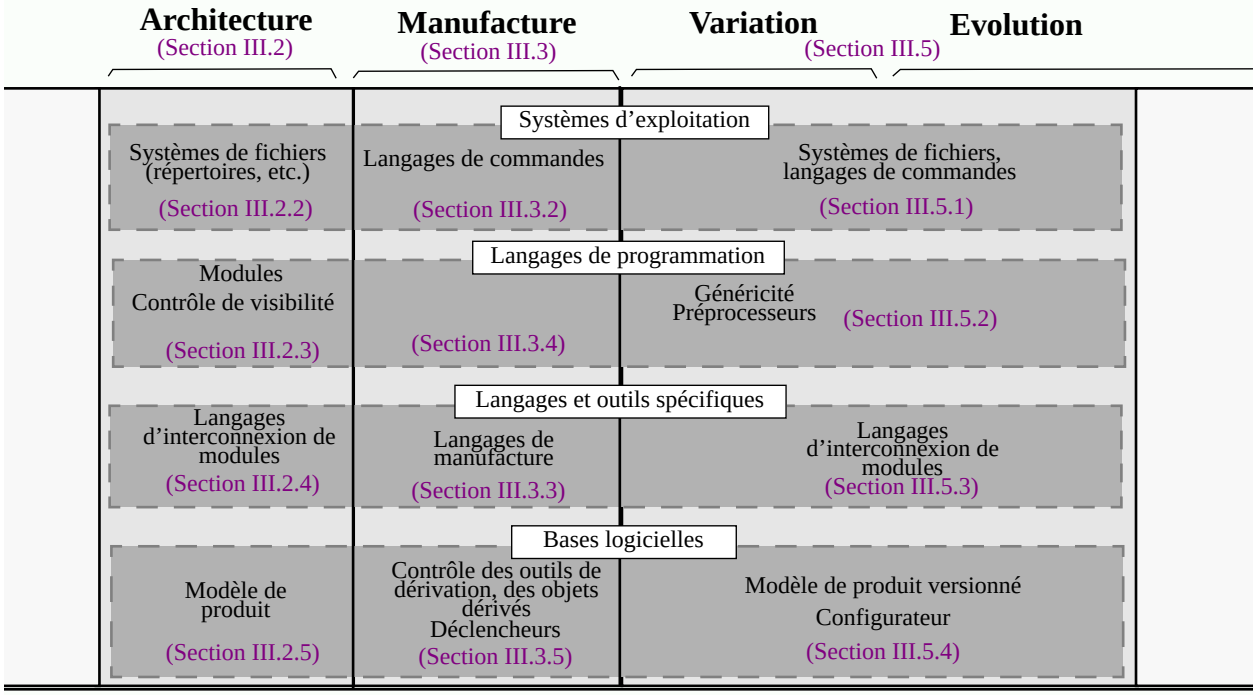
1. "File system" et "command language" ou "shell".
2. L'appellation d'un tel environnement dépend des services offerts. Ceux-ci peuvent aller de services d'intégration d'outils à un support pour le processus logiciel (donc pour la programmation coopérative). Une taxonomie est proposée dans [Fugg93]. Les différents types d'environnements sont discutés plus longuement dans [Melo93]. Ici le terme "base logicielle" a été retenu mais on pourrait aussi utiliser les termes "base de programmes", "bases de données pour le génie logiciel", ou "software repository" en anglais. Ces différents termes ont des connotations parfois différentes, mais dans le cadre de cette thèse il n'est pas utile de rentrer dans les détails.

sur une modélisation du produit logiciel ; généralement à partir de modèles de données entités-relations étendus. Cette approche bénéficie des apports conceptuels et technologiques des bases de données.

En réalité, l'ambition des bases logicielles s'étend souvent à la programmation coopérative. Il s'agit alors de contrôler le processus logiciel et une modélisation élaborée de celui-ci est nécessaire [Melo93]. Cet aspect n'est pas abordé dans cette thèse.

Ces quatre approches peuvent être utilisées pour structurer l'espace technologique comme le montre la figure 80 qui présente des exemples décrits tout au long de ce chapitre.

*figure 80* Structuration de l'espace technologique de la programmation globale




***Des liaisons fortes entre les techniques d'architecture, la manufacture, la variation...***

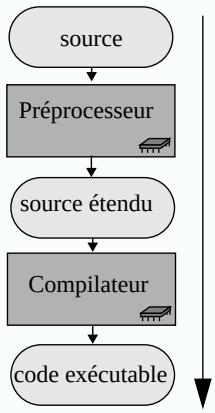
Avant de débuter la présentation des technologies de programmation globale il est utile de souligner que les différentes approches ne sont pas du tout incompatibles ; elles peuvent même apparaître simultanément dans une même configuration.

Un autre aspect rend encore plus complexe ces configurations. Les techniques utilisées pour chaque thème de programmation globale (architecture, manufacture, variation et évolution) ont des interactions si fortes qu'il est impossible de les ignorer.

Dans la figure 81 des exemples triviaux sont présentés pour chaque thème pris indépendamment les uns des autres. Chaque exemple correspond à une dimension : l'architecture pour la dimension horizontale, la manufacture pour la dimension verticale, et finalement la variation et l'évolution pour la troisième dimension.

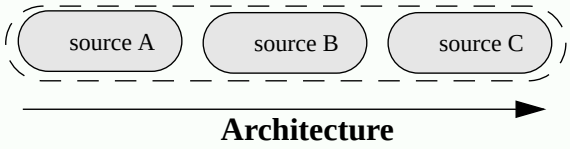
L'exemple présenté dans la figure 82 correspond à une configuration triviale. Les trois dimensions ont été intégrées. Comme on peut le voir leurs interactions donnent lieu à l'apparition de techniques qui n'existaient pas auparavant. C'est le cas par exemple du gestionnaire de configurations rendu nécessaire par les interactions entre architecture et variation.

*figure 81* Architecture, Manufacture, (Variation et évolution) : 3 dimensions



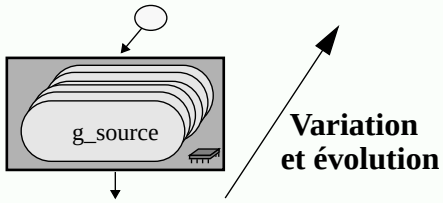

**Manufacture**

Dans cet exemple le texte source d'un programme (monolithique) est tout d'abord "étendu" en utilisant un préprocesseur, puis par la suite compilé.




**Architecture**

Ici le programme n'est plus monolithique mais se décompose en trois modules. Les relations entre ces modules ne sont pas représentées.




Ci-dessus un objet générique a été représenté informellement. Un choix donné en entrée permet de sélectionner automatiquement une variante.

NB. Les conventions graphiques utilisées dans ces figures sont décrites dans l'Annexe A.3.1 ("Conventions graphiques").

Ce chapitre cherche à décrire de manière conceptuelle les configurations complexes que l'on rencontre en pratique. Bien souvent ce ne sont que l'échafaudage de mécanismes hétérogènes ; parfois de très bas niveau. Les différents concepts que nous allons introduire vont permettre d'expliquer de manière rationnelle et cohérente l'exemple de la figure 100 (p.195).

***Plan de ce chapitre...***

Dans ce chapitre chaque thème est présenté successivement. La technologie de variation est plus particulièrement décrite. Celle correspondant aux problèmes d'évolution n'est pas très différente. Il n'y sera fait explicitement référence qu'en de rares occasions.

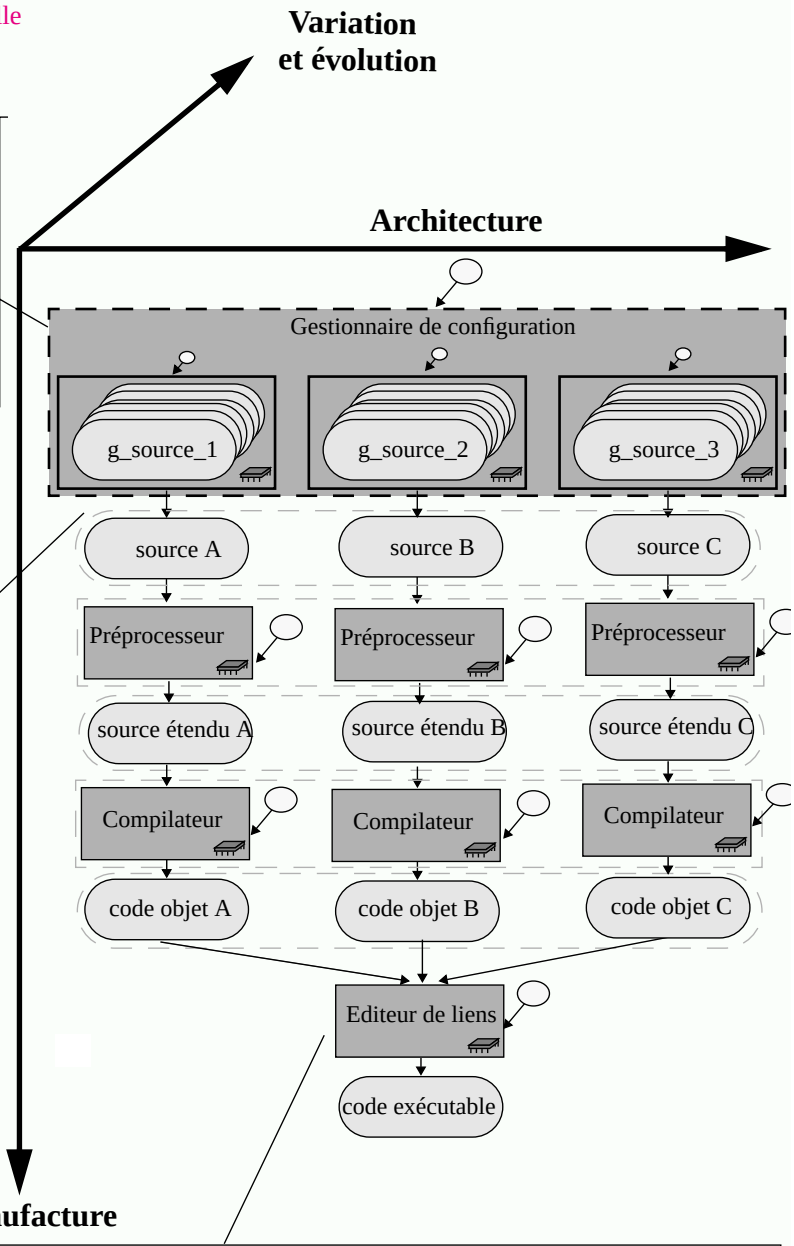
Pour chaque thème, un certain nombre de concepts seront présentés ; d'une part pour combler les lacunes du modèle, d'autre part pour prendre en compte les spécificités du logiciel. Ensuite les apports des 4 approches présentées seront successivement abordés (figure 83).

L'ordre de présentation des thèmes est le même que dans le Chapitre II (objets structurés, dérivés et générés pour le modèle abstrait ; architecture, manufacture, variation pour les aspects concrets).

Un exemple regroupant les différents aspects termine ce chapitre.

*figure 82* Intégration des 3 dimensions

Cet exemple montre de manière informelle l'intégration des 3 dimensions.




***(a)*** **Architecture et variation**

L'intersection entre les problèmes d'architecture et les problèmes de variations donne lieu aux problèmes de gestion de configurations. En terme de modèle il s'agit de définir des objets génériques et il est clair que des techniques particulières doivent être alors utilisées.

***(b)*** **Manufacture et variation**

Dans cet exemple les problèmes de variations ont été traités au niveau du code source. Une étape supplémentaire à été rajoutée dans le processus de manufacture.

En pratique les variations interviennent à chaque étape de la manufacture. Quand les variations sont prises en compte la manufacture est paramétrée par les différents choix. Ce peut être sous la forme de définitions de macros, d'options de compilation, ou encore de chemins d'accès aux librairies lors de l'édition de liens. Comme nous le verrons des étapes de manufacture sont parfois rajoutées uniquement pour améliorer les techniques de variations (c'est le cas par exemple de l'édition de liens statiques et de l'édition de liens dynamique).

***(c)*** **Architecture et manufacture**

Pour simplifier on suppose dans cet exemple que le langage de programmation utilisé est basé sur un mode de compilation indépendante : les modules sont compilés indépendamment les uns des autres mais il est nécessaire d'ajouter une étape supplémentaire au processus de manufacture pour réaliser l'édition de liens.

*figure 83* Architecture logique du Chapitre III

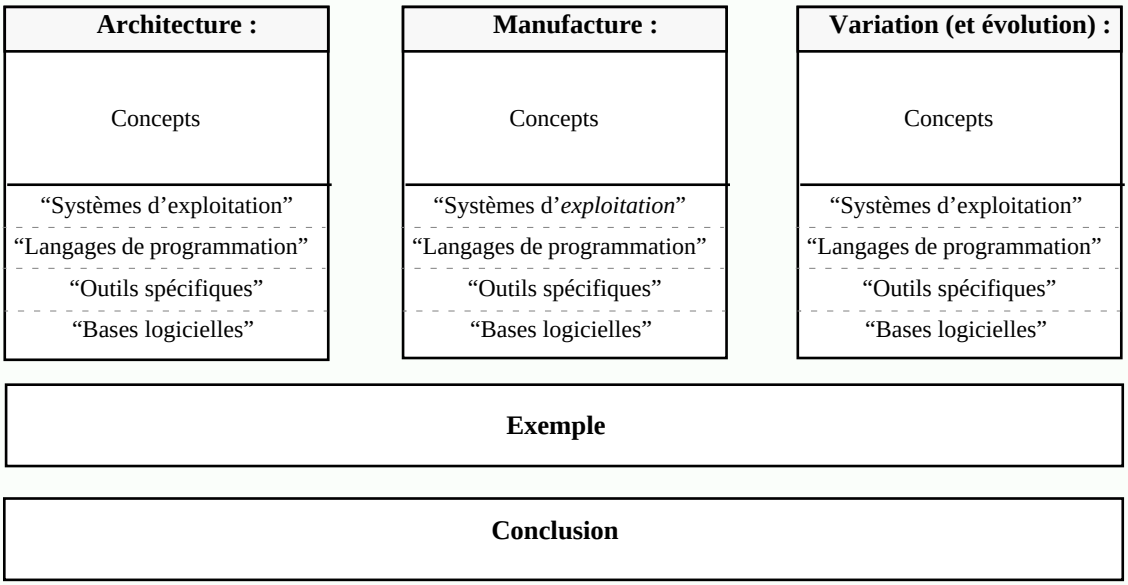
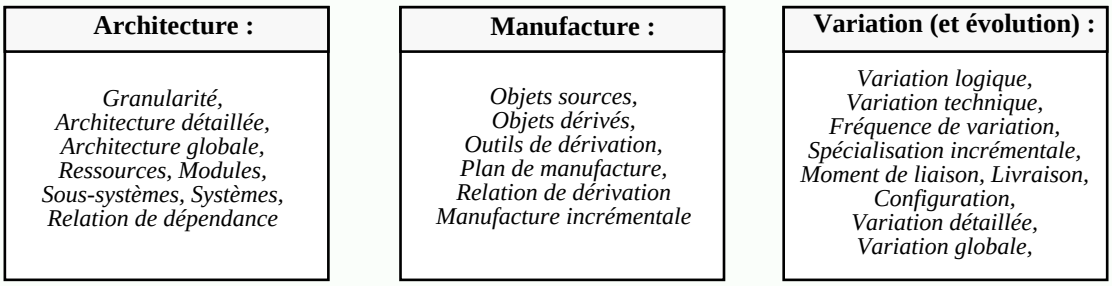



*figure 84* Répartition des concepts



# III.2 Architecture

L'architecture des logiciels est le point de départ de la programmation globale et son importance a été déjà soulignée dans le Chapitre I (Section "Importance de l'architecture et des problèmes associés" (p. 51)).
Dans un premier temps cette section présente un certain nombre de concepts relatifs à l'architecture : granularité, architecture détaillée, architecture globale, etc. (III.3.1).
Ensuite les techniques de représentation de l'architecture sont étudiées pour chaque approche :
• III.2.2 "*Systèmes d'exploitation*" : l'utilisation des systèmes de fichiers ;
• III.2.3 "*Langages de programmation*" : la compilation séparée, les langages modulaires, etc. ;
• III.2.4 "*Langages et outils spécifiques*" : les langages d'interconnexions de modules ;
• III.2.4 "*Bases logicielles*" : les modèles de produits.

## III.2.1 Concepts

L'architecture est basée sur deux points fondamentaux :

- *L'existence de différents niveaux de granularité*[1]. Cet aspect est lié à l'utilisation de *constructeurs* puisqu'il s'agit de définir des objets construits à partir d'objets atomiques. Se placer à un niveau de granularité donné consiste à considérer des objets construits comme étant atomiques. Il s'agit d'un processus d'abstraction dans lequel on cherche à cacher des "détails".
- *Le contrôle des relations existant entre les "grains"*. Il est clair que dans la plupart des cas la structure hiérarchique induite par les constructeurs est trop restrictive. On utilise alors des *références* entre les différents composants. Les relations ainsi formées peuvent devenir très complexes et il est nécessaire de les contrôler, surtout si l'on considère différents niveaux de granularité. En effet passer d'un niveau à l'autre implique non seulement de "masquer" la structure hiérarchique d'un objet mais aussi de "simplifier" les relations entre grains.

Dans le cas précis du logiciel, et plus particulièrement en ce qui concerne le code source, les problèmes d'architecture ont leur base dans la programmation détaillée. Il existe un continuum entre ce domaine et la programmation globale. Ici trois niveaux de granularité sont distingués :

- *La granularité fine* : elle correspond à la programmation détaillée.
- *La granularité moyenne* : elle donne lieu à "l'*architecture détaillée*".
- *La granularité forte* : elle correspond à "l'*architecture globale*".

Répétons que ces niveaux ne sont pas indépendants mais qu'au contraire il s'agit d'un continuum. Si des représentations différentes coexistent dans le logiciel pour chaque niveau il est d'assurer leur intégration. Si elles se chevauchent il est nécessaire d'assurer leur cohérence.

1. Considérer deux niveaux de granularité différents est l'une des caractéristiques généralement prise pour faire la différence entre la programmation détaillée et la programmation globale. Voir la Section "Granularité forte et granularité fine..." (p. 44).

### III.2.1.1 *Granularité fine : programmation détaillée*

En prenant en compte une ***granularité fine***, un logiciel est composé d'instructions, de variables, d'expressions, etc. Si l'on considère l'arbre abstrait d'un programme, les différentes feuilles (les identificateurs, les constantes, etc.) sont des éléments des *domaines basiques* et les différentes constructions syntaxiques sont les *constructeurs*.

De même la décoration de l'arbre, typiquement la liaison faite entre un identificateur et sa définition, correspond généralement à des *références*. En ce sens un programme peut être vu comme un *objet structuré*[1]. Cette vision est à la base des environnements dirigés par la syntaxe [RepsT84] [Donz&al84] [HabeNotk86] [Borr&al88] [BallGV90].

### III.2.1.2 *Granularité moyenne : architecture détaillée*

#### *Ressources...*

Il est impensable de voir un logiciel complexe comme un algorithme monolithique s'exécutant sur une structure de donnée unique. Au contraire il est nécessaire de définir un certain nombre d'abstractions, typiquement des procédures dans les langages de programmation impératifs [Shaw84].

Une procédure est un *objet construit* dans la mesure où il est défini à partir d'une liste d'instructions, de déclarations, etc. Ces abstractions permettent de définir une ***granularité moyenne***, celle où le grain est une ***ressource*** : typiquement une procédure, un type, une variable globale, etc.[2]. Clairement la répartition des instructions dans les procédures ne se fait pas au hasard, elle résulte plutôt du processus de conception.

Les différentes ressources ainsi obtenues ne sont pas indépendantes et font donc référence les unes aux autres. Le graphe correspondant sera désigné "***flot de ressources***". Dans son sens le plus large cette appellation inclue entre autre le graphe d'appels entre procédures. La simplicité du flot de ressources est déterminée par la qualité de la conception.

#### *Modules...*

Le nombre de ressources étant très important dans un logiciel de grande taille il est nécessaire de définir un autre niveau d'abstraction. On arrive alors à la notion de module dans son sens large. Un ***module*** est un ensemble de ressources[3]. Il s'agit donc d'un objet construit.

#### *Architecture détaillée...*

Dans cette thèse nous appellerons ***architecture détaillée*** la répartition des ressources dans les modules ainsi que le flot de ressources correspondant. L'architecture détaillée considère le logiciel selon une granularité moyenne : l'ensemble des ressources est supposé être un domaine atomique et l'ensemble des modules un domaine construit.

---

1. Le modèle introduit dans le chapitre précédent avait pour vocation de décrire des aspects de programmation globale, mais il est si général qu'il est bien évidemment utilisable dans d'autre contexte, ici dans le cas de la programmation détaillée.
2. Dans la suite de ce document le terme "ressource" désignera le grain correspondant au niveau de granularité moyen. Ce terme est indépendant du langage de programmation considéré.
3. L'implémentation de la notion de module varie d'un langage à l'autre. Ici le terme est employé de manière lâche.

Alors que l'architecture a été définie comme un problème de programmation globale, ici les grains de base (i.e. les ressources) sont des entités de programmation détaillée. On se trouve donc à la jonction des deux domaines. Dans "architecture détaillée" l'adjectif "détaillée" a justement été choisi pour faire ressortir cet aspect.

### III.2.1.3 *Granularité forte : architecture globale*

Puisque le nombre de modules peut également être très important, il est utile d'appliquer une fois de plus un processus d'abstraction. Il s'agit alors de construire ce que l'on appellera des systèmes ou des sous-systèmes. Un ***sous-système*** est un ensemble de modules et de sous-systèmes. Un tel constructeur défini donc une structure hiérarchique arborescente. Un ***système*** est la racine d'une telle arborescence[1]. Le flot de ressources induit des relations entre les modules et par la suite entre les sous-systèmes. Ceux-ci ne sont donc pas indépendants.
Nous définissons l'***architecture globale*** comme étant la répartition des modules en systèmes et sous-systèmes et l'ensemble des relations liant ces entités. Dans le cas de l'architecture globale le logiciel est considéré selon une ***granularité forte*** : les modules sont des objets atomiques alors que les systèmes et sous-systèmes sont des objets construits.

### III.2.1.4 *Discussion*

Le problème principal dans le cadre de l'architecture est de trouver une organisation des différents composants facilitant la compréhension et la manipulation du logiciel. L'architecture est le résultat de la phase de conception. Cependant dans le cadre de la maintenance on s'intéresse surtout au résultat obtenu après la phase d'implémentation, c'est à dire aux artefacts résidus de cette architecture dans le code source des programmes où dans toutes autres représentations concrètes d'implémentation. La raison en est simple : dans le cas de logiciels âgés, c'est souvent les seuls documents qui sont systématiquement mis à jour[2].

Toute activité complexe menée sur le logiciel est basée sur son architecture. Pour répondre aux différents besoins il peut être alors utile de faire coexister plusieurs architectures [SoniNH95]. On peut considérer par exemple une décomposition du logiciel en termes de responsabilités des équipes, en termes de fonctionnalités, etc [TillWMS93]. Pour des raisons techniques il existe généralement une seule architecture détaillée[3] mais certains systèmes proposent de gérer plusieurs architectures globales [TillMull93].

TABLEAU 13 Synthèse : trois niveaux de granularité

| Granularité | Domaines atomiques | Domaines construits | Thème |
|---|---|---|---|
| fine | constantes, identificateurs, etc. | instructions, expressions, etc. | programmation détaillée |
| moyenne | *ressources* : procédures, types, etc. | modules | architecture détaillée |
| forte | module | systèmes, sous systèmes | architecture globale |

1. Ces termes et ces notions peuvent varier selon les techniques employées.
2. Traditionnellement la programmation globale est plutôt tournée vers la gestion du produit résultant de l'implémentation (le terme "programmation" qui est représentatif de cette tendance). Dans cette thèse l'architecture fait essentiellement référence à cette phase et elle s'exprime en terme d'entités concrètes d'implémentation. Les méthodes de conception proposent des formalismes pour décrire "l'architecture" du logiciel, mais à niveau plus abstrait et bien souvent la cohérence n'est pas assurée avec l'implémentation.
3. L'architecture détaillée est généralement implicitement décrite dans le code des programmes et celui-ci est unique.

Des technologies différentes sont utilisées pour décrire le logiciel à ces niveaux de granularité. Ci-dessous les principales approches sont présentées. Seule la granularité moyenne et la granularité forte sont étudiées car cette thèse se focalise sur la programmation globale.

### III.2.2 Architecture globale et systèmes d'exploitation

Les systèmes de fichiers sont les constructions les plus rudimentaires utilisées pour décrire l'architecture des logiciels. L'idée est simple ; il s'agit de traduire l'architecture globale en une structure de répertoires et de fichiers. Grossièrement un module est assimilé à un fichier et un sous-système est assimilé à un répertoire. Dans bien des cas, au bout de plusieurs années de maintenance, cette structure devient le seul résidu de l'architecture globale du logiciel[1].

Cette approche présente l'avantage d'être très facile à mettre en oeuvre. Cependant les constructeurs proposés par les systèmes de gestions de fichiers sont très pauvres. Les fichiers correspondant au texte source du logiciel sont noyés dans le système de fichiers. Il est alors indispensable de prendre des conventions pour représenter les différentes informations. Par exemple l'extension du nom du fichier détermine le type de son contenu. Ces conventions varient très largement d'un projet à l'autre ; ce qui rend d'autant plus difficile l'interprétation de structures aussi pauvres.

### III.2.3 Architecture détaillée et langages de programmation

La décomposition du logiciel en différentes unités est une préoccupation qui est rapidement apparue ; au départ essentiellement pour des problèmes pragmatiques ; gérer d'un seul bloc des programmes volumineux n'était pas possible. Des mécanismes basiques et rudimentaires ont été intégrés à la technologie des langages de programmation pour décomposer le texte source : c'est le cas de l'inclusion textuelle (Section IV.2.2.1) et de la compilation indépendante. Fortran déjà proposait de tels mécanismes.

Ce n'est que par la suite que le problème de l'architecture détaillée a été abordé d'un point de vue conceptuel. L'introduction de règles de visibilités plus ou moins complexes [WolfCW88] dans les langages de programmation permettait de mieux contrôler le flot des ressources (via la déclaration de blocs et de ressources locales par exemple).

Cette approche a finalement donné lieu aux langages modulaires dont Ada est aujourd'hui le meilleur représentant. Pour beaucoup il s'agit du premier langage réellement conçu pour la programmation globale et le génie logiciel.

La notion de module, intégrée à ces langages, s'accompagne de la distinction entre interface et implémentation mais aussi du mécanisme de compilation séparée qui, contrairement à la compilation indépendante, effectue des vérifications de cohérence inter-modules. Dans de tels langages des règles de visibilités complexes permettent de contrôler le flot de ressources, non seulement à l'intérieur des modules, mais surtout entre modules [Favr88] [Call91].

1. Cette approche ne permet pas de décrire l'architecture détaillée car le grain est un fichier (il est techniquement impossible de représenter chaque ressource dans un fichier indépendant). Elle ne permet pas non plus de représenter les relations entre ressources.

### III.2.4 Architecture et langages d'interconnexion de modules

Comme il la déjà été signalé, historiquement la distinction entre la programmation globale et la programmation détaillée est liée à l'apparition de MIL-75 [DereKron76] le premier langage d'interconnexion de modules (M.I.L.). Par la suite différents M.I.L. ont été proposés [Tich85] [Perr87] [PietNeig89] [ChoiScac89] [PietNeig89] [SchwStra93]. Il semble cependant qu'ils n'aient pas eu d'impact majeur sur l'industrie du logiciel. Cela est lié essentiellement à des problèmes d'intégration et de cohérence :

- Les langages d'interconnexion de modules décrivent entre autre l'architecture détaillée du logiciel. Comme il s'agit de documents indépendants du source des programmes, leur mise à jour demande un travail supplémentaire et, plus grave, il n'y pas toujours de contrôle de cohérence entre le flot de ressources décrit et celui présent implicitement dans le texte source.
- Avec l'apparition des langages modulaires et des bases logicielles, les langages d'interconnexion de modules dans leur forme la plus simple sont devenus redondants. Par exemple l'une de leurs fonctions était d'effectuer les contrôles de type inter-modules pour pallier aux déficiences des langages de l'époque. Cette fonctionnalité a été directement introduite dans les langages de programmation modulaire. Dans [Tich92] W. Tichy, explique l'échec du M.I.L. Intercol dans ce contexte.
- Les langages d'interconnexion de modules sont à la base de nature descriptive. Cette fonctionnalité prise de manière isolée n'est pas suffisante. De nombreux autres problèmes sont liés à l'architecture (versionnement, stockage des documents, contrôle de la coopération entre agents) et il est alors naturel d'essayer de les résoudre de manière uniforme. Cette constatation a donné lieu à l'intégration de M.I.L. dans des environnements intégrés [Lewe88] [HabeNotk86] et aux bases logicielles.

Aujourd'hui certains langages d'interconnexion de modules subsistent [TrygCG93] [TrygGull95], mais avec parfois des objectifs quelque peu différents. Dans [Dean93] le lecteur trouvera un état de l'art récent.

### III.2.5 Architecture globale et bases logicielles

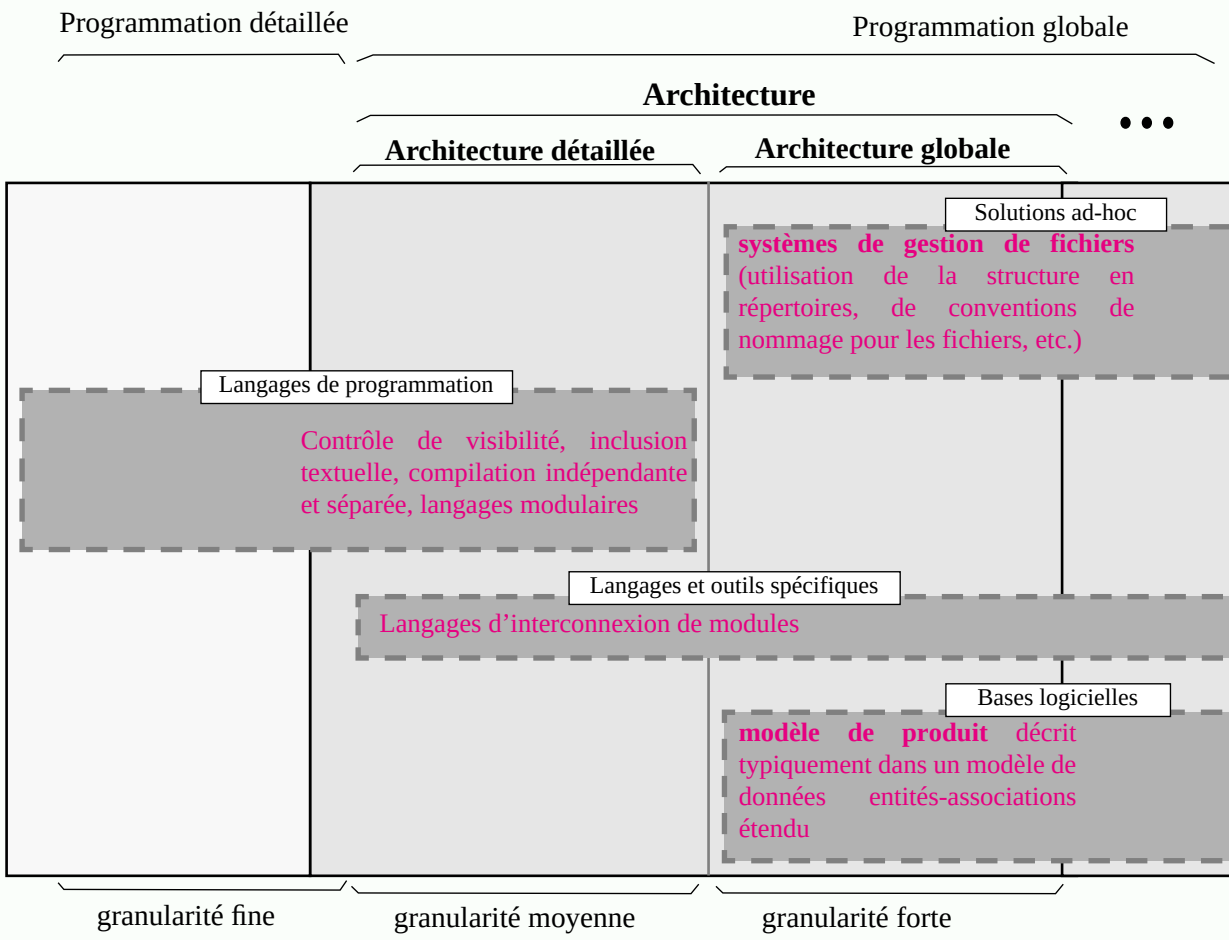
Pendant longtemps le texte source des programmes a été l'artefact principal produit lors du développement des logiciels. Par la suite c'est à l'ensemble des documents formant le produit logiciel que l'on s'est intéressé. Une grande masse d'objets devait alors être manipulée ; décrire leur type et leurs caractéristiques est devenu indispensable. Parallèlement à cela, l'évolution de ces objets a donné lieu au besoin de stocker les différentes versions produites.
Les bases de données logicielles sont issues de ces deux tendances : (1) pouvoir gérer les problèmes de versionnement, (2) utiliser des modèles de données suffisamment riches pour décrire les différentes entités manipulées dans un projet logiciel. Au départ le schéma de données était souvent figé [Estu85] [HabeNotk86]. Par la suite les différents systèmes se sont basés sur l'utilisation de modèle de données plus généraux (essentiellement des modèles entités associations éventuellement étendus par des concepts orientés objets). Ces modèles de données permettent de décrire un ***modèle de produit logiciel*** adapté à chaque projet[1].
Les modèles de données proposés permettent de décrire l'architecture globale des logiciels. Par exemple l'une des fonctionnalités basiques d'une base logicielle est de conserver les relations

entre les différents modules (par exemple la relation de dépendance).
Pour des raisons techniques, dans presque tous les cas, une granularité forte est utilisée : souvent l'unité de stockage est équivalente au fichier. En tout cas des problèmes de performance rendent irréalisable le stockage d'un très grand nombre d'entités de plus faible grain [Lint84] [EmmeSW93]. Généralement le grain le plus fin est donc le module ou l'unité de compilation. Dans ces conditions il n'est pas possible de traiter le problème de l'architecture détaillée.

### III.2.6 Synthèse

Les différentes approches présentées ci-dessus permettent toutes, d'une manière où d'une autre, de représenter (partiellement) l'architecture du logiciel (figure 85).

*figure 85* Techniques de représentation de l'architecture




Ces différentes approches ne sont pas incompatibles ; elles sont même complémentaires. En pratique bien des configurations sont multi-technologiques. Par exemple la solution la plus simple consiste à utiliser pour l'architecture détaillée un langage de programmation et pour l'architecture globale une structure en termes de répertoires. Dans d'autres cas un langage d'interconnexion de module est intégré à une base logicielle [HabeNotk86] [Lewe88].

1. Un modèle de produit logiciel est un schéma de données dans la terminologie base de données ; il décrit le type des entités et des associations utilisées dans un projet logiciel. Le terme "logiciel" est souvent omis.

# III.3 Manufacture

Dans un premier temps cette section présente les principaux concepts relatifs à la manufacture des logiciels et les compare au modèle abstrait (III.3.1).
Ensuite les techniques de description de la manufacture sont étudiées pour les 4 approches identifiées :

- III.3.2 : "*Systèmes d'exploitation*" : l'utilisation des langages de commandes ;
- III.3.3 : "*Langages et outils spécifiques*" : les langages de manufacture (par exemple Make) ;
- III.3.4 : "*Langages de programmation*" : les directives de compilation ;
- III.3.5 : "*Bases logicielles*" : les mécanismes d'activation d'outils.

## III.3.1 Concepts

Etudions tout d'abord la liaison entre la manufacture et le modèle abstrait présenté.

### III.3.1.1 *Manufacture et modèle abstrait*

En ce qui concerne la manufacture, nous avons utilisé dans le modèle abstrait une terminologie issue du génie logiciel : ***objets sources*** et ***objets dérivés*** (Section II.2.4, (p.89)). La liaison est donc directe. Concrètement le texte source d'un programme est un objet source (sa production requière une intervention humaine). Par contre un binaire exécutable est un objet dérivé.

Les opérateurs du modèle correspondent aux ***outils de dérivation***. Il s'agit typiquement d'un compilateur, d'un éditeur de liens, etc. Ces outils sont appliqués à des objets sources ou dérivés et retournent des objets dérivés. Pour être plus précis ce sont des *opérateurs atomiques* ; "atomiques" dans le sens où l'on ne s'intéresse pas à la manière dont ils ont été réalisés[1].

Au contraire les *opérateurs structurés* sont définis à partir d'opérateurs atomiques. D'un point de vue concret ils correspondent par exemple aux fichiers de commandes permettant d'enchaîner plusieurs étapes de manufacture.

En simplifiant, un ***plan de manufacture*** est un graphe dirigé sans cycle décrivant le processus de manufacture pour un ensemble d'objets sources donné. Les noeuds de ce graphe sont les objets.
Les arcs forment la ***relation de dérivation***[2] ; ils sont décorés par les outils de dérivation. Les sources du graphe sont les objets sources.
Décrire le processus de manufacture revient à décrire ce graphe. Un modèle détaillé de manufacture est trouvé dans [Bori86].

1. La distinction entre opérateurs atomiques et opérateurs structurés a été introduite dans la Section II.2.4, (p.89)
2. Cette relation définie entre objets sources et objet dérivés ne doit pas être confondue avec celle communément utilisée dans le domaine de la gestion de versions où l'on dit qu'une version est dérivée d'une autre lorsque elle est le résultat d'une transformation effectuée par un agent humain. Rappelons que c'est en autre pour éviter cette confusion que le terme "manufacture" a été sélectionné plutôt que "dérivation".

### III.3.1.2 *Principaux problèmes concrets*

Les principaux problèmes de la manufacture sont :

- *Cohérence*. Il doit être possible d'assurer, à un instant déterminé, la cohérence entre les objets dérivés et les objets sources. D'un point de vue pratique ceci correspond par exemple à maintenir "à jour" les codes objets par rapport aux textes sources.
- *Efficacité*. Alors que dans le modèle abstrait aucune référence n'était faite au problème d'efficacité, c'est un problème majeur ; entre autre pour ses répercutions sur l'architecture (Section I.4.4.2).
- *Tracabilité*. Les objets dérivés doivent être identifiés précisément et il doit être possible de déterminer quels sont les dérivations et les objets sources qui ont permis sa production (Le scénario présenté à la fin du Chapitre II a montré l'importance d'une telle trace pour détecter l'erreur "monsire")
- *Description claire et concise*. Lorsque le plan de manufacture met en jeu des milliers d'objets la concision et la clarté de sa description sont importantes.

Bien entendu le problème primordial est la cohérence : il est inadmissible de livrer un logiciel incorrectement manufacturé. Les problèmes d'efficacité jouent également un rôle important (Section I.4.4.2). Ils donnent lieu au besoin d'une ***manufacture incrémentale*** (ou ***manufacture sélective***) : on cherche à éviter d'appliquer systématiquement le processus manufacturier dans son intégralité. Nous avons déjà présenté les opérations correspondantes dans le cadre du modèle abstrait (Section II.6.3, "Opérations et manipulation d'objets dérivés et d'opérateurs"). Rappelons simplement que l'incrémentalité en entrée et en sortie peuvent être abordées via les techniques de découpage ou de mémorisation.

### III.3.1.3 *Représentation des opérateurs*

Les opérateurs étant des fonctions, ils sont représentés par des (o-)programmes (Section II.3.2). Nous allons voir que cette idée abstraite correspond à une réalité pratique.

D'un point de vue concret, un fichier de commandes ou un makefile peuvent être vus comme des programmes manufacturiers (écrits respectivement dans un langage de commandes (III.3.2) et dans le langage de manufacture Make (III.3.3)). Un interpréteur correspond à chacun de ces langages. La notion de compilateur est également utilisée en pratique : Make joue souvent le rôle du langage cible et des makefiles sont générés à partir d'autres langages.

Le modèle abstrait présenté est basé sur une vision *fonctionnelle* des transformations : (1) les opérateurs sont vus comme des fonctions ; (2) ils ne génèrent pas d'effets de bord ; (3) il n'y a pas de notion de mémoire ni d'état. En pratique le paradigme impératif et le paradigme déclaratif sont également utilisés[1]. Nous avons déjà noté dans la Section II.6.3 que l'incrémentalité était un problème difficile dans le cas des langages impératifs.

1. Les principales caractéristiques de ces paradigmes sont présentés dans l'Annexe A.2.2, "Classes de langages de programmation".

#### III.3.1.4 *Génie manufacturier*

Puisque les opérateurs structurés sont des (o-)programmes, les problèmes rencontrés lors de leur développement peuvent éventuellement être les mêmes que ceux rencontrés lors du développement de p-programmes.

C'est effectivement le cas : lorsque la description de la manufacture devient particulièrement complexe, les techniques utilisées dans des cas simples ne sont plus suffisantes. Il est nécessaire de définir de nouvelles abstractions pour contrôler leur complexité ; il faut pouvoir décrire l'architecture de ces programmes de manufacture.

Ces remarques suggèrent que la distinction entre programmation détaillée et programmation globale vaut aussi pour la manufacture! On peut alors distinguer la ***manufacture détaillée*** de la ***manufacture globale***.

Il ne s'agit pas que d'une vision de l'esprit ; ce problème est bien réel, même s'il n'est pas formalisé. La description de la manufacture de certains logiciels met en oeuvre des centaines de programmes de manufacture[1]. Cette décomposition est liée au fait qu'une description monolithique n'est pas viable. Même s'il n'existe pas un grand nombre de composants, il n'est pas facile de déterminer quelles sont leurs interactions car celles-ci se font généralement via des mécanismes de bas niveau (par exemple un fichier de commandes permet d'en appeler un autre mais l'échange de données volumineuses doit se faire par effet de bord dans le système de fichiers).

#### III.3.1.5 *Synthèse*

L'intérêt des concepts présentés ci-dessus est multiple. Tout d'abord ceux-ci soulignent le fait que les problèmes de manufacture et d'architecture sont étroitement liés. Ensuite et surtout, il montre que les concepts de programmation détaillée peuvent être utilisés dans le cadre de la programmation globale.
Les 4 sections suivantes présentent les différentes approches proposées.

### III.3.2 Manufacture et systèmes d'exploitation

Une fois de plus la technologie des systèmes d'exploitation est la solution la plus rudimentaire mais de loin la plus fréquente.

*La quasi totalité des opérateurs atomiques (i.e. des outils de dérivation) utilisés en pratique sont prévus pour interagir directement avec un système de fichiers.*

Les o-programmes (c'est à dire la représentation des opérateurs structurés) sont écrits dans des ***langages de commandes***. Ces ***fichiers de commandes*** permettent de générer les objets dérivés. Chaque commande correspond à l'invocation d'un outil de dérivation.

Dans la plupart des cas les langages de commandes ont été construits de manière totalement empirique, sans définir formellement ni syntaxe, ni sémantique. Certains sont réellement

1. Par exemple plus de 400 makefiles sont utilisés dans la génération du système graphique X11R5.

ésotériques [Krak87]. Ils utilisent des mécanismes de très bas niveau (macro-substitution, etc.). Comme il s'agit de langages pouvant être utilisés interactivement, l'un des objectifs est de pouvoir utiliser le minimum de caractères pour décrire une commande donnée ; ce qui bien évidemment donne lieu à des langages cryptiques. Bref, la maintenance des programmes écrits dans ces langages n'a, à aucun moment, été une préoccupation pour leurs concepteurs.

Ces langages sont impératifs. Les outils de dérivation constituent les instructions de bases. Les structures de contrôle dépendent du langage utilisé. Il n'y a pas nécessairement de procédures.
Les données manipulées par ces langages sont des fichiers ou typiquement des chaînes de caractères. Ces données ne sont pas typées.
Le système de fichiers constitue la *mémoire* de ces programmes. Il s'agit d'une mémoire globale et l'exécution d'un fichier de commandes fonctionne uniquement par effet de bord dans cette mémoire.

La manufacture incrémentale est irréalisable : un fichier de commandes, de par sa nature impérative, exécute la totalité des commandes qu'il contient. Cet inconvénient rend l'utilisation des langages de commandes inadaptée comme solution unique.
*Le fait que les outils de dérivation existants soient prévus pour interagir avec les systèmes de fichiers rend indispensable une approche multi-technologique ; les problèmes d'intégration sont donc inévitables.*

### III.3.3 Manufacture et langages de manufacture

La manufacture incrémentale étant un problème essentiel, des outils spécifiques ont été développés. Il s'agit des ***langages de manufacture***.

Make est l'un des premiers et sans nul doute le plus représentatif [Feld79]. Développé dans la première moitié des années 70[1], il reste, 20 ans après, l'un des outils de programmation globale le plus populaire et le plus utilisé. Ce succès s'explique par sa simplicité de mise en oeuvre et par le fait qu'il permet, à un moindre coût, une manufacture incrémentale.
Il s'agit d'un langage hybride, mêlant impératif et déclaratif. Un makefile (i.e. un programme exprimé en langage Make) est un ensemble de règles. Chaque règle décrit comment un but (dans ce cas un objet dérivé) peut être obtenu à partir de sous-buts (dans ce cas un objet source ou dérivé) et de l'exécution d'une suite d'instructions (dans ce cas les invocations des outils de dérivation). En fait un makefile est une représentation proche du graphe de manufacture : les objets forment les noeuds et les instructions à exécuter décorent les arcs.
La machine d'exécution make (i.e. l'interpréteur make) est basée sur le changement d'état du système de fichiers et maintient la contrainte suivante : la date de modification d'un objet dérivé doit être postérieure aux dates de modifications des objets à partir desquels il est obtenu.
Les inconvénients de make ont été très largement décrits dans la littérature. Un grand nombre de substituts ont été proposés, généralement en utilisant un principe similaire, mais aucun ne semble s'être imposé à ce jour [MahlLamp88], [SingBrer92].

1. Bien que le premier article date de 1979, il était déjà utilisé dans les années 75.

Quelques systèmes, au contraire, ont pris comme point de départ une approche fonctionnelle. C'est le cas par exemple de Vesta, un système développé par DEC [LeviMcjo93]. Ce système utilise un langage de programmation fonctionnel polymorphique à typage dynamique[1] [HannLevi93]. Modéliser les outils de dérivation est un problème. Comme nous l'avons vu ce n'est pas à proprement parler des fonctions pures... Les outils de dérivation sont encapsulés afin de les typer et de contrôler les effets de bord [BrowLevi93]. La notion d'état est remplacée par celle d'environnement : les opérateurs prennent en entrée un environnement et retournent des environnements. La manufacture incrémentale est assurée via le principe de mémorisation (système de cache) ( Section II.6.3, Annexe A.3.6, "Mémorisation").

### III.3.4 Manufacture et langages de programmation

A l'origine, les problèmes de manufacture se sont surtout manifestés dans le cas des programmes[2] mais ils sont valables pour la production de n'importe quel type de document (par exemple le formatage de texte à partir de processeurs comme TeX, etc.).

La technologie des langages de programmation n'a pas réellement été modifiée par l'apparition des problèmes de manufacture[3]. En fait c'est l'architecture d'un logiciel qui détermine sa manufacture, tout au moins partiellement. Par exemple si un mode de compilation séparée est utilisé, la relation de dépendance entre unité de compilation détermine un ordre partiel pour ces compilations. Il est alors possible de produire automatiquement un opérateur structuré réalisant la manufacture du logiciel. La plupart des environnements de programmation offrent la possibilité de générer des makefiles.

### III.3.5 Manufacture et bases logicielles

Comme il l'a été dit, l'ambition des bases logicielles couvre souvent tout le spectre de la programmation globale.
L'une des premières fonctionnalités pouvant être offerte est le stockage des objets sources et dérivés mais surtout leur identification.
Dans certains cas, les outils de dérivation sont eux-mêmes sous le contrôle de la base et peuvent être gérés comme des objets. Ceci permet entre autre de traiter le problème de l'évolution des outils.

Certaines bases logicielles sont actives. Elles sont basées sur la notion de déclencheurs [Belk88]. Lorsqu'un événement est détecté (par exemple lorsqu'un objet est modifié), l'action associée à un déclencheur est automatiquement exécutée. Cette action peut elle-même provoquer d'autres événements, donc déclencher d'autres actions.
Dans le cas de la manufacture, l'idée est d'associer à l'événement de modification d'un objet

1. Le typage permet de contrôler de manière fine le déroulement correct de la manufacture. Le polymorphisme est nécessaire dans la mesure ou les outils ne retournent pas toujours les mêmes résultats. C'est également la raison pour laquelle le contrôle de type est dynamique.
2. Le titre de l'article décrivant l'outil Make est représentatif "A program to maintain computer programs" [Feld79].
3. Si ce n'est peut être sous la forme de directives de compilations inclues en commentaires ou encore sous la forme de la clause **separate** du langage ADA qui est lié à la notion de manufacture plutôt qu'à celle d'architecture. Dans [Wink86], une extension d'Ada est proposée dans laquelle certains aspects de la description de la manufacture vont avec la description des versions.

source l'application des outils de dérivation correspondant. Une telle description de la manufacture est déclarative. L'enchaînement des différentes actions n'est pas définie explicitement. Dans de telles conditions la manufacture incrémentale est possible. Cette approche implique que toutes les actions soient effectuées dans la base[1].

L'expérience acquise avec le système Adele  montre

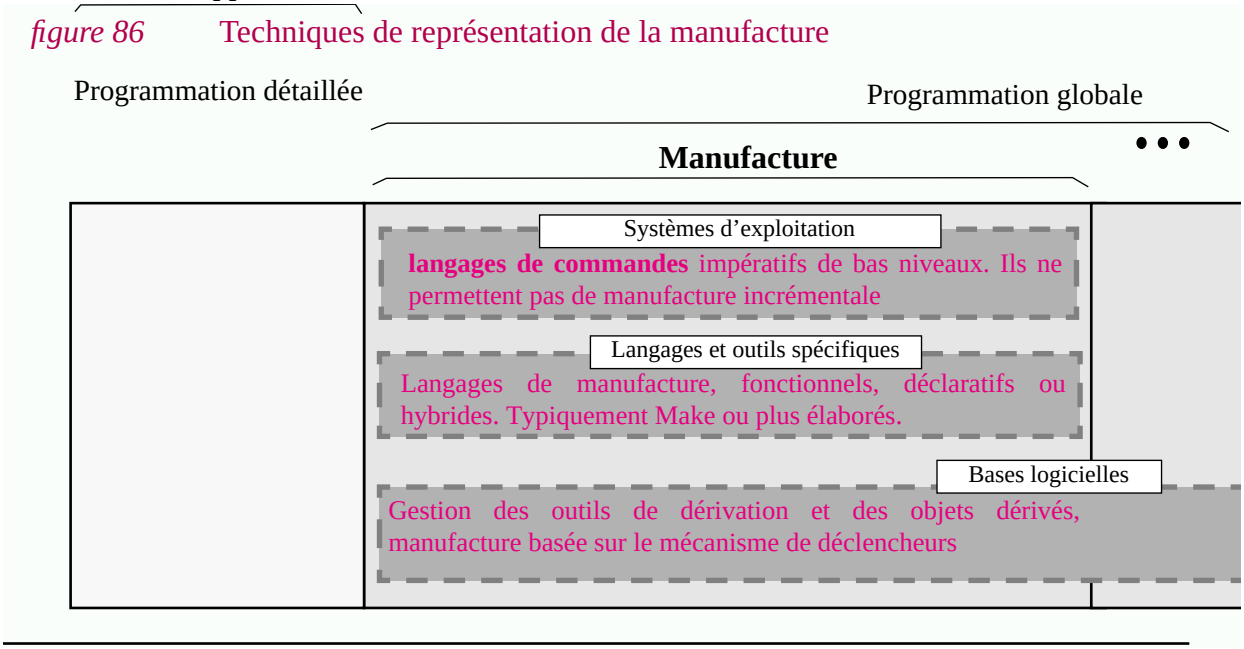
- (1) que le contrôle de la manufacture dans la base logicielle n'est pas commode. Les outils doivent être encapsulés pour masquer le fait qu'ils agissent normalement sur un système de fichiers et pour prendre en compte le passage de paramètres,
- (2) que la description répartie de la manufacture ne facilite pas sa compréhension,
- (3) que les déclencheurs fournissent un mécanisme de programmation extrêmement puissant, mais souvent difficile à contrôler. Des outils de mise au point sont nécessaires.

Pour pallier à ces inconvénients le système Adele propose une approche plus classique pour la manufacture : le système Madele génère un makefile à partir des relations de dépendances contenues dans la base logicielle. Adele est une base logicielle multi-technologie : elle gère l'intégration de la technologie des systèmes de fichiers et de celle de la base.

## III.3.6 Synthèse

Les différentes approches concernant la manufacture sont résumées dans la suivante.

*figure 86* Techniques de représentation de la manufacture




1. Celle-ci joue le rôle de la mémoire globale, tout comme le système de fichiers jouait le rôle de mémoire globale pour les langages de commandes.

# III.4 Variation : concepts

## III.4.1 Introduction

Nous avons souligné dans les chapitres antérieurs que le versionnement était un thème interdisciplinaire[1]. Dans ce chapitre nous concentrons sur les spécificités du logiciel. Même dans ce cadre particulier, nous ne cherchons pas à présenter la notion de version de manière exhaustive. Une vision bien particulière est offerte :

- *Une vision statique*. Les aspects de programmation coopérative ne sont pas pris en compte (problèmes d'accès parallèles, de politiques de création de versions, etc.) Par exemple, un système comme SCCS [Roch74] sera décrit en fonction de la structure des informations qu'il permet de stocker sans s'attacher aux problèmes de "checkin/checkout" ou d'espaces de travail.
- *Variation vs. évolution.* Cette section étudie surtout les problèmes de variations et considère donc essentiellement les versions logiques et plus généralement les variantes. C'est dans ce cadre que la ré-ingénierie est le plus utile (Section I.6.2). Néanmoins la technologie utilisée pour résoudre les problèmes d'évolution n'est pas très différente.
- *Variation et manufacture*. Les problèmes de manufacture sont particulièrement importants dans un contexte multi-versions. Les différentes possibilités de versionnement sont étudiées autant pour les objets sources que pour les objets dérivés.
- *Variation et architecture.* Une attention toute particulière est apportée à l'étude des liaisons existant entre la granularité et le versionnement. On parlera de variation globale et de variation détaillée (Section III.4.6).

Le modèle présenté dans le Chapitre I fait abstraction de nombreux problèmes. Ce chapitre a pour vocation de remédier à ces lacunes. Jusque là nous avons essentiellement abordé le "quoi ?", le "pourquoi ?" et le "comment ?". De nouvelles questions vont être traitées. C'est le cas notamment du "*qui ?*", du "*où ?*" et du "*quand ?*". Par exemple :

- "*Qui utilise les objets génériques ?*" (développeur, utilisateur,...),
- "*Où la manufacture a-t-elle lieu ?*" (site de développement, d'installation,...),
- "*Quand les choix sont-ils faits ?*" (lors de la manufacture, de l'exécution,...).

Ces différents facteurs doivent être traités pour pouvoir comprendre la complexité de la programmation globale et surtout la grande diversité des techniques employées (considérer par exemple la figure 100 (p.195)).

1. Par exemple nous avons vu que la notion de dimension apparaissait dans le cadre des bases de données temporelles, des bases de données géographiques, de la conception assistée par ordinateur et du génie logiciel (Section II.4.3).

## III.4.2 Le contexte du logiciel et ses variations (Pourquoi ? Pour qui ?)

L'introduction des objets génériques dans le modèle abstrait a été justifié par le fait qu'un objet était utilisé dans un *contexte* pouvant varier (se référer à la figure 30 (p.90)). Il est donc naturel d'étudier quel est le contexte du logiciel et quelles sont les raisons expliquant ces variations.

### III.4.2.1 *Les variations du logiciel (Pourquoi ? Pour qui ?)*

Le contexte du logiciel se compose d'*aspects logiques* et d'*aspects techniques* (logiciels ou matériels). On parlera alors de ***variations logiques*** et de ***variations techniques***.
De manière orthogonale trois classes peuvent être distinguées : (1) les variations liées aux utilisateurs clients, (2) aux organisations clientes et (3) au fournisseur du logiciel. Ces deux dimensions sont présentées dans figure 87 (faire l'analogie avec la figure 30.c (page 90)).

Remarquons que même si le logiciel est destiné aux clients, différentes variations sont dues au fournisseur : le logiciel n'est pas seulement exécuté, mais aussi manufacturé, testé, analysé, etc. Pour faciliter le déroulement de ces activités le fournisseur peut être amené à introduire des variations dans le produit logiciel ((c) et (f)).

Bien que la classification proposée ne soit pas stricte elle présente l'avantage de faire ressortir deux caractéristiques essentielles du contexte du logiciel :

- *un grand nombre de variations*. Le domaine des objets génériques est particulièrement complexe (de très nombreuses dimensions sont utilisées[1]).
- *la grande diversité des variations*. Les variations sont dues à de multiples facteurs et ont des caractéristiques parfois très différentes. Par exemple, la fréquence de variation n'est pas la même pour toutes les dimensions (voir la section suivante).

### III.4.2.2 *Fréquence de variation (Quand ?)*

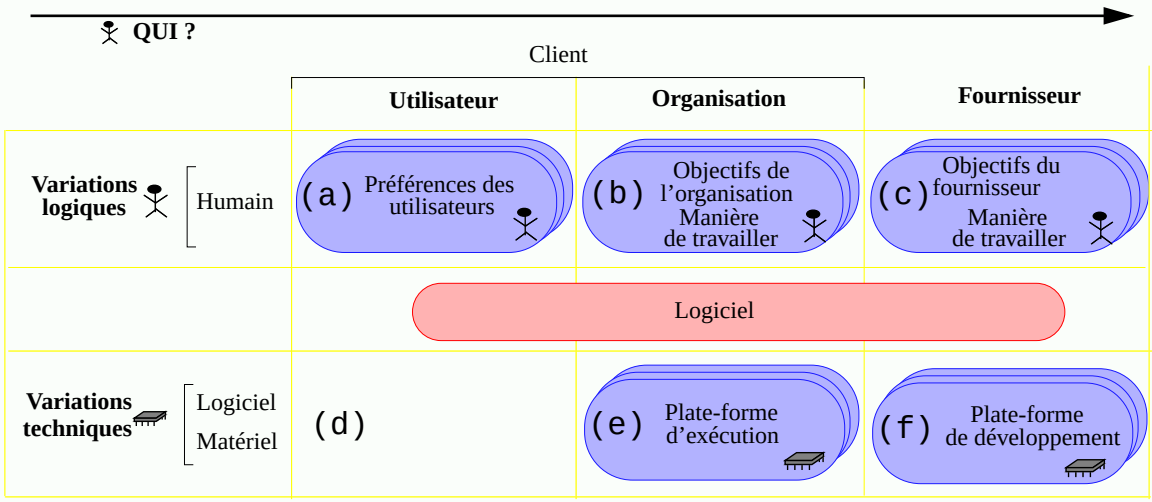
La ***fréquence de variation*** est liée à la période de temps au cours de laquelle un choix est constant. L'ordre de grandeur peut aller de la seconde à plusieurs années comme le montrent les exemples présentés dans la figure 88. Ces aspects, ignorés lors de la définition du modèle abstrait, expliquent en partie la diversité des techniques mises en oeuvre pour répondre à ces variations.

## III.4.3 Spécialisation incrémentale du logiciel (Quand ?)

L'opération de *spécialisation d'objets génériques* a été présentée dans le cadre du modèle abstrait (Section II.6.4.3, (p.124)) mais l'importance de celle-ci n'a pas été soulignée. Nous allons maintenant voir que cette opération joue un rôle fondamental lorsque l'on s'intéresse à la variation de programmes[2].

1. Des dizaines voir des centaines pour certains logiciels.
2. Il nous semble cependant qu'une grande partie des concepts présentés ci-dessous ont leur équivalent dans d'autres domaines. Cette idée n'a pas été approfondie dans le cadre de cette thèse.

*gure 87* Classifications des variations selon deux dimensions




## VARIATIONS TECHNIQUES

### *(d)* Variations techniques dues aux utilisateurs clients

A priori, aucun aspect technique n'est lié à un utilisateur particulier car celui-ci utilise la plate-forme fournie par l'organisation dont il dépend.

### *(e)* Variations techniques dues aux organisations clientes

• ***variation des plates-formes d'exécution*** (matérielle et/ou logicielles).

• ***variation de la plate-forme de manufacture*** (si celle-ci incombe à l'organisation cliente. Si l'organisation cliente ne dispose pas des mêmes outils de dérivation certaines variations sont nécessaires (e.g. pour les logiciels de domaine public livrés sous forme de code source le choix des compilateurs est laissé à l'organisation cliente).

• ***variation des conventions et des besoins techniques***. L'organisation des fichiers en termes de répertoires est potentiellement différente pour chaque site d'installation (chemin d'accès aux codes exécutables, aux ressources communes, etc).

### *(f)* Variations techniques dues au fournisseur

• ***variation des plates-formes et des outils de développement.*** Les avantages et inconvénients des outils de développement sont variés. En pratique on constate parfois des situations assez étonnantes. Lors du développement d'un module, un compilateur A peut être utilisé parce qu'il compile rapidement. Par la suite un compilateur B peut être choisi parce qu'il permet d'utiliser un outil de mise au point puissant. Enfin, avant une livraison, c'est un compilateur C qui servira à générer du code optimisé de bonne qualité.

*figure 88* Exemple de fréquences de variation

**VARIATIONS LOGIQUES**

***(a)*** **Variations logiques dues aux utilisateurs clients**

Un utilisateur devrait pouvoir changer de palette de couleurs ou de format d'impression au gré de sa volonté, tout au long de l'utilisation du logiciel. Ici l'intervalle entre deux changements peut être de l'ordre de l'*heure* où même de la *minute*. Certains de ces choix sont plus stables. C'est le cas par exemple de la langue d'interaction.

***(b)*** **Variations logiques dues aux organisations clientes**

Les contraintes logiques imposées par une organisation sur un logiciel sont normalement plus stables. Au cours des mois ou des années elles peuvent néanmoins changer en fonctions des objectifs de l'organisation et de sa manière de travailler.

***(c)*** **Variations logiques dues au fournisseur**

Supposons qu'un mode trace ait été implanté via l'inclusion d'instructions dans le code source des programmes. Lors de la mise au point il peut être utile d'activer ou de désactiver ce mode interactivement. Il est parfois suffisant de fixer celui-ci pour la durée d'un exécution.

**VARIATIONS TECHNIQUES**

La plate-forme matérielle dont dispose une organisation est normalement relativement stable. Une plate-forme logicielle a tendance à évoluer plus rapidement. La fréquence d'installation de nouveaux services logiciels et de nouvelles versions du système d'exploitation dépend à la fois de leur disponibilité sur le marché et de l'organisation. Généralement l'intervalle entre deux changements peut aller de quelques *mois* à quelques *années*.

### III.4.3.1 *Spécialisation de programmes*

La notion de ***spécialisation de programmes*** provient de la programmation détaillée et est normalement appliquée à des p-programmes. Cette notion est présentée en Annexe A.3.9 et nous nous limiterons donc ici à présenter les éléments essentiels.

#### *Spécialisation, Evaluation partielle et Mixeur...*

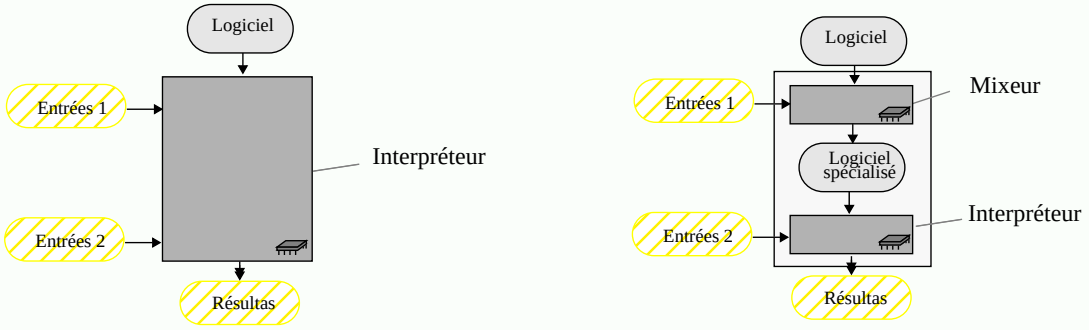
Spécialiser un programme consiste à générer un programme équivalent mais fonctionnant sur un domaine réduit. L'***évaluation partielle*** est une technique correspondant à un cas particulier. Il s'agit de générer un programme spécialisé en fixant la valeur de certains paramètres. Bien évidemment on souhaite obtenir un programme plus simple ou plus efficace.
Un ***mixeur*** prend en entrée le programme à spécialiser ainsi que les valeurs de certains paramètres. Informellement, il interprète tout ce qu'il peut à partir des informations disponibles et ne génère du code que pour ce qui ne peut être résolu dans cette étape. Le code généré peut être écrit dans un langage différent du langage source. Un mixeur réalise donc à la fois une tâche d'interprétation et de compilation (d'où son nom).

#### *Analyse du moment de liaison...*

L'***analyse du moment de liaison***[1] est l'un des problèmes rencontré lors de la spécialisation de programme [JoneGS93]. Il s'agit de déterminer, pour chaque information manipulée par le programme, si celle-ci peut être connue statiquement ou non. L'opposition entre statique et dynamique est relative à la phase de spécialisation : toute information calculable lors de cette phase est dite statique, sinon elle est dynamique.

1. "Binding time analysis"

*figure 89* Spécialisation de programme



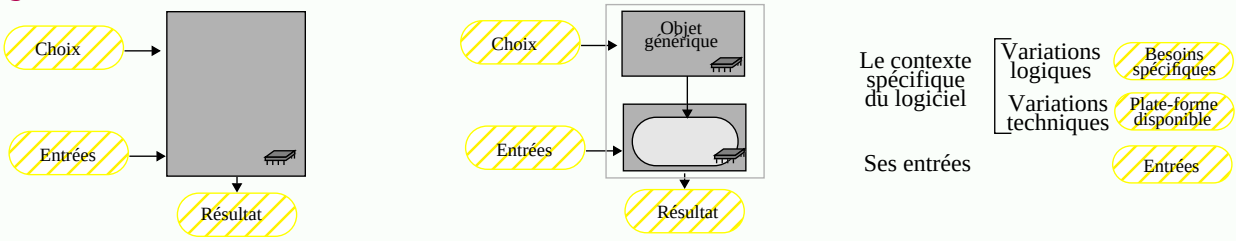

NOTE 1 : Le concept de spécialisation de programme est défini plus en détail dans l'Annexe A.3.9 (figure 149 (p.302)).
NOTE 2 : Les conventions graphiques utilisées dans ces figures et dans les figures suivantes sont décrites dans l'Annexe A.3.1 ("Conventions graphiques"). Elles sont essentielles pour interpréter avec précision des structures complexes où la notion de programmes et données s'intervertit. (Attention notamment à la position des flèches).

### III.4.3.2 *Analogie entre choix et entrées*

Les représentations des objets génériques sont des (g-)programmes (Section II.3.3). Dans le cas du logiciel ces objets génériques génèrent des (p-)programmes. Certains auteurs parlent alors de ***méta-programmes*** [WinkStof88] [WeisCrew93].

Cette remarque n'a pas uniquement pour but d'introduire un terme pompeux. Au contraire il s'agit de faire remarquer une spécificité de la variation de logiciel : un objet générique est une *fonction* qui prend un *choix* en entrée et génère une *fonction.* Cette dernière prend une *entrée*[1] et génère un *résultat* (figure 90.b).
A la place d'avoir deux fonctions, on peut imaginer une seule fonction prenant simultanément le *choix* et l'*entrée* et générant le même *résultat* (figure 90.a).

*figure 90* Choix et entrées




***(a)*** **Une vision abstraite** ***(b)*** **Une vision plus concrète**

Les figures ci-dessus sont basées sur la mise au même niveau (1) de la description du contexte dans lequel le logiciel doit opérer et (2) des entrées sur lequel il doit s'appliquer. Dans les figures suivantes le contexte sera décomposé en variations logiques et variations techniques.
D'un point de vue plus abstrait, remarquons que le passage de la fonction (a) à la fonction dessinée en (b) (plus précisément à l'objet générique) est une transformation bien connue dans le domaine des langages fonctionnels : la curryfication (figure 148 (p.301)) (Annexe A.3.9).

L'idée sous-jacente est qu'il n'existe pas de différence fondamentale entre une entrée de l'utilisateur et un choix (figure 90). Pour simplifier, dans les sections suivantes nous utiliserons souvent le terme "paramètre" pour l'une quelconque de ces possibilités.

1. Fournie par l'utilisateur.

### III.4.3.3 *Production d'un logiciel général, générique ou spécifique*

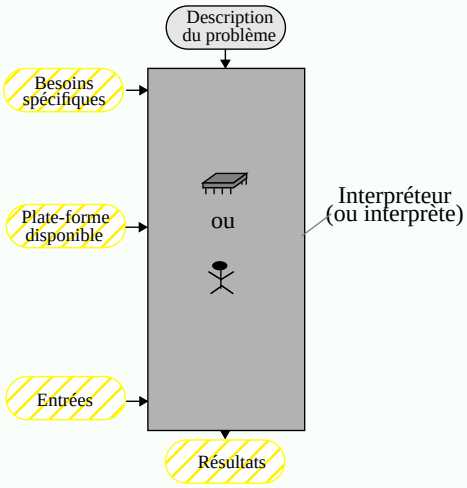
Le raisonnement utilisé dans la section précédente est basé sur le concept de *fonction*. Une fois de plus cette notion va être utilisée accompagnée des concepts associés : programmes, interpréteurs, compilateurs, etc. (Annexe A.3, "Concepts et techniques relatifs aux programmes"). Cette fois, par contre, ce sera pour offrir une vision beaucoup plus globale. Il s'agit de modéliser le cycle de **P**roduction et d'**E**xécution du logiciel ; en allant de la description du problème, du contexte et des entrées, jusqu'à la production des résultats. Dans la suite, pour simplifier, on parlera du ***cycle-PE***.

***D'un point de vue abstrait le cycle-PE peut être vu comme une fonction***

Reste à savoir comment est réalisée cette fonction et quelles sont ses caractéristiques[1].

La description du problème à résoudre (exprimée éventuellement informellement) détermine la fonction calculée. Dans l'absolu cette description peut être vue comme un programme interprété ; par un agent humain ou par une machine (figure 91).

*figure 91* Le cycle-PE vu comme une fonction, la fonction-PE




Il est possible d'imaginer un système capable d'interpréter la description du problème pour un contexte et des entrées donnés. Ici la description du problème joue le rôle du programme.
Deux solutions extrêmes sont envisageables :
- Le cas idéal est celui où l'on dispose d'un *interpréteur*, c'est à dire d'un système exécutant la fonction automatiquement.
- Dans le cas le plus défavorable il s'agit d'un *interprète* (i.e. d'un agent humain) et l'exécution se fait alors "manuellement".

Différentes réalisations sont possibles pour la fonction-PE. Elles différent par le nombre d'étapes utilisées et le moment de liaison des paramètres :

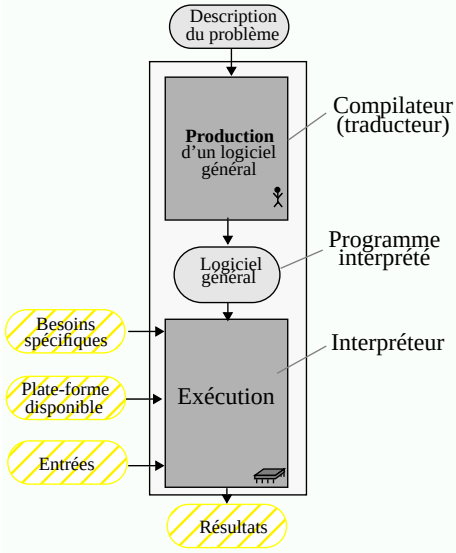
- ***Logiciel général*** : un tel logiciel est utilisable tel quel dans les différents contextes (figure 92.a).
- ***Logiciel générique*** : une phase de génération explicite permet d'obtenir une variante particulière pour un contexte spécifié (figure 92.b).
- ***Logiciel spécifique*** : les spécificités d'un contexte particulier sont prises en compte dès la production du logiciel (figure 92.c).

### III.4.3.4 *Coût de production, efficacité d'exécution et calcul incrémental*

Toutes les solutions présentées dans la figure 92 correspondent à la même fonction-PE. Pourtant elles sont loin d'être équivalentes si l'on considère les trois critères suivants :

1. Cette fonction sera appelée ***fonction-PE*** dans la suite.

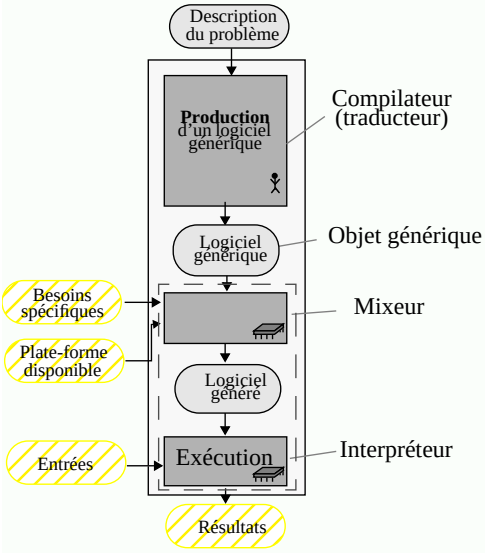
*figure 92* Logiciel général, logiciel générique et logiciel spécifique




### *(a)* Production d'un logiciel général

A partir de la description du problème un logiciel général est élaboré (i.e. produit via une intervention humaine). Ce logiciel général peut être utilisé tel quel dans différents contextes. Il "suffit" pour cela de spécifier le contexte en même temps que les données d'entrées. Ce principe est hélas difficile à mettre en oeuvre car il est difficile de trouver des solutions générales.

Remarquons que la notion de logiciel général correspond à celle d'*objet général* du modèle abstrait (Section II.2.5, (p.90)). L'étape de production du logiciel consiste à traduire la description du problème (exprimée dans un langage formel ou non) en un programme exprimé dans un langage interprétable. Cette idée peut être assimilée à la notion de *compilateur* (ou plutôt de *traducteur*).

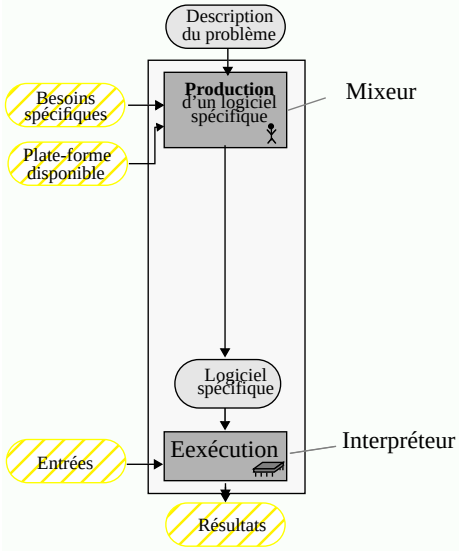



### *(b)* Production d'un logiciel générique

La première phase consiste à produire un logiciel *générique* :

• Tout d'abord il est indispensable de déterminer quelles sont les variations possibles (i.e. de définir le *domaine de l'objet générique*) (II.4). • Ensuite l'objet générique doit être réalisé en utilisant des techniques adéquates.

La deuxième phase correspond à la phase de *génération d'une variante* (au sens du modèle abstrait (II.2.5)). Pour cela il est nécessaire de décrire un contexte spécifique (de *déterminer un choix* selon la terminologie du modèle). Ce problème est étudié plus en détail dans la Section III.4.5. Nous avons vu qu'un objet générique était un programme interprété. Si l'on considère celui-ci comme un méta-programme l'interpréteur est un mixeur : il prend en entrée un (méta)programme et quelques entrées (ici des choix) et génère un programme spécialisé.




### *(c)* Production d'un logiciel spécifique

La production d'un *logiciel spécifique* correspond au cas le plus simple à mettre en oeuvre. Lors d'une telle production, un contexte spécifique est pris en compte En utilisant ces informations, toutes les simplifications possibles sont effectuées. Il ne reste dans le programme généré que le nécessaire. Cette phase peut être assimilée à l'exécution d'un *mixeur*. Cette *spécialisation* s'accompagne d'un changement de langage (on passe d'un langage informel à un langage interprétable) ce qui n'est pas contradictoire avec la notion de mixeur (Annexe A.3.9). Cette phase étant basée sur une activité humaine donc intelligente, les optimisations que l'on peut espérer d'une telle spécialisation sont supérieures à celles généralement obtenues dans le cas d'une spécialisation automatisée.

Un *logiciel spécifique* correspond à la notion d'*objet spécifique* du modèle abstrait (II.2.5). Un tel logiciel est de la même nature qu'une variante générée à partir d'un objet générique (si ce n'est que celui-ci est sans doute plus optimisé pour les raisons citées ci-dessus).

NOTE : Les notions d'*interpréteur* (d'*interprète*), de *compilateur* (de *traducteur*) et de *mixeur* sont respectivement décrites en Annexe A.3.2, Annexe A.3.3 etAnnexe A.3.9.

- la *facilité de production*,
- l'*efficacité d'exécution*,
- la possibilité d'*évolution incrémentale* des paramètres.

TABLEAU 14 Comparaisons entre logiciels généraux, génériques et spécifiques

| Type | (1) Facilité de production | (2) Efficacité d'exécution | (3) Calcul incrémental |
|---|---|---|---|
| Logiciel général | ☹ | ☹ | ☺ |
| Logiciel générique | ☺ | ☺ | ☺ |
| Logiciel spécifique | ☺ | ☺ | ☹ |

La production d'un logiciel général ou générique est souvent plus difficile que dans le cas d'un logiciel spécifique.
Un logiciel général est moins *efficace à l'exécution* qu'un logiciel généré ou qu'un logiciel spécifique : dans le premier cas la description du contexte est interprétée dynamiquement ; dans les autres cas elle est plutôt "compilée".

Si l'on considère seulement ces deux premiers critères, il ressort que les logiciels spécifiques sont les plus avantageux : ils allient *efficacité d'exécution* et *facilité de production*.
L'évaluation est toute autre si l'on considère le troisième critère qui correspond aux problèmes de variations. Les paramètres de la fonction-PE sont susceptibles de changer et comme dans le cas de la manufacture, on est alors confronté au problème du *calcul incrémental*[1].

Bien évidemment plus les paramètres interviennent tôt dans le cycle, plus leur modification entraîne un coût élevé (car il est nécessaire d'effectuer de nouveau de nombreuses transformations). Avec ce critère, les logiciels généraux sont les plus avantageux : la modification du contexte n'implique, ni la génération d'une nouvelle variante, ni la production d'un nouveau logiciel spécifique.

On retrouve donc des résultats intuitifs : la généralité est coûteuse ; la spécificité restrictive. *En fait l'intérêt des logiciels génériques est justement qu'ils correspondent à un compromis entre ces deux extrêmes*.

### III.4.3.5 *Fréquence de variation*

Pour la fonction-PE, comme pour toute fonction sujette au calcul incrémental, la fréquence de variation est un élément essentiel :

- Une fréquence élevée n'est possible qu'au moment de l'exécution. Le traitement d'un problème de variation à ce niveau implique bien sûr une exécution moins efficace.
- Au contraire, si la probabilité pour qu'un paramètre change est très faible, il peut être préférable (ou tout au moins plus simple) d'utiliser une valeur spécifique dès le début du cycle-PE.

*Autrement dit la fréquence de variation d'un paramètre détermine son moment de liaison dans le cycle.*

1. Plus précisément il s'agit ici de l'*incrémentalité horizontal en entrée* : il s'agit d'évaluer de nouveau le résultat d'une fonction après la modification d'un sous-ensemble de ses paramètres.

### III.4.3.6 *Moment de liaison*

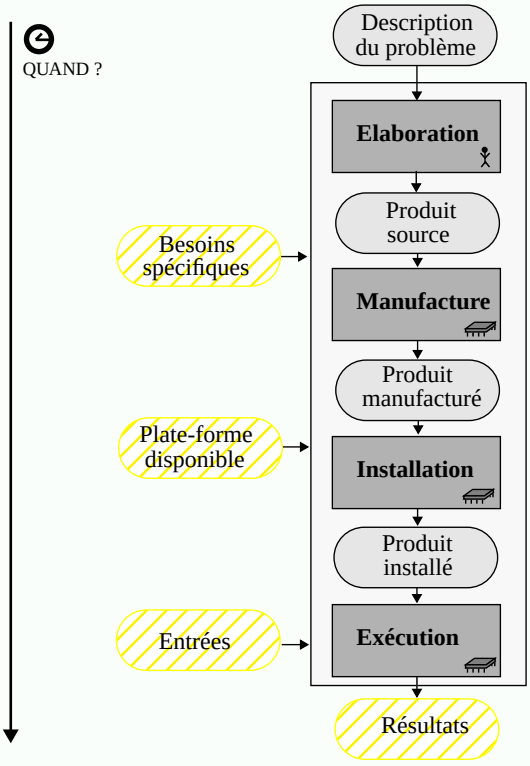
Revenons à la notion de *moment de liaison* dans le cadre de la programmation détaillée. Nous avons vu que dans le cas de l'évaluation partielle, deux alternatives étaient possibles pour le moment de liaison : lors du processus de spécialisation (liaison statique) ou lors de l'exécution du programme (liaison dynamique). Cette distinction statique/dynamique apparaît aussi par exemple pour les techniques de compilation.

En ce qui concerne la programmation globale, les fréquences variations peuvent être très variées (les périodes stables vont de quelques secondes à plusieurs années) (Section III.4.2.2). Les termes "statique" et "dynamique" sont encore utilisés mais c'est toujours par rapport à une phase donnée[1]. L'échelle de temps n'est plus la même non plus.

### III.4.3.7 *Le cycle Elaboration - Manufacture - Installation - Exécution*

Pour que les moments de liaison puissent être définis de manière plus précise, il est nécessaire d'affiner l'échelle de temps du cycle-PE. Considérons le *cycle-EMIE* (i.e. la *fonction-EMIE*) composé(e) de quatre étapes (figure 93) :

*figure 93* Le cycle-EMIE vu comme une fonction, la fonction-EMIE




Ce cycle est un affinement du cycle-PE (figure 82 (p.155)). Il permet de déterminer avec plus de précision le moment de liaison des paramètres. Si on le compare au cycle de vie, il s'agit d'une vision macroscopique en ce qui concerne les premières phases (toutes regroupées dans l'élaboration) et d'une vision plus détaillée en ce qui concerne les dernières étapes.

Bien que souvent confondues, l'étape de manufacture et l'étape d'installation doivent être distinguées. La manufacture a pour but de construire un logiciel mais normalement sans prendre en compte un environnement particulier. Au contraire l'installation correspond à l'intégration d'un produit manufacturé dans un tel environnement. L'analogie peut être faite avec la manufacture (le montage) d'une ampoule électrique et son installation.

- **Elaboration**. L'étape d'*élaboration* regroupe toutes les activités de développement et de maintenance manipulant les *objets sources* du logiciel[2].

1. On parle par exemple d'édition de liens statique ou d'édition de liens dynamique.
2. Ce terme est choisi ici par opposition à la *manufacture* et tente de faire ressortir l'aspect intellectuel et non mécanique des activités mises en jeu. On aurait pu utiliser "développement" où "conception" mais ces termes ont des connotations trop spécifiques. Notons finalement que définie ainsi, l'étape d'élaboration recouvre la quasi totalité du cycle de vie.

- **Manufacture**.
- **Installation**. L'*installation* du logiciel correspond à la phase pendant laquelle celui-ci est intégré dans un environnement particulier. Il s'agit typiquement d'installer un logiciel sur un site client, mais ce peut être aussi sur un autre site de développement. On a donc le choix du "où?".
- **Exécution**. L'exécution du logiciel correspond à la phase finale au cours de laquelle le logiciel est effectivement exécuté[1].

Chaque étape correspond à une transformation plus ou moins complexe. L'élaboration d'un logiciel transforme la description du problème en un *produit source*, ensuite transformé en *produit manufacturé*. L'installation permet d'obtenir un *produit installé* dans un environnement particulier. Finalement l'exécution utilise les données spécifiées par l'utilisateur et retourne le résultat (qui n'est pas de la même nature que le produit logiciel mais qui peut de toute façon être appelé *produit de l'exécution*) (figure 93).

Dans le cas du logiciel le produit élaboré est constitué des codes sources et des documents associés. Le produit manufacturé est généralement un code exécutable. Le produit installé est ce même code exécutable mais intégré dans son futur environnement d'utilisation.

### III.4.3.8 *Spécialisation incrémentale du logiciel*

La liaison des paramètres peut se faire à chaque étape du cycle-EMIE :
(1) lors de l'**E**laboration il peut être décidé de construire un logiciel ayant des caractéristiques spécifiques ; (2) lors de la **M**anufacture, différents assemblages sont possibles. A cela s'ajoute le choix des outils de dérivation. (3) L'étape d'**I**nstallation permet de fixer encore d'autres contraintes puisque l'environnement du composant est connu. (4) Finalement au cours même de l'**E**xécution (ou juste avant), des choix peuvent encore être faits.
Comme il l'a été dit le moment de liaison d'un paramètre dépend de sa *fréquence de variation* ainsi que d'un compromis entre *coût de production* et *efficacité d'exécution*. La grande diversité des variations fait que la liaison des paramètres est répartie tout au long du cycle-PU.

Ce cycle peut être vu comme une série de ***spécialisations incrémentales du logiciel***. Plus de 8 étapes sont distinguées dans l'exemple présenté à la fin de ce chapitre (figure 100 (p.195))[2]. Toutes ces étapes ne sont pas des spécialisations, mais la plupart s'expliquent par le fait que le contexte du logiciel est déterminé incrémentalement.

### III.4.3.9 *Une généralisation du concept de moment de liaison*

Jusqu'à maintenant pour décrire les réalisations de la fonction-PE nous nous sommes focalisés sur le temps. Il est intéressant de constater que d'autres variables interviennent dans le cadre de la programmation globale et coopérative. On ne s'intéresse pas seulement au *moment de liaison* (le *quand?*), mais aussi au *lieu* (le *ou?*) et aux *personnes* responsables de cette liaison (le *qui?*). Ces

1. Cette phase n'a pas été appelée "utilisation" car ici on considère autant l'exécution du logiciel pour le client (son utilisation), que pour le testeur (sa validation).
2. Sans compter le fait que ce phénomène se produit également dans la phase d'élaboration.

aspects, normalement ignorés dans le domaine de la programmation détaillée, sont étudiés dans les sections suivantes car ils donnent lieu à des techniques de variations différentes.

### III.4.4 Les problèmes de livraisons (Qui? Où? Comment?)

Concrètement *qui* exécute la fonction-PE ? Deux réponses sont possibles : le fournisseur ou le client. Cette dichotomie donne lieu à la notion de livraison.

La ***livraison du logiciel*** correspond au moment où le logiciel passe du fournisseur au client[1]. Cette notion qui n'apparaît pas dans le modèle abstrait marque pourtant une frontière importante en pratique : (1) les techniques de variations ne sont pas les mêmes pour le fournisseur et pour le client, (2) il faut pouvoir contrôler le degré de variation laissé au client.
Des techniques spécifiques ont été développées pour résoudre ces problèmes[2] et cet aspect a donc sa place dans cette thèse.

La livraison du logiciel se caractérise par un changement de réponse à la question "***qui*** a le contrôle du logiciel?". En ce qui concerne le "***où***?" notons qu'une livraison s'accompagne dans bien des cas d'un changement de site : le logiciel passe par exemple du site de développement au site d'installation. "***Quand*** la livraison est-elle faite ?". Il s'agit d'interpréter cette question par rapport à l'échelle de temps du cycle-EMIE.
La figure 94 présente le cas particulier de la livraison de produits manufacturés.

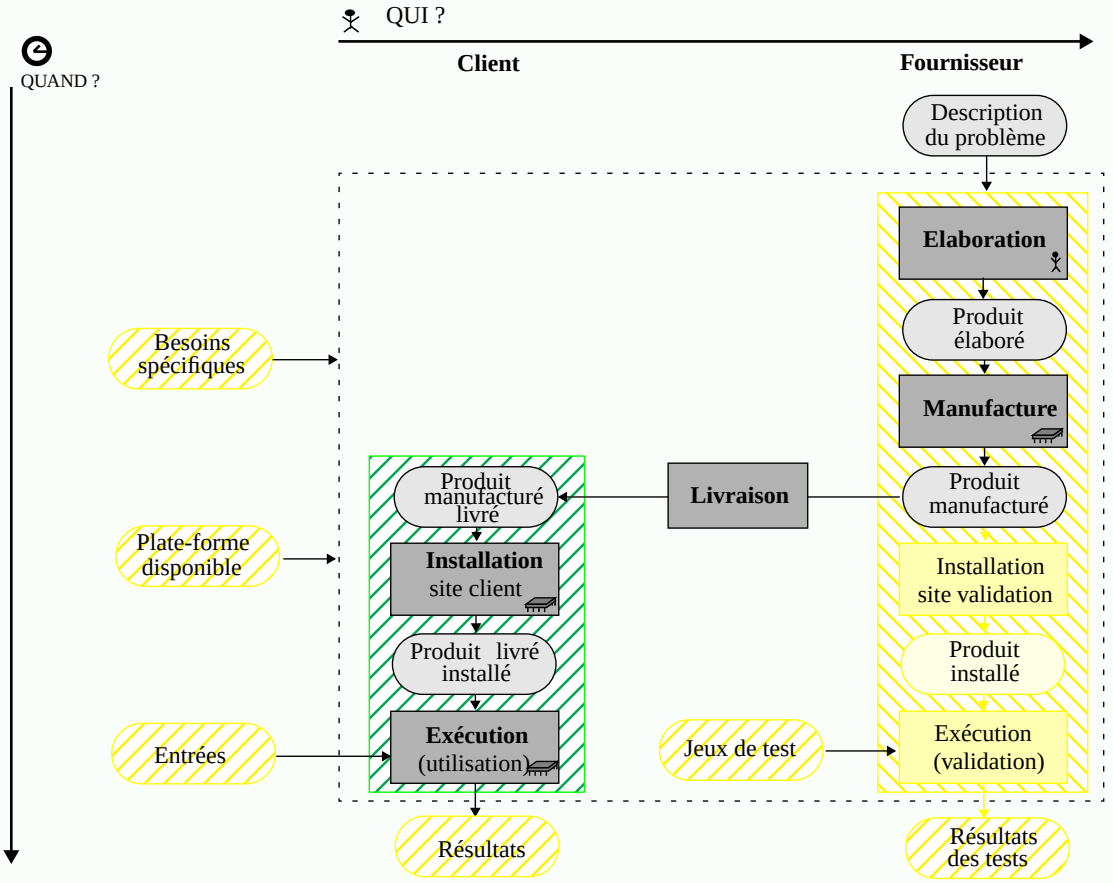
Les différentes possibilités de livraison sont commentées dans la figure 95.
Les deux cas extrêmes ((a) et (e)) ne seront pas considérés par la suite ; même chose pour le cas de la livraison d'un logiciel installé (b) qui n'est pas très différent du cas (c).
Nous nous focaliserons donc sur la livraison du produit source (d) et la livraison du produit manufacturé (c). Plus traditionnellement on parle de "***livraison de sources***" et de "***livraison de binaires***".

En pratique sélectionner un type de livraison est très important. Il s'agit souvent d'un compromis entre *protection, généricité, facilité d'installation et fiabilité*. D'autres aspects techniques interviennent aussi.

- *Protection*. La livraison de binaires est employée dans la grande majorité des cas car cette représentation du logiciel est difficilement exploitable pour plagier le logiciel.
- *Généricité et variations logiques*. La livraison de sources maximise les possibilités d'adaptation aux besoins de l'organisation cliente. Celle-ci peut prendre le contrôle du logiciel et l'adapter à ces propres besoins ; soit en modifiant le source, soit en faisant varier le processus de manufacture.
- *Généricité et variations techniques*. Maintenir et livrer des binaires spécifiques à chaque plate-forme matérielle et logicielle peut impliquer une gestion complexe et coûteuse si celles-ci sont très nombreuses[3].

1. On pourrait parler aussi de livraison dans le cas où un le logiciel passe d'un site de développement à un autre site de validation par exemple. Ce n'est pas le cas ici, bien que ce qui est dit par la suite s'applique également.
2. C'est le cas par exemple de l'édition de liens statiques et de l'édition de liens dynamique. (Voir à la fin de cette section).

*figure 94* Livraison de produits manufacturés




Dans ce schéma la dimension horizontale correspond au "***qui***?" (client ou fournisseur?). La dimension verticale au "***quand***?. Si la livraison est effectuée avant la fin du cycle (c'est le cas ici), le client doit terminer celui-ci. Dans le cas ci-dessus le client a la charge de l'installation.

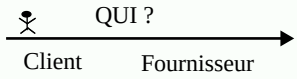
Avant de livrer le logiciel le fournisseur doit bien entendu le tester (voir la partie droite en bas de la figure). Même après la livraison il doit être capable d'exécuter le logiciel, par exemple pour essayer de reproduire les anomalies détectées sur le site client (pour localiser leur cause et les éliminer).

- *Facilité d'installation et fiabilité*. Si les sources sont livrés, la manufacture et l'installation incombent au client. Ces deux tâches peuvent être difficiles à mener et les erreurs commises peuvent compromettre la fiabilité du logiciel.
- *Aspects techniques*. La représentation du logiciel livré n'a de sens qu'avec un outil l'interprétant et le client doit disposer de celui-ci. Par exemple si l'on souhaite faire des livraisons de sources, l'utilisation d'outils de gestion de configurations coûteux ou peu répandus constitue un problème. *Inversement un outil commun comme CPP sera largement utilisé car il fourni une représentation standard pour les "familles" de programmes* (Chapitre IV)

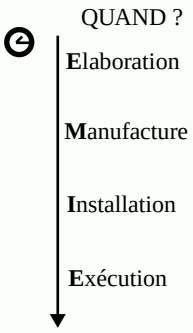
Une vision simplificatrice consiste à dire que la livraison de sources est utilisée dans le cas de logiciels de domaines publiques alors que la livraison de binaires correspond aux logiciels commerciaux. Cette dichotomie est trop catégorique et des solutions intermédiaires sont

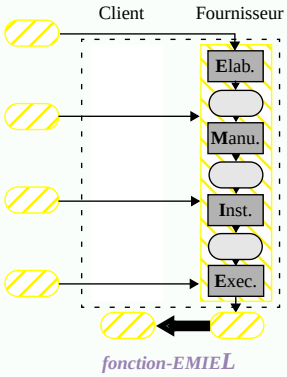
3. Typiquement les sites d'installations utilisent des révisions différentes du systèmes d'exploitation, des systèmes de fenêtrage, etc. Maintenir des binaires spécifiques pour chaque combinaison peut être rédhibitoire. D'autre part à chaque fois qu'un site change de configuration une nouvelle livraison est nécessaire (ce qui implique un coût supplémentaire).

*figure 95* Les différents types de livraison




Dans chaque figure la **L**ivraison est représentée par le symbole ⬅. En fait il s'agit de décomposer la fonction-**EMIE** en deux fonctions : la fonction exécutée par le fournisseur et la fonction exécutée par le client. Reste à déterminer à partir de quelle étape (EMIE) la décomposition se fait. En fait la livraison détermine le "***qui***?" en fonction du "***quand***?"

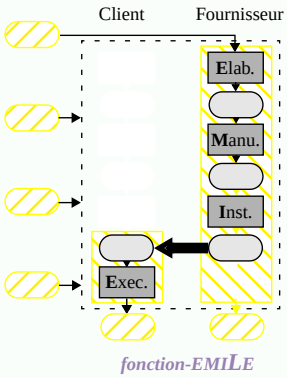



*fonction-EMIEL*

***(a)*** ***Livraison des résultats***

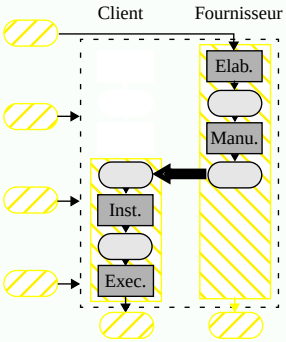
Cas extrême. Le logiciel est exécuté sous le contrôle de son constructeur. C'est le cas typique des centres de calculs où l'utilisateur soumet ses données mais n'interagit pas directement avec le logiciel. Dans ce cas il n'y a tout simplement pas de livraison du produit logiciel.




*fonction-EMILE*

***(b)*** ***Livraison du produit installé***

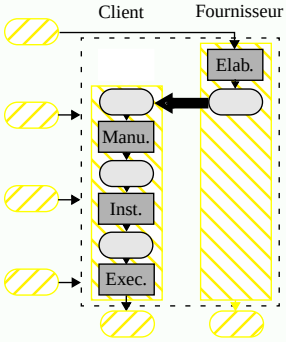
Cas où du personnel est détaché par le fournisseur sur le site client afin d'y intégrer le logiciel. Cela peut être nécessaire si la procédure d'installation est complexe.




*fonction-EMLIE*

***(c)*** ***Livraison du produit manufacturé***

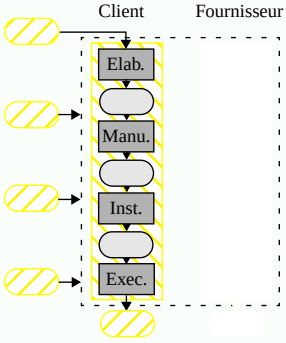
Le logiciel est livré au client sous forme de produit manufacturé. Ce type de livraison est représenté plus en détail dans la figure 94. Différentes étapes pouvant être considérées dans le processus de manufacture, ce cas sera affiné par la suite.




*fonction-ELMIE*

***(d)*** ***Livraison du produit source***

Le logiciel est livré au client avant d'être manufacturé.




*fonction-LEMIE*

***(e)*** ***Logiciel propriétaire***

Cas extrême. Le logiciel est produit, manufacturé, installé et exécuté par le client lui même. Ceci est rendu possible dans le cas où l'utilisateur dispose d'un environnement dédié à l'interprétation ou génération d'applications concernant un domaine spécifique. Par exemple un tableur permet à l'utilisateur de développer lui même un logiciel. Ici, il n'y a plus de fournisseur.

nécessaires. Il est alors naturel de décomposer l'étape de manufacture (voir par exemple la figure 82 (p.155)).

En théorie le résultat de chaque étape pourrait donner lieu à un type de livraison différent. En pratique seule la livraison de codes objets est retenue[1]. Dans ce cas la phase d'*édition de liens* incombe au client. Celui-ci peut alors faire varier le logiciel en choisissant les codes objets à lier.

L'***édition de liens dynamique*** est une solution alternative plus souple, plus sûre et assurant une meilleure protection. Cette technique est fonctionnellement équivalente à l'***édition de liens statique***. Cependant l'édition de liens dynamique est réalisée sur le site client alors que l'édition de liens statique se fait sur le site de développement. Il s'agit donc d'un exemple pour lequel la question "où?" est déterminante. La livraison est une frontière expliquant (en partie) l'existence de ces deux techniques.

### III.4.5 Détermination des choix (Qui ? Comment ?)

Depuis longtemps nous parlons de *contexte* et de *choix*, sans pour autant avoir précisé ***comment*** passer de l'un à l'autre, ***qui*** déterminait les *choix* et ***comment*** ils étaient exprimés (i.e. dans quel formalisme ils étaient représentés). Rappelons qu'un *contexte* est une notion informelle alors qu'un *choix* est une représentation informatique destinée à un objet générique. Il s'agit ici de faciliter le passage de l'un à l'autre (figure 96). En quelque sorte on considère maintenant les problèmes d'***interface des objets génériques***[2].

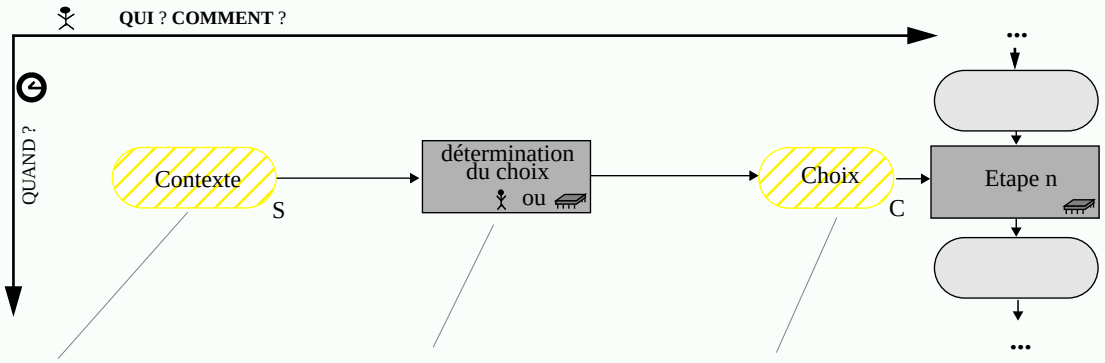
Le contexte d'un logiciel de grande taille peut être très complexe, surtout si celui-ci doit pouvoir s'exécuter sur différentes plates-formes. Il faut dans ce cas prendre en compte les variations du système d'exploitation, des bibliothèques graphiques, des outils de dérivation, etc. Le domaine des objets génériques peut alors être formé de centaines de dimensions![3] L'utilisation des notions présentées dans le cadre du modèle abstrait sont alors essentielles (Section "Abstraction, réduction et simplification de domaine" (p. 109), figure 48, figure 47). La *simplification de domaine* est basée sur l'utilisation de *valeur par défaut*[4]. Cette solution n'est pas restrictive dans la mesure où toutes les variantes d'un objet générique restent accessibles. Au contraire la *réduction du domaine* consiste à ne présenter qu'un sous-ensemble des possibilités réellement offertes par un objet générique[5].

---

1. Par exemple la livraison de sources étendus (par un préprocesseur) n'a pas de sens.
2. Une fonction est un objet mathématique abstrait, mais lorsque l'on passe aux problèmes concrets il est non seulement nécessaire de considérer sa représentation (un programme) mais aussi son interface. Ce raisonnement, valable pour n'importe quelle fonction, est appliqué ici aux objets génériques.
3. Dans le cas de gcc, les sources comportent plus de 600 choix sous la forme de macro définitions.
4. •*Exemple 1*• Le cas des ressources X fournit un bon exemple. Il est possible de spécifier explicitement la valeur d'un paramètre lors de l'exécution d'un client X (par exemple via l'option -bg pour définir la couleur du fond). Dans le cas contraire différentes règles de recherches sont utilisées pour chercher une valeur par défaut. •*Exemple 2*• La construction *#ifdef X #define X #endif* est extensivement utilisée dans le préprocesseur CPP car elle permet de définir une valeur par défaut pour la macro X. (cf Chapitre IV).
5. •*Exemple 3*• Certaines applications utilisent des fichiers de configurations complexes tout en offrant à l'utilisateur une interface graphique ne permettant de modifier que certains des paramètres. •*Exemple 4*• Les logiciels de domaine publique livrés sous forme de code sources sont généralement accompagnés d'une procédure d'installation simplifiée utilisable pour générer certaines configurations seulement.

*figure 96* Détermination d'un choix

Dans les figures précédentes le temps était la dimension prédominante. Maintenant c'est l'horizontale que nous cherchons à compléter pour montrer comment les choix sont déterminés à partir du contexte du logiciel[1]. Ce problème est susceptible d'intervenir à tous les niveaux du cycle. Des techniques différentes seront utilisées mais le principe général est toujours le même.




**(a) Contexte**

• ***Aspects logiques*** :
Les besoins spécifiques des clients ou du fournisseur ne sont pas explicitement représentés. Ils ne peuvent donc pas être traités automatiquement.

• ***Aspects techniques*** :
Les plates-formes logicielles et matérielles ont des représentations concrètes.

**(b) Détermination du choix**

• ***Aspects logiques*** :
La détermination d'un choix à partir d'un contexte est normalement effectuée par un agent humain. Il s'agit d'une analyse des besoins suivie d'une codification de ceux-ci en terme de choix.

• ***Aspects techniques*** :
Dans certains cas il est envisageable de déterminer *automatiquement* les caractéristiques des plates-formes utilisées. En pratique cet aspect est important pour simplifier autant que possible l'installation d'un logiciel. Dans le cas de la livraison de code source, il s'agit par exemple de détecter quels sont les compilateurs disponibles sur un site donné, quelle est la version du système d'exploitation, le nombre d'octets occupés par le type "entier", les fonctions disponibles dans une bibliothèque donnée, etc. Ces informations doivent ensuite être codées en termes de choix (par exemple en terme de macro-définitions passées au préprocesseur). La ***reconnaissance automatique de la plate-forme*** est très largement utilisée en pratique même si elle est basée sur des heuristiques relevant souvent de l'empirisme.

**(c) Choix**

Le choix doit être exprimé dans un formalisme adapté à l'objet générique. Concrètement ce peut être sous la forme d'un langage ad-hoc (e.g. les fichiers des ressources de X) ou encore via l'utilisation d'une interface graphique (e.g. une palette interactive pour choisir les couleurs). Ce peut être aussi sous la forme d'expressions complexes.

[1] En toute rigueur puisqu'il s'agit de problèmes de variations, ce devrait être la troisième dimension si l'on reprend les conventions introduites dans la figure 81 (p.154)

Il est clair que les solutions retenues dépendent des personnes chargées de déterminer les choix (la question *qui?*). Avant la livraison il s'agit du fournisseur, après ce sont les clients. Un langage complexe mais complet peut convenir au fournisseur. Au contraire le langage utilisé par le client doit être simple, même au risque de limiter les possibilités de variations offertes. Une menu est une bonne solution, lorsqu'elle est possible [Gull93] [Zell94].

Nous allons voir dans la Section III.5, ("Variation : approches") que ces aspects sont souvent négligés. De nombreux systèmes les ignorent et la pratique est alors dominée par l'utilisation de techniques ad-hoc.

## III.4.6 Variation et architecture

Une logiciel de grande taille, de part sa nature, ne peut bien évidemment pas être considéré comme un objet atomique. Puisque en terme du modèle abstrait on est en présence d'*objets structurés*, les notions introduites dans la Section II.5, ("Structuration du codomaine d'un objet générique") devraient pouvoir s'appliquer.

Considérons tout d'abord l'intersection entre les problèmes de variations et la granularité. Dans le modèle abstrait deux possibilités ont été présentées (Section II.5.1) : (1) La structure des objets n'est pas utilisée pour la variation ; cette solution a été appelée *variation globale*. (2) La décomposition des objets est prise en compte et chaque composant est susceptible de varier ; cette solution a été appelée *variation détaillée*.

#### *Configurations...*

Avec la première solution, tout se passe comme s'il s'agissait d'objets atomiques. Dans le cadre du génie logiciel on utilise traditionnellement le terme de "*gestion de versions*". Au contraire, en ce qui concerne le deuxième cas, il est nécessaire d'assurer la cohérence entre les variantes choisies pour chaque composant d'un objet structuré (Section II.5.1). Dans le cadre du logiciel on parle traditionnellement de "*gestion de configurations*". Si l'on se réfère au modèle abstrait, une ***configuration*** est une variante d'un objet structuré. Ces termes viennent donc de l'intersection entre les problèmes d'architecture et ceux de variation (et d'évolution).

#### *Variation globale du logiciel vs. variation détaillée du logiciel...*

Ce qui a été dit pour un niveau de granularité peut évidemment être généralisé. Dans le cas du logiciel, ce sont trois niveaux qui ont été introduits : la *granularité fine*, la *granularité moyenne* et la *granularité forte* (Section III.2.1). Les problèmes de variations peuvent être traités à chacun de ces niveaux ; mais aussi à tout niveau intermédiaire puisqu'il s'agit d'un continuum.
*Par convention*, nous appellerons ***variation globale du logiciel*** tout type de variation basé sur une *granularité forte* et ***variation détaillée du logiciel*** tout type de variation basé sur une *granularité inférieure*.

Cette distinction semble rompre la continuité entre ces deux granularités. C'est bien l'objectif recherché car cette rupture correspond à la réalité technologique : on assiste en pratique à une déconnexion importante entre les technologies utilisées pour la variation globale et pour la variation détaillée (voir plus loin la Section III.5, "Variation : approches").

#### *G-P-propriétés...*

Un objet générique est G-*P*-correct si son codomaine est P-correct, c'est à dire si toutes les variantes qu'il est possible de générer vérifient *P* (Section II.2.5). Dans le cas du logiciel les propriétés d'intérêt sont essentiellement la correction sémantique, syntaxique ou lexicale. Un objet générique pourra donc être qualifié de ***G-sémantiquement-correct***, ***G-syntaxiquement-correct*** ou ***G-lexicalement-correct***. Bien évidemment le but est de définir des objets génériques G-sémantiquement-corrects. Nous verrons que certains langages d'objets génériques assurent cette propriété ; autrement dit tous les objets génériques du langage la vérifie[1]. Au contraire dans la plupart des cas cette propriété est impossible à assurer. C'est le cas notamment des systèmes

basés sur une variation globale car ceux-ci n'ont aucun contrôle sur les modules qu'ils manipulent (ceux-ci sont considérés comme atomiques).

#### *Variantes vs. fragments...*

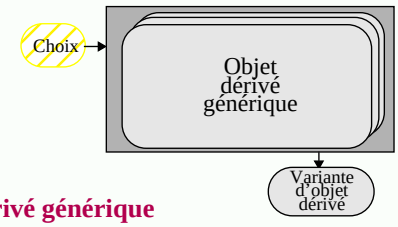
Le modèle abstrait a permis de mettre à jour un phénomène intéressant : plus l'on considère un niveau de granularité fin pour les variations, plus il y a de risques de "découper" les objets sur des frontières non "sémantiques" et de violer des contraintes d'intégrité. On parle alors de *différences* plutôt que de *variantes* (Section II.5.2).
Ce phénomène se produit aussi dans le cas du logiciel. Cela s'explique par le fait qu'il existe bien plus de contraintes de granularité fine que de granularité forte. Par exemple, au niveau de granularité forte il s'agit de vérifier que chaque interface est accompagnée d'une réalisation. Au niveau de granularité fine, il faut vérifier les propriétés vues ci-dessus.
A titre d'illustration, considérons le cas des macro-processeurs (présentés plus en détail dans le Chapitre IV). Ceux-ci permettent de décomposer le *texte source* du logiciel à un niveau de granularité fine, typiquement une ligne de texte ou même une suite de caractères. A l'extrême cette décomposition peut même se faire à un niveau plus fin que les concepts manipulés dans un langage de programmation. Par exemple il est possible de décomposer un lexème en plusieurs caractères et de faire varier les composantes résultantes. Il est alors naturel de parler de "différences" plutôt que de "variantes" (comparer ceci à l'exemple de la figure 52 (p.113)). Soulignons que cette situation n'est pas du tout incompatible avec le fait que le programme dans son intégralité soit sémantiquement-correct.

### III.4.7 Variation et manufacture

Bien souvent lorsque l'on parle de variation dans le contexte du logiciel, on pense aux variations du code source des programmes et donc à des ***objets-sources génériques***[1]. Pourtant c'est le binaire exécutable qu'il est essentiel de faire varier ; finalement, c'est lui qui détermine l'exécution du programme! En conséquence ce sont donc d'***objets-dérivés génériques*** dont on a besoin.
En fait la ***variation d'objets sources*** n'est qu'une méthode particulière permettant de supporter la ***variation d'objets dérivés***. Un objet dérivé n'est pas uniquement défini à partir d'objets sources, mais aussi d'opérateurs. Alors pourquoi ne pas faire varier ceux-ci?
Cette idée donne lieu aux concepts d'***outil de dérivation général*** et d'***outil de dérivation générique***. Ces concepts abstraits correspondent à une réalité bien concrète comme le montrent les exemples de la figure 97. Ils contribuent largement à la complexité des solutions que l'on rencontre en pratique (Cet aspect n'a pas été représenté dans la figure 100 (p.195) pour ne pas la rendre trop complexe. Il est clair qu'il intervient pourtant à de nombreux stades du cycle).

1. Rappelons que la propriété *sémantiquement-correct* fait référence à la sémantique statique du langage.
1. On aurait pu aussi choisir le terme objet générique source. Soulignons que cette thèse n'a pas l'ambition de définir un vocabulaire pour la postérité, ni d'introduire de manière gratuite de nouveaux mots. Il s'agit seulement de définir des concepts clés et même si la terminologie est parfois lourde, elle est préférable à l'utilisation de numéros...

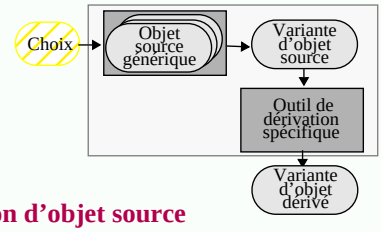
*figure 97* Objet-dérivé générique, variante d'objet dérivé




***(a)* Objet-dérivé générique**

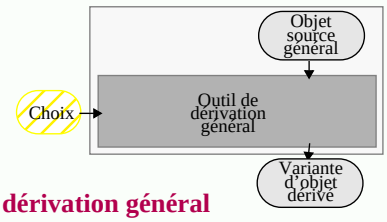
Le problème posé ici est de faire varier les objets dérivés. Par exemple le code exécutable dans le cas du logiciel. Ci-dessus un objet-dérivé générique est représenté de manière abstraite.

Trois solutions particulières sont étudiées. Bien que celles-ci soient présentées une à une, elles ne sont loin d'être indépendantes ; en pratique on trouve des combinaisons complexe de celles-ci.



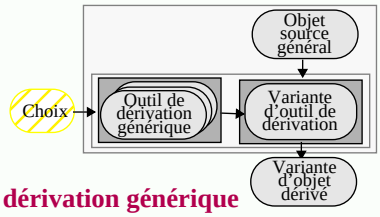

***(b)* Variation d'objet source**

Pour obtenir des objets dérivés ayant des caractéristiques différentes la solution la plus naturelle est sans doute d'utiliser un objet-source générique. Cette technique n'implique aucune modification de l'outil de dérivation. Il suffit de sélectionner une variante particulière puis d'appliquer "normalement" l'outil de dérivation.




***(c)* Outil de dérivation général**

Ajouter des paramètres à un outil de dérivation est également un moyen de faire varier les objets dérivés. Un outil de dérivation général prend en paramètre, non seulement l'objet source, mais également des ***paramètres de dérivation***. C'est le cas typique des compilateurs qui à l'aide d'options de compilation permettent de faire varier le binaire généré. Il s'agit soit de variations peu importantes (par exemple des niveaux d'optimisation du code produit), soit de variations plus radicales (par exemple certains compilateurs sont capables de générer du code machine pour différents micro-processeurs).




***(d)* Outil de dérivation générique**

Dans cette solution, c'est l'outil de dérivation lui-même qui varie. Cette situation pouvant sembler peu réaliste est pourtant courante. Par exemple ce sont toujours des références qui sont utilisées lors de l'invocation d'un outil. Ces références sont génériques : l'outil effectivement exécuté dépend de la plate-forme, des chemins d'accès aux binaires exécutables, etc. Par exemple plusieurs compilateurs peuvent être présents sur une même plate-forme. L'étape de compilation est alors précédée d'une phase de sélection du compilateur.

Remarquons que dans les cas (c) et (d) le même objet *source* est utilisé pour chaque contexte. Il s'agit donc d'un *objet-source général*. Cette solution, si elle est réalisable, est plus économique que la (a) : les problèmes de variations sont pris en compte au niveau de la dérivation et sont donc traités automatiquement.

Le concept de ***langage de programmation portable*** est directement issue de cette remarque : plutôt que de développer et de maintenir un code source générique pouvant s'adapter à chaque plate-forme (par exemple en utilisant un macro-processeur), il est de loin préférable de développer un code source général en utilisant un langage portable. Un compilateur multi-cibles (ou n compilateurs) permettent d'encapsuler les problèmes de variations.

## III.4.8 Synthèse

L'objectif de cette section était de spécialiser les concepts du modèle abstrait au cas de la variation de logiciels et d'introduire les concepts manquants.

Tout d'abord le contexte du logiciel a été décrit. Il ressort de cette étude que le logiciel est soumis à un très grand nombre de variations. La distinction a été faite entre les *variations logiques* et les *variations techniques*. La gamme des fréquences de variation est particulièrement importante : les périodes entre deux modifications peuvent aller de la minute à plusieurs années. Dans de telles conditions il n'est pas étonnant que différentes technologies soient utilisées.

Du point de vue conceptuel, un logiciel est paramétré autant par la description de son contexte (les choix) que par ses entrées. On arrive alors à la conclusion que le passage de la production du logiciel à son exécution peut être vue comme une série de *spécialisations*. Les différentes fréquences de variation des choix déterminent leur *moment de liaison*. Ce concept utilisé dans le cas de la spécialisation de programme doit être étendu dans le cas de la programmation globale. Non seulement il s'agit de déterminer *quand* les choix sont faits, mais en plus il faut savoir *où* et par *qui*. Les *problèmes de livraison* sont à la base de ce besoin. Ils introduisent des étapes de spécialisation supplémentaires.

La distinction a alors été faite entre (1) les *logiciels généraux* qui traitent les variations du contexte lors de l'exécution, (2) les *logiciels génériques* qui traitent ceux-ci dans une phase de génération automatique et (3) les *logiciels spécifiques* qui prennent en compte un contexte spécifique dés lors de la production. Ces différentes solutions ont été comparées.

L'importance du langage utilisé pour la *représentation des choix* a été soulignée. Différentes solutions doivent être proposées en fonction de la personne qui les spécifie. Dans le cas des variations techniques, certains choix peuvent être déterminés automatiquement. Vue la grande complexité des domaines des logiciels génériques, les techniques de *simplification de domaine* sont essentielles.

Tous les concepts présentés ci-dessus sont relatifs à la structuration du domaine des objets génériques. La structuration de leur codomaine donne lieu à la notion de *configuration*. On distingue la *variation globale* du logiciel de sa *variation détaillée* qui se fait à un niveau intra-modulaire. Les objets génériques se différencient par les G-*P*-propriétés qu'ils vérifient.

L'intersection avec les problèmes de manufacture est également très importante. Dans le cas de programmes, finalement le but est d'obtenir des *objets-dérivés génériques*. Traiter les variations au niveau des objets sources n'est qu'une solution particulière ; ces variations peuvent également être traitées au niveau du processus de manufacture ou des outils de dérivation.

Dans cette section le discours s'est limité à des aspects conceptuels. Au contraire dans la section suivante les technologies de variation de logiciels sont présentées.

# III.5 Variation : approches

Cette section a pour but d'étudier les différentes approches rencontrées en pratique pour résoudre les problèmes de variations. Comme dans le cas de l'architecture et de la manufacture, nous considérerons les 4 approches suivantes :

- III.5.1 "*Systèmes d'exploitation*" : système de fichiers et langages de commandes ;
- III.5.2 "*Langage de programmation*" : préprocesseurs, méta-programmes, généricité ;
- III.5.3 "*Langages et outils spécifiques*" : gestionnaires de versions, langage de configuration ;
- III.5.4 "*Bases logicielles*" : modèles de produits versionnés, configurateurs.

## III.5.1 Variation globale et systèmes d'exploitation

Un fois de plus, la technologie des systèmes d'exploitation fournit un solution rudimentaire.

- *Utilisation du système de fichier*. En simplifiant, il s'agit de stocker les variantes du logiciel dans des répertoires différents. L'identification se fait via des conventions de nommage. N'a-t-on pas l'habitude de changer le nom d'un fichier pour définir une nouvelle variante, d'utiliser des répertoires différents pour stocker plusieurs versions d'un logiciel ? Il s'agit là de techniques élémentaires pour créer des *objets génériques en extension* (Section II.3.5).
- *Utilisation de langages de commandes.* L'aspect dynamique des objets génériques peut être traité grâce à des fichiers de commandes. Des commandes peuvent être définies pour :
  - reconnaître automatiquement la plate-forme utilisée (figure 96.b (page 183)),
  - réaliser des fonctions de simplification de domaine (Section II.4.7), (Section III.4.5)
  - sélectionner une variante particulière (via une copie de fichier, la création d'un lien, etc.)[1]. L'utilisation de références génériques pour désigner les outils de dérivation permet les variations d'objets dérivés (Section III.4.7)[2].

Ces solutions, aussi archaïques qu'utilisées, se caractérisent tout de même par le fait d'être intuitives et faciles à mettre en oeuvre, en tout cas tant qu'il s'agit de résoudre des problèmes simples[3]. *Cette approche est néanmoins limitée aux variations globales* (le grain manipulé, le fichier, est généralement associé à la notion de module). De plus, elle ne permet pas d'assurer G-*P*-propriété car elle traite le logiciel comme n'importe quelle autre information.

1. L'installation de nombreux logiciels de domaine public est basée sur l'utilisation de fichiers de commandes tel "configure" du projet gnu. Cette commande remplit les trois fonctionnalités simultanément : (1) elle essaie de définir quelles sont les caractéristiques de la plate-forme utilisée ; (2) elle réduit le domaine (un nombre réduit de configurations "standard" peuvent être engendrées automatiquement à partir de cette commande) ; (3) finalement la variante sélectionnée est construite en copiant des fichiers et en créant des liens symboliques. De telles commandes sont construites de manière empirique et peuvent être complexes. Des commandes de génération automatique sont parfois utilisées (par exemple "autoconfig" dans le projet gnu).
2. Par exemple dans le cas du système Unix, la variable d'environnement $PATH précise le chemin de recherche à utiliser pour les binaires exécutables. A partir d'une référence générique à un outil de dérivation (par exemple le nom de la commande "cc"), et de l'état du système de fichiers, l'interpréteur de commande détermine l'outil de dérivation à exécuter (ici un compilateur c)
3. A une autre échelle, de grandes entreprises productrices de logiciels ont développé en interne des systèmes de gestion de configurations ad-hoc. Même si au départ le développement de tels environnements pouvait sembler peu coûteux en utilisant la technologie des systèmes d'exploitation, il est apparu que leur utilisation et leur maintenance posaient de nombreux problèmes [Dart]. Ces entreprises préfèrent aujourd'hui se tourner vers des solutions commerciales plus "standard" utilisant des bases logicielles (voir plus loin la Section III.5.4). Ce changement de situation correspond au passage de l'étape "artisanale" à l'étape "commerciale" dans le cycle d'évolution présenté dans le Chapitre I (figure 7 (p.20)).

## III.5.2 Variation et langages de programmation

Les approches visant à intégrer les problèmes de variation à la technologie des langages de programmation s'appliquent autant à la variation globale qu'à la variation détaillée. Elles ont été conçues spécifiquement pour la gestion de programmes.

### III.5.2.1 *Variation globale et langages de programmation*

Nous avons vu que la notion de module avait été intégrée dans les langages de programmation pour traiter les problèmes d'architecture. Généralement les techniques correspondantes ont également été prévues pour faciliter la variation globale. En effet la plupart des langages modulaires décomposent un module en deux parties physiquement séparées : l'interface et la réalisation. Cette séparation permet d'associer plusieurs réalisations à une même interface et donc de définir plusieurs variantes pour le module. Plus précisément l'association interface - réalisation n'est pas figée, comme ce serait le cas si ces deux parties étaient physiquement réunies. Cette association n'est déterminée que lors de la compilation. Les ***chemins de recherche*** permettent la sélection de la variante souhaitée parmi une structure donnée (par exemple les "bibliothèques" dans le cas d'environnements Ada). Cette structure joue le rôle d'objet générique et le chemin de recherche celui de choix.

En réalité cette technique peut également être appliquée avec les langages classiques ayant des interactions directes avec le système de fichiers. Des options de compilation permettent de spécifier ces chemins de recherche et ainsi de contrôler le mécanisme d'inclusion textuelle. Ces différents aspects sont discutés plus formellement dans le cadre du préprocesseur CPP (Section III.2.4).

Notons que de telles solutions n'impliquent pas de modification du langage de programmation, mais plutôt du compilateur. Remarquons aussi que les mêmes techniques sont applicables lors de la phase d'édition de liens : le fait de décomposer un programme en plusieurs unités de compilation ne se référençant pas explicitement permet de supporter les variations globales (et les évolutions).

### III.5.2.2 *Variation détaillée et langages de programmation*

Des techniques permettant la variation détaillée de programmes ont également été définies. Les deux mécanismes élémentaires présentés dans le modèle abstrait, à savoir la substitution et la sélection, sont à la base de toutes ces techniques. Elles sont pourtant loin d'être équivalentes et se différencient par les G-*P*-propriétés assurées. Les solutions les plus puissantes assurent que les objets génériques produits sont G-sémantiquement-corrects ; les solutions les plus rudimentaires n'assurent aucune propriété. Selon le niveau assuré on parlera de ***variations textuelles***, de ***variations lexicales***, de ***variations syntaxiques*** ou finalement de ***variations sémantiques***.

Les solutions les plus simples sont bien évidemment celles basées sur la représentation textuelle des programmes. Dans ce cas il s'agit de gérer des objets génériques dont le codomaine est une liste (de caractères ou de lignes). Les concepts étudiés dans le modèle abstrait sont directement applicables. Certains outils sont basés sur le mécanisme de substitution. Dans ce cas on parle de ***macro*** plutôt que de variable et de ***macro-substitution*** plutôt que de substitution. D'autres techniques sont au contraire basées sur le mécanisme de sélection. Il s'agit essentiellement de

l'*assemblage conditionnel* ou de la *compilation conditionnelle*. Comme leur nom l'indique elles ont été respectivement intégrées à la technologie des assembleurs et des compilateurs. Souvent ces techniques apparaissent simultanément dans un même outil, mais ce n'est pas toujours le cas. En tout cas il n'existe pas de terminologie standard dans ce domaine. Le terme *macro-processeur* faisant surtout référence au mécanisme de substitution, nous lui préférerons le terme plus général de *préprocesseur*[1]. Ces techniques ont été développées de manière plus ou moins empiriques dans les années 60-70 [Brow74]. Leur utilisation extensive peut donner lieu à des (meta)programmes très difficiles à comprendre. CPP, le préprocesseur du langage C et C++, est sans doute le représentant le plus connu de cette classe d'outils [Stal92]. Il sera étudié en détails dans le Chapitre IV.

Plus récemment des langages à vocation industrielle ont introduit le concept de *généricité*. Il s'agit par exemple des "templates" de C++, des paquetages génériques d'Ada ou des classes génériques d'Effeil. Ces systèmes sont essentiellement basés sur le mécanisme de substitution. Par exemple en Ada un paquetage peut être paramétré par une constante, un type ou une procédure. L'énorme avantage de ces langages est qu'ils assurent que les objets génériques construits sont G-sémantiquement-corrects. Autrement dit ils permettent des *variations sémantiques* et non pas des *variations textuelles*. Par contre, la mise en oeuvre de tels mécanismes est beaucoup plus complexe et coûteuse que dans le cas des préprocesseurs[2]. Ils sont aussi plus restrictifs (par exemple ils ne permettent pas l'utilisation du mécanisme de sélection ; il n'est généralement pas possible de représenter des fonctions de simplification de domaine[3] et les fonctions d'héritage sont souvent limitées). Des propositions d'extensions sont parfois faites ; par exemple Meta-Ada pour pallier aux limitations des paquetages génériques d'Ada [WinkStof88].

### III.5.3 Variation et outils spécifiques

L'insuffisance des mécanismes offerts par les systèmes d'exploitation a donné lieu à la conception et réalisation de systèmes spécifiques pour traiter les problèmes de variation et d'évolution.

Les *gestionnaires de versions* sont les outils les plus représentatifs. Par exemple SCCS [Roch74], développé au milieu des années 70, permet de gérer à un moindre coût des objets génériques appelées "archives". Cet outil est plus adapté aux problèmes de programmation coopérative et d'évolution qu'aux variations[4] (ses principaux mérites sont d'avoir proposé le modèle "check-in / check-out" pour gérer les accès parallèles et d'utiliser le mécanisme de delta pour stocker efficacement les versions historiques).
Les insuffisances de tels systèmes ont longuement été décrites dans la littérature [EstuFavr89]. Notons simplement qu'il s'agit d'outils de gestion de versions et non pas de gestion de configurations. Bien qu'ils aient initialement étés développés pour gérer l'évolution de

1. En réalité le terme préprocesseur indique simplement qu'une transformation préliminaire est effectuée sur le source d'un programme avant sa manufacture.
2. Notons entre autre actuellement, que peu de compilateurs C++ gèrent les "templates" effectivement.
3. Ada permet néanmoins de définir des valeurs par défaut pour les paramètres génériques.
4. Ses principaux mérites sont d'avoir proposé le modèle "check-in / check-out" pour gérer les accès parallèles et d'utiliser le mécanisme de delta

programmes, ils ne permettent d'assurer aucune G-*P*-propriété car ils manipulent les fichiers indépendamment de leur contenu.

D'autres propositions spécifiques ont été faites. Initialement les l*angages d'interconnexion de modules* se limitaient à décrire l'architecture des logiciels [DereKron76] (Section III.2.4) mais par la suite ceux-ci ont été étendus avec les notions de versions et de configurations. Dans ce dernier cas on parle parfois de ***langages de configuration*** et ceux-ci peuvent également intégrer l'aspect manufacture [Sing92] [TrygGull95].

Les deux approches présentées ci-dessus sont en fait complémentaires. Les systèmes de gestions de versions sont pauvres en ce qui concerne la modélisation et la description du produit logiciel mais fournissent les mécanismes de base pour leur gestion "physique". Les langages d'interconnexion de modules ont les caractéristiques inverses. Chacune de ses approches, prise indépendamment, est insuffisante ; mais elles peuvent utilement être réunies dans une solution multi-technologique.

### III.5.4 Variations globales et bases logicielles

Les bases logicielles intègrent les concepts de versionnement aux *modèles de produit*, ce qui permet de combiner les aspects architecture, variation et évolution. On parlera alors de ***modèle de produit versionné***. Les possibilités offertes sont très variables et dépendent plus ou moins des caractéristiques spécifiques du logiciel.

Dans l'environnement Gandalf [HabeNotk86], par exemple, le *modèle de produit versionné* est défini à partir de la structuration du logiciel en interfaces et réalisations. Plusieurs alternatives de réalisation peuvent être associées à une interface, chaque alternative pouvant exister en plusieurs révisions.

Aujourd'hui au contraire, la tendance est de proposer des modèles de versionnement plus généraux, qui puissent être appliqués à des objets quelconques [Casa96]. Autrement dit on cherche à déconnecter autant que possible les techniques de variations et d'évolution des spécificités liées à l'architecture du logiciel.

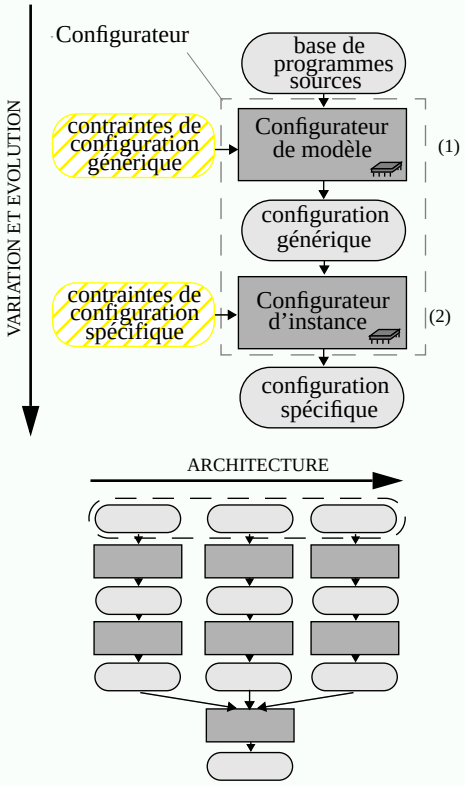
Les bases logicielles ont souvent pour ambition de supporter la gestion de configurations, thème très large (Section I.4.3). Dans le cadre de cette thèse nous nous focalisons sur un composant particulier : le ***configurateur***. En terme de modèle abstrait, un *configurateur* est un interpréteur d'objets génériques structurés ; à partir d'un choix (souvent appelé ***contrainte de sélection***), il génère une variante structurée (i.e. une *configuration*). La base de programmes (ou une partie) joue le rôle de l'objet générique.

Un configurateur dépend (1) du modèle de produit versionné utilisé (i.e. de la structure de l'objet générique), (2) de la structure à générer (i.e. de la structure du codomaine) et (3) des contraintes d'intégrité que la configuration doit respecter (i.e. de l'invariant du codomaine).
Supposons par exemple que le modèle de produit versionné permette d'associer plusieurs versions de réalisation à une même interface (comme dans le cas de Gandalf ou d'Adele par exemple). Configurer un module consiste alors à choisir une réalisation unique pour une interface donnée. Si l'on veut de plus assurer la "complétude" de la configuration, il sera également

nécessaire d'inclure les modules dont dépend (récursivement) le module donné. Dans ce cas configurer revient à parcourir un graphe-et-ou en sélectionnant un seul fils dans le cas d'un noeud-ou et tous les fils dans le cas d'un noeud-et[1] [Tich88].
Des configurateurs "câblés" ont été proposé pour résoudre de tel cas de figures [Estu85]. Proposer un ***configurateur général*** ou un ***configuration générique*** (par rapport au trois aspects présentés ci-dessus) reste du domaine de la recherche .

Il est intéressant de remarquer que dans plusieurs systèmes le processus de configuration se fait incrémentalement [EstuFavr89] [TrygGull95]. Ces systèmes sont donc implicitement basés sur la notion de *spécialisation d'objet générique* (Section II.6.4.3) (Section III.4.3). Souvent on retrouve également le concept de *décomposition parallèle* (Annexe A.3.4.2) (figure 98).

*figure 98* Exemple : le configurateur ADELE




***(a)*** **Spécialisation incrémentale**

A partir d'une contrainte de configuration (i.e. d'un choix), le configurateur (i.e. l'interpréteur d'objet générique) génère une configuration (i.e, une variante structurée). Tout comme aux autres niveaux, le besoin de décomposer cette transformation s'est fait ressentir.

- (1) La première étape est une *spécialisation* de l'objet générique (elle correspond aux choix des alternatives si l'on utilise la terminologie d'ADELE) . Une ***configuration générique*** est produite (on parle aussi parfois de "***modèle de système***").
- (2) C'est seulement dans le deuxième étape que l'on génère une variante. Ceci se fait à partir de l'objet générique spécialisé en (1).

***(b)*** **Décomposition parallèle**

Le processus (a) peut être décomposé en parallèle en s'appuyant sur l'*architecture* du logiciel. Concrètement cela signifie qu'il est possible de générer une configuration pour un sous-système donné et de la réutiliser par la suite avec d'autres configurations. Les avantages généraux de la décomposition parallèle décrits dans l'Annexe A.3.4.2 sont également applicables ici. (Une telle décomposition se retrouve à d'autres niveaux dans le cycle ; considérer par exemple le processus de compilation indépendante).

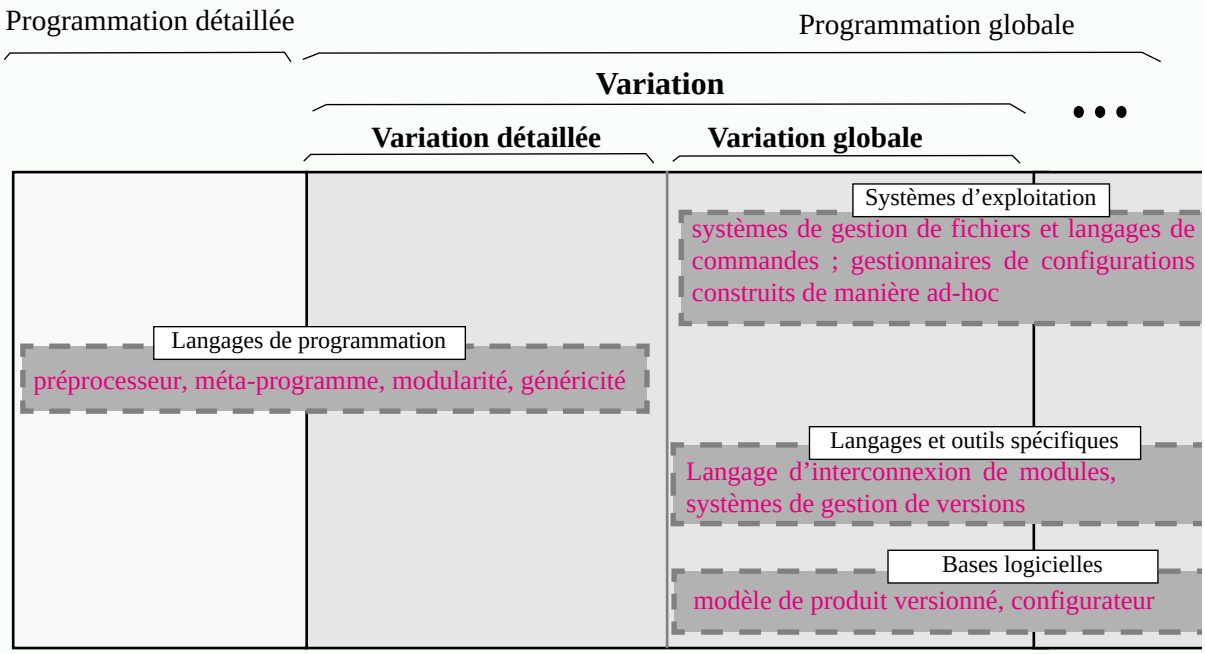
Comme nous l'avons souligné dans la Section III.2.5, les bases logicielles gèrent souvent le logiciel selon une granularité forte (typiquement le fichier est le grain le plus fin). Il en ressort que *les bases logicielles ne permettent pas de résoudre les problèmes de variations détaillées*.

## III.5.5 Synthèse

Les différentes approches concernant les problèmes de variation et d'évolution sont résumées dans la figure 99.

1. Les *noeuds-et* correspondent aux constructeurs (tous les composants doivent être pris). Les *noeuds-ou* correspondent au versionnement (une seule version doit être sélectionnée).

*figure 99* Techniques de représentations des variations



# III.6 Un exemple

Il ressort des sections précédentes

- (1) que les technologies utilisables pour la programmation globale sont très variées,
- (2) qu'aucune technologie ne couvre tout le spectre de la programmation globale.

Comme nous l'avions dit dans l'introduction, ce sont des configurations complexes que l'on trouve en pratique. Ci-dessous un exemple *simplifié* est donné à titre d'illustration.

## III.6.1 Un cycle typique

La figure 100 correspond à la concrétisation du cycle élaboration - manufacture - installation - exécution présenté dans la figure 85 (p.162) . Il s'agit d'un exemple plus détaillé ou un configurateur est utilisé avec des outils que l'on trouve par exemple dans l'environnement unix. Les premières étapes correspondent au gestionnaire de configurations ADELE. Celui-ci distingue explicitement deux étapes (Section III.5.4). Les étapes suivantes sont typiques du système unix avec un langage compilé précédé d'un préprocesseur.

Remarquons tout d'abord le nombre élevé d'étapes. Le lecteur est invité à comparer cette figure à celles présentées dans l'Annexe A.3 (figure 143 (p.297) et figure 144 (p.298)). Ces dernières mettent en jeu la vision que l'on a du même cycle, mais du point de vue de la *programmation détaillée*[1]. La différence considérable est essentiellement liée à l'introduction des problèmes de *programmation globale*.

En fait, la réalité est bien plus complexe encore. Nous n'avons fait ressortir que certains aspects par soucis de simplification.

#### *Aspects soulignés...*

Ce sont surtout des problèmes de *variation* qui sont responsables de cette succession importante d'étapes. Le logiciel doit être adapté à un contexte très complexe et cela doit se faire à différentes phases via une spécialisation incrémentale du logiciel[2].

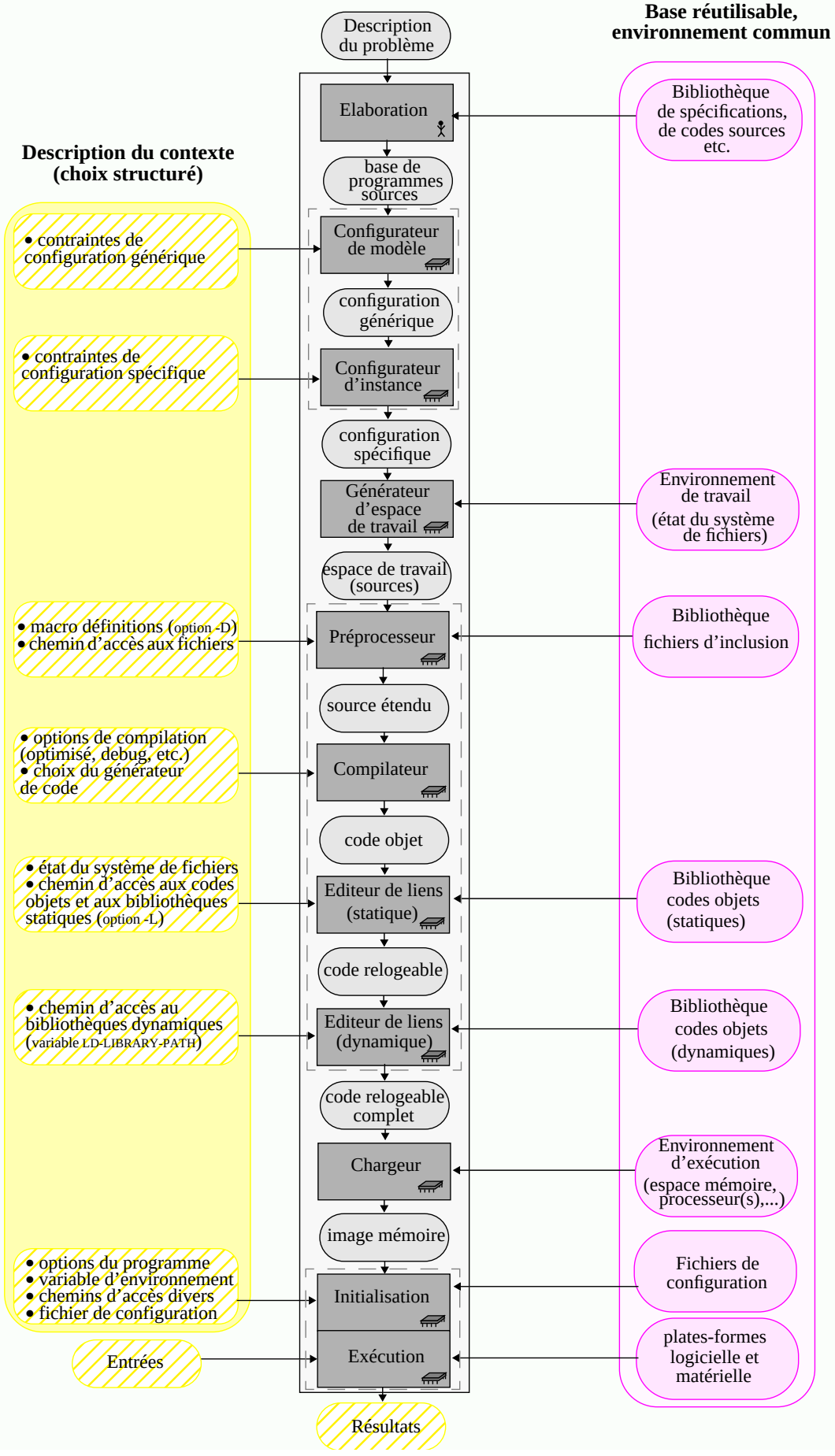
L'aspect *réutilisation* est mis en évidence dans la partie droite de la figure. La partie centrale ne contient que les transformations successives du logiciel lui-même.

Mêlés à ces problèmes, viennent des problèmes de *changement de représentations*. Ceux-ci sont dûs aux besoins des différents intervenants. En schématisant à l'extrême on peut distinguer :

- *L'équipe de gestion du processus*. Pour contrôler la production du logiciel, il est clair qu'une représentation abstraite est utile. Les bases de données logicielles sont les plus à même de fournir les services nécessaires grâce aux modèles de produits qu'elles proposent . Une vision

1. Si ce n'est qu'elles ne représentent pas la phase d'élaboration.
2. Dans la figure on prend en compte les variations techniques jusqu'à l'environnement d'exécution : sur quel(s) processeur(s) le logiciel doit-il s'exécuter, à quelle adresse mémoire doit-il être chargé, etc. Grâce à la technologie des systèmes d'exploitation ses informations peuvent être déterminées très tard dans le cycle.

*figure 100* Un exemple de cycle (vision simplifiée)



globale est indispensable ; toutes les variations et évolutions du logiciel doivent être considérées. C'est donc dans la partie supérieure de la figure que cette technologie est utilisée.

- *Le programmeur*. Au contraire le programmeur ne se focalise à un moment donné qu'à une spécialisation du logiciel. Typiquement il s'agit pour lui d'éditer et de compiler le texte sources des versions qui lui sont assignées. La technologie la plus adaptée est celle des systèmes d'exploitation : d'une part elle lui est familière, d'autre part c'est celle utilisée par tous les outils qu'il manipule. Concrètement le programmeur considère uniquement son "espace de travail". Il s'agit d'une portion du système de fichiers réservé à son usage et contenant le texte source des programmes qu'il doit modifier. Les représentations contrôlées le programmeur apparaissent dans la partie centrale du cycle.
- *Le micro-processeur*. Le but ultime du logiciel est d'être d'exécuté sur un micro-processeur. Il doit donc pouvoir être représenté en mémoire en langage machine. Cette représentation est utilisée dans la partie basse du schéma.

Ces *changements de représentations* sont responsables de l'introduction d'étapes supplémentaires. Par exemple une configuration spécifique est exprimée en termes d'objets de la base logicielle. Un générateur d'espace de travail permet de traduire cette représentation en une représentation adaptée aux programmeurs : le source des programmes sous la forme de fichiers dans une structure en répertoires. Finalement cette représentation est transformée via le compilateur et le chargeur pour obtenir une image mémoire utilisable par le micro-processeur. Si une seule représentation suffisait, ces étapes n'existeraient pas.

#### *Aspects non pris en compte...*

La réalité est bien plus complexe encore ; à ces aspects viennent se greffer les problèmes d'architecture et de manufacture.

L'intégration du concept d'*architecture* introduit une dimension horizontale, comme dans la figure 82 (p.155) et dans la figure 98.b (page 192). En fait la plupart des étapes peuvent être décomposées en parallèle. Les avantages généraux d'une telle décomposition sont présentés dans l'Annexe A.3.4.2. Ils s'appliquent ici. Aucun de ces aspects n'a été représenté sur le schéma.

Les problèmes de *manufacture* n'ont pas non plus été présentés. C'est un plan de manufacture particulier qui est décrit. Ni la représentation du plan de manufacture, ni les variations des outils de dérivation ne sont montrées. Pourtant ils sont essentiels en pratique. Sur un tel graphe devraient figurer les makefiles sélectionnant les outils de dérivation à utiliser, les options de compilation, etc. Dans la plupart des cas ses makefiles peuvent eux mêmes varier ou être décomposés.

## III.6.2 Différents moments de liaison

Beaucoup d'aspects techniques interviennent dans un tel schéma. Le présenter plus en détails serait intéressant mais trop long dans le cadre de cette thèse. Nous avons choisi d'illustrer un seul aspect : la spécialisation incrémentale. Plus particulièrement il s'agit de montrer qu'un même problème peut être abordé de multiples manières en faisant varier le moment de liaison.

Supposons qu'une procédure donnée doive exister en deux variantes, une pour la mise au point et l'autre pour l'exécution standard. Le choix entre ces deux variantes peut être fait à différents moments :

- *(1) Sélection avec un gestionnaire de configurations*. Si l'on dispose d'une base logicielle et d'un gestionnaire de configurations, le texte source de chaque variante peut être rangé dans deux objets différents. Chaque objet a un attribut "debug" dont la valeur est *vrai* ou *faux* selon le cas. Lors de la sélection d'une configuration une expression de la forme 'select debug=<valeur>" permet de sélectionner la variante nécessaire (figure 101.b).
- *(2) Sélection avec le préprocesseur*. Le préprocesseur du langage C est typiquement utilisé pour ce type d'application. Le mécanisme de compilation conditionnelle permet de regrouper dans un même fichier le texte source des deux variantes (figure 101.b). La sélection peut se faire lors de l'appel du préprocesseur en affectant une valeur à la macro debug.
- *(3) Sélection à l'édition de liens*. Une variante de chaque fonction peut être placée dans un fichier source différent, compilée sous forme de code objet et placée dans un répertoire différent. Lors de l'édition de liens, le reste du programme sera lié à l'un ou à l'autre de ces codes objets, par exemple en changeant les règles de recherche des codes objets[1].
- *(4) Sélection lors du chargement via l'édition de liens dynamique*. L'exécution du programme est précédée d'une phase de chargement en mémoire. L'édition de liens dynamique a lieu au cours de cette phase. Dans notre exemple il s'agit de ranger le code objet de chaque variante dans une librairie dynamique différente et lors du chargement, de fournir la règle de recherche correspondante[2].
- *(5) Sélection lors de l'exécution*. La sélection de la variante peut être repoussée jusqu'à l'exécution. Elle peut être faite dans une phase d'initialisation ou même dynamiquement au cours de l'exécution. Le programme de la (figure 101.c) présente une solution technique analogue à celle utilisant le mécanisme de compilation conditionnelle (figure 101.b). Ici "debug" n'est pas une macro mais une variable globale. La partie de code permettant de l'initialiser et/ou de modifier sa valeur en cours d'exécution n'est pas représentée. On peut imaginer que cette variable est initialisée à partir d'une variable d'environnement, d'un fichier ou qu'elle est modifiable interactivement via une interface graphique ou sous un metteur au point en mode pas à pas.

Etudions les caractéristiques générales de ces solutions :

- *Liaison statique vs. dynamique*[3]. Plus la sélection est faite tardivement plus la fréquence de variation peut être importante. Dans ce cas précis, il peut être pénalisant de devoir recompiler le logiciel à chaque variation du paramètre *debug*. On préfère alors la solution (5) où la liaison est dynamique. Par contre, lors de la livraison, éliminer le code spécifique à la mise au point peut être important. On voudrait donc dans ce cas que la liaison soit statique. Autrement dit, il est intéressant d'avoir des représentations qui permettent de choisir le moment de liaison. Des techniques ad-hoc basées sur l'utilisation de préprocesseurs offrent cette possibilité à un

1. Via l'option -L sous unix.
2. Sous unix ceci se fait via la variable d'environnement LD_LIBRARY_PATH.
3. Vu le nombre d'étapes, il s'agit bien sûr d'un abus de langage. Les termes statiques ou dynamiques sont relatifs à une étape spécifique. Cependant souvent *dynamique* fait référence à l'exécution du logiciel.

moindre coût (voir par exemple les “bidouilles” proposées par [Abac89]). Un évaluateur partiel constitue une autre alternative (Annexe A.3.9). Les techniques d’édition de liens statique ou dynamique offrent une dualité intéressante, mais pour les variations globales.

- *Variation globale vs. variation détaillée*. Les solutions (1), (3) et (4) correspondent à une variation globale, les solutions (2) et (5) à une variation détaillée. Dans le premier cas les parties communes ne sont pas factorisées ; les deux variantes sont gérées indépendamment l’une de l’autre et l’on assiste alors au problème de la maintenance multiple (Section I.4.4.5). Inversement les solutions (2) et (5) permettent au programmeur de manipuler simultanément les différentes variantes au prix d’efforts supplémentaires.

Dans cet exemple, des solutions particulières ont été retenues mais il en existe de nombreuses autres. Par exemple la solution (1) peut être substituée par une commande agissant sur l’état du système de fichiers (comme par exemple la commande “config” des logiciels de domaine publique) ; le même résultat peut aussi être obtenu grâce à l’utilisation de chemins de recherche avec un préprocesseur, ou avec un compilateur ; à la place de la solution (2) on aurait pu utiliser un paquetage générique Ada paramétré par la constante Debug, etc.

*figure 101* Deux variantes pour la procédure f

```
{   debug -> false ;
    source ->
        procedure f() ;
            /* fonction
            normale
            */
}
;
{   debug -> true ;
    source ->
        procedure f() ;
            /* fonction  pour
            la mise au point
            */
}
```

***(a)*** **Sélection avec un gestionnaire de configuration**
**Solution (1)**

Chaque variante est rangée dans un objet différent. La sélection est faite lors de la gestion de configurations à partir de l’attribut ‘debug’.

```
procedure f() ;
#ifdef debug
   /* corps de la fonction pour la mise au point */
#else
   /* corps de la fonction normale */
#endif
```

***(b)*** **Sélection à l’aide du préprocesseur**
**Solution (2)**

Les lignes précédées du # sont interprétées par le préprocesseur et selon la valeur de la macro ‘debug’ l’une ou l’autre des deux parties sera inclue.

```
var debug : boolean ;
...
procedure f() ;
if debug then
   /* corps de la fonction pour la mise au point */
else
   /* corps de la fonction normale */
endif
```

***(c)*** **Sélection lors de l’exécution**
**Solution (5)**

La variable debug est une variable booléenne. Ici la manière de l’initialiser ou de la modifier n’est pas représentée. L’instruction conditionnelle est interprétée lors de l’exécution du programme.

## III.6.3 Synthèse

Pour résoudre les problèmes de programmation globale, en pratique on trouve des configurations technologiques très hétérogènes. Cette hétérogénéité est, dans certains cas, due à des facteurs purement historiques. On retrouve cependant l’application de concepts généraux comme par exemple la spécialisation incrémentale.

Notons finalement que les techniques proposées ici proviennent autant des travaux de programmation globale que de programmation détaillée. Cet exemple est intéressant dans la mesure où il renforce l’idée qu’il existe un continuum entre ces deux disciplines.

# III.7 Conclusion

L'objectif de ce chapitre était de présenter la technologie de la programmation globale en s'appuyant sur le modèle abstrait. Ci-dessous les principaux apports sont résumés. L'espace technologique est tout d'abord présenté sous forme synthétique, puis en faisant ressortir la distance entre l'état de l'art et l'état de la pratique. A partir de l'étude des technologies existantes, quelques recommandations sont faites pour le développement futur d'environnements de programmation globale. Nous concluons par le besoin de développer la ré-ingénierie globale.

## III.7.1 Principaux apports de ce chapitre

Dans ce chapitre il s'agissait (1) d'affiner et de spécialiser les concepts du modèle abstrait, (2) d'ajouter les concepts manquants, (3) d'étudier les différentes techniques, (4) de les classer et (5) d'étudier leur interactions.

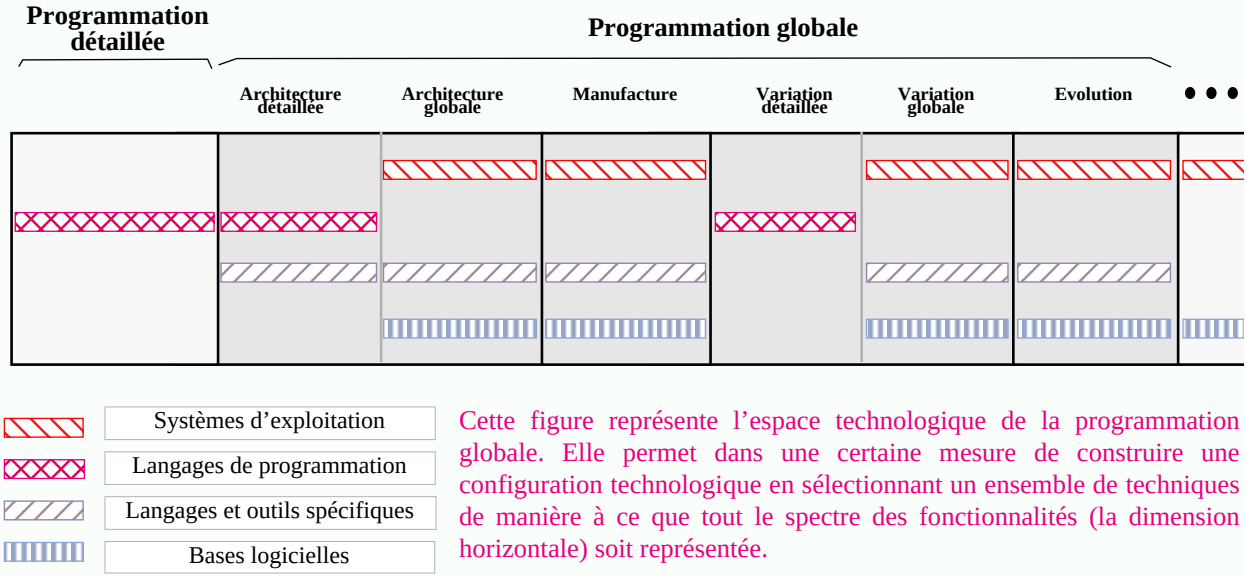
- *Concepts affinés*. Dans le modèle abstrait, les concepts de constructeurs et de références ont été introduits. Ici ils ont été appliqués au cas du logiciel et ont permis de distinguer trois niveaux de granularité. En réalité, il existe un continuum entre la programmation détaillée et la programmation globale.
- *Concepts ajoutés*. Le modèle abstrait ne faisait pas ressortir explicitement le besoin de spécialisation incrémentale pour les objets génériques. Ce concept a été introduit pour modéliser les usages pratiques des différentes technologies. Une fois de plus, réutiliser des concepts de programmation détaillée s'est avéré fort utile.
- *Présentation des techniques*. Un spectre très large a été présenté. Par exemple pour les problèmes de variation celui-ci va des paquetages génériques Ada aux configurateurs en passant par les éditeurs de liens dynamiques. Un tel éventail de techniques hétérogènes n'aurait sans doute pas pu être abordé sans se baser sur le modèle abstrait présenté dans le Chapitre II et sans la structuration de la programmation globale introduite dans le Chapitre I.
- *Classification des techniques*. La structuration de l'espace logique a été améliorée notamment en faisant la distinction entre l'architecture détaillée et l'architecture globale mais aussi la variation détaillée et la variation globale. Orthogonalement, nous avons vu que 4 lignes principales permettent de décrire les approches de programmation globale. L'apparition de ces différentes lignes a été présentée comme un phénomène général face à l'apparition de nouveaux problèmes.
- *Etude des interactions entre techniques*. Un exemple a permis de souligner le grand nombre de techniques hétérogènes interagissant en pratique. Malgré la complexité apparente d'une telle configuration, il a été montré qu'elle n'était pas le fruit du hasard, mais qu'au contraire des explications rationnelles pouvaient être données à partir des concepts introduits[1].

1. Cette vision contraste avec la vision la plus commune qui consiste à ne voir des outils comme les éditeurs de liens que d'un point de vue technique [HendDG91].

## III.7.2 Espace technologique de la programmation globale

En réalité, l'un des principaux apports de ce chapitre est d'avoir établi des "cartes" de l'espace technologique de la programmation globale : la figure 85 (p.162) pour l'architecture, la figure 85 (p.162) pour la manufacture et la figure 99 (p.193) pour la variation et l'évolution. Une vision globale est offerte dans figure 102 ci-dessous.

*figure 102* Structure de l'espace technologique de la programmation globale.




Cette figure représente l'espace technologique de la programmation globale. Elle permet dans une certaine mesure de construire une configuration technologique en sélectionnant un ensemble de techniques de manière à ce que tout le spectre des fonctionnalités (la dimension horizontale) soit représentée.

L'interprétation de cette figure donne lieu à quelques points remarquables :

- Les fonctionnalités offertes par les outils spécifiques (▨) chevauchent soit les bases logicielles (▥), soit les langages de programmation (⊠). Dans le premier cas, cela est dû au fait que les bases logicielles offrent des solutions intégrées. Dans le cas des langages de programmation, on retrouve la redondance des MILs par rapport aux langages de programmation modulaire.
- Les bases logicielles (▥) couvrent les mêmes fonctionnalités que les systèmes d'exploitation (▧). Cela s'explique par le fait que les deux technologies visent à gérer des objets de granularité forte (dans le premier cas il s'agit de fichiers, dans le deuxième d'objets logiciels).
- L'intersection entre la technologie des langages de programmation (⊠) et la programmation globale n'est pas négligeable. Elle se fait à un niveau de granularité fin ou moyen.
- *La technologie des langages de programmation (⊠) est indispensable pour gérer les problèmes de variation détaillée et donc pour éviter les problèmes de maintenance multiple*.

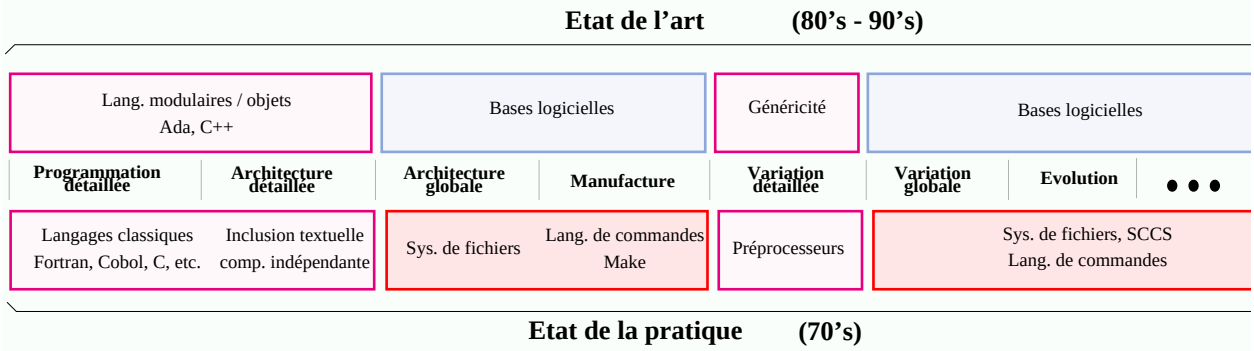
Cette dernière remarque vaut la peine d'être notée car cet aspect est souvent négligé [WinkStof88][Mahl94].

L'interprétation de cette carte globale doit néanmoins se faire avec beaucoup de précautions. Elle ne remplace pas les cartes détaillées car elle ne montre pas, par exemple, l'adéquation des services proposés. Il ne faudrait pas déduire que les MIL et les langages de programmation sont toujours redondants en ce qui concerne la description de l'architecture détaillée. Ce n'est le cas que pour des langages comme Fortran!

## III.7.3 Etat de l'art vs. Etat de la pratique

Plus généralement la carte globale ne distingue pas l'état de l'art de l'état de la pratique. La figure 103 permet de remédier à ce problème. Elle compare, les technologies appliquées en pratique, aux technologies qui pourraient être utilisées et qui font partie de l'état de l'art.

*figure 103* Technologies : état de l'art vs. état de la pratique




La vision proposée correspond bien évidemment à une simplification de la réalité, mais elle permet de faire ressortir les points suivants :

- *Etat de la pratique*. La technologie utilisée actuellement pour la programmation globale provient des années 70. Elle est essentiellement basée sur deux composantes : d'une part la technologie des langages de programmation pour la granularité fine et moyenne, d'autre part la technologie des systèmes d'exploitation pour la granularité forte[1].
- *Etat de l'art*. Une décomposition similaire s'applique à l'état de l'art : les bases logicielles substituent les systèmes d'exploitation et les langages de programmation sont plus modernes. Ces technologies, mises au point au cours des années 80 ou 90, pourraient changer la physionomie de la programmation globale.

Ce chapitre confirme donc la tendance générale annoncée dans le Chapitre I, plus particulièrement dans la Section I.4.5, (p.55) et Section I.6.1, (p.72) : la pratique de programmation globale est d'une certaine manière à un tournant de son histoire ; elle est en train de passer de la phase artisanale à la phase commerciale.

## III.7.4 Améliorer l'état de l'art

En se basant sur l'étude menée dans ce chapitre, quelques recommandations peuvent alors être faites pour améliorer la construction des futurs environnements de programmation globale.

La première recommandation est de *porter une attention particulière aux problèmes d'intégration car les solutions multi-technologiques semblent inévitables en pratique*. Considérons les deux dimensions de la figure :

- *Intégration horizontale*. Il est nécessaire d'assurer une meilleure intégration entre la technologie des langages de programmation et des bases logicielles. Ces technologies sont

1. Des outils spécifiques sont utilisés (par exemple Make et SCCS) mais en simplifiant ils peuvent être associés à la technologie des systèmes d'exploitation ; d'ailleurs ces outils sont depuis longtemps partie intégrante du système unix.

respectivement basées sur une granularité fine et une granularité forte. Pourtant une rupture technologique n'est pas adaptée au continuum qui existe entre les concepts. Concrètement en attendant qu'une technologie unique soit utilisable, il est nécessaire de *proposer des outils permettant l'extraction d'informations de programmation globale à partir du code source des programmes*. Cette recommandation est valable autant pour la technologie correspondant à l'état de l'art que pour celle correspondant à l'état de la pratique.

- *Intégration verticale*. Proposer de nouvelles technologies est une chose, mais celles-ci ne peuvent que rarement vivre en autarcie. Autrement dit, pour leur permettre un succès pratique, il est souvent nécessaire de les intégrer à la technologie existante (sur la figure d'assurer des connexions verticales). Dans le cas de la programmation détaillée, ce besoin a donné lieu aux approches multi-langages, au besoin de pouvoir appeler du C ou du Fortran à partir du langage Ada, au succès important de C++ par rapport à des langages plus avancés, etc. Dans le cas de la programmation globale, *les bases logicielles doivent intégrer leur technologie propre à celle des systèmes d'exploitation*.

La deuxième recommandation est de *garder à l'esprit que les logiciels, aujourd'hui développés en utilisant les environnements de programmation globale, devront être maintenus demain. Toute technologie n'est que transitoire*. Cette remarque mène à deux points :

- *Ne pas se focaliser uniquement sur la production*. Pendant une très longue période, l'unique préoccupation dans le domaine de la programmation détaillée était de produire des logiciels qui "marchent". Les techniques développées visaient à faciliter l'*écriture* des programmes. Par contre, il est aujourd'hui clair que le problème est aussi de les *lire*, de les *analyser*, de les *transformer*, etc. Hélas, cette constatation n'est pas encore claire dans le domaine de la programmation globale. Dans certains cas, les langages proposés restent ésotériques (à l'image des langages de commandes, concis mais peu lisibles).
  Par manque d'espace nous n'avons pas pu étudier les opérations associées à chaque représentation ; pourtant on s'aperçoit, dans la plupart des cas, que seules les opérations indispensables sont offertes. Par exemple, qu'ils soient âgés ou plus récents, les outils de manufacture se contentent souvent d'exécuter la fonction représentée. Il en est de même pour les objets génériques. Bien souvent aucun outil d'analyse, de visualisation, de mesure ou de mise au point n'est disponible. A terme, toutes ces fonctionnalités devraient être présentes dans un environnement de programmation globale.
- *Formaliser autant que possible*. Si l'on admet que les informations qui seront décrites avec un tel environnement devront être maintenues pendant des années, il est clair que l'on doit chercher à définir des abstractions et ne pas se limiter à produire des représentations. Si cela n'est pas fait on risque, dans quelques décennies, de se trouver dans une situation analogue à celle connue aujourd'hui dans le domaine de la programmation détaillée. Il est possible de définir des abstractions a posteriori (ce que l'on tente de faire dans cette thèse), mais il de loin préférable de les définir a priori!

La troisième recommandation est de *prendre garde aux problèmes de performances et de valider les approches sur des exemples en grandeur nature.* En effet, la plupart des technologies actuelles sont confrontées à de sérieux problèmes de performances pouvant les rendre inutilisables dans certaines situations.

Ces différents points sont complémentaires, ils ne s'opposent pas. Par contre, il est vrai qu'il est souvent difficile de les mener tous en parallèle dans un projet de programmation globale. Comme nous l'avons vu dans le Chapitre I, les ressources affectées à ce thème sont traditionnellement moindre par rapport à celles dirigées à la programmation détaillée. Peut être cette relation devrait-elle être revue dans le futur pour équilibrer l'importance et les moyens mis en oeuvre ?

### III.7.5 Améliorer l'état de la pratique

Le discours tenu ci-dessus est lié à la différence entre l'état de l'art et l'état de la pratique. Les recommandations faites visent à améliorer l'état de l'art (le haut de la figure 103) tout en évitant une déconnexion totale avec l'état actuel de la pratique (le bas).

Nous nous intéressons plutôt à la vision duale du problème : améliorer l'état de la pratique (le bas) tout en favorisant si possible le transfert d'informations vers le haut. Autrement dit, nous cherchons à faciliter la maintenance et à la ré-ingénierie globale des logiciels âgés.
Soulignons tout de suite que la modernisation n'est qu'un aspect particulier de ce problème et que dans l'état actuel des connaissances, il faut être bien modeste.

Le dernier chapitre de cette thèse fait une incursion dans le domaine de la ré-ingéniérie globale en observant plus particulièrement des outils venus d'un autre temps : les préprocesseurs.

# Chapitre IV
# Maintenance et Ré-ingénierie globale en présence de préprocesseurs

# INDEX

## Symbols



## A



## C



## D



## F



## G



## I



## L



## M



## O



## P

# S

# LISTE DES FIGURES

# CHAPITRE IV

# Maintenance et Ré-ingénierie globale en présence de préprocesseurs

## IV.1 Introduction

*"/* You are not expected to understand this! */"*[1]

Alors que les chapitres précédents ont abordé des thèmes relativement généraux, ce chapitre se concentre sur un thème plus spécifique : l'utilisation de préprocesseurs et les problèmes de maintenance associés. Plus particulièrement nous nous intéressons à CPP, le préprocesseur du langage C. La démarche décrite dans ce chapitre peut être vue comme une illustration du discours tenu dans le reste de la thèse. Cette démarche vise également à définir des services facilitant la maintenance et la ré-ingénierie des préprocesseurs. Comme nous allons le voir ces outils sont responsables de nombreux problèmes. Certains meta-programmes sont en effet incompréhensibles.

Le préprocesseur CPP est tout d'abord étudié d'un point de vue concret (Section IV.2). Le lecteur se rendra compte rapidement que penser en termes si concrets n'est pas aisé. Cela ne permet ni de réutiliser les connaissances acquises dans d'autres contextes, ni de développer des raisonnements réutilisables. Les sections qui suivent ont donc pour but de décrire cet outil à un niveau abstrait.

Le préprocesseur CPP est interprété en fonction du modèle abstrait dans la Section IV.3. L'idée sous-jacente est de pouvoir utiliser les concepts introduits dans le Chapitre II et ainsi de situer cet outil par rapport aux autres systèmes.

Le modèle abstrait n'est pas suffisamment précis pour pouvoir être utilisé dans un contexte de ré-ingénierie. Des abstractions plus adaptées sont alors proposées (Section IV.4). Il s'agit en fait, d'un langage nommé APP, équivalent à CPP, mais défini rigoureusement et utilisant des concepts de programmation détaillée. Ce processus d'abstraction est ensuite utilisé pour définir différentes techniques facilitant la compréhension de familles de programmes (Section IV.4). Finalement ce chapitre se termine par la présentation d'un prototype démontrant l'utilité de ces techniques.

1. Commentaire apparaissant dans un ancien noyau unix [SpenColly92].

# IV.2 CPP d'un point de vue concret

> *"Among the facilities, techniques, and ideas C++ inherited from C was the C preprocessor, CPP. I didn't like CPP at all, and I still don't like it. ... Occasionally, even the most extreme uses of CPP are useful, but its facilities are so unstructured and intrusive that they are a constant problem to programmers, maintainers, people porting code, and tool builders. ... CPP isn't even a very good macroprocessor. Consequently, I set out to make CPP redundant. That task turned out to be far harder than expected. CPP may be ugly, but it is hard to find better-structured and efficient alternatives for all of its varied uses." (B. Stroustrup).*

L'acronyme CPP est normalement utilisé pour désigner le ***préprocesseur du langage C*** ("the *C Pre*Processor"). Une interprétation légèrement différente sera retenue dans cette thèse : l'acronyme ***CPP*** désignera ce même préprocesseur mais en le voyant comme un ***préprocesseur concret*** ("a *Concrète Pre*Processor"). Ce choix est lié au fait que

- le préprocesseur CPP est utilisé avec d'autres langages que le langage C ;
- c'est un outil concret (par opposition au préprocesseur APP présenté par la suite).

Dans cette section sont successivement décrites l'importance de CPP (IV.2.1), ses mécanismes (IV.2.2), leurs usages (IV.2.3) et les problèmes associés (IV.2.4). C'est une vision concrète et pratique qui est proposée.

## IV.2.1 Importance

La première question que l'on peut se poser est "pourquoi prendre CPP comme cas d'étude ?".

- CPP est associé au langage C. Ces outils sont très utilisés, autant dans l'industrie que dans la recherche. Ce sont des piliers du système Unix. Ils sont disponibles sur de très nombreuses plates-formes [Kris95]. L'interface avec le système Unix se fait via l'utilisation du préprocesseur CPP.
- Avec le C, CPP peut être vu comme un outil du passé et du présent. Dans l'avenir, C++ risque de devenir un langage majeur dans l'industrie. CPP lui est également associé. Même si les constructions offertes par C++ permettent de limiter l'emploi de ce préprocesseur, le style de programmation C risque d'être présent dans l'industrie pendant des années encore (et avec lui l'usage des mécanismes de bas niveaux du préprocesseur).
- L'usage de CPP n'est pas limité à l'écriture de programmes C ou C++. Il est aussi utilisé avec des dialectes de Pascal ou d'autres langages de programmation, des makefiles ou des fichiers de textes. D'un point de vue pragmatique, c'est sa disponibilité et sa simplicité qui fait de CPP un "préprocesseur à tout faire", même s'il ne s'agit pas d'une solution optimale.
- Les mécanismes proposés par CPP sont relativement simples et représentatifs de ceux que l'on trouve généralement dans d'autres préprocesseurs. S'il est vrai que ce chapitre est principalement basé sur CPP, les caractéristiques de celui-ci sont décrites très précisément afin de pouvoir déterminer à chaque instant ce qui lui est spécifique et ce qui ne l'est pas.

- CPP et sans doute l'outil de variation détaillée le plus populaire et le plus utilisé. Bien qu'il s'agisse d'un langage de bas niveau, il est souvent cité dans les conférences sur la gestion de configurations ou la réutilisation (généralement comme référence de l'état de la pratique et pour citer ses inconvénients).
- Des logiciels provenant de nombreuses sources sont disponibles. C'est le cas par exemple des logiciels de domaines publiques écrits en langage C. Ils utilisent CPP de manière extensive pour des raisons de portabilité. Cette masse conséquente de logiciels peut être utile pour valider une approche de ré-ingénierie dans un contexte de recherche. La disponibilité de code "étranger" permet de tester la validité d'outils de compréhension de programmes.
- CPP est basé sur une représentation textuelle des programmes, une technologie simple mais efficace. En dépit de ses inconvénients évidents, une telle représentation reste encore très utilisée. Des outils comme emacs, sccs ou diff sont caractéristiques de l'état de la pratique et risquent d'être utilisés pendant encore longtemps.

Autrement dit CPP a été, reste et restera sans doute très utilisé[1].

### IV.2.2 Mécanismes

Dans cette section, les caractéristiques principales du préprocesseur CPP sont présentées informellement ; tout comme dans la plupart des ouvrages consacrés au langage C. A notre connaissance, il n'existe pas de vocabulaire standard, ni de document décrivant formellement cet outil. La Section IV.4 a pour but de pallier à ce manque. Ici il s'agit de donner une idée intuitive de ce langage.

Comme beaucoup d'autres préprocesseurs, CPP a été conçu de manière empirique et sans prendre en compte les principes aujourd'hui jugés fondamentaux pour définir un langage. Il n'est fait aucune distinction entre lexique, syntaxe et sémantique. Les interpréteurs CPP sont écrits de manière ad-hoc. Ils considèrent le fichier comme un flot de caractères et effectuent l'analyse et les traitements simultanément. Cette confusion a donné lieu à de nombreuses incohérences. Les descriptions de CPP sont considérablement alourdies par la prise en compte des cas particuliers et des aspects purement syntaxiques. Par la suite nous ferons abstraction de certains détails, quitte à être imprécis, voir inexacts[2].

Un fichier CPP est une liste de lignes. Celles commençant par un caractère dièse (#) sont des ***directives CPP***. Elles permettent de contrôler le déroulement du préprocesseur. Il existe 3 groupes principaux de directives. Chaque groupe correspond à un mécanisme différent : (1) l'inclusion textuelle, (2) la macro-substitution, (3) la compilation conditionnelle.

Finalement le passage de paramètres à CPP se fait via la *ligne de commande* (IV.2.2.4).

1. En réalité, CPP n'est pas un préprocesseur, mais une famille de préprocesseur. Il existe de très nombreuses variantes de cet outil et celui-ci évolue [Stal92] [Fowl&al95a]. Nous nous sommes plus particulièrement intéressé à l'implémentation proposée dans le cadre du projet gnu (cccp [Stal92]).
2. Présenter ces détails est sans intérêt dans cette thèse.

### IV.2.2.1 *Inclusion textuelle*

Le mécanisme d'*inclusion textuelle* est mis en oeuvre via une directive de la forme `#INCLUDE "<FILENAME>"`. Celle-ci est remplacée par l'intégralité du fichier spécifié. Ce mécanisme permet de décomposer un fichier monolithique en plusieurs fichiers et donc de lui donner une certaine structure. Les directives `#INCLUDE` peuvent être imbriquées mais sans former de cycles.
En fait, c'est un nom de fichier qui est spécifié, pas le contenu d'un fichier. Le résultat obtenu dépend donc de l'état du système de fichiers lors de l'exécution du préprocesseur, c'est à dire de la correspondance `nom-de-fichier -> fichier`.
Cette correspondance peut également être altérée par l'utilisation de *chemins de recherche*. En effet, lors de l'invocation du préprocesseur, il est possible de spécifier une liste de répertoires dans lesquels sera successivement recherché le fichier.

### IV.2.2.2 *Macro-définition et macro-substitution*

Une *macro* est un couple `(nom-de-macro,valeur-de-macro)`. Bien souvent, par abus de langage on parle de "macro" pour l'une ou l'autre de ces deux composantes. La valeur est une chaîne de caractères éventuellement vide[1]. Le mécanisme de *macro-substitution* consiste à remplacer chaque *occurrence de macro* apparaissant dans le fichier CPP par la valeur correspondante. Cette valeur peut elle-même contenir des occurrences de macros, mais sans former de cycles[2].
Dans le cas de la macro-substitution, c'est la correspondance `nom-de-macro -> valeur-de-macro` qui est considérée. La valeur initiale de cette fonction est déterminée lors de l'invocation du préprocesseur (Section IV.2.2.4). Il est également possible de faire varier celle-ci au cours de l'interprétation du fichier CPP en utilisant des mécanismes de *macro-définition* et de *macro-élimination*. La directive `#DEFINE <MACRONAME> <MACROVALUE>` définit une nouvelle macro alors que la directive `#UNDEF <MACRONAME>` permet de l'éliminer.
Ces directives sont traitées en séquence et la portée d'une macro va de sa définition jusqu'à son éventuelle élimination.
Remarquons que, lors d'une macro-définition, la valeur est rangée telle quelle, sans être interprétée. C'est lors de l'utilisation de cette macro que les éventuelles occurrences de macros seront substituées.

### IV.2.2.3 *Compilation conditionnelle*

La *compilation conditionnelle* permet d'inclure ou d'exclure des portions de texte pendant l'interprétation du fichier CPP. Ce mécanisme est basé sur l'emploi de la séquence de directives suivante :

`#IF <CONDITION> <THEN-PART> #ELSE <ELSE-PART> #ENDIF`.

Le terme `<CONDITION>` est une expression entière conforme au langage C[3]. L'évaluation de cette condition se fait en deux temps :

- *Macro-expansion*. Toute occurrence de macro est substituée par sa valeur ou par 0 si celle-ci

1. On devrait distinguer les *macro-simples* qui sont effectivement des chaînes de caractères, des *macros paramétrées*. Le deuxième cas n'a pas d'importance particulière pour le discours de ce chapitre et nous n'en parlerons qu'en de rares occasions.
2. En réalité le comportement de CPP dans de telles conditions dépend des implémentations.
3. Une condition ne met en jeu que des valeurs entières mais il est possible d'utiliser la grande variété des d'opérations définies dans le langage C (opérations arithmétiques, bits à bits, relationnelles, etc.). Cependant on notera l'absence de type énuméré et de valeurs booléennes. L'entier 0 est considéré comme faux, toute autre valeur correspond à vrai.

n'est pas définie (cette dernière règle est arbitraire!). L'opérateur unaire noté `DEFINED` suivi d'un nom de macro permet de tester si celle-ci est définie ou non. A la fin de cette phase de macro-expansion, une expression constante (e.g. 3+1<2) doit être obtenue sinon une erreur est engendrée.

- *Evaluation de l'expression constante*. L'expression est ensuite évaluée en utilisant la sémantique des opérations du langage C.

Si la condition évaluée vaut 0, la partie `<ELSE-PART>` est sélectionnée, sinon ce sera la partie `<THEN-PART>`. Les directives `#ELIF <CONDITION>`, `#IFDEF <MACRONAME>` et `#IFNDEF <MACRONAME>` ne sont que des raccourcis d'écriture.

### IV.2.2.4 *Ligne de commande et prédéfinitions.*

Le préprocesseur CPP peut ou non être intégré au compilateur C. Généralement il s'agit d'un outil découplé. Bien que le langage C ne spécifie pas la manière dont les paramètres sont passés à ce préprocesseur, il existe sur unix un standard de facto en ce qui concerne l'invocation de cet outil. ***La ligne de commande CPP*** permet d'indiquer :

- Le *nom du fichier d'entrée*.
- Le ***chemin de recherche*** utilisé pour déterminer la correspondance `nom-de-fichier -> fichier`. Ceci se fait via la répétition d'options de la forme `-I<DIRNAME>`.
- La *valeur initiale des macros* via les options `-D<MACRONAME>=<MACROVALUE>` pour les définitions et `-U<MACROVALUE>` pour les éliminations. Les macros qui ne reçoivent pas de valeur sont considérées comme indéfinies.

Le lecteur est en droit de se demander pourquoi il existe une option d'élimination de macros si, de toute façon, les macros dont la valeur n'est pas spécifiée sont considérées comme indéfinies. En fait c'est cette dernière affirmation qui est fausse.
La première action réalisée par le préprocesseur est de définir un chemin de recherche et des macros. On parlera respectivement du ***chemin de recherche prédéfini*** et des ***macros prédéfinies***. Ces ***prédéfinitions*** dépendent de chaque implémentation de CPP et devraient simplifier les problèmes de portage entre plates-formes[1]. L'usage des macro prédéfinies est commun en pratique même s'il soulève de nombreux problèmes [Stal92].

### IV.2.2.5 *Synthèse*

CPP propose uniquement des mécanismes de bas niveau : la macro-substitution, la macro-définition, l'inclusion textuelle, les chemins de recherche, et finalement la compilation conditionnelle. En fait tous ces mécanismes sont étroitement liés.

1. Par exemple le répertoire "/usr/include" fait généralement partie du chemin de recherche prédéfini. La macro nommée "unix" est usuellement définie sur cette plate-forme et il est ainsi possible d'inclure ou d'exclure des portions de code en fonction de cette information.

## IV.2.3 Usages

Dans la section précédente le préprocesseur CPP a été présenté d'un point de vue technique ; sans faire allusion, ou presque, à la manière dont celui-ci était utilisé. Ici au contraire, notre but est d'examiner l'usage qui en est fait dans le cadre des langages de programmation, plus particulièrement dans le cas de C. Il s'agit de donner au lecteur une idée de la vision que pourrait avoir un chargé de maintenance sur l'utilisation pratique d'un préprocesseur. Certains exemples sont donnés dans cette intention.
Cinq facettes de CPP sont considérées : (1) CPP et programmation détaillée, (2) CPP et architecture, (3) CPP et portabilité, (4) CPP et configuration, (5) CPP et réutilisation. Ce découpage est arbitraire car en fait ces différents aspects interviennent simultanément. Tous reposent d'ailleurs sur l'utilisation des mêmes mécanismes de base. L'un des problèmes de retro-ingénierie est justement de retrouver l'usage à partir des mécanismes.

### IV.2.3.1 *CPP et programmation détaillée*

La possibilité d'associer un nom à une chaîne de caractères est un moyen d'abstraction aussi rudimentaire qu'utilisé.
Très souvent la définition de macro permet de pallier à l'absence de définition de constantes dans le langage C.
Une définition de macro peut également être associée à d'autres fragments du langage, par exemple aux instructions. Dans ce cas l'idée est de définir des ***procédures "en lignes"***. Il s'agit d'utiliser le mécanisme de substitution lors de l'exécution du préprocesseur plutôt qu'un appel de procédure dynamiquement. C++ a avantageusement intégré cette possibilité sous la forme du mot clé `INLINE` précédant la définition d'une fonction.

### IV.2.3.2 *CPP et architecture*

Grâce à la compilation séparée, un logiciel peut être décomposé en différentes unités de compilation. C'est une étape importante en direction de la modularité. Par contre la distinction entre interface et réalisation n'est pas faite et la technologie standard des éditeurs de liens n'assure aucun contrôle de types inter-modules. Pour pallier à cet inconvénient l'inclusion textuelle est employée pour implémenter la notion d'interface. Les définitions "exportées" par un module sont regroupées dans un même fichier d'inclusion (typiquement les définitions de constantes, de types, les entêtes de procédures, etc.). Ce fichier est inclus textuellement par les différents modules clients pour assurer la cohérence de type entre les utilisations d'une ressource et sa définition. Les bibliothèques du système Unix sont représentatives de cet usage. Si l'inclusion textuelle n'a pas encore été abandonnée, malgré les inconvénients qu'elle présente, c'est sans doute parce qu'elle est immédiate à mettre en oeuvre et relativement efficace[1].

1. Les problèmes d'efficacité en terme de temps de compilation ne doivent pas être négligés, surtout lorsque l'on sait que, dans certaines applications, 70% du temps de compilation est consacré à la lecture des interfaces (par exemple dans le cas de programmes graphiques où un nombre très important de déclarations sont nécessaires). Ce problème a d'ailleurs donné lieu à des implémentations particulières du préprocesseur CPP où les fichiers d'inclusion sont "précompilés" [Litm93].

### IV.2.3.3 *CPP et portabilité*

L'usage des macro-processeurs a souvent été associé aux problèmes de portabilité [Brow74] [Brow77]. C'est le cas aussi de CPP [Fowl&al95b].
L'importance de cette problématique a été étudiée dans le Chapitre I (Section I.4.4.5). Rappelons simplement que les chargés de maintenance sont confrontés à de fortes contraintes de temps. Sous la pression, des solutions "vite-fait-mal-fait" risquent d'être retenues. Porter un logiciel consiste plus souvent à rapiécer le texte source des programmes plutôt qu'à restructurer le logiciel si cela est nécessaire.
Dans un tel contexte, la flexibilité de la compilation conditionnelle est fort utile et des constructions du style `#IFDEF SUN ... #ELSE ... #ENDIF` apparaissent rapidement dans le code source. Elles permettent de sélectionner les parties de code spécifiques à chaque plate-forme. L'utilisation des macros prédéfinies est alors la règle (sun, unix, vms, msdos,...).
Très souvent les problèmes sont dûs à des différences entre les bibliothèques systèmes. Celles-ci sont elles-mêmes truffées de directives du préprocesseur.
Clairement l'apparition de standards comme POSIX permettent de limiter les variations entre systèmes, mais comme le soulignent de nombreux industriels, l'application de ces standards ne résout que partiellement les problèmes de portage [SpenColly92] [TilbCroo92]. Aujourd'hui ceux-ci ne sont plus uniquement associés aux bibliothèques des systèmes d'exploitation, mais s'étendent au contraire aux gestionnaires de fenêtres, de réseaux ou encore d'objets persistants. Etablir des standards ne peut se faire qu'après des années de maturité. Dans un monde compétitif et en pleine évolution ceci est bien difficile[1].

### IV.2.3.4 *CPP et configuration*

Même si l'on omet les problèmes de portabilité, le préprocesseur CPP est utilisé pour décrire des familles de programmes. Son importance et plus particulièrement ses inconvénients, ont été soulignés à maintes reprises dans la littérature concernant la gestion de configurations [SpenColly92] [GentMSC89] [PlaiWadg93b] [Zell94] [Fowl&al95b]. Les chemins de recherche et compilation conditionnelle peuvent être utilisés pour représenter autant des versions logiques, des versions historiques que des versions coopératives (Section I.4.3.2). Ci-dessous nous ne considérons que la compilation détaillée car c'est ce mécanisme de variation détaillée qui est spécifique à l'utilisation des préprocesseurs.

- *Versions logiques*. L'exemple le plus classique de versions logiques correspond à la situation dans laquelle le programmeur souhaite disposer à la fois d'une version de mise au point et d'une version standard. Les fragments de programmes correspondant aux traces et aux vérifications sont entourées par des directives de compilation conditionnelle contrôlées par la macro `DEBUG` [Abac89]. Une telle solution présente l'avantage (et l'inconvénient) de pouvoir être immédiatement mise en oeuvre, sans nécessiter aucune planification. Même si cela semble être un contre-sens du point de vue génie logiciel, ce ne l'est plus juste avant un "deadline".
  Cette même technique est bien sûre utilisable pour représenter des versions logiques d'une

1. Leurs détracteurs ironisent en faisant remarquer que "l'avantage des standards, c'est qu'il y en a tellement que l'on peut les choisir !".

autre nature, par exemple pour satisfaire les besoins de clients différents. Son principal avantage est que la sélection peut se faire avant l'exécution[1].

- *Versions historiques*. Dans certains cas on constate que la compilation conditionnelle est utilisée pour représenter des versions historiques. Clairement conserver toutes les révisions d'un composant au sein d'un même fichier est impraticable : la présence de toutes les différences entourées par des clauses de compilation conditionnelle le rendrait rapidement illisible. En général cette information n'est d'ailleurs pas très utile au programmeur, sauf si seules certaines versions de fragments sont conservées. Il s'agit alors de "mettre en commentaire" certains fragments de programmes devenus obsolètes mais jugés utiles pour expliquer l'histoire du code. Cet usage est curieusement assez répandu et il explique la présence de constructions de la forme `#IFDEF 0 ... #ENDIF` dans bien des programmes C[2] [SpenColly92].
- *Versions coopératives*. Si plusieurs personnes travaillant sur un même composant veulent être au courant des modifications proposées par les autres, des constructions de la forme `#IFDEF BOB ... #ENDIF` sont susceptibles d'apparaître. Bien évidemment la condition peut faire référence à une activité ou à un rôle plutôt qu'à un personne physique. Cette technique ad-hoc permet à de mener en alternance plusieurs tâches en assurant à la fois indépendance (le programmeur peut compiler sa version indépendamment de celle des autres) et coordination (il peut étudier au fur et à mesure les modifications introduites pas les autres). En l'absence d'outils spécifiques (comme par exemple ICE [Zell94]), elle n'est cependant applicable (appliquée) que dans des contextes particuliers (une ou deux personnes coordonnées et effectuant des modifications mineures).

En résumé, les préprocesseurs sont mieux adaptés à la gestion de versions logiques qu'à celle de versions historiques ou coopératives.

### IV.2.3.5 *CPP et réutilisation*

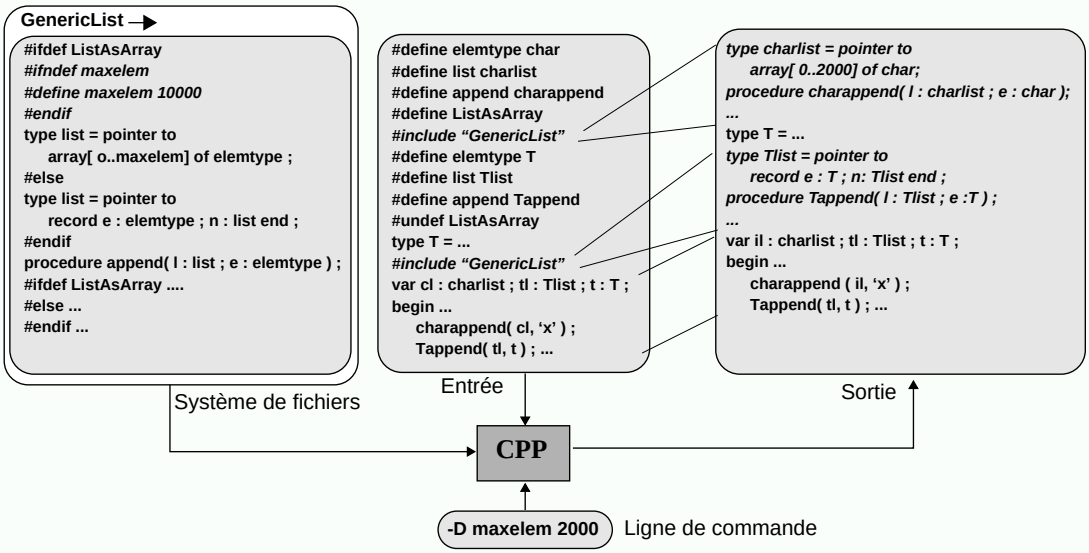
Les mécanismes proposés par CPP offrent des possibilités importantes de parametrisation et de généricité. Cet outil a donc un sens dans le domaine de la réutilisation. Son importance (et ses inconvénients) ont d'ailleurs été mentionnés dans ce domaine (voir par exemple [GrosSnelt93]). Nous nous contenterons ici de donner un exemple (figure 104).

---

1. Ces problèmes sont souvent assimilés aux pratiques peu orthodoxes rencontrées en C, et pour cela ils sont négligés même s'ils sont bien réels. Il est intéressant de remarquer qu'ils sont d'ailleurs intervenus dans la conception d'Eiffel [Meye92], l'un des langages à vocation industrielle les plus modernes. L'instruction composée `DEBUG(<CONDITIONLIST>)<STATEMENTLIST> END` est ignorée à l'exécution si la condition exprimée en termes d'options de compilation est évaluée à faux (il s'agit d'une disjonction). Ce mécanisme est une restriction de la compilation conditionnelle dans la mesure où elle ne s'appliquent qu'à des instructions et non pas à n'importe quel fragment du langage. Le mot clé `DEBUG` suggère que la mise au point est un cas particulier de versions logiques et qu'à ce titre il requiert un traitement spécial. Il n'est pas clair que cette assertion soit justifiée.
2. Ces constructions peuvent sembler bien énigmatiques au premier abord car le fragment de code n'est jamais inclus. Elle correspond pourtant à une opération beaucoup plus naturelle : mettre en commentaire (/* et */) une partie de code. D'un point de vue technique cette solution est préférée car elle permet l'imbrication de commentaires.

*figure 104* Un exemple de généricité avec CPP

A la première lecture, le lecteur trouvera sans doute cet exemple bien peu compréhensible. Alors pourquoi le faire figurer dans cette thèse ? Tout simplement parce qu'il s'agit d'un exemple relativement simple de généricité et qu'il illustre pourtant bien les difficultés auxquelles pourrait être confronté un chargé de maintenance. Celui-ci ne bénéficie ni des explications ci-dessous, ni des aides graphiques présentes dans le schéma. Le lecteur peut aussi constater la gêne visuelle due à la présence de nombreuses directives ainsi que la difficulté à identifier les blocs #ifdef ... #endif en l'absence d'indentation dans le fichier GenericList.




• **Fichier GenericList**. Ce fichier contient des définitions génériques permettant de manipuler des listes d'éléments de type "elemtype". Dans l'exemple le type "list" et la procédure "append" sont respectivement déclarés. Deux types d'implémentations sont proposées : des listes gérées dans un tableau ou des listes chaînées utilisant des pointeurs. La macro ListAsArray permet de sélectionner l'une ou l'autres de ces deux alternatives grâce au mécanisme de compilation conditionnelle. Dans le premier cas la dimension du tableau est un paramètre pouvant être changé. Si aucune valeur n'est spécifiée, la valeur par défaut sera 10000.

• **Fichier d'entrée.** Deux instanciations du module générique sont utilisées ; respectivement pour des listes de caractères stockées dans des tableaux et pour des listes d'éléments de type T gérées avec des pointeurs. Les définitions de macro précédant chaque directive d'inclusion textuelle ont pour but de renommer les entités déclarées dans le module générique et de définir le type des éléments de la liste. Dans la dernière partie du fichier un extrait du programme client est montré. Les listes de caractères et d'éléments de type T sont manipulées naturellement en utilisant des procédures différentes.

• **Ligne de commande.** Dans l'exemple, l'application du préprocesseur se fait en définissant, via la ligne de commande, une taille maximale de 2000 éléments pour les listes. Remarquons que cette décision peut avoir été prise par le programmeur lors de la compilation et qu'elle n'implique aucune modification du code source.

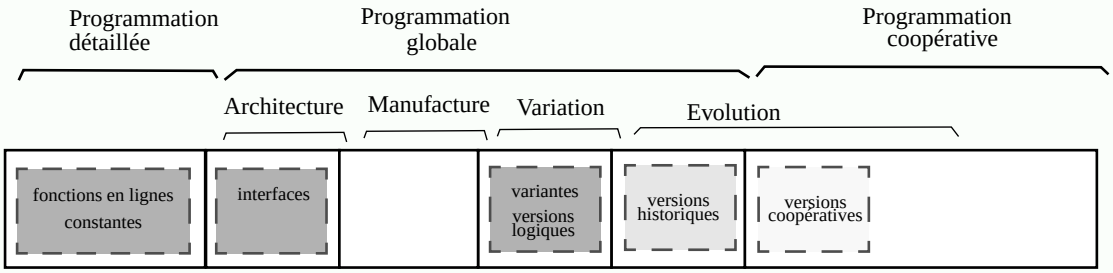
• **Fichier de sortie.** Dans le fichier de sortie on remarquera les déclarations nécessaires pour les listes de caractères, puis pour les listes d'éléments de types T. Dans la dernière partie de ce fichier le corps du programme est recopié tel quel. On remarquera que les marques reliant le fichier d'entrée et de sortie sont utiles pour "suivre à la trace" l'exécution de préprocesseur.

Remarquons finalement que le contrôle de types sera naturellement assuré par le langage lors de la compilation. Cette solution est donc relativement sûre. Par contre il n'est pas possible d'assurer statiquement la validité des programmes générés à l'aide du module générique.

### IV.2.3.6 *Synthèse*

Les sections ci-dessus ont présenté le rôle de CPP dans le domaine de la programmation détaillée, de l'architecture, de la portabilité, de la gestion de configurations et de la réutilisation. Il ressort de cette analyse que le spectre couvert par cet outil est particulièrement large. Ce n'est pas étonnant dans la mesure où les mécanismes proposés sont aussi flexibles que de bas niveau. CPP peut être considéré comme un assembleur pour la programmation globale (figure 105).

*figure 105* Usages de CPP




## IV.2.4 Problèmes

*"Conditional compilation constructs (#ifdef), especially nested ones, lead to programs that are virtually unreadable by humans" (W.M. Gentleman).*

Après une section montrant la gamme importante d'utilisation de CPP vient naturellement le revers de la médaille : les problèmes. CPP est connu comme un outil aussi problématique que flexible [Gent89] [Spul90] [SpulSaje92] [SpenColly92] [TilbCroo92] [Stal92] [KronSnel94] [Sing92] [KronSnel94] [Stro94] [Favr95a].

L'exemple de module générique présenté ci-dessus (figure 104) montre que comprendre des méta-programmes écrits avec CPP est loin d'être facile. L'utilisation extensive de cet outil peut devenir un cauchemar pour la maintenance. Des morceaux de code incompréhensibles sont reproduits dans [SpenColly92]. Il n'est pas nécessaire d'aller chercher bien loin des exemples ; le lecteur est invité à consulter le contenu de quelques interfaces Unix "portables"[1] sur la station de travail qu'il utilise. Dans la figure 106 certaines parties de code ont été extraites de telles interfaces. Les commentaires sont forts utiles mais encore faut-il qu'ils existent et qu'ils soient à jour... Après des années de maintenance ce n'est pas toujours le cas[2].

Constater que CPP pose des problèmes n'est pas suffisant. Reste à les identifier plus précisément. Ci-dessous différents *p*roblèmes *c*oncrets sont présentés succinctement.

PC1 La sémantique de CPP n'est pas claire. Dans les cas limites, il est difficile de prévoir le comportement de cet outil. D'ailleurs il dépend parfois des implémentations. Même dans des cas simples, la sémantique de certaines directives peuvent dérouter le programmeur.

PC2 Dans les situations anormales, CPP applique des règles arbitraires et dangereuses plutôt que de signaler des erreurs. La philosophie générale peut être résumée ainsi : "générer éventuellement n'importe quoi, mais générer"[3]. Détecter l'occurence des situations anormales est quasiment impossible manuellement.

1. Dans un noyau unix on trouve le commentaire "/* *you are not expected to understand this* */".
2. *"Common wisdom, n: something that is widely known but usually ignored.* (Unix programmer's definition)".
3. Par exemple [Stal92] décrit l'implémentation CPP de gnu. On trouve de nombreuses règles arbitraires annotées avec des phrases telles que "*It is not clear that this behaviour would ever be useful, but it is specified by the ANSI C standard, so you may need to understand it*". Autre exemple ; le fait de substituer les macros non définies par 0 dans les conditions est aussi arbitraire que dangereux surtout dans des expressions arithmétiques (e.g. A+B<C).

*figure 106* Du code utilisé tous les jours...

```
/*  @(#)buserr.h 2.2 92/05/27 SMI*/

/* This file is not needed for sun4m builds.
 * For historical reasons it was copied into this directory.
 * Since Some files in the non-sun4m directories include
 * this file, I will offer an empty file here
 */
```

```
/*
 * @(#)keyboard.h 1.11 88/09/28
 * Copyright (c) 1986 by Sun Microsystems, Inc.
...
 * Since this file is included by <mon/eeprom.h>, which is
 * included by <machine/eeprom.h>, kernel files (specifically
 * kbd.c) include it, too. The kernel file <sundev/kbd.h>
 * contains parallel definitions for the constants in this file.
 * All this should be cleaned up, but in the meantime, the
 * following #ifndef is a kludgey solution.
 */
#ifndef IDLEKEY
...
```

***(a)* Du code commenté**

Ces extraits proviennent de bibliothèques standards. Les commentaires reflètent des situations communes en pratique. Sans eux le chargé de maintenance aurait sans doute bien du mal à comprendre ces structures. Remarquons au passage qu'après presque 10 ans le deuxième fichier (keyboard.h) est encore en service tel quel ; aucune restructuration n'a été faite. Quel chargé de maintenance, après une telle période, pourrait se rappeler de ce rapiéçage sans la présence d'un commentaire ? (D'ailleurs est il à jour ?)

```
/* name of this site */
#ifdef GETHOSTNAME
    char *hostname;
#   undef SITENAME
#   define SITENAME hostname
#else /* !GETHOSTNAME */
#   ifdef DOUNAME
#       include <sys/utsname.h>
        struct utsname utsn;
#       undef SITENAME
#       define SITENAME utsn.nodename
#   else /* !DOUNAME */
#       ifdef PHOSTNAME
            char *hostname;
#           undef SITENAME
#           define SITENAME hostname
#       else /* !PHOSTNAME */
#           ifdef WHOAMI
#               undef SITENAME
#               define SITENAME sysname
#           endif /* WHOAMI */
#       endif /* PHOSTNAME */
#   endif /* DOUNAME */
#endif /* GETHOSTNAME */
```

***(b)* Du code...**

Bien souvent les commentaires sont absents...

*"It's most unlikely that **anyone** understands this code any more. In such situations, maintenance is reduced to hit-or-miss patching."* (extrait de [SpenColly92]).

PC3 CPP n'interprète pas les macros lors de leur définition mais lors de leur substitution. Considérons par exemple la séquence *#define A 1  #define B A  ...  #define A 10 ... B* . Lorsqu'il lit la définition de B, le lecteur va sans doute penser que B vaut 1, pourtant la dernière occurrence de B est substituée par 10 (en fait par A puis par 10). Bien souvent de telles situations correspondent à des erreurs ou tout au moins risquent de provoquer des erreurs. Leur détection est très difficile ; surtout qu'elles interviennent généralement dans les cas les plus complexes : ceux où de nombreux fichiers sont impliqués.

PC4 CPP ne propose aucun mécanisme d'encapsulation. La définition d'une nouvelle macro dans un fichier peut provoquer des effets de bords dans d'autres fichiers. En effet, comme la portée des macros est globale, une nouvelle définition a un effet sur *tout* le texte suivant cette définition ; lors de *toutes* les exécutions de CPP faisant référence au fichier modifié. Autrement dit, il est préférable de choisir des noms de macros dont on est sûr qu'ils ne seront pas utilisés ailleurs ! De surcroît certaines implémentations anciennes limitent la taille des noms de macros à 8 caractères. Avec de telles contraintes, il n'est pas surprenant qu'ils ne soient pas toujours très explicites...

PC5 Il est très difficile de savoir quels sont les paramètres d'un fragment de texte CPP. Par exemple quelles sont les macros pouvant influer sur le code présenté dans la figure 106.b ? La valeur initiale de la macro "SITENAME" a-t-elle une influence dans certains cas ?

PC6 Souvent le chargé de maintenance cherche à se focaliser sur une seule variante. Il doit alors exécuter *mentalement* les directives CPP pour déterminer quels sont les fragments qui font partie de cette variante. L'humain n'est pas un préprocesseur et pourtant il doit remplir les

mêmes tâches !

PC7 De nombreuses variantes peuvent être générées à partir d'un fichier CPP, mais sans savoir quels sont les paramètres de ce fichier (PC5), comment connaître leur nombre ? Comment tester ces variantes dans de telles conditions ? En pratique seul un petit nombre de variantes sont testées. Même si un effort plus conséquent est fait, comment savoir quelle est la couverture de tests ?

PC8 Lire une définition de macro n'est pas suffisant pour déterminer son utilité. En effet une macro peut être utilisée comme un sélecteur de version (typiquement *#define debug* 1) ou comme un paramètre (par exemple *#define size 1000*). Il est alors nécessaire d'observer toutes les occurrences d'une macro pour en déterminer l'utilité (les occurrences de macros sont souvent physiquement éloignées de leur définition et il faut éventuellement consulter des centaines de lignes dans de nombreux fichiers...).

PC9 Par défaut une valeur indéfinie est affectée aux noms de macros. Il n'y a donc pas de différence entre l'absence volontaire d'une définition et un oubli malencontreux.

PC10 Dans la mesure où la compréhension de fichiers CPP est difficile, le chargé de maintenance tente de faire le moins de modifications possible. Dans bien des cas des variantes deviennent inutiles. Comme il n'est pas toujours évident de savoir quels fragments de textes éliminer, ceux-ci sont souvent laissés tels quels. Peu à peu les fichiers CPP "s'encrassent" avec du code mort quasiment impossible à détecter.

Les problèmes mentionnés ci-dessus sont indépendants du fait que CPP soit utilisé avec un langage de programmation ou non. Si c'est le cas, d'autres problèmes encore apparaissent !

PC11 CPP est basé sur une représentation textuelle et non pas syntaxique [WinkStof88] [SpulSaje92] [WeisCrew93]. Les directives peuvent donc "rompre" certaines frontières syntaxiques. C'est gênant pour la lecture mais ce n'est pas grave en soit. Par contre dans certains cas, cela peut mener à des problèmes bien plus subtils. Par exemple la séquence `#define A 1 ... #define B A+4 .... c := B*2` affecte 9 à la variable c ! En réalité l'expression B*2 est substituée par 1+4*2... De tels problèmes peuvent être aussi graves que difficiles à détecter sans outils.

PC12 Lire un programme n'est pas aisé ; lire un fichier CPP n'est pas simple ; lire un méta-programme peut devenir quasiment impossible ! Le problème est lié au fait qu'il faut simultanément considérer les entités du préprocesseur et celles du langage de programmation[1]. Les flots d'informations de chaque niveau s'enchevêtrent. Entre autres il n'y a pas de différences lexicales entre les noms de macros et les autres identificateurs du langage, ce qui permet de créer des alias (comme par exemple "SITENAME" dans la figure 106.b). De même la compréhension du fichier `GenericList` (figure 104) met en oeuvre aussi bien des concepts de CPP que ceux de C.

PC13 Alors que les chargés de maintenance considèrent l'entrée de CPP, les compilateurs ou les autres outils considèrent habituellement sa sortie. Cette distance peut être une source d'incompréhension et il n'est pas toujours facile de faire la relation entre l'entrée et la sortie

1. Dans certains cas les concepts manipulés sont déconnectés voir incohérents. Ce n'est pas le cas de l'association CPP/C : par exemple les conditions CPP s'expriment avec les opérateurs de C. De même CTF est bien intégré à PL/1.

du préprocesseur [LivaSmal94].

PC14 Aucun contrôle sémantique n'est assuré sur un fichier CPP. Pour être sûr qu'une famille de programmes est correcte on devrait tester successivement chaque variante ; mais pour cela encore faudrait-il savoir comment les obtenir (PC5,PC7). Cette caractéristique contraste par rapport aux constructions génériques proposées dans des langages comme Ada qui assurent qu'une unité générique est sémantiquement correcte.

PC15 La présence de directives CPP est un problème pour l'utilisation d'outils d'analyse ou de transformation de programmes. Dans certains cas, elle impose des limitations sérieuses sur la possibilité d'appliquer des techniques pourtant fort utiles. Le problème provient du fait que celles-ci sont prévues pour prendre en compte une seule variante simultanément. Même si certains outils incluent des traitements spécifiques pour lever cette limitation, ils ne prennent en général pas en compte toutes les spécificités du préprocesseur et ne sont donc pas applicables en grandeur nature sur des programmes C industriels [Krus83] [Munc93] [KronSnel94] [Zell94].

Cette énumération de problèmes est loin d'être exhaustive. Elle donne néanmoins une idée des difficultés que rencontrent les chargés de maintenance en pratique et explique pourquoi l'on parle parfois de "cauchemar" et de "maux de têtes".

### IV.2.5 Conclusion

Dans cette section nous avons vu que :

- CPP est très utilisé en pratique (IV.2.1).
- Les mécanismes qu'il propose sont de bas niveau (IV.2.2).
- Leurs usages sont variés et concernent différents aspects de programmation globale (IV.2.3).
- Cet outil est obsolète, de conception douteuse et pose de nombreux problèmes (IV.2.4).

La première et la dernière remarque donnent lieu à ce que nous avons appelé ***le paradoxe CPP*** [Favr95a]. Il s'agit en fait d'un ensemble de paradoxes énoncés dans l'introduction de cette thèse :

> L'utilisation extensive et excessive d'un préprocesseur rend les programmes impossibles à lire ; pourtant ils sont utilisés pour écrire de grands volumes de code... L'utilisation d'un préprocesseur rend la maintenance difficile et pourtant ils sont largement utilisés par les chargés de maintenance... La présence des préprocesseurs est un problème pour les programmeurs, les chargés de maintenance, les constructeurs d'outils et pourtant ils sont toujours là... *Malgré les problèmes occasionnés par les préprocesseurs, aucun support ne leur est dédié...*

#### *Différentes réactions possibles...*

Face à cette foule de problèmes, plusieurs réactions sont possibles :

- Les chargés de maintenance peuvent prendre une aspirine et continuer à rapiécer les rapiéçages avant l'expiration des délais.
- Dans le monde académique, on aura plutôt tendance à se détourner de tels problèmes jugés trop "techniques". Certains chercheurs diront sans doute "de toute façon avec la méthode 'alpha' cela n'arrivera pas" et 20 ans après, quand des millions de lignes de code auront été développées avec 'alpha', ils pourront dire "avec la méthode 'bêta' cela n'arrivera pas...", etc.

Il ne s'agit là que des extrêmes ; des solutions intermédiaires existent :

- Dans l'industrie, une discipline peut être imposée pour l'usage futur de ces outils [Abac89] [SpenColly92]. Un "meilleur" préprocesseur peut aussi être réalisé [TilbCroo92].
- Des recherches peuvent être entreprises pour concevoir des outils beaucoup plus élaborés et généraux [TrygGull95] ou tout au moins conceptuellement plus propres [Zell94]. Proposer des outils et des méthodes limitant les problèmes tout en gardant la même technologie est une autre alternative [GentMSC89] [PlaiWadg93b] [Fowl&al95b].

Hélas, ces solutions sont uniquement tournées vers le futur et ne prennent pas en compte la maintenance du code existant. Pourtant, c'est dans ce contexte que les problèmes sont les plus épineux. Définir des outils facilitant la maintenance et la ré-ingénierie de tels programmes est un but tout à fait valide.

- Des solutions ad-hoc rapidement applicables peuvent être proposées [Spul90]. Hélas, ces solutions sont spécifiques à un préprocesseur donné.
- Au contraire des outils de maintenance et de ré-ingéniérie peuvent être basés sur l'utilisation d'abstractions [KronSnel94] [Snel95]. Cette approche est plus coûteuse, en tout cas à court terme.

Toutes les réactions présentées ci-dessus correspondent à des points de vue différents mais tout à fait *complémentaires*. Il ne s'agit en fait que de la spécialisation du discours tenu dans le Chapitre I au cas des préprocesseurs. Le modèle hélicoïdal peut être utilisé pour situer les différentes approches.

# IV.3 CPP et le modèle abstrait

Jusque là CPP a été présenté d'un point de vue très concret. Le Chapitre II vantait le mérite des abstractions. Il est temps maintenant d'appliquer ce discours au cas précis de CPP.

Le fait de chercher à utiliser des abstractions peut correspondre à deux objectifs :

- O3 Favoriser un processus de rationalisation dans le domaine de la programmation globale.
- O6 Faciliter la maintenance et la ré-ingénierie de programmes utilisant des préprocesseurs.

Cette section se focalise sur le premier objectif (O3). Il s'agit plus particulièrement de montrer que, malgré ses apparences souvent très frustes, CPP n'est pas très éloigné d'autres systèmes plus modernes.

CPP étant essentiellement utilisé pour résoudre des problèmes de programmation globale, on doit pouvoir utiliser les abstractions introduites dans le modèle abstrait (Chapitre II). Même si elles ne sont pas directement utilisables, elles devraient tout au moins faciliter la compréhension des problèmes et permettre de situer ce préprocesseur par rapport aux autres systèmes.

Puisque CPP se concentre sur les problèmes d'architecture et de variation, les notions d'*objets structurés* (II.2.3) et d'*objets génériques* (II.2.5) doivent pouvoir être appliquées. C'est le cas aussi des concepts associés : *domaine* et *codomaine* (II.2.5), *espace à n-dimensions* (II.4.5) *mécanisme de substitution* (II.3.4) et *sélection* (II.3.5), *estampilles* (II.4.5) et *fragments* (II.5.2), *héritage des choix* (II.5.3.2) et *synthèse des variantes* (II.5.3.1), *fonction d'héritage* (II.5.3.2), *fonction de simplification de domaine* (II.4.7), etc.

Ici nous cherchons à mettre en relation ces concepts abstraits aux techniques concrètes CPP.

## IV.3.1 Objets génériques structurés

CPP peut être vu comme un *langage d'objets génériques structurés*. L'outil CPP est bien évidemment l'*interpréteur* de ce langage et les fichiers CPP correspondent à des (*g-*)*programmes*, c'est à dire à des représentations concrètes d'objets génériques.

D'un point de vue abstrait les (g-)programmes CPP sont des *fonctions*.
Le *codomaine* de ces fonctions est l'ensemble des listes de lignes de textes (donc $\mathcal{C}$=[[char]]). Il s'agit donc d'un *codomaine structuré*. CPP est basé sur une *variation détaillée* : les différentes lignes d'un fichier peuvent varier indépendamment les unes des autres.

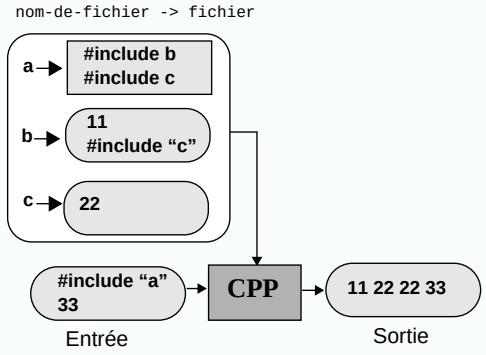
## IV.3.2 Mécanismes de substitution

Bien que les mécanismes proposés par CPP semblent très différents, ceux-ci ne sont, en réalité, que des cas particuliers des mécanismes présentés dans le modèle abstrait. Considérons tout d'abord le *mécanisme de substitution* (II.3.4). Celui-ci est basé sur la notion de *variable* et d'*occurence de variable*. Un *mécanisme de liaison* permet d'associer une *valeur* à une variable. Toutes les occurrences liées de cette variable sont substituées par cette valeur (II.3.4).

En première approximation, l'inclusion textuelle peut être vue comme un mécanisme de substitution. C'est bien évidemment le cas aussi pour la macro-substitution. La figure 107 présente des représentations sémantiquement équivalentes.

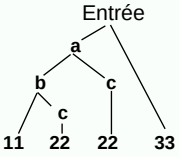
*figure 107* Inclusion textuelle et macro substitution : substitutions

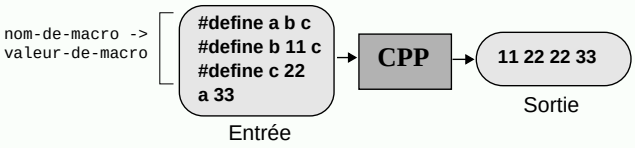
Chaque représentation est basée sur l'utilisation de trois variables : a,b et c. La variable a est liée avec une valeur contenant une occurence de b et une occurence de c. La variable b contient une occurence de c. Finalement la valeur constante "22" est liée à c. Le résultat obtenu est le même dans chaque cas. Simplement les domaines des variables sont différents, tout comme le mécanisme de liaison et la syntaxe. Dans le premier cas le mécanisme de liaison est externe au programme ; dans le deuxième cas il est interne.



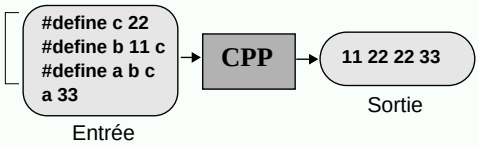

***(a)* Inclusion textuelle**

Dans cet exemple, le fichier d'entrée contient une directive d'inclusion textuelle faisant référence au fichier nommé "a". Elle est substituée par le fichier d'inclusion correspondant ; lui-même contient deux directives d'inclusion, elles-mêmes substituées, etc. L'arbre de substitution correspondant est représenté ci-contre.




***(b)* Macro-substitution (solution 1)**




***(c)* Macro-substitution (solution 2)**

La fonction `nom-de-macro -> valeur-de-macro` est construite dynamiquement lors de l'interprétation du fichier grâce aux directives #define. Aucune substitution n'est faite à ce moment là. Finalement l'occurence de la macro a est substituée par la valeur correspondante et ainsi de suite.

Le tableau 15 fait l'analogie entre l'inclusion textuelle et la macro-substitution. L'une des différences est que le premier mécanisme est habituellement utilisé avec une granularité forte (un fichier est généralement une "longue" chaîne de caractères) alors que la seconde correspond plutôt à une granularité fine (une macro est une "petite" chaîne de caractères). Ces considérations font que la vision habituelle de chaque mécanisme diverge une fois de plus pour des raisons liées à la technologie des systèmes d'exploitation.

TABLEAU 15 Comparaison entre l'inclusion textuelle et la macro substitution

| | **Inclusion textuelle** | **Macro substitution** |
|---|---|---|
| occurence de variable | directive d'inclusion | occurrence de macro |
| valeur | fichier | macro |
| granularité | granularité forte | granularité fine |
| fonction utilisée | `nom-de-fichier -> fichier` | `nom-de-macro -> valeur-de-macro` |
| variation de cette fonction | constante durant l'exécution | varie au cours de l'exécution |
| mécanisme de liaison | création d'un fichier<br>changement d'une règle de recherche | définition de macro<br>prédéfinition |
| moment de liaison | pas de substitution lors de la définition,<br>mais lors de l'utilisation | pas de substitution lors de la définition, mais<br>lors de l'utilisation |

TABLEAU 15 Comparaison entre l'inclusion textuelle et la macro substitution

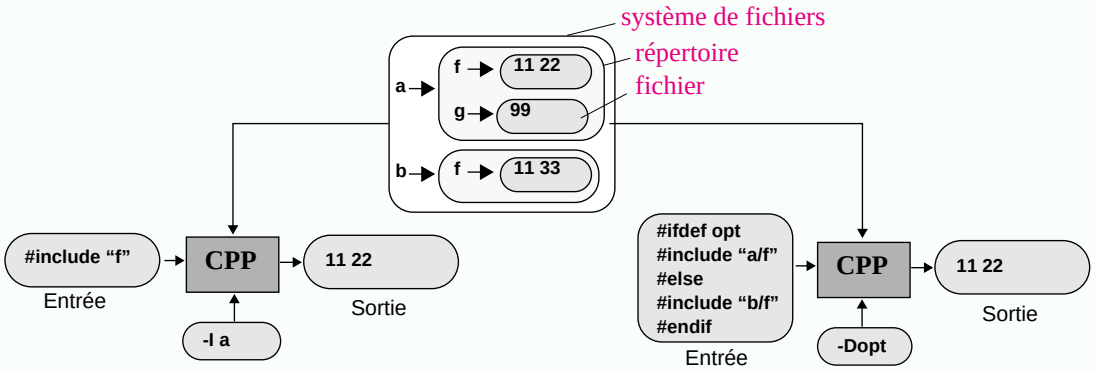
| | Inclusion textuelle | Macro substitution |
|---|---|---|
| valeur non définie | génère une erreur (fichier inexistant) | le nom de la macro est laissé intact |
| ré-entrance | un fichier CPP peut contenir n'importe quelles directives CPP. Elles seront à leur tour interprétées. | une macro CPP ne peut pas contenir des directives CPP, hormis des occurrences de macro qui seront à leur tour interprétées. |

Remarquons finalement qu'un fichier d'inclusion peut contenir n'importe quelle directive CPP. Ce n'est pas le cas des macros. Nous verrons par la suite que cette dissymétrie a des conséquences importantes (Section IV.4).

### IV.3.3 Mécanismes de sélection

Le *mécanisme de sélection* est basé sur le choix d'un élément parmi un ensemble décrit en extension. Dans le cas de CPP une telle sélection peut se faire grâce au mécanisme de compilation conditionnelle mais aussi via l'utilisation de chemins de recherche. La figure 108 met en relation ces mécanismes en présentant deux représentations équivalentes.

*figure 108* Chemins de recherche et compilation conditionnelle : sélections




***(a)* Sélection par un chemin de recherche**
L'option -I permet de sélectionner le répertoire nommé "a". Le nom de fichier "f" est donc associé au fichier "a/f". Inversement on aurait pu sélectionner le répertoire "b".

***(b)* Sélection par compilation conditionnelle**
Cet exemple utilise la compilation conditionnelle pour obtenir le même résultat avec le même système de fichier qu'en (a). Si la macro "opt" est définie c'est "a/f" qui sera sélectionné, sinon ce sera "b/f".

### IV.3.4 Objets génériques par morceaux en intention

Dans le Chapitre II et II.5 de nombreuses variations ont été présentées autour du mécanisme de sélection (II.4 et II.5). Celles-ci peuvent être directement appliquées à une restriction de CPP ne contenant que la compilation conditionnelle.

Le *domaine* des objets génériques construits est alors l'ensemble des fonctions partielles associant des valeurs entières aux noms de macros (i.e. $\mathcal{D}$=`nom-de-macro->int`). Par exemple `{DEBUG->1;SIZE->30}` ou `{SIZE->50;A->3;B->5}` sont deux *choix* différents. Un tel domaine ne limite pas a priori le nombre de *dimensions* de l'espace représenté (II.4.3)[1]. Concrètement cela

1. ce qui contraste avec des modèles comme Extra-V [Scio94].

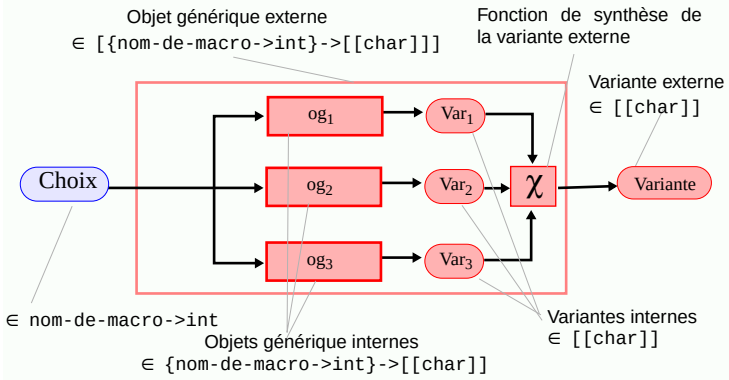
veut dire que dans le cas de CPP, il est possible de donner une valeur à n'importe quelle macro, indépendamment du fait que celle-ci intervienne ou non dans le processus de sélection.

Les *estampilles*, ici les conditions, décrivent des *ensembles de choix en intention* (II.4.5.3) (i.e. $\mathcal{E}$={`nom-de-macro->int`}$_i$ ). Par exemple ‘`SIZE<100`’ est une estampille décrivant en intention un sous-ensemble de choix. Les deux exemples ci-dessus font partie de ce sous-ensemble.

Les estampilles sont associées aux *fragments*, ici à des listes de lignes (donc $\mathcal{F}$=[[char]]), sous la forme d'arbres puisque ce sont des *constructions conditionnelles imbriquées* qui sont utilisées (II.4.6). Les fichiers CPP basés uniquement sur la compilation conditionnelle ne sont qu'une répétition de telles constructions conditionnelles et l'on a donc $\mathcal{G}$ = [ Tree( {`nom-de-macro->valeur-de-macro`}$_i$ x [[char]] ) ]. En terme de modèle abstrait il s'agit d'une liste d'*objets génériques par morceaux en intention* basés sur des *constructions conditionnelles*.

En l'absence du mécanisme de définition de macro, les objets génériques ainsi construits sont basés sur un *domaine universel* ; autrement dit toutes les conditions du fichier sont contrôlées par un choix unique (figure 109).

*figure 109* Un domaine universel pour CPP sans macro-définitions




Cette figure est à comparer à la figure 53 (p.114). L'objet générique est une liste de trois objets génériques internes (par exemple ce peut être des directives de compilation conditionnelle). Le même choix est passé a tous les objets génériques. Chacun génère une variante. Celles-ci sont ensuite regroupées grâce à la fonction de synthèse $\chi$ qui ici correspond à la concaténation des lignes produites (l'opérateur @).

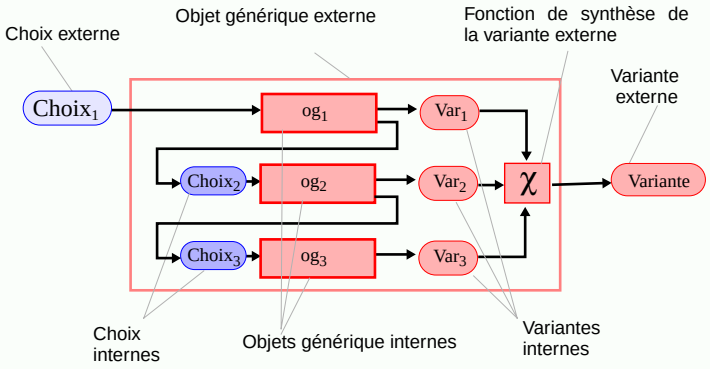
Cette discussion montre d'une part que le modèle abstrait est adapté et que les notions de *factorisation* et de *développement* sont applicables (II.4.5.1), d'autre part que cette restriction de CPP est un modèle compatible avec des systèmes comme CoV [Lie90] [Munc93], P-Edit [Krus83] ou ICE [Zell94]. Cette dernière constatation est plus ou moins claire pour les auteurs de ces outils. Par contre ils ne mentionnent pas toujours explicitement que c'est à cette restriction de CPP qu'ils font référence.

## IV.3.5 Modèle de calcul impératif

Dans le Chapitre II nous avions présenté le *mode de calcul fonctionnel* basé sur l'*héritage des choix internes* et la *synthèse de la variante externe* (II.5.3). Avec un tel mode de calcul l'ordre de génération des variantes internes n'a pas d'importance.
Au contraire, CPP est basé sur un ***mode de calcul impératif***. En effet le mécanisme de définition de macros qui permet de changer la valeur d'un choix en fonction de la sélection d'une variante. Concrètement cette caractéristique correspond aux cas où des définitions de macros sont inclues dans des directives de compilation conditionnelle. La sélection d'une variante modifie alors le

choix, ce qui bien évidemment a une influence sur les objets génériques qui suivent. Ce mode de calcul est illustré par la figure 110.

*figure 110* Modèle de calcul impératif




(à comparer avec la figure précédente et avec la figure 53 (p.114))

La sélection d'une variante génère une variante, mais elle peut aussi modifier la valeur des macros passées aux objets génériques qui suivent.

Un tel mode de calcul est loin d'être satisfaisant car toute modification d'une macro peut entraîner des effets de bords (PC7). De plus la logique de contrôle de cohérence (les fonctions d'héritages dans le cas fonctionnel) est répartie dans le code ; ce qui le rend beaucoup plus complexe à comprendre.

## IV.3.6 Simplification de domaine

Lorsque le domaine d'un objet générique est très complexe, il peut être nécessaire d'utiliser des techniques de *simplification de domaine* (II.4.7). Certains logiciels portables font intervenir des centaines de macros différentes ; dans de telles conditions ces techniques sont indispensables.

L'une des techniques est d'utiliser des *valeurs par défaut* pour les paramètres. Ada offre cette possibilité. Dans le cas de CPP on trouve couramment des constructions de la forme suivante : `#IFNDEF A #DEFINE A 4 #ENDIF` (si la macro `A` n'est pas définie, la valeur 4 lui est affectée).

Parfois il est utile de *réduire le domaine* ou de le rendre plus *abstrait*. C'est le cas notamment lorsque un objet générique est paramétré par un grand nombre de dimensions mais que seules certaines configurations sont valides ou ont un sens pratique. Par exemple un programme C portable peut utiliser un très grand nombre de macros telles que `HAS_SETRUID`, `HAS_SETREGID`, `HAS_SOCKET`, etc. Le nombre de combinaisons possibles est exponentiel alors qu'en pratique seul un nombre limité de plates-formes existe. Dans ce cas on pourra trouver dans le code des directives de la forme `#IFDEF TRUCIX #DEFINE HAS_SETRUID 1 #DEFINE HAS_SOCKET 1 #ELIF GROSIX ...` Concrètement il s'agit plus souvent de fichiers de configuration différents pour chaque plate-forme. Le principe est le même : le choix du fichier (un choix abstrait) détermine les valeurs de différentes macros (un choix concret).

## IV.3.7 Un exemple

Pour faire ressortir les similarités et les différences existant entre le modèle abstrait et la réalité CPP, l'un des moyens le plus simple est sans doute de reprendre le même exemple. C'est ce qui est fait dans la figure 111[1].

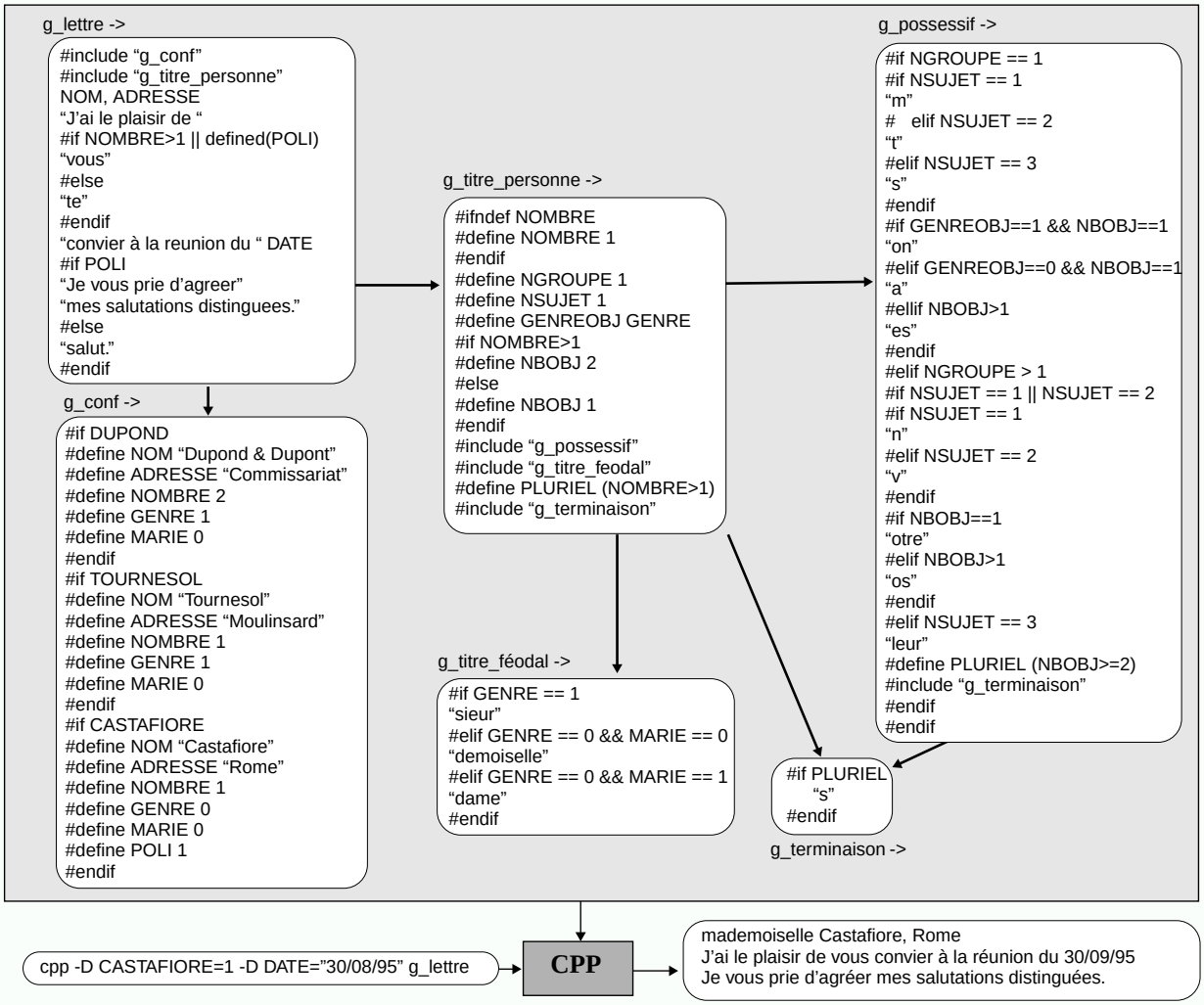
Si l'on compare cette représentation à celles présentées dans le Chapitre II, il apparaît clairement que ces dernières, issues du modèle fonctionnel, sont beaucoup plus explicites et beaucoup plus faciles à comprendre. Dans la représentation CPP on retrouve les mêmes concepts mais il faut bien les chercher ! Ceux-ci sont dilués dans le code et n'apparaissent que sous la forme de compositions de mécanismes de bas niveaux.

### IV.3.8 Conclusion

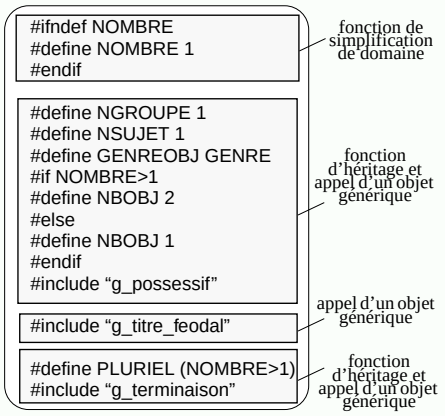
Dans cette section nous avons vu que le modèle abstrait pouvait être appliqué au cas de CPP. En tout cas il permet de mieux comprendre les concepts sous-jacents. L'utilisation d'abstractions a permis de montrer que des mécanismes comme l'inclusion textuelle et la macro-substitution étaient en fait beaucoup plus proches qu'ils ne pouvaient le sembler au premier abord. Interpréter CPP en fonction du modèle abstrait permet aussi de le comparer à d'autres systèmes. En fait si l'objectif poursuivi dans cette thèse était uniquement de contribuer à une rationalisation de la programmation globale (objectif O3), dans une certaine mesure on pourrait s'estimer satisfait.

1. L'objet générique construit **G_LETTRE** n'avait pas été explicité dans le . Il est basé sur g_titre_personne qui lui, a été longuement décrit dans la Section II.7.1. La figure 94 doit être rapprochée à la figure 49 (p.110) et à figure 58 (p.126).

*figure 111* Représentation de l'objet générique g_lettre en CPP




Le cadre supérieur correspond à la portion du système de fichiers dans lequel sont représentés les objets génériques. Remarquons tout d'abord que comprendre une telle représentation n'est pas une mince affaire ! Et pourtant le système est simple. Le chargé de maintenance n'a généralement à sa disposition qu'un éditeur de texte et des outils de recherche textuelle comme 'grep'. Lorsqu'il entame la lecture d'un fichier la gêne visuelle introduite par les caractères # et le manque d'indentation se fait tout de suite ressentir. Si l'on fait abstraction de ces problèmes, la difficulté suivante est d'obtenir une vision globale du système. Il ne voit que six fichiers et comprendre un morceau de code implique un va et vient quasi-continu pour savoir où sont définies les macros et où elles sont utilisées. Le fait que l'ordre d'exécution soit important dans le modèle impératif rend la tâche encore plus complexe : non seulement il faut comprendre la fonction des fichiers mais en plus il faut savoir dans quels ordre ils interviennent. Exécuter le système pour des exemples particuliers peut aider à le comprendre. Encore faut il savoir quels sont les paramètres de g_lettre...




En l'absence de modèle abstrait le chargé de maintenance peut comprendre et modifier cet ensemble de fichiers mais de manière intuitive. Il lui est probablement difficile d'expliquer conceptuellement son fonctionnement. Le fichier présenté ci-contre peut être vu comme une suite de définitions de macros, d'instructions conditionnelles ou d'inclusions textuelles..

Au contraire si l'on utilise le modèle abstrait on peut au contraire reconnaître l'occurrence de différents concepts dans le code. Par exemple le premier bloc correspond à la définition d'une valeur par défaut pour le paramètre NOMBRE (*une fonction de simplification*). Le deuxième bloc correspond à l'appel de l'objet générique "g_possessif" avec les paramètres NGROUPE, NSUJET, GENREOBJ et NBOBJ (*des fonctions d'héritages*). De même on peut reconnaître en g_conf une *fonction d'abstraction de domaine.*

# IV.4 APP, un préprocesseur abstrait

Face à la foule de problèmes pragmatiques posés par l'utilisation des préprocesseurs, l'un des objectifs spécifiques de cette thèse est de fournir une aide concrète au chargé de maintenance (objectif O6[1]).

*Dans un milieu industriel, on chercherait probablement à construire un outil ad-hoc proposant des solutions spécifiques. Ici au contraire, une démarche rigoureuse et générale est suivie.*

L'objectif de cette section est de décrire formellement la sémantique du préprocesseur CPP.

Cette idée peut sembler surprenante lorsque l'on sait que cet outil a été défini il y a plus de 20 ans de manière empirique et parfois arbitraire. Il est clair qu'une telle démarche devrait se faire a priori et non pas a posteriori.

Décrire la sémantique de CPP n'est pourtant pas seulement un exercice de style. Une telle définition est essentielle si l'on s'intéresse à des outils de maintenance et de ré-ingénierie basés sur des transformations automatiques de programmes.

Rappelons aussi que cette thèse a pour but de montrer que *les techniques de programmation détaillée peuvent être utiles dans le cadre de la programmation globale*. Dans la Section IV.4.2 nous donnons la sémantique dénotationnelle des objets génériques basés sur CPP, alors pourquoi ne pas essayer de le faire pour d'autres systèmes de variation plus récents ?

#### *APP, un langage abstrait*

Lorsque l'on considère un préprocesseur comme CPP, tout de suite une masse très importante de détails lexicaux et syntaxiques obscurcissent la sémantique sous-jacente. Cela est dû à une *mauvaise définition du langage* où syntaxe et sémantique n'ont pas été clairement définies, ni même distinguées.

Pour faire face à ce problème nous allons définir un ***préprocesseur abstrait*** appelé ***APP*** (Abstract PreProcessor)[2]. APP est un langage abstrait ; abstrait dans le sens ou il n'a pas de syntaxe concrète.

Nous seulement APP permet de faire abstraction des détails lexicaux et syntaxiques, mais il se distingue de CPP par le fait qu'il utilise des concepts issus des langages de programmation. Par exemple APP utilise les notions d'instruction d'affectation et d'appel de procédures à la place des définitions de macros et des inclusions de fichiers.

Au premier abord, cette abstraction peut paraître déconcertante pour l'utilisateur pragmatique de CPP. Par contre, elle présente l'énorme avantage de se placer dans un contexte plus "standard". Il

1. O6 Faciliter la maintenance et la ré-ingénierie de programmes utilisant des préprocesseurs.
2. Récemment cet acronyme a également été utilisé pour désigner une extension du préprocesseur CPP proposée par ATT [Rose95]. Dans ce cas APP signifie "Annotated PreProcessor". Cet outil permet d'insérer des assertions dans du code C ou C++.

est alors possible de se référer à des concepts connus et d'utiliser des résultats déjà établis (analyse statique inter-procédurale, distinction entre liaison dynamique et liaison statique, etc.).

Cette démarche peut être vue comme une tentative de sortir un outil comme CPP de son ghetto : un outil trop spécifique et obsolète du point de vue de la recherche est souvent considéré avec dédain. En soit ce n'est vraiment pas un problème ; le problème vient du fait que de tels outils sont encore largement utilisés dans l'industrie ! Il ne s'agit pas d'essayer de "donner un air respectable" à CPP, mais plutôt de pouvoir bénéficier des efforts menés dans d'autres contextes de recherche plus actifs.

Finalement, bien qu'APP corresponde essentiellement à une abstraction de CPP, il est possible d'y intégrer des caractéristiques provenant d'autres préprocesseurs pour les étudier et les évaluer[1].

Dans cette section nous présentons tout d'abord le schéma d'abstraction permettant de passer de CPP à APP (Section IV.4.1), ensuite la sémantique dénotationnelle du langage APP est définie (Section IV.4.2) ; finalement la Section IV.4.3 dégage les caractéristiques essentielles de ce langage.

## IV.4.1 De CPP à APP

La première étape dans un processus de ré-ingénierie est de passer d'une représentation concrète à un représentation plus abstraite. Ici il s'agit de passer d'un fichier CPP à un programme APP.

Le schéma d'abstraction est défini informellement. Il s'agit surtout de donner une vision intuitive de cette transformation (les exemples de la figure 112 peuvent être consultés au fur et à mesure de la description). Notons finalement qu'en tant que langage abstrait, APP n'a pas de syntaxe concrète, mais uniquement une syntaxe abstraite (définie formellement plus loin). Ci-dessous certains détails syntaxiques apparaissent mais ce n'est que pour faciliter la compréhension du langage.

### IV.4.1.1 *Abstraction d'une directive CPP vers une instruction APP*

Les directives CPP correspondent aux ***instructions APP*** (tableau 17). Une liste de directives correspond bien évidemment à une ***séquence d'instructions***. Les macros définitions correspondent aux ***instructions d'affectations***. La compilation conditionnelle est bien évidemment une ***instruction conditionnelle***. Un fragment de texte par contre se traduit par une ***instruction de sortie***. L'inclusion textuelle est décrite dans la section suivante.

1. Il est par exemple aisé d'ajouter des instructions d'itérations (De telles constructions sont présentes dans CTF alors que CPP se limite aux instructions conditionnelles).

TABLEAU 16 Abstraction d'une directive CPP vers une directive APP

| terme concret | CPP | APP | terme abstrait |
|---|---|---|---|
| directive cpp | <CppDir.> ::= <SequenceDir> \| <Conditionals> \| <TextFragment> \| <MacroDef> \| <MacroUndef> \| <TextIncl.> | <Stmt> ::= <SeqStmt> \| <IfStmt> \| <OutStmt> \| <AssignStmt> \| <CallStmt> | instruction app |
| séquence de directives | <SequenceDir.>::= <CppDir.> <CppDir.> | <SeqStmt> ::= <Stmt> **;** <Stmt> | séquence d'instructions |
| compilation conditionnelle | <Conditionals> ::= **#if** <CondTokenList>$_1$ <CppDir.>$_1$ **#elsif** <CondTokenlList>$_2$ <CppDir.>$_2$ **#elsif** ... **...** **#else** <CppDir.>$_n$ **#endif** | <IfStmt> ::= **IF** <CondTerm>$_1$ **THEN** <Stmt>$_1$ **ELSE IF** <CondTerm>$_2$ **THEN** <Stmt>$_2$ **ELSE ...** **...** **ELSE** <Stmt>$_n$ | instruction conditionnelle |
| condition | <CondTokenList> ::= <TokenList> | <CondTerm> ::= <Term> | terme-condition |
| fragment de texte | <TextFragment> ::= <TokenList> | <OutStmt> ::= **OUT** <Term> | instruction de sortie |
| liste de lexèmes | <TokenList> ::= [ <Token> ] | <Term> ::= [ <Factor> ] | terme |
| lexème | <Token>::=<MacroNameOcc> \| <StdToken> | <Factor> ::= <Const> \| <VariOcc> | facteur |
| lexème standard | <StdToken> | <Const> | constante |
| occurence de macro | <MacroNameOcc> ::= <MacroName> | <VarOcc> ::= **$**<Var> | occurence de variable |
| nom de macro | <MacroName> | <Var> | variable |
| définition de macro | <MacroDef> ::= **#define** <MacroName> <MacroVal> | <AssignStmt> ::= <Var> **:= "**<Value>**"** | affectation d'une variable |
| valeur d'une macro | <MacroVal>::=<TokenList> \| <Param.Macro> | <Value> ::= <Term> \| <Function> | valeur |
| macro paramétrée | <Param.Macro>::= **(**[<Var>]**)** <TokenList> | <Function> ::= **FUN** [<Var>].<Term> | fonction |
| élimination | <MacroUndef> ::= **#undef** <MacroName> | <AssignStmt> ::= **RESET** <Var> | affectation à nil |

### IV.4.1.2 *Abstraction d'un fichier CPP vers une procédure APP*

L'organisation de programmes CPP en plusieurs fichiers est une technique de structuration. Dans APP, un fichier correspond à une ***procédure APP*** (tableau 17). Une directive d'inclusion textuelle est alors un ***appel de procédure***. Intuitivement, signalons que cette analogie a été choisie car l'inclusion d'un fichier peut provoquer des effets de bords si celui-ci contient des définitions de macros (des affectations).

TABLEAU 17 Abstraction d'un fichier CPP vers une procédure APP

| terme concret | CPP | APP | terme abstrait |
|---|---|---|---|
| fichier cpp | <CppFile> ::= <CppFileName> <CppFileContent> | <Proc> ::= **PROCEDURE** <ProcId> **IS** **BEGIN** <Procbody> **END** | procédure app |
| nom de fichier | <CppFileName> | <ProcName> | nom de procédure |
| contenu d'un fichier | <Filecontent> ::= <CppDir.> | <ProcBody> ::= <Stmt> | corps d'une procédure |
| inclusion textuelle | <TextIncl.> ::= **#include "**<CppFilName>**"** | <CallStmt> ::= **CALL** <ProcName> | appel de procédure |

### IV.4.1.3 *Abstraction d'un répertoire vers un module APP*

A un niveau de granularité plus forte, l'organisation des fichiers CPP en répertoires est aussi une technique de structuration. De telles représentations se traduisent naturellement en une structure de *modules APP*[1] (tableau 19).

TABLEAU 18 Abstraction d'un répertoire vers un programme APP

| terme concret | CPP | APP | terme abstrait |
|---|---|---|---|
| nom de répertoire | <DirectoryName> | <ModName> | nom d'un module |
| répertoire | <Directory> ::= <DirectoryName> { <CppFile> } | <Module> ::= **MODULE** <ModName> **IS** {<Proc>} **END** | module |

### IV.4.1.4 *Abstraction d'une ligne de commande à un programme APP*

Si l'on veut modéliser le comportement de CPP, il faut aussi prendre en compte la ligne de commande. En fait l'interprétation du fichier CPP principal est précédé de deux étapes : (1) tout d'abord une phase d'initialisation réalise les prédéfinitions ; (2) ensuite les valeurs des paramètres indiqués sur la ligne de commande sont associées aux macros correspondantes.
Une ligne de commande CPP correspond à un *programme principal APP*[2] (tableau 19). L'interprétation de celle-ci correspond à l'exécution de ce programme. Elle dépend du répertoire dans lequel elle est exécutée. De la même manière nous verrons que la sémantique d'un programme dépend du module associé (à cause des procédures qu'il appelle et qui sont dans ce module)[3].

TABLEAU 19 Abstraction d'une ligne de commande CPP vers un programme APP

| terme concret | CPP | APP | terme abstrait |
|---|---|---|---|
| ligne de commande | <CommandLine> ::= **cpp** [ <DefOpt> \| <UndefOption> ] <FileName> | <Program> ::= **PROGRAM** **BEGIN** **CALL** init **;** [ <AssignStmt> ] **;** **CALL** <ProcName> **END** | programme principal |
| définition de macro | <DefOpt> ::= **-D**<MacroName>**=**<TokenList> | <AssignStmt> ::= <Var> **:=** "<Value>" | affectation |
| élimination de macro | <UndefOpt>::= **-U**<MacroName> | <AssignStmt> ::= **RESET** <Var> | élimination |

### IV.4.1.5 *Exemples*

Le schéma d'abstraction de CPP vers APP est illustré par la figure 112. Elle présente différents extraits de la lettre générique.
Insistons sur le fait qu'il ne faut pas se focaliser sur le changement de syntaxe : la syntaxe utilisée pour écrire les programmes APP n'a strictement aucune importance ; elle n'est là que pour faciliter la compréhension. Les extraits présentés dans la figure 112 correspondent en fait à des arbres abstraits.

1. Comme dans des langages modulaires comme Modula 2, cette structure est hiérarchique (i.e. un répertoire est un ensemble de fichiers et de répertoires ; un module est un ensemble de ressources et de modules). Dans la suite nous nous limiterons généralement à une structure plate, mais c'est sans perte de généralité et uniquement pour simplifier l'exposé.
2. Par la suite le terme "programme" sera utilisé plutôt que "programme principal", ce qui rend ambigu l'utilisation de ce terme mais qui ne devrait pas poser de problèmes particuliers.
3. Le schéma d'abstraction ci-dessus ne prend pas en compte les chemins de recherche (l'option -I de la ligne de commande).

*figure 112* Extraits de la lettre générique en langage APP

```
PROCEDURE g_lettre IS
   BEGIN
      CALL g_conf ;
      CALL g_possessif;
      OUT $NOM;
      OUT $ADRESSE ;
      OUT "J'ai le plaisir de";
      IF defined $NOMBRE THEN
         OUT "vous
      ELSE
         OUT "te" ;
      OUT "convier a la reunion du " $DATE;
      IF defined $POLI THEN
         OUT "Je vous prie d'agreer";
         OUT "mes salutations distinguees."
      ELSE
         OUT "salut."
   END
```

```
PROGRAM IS
   BEGIN
      CALL init ;
      CASTAFIORE := "1" ;
      DATE := "30/09/95" ;
      CALL g_lettre
   END
```

```
PROCEDURE g_titre-personne IS
   BEGIN
      IF ! defined $NOMBRE THEN
         NOMBRE := "1" ;
      ELSE
         NOP ;
      NGROUPE := "1" ;
      NSUJET := "1" ;
      GENREOBJ := "$GENRE" ;
      ...
   END
```

```
MODULE m_lettre IS

   PROCEDURE g_lettre IS
      BEGIN
         CALL g_conf ;
         CALL m_titre.g_possessif ;
         ...
      END
   PROCEDURE g_conf IS
      ...

   MODULE m_titre IS
      PROCEDURE g_titre_personne IS
         ...
      PROCEDURE g_possessif IS
         ...
      PROCEDURE g_terminaison IS
         ...
   END

   PROCEDURE init IS
         ...
END
```

### IV.4.1.6 *Conclusion*

Concrètement la transformation présentée correspond à une phase d'analyse lexicale et syntaxique suivie d'une phase d'abstraction syntaxique[1].

Le principal attrait de ce processus d'abstraction est qu'il permet de se détacher de la terminologie spécifique à CPP pour raisonner en utilisant des notions de programmation détaillée connues.

## IV.4.2 Sémantique dénotationnelle du langage APP

Il s'agit maintenant de définir formellement le langage APP[2]. Sa sémantique est la même que celle de CPP mais pour alléger le discours, il n'est fait référence qu'aux concepts abstraits. Du coup, pour bénéficier pleinement de la lecture de cette section, il est important d'avoir en tête la correspondance entre CPP et APP[3].

1. Cette transformation n'est pas toujours aussi simple qu'il ne le parait car CPP n'est pas un langage hors-contexte. C'est d'ailleurs pour cela que dans l'arbre abstrait APP les conditions sont représentés comme des termes (une liste de facteurs) plutôt que par des arbres abstraits d'un langage d'expressions. Ceci dit presque toutes les difficultés peuvent être levées. Un prototype automatisant cette transformation a été réalisé (Section IV.6.2). Certains problèmes subsistent, mais ils correspondent à des cas très rares en pratique. Par exemple dans le fragment de code suivant "#define b 2" ne doit être reconnu comme une définition de macro qui si p n'est pas une macro paramétrée ; si c'est le cas "#define b 2" joue le rôle du paramètre. L'analyse d'un tel fragment de code n'est donc pas hors-contexte.

```
p(
#define b 2
)
```

2. L'utilisation du terme "formel" peut être contesté car certaines libertés ont été prises dans la manière de définir la sémantique du langage. La définition que l'on donne semble néanmoins suffisante pour l'usage qui en est fait dans cette thèse.
3. En réalité nous considérons ici une restriction de CPP : la sémantique décrite ne prend pas en compte les macros paramétrées, ni les modules imbriqués, ni les chemins de recherche. Il n'est pas très difficile d'intégrer ces constructions, mais leur prise en compte obscurcit la sémantique.

### IV.4.2.1 *Syntaxe abstraite des procédures APP*

La syntaxe abstraite des procédures APP est décrite dans la figure 113. Des mots clés et des signes de ponctuation ont été introduits pour faciliter la compréhension des programmes APP et donner une idée intuitive de leur sémantique. Par exemple, un **$** précède les occurrences de variables dans les termes pour suggérer le fait que celles-ci vont être substituées par leur valeur.

*figure 113* Syntaxe abstraite des procédures APP

***(b)* Domaines syntaxiques basiques**

| | | |
|---|---|---|
| pn | ∈ | ProcName |
| mn | ∈ | ModName |
| c | ∈ | Const |
| x | ∈ | Var |

***(c)* Domaines syntaxiques construits**

| | | |
|---|---|---|
| p | ∈ | Proc |
| s | ∈ | Stmt |
| ct | ∈ | CondTerm |
| t | ∈ | Term |
| f | ∈ | Factor |
| xo | ∈ | VarOcc |
| val | ∈ | Value |
| func | ∈ | Function |

***(d)* Syntaxe abstraite**

```
p    ::= PROCEDURE pn IS BEGIN s END
s    ::= NOP
       | x := "val"
       | RESET x
       | IF ct THEN s ELSE s
       | CALL pn
       | OUT t
       | s ; s
ct   ::= t
t    ::= [ f ]
f    ::= c | xo
xo   ::= $ x
val  ::= t | func
func ::= FUN [x] -> t
cm   ::= STD | LIB
```

### IV.4.2.2 *Domaines sémantiques et fonctions sémantiques*

Les principaux domaines et fonctions sémantiques sont présentés dans la figure 114.

*figure 114* Domaines et fonctions sémantiques pour les procédures APP

***(a)* Domaines sémantiques basiques**

| | | |
|---|---|---|
| φ | ∈ | *Fragment* |
| η | ∈ | ℵ |

***(b)* Domaines sémantiques construits**

| | | |
|---|---|---|
| ρ | ∈ | *Mem* = Var -> Value |
| π | ∈ | *ProcEnv* = ProcName -> Proc |
| ϕ | ∈ | *Fragments* = [ *Fragment* ] |

***(c)* Fonctions sémantiques**

| | | |
|---|---|---|
| *SemProc* | ∈ | Proc -> *ProcEnv* -> *Mem* -> (*Fragments* × *Mem*) |
| *SemStmt* | ∈ | Stmt -> *ProcEnv* -> *Mem* -> (*Fragments* × *Mem*) |
| *SemTerm* | ∈ | Term -> *Mem* -> *Fragments* |
| *SemCondTerm* | ∈ | CondTerm -> *Mem* -> ℵ |

Un programme APP génère une structure sémantique *Fragments* à partir d'un ensemble d'éléments du domaine sémantique *Fragment*. L'ensemble des entiers ℵ est utilisé pour l'évaluation des conditions car CPP hérite du langage C l'absence de booléens. Tous les opérateurs (y compris relationnels) opèrent sur des entiers.

APP est un langage impératif et les différentes instructions APP sont susceptibles de modifier une mémoire (domaine sémantique *Mem*). Cette mémoire associe à chaque variable une valeur (ici des termes ou des fonctions). Le domaine sémantique *ProcEnv* correspond à l'environnement de procédures. Lors de l'appel d'une procédure il permet de déterminer la procédure à appeler[1].

### IV.4.2.3 *Sémantique des procédures et des instructions*

La sémantique des procédures *SemProc* est directement liée à la sémantique des instructions *SemStmt* (figure 115).

*figure 115* Définition de *SemProc* et de *SemStmt*.

$$\begin{aligned}
&\mathit{SemProc} \in \quad \text{Proc} \to \mathit{ProcEnv} \to \mathit{Mem} \to (\mathit{Fragments} \times \mathit{Mem})\\
&\mathit{SemStmt} \in \quad \text{Stmt} \to \mathit{ProcEnv} \to \mathit{Mem} \to (\mathit{Fragments} \times \mathit{Mem})
\end{aligned}$$

$$\mathit{SemProc}\ [\![\ \texttt{PROCEDURE}\ \text{pn}\ \texttt{IS}\ \texttt{BEGIN}\ \text{s}\ \texttt{END}\ ]\!]\ \pi\ \rho = \mathit{SemStmt}\ [\![\ \text{s}\ ]\!]\ \pi\ \rho$$

$$\begin{aligned}
\mathit{SemStmt}\ [\![\ \texttt{NOP}\ ]\!]\ \pi\ \rho &= (\ \mathit{EmptyFragments}\ ,\ \rho\ )\\
\mathit{SemStmt}\ [\![\ \text{x}\ \texttt{:=}\ \texttt{“t”}\ ]\!]\ \pi\ \rho &= (\ \mathit{EmptyFragments}\ ,\ (\ \rho \oplus \{\ \text{x} \to \text{t}\ \})\ )\\
\mathit{SemStmt}\ [\![\ \texttt{RESET}\ \text{x}\ ]\!]\ \pi\ \rho &= (\ \mathit{EmptyFragments}\ ,\ (\{\text{x}\} \triangleleft \rho\ )\ )\\
\mathit{SemStmt}\ [\![\ \texttt{OUT}\ \text{t}\ ]\!]\ \pi\ \rho &= ((\mathit{SemTerm}\ [\![\ \text{t}\ ]\!]\ \rho), \rho)\\
\mathit{SemStmt}\ [\![\ \texttt{CALL}\ \text{pn}\ ]\!]\ \pi\ \rho &= \mathit{SemProc}\ [\![\ \pi\ \text{pn}\ ]\!]\ \pi\ \rho
\end{aligned}$$

$$\begin{aligned}
&\mathit{SemStmt}\ [\![\ \texttt{IF}\ \text{ct}\ \texttt{THEN}\ \text{s}_1\ \texttt{ELSE}\ \text{s}_2\ ]\!]\ \pi\ \rho =\\
&\qquad \mathit{SemStmt}\ [\![\ \textbf{if}\ (\mathit{SemCondTerm}\ [\![\ \text{ct}\ ]\!]\ \rho) \neq 0\ \textbf{then}\ \text{s1}\ \textbf{else}\ \text{s2}\ ]\!]\ \pi\ \rho
\end{aligned}$$

$$\begin{aligned}
\mathit{SemStmt}\ [\![\ \text{s}_1\ \texttt{;}\ \text{s}_2\ ]\!]\ \pi\ \rho = \quad & \textbf{let}\ \ (\phi_1,\rho_1) = \mathit{SemStmt}\ [\![\ \text{s}_1\ ]\!]\ \pi\ \rho\ \textbf{in}\\
& \textbf{let}\ (\phi_2,\rho_2) = \mathit{SemStmt}\ [\![\ \text{s}_2\ ]\!]\ \pi\ \rho_1\ \textbf{in}\\
& (\ (\mathit{FragmentsSyntesis}\ \phi_1\ \phi_2)\ ,\ \rho_2\ )
\end{aligned}$$

La dénotation d'une instruction est une fonction Stmt -> *ProcEnv* ->*Mem* -> (*Fragments* × *Mem*). Autrement dit le comportement d'une instruction APP est déterminée par l'environnement de procédures π et la mémoire ρ.

- L'instruction **NOP** n'a pas d'effet sur la mémoire. Elle génère une structure vide *EmptyFragments* (ici une liste vide de *Fragment*).
- L'affectation x **:=** **“**t**”** ne produit aucun *Fragments*. Elle range le terme t dans la mémoire ρ. L'ancienne valeur, s'il elle existait est perdue. Notons que terme t n'est pas évalué. La mémoire contient donc des valeurs “symboliques”. Il s'agit en fait ici d'une liaison dynamique et non pas statique comme dans la plupart des langages. Cet aspect est décrit plus longuement dans la Section IV.4.3.
- L'instruction **RESET** x détruit de la mémoire l'association correspondant à la variable x si celle-ci existait (rappelons que la mémoire est une fonction partielle).
- L'instruction de sortie **OUT** t produit un *Fragments* en évaluant le terme t dans la mémoire ρ. Celle-ci est inchangée. La sémantique d'un terme, *SemTerm*, est décrite dans la section suivante.
- L'appel de procédure consiste tout simplement à exécuter le corps de la procédure π(pn).
- L'instruction conditionnelle **IF** ct **THEN** $s_1$ **ELSE** $s_2$ exécute soit $s_1$, soit $s_2$, et ce en fonction de l'évaluation du terme-condition ct dans la mémoire courante ρ (voir la sémantique des termes conditions (*SemCondTerm*) dans la Section IV.4.2.5).
- $s_1$ **;** $s_2$ est la composition séquentielle des instructions $s_1$ et $s_2$. Elle consiste à faire la *synthèse*

1. Ici pour simplifier nous ne prenons pas en compte les chemins de recherche.

des *Fragments* générés par chaque instruction. Dans le cas d'APP la fonction de synthèse *FragmentsSyntesis* est la concaténation de listes (@) puisque un *Fragments* est une liste de *Fragment*. C'est pour cela que l'instruction **OUT** peut être vue comme une instruction de sortie séquentielle : chaque instruction concatène un nouveau *Fragments* au *Fragments* généré depuis le début du programme.
Remarquons finalement l'exécution de $s_2$ *hérite* de $\rho_1$, la mémoire résultant de l'application de $s_1$ à la mémoire initiale $\rho$ (ce qui est naturel pour un langage impératif).

### IV.4.2.4 *Sémantique des termes*

Jusqu'à maintenant nous n'avons pas décrit la substitution, ni donné *SemTerm* la sémantique des termes. Ces constructions syntaxiques apparaissent dans l'instruction de sortie **OUT**. Leur sémantique est définie dans la figure 116.

*figure 116* Sémantique des termes et des facteurs

```
SemTerm ∈       Term -> Mem -> Fragments
SemFactor ∈     Factor -> Mem -> Fragments
SemConst ∈      Const -> Fragment
SemUndefinedVarOcc∈ VarOcc -> Fragment

SemTerm [[ [] ]] ρ          = EmptyFragments

SemTerm [[ f :: t ]] ρ      = FragmentsSynthesis (SemFactor[[ f ]]) (SemTerm [[ t ]] ρ)

SemFactor[[ c ]] ρ          = [ SemConst[[ c ]] ]

SemFactor[[ $x ]] ρ         = ( if x ∈ dom( ρ )
                                then SemTerm [[ (ρ x) ]] ({x} ⩤ ρ)
                                else [ SemUndefinedVarOcc[[ x ]] ] )
```

Un terme est une structure (ici une liste) à base de constantes et d'occurrences de variables. Sa dénotation est synthétisée à partir de la dénotation de chaque composante via la fonction *FragmentsSynthesis* (ici la concaténation de listes @).
Une constante dénote toujours le même *Fragment* (La fonction *SemConst* est une donnée du problème. Dans le cas de CPP un lexème du fichier source est recopié tel quel dans le fichier de sortie). Par contre une occurence de variable est substituée par la valeur associée à cette variable. Si cette variable n'est pas définie sa dénotation est égale au fragment donné par la fonction *SemUndefinedVarOcc* (dans le cas de CPP le nom de la macro est recopié tel quel).
La dénotation d'un terme, tout comme celle d'une occurence de variable, est une fonction de la forme *Mem -> Fragments*.

### IV.4.2.5 *Sémantique des termes-conditions*

Un terme-condition apparaît dans une instruction conditionnelle. Son évaluation se fait en trois étapes et fait intervenir un autre langage. (1) Le terme condition est "étendu", autrement dit les variables sont substituées par leur valeurs (fonction $\text{Expand}_{\text{CondTerm}}$). (2) Le résultat ainsi obtenu est considéré comme une suite de lexèmes; après une analyse syntaxique et une abstraction (fonction $\text{ParseAndAbstract}_{\text{Cond}}$), on obtient une expression d'un langage que l'on appellera $L_{\text{cond}}$. (3) Finalement cette expression est évaluée et retourne la valeur correspondant à la condition.

### *Expansion d'un terme-condition en une condition*

Un terme-condition est syntaxiquement identique à un terme. Pourtant, même si l'on considère uniquement leur expansion, ils ne sont pas équivalents. Les règles de substitution des occurrences de variables ne sont pas exactement les mêmes (figure 117).

- La construction ConstDefined **$**x permet de "tester" si la variable x est définie.
- Une occurence de variable qui n'est pas définie est substituée par 0 ! Cette décision est arbitraire et correspond à l'une des incohérences de CPP. Le fait d'avoir une valeur par défaut dans un terme est explicable, pas dans une condition.

*figure 117* Sémantique des termes-conditions

$$\mathit{SemCondTerm} \in \text{CondTerm} \to \mathit{Mem} \to \aleph$$

$$\mathit{SemCondTerm}\ [\![\ \text{ct}\ ]\!]\ \rho = \mathit{SemCond}\ \text{o ParseAndAbstract}_{\text{Cond}}\ \text{o Expand}_{\text{CondTerm}}$$

$$\text{Expand}_{\text{CondTerm}} \in \text{CondTerm} \to \mathit{Mem} \to [\text{Const}]$$

$\text{Expand}_{\text{CondTerm}}(\ [\ ]\ )\ \rho$ = []

$\text{Expand}_{\text{CondTerm}}(\ \text{ConstDefined} :: c :: t\ )\ \rho$ = *erreur*

$\text{Expand}_{\text{CondTerm}}(\ \text{ConstDefined} :: \$x :: t\ )\ \rho$ = **(if** $x \in$ **dom(** $\rho$ **)**
**then** [ConstantOne] **else**[ConstantZero])
@ $\text{Expand}_{\text{CondTerm}}$ (t) $\rho$

$\text{Expand}_{\text{CondTerm}}(\ \$x :: t\ )\ \rho$ = **if** $x \in$ **dom(** $\rho$ **)**
**then** $\text{Expand}_{\text{CondTerm}}$ ( t ) ({x} $\lhd$ $\rho$)
**else** [ConstantZero]

$\text{Expand}_{\text{CondTerm}}(\ c::t\ )\ \rho$ = [c] @ $\text{Expand}_{\text{CondTerm}}$ (t) $\rho$

### *Evaluation d'une condition (Expression du langage $L_{cond}$)*

Ci-dessous la syntaxe abstraite du langage $L_{cond}$ est définie. Il s'agit d'un langage d'expressions constantes (figure 118).

*figure 118* Syntaxe abstraite du langage des conditions $L_{cond}$

**(a) Domaines syntaxiques basiques**

| | | |
|---|---|---|
| n | ∈ | Number |
| uop | ∈ | UnOp {**NOT**, **-**, ...} |
| bop | ∈ | BinOp = {**=**,**<**,**>**,**+**,**-**,*****,...} |
| top | ∈ | TernOp = { **IF** } |

**(b) Domaines syntaxiques construits**

| | | |
|---|---|---|
| c | ∈ | Cond |

**(c) Syntaxe abstraite**

| | | |
|---|---|---|
| c | ::= | n |
| | \| | uop c |
| | \| | c bop c |
| | \| | top c c c |

La sémantique de $L_{cond}$ hérite du langage C : $\aleph$ est le seul domaine sémantique et toutes les opérations y compris les opérations relationnelles (<,>,=,etc.) sont définies sur ce domaine. Par convention 0 correspond à faux (cf la sémantique de l'instruction conditionnelle figure 115 (p.238)).

*figure 119* Domaines et fonctions sémantiques pour $L_{cond}$

**(a) Domaines sémantiques basiques**

$\eta \in \aleph$
$\omega_1 \in Op1 = \aleph \rightarrow \aleph$
$\omega_2 \in Op2 = \aleph \rightarrow \aleph \rightarrow \aleph$
$\omega_3 \in Op3 = \aleph \rightarrow \aleph \rightarrow \aleph \rightarrow \aleph$

**(b) Fonctions sémantiques**

*SemCond* ∈ Cond -> ℵ
*SemNumber* ∈ Number -> ℵ
*SemUnOp* ∈ UnOp -> *Op1*
*SemBinOp* ∈ BinOp -> *Op2*
*SemTernOp* ∈ TernOp -> *Op3*

**(c) Définition de** *SemExpr*

*SemExpr [[* n *]]* = *SemNumber [[* n *]]*
*SemExpr [[* uop e *]]* = *SemUnOp [[* uop *]]* e
*SemExpr [[* $e_1$ bop $e_2$ *]]* = *SemBinOp [[* bop *]]* (*SemExpr [[*$e_1$ *]]*) (*SemExpr [[*$e_2$ *]]*)
*SemExpr [[* top $e_1$ $e_2$ $e_3$ *]]* = *SemBinOp [[* top *]]* (*SemExpr [[*$e_1$ *]]*) (*SemExpr [[*$e_2$ *]]*)(*SemExpr [[*$e_3$ *]]*)

### IV.4.2.6 *Syntaxe abstraite et sémantique des modules*

Nous n'avons pas encore décrit comment était obtenu l'environnement de procédures. En fait il s'agit tout simplement de la dénotation d'un module. Autrement dit un module peut être vu comme une fonction retournant la procédure correspondant à un nom de procédure donné.

*figure 120* Syntaxe abstraite et sémantique d'un module

**(a) Domaine syntaxique basique**

mn ∈ ModName

**(b) Domaine syntaxique construit**

mod ∈ Module

**(c) Syntaxe abstraite**

mod ::= `MODULE` mn `IS` [p] `END`

**(d) Fonction sémantique**

*SemMod* ∈ Mod -> *ProcEnv*

### IV.4.2.7 *Syntaxe abstraite et sémantique des programmes*

La sémantique des programmes APP est simple. C'est la même que celle de l'instruction qui constitue son corps, mis à part que la mémoire initiale est vide et que le résultat d'un programme est un *Fragments* (autrement dit la mémoire à la fin de l'exécution du programme n'a plus d'intérêt)[1].

*figure 121* Syntaxe abstraite du langage des programmes APP

**(a) Domaine syntaxique construit**

prg ∈ Program

**(b) Syntaxe abstraite**

prg ::= `PROGRAM IS BEGIN` s `END`

**(c) Fonction sémantique**

*SemProg* ∈ Program -> ProcEnv -> *Fragments*

*SemProg[[* `PROGRAM IS BEGIN` s `END` *]]* π = ϕ
**where** (ϕ,ρ) = *SemStmt [[* s *]]* π {}

1. La sémantique proposée ici pour les programmes est générale. En pratique, le schéma d'abstraction de CPP vers APP, ne produit que des programmes d'une forme particulière. La procédure "init" appelée systématiquement permet de modéliser l'effet des prédéfinitions.

### IV.4.3 Caractéristiques marquantes du langage APP

La sémantique du langage APP a été présentée de manière descriptive, sans beaucoup la commenter. Cette section au contraire dégage les caractéristiques remarquables de ce langage impératif.

- **Liaison dynamique**.
  L'affectation APP correspond à une ***liaison dynamique*** plutôt qu'à une ***liaison statique***[1]. Ceci permet d'expliquer le problème PC3 en s'appuyant sur des notions connues.
  Concrètement, l'utilisation d'une liaison dynamique dans APP se traduit par le fait que les variables libres d'un terme ne sont pas liées lors de son affectation à une variable mais plutôt lors de l'utilisation de cette variable. Par exemple la séquence `A:="1";B:="$A$A";A:="2";OUT $B` affiche *2 2* puisque la variable libre `A` du terme affecté à `B` n'est pas évalué lors de l'affectation, mais lors de l'utilisation. La mémoire contient donc des valeurs symboliques (les "" utilisés dans l'instruction d'affectation sont justement là pour rendre plus explicite cette propriété).
  Aujourd'hui la *liaison statique* est retenue par presque tous les langages de programmation, sauf certains dialectes Lisp. Sa sémantique est naturelle.
  La *liaison dynamique* est moins satisfaisante. L'utilisation de ce type de liaison est susceptible d'introduire confusions et erreurs (c'est d'ailleurs le cas dans CPP (PC3)). Nous allons voir également que la liaison dynamique pose des problèmes lors de l'analyse de flots de données (voir la Section IV.5.2)[2].
  Remarquons finalement que l'ordre des affectations symboliques n'a pas d'importance, pourvu évidemment qu'il n'y ait pas d'utilisations entre les différentes instructions. C'est ce qui explique l'équivalence des programmes de la figure 107 (p.226).
- **Procédures, passage de paramètres et variables locales.**
  L'appel de procédures dans APP se fait sans passage de paramètres. Il n'y a pas non plus de variables locales. Autrement dit tout passage d'informations se fait par effets de bords dans la mémoire globale ! Les procédures n'ayant pas de signatures, elles sont difficiles à utiliser ; surtout lorsque un programme manipule un grand nombre de variables. Clairement cette technologie est vraiment primitive par rapport aux langages de programmation. Ces caractéristiques expliquent les problèmes concrets PC4, PC5, PC7.
  Dans un tel contexte une analyse de flot de données semble pour le moins nécessaire. Des techniques sont proposées dans la Section IV.5.2.
- **Types et déclarations de variables**.
  Le langage APP n'étant pas typé, déterminer l'usage des variables est plus difficile encore

1. Ces concepts ont été étudiés dans le domaine des langages de programmation fonctionnels [FielHarr88].
2. On est alors en droit de se demander pourquoi c'est justement la liaison dynamique qui a été choisie dans CPP. Si ce n'est pas un hasard, c'est sans doute pour des problèmes d'efficacité et de facilité de mise en oeuvre. Si l'on veut réaliser une liaison statique dans un langage fonctionnel, il est nécessaire de manipuler la "fermeture" de chaque fonction dans son environnement de définition [FielHarr88]. Ici le problème est le même et deux solutions classiques peuvent être envisagées (1) La première consiste à associer à chaque terme affecté, l'état de la mémoire au moment de son affectation. Dans l'exemple la fermeture du terme `$A` rangé dans la variable `B` est le couple ([`$A`], {`A`->[`1`]}) puisque la valeur associée à la variable `A` était [`1`] au moment de l'affectation. Cette solution n'est pas directe à mettre en oeuvre ; elle n'a pas été retenue pour CPP. (2) La deuxième solution consiste à procéder par "copies", c'est à dire à substituer chaque variable libre par sa valeur lors de l'affectation ; quitte à dupliquer dans le cas où il y a plusieurs occurrences d'une même variable. La valeur associée à la variable `B` sera [`1`;`1`]. Si le terme avait contenu 10 occurrences de a celles-ci auraient été dupliquées. Cette solution est naturelle mais elle présente l'inconvénient de stocker en mémoire des valeurs plus volumineuses.

(PC8). Une variable peut être utilisée comme condition ou simplement comme fragment. Dans le premier cas, devoir tout coder en termes d'entiers ne facilite pas la compréhension des programmes. Le domaine d'une variable ne pouvant même pas être restreint à un intervalle particulier, il est nécessaire d'examiner toutes les expressions dans laquelle elle intervient pour savoir quelles sont les valeurs qui peuvent logiquement lui être affectée. Le problème est accru par le fait qu'il n'est même pas nécessaire de déclarer les variables explicitement. Dans ces conditions, déterminer manuellement la signature d'une procédure est bien difficile (PC5).

- **Instructions de contrôle**.
  Tel que nous l'avons défini, APP ne propose que deux instructions de contrôle, la séquence et l'instruction conditionnelle. Il n'y a ni instructions de branchement (goto), ni instructions itératives. Ces caractéristiques jouent un rôle très important lorsque l'on veut appliquer des techniques d'analyse de programmes[1].
- **Instruction de sortie, fragments et séquence.**
  La composition séquentielle de deux instructions de sortie est équivalente à l'instruction de sortie où les deux termes ont été concaténés. Autrement dit la sémantique des instructions `OUT` $t_1$`;OUT` $t_2$ et `OUT` $t_1$ @ $t_2$ est la même.
  En fait le langage APP tel que nous l'avons défini dépend des domaines sémantiques *Fragment* et *Fragments*, de la valeur *EmptyFragments* et de la fonction *FragmentsSynthesis*. Pour le cas de CPP nous avons posé *Fragment*=[Char], *Fragments*=[*Fragment*], *EmptyFragments*=[] et *FragmentsSynthesis*=@ car le codomaine doit être [[Char]] (CPP produit une listes de lignes). APP peut être utilisé dans d'autres situations en changeant la valeur de ces paramètres. Par exemple *Fragment*='a, *Fragments*={*Fragment*}, *EmptyFragments*={} et *FragmentsSynthesis*=$\cup$ correspond à un préprocesseur générant des ensembles d'éléments de type 'a.

## IV.4.4 Conclusion

Dans cette section nous sommes partis de CPP, un outil des années 70 utilisant une terminologie et des concepts spécifiques. Il a été montré comment l'on pouvait passer de ce langage à APP, un préprocesseur abstrait. Ce processus d'abstraction permet de raisonner en utilisant des notions connues ; celles provenant de la programmation détaillée.

1. Ceci dit, de telles constructions pourraient facilement être intégrées à APP si cela était nécessaire. C'est le cas si l'on voulait modéliser CTF (compile time facility), le préprocesseur associé à PL/1. De même certains assembleurs permettent l'utilisation d'instructions de branchement quelconques [Wink87]. L'intérêt d'APP est justement de pouvoir facilement changer les caractéristiques du préprocesseur étudié tout en conservant la même approche. En ce sens notre approche est générale.

# IV.5 Quelques exemples d'applications

Si nous nous sommes appliqués à montrer que CPP pouvait être vu comme un langage de programmation, si la sémantique dénotationnelle du langage APP a été décrite, ce n'est pas uniquement comme exercices de style. Nous cherchons à faciliter la maintenance et la ré-ingéniérie des familles de programmes. L'objectif de cette section est de montrer que le processus d'abstraction proposé fournit une aide précieuse. En fait, l'idée sous-jacente est très simple : puisque CPP peut être vu comme un langage, on peut utiliser des concepts et des techniques de programmation détaillée.

## IV.5.1 Une classification

La programmation détaillée est un thème si large qu'un très grand nombre de concepts peuvent être réutilisés. Il n'est pas possible dans ce chapitre de décrire l'ensemble des techniques utiles. Nous nous limiterons donc à présenter quelques exemples significatifs. La plupart des techniques retenues ont été mises en oeuvre dans un prototype (Section IV.6.2).

Les techniques d'analyse peuvent être classées en fonction des aspects concernés (lexicaux, syntaxiques ou sémantiques), des constructions disponibles dans le langage (e.g. appel de procédure, instruction conditionnelle, liaison dynamique, etc.) et des informations fournies lors de l'analyse (analyse statique ou analyse dynamique) (figure 122).

*figure 122* Classification des techniques d'analyse pour les préprocesseurs

***(a)* Analyse statique ou dynamique ?**

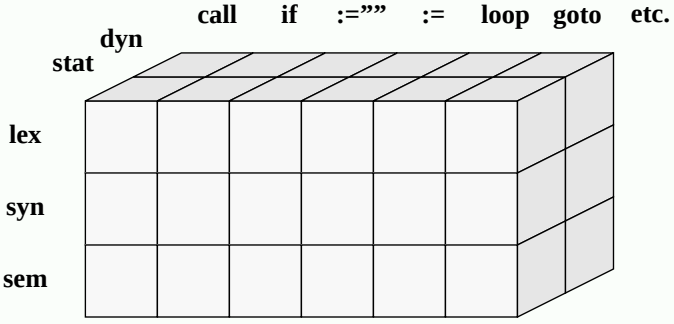
Une technique d'analyse peut être statique ou dynamique. Dans le deuxième cas les entrées du programme sont entièrement spécifiées.

***(b)* Préprocesseur lexical, syntaxique ou sémantique ?**

Un préprocesseur peut être intégré plus ou moins profondément au langage de programmation. Ici nous distinguons trois niveaux essentiels. $APP_{lex}$ a connaissance uniquement du lexique du langage. $APP_{syn}$ a des connaissances syntaxiques et finalement $APP_{sem}$ considère la sémantique statique du langage.

***(c)* Constructions disponibles**

Les techniques d'analyse dépendent très largement des constructions disponibles dans le langage. Par exemple la présence de boucles donne lieu à des équations de flot de données récursives. La présence d'un flot de contrôle arbitraire est également difficile à traiter.




Si l'on se réfère à la dimension verticale de cette figure, il est temps de mentionner que la définition d'APP proposée correspond en fait à $APP_{lex}$. En effet tel qu'il a été défini, ce langage manipule des listes de fragments, plus concrètement des listes de lexèmes. Les techniques décrites dans la suite se limitent à cette classe. Autrement dit, elles ne considèrent aucune information sémantique et syntaxique. Elles peuvent donc être appliquées aussi bien à du texte quelconque qu'à des programmes (les exemples choisis retiennent la première solution)[1].

Dans la suite, nous nous limiterons au cas particulier de CPP et aux instructions que propose ce préprocesseur. Autrement dit si l'on considère la dimension horizontale de la figure, on ne prendra en compte que les cases marquées call, if et :="" puisqu'elles correspondent respectivement à l'inclusion textuelle, à la compilation conditionnelle et à la définition de macro.

Finalement nous avons choisi de présenter uniquement des techniques d'analyse statique.

## IV.5.2 Analyse de flot de données

Les techniques d'***analyse de flot de données*** sont fort utiles dans le domaine de la maintenance et de la ré-ingéniérie de p-programmes. Elles le sont bien plus encore dans le cas d'APP car ce langage utilise une mémoire globale. Tout échange d'informations est alors une indication précieuse. Dans cette section nous définissons quelques fonctions élémentaires utilisées dans les sections suivantes ; entre autres pour générer automatiquement des signatures pour les procédures.

Dans la figure 115 les fonctions *OutDefStmt*, *SuppVarStmt* et *InVarStmt* sont définies de manière récursive en se basant sur la syntaxe abstraite d'APP. De telles fonctions sont analogues à celles habituellement définies sur les langages de programmation classiques (voir par exemple [KeabRM88] ou [AhoSU89]). Nous avons dû cependant les adapter à notre problème et à la liaison dynamique.

- *OutDefStmt*. Cette fonction retourne l'ensemble des définitions "visibles" à la fin de l'exécution d'une instruction donnée. Ici une "définition" est un couple indiquant la valeur associée à une variable. La fonction *OutVarStmt* est similaire, si ce n'est qu'elle ne retourne que le nom des variables. Cette information est intéressante car elle correspond aux variables qui *peuvent* être modifiées par l'instruction.
- *SuppVarStmt*. Cette fonction donne l'ensemble des variables qui sont "supprimées" par l'instruction. Si une variable est dans cette liste on est *sûr* qu'elle est affectée dans l'instruction. Autrement dit les définitions de cette variable qui étaient visibles en entrée ne le sont plus en sortie (i.e. la valeur initiale n'est plus accessible).
- *InVarStmt*. Une variable appartenant à l'ensemble défini par cette fonction *peut* modifier le comportement de l'instruction ; en tout cas au moins une occurrence de cette variable apparaît dans un terme-condition ou dans un terme[1].
  En fait il est utile de raffiner cette fonction pour indiquer si une occurence de la variable apparaît dans une condition (la variable sera appelée "***sélecteur***"), ou dans un terme (on parlera alors de "***substitut***"[2]). On définit ainsi deux fonctions *InSelectStmt* et *InSubstStmt* (celles-ci

1. En contrepartie les services offerts sont bien évidemment moindres. Il s'agit donc d'une limitation. En réalité, nous nous sommes également intéressés à certains aspects d'$APP_{syn}$ et d'$APP_{sem}$ mais par manque de place et de temps, nous avons choisi de ne pas faire figurer ces travaux dans cette thèse. Quelques idées préliminaires sont présentées dans [Favr93].

1. La liaison dynamique rend l'utilisation de cette fonction plus complexe que dans le cas d'une liaison statique. Dans ce dernier cas une variable n'apparaissant pas dans cet ensemble ne peut pas avoir d'effet sur le résultat de l'instruction. Ici par contre, il faut être plus prudent. Une telle variable ne peut avoir une influence que dans le cas d'une mémoire dans laquelle la variable est référencée directement ou indirectement par l'une des variables de l'ensemble. Par exemple *InVarStmt* [[ y:="$x";OUT $z ]] = {x,z} ce qui signifie que les valeurs de x et de z peuvent modifier le comportement de l'instruction. Par contre la variable v ne peut modifier le comportement de l'instruction que dans le cas d'une mémoire initiale dans laquelle x ou z référence v directement ou indirectement. C'est le cas si mem = { z -> "$r" ; r -> "$v" }, ce ne l'est pas si mem = { z -> "$r" ; r -> "1" }. Nous verrons par la suite quelles sont les implications pratiques d'une telle règle.

*figure 123* Définition de *OutDef, OutVar, SuppVar* et *InVar.*

L'objectif de cette figure est de montrer que quelques lignes seulement suffisent pour définir rigoureusement des propriétés relativement complexes. D'où l'avantage d'avoir défini formellement le langage APP...

*OutDefStmt* ∈ Stmt -> *ProcEnv* -> {Var x (Term ∪ nil)}
*OutVarStmt* ∈ Stmt -> *ProcEnv* -> {Var}
*SuppVarStmt*∈ Stmt -> *ProcEnv* -> {Var}
*InVarStmt* ∈ Stmt -> *ProcEnv* -> {Var}

*OutDefProc* ∈ Stmt -> *ProcEnv* -> {Var x (Term ∪ nil) }
*OutVarProc* ∈ Stmt -> *ProcEnv* -> {Var }
*SuppVarProc*∈ Stmt -> *ProcEnv* -> {Var}
*InVarProc* ∈ Stmt -> *ProcEnv* -> {Var}

*OutDefStmt [[* **NOP** *]]* π = {}
*OutDefStmt [[* x **:=** "t" *]]* π = {(x,"t")}
*OutDefStmt [[* **RESET** x *]]* π = {(x,nil)}
*OutDefStmt [[* **OUT** t *]]* π = {}
*OutDefStmt [[* **CALL** pn *]]* π = (*OutDefProc [[*π pn*]]*π*)*
*OutDefStmt [[* **IF** t **THEN** $s_1$ **ELSE** $s_2$ *]]*π = (*OutDefStmt [[*$s_1$*]]*π) ∪ (*OutDefStmt[[*$s_2$*]]*π)
*OutDefStmt [[* $s_1$ **;** $s_2$ *]]* π = (*OutDefStmt[[*$s_1$*]]*π) *XXX* (*SuppVarStmt[[*$s_2$*]]*π)
∪ (*OutDef Stmt[[*$s_2$*]]*π)

*OutVarProc[[* p *]]* π = $\pi_{[1]}$ ( *OutVarProc[[* p *]]* π )
*OutVarStmt[[* s *]]* π= $\pi_{[1]}$ ( *OutVarStmt[[* p *]]* π )

*OutDefProc [[* **PROCEDURE** pn **IS BEGIN** s **END** *]]* π = *OutDefStmt [[*s*]]* π

*SuppVarStmt [[* **NOP** *]]* π = {}
*SuppVarStmt [[* x **:=** "t" *]]* π = {x}
*SuppVarStmt [[* **RESET** x *]]* π = {x}
*SuppVarStmt [[* **OUT** t *]]* π = {}
*SuppVarStmt [[* **CALL** pn *]]* π = (*SuppVarProc [[*π pn*]]*π)
*SuppVarStmt [[* **IF** t **THEN** $s_1$ **ELSE** $s_2$ *]]*π = (*SuppVarStmt [[*$s_1$*]]*π) ∩ (*SuppVarStmt[[*$s_2$*]]*π)
*SuppVarStmt [[* $s_1$ **;** $s_2$ *]]* π = (*SuppVarStmt[[*$s_1$*]]*π) ∪ (*SuppVarStmt[[*$s_2$*]]*π)

*SuppVarProc [[* **PROCEDURE** pn **IS BEGIN** s **END** *]]* π = *SuppStmt[[* s *]]* π

*InVarStmt [[* **NOP** *]]* π = {}
*InVarStmt [[* x **:=** "t" *]]* π = *VarInTerm[[* t *]]*
*InVarStmt [[* **RESET** x *]]* π = {}
*InVarStmt [[* **OUT** t *]]* π = *VarInTerm[[* t *]]*
*InVarStmt [[* **CALL** pn *]]* π = *InVarProc[[* π pn *]]* π
*InVarStmt [[* **IF** t **THEN** $s_1$ **ELSE** $s_2$ *]]* π = *VarInTerm[[* t *]]* ∪ (*InVarStmt [[*$s_1$*]]* π) ∪ (*InVarStmt[[*$s_2$*]]* π)
*InVarStmt [[* $s_1$ **;** $s_2$ *]]*π = (*InVarStmt[[*$s_1$*]]*π) ∪ ((*InVarStmt[[*$s_2$*]]*π) - (*SuppVarStmt[[*$s_1$*]]*π))

*VarInTerm* ∈ Term -> {Var}
*VarInTerm[[* [] *]]* = {}
*VarInTerm[[* **$**x :: t *]]* = {x}∪ *VarInTerm[[* t *]]*
*VarInTerm[[* c::t *]]* = *VarInTerm[[* t *]]*

*InVarProc [[* **PROCEDURE** pn **IS BEGIN** s **END** *]]* π = *InVarStmt [[*s*]]* π

ne sont pas représentées dans la figure). Les ensembles produits ne sont pas nécessairement disjoints car une variable peut à la fois être un sélecteur et un substitut.

Les fonctions *OutDefProc*, *SuppVarProc* et *InVarProc* sont équivalentes si ce n'est qu'elles s'appliquent aux procédures. En fait il est nécessaire de connaître l'environnement de procédures pour effectuer cette analyse car chaque appel est virtuellement substitué par le corps de la procédure correspondante. On suppose donc que le graphe d'appel des procédures n'est pas cyclique.

2. Le terme "sélecteur" provient du modèle abstrait présenté dans le Chapitre II (Section II.3.5). Par contre nous avons choisi "substitut" plutôt que "paramètre" (Section II.3.4) car ce dernier terme est d'usage courant dans les langages de programmation. Ici il désigne indifféremment "sélecteur" ou "substitut".

## IV.5.3 Génération de signatures

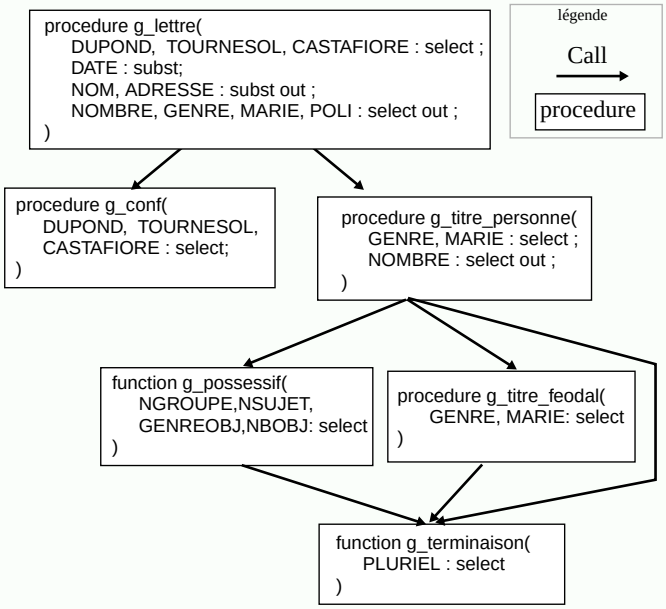
Les fonctions définies ci-dessus peuvent être directement utilisées pour la ***génération automatique de signatures***. Il s'agit là d'une aide fondamentale pour le chargé de maintenance (voir les problèmes concrets PC5, PC7 et PC8). En fait, il n'y a pas de notion précise de "***signature***" ; il s'agit simplement d'informations de flot de données résumant les caractéristiques d'une procédure. En fonction de ses besoins, le chargé de maintenance doit pouvoir affiner ces informations ou au contraire les simplifier.

Dans la figure 124 la lettre générique est analysée et des signatures simples sont produites pour chaque procédure. Le lecteur pourra vérifier qu'une telle représentation améliore considérablement la compréhension du système en se reportant à la représentation CPP présentée dans la figure 111 (p.231). Avait-il deviné quelle était la signature de g_lettre ?

*figure 124* Signatures simples

Le graphe présenté est obtenu directement à partir du graphe d'appel (IV.5.4) et des fonctions *InSelectStmt*, *InSubstStmt* et *OutVarStmt*. Le chargé de maintenance a choisi de n'afficher que les paramètres d'entrée (les variables qui font partie de *InSelectStmt* ou de *InSubstStmt* annotées respectivement "select" ou "subst"). Pour ces paramètres il est indiqué si la procédure peut les modifier ou non (annotation "out").

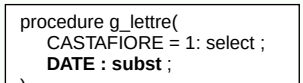
Ces signatures permettent au chargé de maintenance de savoir quels sont les paramètres qu'il peut ou doit définir lorsqu'il utilise une procédure.
Remarquons aussi que l'annotation "select out" peut correspondre à la présence d'une valeur par défaut. C'est le cas par exemple du paramètre NOMBRE dans la procédure g_titre_personne.




Le chargé de maintenance peut vouloir se concentrer sur des cas particuliers et explorer les signatures dans de tels cas. Un système interactif fournit alors une aide appréciable (figure 125).

*figure 125* Exploration interactive de signatures

Comme tous les paramètres ne sont pas toujours nécessaires (à cause des valeurs par défaut), le chargé de maintenance peut se demander quel sont ceux qui doivent être affectés dans une situation donnée. Par exemple il peut désirer savoir quels sont les paramètres à indiquer après avoir sélectionné le cas où CASTAFIORE = 1. La signature ci-contre est générée de la même manière mais après une spécialisation de la procédure (voir plus loin la Section IV.5.8). Cette signature indique qu'il ne manque que la valeur du substitut DATE.

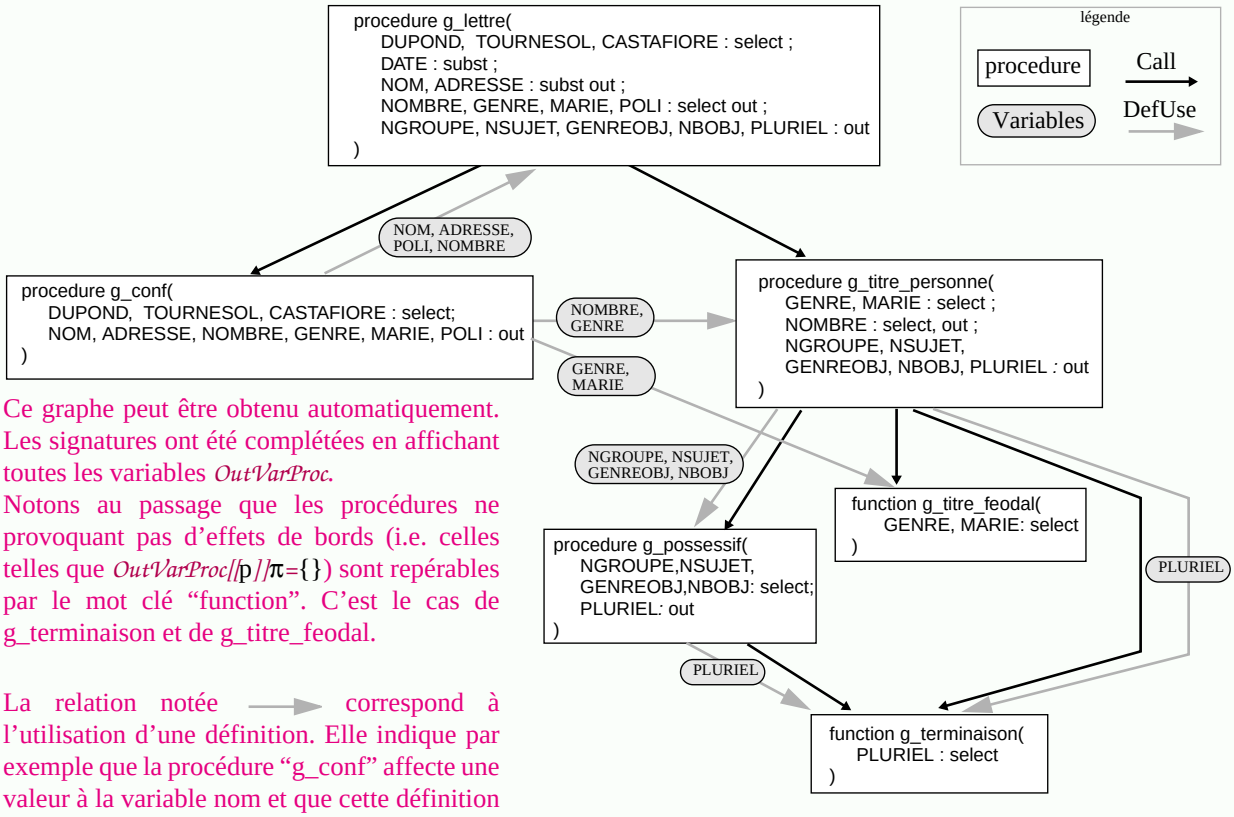



## IV.5.4 Graphe de flot de données inter-procédural et inter-modulaire

Au contraire le chargé de maintenance peut vouloir plus d'informations. C'est le cas par exemple s'il doit réutiliser une procédure : s'il veut éviter des effets de bords, il doit connaître les variables produites ou modifiées par celle-ci. Plus généralement, s'il doit modifier un module, il doit le

comprendre. Le ***flot inter-procédural de données*** peut alors lui donner des indications intéressantes. Il est notamment possible de reconnaître un certain nombre de “patrons” issus du modèle abstrait (figure 126).

*figure 126* Flot inter-procédural de données




Ce graphe peut être obtenu automatiquement. Les signatures ont été complétées en affichant toutes les variables *OutVarProc*.
Notons au passage que les procédures ne provoquant pas d’effets de bords (i.e. celles telles que *OutVarProc*[]p[]π={}) sont repérables par le mot clé “function”. C’est le cas de g_terminaison et de g_titre_feodal.

La relation notée → correspond à l’utilisation d’une définition. Elle indique par exemple que la procédure “g_conf” affecte une valeur à la variable nom et que cette définition est (potentiellement) utilisée dans “g_lettre”.

Comparer cette relation au graphe d’appel est fort instructif car cela permet de déterminer si les valeurs sont synthétisées ou héritées (dans le deuxième cas le flot des données va dans le même sens que l’appel de procédure). Grâce au modèle abstrait nous avons vu que l’héritage des choix correspondait à la situation “normale” : celle où un objet générique détermine le choix de ses variantes internes. Dans l’exemple le sous-système correspondant à g_titre_personne répond à cette règle.
Au contraire, les fonctions de simplification de domaine sont généralement repérables par la synthèse de valeurs. Dans le cas présent, la procédure g_conf devrait attirer l’attention du chargé de maintenance. Tout laisse à penser (y compris son nom...) qu’il s’agit d’une procédure de “configuration” utilisée pour définir les paramètres des autres objets génériques. Sa signature indique que DUPOND, TOURNESOL et CASTAFIORE forme le domaine abstrait alors que NOM, ADRESSE, NOMBRE, GENRE, MARIE et POLI font partie du domaine concret. Visiblement, si l’on observe la signature de g_lettre, ces derniers paramètres peuvent aussi être définis indépendamment (ce sont aussi des paramètres d’entrées). Autrement dit d’autres “configurations” que celles proposées dans g_conf sont sans doute possibles.

Le flot inter-procédural de données peut aussi assister le chargé de maintenance à déterminer quelles sont les variables “locales” à une procédure donnée. APP ne propose aucun mécanisme d’encapsulation et chaque affectation a potentiellement un effet sur n’importe quelle partie d’un module. Pourtant il est clair qu’en pratique l’usage de certaines variables est logiquement localisé. Dans l’exemple, les variables GENREOBJ et NBOBJ peuvent être considérées comme locales au sous-système g_titre_personne car elles ne sont utilisées que dans celui-ci. Bien sûr cette hypothèse peut être infirmée par d’autres modules ; par exemple si une procédure utilisant g_lettre a le mauvais goût de consulter ou de modifier NBOBJ. L’hypothèse de localité facilite néanmoins la compréhension de ce module particulier.

A un niveau de granularité plus fort on peut aussi considérer le ***flot inter-modulaire de données***.

## IV.5.5 Graphe de flot de contrôle inter-procédural et inter-modulaire

Jusque là nous nous sommes surtout concentrés sur le flot de données. Une autre alternative est d'étudier le ***flot de contrôle***. Cette information peut être considérée à différents niveaux de granularité.

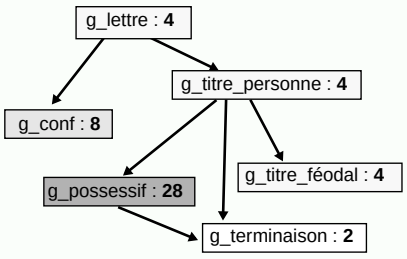
A un niveau fin on parle tout simplement du ***graphe de flot de contrôle*** (CFG en anglais). Grossièrement les noeuds de ce graphe sont les instructions et les conditions ; les arcs représentent le passage possible d'un noeud à l'autre. Dans le cas d'APP ce graphe n'est en fait qu'un graphe dirigé sans cycle puisqu'il n'y a ni instruction itérative, ni instruction de branchement (Section IV.4.3)[1].
Dans le cas de plusieurs procédures, il est courant de définir le ***graphe d'appel***. Les noeuds sont des procédures ; il s'agit d'un ***graphe de flot de contrôle inter-procédural***[2].
Finalement à un niveau de granularité encore plus fort, il peut être utile de considérer le ***graphe de flot de contrôle inter-modulaire***. Les noeuds sont alors des modules.

Ces différents graphes sont traditionnellement utilisés pour définir des mesures de complexité ou de qualité. Il s'agit de pouvoir repérer les composants complexes que l'on suppose "à risques".
Par exemple à un niveau de granularité fin, le graphe de flot de contrôle peut servir à calculer le nombre total de chemins, la complexité cyclomatique de Mac Cabe, etc. En fait de très nombreuses mesures peuvent être définies à ce niveau de granularité. Le tout est de déterminer quelles sont celles qui sont représentatives et dans quelles conditions elles le sont (figure 127).

*figure 127* Nombre de chemins

La définition inductive présentée ci-contre correspond au nombre total de chemins internes (les appels de procédures ne comptent pas). Le graphe d'appel présenté est décoré et coloré en utilisant cette mesure. Elle donne une première idée au chargé de maintenance de la complexité des procédures. Il détecte ainsi, sans consulter le code, que la procédure g_possessif est plus complexe que les autres.

$PathsStmt \in$ Stmt -> int
$PathsStmt\ [\![$ **NOP** $]\!] = 1$
$PathsStmt\ [\![$ x **:=** "t" $]\!] = 1$
$PathsStmt\ [\![$ **RESET** x $]\!] = 1$
$PathsStmt\ [\![$ **OUT** t $]\!] = 1$
$PathsStmt\ [\![$ **CALL** pn $]\!] = 1$
$PathsStmt\ [\![$ **IF** t **THEN** $s_1$ **ELSE** $s_2$ $]\!] = PathStmt\ [\![s_1]\!] + PathStmt\ [\![s_2]\!]$
$PathsStmt\ [\![$ $s_1$ **;** $s_2$ $]\!] = PathStmt\ [\![s_1]\!] * PathStmt\ [\![s_2]\!]$

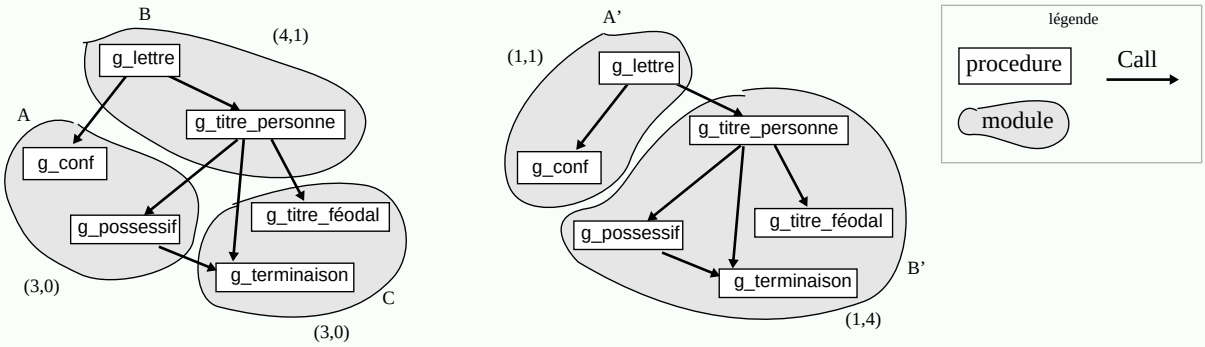



Comme toutes les mesures, cette mesure doit être considérée avec beaucoup de précautions. En pratique, celle-ci n'est pas très utile car le nombre de chemins total est une fonction exponentielle du nombre d'instructions conditionnelles mises en séquence. Très rapidement cette fonction retourne des valeurs n'ayant plus beaucoup de sens. Par exemple le nombre total de chemins du système g_lettre (en comptant les appels de procédures) est de ... 29696 ! Evidemment ces chemins ne sont pas tous possibles ; il n'existe pas 29696 variantes de la lettre !
Cet exemple ne remet pas en compte l'utilité des mesures, simplement il indique que dans ce cas précis, compter le nombre de chemins de cette manière grossière n'est pas très utile.

Si l'on considère le flot de contrôle à un niveau de granularité plus fort, ce sont les notions de couplage et de cohésion de modules qui interviennent[3]. De telles mesures peuvent être utilisées pour donner des indications sur la qualité d'une architecture et ainsi assister le chargé de maintenance à limiter la déstructuration de modules APP a priori ou a posteriori (figure 128).

1. Cette caractéristique simplifie considérablement les propriétés définies à partir de ce graphe
2. Si l'on utilise les termes conventionnels, le graphe d'appel correspond au graphe d'inclusion entre fichiers ; il modélise ce qui est généralement appelé la "relation de dépendance".
3. De telles mesures peuvent faire intervenir d'autres caractéristiques que le flot de contrôle.

*figure 128* Couplage et cohésion




Deux architectures différentes sont présentées pour le système g_lettre. Les couples (i,j) associés à chaque module correspondent à des mesures triviales de couplage et de cohésion basées sur le graphe d'appel : il s'agit respectivement du nombre d'arcs intra et inter-modulaires. L'objectif étant de maximiser le couplage et de diminuer la cohésion, la deuxième solution semble préférable. L'exemple proposé est excessivement simple, mais lorsque le chargé de maintenance est confronté à des systèmes complexes déstructurés au cours des années, ce genre de mesures se révèle fort utile. Elles peuvent par exemple attirer l'attention du chargé de maintenance sur des procédures qui semblent mal placées. Une fois de plus le chargé de maintenance n'a pas eu besoin de se plonger dans le code en premier lieu

## IV.5.6 Graphe de dépendances de programmes (PDG)

Il est possible d'intégrer le flot de données et le flot de contrôle dans un même graphe[1]. Cependant le graphe de flot de contrôle fait apparaître des informations qui finalement n'ont pas beaucoup d'importance d'un point de vue sémantique. Par exemple l'ordre entre deux instructions est toujours représenté explicitement, même si ces instructions sont totalement indépendantes et peuvent être exécutées dans n'importe quel ordre.

Le ***graphe de dépendance de programme*** (***PDG*** en anglais) est une structure permettant d'éviter ces problèmes. Ce type de représentation a été largement étudié dans la littérature relative à la programmation détaillée (Annexe A.3.8). Ici nous allons uniquement présenter l'adaptation que nous proposons pour APP ; les caractéristiques originales de ce langage sont la liaison dynamique et l'importance accordée aux instructions de sortie.

### IV.5.6.1 *Graphes de dépendances pour APP*

Pour simplifier, seul le cas mono-procédural est présenté. Autrement dit, il s'agit de représenter le graphe de dépendance d'une procédure ne contenant aucun appel[2].

Le plus simple, pour comprendre ce qu'est un PDG, est de considérer un exemple particulier tout en étudiant les caractéristiques générales de ces graphes (figure 129).

Un noeud d'un APP/PDG peut être :

- un *terme-condition* ;

1. Un solution traditionnelle consiste a associer à chaque occurence de variable l'ensemble de définitions visibles en ce point et inversement d'associer à chaque définition l'ensemble des occurrences de variables qu'elle peut atteindre (on parle parfois de ***chaîne utilisation-définition*** et de ***chaîne définition-utilisation***).
2. On peut se ramener à ce cas de figure en "mettant à plat" les procédures. Cette opération correspond à la répétition de l'opération "inclure" (Section II.6.2).

- une *instruction simple*, c'est à dire une définition de variable[1] ou une instruction de sortie ;
- un *noeud "Entry"* correspondant à l'entrée de la procédure ;
- un *noeud d'initialisation* de variable (voir ci-dessous).

Les arcs sont de trois types[2] :

- *Dépendance de contrôle*. Intuitivement ces arcs permettent de représenter l'imbrication des instructions conditionnelles et le fait que l'exécution d'une instruction imbriquée dépende de l'évaluation du terme-condition. L'origine d'un tel arc est un terme-condition ; autrement dit un noeud contrôlant l'exécution des instructions imbriquées (le noeud "Entry" est un cas particulier). La destination d'un arc est, soit une instruction simple, soit un autre terme-condition dans le cas d'une instruction conditionnelle imbriquée[3]. Les arcs sont décorés par une valeur booléenne pour préciser la branche de l'instruction conditionnelle à laquelle ils appartiennent.
- *Dépendance de donnée*. Intuitivement ces arcs indiquent que l'exécution d'une instruction ou l'évaluation d'un terme-condition dépend potentiellement d'une définition de variable. C'est le cas bien évidemment si l'instruction contient une occurrence de cette variable. Les noeuds d'initialisation sont utilisés dans le cas où la valeur initiale d'une variable est accessible.
- *Dépendance de sortie*[4]. L'ordre entre les instructions de sortie est bien évidemment important. Un arc représentant une dépendance de sortie va donc d'une instruction de sortie à une autre instruction de sortie qui doit la suivre[5]. Pour simplifier les figures la dépendance de sortie ne sera généralement pas représentée explicitement.

L'une des caractéristiques à retenir est que plusieurs programmes peuvent correspondre au même PDG ; dans ce cas ils sont sémantiquement équivalents. En fait, pour générer un programme, il suffit de parcourir un PDG en respectant l'ordre partiel induit par les relations de dépendances. Les PDG sont donc des représentations internes possibles pour les programmes (figure 129).

### IV.5.6.2 *Graphe de dépendance en présence d'une liaison dynamique*

Normalement les graphes de dépendances sont définis pour des langages utilisant une liaison statique. Dans un tel cas une affectation comme `c := $d` est donnée-dépendante de toutes les définitions de `d` visibles en ce point du programme. Cette sémantique est naturelle. En fait les informations contenues dans un PDG pourraient être obtenues plus ou moins facilement par le chargé de maintenance grâce à une lecture attentive du code source. L'absence d'instructions de branchement et d'itérations rend cette tâche pénible mais réalisable.

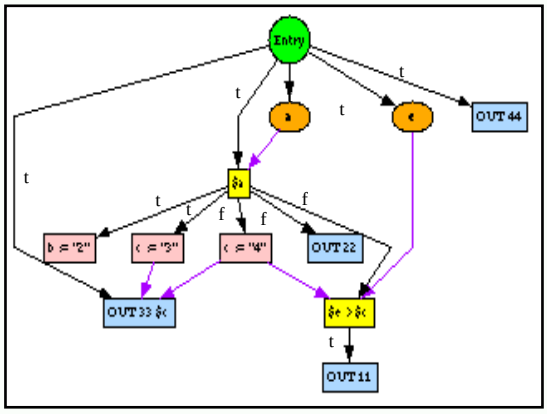
---

1. Une instruction `x := "t"` ou `RESET t`
2. Des solutions plus fines sont parfois proposées, entre autre pour différencier les dépendances associées aux instructions itératives. Dans le cas d'APP on peut se limiter à l'instruction conditionnelle et cette distinction n'a donc pas lieu d'être. D'autres variations sont possibles ; ici il n'est pas nécessaire de les prendre en compte car nous nous intéressons principalement aux problèmes de découpe de programmes. Les deux relations présentées sont suffisantes.
3. Traditionnellement ce sont les relations inverses des dépendances qui sont représentées : les arcs vont d'une entité aux entités qui dépendent d'elle et non pas l'inverse. Nous avons ici conservé les mêmes conventions.
4. Cette dépendance n'est normalement pas prise en compte dans les PDG standards. Ici elle est introduite car l'instruction de sortie joue un rôle particulièrement important dans APP.
5. En fait ce type de dépendance pourrait être vu comme un cas particulier de la dépendance de donnée si l'on transformait les instructions `OUT t` en des instructions d'affectation `output := $output @ t`.

*figure 129* Un graphe de dépendance pour deux procédures simples

Les procédures $p_1$ et $p_2$ ont le même PDG ; elles sont donc sémantiquement équivalentes. L'ordre entre `b := "2"` et `c := "3"` n'a pas d'importance.

```
PROCEDURE p1 IS
BEGIN
   IF $a THEN
      b := "2" ;
      c := "3"
   ELSE
      c := "4" ;
      IF $e > $c
         THEN OUT 11
         ELSE NOP ;
      OUT 22 ;
   OUT 33 $c ;
   OUT 44
END
```




```
PROCEDURE p2 IS
BEGIN
   IF $a THEN
      c := "3" ;
      b := "2"
   ELSE
      c := "4" ;
      IF $e > $c
         THEN OUT 11
         ELSE NOP ;
      OUT 22 ;
   OUT 33 $c ;
   OUT 44
END
```

La représentation graphique présentée ci-dessus ne montre pas les dépendances de sortie. Les dépendances de contrôle sont décorées par "t" et "f" (pour "true" et "false"). Les ovales sont des noeuds d'initialisation. Au premier coup d'oeil le chargé de maintenance peut se rendre compte que les variables `a` et `e` sont des paramètres de la procédure ; ce qui n'est pas évident si l'on considère le code source. Les termes-conditions `$a` et `$e > $c` dépendent respectivement de ces valeurs. Notons aussi par exemple que l'instruction `OUT 33 $c` dépend des deux définitions de `c`.

Au contraire *APP est basé sur une liaison dynamique ; cette caractéristique rend virtuellement impossible la production de telles informations mentalement, même pour des exemples de quelques lignes*. La figure 130 présente l'un de ces exemples. Les caractéristiques qu'il met en oeuvre sont présentées ci-dessous.

Une instruction d'affectation APP n'est jamais donnée-dépendante. Comme conséquence, l'ordre d'affectation des variables n'a jamais d'importance, même si elles se référencent les unes les autres. Par contre, ces liaisons sont prises en compte au moment où les variables sont utilisées (il s'agit tout simplement de la sémantique de la liaison dynamique). Ce comportement ne pose aucun problème pour un préprocesseur, c'est n'est pas le cas pour un humain (PC3) !

Que ressort-il d'un exemple comme celui de la figure 130 ?

- *Certains programmes APP sont incompréhensibles, dans le sens propre du terme*. Par exemple, quel chargé de maintenance se rendrait compte de toutes les dépendances décrites dans le PDG ? Sans cette information, qui oserait modifier ou restructurer une telle procédure ?[1] L'exemple peut paraître surfait, mais le code présenté dans l'annexe n'est il pas plus complexe encore ? Quelle est la probabilité pour qu'un tel système soit modifié sans provoquer des effets de bords ?
- *Si les informations contenues dans les graphes de dépendances sont utiles, leur construction doit être automatisée*. Il est alors nécessaire d'adapter les algorithmes classiques de constructions de PDG au cas de la liaison dynamique. Un algorithme a été défini et implémenté dans le cadre de cette thèse (Section IV.6.2). Celui-ci ne sera pas détaillé ici.

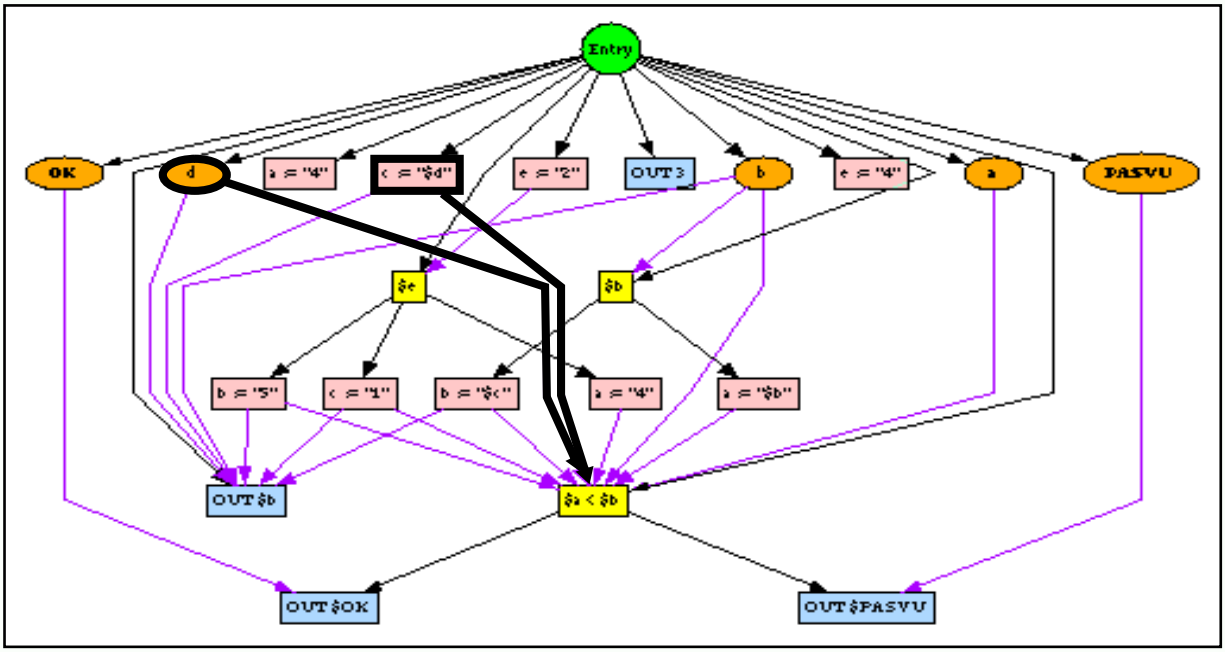
Les PDG se révèlent être très utiles dans le cadre de la maintenance et de la ré-ingénierie. Nous avons déjà vu sur des exemples qu'ils permettaient de déterminer avec précision les paramètres

1. Réponse : certains chargés de maintenance inconscients ou contraints de porter le logiciel.

*figure 130* Graphe de dépendance et liaison dynamique

```
PROCEDURE q1 IS
/* you are not expected to understand this ! */
BEGIN
   e := "4" ;
   IF $b THEN a := "$b" ELSE b := "$c" ;
   c := "$d" ;
   OUT 3 ;
   e := "2" ;
   IF $e THEN a := "4";b := "5"ELSE c:= "1" ;
   IF $a < $b THEN OUT $OK ELSE OUT $PASVU ;
   e := "4" ;
   OUT $b ;
END
```

Le lecteur n'est pas sensé comprendre la procédure $q_1$ .
(cette remarque ne rappelle-t-elle pas le commentaire trouvé dans un noyau unix ?).
Ci-dessous le PDG correspondant est présenté. Les dépendances de sortie ont été omises. Le lecteur pourra vérifier qu'aucune définition de variable n'est donnée-dépendante d'une autre (i.e. aucun arc ne va d'une définition à une autre définition). Plus particulièrement on peut voir que l'affectation `c := "$d"` ne dépend pas de la valeur initiale (d). Par contre cette liaison se retrouve plus loin dans le terme-condition `$a < $b` !




d'entrée d'une procédure[1]. Bien que de multiples autres utilisations soient possibles, la plus importante est sans nul doute le découpage.

## IV.5.7 Découpage de procédures

Les techniques de ***découpage*** ("***slicing***" en anglais) font partie des techniques d'analyse de programmes les plus en étudiées actuellement[2]. Le lecteur trouvera dans l'Annexe A.3.10 ("Découpage") de plus amples informations.

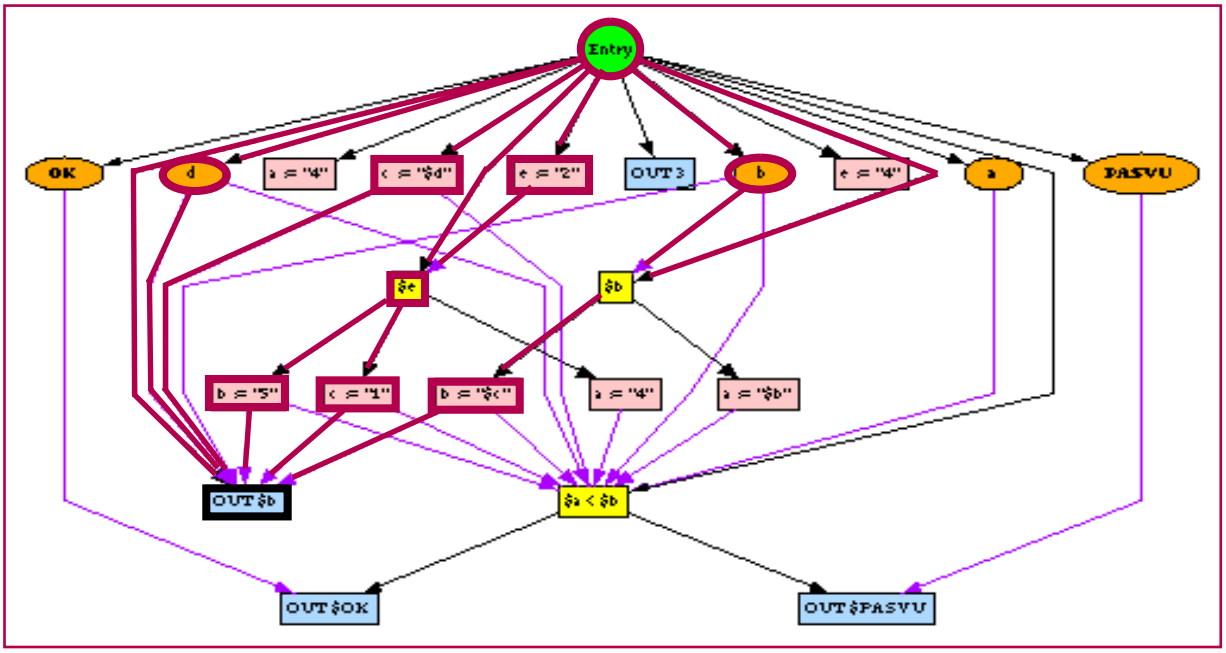
Initialement la notion de ***découpe*** d'un programme ("***program slice***") a été introduite par Weiser [Weis84]. Comme il l'a fait remarquer, lors de la mise-au-point d'un programme, si les valeurs de certaines variables ne sont pas correctes en un point donné, le programmeur cherche à ne lire que

1. Ces graphes améliorent nettement le calcul des propriétés *InVarStmt*, *InSubstStmt*, *InSelectStmt*. On obtient des approximations bien plus précises que celles calculées avec les opérations récursives présentées auparavant. Par exemple, le PDG de la figure 130 montre que le paramètre d est un sélecteur alors que cela n'apparaît pas explicitement (cette variable ne figure dans aucun terme-condition).
2. [Tip94] fourni sans doute le guide le plus complet et le plus à jour en ce qui concerne la littérature relative aux techniques de découpage. Cet article comprend plus de 87 références sur le sujet ; 60% sont postérieures à 1990. Ces chiffres témoignent de l'activité qui règne autour de ce sujet. [Lyle94] est une autre lecture d'intérêt.

les instructions pouvant influer sur le calcul de ces variables. Mentalement il ne s'intéresse qu'à une "découpe" du programme. Le problème qui se pose alors est de pouvoir automatiser le calcul de telles découpes.

Il existe de très nombreuses variations sur ce thème. Ici pour simplifier l'exposé nous allons uniquement présenter le cas des ***PDG-découpes arrières statiques***[1]. Comme leur nom l'indique, ces découpes peuvent facilement être calculées à partir du PDG d'un programme *P :* il s'agit du sous-graphe obtenu en suivant les dépendances de contrôle et de données dans le sens inverse. On part du noeud correspondant à l'instruction *I* que l'on souhaite étudier. Le sous-graphe ainsi obtenu correspond à un programme *P'* sémantiquement équivalent en ce point. Plus précisément pour toute exécution de *P* et *P'* le comportement de l'instruction *I* est équivalent[2] (figure 131).

*figure 131* Exemple de PDG-découpe arrière statique




Le sous-graphe présenté ci-dessus correspond à une découpe arrière de la procédure $q_1$ pour l'instruction `OUT $b`. Elle met en évidence les noeuds du PDG dont dépend cette instruction.

La procédure $q_{slice}$ ci-contre est obtenue en parcourant ce sous-graphe. Cette découpe de $q_1$ permet au chargé de maintenance soucieux de comprendre le calcul de $b de ne se concentrer que sur les parties de $q_1$ impliquées. En générant la signature de cette découpe arrière, le chargé de maintenance découvre que cette instruction ne dépend que des paramètres (b) et (d). Cette information est apparaît directement dans le PDG ci-dessus.

```
PROCEDURE qslice IS
BEGIN
   IF $b THEN NOP ELSE b := "$c" ;
   c := "$d" ;
   e := "2" ;
   IF $e THEN b := "5" ELSE c:= "1" ;
   OUT $b
END
```

L'exemple précédent montre que même pour une procédure de quelque lignes, déterminer mentalement une découpe n'est pas facile ; surtout en présence d'un mécanisme de liaison dynamique. En fait la situation est bien plus critique encore dans le cas multi-procédural car une

1. Cette technique a été mise en oeuvre dans un prototype. C'est la cas aussi des PDG-découpes avant statiques. Les PDG-découpes sont moins générales que celles définies par Weiser mais elles sont mieux adaptées à notre problème.
2. Les dépendances de sortie ne sont pas prises en compte pour le calcul de sous graphe car l'on ne cherche pas à obtenir un programme donnant la même sortie depuis le début de l'exécution. On ne s'intéresse qu'au fragment produit localement par l'instruction.

simple lecture de la procédure ne suffit pas. Au contraire il est nécessaire d'examiner successivement différentes procédures en tenant compte de leur ordre d'appel. Cette tâche devient rapidement trop complexe pour que le chargé de maintenance puisse avoir réellement confiance dans le résultat qu'il obtient mentalement.

Techniquement par contre il est possible de construire des graphes de dépendances dans le cas multi-procédural et d'exploiter ceux-ci pour calculer des découpes. On parle alors de SDG (System Dependence Graph) plutot que de PDG (Annexe A.3.8). On peut aussi se ramener au cas mono-procédural en mettant à plat le graphe d'appel des procédures grâce à l'opération "inclure". Quelque soit la méthode utilisée, les techniques de découpage se révèlent fort utiles pour faciliter la compréhension de systèmes comme **G_LETTRE** (figure 132).

*figure 132* Découpes inter-procédurales

```
#include "g_conf"
#include "g_titre_personne"
NOM, ADRESSE
"J'ai le plaisir de "
#if NOMBRE>1 || defined(POLI)
"vous"
#else
"te"
#endif
"convier à la reunion du " DATE
#if POLI
"Je vous prie d'agreer"
"mes salutations distinguees."
#else
"salut."
#endif
```

Supposons que le chargé de maintenance lise la procédure "**G_LETTRE**" représenté à gauche et qu'il souhaite déterminer statiquement quel peut être le comportement de la ligne NOM, ADRESSE (il cherche à savoir quelles sont les expansions possibles de ces macros et dans quels cas). Grâce à un outil interactif calculant des découpes, désigner cette ligne est suffisant pour mettre à jour les parties de la procédure influant sur ce calcul. Dans cet exemple précis, on découvre que seul l'appel à la procédure **G_CONF** peut avoir une influence. La découpe de droite présente une information plus précise après une "mise à plat" des procédures. Il est plus facile pour le chargé de maintenance de comprendre ces 13 lignes que les 96 lignes réparties dans les 6 procédures de la figure 111 (p.231).

```
/* procedure "g_conf" */
IF $DUPOND THEN
    NOM := ""Dupond & Dupont""
    ADRESSE := ""Commissariat""
ELSE NOP
IF $TOURNESOL THEN
    NOM := ""Tournesol""
    ADRESSE := ""Moulinsard""
ELSE NOP
IF $CASTAFIORE THEN
    NOM := ""Castafiore""
    ADRESSE := ""Rome""
ELSE NOP
/* procedure g_lettre */
OUT $NOM , $ADRESSE
```

Les appels de procédures sont les instructions les plus significatives, en tout cas pour comprendre l'architecture d'un système. En même temps leur effet est des plus difficile à cerner. Comme nous l'avons vu le langage APP ne permet pas le passage de paramètre et tout transfert d'informations se fait via des variables globales. Dans APP, la liaison entre un paramètre formel et un paramètre effectif se fait donc via l'affectation d'une variable. Le problème est qu'une telle instruction d'affectation n'est pas nécessairement proche de l'appel de la procédure et il est alors difficile de s'apercevoir qu'il s'agit en fait d'un passage de paramètre. Lorsque plusieurs appels de procédures sont présents dans une même portion de code il est encore plus difficile de mettre en relation les affectations aux appels. La technique de découpage est alors fort utile comme le montre l'exemple de la figure 133. D'un point de vue conceptuel, elle permet au chargé de maintenance de retrouver les fonctions d'héritage des choix à partir des mécanismes de bas niveaux proposés par APP. C'est donc un bon exemple de retro-ingénierie[1].

1. Ces différents exemples suggèrent qu'il est sans doute possible de passer semi-automatiquement de la forme impérative APP à une forme fonctionnelle comme celle décrite dans le modèle abstrait (Chapitre II). L'avantage d'une telle transformation est évident lorsque l'on compare la facilité de compréhension de chaque type de représentation. Dans l'état actuel de nos travaux, il est encore trop tôt pour décrire de telles transformations et les problèmes qu'elles posent ; il s'agit d'une piste future qui semble particulièrement intéressante.

*figure 133* Découpes et appels de procédure

En observant la procédure **G_TITRE_PERSONNE** présentée ci-contre, un chargé de maintenance connaissant bien les fonctionnalités de ce système pourrait peut être avoir une idée assez précise de la relation entres les différentes affectations présentes dans cette procédure et les trois appels qui s'y trouvent (appels à **G_POSSESSIF**, **G_TITRE_FEODAL** et **G_TERMINAISON**). Cependant après quelques années de maintenance et des éventuels rapiéçages, une lecture des différentes procédures n'est pas suffisante pour se convaincre du rôle de chaque instruction.
Dans de telles conditions, observer les découpes arrières relatives à chaque appel de procédure est fort instructif. Dans (a) on voit pas exemple que les premières instructions correspondent à la fonction d'héritage pour l'appel à **G_POSSESSIF**. La découpe (b) met en évidence le fait que dans le cas du titre féodal il s'agit de la fonction identité (les paramètres de cette procédure doivent être définis dans la procédure appelant **G_TITRE_PERSONNE**). En fait ces trois découpes permettent de retrouver a posteriori les informations conceptuelles comme celles qui sont présentées dans le bas de la figure 111 (p.231).

```
#ifndef NOMBRE
#define NOMBRE 1
#endif
#define NGROUPE 1
#define NSUJET 1
#define GENREOBJ GENRE
#if NOMBRE>1
#define NBOBJ 2
#else
#define NBOBJ 1
#endif
#include “g_possessif”
#include “g_titre_feodal”
#define PLURIEL (NOMBRE>1)
#include “g_terminaison”
```

```
#ifndef NOMBRE
#define NOMBRE 1
#endif
#define NGROUPE 1
#define NSUJET 1
#define GENREOBJ GENRE
#if NOMBRE>1
#define NBOBJ 2
#else
#define NBOBJ 1
#endif
#include “g_possessif”
#include “g_titre_feodal”
#define PLURIEL (NOMBRE>1)
#include “g_terminaison”
```

*(a)* **découpe pour l'appel à G_POSSESSIF**

```
#ifndef NOMBRE
#define NOMBRE 1
#endif
#define NGROUPE 1
#define NSUJET 1
#define GENREOBJ GENRE
#if NOMBRE>1
#define NBOBJ 2
#else
#define NBOBJ 1
#endif
#include “g_possessif”
#include “g_titre_feodal”
#define PLURIEL (NOMBRE>1)
#include “g_terminaison”
```

*(b)* **découpe pour l'appel à G_TITRE_FEODAL**

```
#ifndef NOMBRE
#define NOMBRE 1
#endif
#define NGROUPE 1
#define NSUJET 1
#define GENREOBJ GENRE
#if NOMBRE>1
#define NBOBJ 2
#else
#define NBOBJ 1
#endif
#include “g_possessif”
#include “g_titre_feodal”
#define PLURIEL (NOMBRE>1)
#include “g_terminaison”
```

*(c)* **découpe pour l'appel à G_PLURIEL**

## IV.5.8 Spécialisation de procédures et élimination du code mort

La spécialisation est une autre technique de réduction de programme (Annexe A.3.9, "Spécialisation de programmes"). Nous avons déjà vu dans le Chapitre II que cette opération correspondait d'un point de vue abstrait à la restriction de domaine d'une fonction (par exemple l'opération ◁ de Z). Ici il s'agit de déterminer l'opération concrète correspondant à la représentation APP ; plus particulièrement on cherche à réaliser un *mixeur* pour ce langage.

### IV.5.8.1 *Principe général*

En fait plusieurs techniques sont possibles (Annexe A.3.9) mais elles reposent toutes plus ou moins sur le principe suivant. Très grossièrement si l'on dispose d'un langage impératif L dont la sémantique dénotationnelle est $Sem_{\mathcal{L}} \in$ Stmt -> *Mem* -> *Mem* (autrement dit les instructions modifient une mémoire) alors schématiquement un mixeur pour ce langage sera de la forme $Mix_{\mathcal{L}} \in$ Stmt -> *MixMem* -> *MixMem* × Stmt. Autrement dit un mixeur prend en entrée une instruction tout comme un interpréteur, mais à la place de transformer une mémoire déterminée en une nouvelle mémoire, il joue deux rôles simultanément : (1) il transforme une contrainte sur une mémoire en une nouvelle contrainte et (2) il retourne une instruction spécialisée. En fait le terme "mixeur" vient du fait qu'un tel système rempli en même temps le rôle d'un interpréteur et de compilateur : il interprète

tout ce qu'il peut interpréter en exploitant la contrainte et génère une instruction pour le code qui n'a pas pu interprété statiquement.

Le plus simple est sans doute de prendre un exemple. Ci-dessous une même instruction d'un langage L est respectivement interprétée et spécialisée.

- $Sem_{\mathcal{L}}$[[ **IF** a=2 **THEN** b :=2 **ELSE** c := 2*a ]] {a->3 ; b->4; c->2} = {a->3 ; b-> 2 ; c -> 6 }
- $Mix_{\mathcal{L}}$[[ **IF** a=2 **THEN** b :=2 **ELSE** c := 2*a ]] ‘a>3’ = ( ‘a>3 ∧ c=2*a’ , c := 2*a )

Dans le premier cas la mémoire est une fonction totale de {a;b;c} dans l'ensemble des entiers. On obtient une mémoire de même nature. Dans le deuxième cas la contrainte définie par le prédicat ‘a>3’ fait que la branche “sinon” est systématiquement sélectionnée. En sortie on sait que la contrainte ‘a>3 ∧ c=2*a’ sera vérifiée. L'instruction spécialisée est c := 2*a. Pour toute mémoire vérifiant ‘a>3’ cette instruction est sémantiquement équivalente à l'expression composée. Par contre elle est plus simple ; c'est justement ce que l'on recherche lorsque l'on spécialise un programme : il s'agit d'une technique de réduction de programmes.

### IV.5.8.2 *Contraintes sur le domaines*

Les différentes techniques de spécialisation sont plus où moins complexes à mettre en oeuvre et se différencient essentiellement par le type de contraintes qui peut être imposé sur le domaine (c'est à dire sur la forme de *MixMem* et la manière de le représenter).

Grossièrement l'*évaluation partielle* consiste à associer une valeur constante à certains paramètres (pour une fonction f(a,b,c) on pose par exemple a=1) [JoneGS93]. Autrement dit la description du nouveau domaine se fait en imposant un prédicat de la forme $x_1=t_1 \wedge ... \wedge x_n=t_n$ où toutes les variables $x_i$ sont distinctes ; autrement dit une seule valeur peut être associée à une valeur donnée. Cette restriction rend l'évaluation partielle plus facile à mettre en oeuvre que les autres techniques.

A l'autre extrême l'*exécution symbolique* permet de restreindre le domaine en utilisant un prédicat quelconque (par exemple $a<b \wedge c=4 \vee c=b+a^2$ ) souvent appelé condition de chemin (PC, Path Condition en anglais) [King81] [CoenDGM91]. Cette dernière solution est complexe à mettre en oeuvre car elle implique la manipulation et (la simplification) d'expressions symboliques.

### IV.5.8.3 *Un mixeur pour APP*

Dans le cadre de cette thèse, un mixeur a été implémenté en se basant sur la sémantique dénotationnelle du langage APP. Décrire en détail ce mixeur serait trop long ; contentons nous d'indiquer les principes directeurs.
En ce qui concerne l'expression des contraintes, actuellement c'est une solution intermédiaire qui a été retenue : elle consiste à associer un ensemble de valeurs à un sous-ensemble des paramètres. Plus précisément les contraintes sur le domaine sont de la forme $(x_1=t_{11} \vee x_1=t_{12} \vee ... \; x_1=t_{1k_1}) \wedge (x_2=t_{21} \; ...) \; ... \wedge (x_n=t_{n1} \vee ... \; x_1=t_{1k_n})$ où les variables $x_i$ sont toutes distinctes. On parlera de “multi-mémoire” dans la mesure où une contrainte peut être représentée comme une “mémoire” contenant non pas *une* seule valeur mais plutot plusieurs valeurs possibles (la valeur nil correspond au cas ou l'on sait que la variable (la macro) n'est pas initialisée).

Le principe général présenté ci-dessus a été appliqué. Un interpréteur APP génère un *Fragments* et une mémoire à partir d'une instruction et d'une mémoire. Le mixeur APP prend une instruction et une "multi-mémoire" et génère une "multi-mémoire" ainsi qu'une instruction spécialisée plutôt qu'un *Fragments*. La figure 134 compare les principales fonctions sémantiques d'APP et celles correspondant au mixeur.

*figure 134* Sémantique du mixeur APP vs. sémantique d'un interpréteur APP

**mixeur**

| | | |
|---|---|---|
| *MixMem* | = | { Var x (Term ∪ nil) } x {Var } |
| *MixProc* | ∈ | Stmt -> *ProcEnv* -> *MixMem* -> (*MixMem* x Proc) |
| *MixStmt* | ∈ | Stmt -> *ProcEnv* -> *MixMem* -> (*MixMem* x Stmt) |
| *MixTerm* | ∈ | Term -> *MixMem* -> Term |
| *MixCondTerm* | ∈ | CondTerm -> *MixMem* -> ({ ℵ } x Bool x Bool) x CondTerm |

**interpréteur**

| | |
|---|---|
| *Mem* | = Var -> Term |
| *SemProc* | ∈ Proc -> *ProcEnv*-> *Mem* -> (*Fragments* × *Mem*) |
| *SemStmt* | ∈ Stmt -> *ProcEnv* -> *Mem* -> (*Fragments* × *Mem*) |
| *SemTerm* | ∈ Term -> *Mem* -> *Fragments* |
| *SemCondTerm* | ∈ CondTerm -> *Mem* -> ℵ |

## IV.5.9 Elimination des définitions inutiles et des instructions vides

Spécialiser un programme peut introduire de nombreuses définitions inutiles. En effet, puisque l'on substitue une variable par sa valeur lorsque celle-ci est connue, l'affectation qui donnait cette valeur à la variable devient inutile.

Si l'on élimine des définitions inutiles, une instruction conditionnelle peut elle même devenir inutile ; inutile dans le sens où elle ne contient aucune instruction, ni dans la partie THEN, ni dans la partie ELSE. Il est alors possible de supprimer cette instruction conditionnelle. Un terme condition est alors éliminé, et avec lui des occurrences de variables. Ainsi certaines définitions de variables peuvent à leurs tours devenir inutiles...

Il est donc nécessaire d'alterner l'élimination des définitions inutiles et l'élimination des instructions vides jusqu'à ce que l'on atteigne un point fixe.

Ces techniques peuvent être appliquées après une spécialisation ou indépendamment.

## IV.5.10Synthèse

Ci-dessus différents concepts et techniques provenant de la programmation détaillée ont été présentés en expliquant comment ils pouvaient être utilisés dans le cadre de la compréhension de programmes APP.

Seuls les exemples les plus représentatifs de nos travaux ont été retenus. Dans le cadre de cette thèse nous nous sommes en fait intéressés à d'autres aspects et avons déjà obtenu certains résultats préliminaires. Ceux-ci n'ont pas été inclus ici pour ne pas surcharger ce chapitre.

Quoi-qu'il en soit, on comprendra aisément que bien d'autres techniques de programmation détaillée puissent être appliquées. L'analogie faite entre APP et un langage de programmation est une source d'inspiration très prometteuse.

Dans tout ce chapitre, les exemples retenus pour illustrer le discours sont particulièrement simples. Pourtant certains sont quasiment incompréhensibles pour l'être humain. Les techniques décrites ont originellement été développées pour être appliquées aux p-programmes d'une complexité bien supérieure. Par certains aspects, le langage APP pourrait être considéré comme un langage-jouet ; de même les exemples traités seraient vus comme des cas d'écoles dans le domaine de la programmation détaillée.

*En fait, le cas des préprocesseurs est particulièrement intéressant car il permet d'appliquer à des cas simples des techniques conçues pour gérer des cas complexes*. C'est bien évidemment une situation idéale dans la mesure où ces techniques vont pouvoir être appliquées efficacement.

On peut alors se demander si la simplicité apparente des programmes APP est due uniquement à nos exemples et si cette assertion reste vraie sur des programmes industriels. Notre expérience nous pousse à croire que c'est le cas, mais cela reste à prouver. Soulignons simplement que la complexité des programmes APP existant est limitée par celle que peut gérer l'être humain.

# IV.6 Expériences

Pour des raisons didactiques, le discours tenu dans cette thèse n'a été illustré que par des exemples simples et abstraits. Cela ne veut pas dire pour autant que nous n'ayons jamais considéré des programmes de grandes tailles.
Dans cette section nous décrivons tout d'abord quelques expériences préliminaires. Elles expliquent et justifient la démarche suivie dans ce chapitre (IV.6.1).
Un prototype a été réalisé pour valider les idées avancées ; ses fonctionnalités sont présentées dans la Section IV.6.2 ; puis son architecture est discutée dans la Section IV.6.3 en prenant en compte quelques considérations d'implémentation.

## IV.6.1 Expériences préliminaires

Nos premiers travaux portaient sur la définition d'une représentation multi-langages adaptée à la programmation globale [Favr88] [EstuFavr89] [Favr89]. Nous nous intéressions aux logiciels modulaires ; il s'agissait grossièrement de définir un modèle de données et un langage de requête facilitant l'analyse de l'architecture détaillée. L'approche retenue était relativement académique dans la mesure où elle supposait que les logiciels manipulés vérifiaient un certain nombre de contraintes.

Une expérience professionnelle de deux ans en tant qu'ingénieur système nous a montré (1) qu'en fait la plupart des logiciels existants n'étaient pas à proprement parler "modulaires", en tous cas que leur architecture était loin d'être explicite, (2) que les problèmes de variations étaient bien réels même s'ils étaient plutôt flous, (3) que si l'on désirait proposer des services utiles, il était préférable de construire des outils pouvant être adaptés aux logiciels tels qu'ils étaient plutôt que tels que l'on voudrait qu'ils soient.

L'installation et l'étude de nombreux logiciels (X11, Motif, emacs, gcc, etc.) nous ont fait découvrir les problèmes pragmatiques que posait l'utilisation des préprocesseurs et a renforcé l'idée que la compréhension de familles de programmes était un thème important.

Après avoir décidé de nous concentrer sur CPP comme cas d'étude, différentes pistes ont été suivies parallèlement :

- Pour mieux comprendre quel est l'usage du préprocesseur dans le cas des logiciels de grandes tailles, des outils rudimentaires d'analyse ont étés construits à partir de langages de commandes et d'outils unix comme grep, sed et awk. Plus de 12 millions de lignes de codes ont été analysées avec ces outils[1]. Ceci dit les informations obtenues n'étaient pas très fines et se sont révélées peu exploitables. Il s'agissait par exemple de déterminer les paramètres d'un logiciel, notamment le nombre de sélecteurs et de substituts mais en utilisant des techniques ad-hoc. Par exemple dans le cas de Motif, 6288 macros seraient définies, 1211

1. X11R5, Motif et la plupart des logiciels de domaine public installés sur le serveur de notre laboratoire. X11R5 à lui tout seul représente déjà plus de 3 millions de lignes de code !

d'entres elles seraient des sélecteurs (ces macros apparaissent dans des directives #if), parmi celles-ci 220 seraient des paramètres sélecteurs (aucune définition de ces macros n'est présente dans le logiciel analysé). La méthode employée pour calculer de tels chiffres fait qu'ils ne sont pas vraiment fiables. Notamment les bibliothèques ne sont pas prises en compte.

- Parallèlement à cela nous nous sommes plongé dans le code de cccp, une implémentation de CPP[1]. L'idée initiale était (1) d'étudier plus en détail l'usage du préprocesseur sur un exemple particulier mais volumineux ; (2) de pouvoir déterminer avec précision la sémantique du préprocesseur étudié (bien que cet outil soit bien documenté [Stal92], dans les cas limites seule la lecture du code permet de comprendre son comportement !) ; (3) de pouvoir "instrumenter" celui-ci pour qu'il puisse générer des mesures plus précises que celles calculées avec les outils décrits ci-dessus.

Plusieurs conclusions ressortent de ces expériences : (1) étudier l'usage des préprocesseurs est aussi difficile qu'instructif, (2) les solutions ad-hoc proposées pour analyser les logiciels ont rapidement montré leurs limites, (3) raisonner en termes concrets n'est propice ni à la compréhension du problème, ni à l'élaboration d'outils évolués.

Ces expériences préliminaires nous ont donc poussés à étudier CPP selon un point de vue abstrait, d'où la définition d'APP et de sa sémantique.

## IV.6.2 Le prototype APP/Champollion

Pour valider les idées avancées dans les sections antérieures, la plupart des techniques décrites ont été implémentées dans un prototype appelé APP/Champollion

### IV.6.2.1 *Fonctionnalités offertes*

Actuellement le prototype proposé offre les fonctionnalités suivantes :

- *Analyse et synthèse*. APP n'étant qu'un langage abstrait, la première fonctionnalité à offrir est bien évidemment de pouvoir passer d'une représentation CPP à une représentation APP et inversement. Concrètement le prototype réalisé permet de générer (1) une procédure APP à partir d'un fichier CPP, (2) un module APP à partir d'un nom de répertoire unix, (3) un programme principal APP à partir d'une ligne de commande CPP.
- *Affichage.* Les programmes APP n'ont pas de syntaxe concrète puisque ce sont des arbres abstraits. Pour faciliter la mise au point et l'élaboration des exemples décrits dans cette thèse le prototype affiche les différentes entités en utilisant une syntaxe similaire à celle utilisée jusqu'ici.
- *Interprétation*. Un interpréteur du langage APP a été construit en suivant fidèlement la sémantique dénotationnelle du langage. Cette fonctionnalité est indispensable pour vérifier l'équivalence entre CPP et APP.
- *Génération de signatures*. Grâce aux techniques décrites dans la Section IV.5.3 et IV.5.6 le prototype permet de générer les signatures de procédures APP dans un format similaire à celui

1. Il s'agit d'un programme C monolithique de 8000 lignes ! Ce programme est portable et utilise le préprocesseur de manière extensive.

présenté dans les exemples. Ces signatures peuvent être plus ou moins affinées.

- *Génération du graphe de flot de contrôle, du graphe d'appel et du graphe de dépendances*. En partant d'un module ou d'une procédure APP, ces différents graphes sont produits et peuvent être représentés sous trois formes différentes : (1) sous forme d'une structure de données utilisable par les autres composants, (2) sous forme d'une représentation textuelle (3) sous forme graphique.
- *Mesures.* Quelques mesures élémentaires ont été définies et implantées, comme par exemple le nombre total de chemins d'une procédure, le niveau d'imbrication maximum pour les instructions conditionnelles, etc.
- *Spécialisation et élimination de code mort*. Un mixeur a été implémenté en utilisant les techniques présentées dans la Section IV.5.8. L'élimination des définitions et des instructions inutiles a également été implantée. Ces deux outils peuvent être utilisés conjointement ou séparément.
- *Découpe*. Un outil permettant de réaliser des découpes statiques avant et arrière a été réalisé.

D'autres fonctionnalités mineures sont intégrées au prototype. En fait comme nous allons le voir celui-ci est extensible.

### IV.6.2.2 *Restrictions fonctionnelles*

Le prototype actuel souffre des restrictions suivantes :

- Seul un sous-ensemble du langage CPP est accepté en entrée, notamment les macro-paramétrées ne sont pas prises en compte et certaines conventions lexicales sont incorrectement traitées (par exemple l'utilisation du caractère d'échappement \ en fin de ligne). Les directives spécifiques aux différentes implémentations de CPP (#pragma, #error, etc.) n'ont pas non plus été intégrées.
- Toutes les techniques d'analyse se font dans un environnement de procédures fixe. Autrement dit les variations impliquées par les chemins de recherches ne sont pas prises en compte.
- Le mixeur perd de nombreuses opportunités pour simplifier et évaluer les termes-conditions. Par exemple le terme-condition (1 || **$**a) sera laissé intact. Cette restriction ne compromet en rien la validité de la spécialisation, simplement elle diminue son efficacité.
- Dans la version actuelle les algorithmes de découpage et de spécialisation ne prennent pas en compte le cas inter-procédural. Plus exactement, ils commencent par "aplatir" le graphe d'appel et calculent des résultats corrects mais ne sont pas capables de reformer les procédures correspondantes.

### IV.6.2.3 *Evaluation*

Les fonctionnalités offertes par le prototype proposé couvrent presque l'intégralité des concepts avancés dans ce chapitre (nous n'avons pas, par exemple, implémenté les mesures de couplage et de cohésion). Pour l'instant, les restrictions décrites ci-dessus nous empêchent de faire des expérimentations "grandeur-nature" sur des programmes industriels (par exemple la plupart de ces programmes utilisent des définitions de macros paramétrées). Ceci dit, l'ANNEXE B contient de nombreux exemples montrant les possibilités offertes par le prototype.

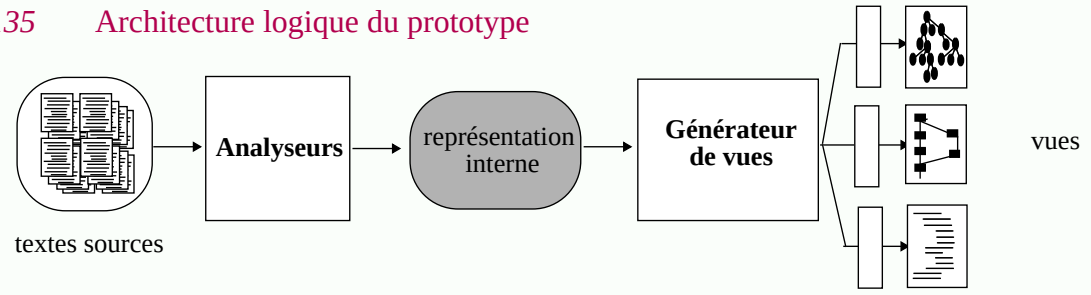
## IV.6.3 Architecture et implémentation de APP/Champollion

Le prototype réalisé est intéressant de part les fonctionnalités qu'il offre mais aussi de part son architecture.

### IV.6.3.1 *Architecture logique*

Si l'on cherche à obtenir un système ouvert et extensible, il faut bien évidemment éviter de construire un système monolithique. L'architecture logique du prototype correspond à une architecture devenue classique pour les environnements de maintenance et de ré-ingéniérie [Favr93] [Arno93]. La figure 130 présente une vision simplifiée d'une telle architecture. Ici la représentation interne correspond aux arbres abstraits APP. Sur cette figure l'aspect réutilisation n'est pas explicité. L'un des problèmes et de disposer d'outils ou de bibliothèques simplifiant l'écriture des composants intervenants à chaque étape.

*figure 135* Architecture logique du prototype




### IV.6.3.2 *Implémentation*

Bien qu'il soit difficile de mesurer avec précision le temps pris pour réaliser l'ensemble du prototype, il s'agit environ de trois ou quatre semaines de travail. Si l'on considère la gamme étendue des fonctionnalités offertes et les résultats obtenus, cette durée peut être considérée comme tout à fait satisfaisante. Selon nous, ce résultat encourageant est principalement dû à trois facteurs :

- *Rigueur et abstraction*. Toutes les techniques implémentées avaient été étudiées longuement auparavant ; en se basant sur des abstractions plutôt que des représentations particulières. La rigueur s'est révélée payante. Par exemple, à partir de la sémantique dénotationnelle d'APP, un interpréteur a pu être réalisé en quelques jours seulement.
- *Outils facilitant le prototypage*. Un langage fonctionnel fortement typé a été utilisé (un dialecte de ML). Ce langage s'est révélé tout à fait adapté à notre contexte. Les constructions qu'il offre ont permis de décrire de manière très concise algorithmes et structures de données. Par exemple la description des arbres abstraits APP se fait en quelques lignes seulement et les différentes définitions dirigées par la syntaxe présentées dans ce chapitre s'écrivent presque telles quelles grâce au "pattern matching". Le fait de disposer d'un langage interprété facilite énormément la mise au point et la création de nouvelles vues.
- *Réutilisation*. A chaque étape différents composants ou outils ont pu être réutilisés. Par exemple les analyseurs ont été générés automatiquement à partir de la syntaxe des langages[1] ; les représentations graphiques sont gérées par un éditeur de graphes ; un éditeur de texte permet l'affichage des représentations textuelles ; des bibliothèques génériques ont étés définies pour manipuler les structures de graphes, elles offrent entre autres des opérations de

fermeture transitive, de calcul de sous-graphes, etc. La génération de nouvelles vues est grandement facilitée dans un tel contexte. Par exemple trois heures ont été nécessaires pour intégrer au prototype la possibilité de construire le graphe de flot de contrôle et de les afficher graphiquement sur un écran couleur.

Concrètement, le prototype a été programmé en *camllight*[1]. La totalité des services décrits représente environ 5000 lignes de programmes. A titre d'exemple l'interpréteur représente 230 lignes[2]. Les analyseurs lexicaux et syntaxiques ont été respectivement générés à partir de *camllex* et *camlyacc* (des adaptations de lex et de yacc). Ce système est (faiblement) intégré avec *emacs* pour l'affichage des vues textuelles (via l'utilisation d'emacsclient) et avec *DaVinci* pour l'affichage de graphes[3]. L'exécution se fait actuellement sous la forme de trois processus communiquant par des fichiers.

### IV.6.3.3 *Evaluation*

Comme nous l'avons dit dans la section antérieure, jusqu'à maintenant le prototype n'a pas pu être testé sur des exemples en grandeur nature car il ne prend en compte pour l'instant qu'une restriction du langage CPP. L'implémentation actuelle a été réalisée dans un style purement fonctionnel (hormis la gestion des graphes) et l'on peut donc s'attendre à des dégradations de performances sur des exemples de grandes tailles. En tout cas pour tous les exemples présentés dans l'annexe , les temps de réponses ont été négligeables (quelques secondes au plus pour générer le graphe d'appel, le graphe de flot de données, le graphe de dépendance et calculer des découpes).

1. En fait 3 analyseurs sont nécessaires : (1) un pour le langage APP, il permet d'analyser les fichiers CPP ; (2) un pour le langage $L_{cond}$ c'est à dire pour l'analyse des conditions (cette analyse est faite dynamiquement lors de l'interprétation d'un programme APP (Section IV.4.2.5)); (3) un pour le langage de programmes APP, il permet d'analyser une ligne de commande CPP .
1. Ce langage est développé à l'Inria.
2. On remarquera la concision des programmes écrits avec ce langage fonctionnel. Rappelons que cppp, un interpréteur de CPP, représente plus de 8000 lignes de code C soit environ 40 fois plus.
3. DaVinci a été développé à l'université de Bremen (allemagne).

# Conclusion

## C.1 Rappel du cadre de ce travail

Une discipline d'***ingénierie*** a pour but d'appliquer des connaissances scientifiques à la résolution de problèmes industriels. Le développement d'une telle discipline dépend (1) de la formation des personnes qui la pratiquent, (2) d'interactions fortes entre la recherche et l'industrie.

Le ***génie logiciel*** est une discipline bien jeune. Il faut reconnaître que l'utilisation de connaissances scientifiques est encore rare au cours de la production industrielle de logiciels. L'enseignement du génie logiciel en est à ses débuts ; la communication entre recherche et industrie est difficile. Alors que la recherche fondamentale se projette dans le futur, l'industrie est confrontée quotidiennement au présent et surtout au passé.

La ***maintenance*** des logiciels est l'un des problèmes critiques du génie logiciel. Actuellement les difficultés rencontrées dans ce domaine sont dues entre autres : (1) à un manque d'intérêt, (2) à la mauvaise qualité des logiciels âgés. Depuis longtemps l'approche la plus prisée consiste à essayer d'améliorer la qualité des logiciels lors du développement. Bien que cette approche soit *indispensable* elle n'est pas *suffisante.* Entre autre parce qu'elle ignore les logiciels âgés.

Face aux ***logiciels âgés***, deux approches radicalement opposées sont possibles : (1) *maintenir* ces logiciels, c'est à dire accepter la détérioration de leur qualité jusqu'à ce qu'ils deviennent inutilisables. C'est la solution de facilité, mais elle est plus que risquée... (2) *développer* de nouveaux logiciels, ce qui est long, coûteux et risqué.

La ***ré-ingénierie*** est une solution intermédiaire mais ce n'est pas une solution miracle. Du point de vue du génie logiciel, cette approche se distingue par le fait qu'elle permet de concilier recherche et industrie.

La ***programmation globale*** tente de contrôler l'évolution du logiciel. C'est le cas plus particulièrement de la gestion de configurations. Cette discipline est aujourd'hui en pleine expansion et des systèmes de plus en plus nombreux voient le jour. Hélas il n'est pas facile de comparer ces systèmes ni de les évaluer. Cette difficulté est liée au faible niveau de maturité du domaine. Malgré les avancées récentes en terme de technologie, l'état de la pratique reste très largement dominé par l'utilisation de systèmes ad-hoc issus des années 70.

Les ***préprocesseurs*** tel que CPP font justement partie de cette technologie ad-hoc. Développés dans un contexte de programmation détaillée, ils sont utilisés pour résoudre des problèmes de programmation globale, typiquement des problèmes d'architecture et de variation. Malgré les problèmes qu'ils causent, aucun support ne leur est dédié.

## C.2 Contributions

#### *Une clarification des concepts...*

Presque tous les thèmes présentés ci-dessus sont méconnus dans le milieu académique. Ils sont associés à une image négative car les concepts manipulés sont flous. Dans ce document nous nous sommes attachés à introduire chaque domaine en faisant ressortir les idées principales et en analysant les différents problèmes. Leur importance a été soulignée par une collection importante d'informations et les termes clés ont été présentés et discutés. Certaines informations contenues dans le Chapitre I ont servi comme support de cours dans une école d'ingénieur.

Pour faire ressortir les interactions entre la recherche et l'industrie, nous avons proposé le "**modèle hélicoïdal**". Il schématise les relations entre la recherche fondamentale, la recherche appliquée et l'industrie pour montrer que l'évolution de chaque discipline est complémentaire. Nous utilisons aussi ce modèle pour faire ressortir l'importance de la ré-ingénierie comme moyen de transfert de technologie.

Une nouvelle manière de définir la programmation globale est proposée. Elle se base non pas sur des techniques ou des méthodes mais plutôt sur les notions considérées : la programmation détaillée est basée sur la notion d'algorithme et de structure de données ; la **programmation globale** sur les notions d'**architecture**, de **manufacture**, de **variation** et d'**évolution**. Cette décomposition permet de situer des disciplines comme la gestion de configurations ou la réutilisation. Elle reconnaît le fait que plusieurs notions peuvent intervenir dans un même outil : par exemple un langage de programmation comme Ada met en oeuvre des notions de programmation détaillée mais aussi de programmation globale (par exemple les paquetages correspondent à l'architecture et la généricité aux variations).

Les domaines intervenant dans cette thèse étant flous, c'est vrai a fortiori pour leurs interactions. Les relations entre la maintenance, la ré-ingénierie et la réutilisation ont été précisées. La comparaison entre programmation détaillée et programmation globale nous a permis de souligner le **besoin d'un processus de rationalisation dans le domaine de la programmation globale**.

La programmation globale et la ré-ingénierie sont deux thèmes normalement éloignés. Les rapprocher a permis de **mettre en évidence un concept : la ré-ingénierie globale**. Il s'agit d'appliquer la ré-ingénierie aux artefacts du logiciel relevant de la programmation globale. Cette idée a donné lieu à une publication dans le premier Workshop du SEI sur le thème de la ré-ingénierie [Fav94].

### *Un point de vue abstrait sur la programmation globale...*

Un intérêt particulier a été apporté à l'**étude de la programmation globale** ; d'une part pour faire **un premier pas dans la direction d'une rationalisation** ; d'autre part pour définir plus précisément la **problématique de la ré-ingénierie globale**. Dans cette intention un modèle abstrait a été présenté, puis la technologie de programmation globale a été étudiée en se basant sur ce modèle.

L'idée nous ayant poussé à définir un **modèle abstrait pour la programmation globale** était de dépouiller ce domaine de tous les détails concrets rendant obscurs les concepts utilisés. Le modèle proposé est si abstrait qu'il ne fait pas référence au logiciel, ce qui permet de comparer les différentes notions présentées avec celles d'autres domaines.

Le modèle abstrait est basé (1) sur l'utilisation de la **théorie des ensembles**, (2) sur des **notions de programmation détaillée** (par exemple la notion de programme et de sémantique dénotationnelle) et finalement (3) sur certaines **notions provenant des bases de données** (par exemple les modèles relationnels imbriqués ou les bases de données temporelles).

Trois concepts principaux sont à la base de ce modèle : les **objets structurés**, les **objets dérivés** et les **objets génériques**. Ils correspondent respectivement aux problèmes d'architecture, de manufacture et de variation (et d'évolution). Nous nous sommes plus particulièrement intéressés aux objets génériques et à leur représentation concrète.

Grossièrement, un objet générique correspond à la notion de "composant générique" dans le domaine de la réutilisation ou de "groupe de versions" dans le domaine de la gestion de configurations. Un objet générique permet de générer une variante à partir d'un "choix" donné. Alors que d'un point de vue abstrait **un objet générique est une fonction, sa représentation concrète est un programme** (dans un sens large du terme).

Deux mécanismes basiques peuvent être utilisés pour représenter les objets génériques : un **mécanisme de substitution** et un **mécanisme de sélection**. En simplifiant, le mécanisme de sélection permet de représenter des fonctions en extension (i.e. des fonctions "tabulées") ; ceci explique le rapprochement avec les techniques relationnelles utilisées dans les bases de données.

Le modèle abstrait permet aussi de décrire les opérations définies sur les différents types d'objets et sur leur représentation. Par exemple d'un point de vue abstrait la génération d'une variante correspond à l'application d'une fonction ; d'un point de vue concret cette opération est l'interprétation d'un programme. L'étude de ces opérations a mis à jour différents concepts comme par exemple la **spécialisation d'objets génériques** qui correspond à la notion de spécialisation de programme et donc à la restriction de fonction.

Pour faciliter la compréhension de ce modèle, un **exemple trivial** a été développé à partir de quelques mots et de leurs décompositions possibles ("monsieur", "madame", "mademoiselle"). Cet exemple permet de montrer de nombreuses caractéristiques concernant l'architecture, la manufacture, la variation et l'évolution. Pourtant à aucun moment il ne rentre dans les détails techniques si caractéristiques de la technologie du logiciel.

La **technologie de la programmation globale** a ensuite été étudiée. Ses aspects les plus “concrets” ont été présentés mais en se basant sur le modèle abstrait. Un certain nombre de concepts ont ainsi été définis ; par exemple la distinction entre l’architecture détaillée et l’architecture globale. L’influence des problèmes de granularité a été soulignée en de maintes occasions.

Pour chaque thème de la programmation globale, nous avons étudié (1) **les techniques ad-hoc** (elles sont généralement issues des années 70), (2) **l’intégration des concepts de programmation globale dans les langages de programmation**, (3) **les outils spécifiques**, (4) **les bases logicielles**. Des techniques bien différentes ont ainsi pu être rapprochées.

Par exemple, **l’architecture des logiciels** peut être représentée (1) via une organisation particulière du système de fichiers, (2) via l’intégration du concept de module dans les langages modulaires, (3) via l’utilisation de langages d’interconnections de modules (les MIL), (4) ou via l’utilisation d’un modèle de produits dans les bases logicielles.

**La technologie concernant les problèmes de variations et d’évolution** a été plus particulièrement abordée. Il s’agit par exemple (1) d’une manière d’organiser le système de fichiers, (2) de l’introduction de la généricité dans des langages de programmation ou de l’utilisation de préprocesseurs, (3) de systèmes élémentaires comme RCS, ou (4) de gestionnaires de configurations basés sur des modèles de versionnement élaborés. L’importance du niveau de granularité considéré a été souligné ; la distinction entre variation détaillée et variation globale s’est révélée fort utile.

Il est important de constater que ces différentes technologies sont souvent utilisées simultanément. Un **exemple caractéristique** mettant en jeu le système Unix révèle qu’en pratique ce sont des **échafaudages de techniques hétérogènes** qui sont utilisées pour faire face aux problèmes de programmation globale. Par exemple les problèmes de variations sont abordés via l’utilisation de gestionnaires de configurations mais aussi via l’utilisation de préprocesseurs, d’options de compilations, d’éditeurs de liens statiques ou dynamiques, de fichiers de configurations, de variables d’environnement, etc.

C’est d’un point de vue conceptuel que cet enchevêtrement de techniques a été étudié. **Le cycle de production et d’exécution du logiciel peut être vu comme une fonction** prenant en paramètre aussi bien les entrées du programme que la description du contexte spécifique du logiciel. En schématisant, les différentes étapes de ce cycle correspondent à une **spécialisation incrémentalle du logiciel** : un logiciel général est transformé en un logiciel spécifique en introduisant peu à peu des informations décrivant le contexte spécifique du logiciel.

Il a été montré que **quelques concepts seulement sont suffisants pour décrire un cycle complexe**.

Etudier la technologie de la programmation globale a clairement montré le contraste existant entre l’état de la pratique, dominé par l’utilisation de techniques ad-hoc, et l’état de l’art. Cette distance a été utilisée pour affiner le concept de **ré-ingénierie globale**.

Dans ce document **les préprocesseurs ont été présentés sous un angle original**. Dans l'industrie ceux-ci sont normalement vus comme des mécanismes ad-hoc pour résoudre des problèmes mal conceptualisés ; dans le domaine de la recherche ils sont généralement ignorés. Ici au contraire nous proposons à la fois une vision concrète et abstraite.

Nous nous sommes plus particulièrement intéressés à **CPP, le préprocesseur du langage C**. Cet outil a été décrit en détail. Les coutumes liées à son utilisation ont été étudiées. Nous avons mis en évidence le fait que les problèmes abordés étaient essentiellement des problèmes de programmation globale. Le modèle abstrait a donc pu être utilisé.

**Les mécanismes du préprocesseur CPP ont été analysés**. Ce préprocesseur met en jeu à la fois des mécanismes de sélection (la compilation conditionnelle et les chemins de recherche) et des mécanismes de substitution (la macro substitution et l'inclusion textuelle).

Plutôt que d'étudier le préprocesseur CPP sous ses angles les plus repoussants, **un préprocesseur abstrait (APP) a été défini** ainsi que le schéma d'abstraction correspondant. APP a été présenté sous la forme d'un langage de programmation trivial ce qui a permis de faire le **rapprochement entre les mécanismes de CPP et des notions classiques de programmation détaillée**. Par exemple il a été montré que la définition de macro correspondait à une instruction d'affectation et que l'inclusion textuelle correspondait à un appel de procédure.

**La sémantique dénotationnelle de APP a été décrite**. A cette occasion des caractéristiques spécifiques de CPP ont été mises à jour tout en utilisant des concepts connus. Par exemple la définition de macro utilise une liaison dynamique. De telles affirmations n'auraient pu être faites sans bases formelles les rendant explicites.

Ce n'est pas uniquement par curiosité intellectuelle qu'une telle abstraction a été proposée ; c'est surtout parce que **se ramener à la notion de programme permet de réutiliser un grand nombre de concepts et de techniques**. Nous avons ainsi pu utiliser dans un contexte non traditionnel les notions d'interprétation, de compilation, d'évaluation partielle, d'analyse de flots de données, de découpage, de mise au point, etc.

A cette occasion **certaines techniques d'analyses ont été étendues**. Par exemple les techniques de découpages ont été définies dans le cadre d'une liaison dynamique alors que les algorithmes conventionnels se limitent à une liaison statique.

Il est par ailleurs tout à fait intéressant de remarquer que dans le contexte particulier dans lequel nous nous plaçons la ré-ingénierie et les préprocesseurs) il est possible d'utiliser des techniques abandonnées dans les autres domaines car les caractéristiques des informations à traiter sont différentes.

Un prototype mettant en oeuvre les différentes techniques proposées a été réalisé et génére avec succès les informations souhaitées.

## C.3 Perspectives

#### *A propos des préprocesseurs...*

Notre objectif à court terme est de **réaliser des expérimentations sur des logiciels de grandes tailles** tels que le compilateur GCC du projet gnu et le système graphique X-Window. Ayant déjà été confrontés aux problèmes de la compréhension globale de tels logiciels, nous sommes arrivés à la conclusion que des outils spécifiques étaient indispensables.

Il nous semble nécessaire de renforcer les recherches dans le domaine de la **visualisation des objets génériques** mais aussi des **métriques** ; l'idée étant de pouvoir isoler les différents composants "à risques" pour leur appliquer un traitement spécifique. Les techniques se révélant utiles pourront être intégrées dans des environnements de programmation déjà existants ou dans des environnements de ré-ingénierie pour ainsi être applicables dans l'industrie.

Les problèmes de portabilité restent aujourd'hui une réalité industrielle. Un **environnement de portage** serait utile aux chargés de maintenance pour connaître les caractéristiques des différentes plates-formes existantes et surtout les différences qui les séparent. Aujourd'hui cette connaissance reste diffuse. Il s'agit surtout d'un "savoir-faire" qu'acquière chaque chargé de maintenance au cours de nombreux portages. Ce savoir-faire, augmenté des informations extraites via une comparaison systématique des bibliothèques systèmes, pourrait être la base d'un environnement de portage.

#### *A propos de la programmation globale...*

Il serait utile de **revoir certains aspects du modèle abstrait et le compléter**. Utiliser des concepts orientés objet est peut être possible. Il serait également utile d'y intégrer la notion de spécialisation incrémentalle (cette notion a été développée dans le cas particulier du génie logiciel mais elle doit pouvoir être appliquée à d'autres domaines).

**Poursuivre le processus de rationalisation de la programmation globale** nous semble indispensable. Dans cette thèse, nous n'avons fait qu'un tout petit pas dans cette direction ; mais même si la progression risque d'être longue et difficile, la programmation globale doit, un jour ou l'autre, devenir une activité d'*ingénierie* dans le sens propre du terme. Pourquoi à terme ne décrirait-on pas la **sémantique des outils de programmation globale** avant de les construire ? Pourquoi ne serait-il pas possible de comparer précisément les options retenues dans les systèmes existants? Le nombre de systèmes proposés va grandissant et dans le futur une rationalisation risque d'être indispensable.

**Clairement le spectre des concepts pris en compte doit être élargi**. Dans cette thèse nous nous sommes concentrés sur les problèmes de sélection de versions (dans le sens large) au détriment de nombreux autres aspects comme par exemple la coopération. Elargir ne doit pas pour autant signifier rajouter ou mélanger. Autrement dit, avant d'être étudié en détail, chaque thème doit être clairement identifié et isolé autant que possible. Par exemple, dans le cas des langages de programmation, aujourd'hui la description des entrées-sorties est clairement déconnectée des règles de visibilité ou de la syntaxe du langage. Il faut peut être aussi abandonner tout de suite l'idée de vouloir décrire formellement la totalité d'un système complexe (comme il faudra dans le futur abandonner l'idée de construire les systèmes uniquement de manière empirique).

Si ce travail d'abstraction a pu être mené à bien dans le domaine de la programmation globale, c'est surtout parce qu'il était indépendant de tout système particulier. Par contre, pour appliquer concrètement des résultats, s'appuyer sur un environnement de programmation globale est indispensable. **Nos travaux pourraient être intégrés dans l'environnement ADELE**. En fait plusieurs points sont complémentaires par rapport aux travaux de recherche menés actuellement autour de ce système.

Un nouveau modèle de données est en cours de définition. Une attention toute particulière a été apportée aux problèmes d'évolution du logiciel (aux versions historiques et aux versions coopératives). Nos travaux sont complémentaires car ils s'intéressent surtout aux problèmes de variations (aux versions logiques). **Formaliser autant que possible le modèle d'ADELE, son langage et son gestionnaire de configurations** constitue un prolongement possible aux travaux entrepris dans cette thèse.

ADELE a la particularité d'être un système mixte, c'est à dire qu'il est prévu pour contrôler simultanément deux technologies de programmation globale : (1) la base logicielle ; elle utilise un modèle de données adapté à la description d'informations de programmation globale, (2) le système de fichiers ; même s'il est très pauvre c'est un standard d'intégration de-facto. Avoir étudié la technologie ad-hoc et ses usages est bien évidement un avantage dans un tel contexte. L'une des perspectives importantes de nos travaux consiste à **faciliter l'introduction des logiciels existants dans une base logicielle** (ce pourrait être un atout majeur lors de l'adoption par une entreprise d'un gestionnaire de configurations). Une fois le logiciel introduit il s'agit d'**assurer la cohérence entre les différentes représentations**.

Plus particulièrement il nous semble essentiel d'entreprendre des travaux permettant d'**assurer un meilleur contrôle entre la technologie employée pour les variations détaillées (typiquement des préprocesseurs) et celle utilisée pour les variations globales** (la granularité manipulée par ADELE correspond aux fichiers). Pour cela les informations extraites des textes sources des programmes sont bien évidement indispensables ; intégrées aux informations manipulées par le gestionnaire de configurations, elles devraient permettre de contrôler bien plus précisément les variations du logiciel.

### *A propos de la ré-ingénierie globale...*

**Poursuivre des recherches dans le domaine de la ré-ingénierie globale** apparaît essentiel. En premier lieu il s'agit bien sûr de faciliter la maintenance des logiciels existants. L'un des aspects les plus importants est sans doute la **ré-ingénierie de l'architecture**. Et c'est dans cette direction que nous comptons nous diriger. L'originalité de tels travaux consisterait à prendre en compte les problèmes de variations abordés dans cette thèse. Par exemple aujourd'hui un certain nombre de techniques sont disponibles pour restructurer l'architecture des logiciels. Ces travaux se limitent cependant à une seule variante alors qu'en pratique ce sont à des familles de programmes que l'on a affaire. Dans le domaine de la ré-ingénierie globale et même tout simplement de la programmation globale, il faudra aussi se pencher sur les problèmes de **visualisation de structures variantes**.

Dans cette thèse nous avons essentiellement étudié l'intersection entre la ré-ingénierie et la gestion de configurations, mais les travaux concernant la ré-ingénierie pour la réutilisation sont

proches. Bien que cette voie n'ait pas été poursuivie dans cette thèse, les techniques proposées ont certainement **un rôle à jouer dans l'identification et la compréhension des composants réutilisables**.

Finalement, **étudier la technologie nécessaire à la ré-ingénierie globale** nous semble être un axe de recherche prometteur. Les concepts et les techniques de bases de données pourraient jouer un rôle plus important dans ce domaine qu'elles ne le jouent actuellement pour la ré-ingénierie détaillée (très schématiquement remarquons qu'il est plus simple de stocker et de manipuler un graphe d'appels qu'un arbre abstrait). Le besoin de langages de requêtes pour la ré-ingénierie a souvent été souligné. Y intégrer les problèmes de variations est un sujet de recherche pour le futur.

# ANNEXE A

# Fondements:

# Des ensembles aux programmes

# INDEX

# R



# S



# T

# LISTE DES FIGURES

# ANNEXE A

# Fondements:
# Des ensembles aux programmes

**Cette annexe à pour but de présenter les fondements sur lesquels repose cette thèse.**

#### *Rassembler des notions provenant de différents domaines...*

*Le problème que l'on tente de résoudre dans cette annexe est de faire cohabiter des termes, des principes et notations provenant de différents domaines* (bases de données, langages de spécifications, programmation détaillée). Par exemple le terme "relation" est employé dans chaque domaine mais souvent avec une définition différente. Il s'agit parfois d'ensemble de tuples, parfois d'enregistrements; les relations peuvent être binaires ou non, etc.

#### *Des concepts utilisés aux limites...*

Dans cette thèse certaines idées sont basées sur l'utilisation de concepts classiques mais dans des contextes non traditionnels. Par exemple des concepts de programmation détaillée sont utilisés dans le cadre de la programmation globale. *Cette annexe est remplie de remarques triviales dans un contexte classique mais qui se révèlent fort intéressantes lorsque on les considèrent sous un autre angle*. Dans certains cas ce sont des zones proches des "limites" qui sont utilisées ; préciser ce que recouvre le concept est alors essentiel.

#### *Des notations...*

*Cette annexe à aussi pour but d'introduire les notations utilisées tout au long de cette thèse*. Même si la plupart des opérations sont classiques les notations varient selon les domaines et les systèmes.

#### *Quelques points originaux...*

Finalement dans cette annexe la manière de présenter certaines opérations est plus originale. Elle correspond à des aspects qui sont considérés comme important dans cette thèse. Par exemple nous décrivons les opérations de "factorisations", de "développements", de "formulation" et "d'énumération".

***Plan...***

Cette annexe débute tout d'abord par des objets mathématiques abstraits : les ensembles, les relations et les fonctions ; pour aller jusqu'à des représentations informatiques concrètes : les programmes. Finalement un certain nombre de techniques appliquées aux programmes sont présentées.

- **Des ensembles aux fonctions**.
  Cette partie montre le continuum qu'il existe entre les ensembles et les fonctions. Elle introduit les constructions usuelles: ensembles, tuples, enregistrements, listes, relations, associations et finalement fonctions. Différentes opérations sont également présentées.
- **Des fonctions aux programmes**.
  Cette partie définit les programmes comme étant des représentations informatiques de fonctions et présente succinctement quelques concepts usuels utilisés dans cette thèse.
- **Concepts et techniques relatifs aux programmes**.
  Les concepts d'interpréteur et de compilateur sont présentés ; tout comme des techniques très générales (décomposition séquentielle ou parallèle, calcul incrémental) ou plus spécifiques (spécialisation de programmes, découpage).

# A.1 Des ensembles aux fonctions

La plupart des notations  proviennent de formalismes existants. Pour comprendre les tableaux présentés ci-dessous mentionnons simplement quelques conventions : le symbole “=›” désigne une fonction totale, “–›” désigne une fonction partielle. Une expression de la forme ‘a dans une expression de type correspond à une variable d’ensemble comme dans le langage ML (elles sont quantifiées universellement).

## A.1.1 Ensembles

La théorie des ensembles est un point de départ classique. En plus des opérations standard nous introduisons le prédicat ‘one’ défini pour les singletons et retournant l’unique élément.

TABLEAU 20 Ensembles

| Nom | Symbole | Commentaires |
|---|---|---|
| Variable d’ensemble | ‘_ | Utilisé dans les expressions d’ensembles, ML |
| Ensemble vide | {} ou ∅ | {} ∈ Set<br>MATH: ∅, Z: {}, VDM: {} |
| Constructeur d’ensemble | ⊙ | (⊙) ∈ ‘a × {‘a} => {‘a}<br>Analogue au constructeur de liste ::, VDM: ⊙<br>a ⊙ ( b ⊙ ( c⊙ {} ) = {a;b;c} |
| Notation d’ensemble en extension | { _ ; _ ; ... } | Les éléments de l’ensemble sont séparés par des ‘;’. Il s’agit d’une notation équivalente _ ⊙ ( _ ⊙ { ... }) |
| Différence ensembliste | _ \ _ | (\) ∈ {‘a} × {‘a} => {‘a}<br>Z: \, VDM: - |
| Ensemble des parties | { _ } | ({}) ∈ ‘a => {‘a}<br>VDM: -set , Z: *P* |
| Appartenance | _ ∈ _ | (∈) ∈ ‘a × {‘a} => Bool |
| Inclusion d’ensemble | _ ⊆ _ | (⊆) ∈ {‘a} × {‘a} => Bool |
| Inclusion stricte | _ ⊂ _ | (⊂) ∈ {‘a} × {‘a} => Bool |
| Cardinalité | card _ | card ∈ {‘a} –› Int<br>Définie seulement pour les ensembles finis<br>Z: # |
| Elément unique | one _ | one ∈ {‘a} –› ‘a<br>one = λ s. if card s = 1 then {x} -> x<br>Fonction définie seulement pour les pour les singletons: retourne l’unique élément de l’ensemble. |
| Ensemble des booléens | Bool | Bool = {true,false} |
| Ensemble des entiers | Int | Int ∈ Set |
| Ensemble des caractères | Char | Char ∈ Set<br>Les caractères seront notés entre ‘ et ‘. Il seront omis |
| Ensemble des chaînes de caractères | Str | Str = [Char]<br>Les chaînes de caractères sont notées entre “ et “. |

### A.1.2 Extension vs. compréhension: énumération vs. formulation

Un *ensemble* peut être défini en ***extension*** ou en ***compréhension*** (On dit aussi en par ***énumération*** et en ***intention***). Dans le premier cas tous les éléments sont énumérés; ce n'est possible que pour les ensembles finis. Dans le deuxième cas on donne une ***formule*** ou expression devant être *interprétée* par le lecteur[1]. Nous utiliserons les notations informelles $\{\_\}_e$ et $\{\_\}_i$ pour désigner des ensembles définis respectivement en *e*xtension et en *i*ntention.

Décrire un ensemble en extension est intéressant parce que il est possible d'exprimer de manière concise et expressive des ensembles complexes. D'un point de vue informatique il peut être également plus intéressant de stocker une formule qu'un ensemble en énumération.

L'une des méthodes pour décrire un ensemble en compréhension est d'utiliser un *prédicat* (i.e une fonction définie sur un ensemble et retournant une valeur booléenne). L'*extension d'un prédicat sur un ensemble* est l'ensemble des éléments de celui-ci qui vérifient le prédicat (i.e. dont l'image est la valeur "vrai"). Cette fonction est notée "ext".

Plus généralement, dans cette thèse on appellera ***énumération*** l'opération permettant de passer d'un ensemble en compréhension à un ensemble en extension. Celle-ci n'est définie que pour des ensembles finis.

Inversement, on appellera ***formulation*** chaque opération permettant de passer d'un ensemble en extension à un ensemble en compréhension (i.e. à une formule). Cette opération n'est pas unique car comme il a été dit plusieurs formules peuvent être associées à un ensemble en extension. Certaines sont souvent jugées "meilleures" que d'autre. Parfois il existe même un optimum (figure 136). En pratique il n'est pas forcément rentable de rechercher cet optimum et l'on se contentera souvent d'une "bonne" solution.

On appellera ***reformulation*** le passage d'une formule à une autre formule équivalente.

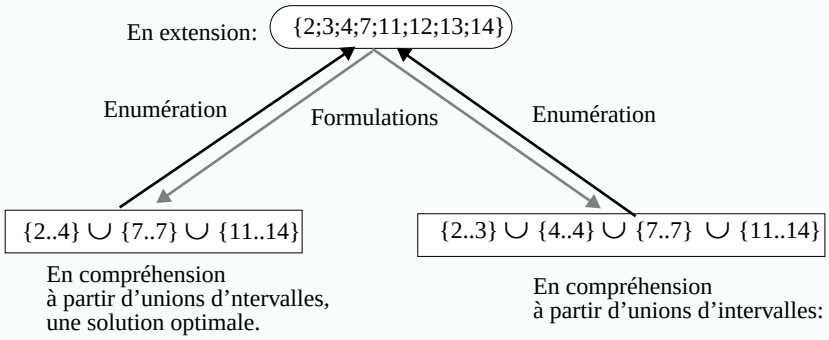
### A.1.3 Intervalles

Les ***intervalles*** sont définis sur des ensembles totalement ordonnés[2]. Remarquons simplement qu'un intervalle est un sous-ensemble et qu'il peut être représenté de manière compacte. Par exemple la notation en compréhension {3..7} pourra être interprétée comme étant l'ensemble {3;4;5;6;7}. Comme le suggère la figure 136, un ensemble quelconque d'entiers peut être décrit à partir d'une union d'intervalles.

Les bases de données temporelles font un usage intensif de ces objets car ils permettent de représenter de manière compacte un ensemble d'instants . Notons au passage que l'union de deux intervalles n'est pas nécessairement un intervalle.

1. Ici nous nous intéressons au point de vue informatique et c'est à un interpréteur qu'incombera cette tâche.
2. Un tel ensemble peut être fini ou non, discret ou non; les intervalles peuvent être ouverts ou fermés, etc. Il n'est pas nécessaire dans cette thèse d'entrer dans en détails.

*figure 136* Ensemble ordonné en extension / compréhension à partir d'intervalles




## A.1.4 Tuples et enregistrements

Les ***tuples*** sont les éléments de produits cartésiens d'ensembles. Il y a un nombre fixe de *composants* appartenant chacun à des ensembles éventuellement différents. L'ordre est important. Chaque composant est repéré par son indice, de gauche à droite, à partir de 1. Les ***enregistrements*** sont similaires aux tuples si ce n'est que chaque composant est nommé.

TABLEAU 21 Tuples et enregistrement

| Nom | Symbole | Commentaires |
|---|---|---|
| Produit cartésien | $\_ \times \_$ | |
| Ensembles de tuples | Tuples | L'ensemble des tuples |
| Constructeur de tuples | $( \_ , \ldots \_ )$ | $(,) \in$ ‘$a_1 \times \ldots$ ‘$a_n \Rightarrow$ ‘$a_1 \times \ldots$ ‘$a_n$<br>La virgule est réservée à la construction de tuples ou d'enregistrement. On utilise un ‘;’ pour séparer les éléments d'ensembles ou de listes. |
| Restructuration de tuples | $\#_{[i1; \ldots im]} \_$ | $(\#_{[i1; \ldots im]}) \in$ ‘$a_1 \times \ldots$ ‘$a_n \Rightarrow$ ‘$a_{i1} \times \ldots$ ‘$a_{im}$<br>Un nouveau tuple est formé en recopiant les valeurs des composants spécifiés dans l'ordre spécifié. Cet opérateur peut servir pour la projection de tuples mais il est aussi possible de spécifier plusieurs fois le même indice. Par facilité d'écriture $\#_i$ représente la valeur du $i^{eme}$ composant.<br>$\#_{[1;3;1;2]}$(‘a‘,‘b‘,‘f‘) = (‘a‘,‘f‘,‘a‘,‘b‘)<br>$\#_1$ (4,‘a‘) = 4 |
| Constructeur d'enregistrement | $( n_1 \text{ -> } \_ , \ldots n_m \text{ -> } \_ )$ | VDM: objet composite :: $n_1 : t_1 \ldots n_1 : t_1$ |

## A.1.5 Relations

Le concept de relation est utilisé dans différents domaines. Parfois il s'agit de relations binaires, parfois de relations n-aires. Ici le terme ***relation*** sera utilisé dans le deuxième cas. Une relation n-aire est un *ensemble* de tuples de longueur $n^1$. Chaque composant est numéroté de 1 à n, de gauche à droite.

En plus des opérations définies sur les ensembles deux opérations sont traditionnellement définies sur les relations: la projection ($\pi$) et la sélection ($\sigma$). La première permet d'éliminer des composants, la deuxième permet de "filtrer" les tuples vérifiant un prédicat donné.

TABLEAU 22 Relations

| Nom | Symbole | Commentaires |
|---|---|---|
| Relation | Rel | Rel = {'$a_1$ × '$a_2$... '$a_n$}<br>Ensembles des relations |
| Restructuration de relation | $\Psi_{[i1;...im]}$ _ | ($\Psi_{[i1;...im]}$) {'$a_1$ × '$a_2$ ... '$a_n$} => {'$a_{i1}$ × '$a_{i2}$ ... '$a_{im}$}<br>Cet opérateur permet de permuter, de dupliquer ou d'éliminer les composants. Il applique l'opérateur $\#_{[i1;...im]}$ à chaque élément de la relation.<br>$\Psi_{[i1;...im]}$ r = $\#_{[i1;...im]}$ (\| r \|)<br>$\Psi_{[3;1;1]}$ { '$a_1$× '$a_2$ × '$a_3$ } => ('$a_3$× '$a_1$ × '$a_1$ ) |
| Projection de relation | $\pi_{[i1;...in]}$ _ | ($\pi_{[i1;...im]}$) ∈ × {'$a_1$ × '$a_2$ ... '$a_n$} => {'$a_{i1}$ × '$a_{ij}$ ... '$a_{ij}$}<br>La projection est une restriction de l'opérateur $\Psi_{[i1;...im]}$ au cas où les indices peuvent être éliminés, permutés, mais pas dupliqués. Par simplification les crochets peuvent être omis pour une liste d'une seul élément. $\pi_i$ représente donc la projection du $n^{ieme}$ composant.<br>$\pi_{[i1;...im]}$ = $\Psi_{[i1;...im]}$ |
| Sélection de relation | $\sigma$_ _ | ($\sigma$) ∈ ({'$a_1$ × '$a_2$... '$a_n$} => Bool) =><br>{'$a_1$ × '$a_2$... '$a_n$} => {'$a_1$ × '$a_2$... '$a_n$}<br>Seul les tuples vérifiant le prédicat sont retenus. |

## A.1.6 Associations

Dans ce document le terme "***association***" sera utilisé comme synonyme de relations binaires[1] et il s'agit donc de relations particulières. Le fait que celles-ci soient binaires fait qu'une interprétation particulière peut leur être donnée: le premier élément est associé au deuxième. Un ordre est donc implicite. Une notation particulière est utilisée pour définir une association, à la place d'écrire {D × C} on préférera D «–» C. De même les éléments des associations sont appelés ***maplets*** et une notation spéciale est utilisée: 1->2 à la place de (1,2)[2]. L'ensemble D est appelé domaine alors que l'ensemble C est appelé codomaine[3].

### *Association et relation n-aire...*

En fait les ensembles peuvent eux mêmes être des produits cartésiens d'ensembles. Il n'y a alors pas de différences entre une relation n-aire et une association si ce n'est que l'ensemble des composants a été "coupé" en deux: certains composants font partie du domaine, les autres font

1. ou un ensemble d'enregistrement, selon les modèles de données. Cela revient au même: dans le premier cas les composants sont repérés par des numéros, dans le deuxième cas par des noms. La définitions des opérations est analogue dans les deux cas, aux problèmes de renommage et de changement d'indices prêt. Par exemple certaines algèbres proposent un opérateur "renommer" dans le cas de relations définies sur des enregistrements.

1. Dans le langage Z le terme "relation" est utilisé pour des relations binaires. C'est souvent le cas aussi en mathématique.
2. Ces distinctions sont faites dans les langages de spécifications Z et VDM où le terme "maplet" est utilisé autant en français qu'en anglais. Comme dans Z, le symbole -> est utilisé même si plusieurs éléments sont en correspondance. Par exemple {1->2; 1->4} est une association (mais n'est pas une fonction). Dans VDM cette notation est réservée au correspondance (ici les fonctions).
3. Pour être plus exact il s'agit plutôt du "domaine de définition" et du "codomaine de définition". Il s'agit du domaine et du codomaine seulement dans le cas de fonction totales. En pratique cet abus de langage est souvent fait.

partie du codomaine. Par exemple X × Y «–» Z est vu comme une association, mais c'est aussi une relation tertiaire de la forme { X × Y × Z }. Un élément de cette association est par exemple (1,3)->2 qui est aussi un triplet (1,3,2).

Dans cette thèse nous introduisons un opérateur général, noté $_{[i1;...il]}\Psi_{[i1;...im]}$ afin de passer facilement d'une relation à une association. Chaque liste d'indices correspond respectivement à la liste des composants faisant partie du domaine et du codomaine. Par exemple à partir de la relation tertiaire Mariage = { Homme × Femme × Date } il est possible de définir les associations suivantes :

$_{[1;2]}\Psi_{[3]}$ Mariage = DateDeMarriageDe = Homme × Femme «–» Date
$_{[1]}\Psi_{[2;3]}$ Mariage = S'estMariéAvecLe = Homme «–» Femme × Date
$_{[2]}\Psi_{[1]}$ Mariage = MariDe = Femme «–» Homme
$_{[2]}\Psi_{[3]}$ Mariage = MariéeLe = Femme «–» Date

*Cet opérateur permet de faire le lien entre les opérateurs des modèles relationnels définis dans le cadre des bases de données et les opérateurs spécifiques aux associations et fonctions provenant typiquement des langages de spécifications ensemblistes et des langages fonctionnels.*

### *Opérateurs spécifiques aux associations...*

Les opérateurs définis sur les relations (et donc sur les ensembles) peuvent êtres appliquées aux associations. C'est le cas par exemple de l'union, de l'intersection, de la projection, etc. Par contre le résultat de ces opérateurs n'est pas nécessairement une association. Par exemple le produit cartésien de deux associations est une relation, pas une association[1].

En plus des opérateurs définis sur les ensembles et les relations, des opérateurs spécifiques sont définis. Les opérateurs présentés ci-dessous sont "classiques" dans les langages de spécifications ensemblistes (dom,rng,.$^{-1}$,◁,▷,⩤,⩥,img,o,etc.).

TABLEAU 23 Associations

| Nom | Symbole | Commentaires |
|---|---|---|
| Constructeur d'associations | - «–» - | («–») ∈ {'a} × {'b} ⇒ { 'a x 'b }<br>'a «–» 'b est équivalent à {'a × 'b}<br>(Analogue à Z où X ↔ Y est équivalent à P( X × Y) )<br>Z: ↔ |
| Association à partir d'une relation | $\_\Psi\_\ \_$ | $([i1;...il]\Psi_{[i1;...im]}) \in \times$ {'$a_1$ × '$a_2$ ... '$a_n$} ⇒<br>'$a_{i1}$ × '$a_{i2}$ ... '$a_{il}$ «–» '$a_{i1}$ × '$a_{i2}$ ... '$a_{im}$<br>Cet opérateur permet d'obtenir une association à partir d'une relation n-aire. Les indices peuvent être dupliqués, supprimés, permutés.<br>$_{[4]}\Psi_{[3;2]}$ { '$a_1$× '$a_2$ × '$a_3$ × '$a_4$ } => ( '$a_4$ «–» '$a_3$ × '$a_2$ ) |
| Domaine | dom | dom ∈ 'a «–» 'b ⇒ {'a}<br>dom f ⊆ ddom f<br>si dom f = ddom f alors c'est une fonction totale, sinon c'est une fonction partielle |

1. Nous définissons les opérateurs $\times_{dom}$ et $\times_{rng}$ pour pallier à ce problème.

TABLEAU 23 Associations

| Nom | Symbole | Commentaires |
|---|---|---|
| Codomaine | rng | rng ∈ ‘a «—» ‘b => {‘b}<br>rng f ⊆ drng f<br>VDM, Z: ran |
| Domaine de définition | ddom | ddom ∈ ‘a «—» ‘b => {‘a}<br>L’ensemble apparaissant dans la définition de l’association appelée ‘source’ en Z |
| Codomaine de définition | drng | drng ∈ ‘a «—» ‘b => {‘a}<br>L’ensemble apparaissant dans la définition de l’association appelé ‘target’ en Z |
| Inverse d’une association | $\_^{-1}$ | $(^{-1}) \in$ (‘a «—» ‘b) => (‘b «—» ‘a)<br>$f^{-1} = {}_{irng(f)}\Psi_{idom(f)}\, f$<br>VDM: $^{-1}$ Z: $^{-1}$<br>$\{ 1\text{->}2; 1\text{->}4; 3\text{->}2\}^{-1} = \{2\text{->}1; 4\text{->}1; 2\text{->}3\}$ |
| Restriction du domaine | _ ◁ _ | (◁) ∈ {‘a} × (‘a «—» ‘b) => (‘a«—» ‘b)<br>La fonction est restreinte aux éléments du domaine appartenant à l’ensemble spécifié.<br>s◁ f = { x -> y \| x ∈ s ∧ (x->y) ∈ f }<br>mathématique ⎡f , VDM: ◁, Z: ◁<br>{1;3;5} ◁ { 1->2; 2->4; 3->9} = {1->2; 3->9} |
| Restriction du codomaine | - ▷ - | (▷) ∈ (‘a «—» ‘b) × {‘b} => (‘a «—» ‘b)<br>VDM: ▷, Z: ▷ |
| Suppression de domaine | _ ⩤ _ | (⩤) ∈ {‘a} × (‘a «—» ‘b) => (‘a «—» ‘b)<br>VDM: ⩤,Z ⩤ |
| Suppression de codomaine | _ ⩥ _ | (⩥) ∈ (‘a «—» ‘b) × {‘a} => (‘a «—» ‘b)<br>VDM:⩥, Z:⩥ |
| Image d’un ensemble | img _ _<br>_ (\| _ \|) | img ∈ (‘a «—» ‘b) × {‘a} => {‘b}<br>img f s = f(\| s \|) = rng( s ◁ f )<br>Z: (\| _ \|) |
| Composition d’association | _ ; _<br>_ o _ | (;) ∈ (‘a «—» ‘b) × (‘b «—» ‘c) => (‘a «—» ‘c)<br>(o) ∈ (‘b «—» ‘c) × (‘a «—» ‘b) => (‘a «—» ‘c)<br>f o g = g ; f<br>VDM: pas Z: ; MATH: o |
| Produit cartésien sur le domaine | $\_ \times_{dom} \_$ | $(\times_{dom}) \in$ (‘d «—» ‘c ) × {‘$d_1$} =><br>(‘d × ‘$d_1$ «—» ‘c )<br>$f \times_{dom} s = (f^{-1} \times_{rng} s)^{-1}$<br>Le produit cartésien est fait avec le domaine.<br>$\{ 1\text{->} 2 ; 3\text{->}3 \} \times_{dom} \{ true; false \} =$<br>{(1,true)->2; (1,false)->2; (3,true)->3; (3,false)->3} |

TABLEAU 23 Associations

| Nom | Symbole | Commentaires |
|---|---|---|
| Produit cartésien sur le codomaine | $\_ \times_{rng} \_$ | $(\times_{rng}) \in (`d \leftarrow\!\!\!\!\circ\!\!\!\!\rightarrow `c) \times \{`c_1\} \Rightarrow (`d \leftarrow\!\!\!\!\circ\!\!\!\!\rightarrow `c \times `c_1)$<br>$f \times_{rng} s = {}_{idom(f)}\Psi_{irng(f)} \circ (map\ (\lambda\ x.\ x + ilen(f))\ irel(s))\ (f \times s)$<br>Le produit cartésien est fait avec le codomaine, i.e. est ajouté au codomaine.<br>{ 1->2; 3->3 } $\times_{rng}$ { true; false } =<br>{1->(2,true) ; 1->(2,false); 3->(3,delete); 3->(3;false) } |
| Projection du domaine | $\pi_{dom(i)}\ \_$ | $\pi_{dom,1} \in (d_1 \times `d_2 \rightarrow `c) \rightarrow (`d_1 \leftarrow\!\!\!\!\circ\!\!\!\!\rightarrow `c)$<br>$\pi_{dom,1}\ a\ x_1 = a\ x_1$<br>$\pi_{dom,1 = idom(f)}\Psi_i$<br>$\pi_{dom,1}$ { (1,3) -> a ; (1,4) -> a ; (2,3) -> b ; (2,4)-> b }<br>= { 1 -> {a}, 2 -> {b} } |

Il est intéressant de pouvoir spécifier la cardinalité d'une association. Une fonction peut être partielle ou totale. Ce peut être une injection, surjection, ou bijection, etc. Ici nous utilisons un notation intuitive inspirée de Z0 pour spécifier ces contraintes. Une double flèche » indique une fonction multivaluée alors que › indique une fonction monovaluée. Le corps de la flèche indique si la fonction est partielle (–) ou totale (=). On pourra omettre l'une des deux directions, qui dans ce cas sera considérée comme étant une fonction partielle et multivaluée (cas le plus général). A titre d'exemple «–» est une relation quelconque, et est équivalent à la notation –» ou «–. Une fonction partielle sera notée –› alors qu'une fonction totale sera notée =›. Injections, surjections et bijections se noterons respectivement «==›, ‹–=›, ‹==›. Nous ne proposons pas de notation différentiant les fonctions finies des fonctions infinies comme c'est la cas dans VDM ou Z. De toutes façon nous nous intéressons généralement à des fonctions finies.

## A.1.7 Fonctions

Comme dans Z les ***fonctions*** sont des cas particuliers des associations. Les opérateurs définis sur les associations peuvent être appliqués aux fonctions. Le résultat n'est cependant pas nécessairement une fonction. Par exemple l'union de deux fonctions n'est définie que si les domaines correspondants sont différents. L'opérateur de surcharge ⊕ est défini à cet effet. Le produit cartésien sur le domaine d'une fonction ($\times_{dom}$) est une fonction.

TABLEAU 24 Fonctions

| Nom | Symbole | Commentaires |
|---|---|---|
| Constructeur de fonction partielle | - –› - | $(\rightarrow) \in `a \times `b \Rightarrow (`a \rightarrow `b)$ |
| Constructeur de fonction totale | - =› - | $(\Rightarrow) \in `a \times `b \Rightarrow$ Fun |
| Application duale | $\_ \#\ \_$ | $(\#) \in (`a \rightarrow `b_1) \times (`a \rightarrow `b_2) \Rightarrow (`a \rightarrow `b_1 \times `b_2)$<br>(f # g) x = (f(x),g(x))<br>[Macl83]: # |

TABLEAU 24 Fonctions

| Nom | Symbole | Commentaires |
|---|---|---|
| Application parallèle | _ ‖ _ | (#) ∈ (‘$a_1$ «–» ‘$b_1$) × (‘$a_2$ «–» ‘$b_2$) => (‘$a_1$ × ‘$a_2$ «–» ‘$b_1$ × ‘$b_2$)<br>(f ‖ g)(x,y) = (f(x),g(y))<br>[Macl83]: ‖ |
| Application d’une fonction | _ ( _ ) | () (‘a –> ‘b) × ‘a –> ‘b |
| Surcharge de fonction | _ ⊕ _ | (⊕) ∈ (‘a –> ‘b) × (‘a –> ‘b) => (‘a –> ‘b)<br>f ⊕ g = ((dom g) ⩤ f) ∪ g<br>VDM: † , Z: ⊕ |
| Fonction identité | Id _ | Id = λ a.a ∈ ‘a => ‘a |

## A.1.8 Fonction en extension ou en compréhension

Tout comme les ensembles, les fonctions peuvent naturellement être définies en extension ou en compréhension. Par exemple la restriction de la fonction carré sur l’intervalle {1..3} sera écrite en extension { 1 -> 2; 2 -> 4; 3 -> 9}.

Pour les fonctions en compréhension nous utiliserons aussi la ***lambda-notation*** issue du λ-calcul (i.e “λx•expression” dénote la fonction traditionnellement définie par f(x)=expression) et celles proposées dans le langage ML (par exemple “fun x -> expression”). Rappelons simplement que l’application de fonction est associative à gauche alors que l’opérateur –› est associatif à droite. Autrement dit l’expression “f x y” est équivalente à “((f x) y)” et l’expression de type “ ‘a -> ‘b -> ‘c ” est équivalente à “ ‘a -> (‘b -> ‘c)“.

Les facilités d’écriture telle que l’expression conditionnelle et le “pattern matching” seront également utilisés.

## A.1.9 Listes

Les ***listes*** sont des structures apparaissant très largement en pratique. Ces structures peuvent être représentées comme des ensembles (par exemple dans Z une liste est une fonction définie sur des entiers). Cependant avoir des opérations définies sur des listes peut simplifier considérablement l’expression de structures informatiques. Dans cette thèse nous considérerons des listes homogènes, c’est à dire dont tous les éléments sont du même types (alors que les tuples sont hétérogènes mais de longueur fixe).

TABLEAU 25 Listes

| Nom | Symbole | Commentaires |
|---|---|---|
| Constructeur de listes | [ _ ] | ([]) ∈ ‘a => [ ‘a ]<br>VDM: [], Z: <>, LISP: {}, HASKELL: [] |
| Liste vide | [] | [] ∈ []<br>LISP: (), VDM: [], Z: <> |
| Constructeur de listes | _ :: _ | (::) ∈ ‘a × [‘a] => [‘a]<br>LISP: cons, HASKEL: :, CAML: :: |
| Notation de liste | [ _ ; _ ... ] | Les éléments des listes sont séparés par des ‘;’ |

TABLEAU 25 Listes

| Nom | Symbole | Commentaires |
|---|---|---|
| Concaténation | _ @ _ | (@) ∈ [‘a] × [‘a] ⇒ [‘a]<br>VDM: Z: flèche courbe |
| Premier élément | hd _ | hd ∈ [‘a] → ‘a<br>VDM: hd, LISP: car , Z: head, HASKELL: head |
| Reste de la liste | tl _ | (×) ∈ [‘a] ⇒ ‘a<br>VDM: tl, LISP: car, Z: tail, HASKELL: head |

## A.1.10Factorisation vs. développement

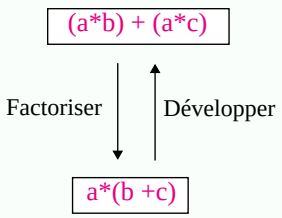
Dans cette section nous nous attardons plus longuement sur un ensemble d’opérations que l’on appellera “*factorisation*” et “*développement*”. Il s’agit en fait d’une généralisation des opérations d’imbrications des modèles relationnels imbriqués (“nest” et “unnest”) . Dans le domaine des bases de données temporelles, on trouve aussi les opérations plier et déplier (“fold” et “unfold” . Parfois un opérateur “restructure” remplit des fonctions analogues .

*Factoriser* c’est transformer une expression dans l’intention de maximiser les parties communes.

#### *Factoriser dans le cadre des mathématiques...*

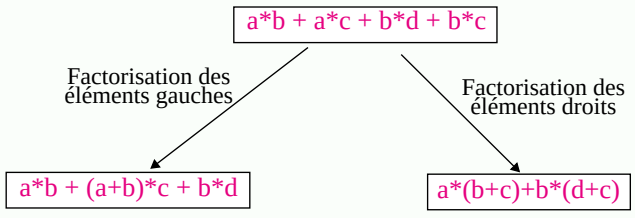
Le terme vient des mathématiques. Factoriser est transformer une expression en produit de facteurs [Laro66]. La figure 137.a présente un exemple trivial de factorisation. Le contraire de factoriser c’est *développer* (ou distribuer). Bien évidemment ces transformations ne sont pertinentes que si les expressions sont équivalentes, c’est à dire dans le cas (a), si l’opération “*” est distributive par rapport à l’opération “+”. Si les opérateurs sont commutatifs et/ou associatifs il y a bien plus d’opportunités pour factoriser une expression que s’ils ne le sont pas. Remarquons finalement que plusieurs factorisations sont possibles pour une même expression (figure 137.b).

*figure 137* Factoriser et développer des expressions




***(a)* Factoriser vs. développer**

Après la factorisation “a” n’est plus dupliqué dans l’expression.




***(b)* Plusieurs factorisations possibles**

On suppose que l’opérateur “+” est commutatif mais que “*” ne l’est pas. Il est possible de factoriser les éléments gauches du produit ou les éléments droits du produit.

#### *Factoriser dans le cadre du modèle relationnel imbriqué...*

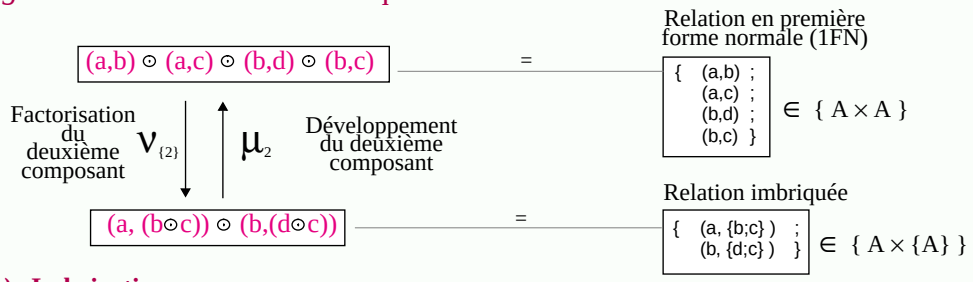
Généralement le modèle relationnel n’autorise que des relations en première forme normale, c’est à dire des relations dont les composants sont des valeurs atomiques. Les modèles relationnels imbriqués relâchent cette contrainte et permettent d’avoir des ensembles voir des relations comme

attribut (d'où le terme imbriqué) et ce de manière récursive. La structure de chaque tuple doit néanmoins être régulière, c'est à dire que tous les tuples sont du même type. Par exemple { (a,{1;3}); (c,{3}); (e,{}) } est une relation de type { Char × {Int} }. Par contre { (a,2); (b,{4,5}) } n'est pas une relation imbriquée.

L'intérêt des relations imbriquées est bien évidemment de pouvoir représenter plus naturellement certaines informations. Par exemple une personne ayant plusieurs numéro de téléphones et plusieurs enfants pourra naturellement être représentée à partir d'un triplet Nom × { NumTelephone } × { Enfant }.

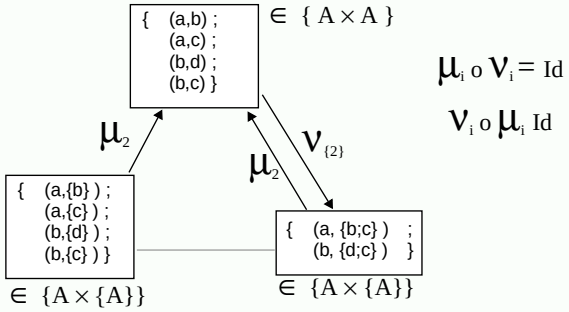
L'autre avantage d'un tel modèle de données et que grâce à des opérations de factorisation et de développement la duplication d'information peut être réduite et les relations peuvent être stockées de manière plus compacte. Deux opérations sont traditionnellement définie: l'imbrication ("nest" notée ν ) et la "désimbrication" ("unnest" notée μ). La première correspond à ce nous appelons dans cette thèse la factorisation et la seconde au développement.

*figure 138* Relations imbriquées



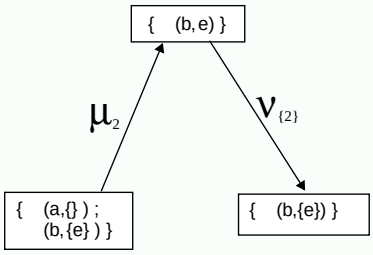

***(a)* Imbrication**

L'opérateur de construction d'ensemble "⊙" est commutatif. L'opérateur de construction de tuples "," n'est pas commutatif.




***(b)* Relation entre μ et ν**

Comme le montre l'exemple ci-dessus la séquence développement puis factorisation n'est pas l'identité. Par contre en factorisant puis en développant on obtient toujours le même résultat .




***(c)* Valeur nulle**

Lorsque l'on développe une relation, il est possible de "perdre" de l'information. Par exemple dans le cas ci-dessus, l'information "a" est perdue. Pour éviter ce problème certains modèles introduisent des "valeurs nulles" dans les modèles de données.

# A.2 Des fonctions aux programmes

Un ***langage*** est un ensemble d'*expressions* construites à partir d'un vocabulaire ou ***lexique*** de base en utilisant des règles de structuration déterminée par la *syntaxe* du langage. Dans cette thèse nous nous intéressons à un classe particulière de langage: les langages de programmation.

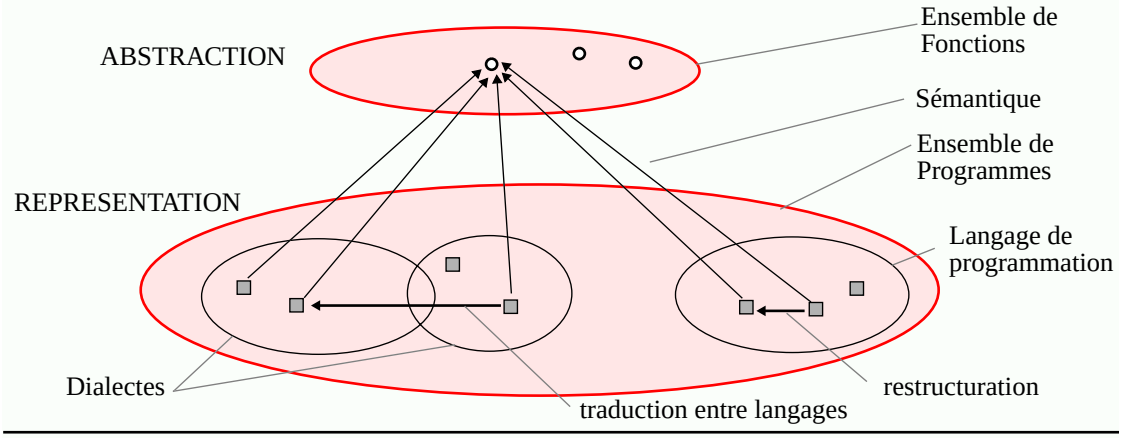
## A.2.1 Programmes et langages de programmation

Une ***fonction*** est un objet mathématique. Il s'agit d'un objet abstrait.

Un ***programme*** est la représentation informatique d'une fonction exprimée dans un *langage de programmation*.

La sémantique d'un langage associe à un programme donné une seule fonction: sa dénotation . Inversement plusieurs programmes peuvent avoir la même dénotation. Ils seront alors dits ***équivalents***. En général l'équivalence de deux programmes n'est pas décidable. Par contre on peut définir des transformations de programmes préservant la sémantique. Il s'agit justement de problèmes de restructuration.

*figure 139* Fonctions, Programmes et Sémantique




L'ensemble des fonctions qui peuvent être décrites à partir d'un langage de programmation détermine le pouvoir d'expression de ce langage. Dans le cadre de cette thèse il n'est pas fait d'hypothèse sur cet ensemble. Le terme "programme" est utilisé pour toute représentation d'une fonction. Sous l'appellation "langage de programmation" seront rassemblés des langages permettant d'exprimer autant des fonctions calculables générales que des classes de fonctions très spécifiques (Chapitre II).

Des langages de programmation différents permettent d'exprimer des programmes correspondant à une même fonction. Dans certains cas certains langages de programmations sont très proches et l'on parle alors de *dialectes*. Eventuellement un même programme peut faire partie de plusieurs langages de programmation.

## A.2.2 Classes de langages de programmation

Différentes taxonomies ont été proposées au cours du temps pour classer les langages de programmation en fonction des mécanismes sur lesquels ils reposent. Nous distinguons trois classes principales: les langages impératifs, les langages applicatifs et les langages déclaratifs.

### A.2.2.1 *langages impératifs*

Les *langages impératifs*[1] se caractérisent généralement par le concept de machine à état, d'instructions, d'affectation et de contrôle explicite. Un programme spécifie une séquence d'instructions modifiant l'état d'une machine.
Des supports d'exécution efficaces sont disponibles pour les programmes impératifs. Par contre élaborer de tels programmes implique une description opératoire et explicite de l'algorithme de calcul. De ce point de vue, les programmes impératifs sont moins abstraits que les programmes déclaratifs ou applicatifs. La difficulté rencontrée pour décrire la sémantique de ces langages est aussi un autre inconvénient.

### A.2.2.2 *langages fonctionnels*

Les *langages applicatifs* (ou *langages fonctionnels*) sont basés sur l'application de fonction comme technique basique de calcul. Les concepts manipulés sont proches des mathématiques. Il est donc facile d'en définir la sémantique et d'opérer des traitements automatiques sur les programmes applicatifs. Hélas l'exécution de tels programmes et généralement moins efficace que dans le cas de programmes impératifs.

### A.2.2.3 *langages déclaratifs*

Les *langages déclaratifs* se caractérisent par le fait qu'ils permettent d'exprimer les relations existant entre différents éléments sans pour autant décrire explicitement comment maintenir la validité de ces relations[2]. Typiquement les langages déclaratifs sont basés sur l'expression de règles. Les programmes déclaratifs ne décrivent pas comment obtenir un résultat. Le mécanisme d'exécution a la charge de proposer des techniques de résolutions.
La différence entre les données et les programmes n'est pas nette dans le cas des langages déclaratifs.
La facilité d'expression est un avantage majeur des langages déclaratifs. L'inconvénient et que l'exécution des programmes est généralement beaucoup moins efficace que dans le cas de langages impératifs ou applicatifs.

### A.2.2.4 *Langages hybrides*

Les trois classes de langages présentées ne sont pas disjointes. Différents langages combinent les caractéristiques présentées ci-dessus. Des constructions impératives sont souvent introduites dans les langages applicatifs ou déclaratifs pour des raisons de performance.

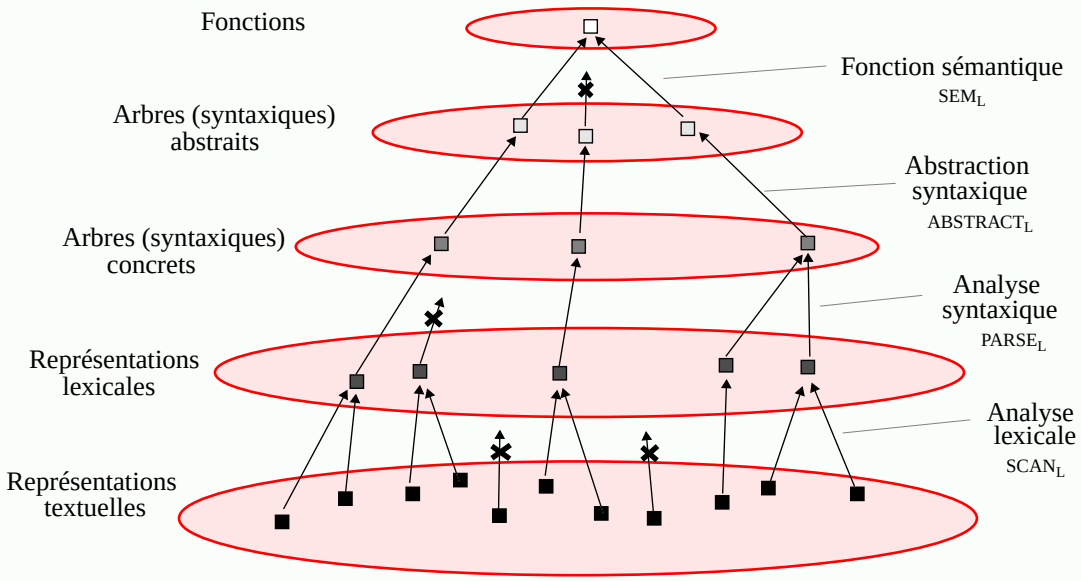
---

1. Ces langages sont parfois appelés langages procéduraux.
2. Tout au long de cette thèse, il est dit qu'un programme était la représentation concrète d'une fonction. Dans le cas de langages déclaratifs, il s'agit plus exactement d'une relation (Une question peut donner lieu à plusieurs réponses).

### A.2.3 Texte source, analyse lexicale, syntaxe concrète et syntaxe abstraite

Pour définir un langage de programmation ou manipuler des programmes il est utile de considérer différents niveaux plus où moins abstraits. Ces niveaux sont représentés dans la partie inférieure de la figure 140[1].

*figure 140* Différents niveaux




Un programme est la concrétisation d'une fonction donnée. Cette figure montre différents niveaux d'abstraction. Les flèches indiquées ✖ correspondent aux programmes incorrects. La représentation la plus concrète est la représentation textuelle. L'abstraction maximum est la fonction en tant qu'objet mathématique.

Bien qu'il y soit souvent associé ce schéma n'est pas limité aux langages de programmation mais peut s'appliquer à d'autre type de langages.

- *Représentation textuelle*. La représentation la plus concrète pour un programme est traditionnellement une séquence de caractères. L'avantage évident de cette représentation est qu'elle est linéaire et facile à manipuler. On parlera de ***représentation textuelle*** ou de ***texte source*** le terme "texte" prenant alors tout son poids[2].
- *Représentation lexicale*. L'analyse lexicale permet de faire abstraction d'un certain nombre de détails, typiquement les commentaires, les espaces et les tabulations. Elle permet de représenter le logiciel sous forme d'une séquence de lexèmes. On parlera de ***représentation lexicale***. Cette représentation n'est que rarement utilisée explicitement . L'***analyse lexicale*** est la transformation permettant de passer de la représentation textuelle à la représentation lexicale. Cette fonction sera notée $SCAN_L$ dans ce document.
- *Arbre syntaxique concret*. La ***syntaxe concrète*** d'un langage est exprimée grâce à une ***grammaire concrète***. La représentation correspondante sera appelé ***arbre syntaxique concret***. La transformation permettant de passer de la représentation lexicale à un arbre syntaxique

1.

2. Par contre les termes "*code source*" et "*programme source*" ne font pas explicitement référence à cette représentation particulière mais seulement au fait que .

concret est appelée ***analyse syntaxique***. Elle sera notée $PARSE_L$.

- *Arbre abstrait*. La ***syntaxe abstraite*** permet de représenter un programme à un niveau d'abstraction plus élevé. Le formalisme utilisé pour définir une telle syntaxe est une ***grammaire abstraite*** et la représentation correspondante d'un programme est un arbre syntaxique abstrait ou plus simplement un ***arbre abstrait***. L'intérêt de ce niveau est qu'il permet de faire abstraction du "sucre syntaxique": séparateurs de listes, délimiteurs, etc. La transformation permettant de passer d'un arbre syntaxique concret à un arbre syntaxique abstrait sera appelé ***abstraction syntaxique*** dans ce document et notée $ABSTRACT_L$.

Même si dans beaucoup d'outils d'analyse ces différents niveaux ne sont pas séparés explicitement, faire la distinction est important pour comprendre les différentes approches de manipulation de programmes. Par exemple il est intéressant de classer les outils de calcul de différences entre deux programmes en fonction de la représentation utilisée: textuelle, lexicale, syntaxique, ou sémantique.

Remarquons aussi que l'analyse lexicale et l'analyse syntaxique retournent des erreurs (respectivement lexicales et syntaxiques) si les programmes ne sont pas corrects. On parlera alors de programmes ***lexicalement corrects*** et ***syntaxiquement corrects***.

Naturellement, plusieurs représentations concrètes peuvent correspondre à une même représentation abstraite. La figure 140 (p.293) illustre cette propriété.

Les différents niveaux présentés ci-dessus ne décrivent que l'aspect *structurel* d'un programme. Les aspects *sémantiques* sont présentés dans les sections suivantes

# A.3 Concepts et techniques relatifs aux programmes

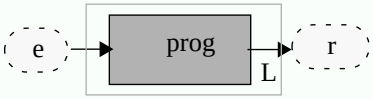
## A.3.1 Conventions graphiques

Afin d'illustrer graphiquement différentes notions (dont celle d'interpréteur, de compilateur, de mixeur, etc.), nous utiliserons souvent des figures. Les conventions graphiques utilisées sont présentées dans la figure 142 (p.296). Bien qu'elle puissent paraître superflues dans le cas de figures simples, elles deviennent *essentielles* pour interpréter *avec précision* des structures complexes.

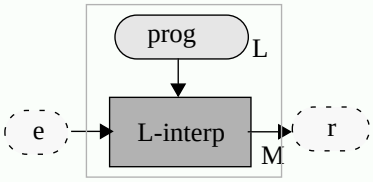
## A.3.2 Interpréteur

Un programme est la représentation informatique d'une fonction. Mais pour qu'un langage de programmation soit exécutable ceci n'est pas suffisant: un interpréteur est nécessaire.
La figure 143 présente les différentes représentations possibles.

*figure 141* Programmes et programme interprétés




**(a) Un programme en langage L (vision abstraite)**




**(b) Un programme interprété**


**(c) Un programme en langage L**

Ces trois représentations sont équivalentes. La première est la plus abstraite. On s'intéresse plus à la fonction dénotée qu'à sa représentation. Dans le cas (b) l'interpréteur est explicitement représenté. La figure (b) est utilisée dans le cas de programme généré. Elle est basée sur la dualité programme/fonction.

Concrètement un ***interpréteur*** peut être:
- un humain. Dans ce cas on utilisera plutôt le terme ***interprète***[1]. Le calcul de la fonction représentée par le programme se fait donc "manuellement", et n'est pas nécessairement déterministe (figure 143.a).
- une machine physique (figure 143.b).
- un programme lui même interprété (figure 143.c). On a alors plusieurs niveaux d'interprétations.

Informellement un interpréteur retourne un résultat à partir d'un programme et d'une entrée (figure 141.b). Evidement la fonction ainsi calculée doit être la fonction représentée par le

1. Interpréteur est un terme informatique alors qu'interprète est un terme courant de la langue française.

*figure 142* Conventions graphiques

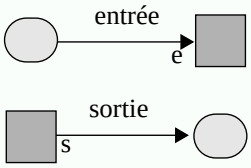

***(a)* Données**

La lettre, si elle est spécifiée, précise le formalisme dans lequel est représentée la donnée. La différence entre un programme (b) et une donnée quelconque (a) ne sera faite que lorsque nécessaire.

***(b)* Programme vu comme une donnée**

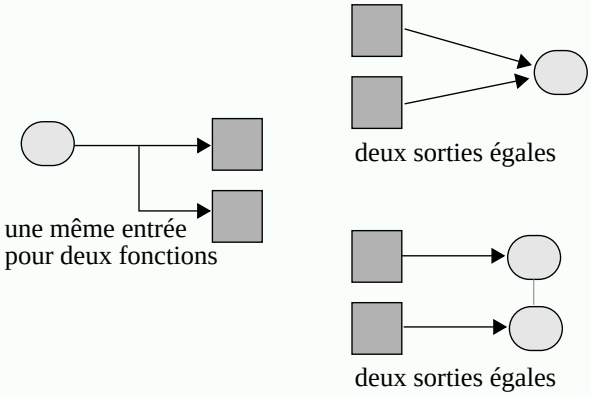
***(c)* Programme vu comme une fonction**

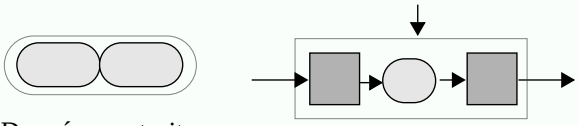
Les programmes peuvent à la fois être vus comme des données (b) où comme des fonctions (c). Le langage de programmation sera mentionné ou non. Le symbole  est utilisé lorsque c'est un agent humain qui est supposé interpréter la donnée ou exécuter la fonction. Inversement le symbole  indique que l'agent est une machine physique, par exemple un micro-processeur, ou une machine abstraite, mais en tout cas que la transformation est essentiellement automatique (Même si c'est un agent humain qui fournit les entrées et interprète les sorties).




Les entrées et les sorties peuvent être nommées. Une flèche ne peut connecter qu'un carré à un rond ou un rond à un carré. Dans le premier cas il s'agit d'une sortie dans le deuxième cas d'une entrée.

Deux flèches provenant du même rond et arrivant sur deux carrés est naturellement une entrée pour les deux fonctions. Par contre, par convention si deux flèches arrivent sur le même rond il s'agit de deux sorties égales. Normalement, un trait fin reliant deux carrés ou deux ronds indique l'égalité de ceux-ci.



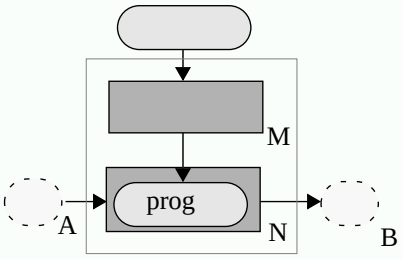




Un cadre autour de rond ou de carrés est une donnée ou une fonction construite. Dans le deuxième cas une flèche traversant le cadre précise quelle est la fonction utilisant cette donnée. Une flèche arrivant sur le cadre ne précise pas sa destination.

Dans l'exemple (d) le programme "prog" est à la fois vu

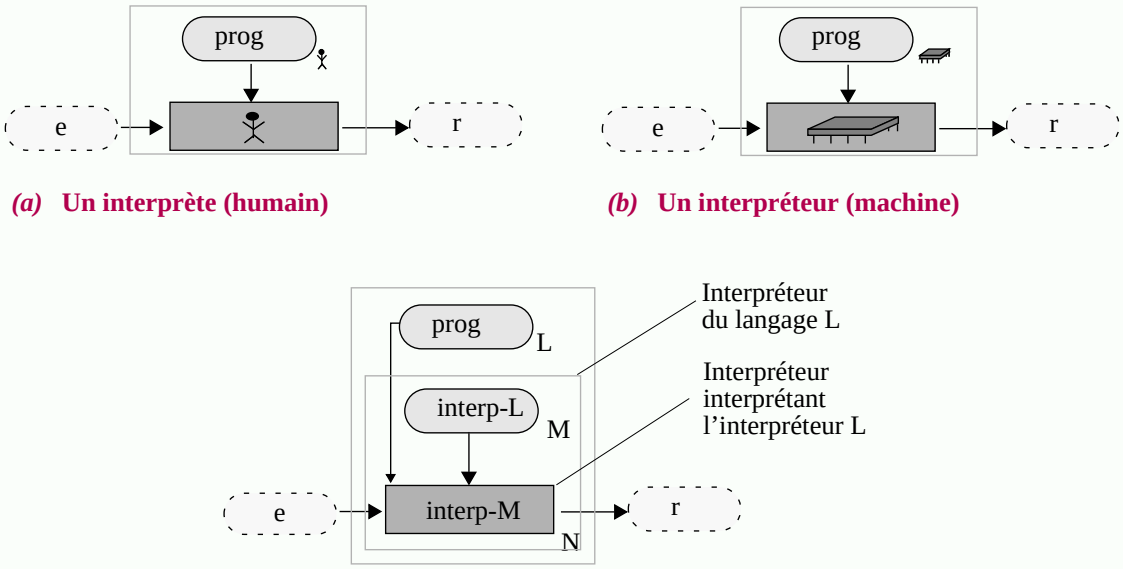
- comme une donnée (car il est généré par une fonction)
- comme une fonction prenant en entrée une donnée exprimée dans un formalisme A et retournant en sortie une donnée exprimée dans un formalisme B.

Remarquer le point de connections des flèches qui indique s'il s'agit d'une entrée ou d'une sortie. De telles conventions sont fort utiles pour interpréter des figures complexes




***(d)* Exemple**

*figure 143* Interprètes et interpréteurs




*(a)* **Un interprète (humain)**

*(b)* **Un interpréteur (machine)**

*(c)* **Plusieurs niveaux d'interprétation**

programme. Plus formellement les interpréteurs de langage L implémentés en langage M sont caractérisés par :

$$\text{interp est un interpréteur-L-en-M} \Leftrightarrow \forall \text{ prog} . \forall \text{ e. L( prog ) e = M ( interp ) (prog,e)} \quad \text{(EQ 1)}$$

Sous cette condition on passera de la figure 141.a à la figure 141.b librement selon que l'on s'intéresse seulement au programme en tant que fonction ou au contraire si l'on souhaite mettre en évidence son interprétation.

L'intérêt d'un interpréteur-L est bien évidemment qu'il permet d'exécuter un programme en L. Ils permettent d'augmenter le niveau d'abstraction dans lequel les programmes doivent être écrits.
Un des inconvénients des interpréteurs et qu'aucune erreur ne peut être détectée avant l'exécution réelle du programme. Remarquons que découvrir une erreur en cours d'exécution peut avoir des répercutions catastrophique.Un autre inconvénient est qu'ils ne sont généralement pas efficaces. A chaque exécution le programme est interprété de nouveau.

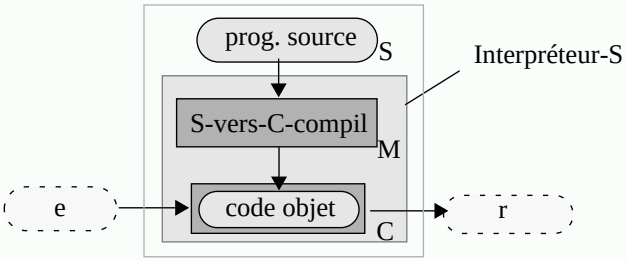
## A.3.3 Compilateur

Un ***compilateur*** permet de traduire un programme donné (le programme source) exprimé en langage source S en un programme équivalent exprimé en langage cible C, (le programme objet, ou code objet). Plus formellement un compilateur implémenté en langage M est caractérisé par:

$$\text{compil est un compilateur-S-vers-C-en-M} \Leftrightarrow \forall \text{ prog} . \forall \text{ e. sem-S( prog ) e = sem-C( sem-M(compil) prog ) (prog,e)} \quad \text{(EQ 2)}$$

.

*figure 144* Compilateur




Grâce à un compilateur-S-vers-C il est possible d'exécuter des programmes en langage S à l'aide d'interpréteurs-C. Cette possibilité est intéressante si S est un langage plus adapté à l'expression des problèmes (par exemple un langage plus abstrait). Compiler peut également être intéressant si l'on dispose d'un interpréteur-C efficace. C'est le cas par exemple de langages de programmation de haut niveau qui sont compilés en langage machine.
Généralement un compilateur-C est plus efficace qu'un interpréteur-C.
Un autre avantage important des compilateurs par rapport aux interpréteurs et que ceux-ci sont capables de détecter des erreurs au cours de la compilation et ce avant l'exécution réelle de programmes et indépendamment des données d'entrées.

Pour une programme source donné il existe en général une infinité de programmes cibles équivalents. D'un point de vue fonctionnel un seul est suffisant pour exécuter un programme. Par contre on peut s'intéresser aux caractéristiques non fonctionnelles des programmes. Selon les applications données on peut vouloir un programme efficace en terme de temps d'exécution ou en terme de ressources utilisées. A la place d'avoir n-compilateurs spécialisés, il est plus simple d'en avoir un seul mais paramétré par ce que l'on appelle des *options de compilations*. Cette technique est largement utilisée en pratique.

## A.3.4 Décomposition

Décomposer un programme en différents composants offre les avantages suivants:
- les différents composants peuvent être réutilisés,
- les résultats intermédiaires peuvent être réutilisés pour d'autres transformations,,
- les résultats intermédiaires sont disponibles avant le résultat final,
- il est plus simple de tester chaque composant plutôt que le programme dans son intégralité.

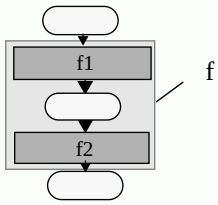
Nous considérons ici deux types basiques de composition : la décomposition séquentielle et la décomposition parallèle.

### A.3.4.1 *Décomposition séquentielle*

Dans les langages impératifs on peut décomposer un programme en plusieurs programmes séquentiels. Dans les langages applicatifs cette décomposition correspond à l'utilisation de la composition de fonction (l'opérateur o ou ; ).

L'exécution de f1 ou de f2 est généralement moins coûteuse que celle de f. (Ce n'est pas nécessairement le cas pour f1 o f2). Les avantages généraux de la décomposition sont évidemment valides.

*figure 145* Décomposition séquentielle, Composition de fonctions



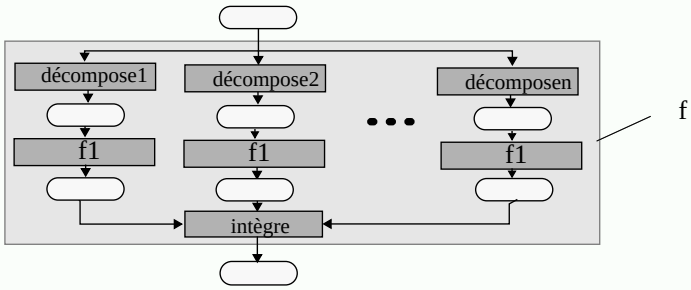

f(x) = f2 ( f1(x) )

### A.3.4.2 *Décomposition parallèle*

Parfois il est possible de décomposer une fonction en plusieurs fonctions pouvant être calculées en parallèle puis d'intégrer les résultats. La figure 146 présente cette idée.

*figure 146* Décomposition parallèle

f(x)= intègre( f1(décompose1(x)), f2(décompose2(x)), ... fn(décomposen(x)) )




Il s'agit d'une forme générale. En pratique les fonctions qui servent a décomposer l'entrée sont typiquement des sélections ou correspondent à la fonction identité. Par exemple si l'entrée est un tuple, chaque fonction peut correspondre à la sélection d'un élément du tuple.
La fonction "intègre" permet de rassembler les résultats. De même cette fonction peut être triviale. Il peut par exemple s'agir par exemple de reconstituer un tuple à partir des différents composants. Inversement cette fonction peut être particulièrement complexe.
En plus des avantages généraux de la décomposition, la décomposition parallèle est évidemment utile dans la mesure où les différents résultats intermédiaires peuvent être calculés parallèlement et à des instants différents. Si les décompositions et l'intégration ne sont pas trop complexes il est possible de réduire ainsi le temps de calcul.

## A.3.5 Restructuration

Si on a des équivalences entres les différentes constructions du langage il est possible de transformer un programme en conservant la même dénotation. Ceci peut être fait pour améliorer une qualité du programme facilitée de compréhension, efficacité, etc.
Dans le cas de langages applicatifs il est particulièrement intéressant de connaître les relations existant entre les différents opérateurs de bases, (par exemple associativité, distributivité, commutativité, etc.) car celles-ci permettent de restructurer un programme.
Cette étape de restructuration peu être intégrée à la phase de compilation, soit en amont en restructurant le programme source, soit en aval en restructurant le programme objet.

## A.3.6 Mémorisation

La notion de ***mémorisation***[1], plus particulièrement celle de ***mémo-fonction*** a été introduite dans le domaine des langages fonctionnels.

L'idée de base de la *mémorisation* et de la *réutilisation* est d'éviter d'appliquer plusieurs fois la même fonction à un même paramètre en mémorisant les résultats déjà calculés dans une *mémoire cache* et en les réutilisant lorsque nécessaire[2].
Il s'agit d'une technique d'optimisation. Fonctionnellement la mémorisation n'apporte rien. Par contre en pratique l'utilisation de cette technique peut être critique.

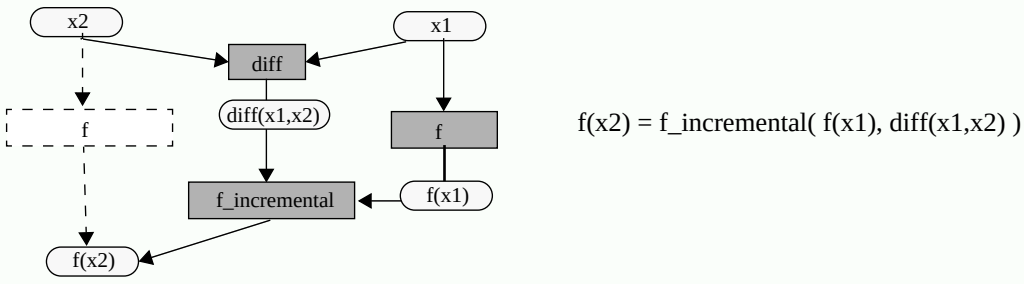
Schématiquement cette technique peut être décrite comme suit. Une mémoire associative permet d'associer au couple (paramètre, fonction) le résultat de l'application de cette fonction (cette mémoire est parfois appelée mémo-table). Au cours de l'exécution du programme, avant d'appliquer une fonction à une valeur la mémoire cache est consultée pour voir si le couple correspondant est trouvé. Si c'est le cas le résultat mémorisé est retourné sinon l'application de la fonction est effectivement évalué et éventuellement mémorisé. Notons qu'il est indispensable que la fonction soit sans effets de bords. La mémorisation est donc une technique d'interprétation.

De nombreuses variantes et optimisations sont possibles autour de cette idée. Quelles sont les applications de fonctions à mémoriser ? Quand ? Comment rendre la recherche dans la mémoire efficace ? Si la capacité de la mémoire est bornée comment sélectionner les applications de fonction à mémoriser ? Quand et quelles sont les applications à effacer ? Quelle est la durée de vie de la mémoire ? Comment identifier la fonction ? Comment stocker le paramètre et le résultat ?

## A.3.7 Approximation, Calcul incrémental

Supposons qu'il faille évaluer f(x2). Supposons aussi que l'on dispose de f(x1) où que f(x1) est moins coûteux à évaluer que f(x2). Une méthode consiste à calculer les "différences" entre x1 et x2, puis à appliquer une certaine fonction au résultat obtenu ainsi qu'à f(x1). Intuitivement ceci consiste à supposer que x1 et x2 sont suffisamment "proches" pour que l'on puisse calculer f(x2) à partir de f(x1) et des "différences" existant entre x1 et x2. Dans le cadre de cette thèse se schéma de calcul sera qualifié d'approximation. .

*figure 147* Calcul incrémental




1. "memoisation" en anglais.
2. L'utilisation du terme réutilisation que nous faisons ici est compatible avec le terme réutilisation de logiciels.

En fait ici, il n'est pas spécifié ce que sont exactement les fonctions diff et f-incrémental. En tout cas il est clair que leur performance détermine l'intérêt de cette technique.

L'approximation est largement utilisée en pratique dans le cas ou x2 et x1 sont deux états successifs d'un même objet. Typiquement si f est une vue sur cet objet, il s'agit de *calculer incrémentallement* les différents états successifs de cette vue. Concrètement cela signifie qu'à la place d'évaluer f à chaque changement d'état de l'objet, la vue est approximée à partir de l'état antérieur. Notons au passage que dans certains cas diff(x1,x2) est une donnée du problème car celle-ci représente le changement d'état de x1 à x2. Cette technique en est alors d'autant plus intéressante.

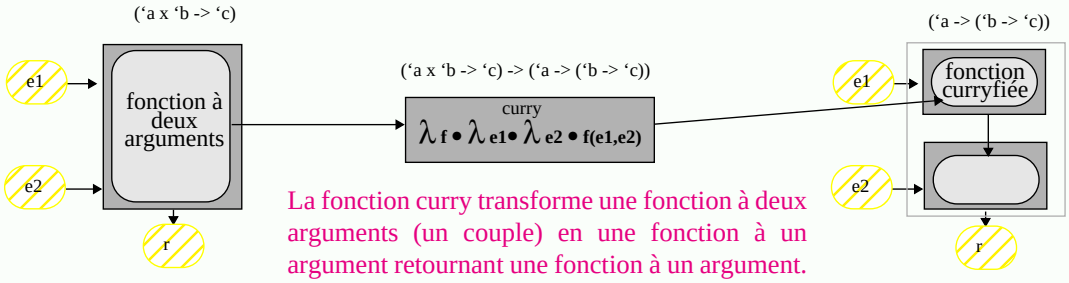
## A.3.8 Graphe de dépendances de programmes (PDG)

Les graphes de dépendances de programme (PDG) ont été introduit dans le début des années 80 pour l'optimisation de programmes [OtteOtte84]. Cette représentation des programmes a été largement étudiée par la suite ; entre autres pour la fusion de programmes, leur parallélisation, la génération de tests, la définition de mesures de complexité, la mise-au-point, etc. [HorwReps92] constitue un très bon guide pour la littérature dans ce domaine. L'une des caractéristiques des PDG est qu'elle permet de calculer facilement des découpes de programmes. Différentes variantes de PDG ont été proposées au cours du temps . Celles-ci se caractérisent entre autres par les constructions du langage prises en compte. Par exemple dans [HorwReps89] la distinction est faite entre les "graphes de dépendances de programmes" (PDG) et les "graphe de dépendance de systèmes" (SDG). Le premier cas correspond à un contexte mono-procédure alors que dans le deuxième cas plusieurs procédures peuvent être prises en compte.

## A.3.9 Spécialisation de programmes

D'un point de vue abstrait, la ***spécialisation de programmes*** correspond à la réduction de domaine, autrement dit à la notion de restriction de fonction en mathématique. Cette notion est à rapprocher de la curryfication utilisée dans le domaine de la logique ou dans les langages fonctionnels.

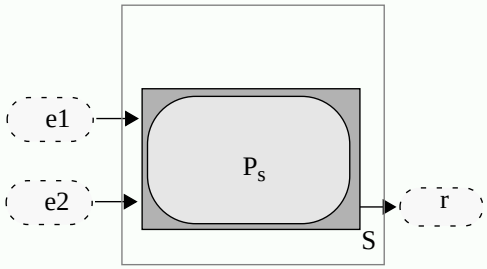
*figure 148* Curryfication



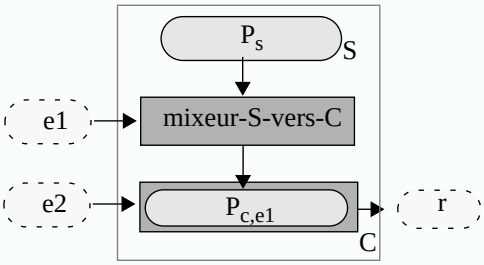

Concrètement la spécialisation de programmes est une technique permettant, à partir d'un programme donné, de générer un programme sémantiquement équivallent mais opérant sur un domaine plus réduit. Bien évidemment on cherche à obtenir un programme spécialisé *plus simple*. Il s'agit donc d'une technique de *réduction de programmes*.

L'*évaluation partielle* est une technique de spécialisation de programme particulièrement développée [Paga91] [JoneGS93], en tout cas au niveau théorique.
L'*exécution symbolique* est une technique plus générale, plus complexe mais qui peut être utilisée pour la spécialisation de programme [CheaHT79] [King81] [CoenD91] [CoenDGM91] ou plus généralement dans le contexte de la maintenance [CowaInce91]. Cette dernière technique, largement étudiée dans les années 70 et 80 à rapidement montré ses limites dans le cas de programmes non triviaux. Elle reste cependant applicable dans certains cas particuliers (c'est le cas par exemple dans le cas de cette thèse pour la programmation globale).
L'une des manières de spécialiser un programme et d'utiliser un *mixeur* [JoneGS93]. Le terme "mixeur" vient du fait qu'un tel système rempli en même temps le rôle d'un interpréteur et de compilateur : il interprète tout ce qu'il peut interpréter en exploitant les paramètres qui lui sont fourni et génère une instruction pour le code qui n'a pas pu interprété statiquement.

*figure 149* Evaluation partielle à l'aide d'un mixeur

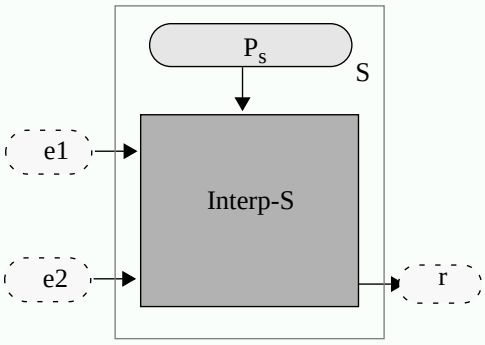



***(a)* Une fonction à deux paramètres**

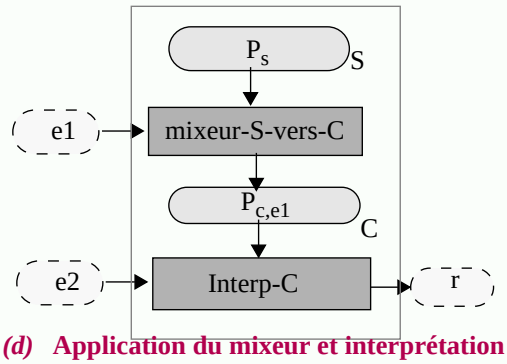



***(b)* Application du mixeur**

Le programme P correspondant à la fonction présentée en (a) est passé au mixeur avec le premier paramètre pour générer un programme Pe1 dans laquelle e1 est fixé (b). Ces deux schémas sont équivalent si l'on à: sem-S(P) (e1,e2) = sem-C( sem-M(mix )(P,e1) ) d2

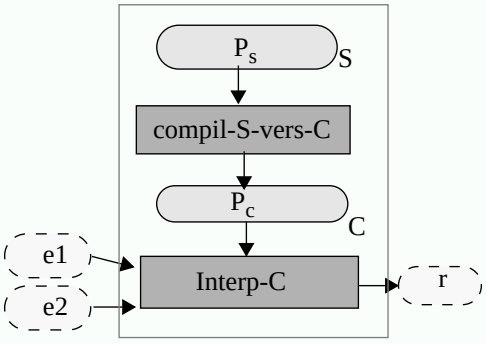



***(c)* Le programme p interprété**




***(d)* Application du mixeur et interprétation**

Les figures (c) et (b) sont respectivement équivalentes aux figures (a) et (b) si ce n'est que les interprétations de P et $P_{e1}$ sont explicitées. Ces deux figures montrent plus clairement les propriétés de l'évaluation partielle.




***(e)* Rappel: un compilateur-S-vers-C**

La figure (e) n'est présentée ici que pour permettre au lecteur de comparer trois manières de calculer une fonction (tout équivalente à (a)):
- l'interprétation (c),
- la compilation (e),
- l'évaluation partielle (b).

Il est intéressant de remarquer les analogies entre un mixeur et un compilateur (tous deux permettent de passer d'un langage S à un langage C) et entre un mixeur et un interpréteur (tous deux permettent d'interpréter le programme pour e1). Ces analogies expliquent le terme mixeur.

## A.3.10Découpage

Les techniques de ***découpage*** (“***slicing***” en anglais) font partie des techniques d’analyse de programmes les plus en étudiées actuellement[1]. Initialement la notion de ***découpe*** d’un programme (“***program slice***”) a été introduite par Weiser [Weis84]. Comme il l’a fait remarquer, lors de la mise-au-point d’un programme, si les valeurs de certaines variables ne sont pas correctes en un point donné, le programmeur cherche à ne lire que les instructions pouvant influer sur le calcul de ces variables. Mentalement il ne s’intéresse qu’à une “découpe” du programme. Le problème qui se pose alors est de pouvoir automatiser le calcul de telles découpes.

Plus précisément, selon Weiser, une découpe d’un programme *P* se calcule en un point *pt* et pour un ensemble de variables *V.* Il s’agit d’un programme plus simple, mais qui produit les mêmes valeurs pour toutes les variables de *V* au point *pt*.

Si l’on considère uniquement le cas ou *pt* correspond à la fin du programme, l’opération abstraite correspondant au découpage est la composition de la fonction représentée par une opération de projection.

Par la suite de nombreuses variantes à ce problème ont été introduites. Les principales sont :

- ***Découpe statique*** *vs.* ***découpe dynamique***. Dans le deuxième cas on s’intéresse à une exécution particulière du programme pour des valeurs spécifiques des paramètres. Ces techniques sont utilisées essentiellement pour la mise au point.
- ***Découpe avant*** *vs.* ***découpe arrière***. La définition de Weiser présentée ci-dessus correspond à une découpe arrière. Il s’agit d’un programme valide donnant le même résultat *jusqu’au* point *pt*. Au contraire une découpe avant est l’ensemble des instructions du programmes qui peuvent être impactées par une modification de l’instruction se trouvant en *pt*. Cette dernière technique correspond à une analyse d’impacts. Elle peut être utilisée pour les tests de régressions (il n’est pas nécessaire de tester de nouveau les instructions qui ne sont pas impactées par une modification).

1. [Tip94] fourni sans doute le guide le plus complet et le plus à jour en ce qui concerne la littérature relative aux techniques de découpage. Cet article comprend plus de 87 références sur le sujet ; 60% sont postérieures à 1990. Ces chiffres témoignent de l’activité qui règne autour de ce sujet. .

# ANNEXE B

# Le prototype APP

## B.1 Exemples

Cette section contient des exemples de résultats produits avec le prototype réalisé. Ceux-ci sont basés sur l'exemple de la lettre générique développé tout au long de ce thèse et montrent l'application de techniques présentées dans le Chapitre IV. Les différents graphes présentés ont été produits avec l'outil DaVinci développé à l'Université de Brème.

### B.1.1 *Module lettre*

Les 6 fichiers ci-dessous correspondent au système lettre. Du point de vue du langage APP, ils constituent un module.

```
#define  CASTAFIORE 1
#include “g_conf”
#include “g_titre_personne”
NOM, ADRESSE
“J’ai le plaisir de “
#if NOMBRE>1 || defined(POLI)
  “vous”
#else
  “te”
#endif
“convier a la reunion du “ DATE.
#if POLI
  “Je vous prie d’agréer mes salutations distinguées.”
#else
  “salut.”
#endif
```

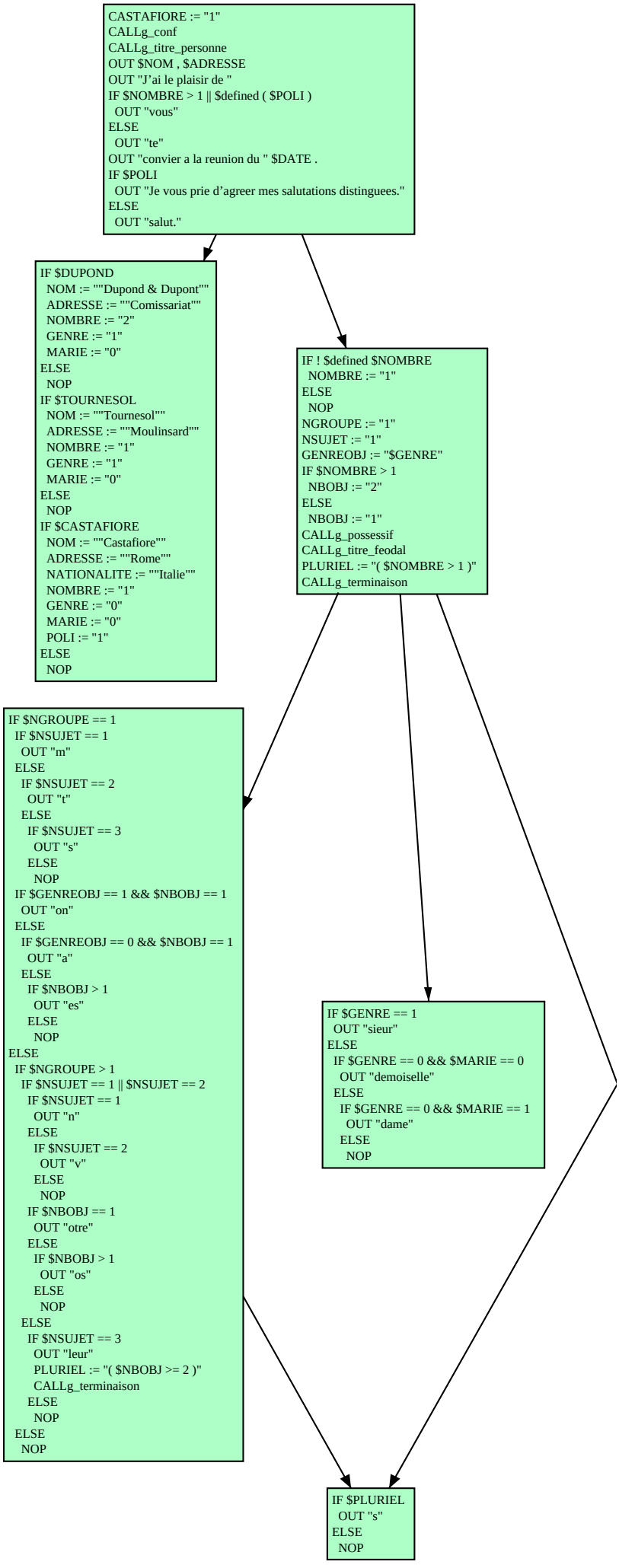
```
#if DUPOND
#define NOM “Dupond & Dupont”
#define ADRESSE “Commissariat”
#define NOMBRE 2
#define GENRE 1
#define MARIE 0
#endif
#if TOURNESOL
#define NOM “Tournesol”
#define ADRESSE “Moulinsard”
#define NOMBRE 1
#define GENRE 1
#define MARIE 0
#endif
#if CASTAFIORE
#define NOM “Castafiore”
#define ADRESSE “Rome”
#define NATIONALITE “Italie”
#define NOMBRE 1
#define GENRE 0
#define MARIE 0
#define POLI 1
#endif
```

```
#ifndef NOMBRE
    #define NOMBRE 1
#endif
#define NGROUPE 1
#define NSUJET 1
#define GENREOBJ GENRE
#if NOMBRE>1
    #define NBOBJ 2
#else
    #define NBOBJ 1
#endif
#include “g_possessif”
#include “g_titre_feodal”
#define PLURIEL (NOMBRE>1)
#include “g_terminaison”
```

```
#if GENRE == 1
    “sieur”
#elif GENRE == 0 && MARIE == 0
    “demoiselle”
#elif GENRE == 0 && MARIE == 1
    “dame”
#endif
```

```
#if NGROUPE == 1
    #if NSUJET == 1
        “m”
    #elif NSUJET == 2
        “t”
    #elif NSUJET == 3
        “s”
    #endif
    #if GENREOBJ==1 && NBOBJ==1
        “on”
    #elif GENREOBJ==0 && NBOBJ==1
        “a”
    #elif NBOBJ>1
        “es”
    #endif
#elif NGROUPE > 1
    #if NSUJET == 1 || NSUJET == 2
        #if NSUJET == 1
            “n”
        #elif NSUJET == 2
            “v”
        #endif
        #if NBOBJ==1
            “otre”
        #elif NBOBJ>1
            “os”
             #endif
    #elif NSUJET == 3
        “leur”
        #define PLURIEL (NBOBJ>=2)
        #include “g_terminaison”
      #endif
#endif
```

```
#if PLURIEL
    “s”
#endif
```




La figure de droite représente le graphe d'appel entre les procédures du module. Sans aucune connaissance préablable de ce système, le chargé de maintenance peut déduire de ce graphe que g_lettre est la procédure principale et qu'une première lecture devrait commencer là. On s'apperçoit également que le sous-graphe ayant g_titre comme racine est un sous-sytème indépendant. Peut être serait il judicieux de rassembler les quatres procédures dans un sous-module. D'un point de vue concret cette opération consiste à créer un répertoire contenant les quatres fichiers.

### B.1.2 *Procédure g_titre_personne*

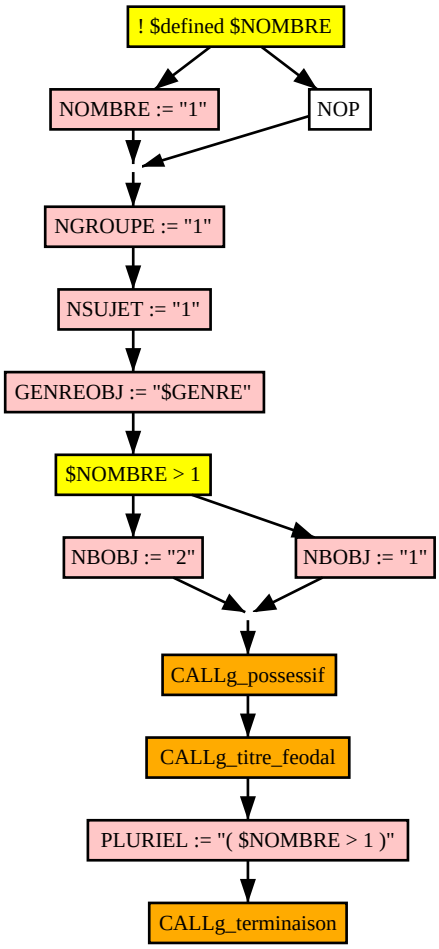
Le fichier CPP g_titre_personne

```
#ifndef NOMBRE
    #define NOMBRE 1
#endif
#define NGROUPE 1
#define NSUJET 1
#define GENREOBJ GENRE
#if NOMBRE>1
    #define NBOBJ 2
#else
    #define NBOBJ 1
#endif
#include “g_possessif”
#include “g_titre_feodal”
#define PLURIEL (NOMBRE>1)
#include “g_terminaison”
```

La procédure APP correspondante.

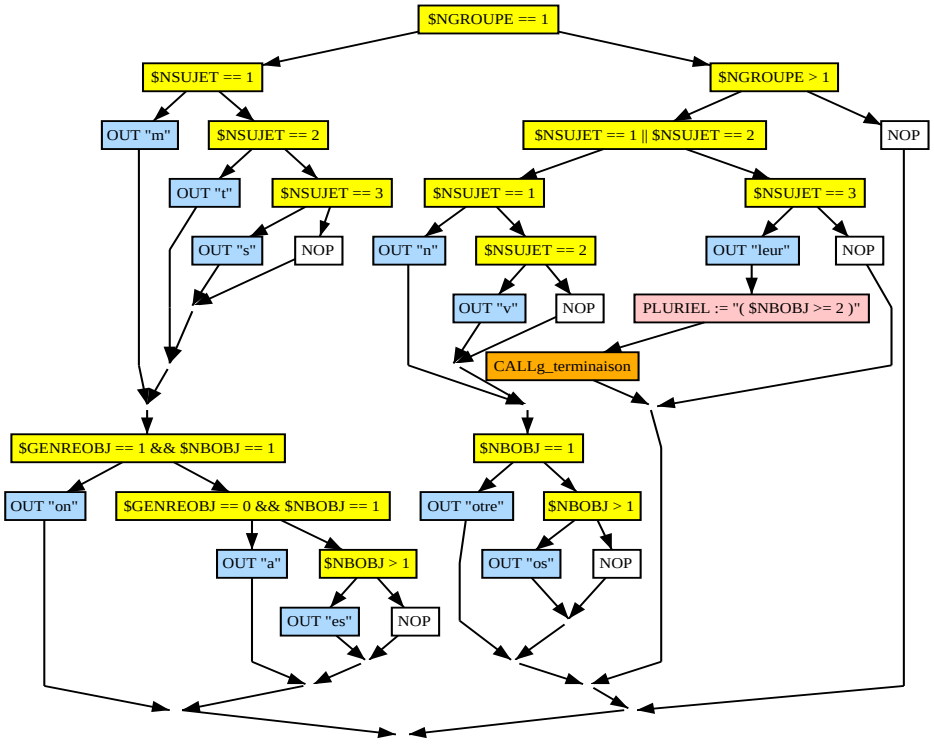
```
PROCEDURE g_titre_personne IS
 BEGIN
  IF ! $defined $NOMBRE
   NOMBRE := “1”
  ELSE
   NOP
  NGROUPE := “1”
  NSUJET := “1”
  GENREOBJ := “$GENRE”
  IF $NOMBRE > 1
   NBOBJ := “2”
  ELSE
   NBOBJ := “1”
  CALLg_possessif
  CALLg_titre_feodal
  PLURIEL := “( $NOMBRE > 1 )”
  CALLg_terminaison
 END
```

Le graphe de flot de controle.



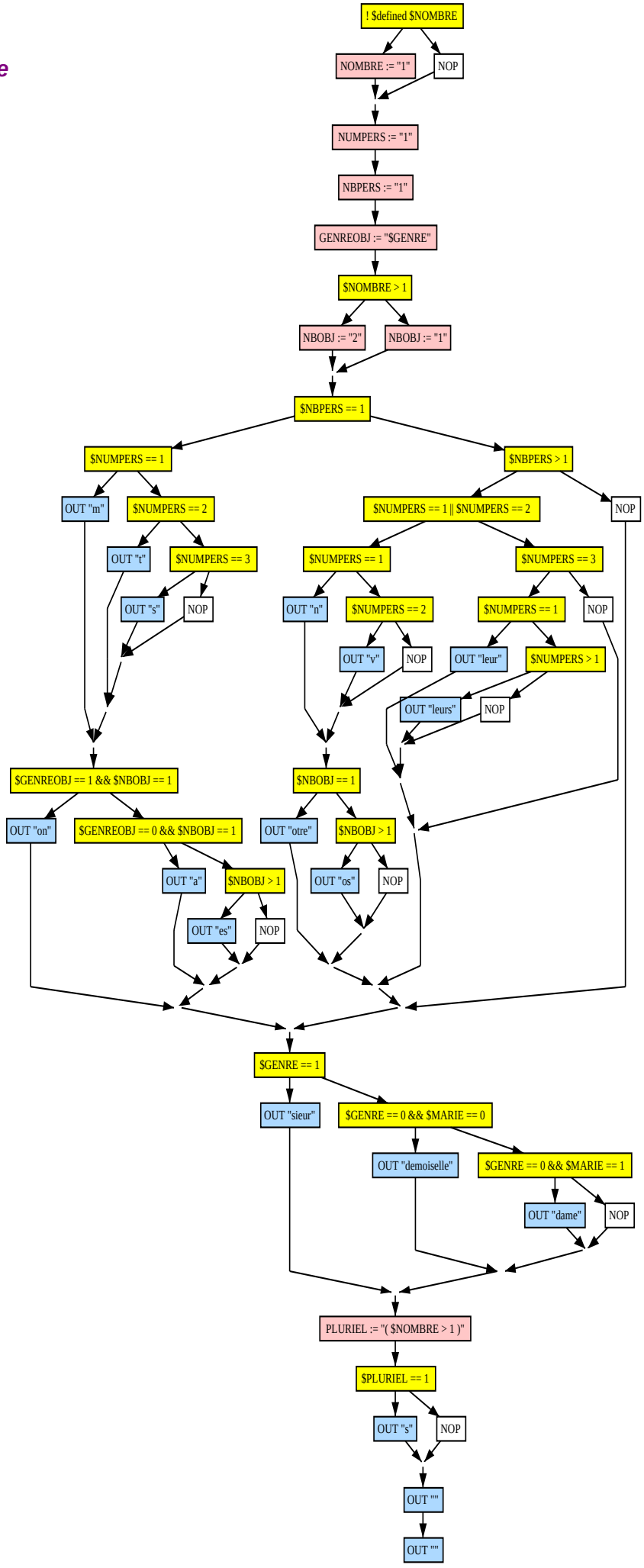

## B.1.3 Procédure g_possessif

```
#if NGROUPE == 1
    #if NSUJET == 1
        “m”
    #elif NSUJET == 2
        “t”
    #elif NSUJET == 3
        “s”
    #endif
    #if GENREOBJ==1 && NBOBJ==1
        “on”
    #elif GENREOBJ==0 && NBOBJ==1
        “a”
    #elif NBOBJ>1
        “es”
    #endif
#elif NGROUPE > 1
    #if NSUJET == 1 || NSUJET == 2
        #if NSUJET == 1
            “n”
        #elif NSUJET == 2
            “v”
        #endif
        #if NBOBJ==1
            “otre”
        #elif NBOBJ>1
            “os”
            #endif
    #elif NSUJET == 3
        “leur”
        #define PLURIEL (NBOBJ>=2)
        #include “g_terminaison”
       #endif
#endif
```



### B.1.4 *Procédure g_titre_personne étendue*

```
#ifndef NOMBRE
    #define NOMBRE 1
#endif
#define NUMPERS 1
#define NBPERS 1
#define GENREOBJ GENRE
#if NOMBRE>1
    #define NBOBJ 2
#else
    #define NBOBJ 1
#endif
#if NBPERS == 1
    #if NUMPERS == 1
        “m”
    #elif NUMPERS == 2
        “t”
    #elif NUMPERS == 3
        “s”
    #endif
    #if GENREOBJ==1 && NBOBJ==1
        “on”
    #elif GENREOBJ==0 && NBOBJ==1
        “a”
    #elif NBOBJ>1
        “es”
    #endif
#elif NBPERS > 1
    #if NUMPERS == 1 || NUMPERS == 2
        #if NUMPERS == 1
            “n”
        #elif NUMPERS == 2
            “v”
        #endif
        #if NBOBJ==1
            “otre”
        #elif NBOBJ>1
            “os”
             #endif
    #elif NUMPERS == 3
        #if NUMPERS == 1
            “leur”
        #elif NUMPERS > 1
            “leurs”
        #endif
      #endif
#endif
#if GENRE == 1
    “sieur”
#elif GENRE == 0 && MARIE == 0
    “demoiselle”
#elif GENRE == 0 && MARIE == 1
    “dame”
#endif
#define PLURIEL (NOMBRE>1)
#if PLURIEL == 1
    “s”
#endif
```



### B.1.5 *Elimination du code mort de g_titre_personne étendue*

La procédure précédente est spécialisée avec une mémoire mixte vide. Autrement dit on ne spécifie aucune contrainte. Dans ce cas le code mort est éliminé. Ici une grande partie du code disparaît. Ce qui est normal car le titre d'une personne ne peut commencer que par "mon", "ma" ou "mes" ; donc seuls les possessifs de la première personne du premier groupe sont utilisés. Dans le code cela correspond aux définitions :

```
#define NUMPERS 1
#define NBPERS 1
```

Ci-dessous la procédure est affichée juste après la phase de spécialisation. Cette phase élimine les chemins impossibles et substitue les variables par leurs valeurs lorsque c'est possible. Par exemple on notera que la variable GENREOBJ a été substituée par GENRE.

```
PROCEDURE g_titre_personne_etendu IS
 BEGIN
  IF ! $defined $NOMBRE
   NOMBRE := "1"
  ELSE
   NOP
  NUMPERS := "1"
  NBPERS := "1"
  GENREOBJ := "$GENRE"
  IF $NOMBRE > 1
   NBOBJ := "2"
  ELSE
   NBOBJ := "1"
  OUT "m"
  IF $GENRE == 1 && $NBOBJ == 1
   OUT "on"
  ELSE
   IF $GENRE == 0 && $NBOBJ == 1
    OUT "a"
   ELSE
    IF $NBOBJ > 1
     OUT "es"
    ELSE
     NOP
  IF $GENRE == 1
   OUT "sieur"
  ELSE
   IF $GENRE == 0 && $MARIE == 0
    OUT "demoiselle"
   ELSE
    IF $GENRE == 0 && $MARIE == 1
     OUT "dame"
    ELSE
     NOP
  PLURIEL := "( $NOMBRE > 1 )"
  IF ( $NOMBRE > 1 ) == 1
   OUT "s"
  ELSE
   NOP
  OUT ""
  OUT ""
 END
```

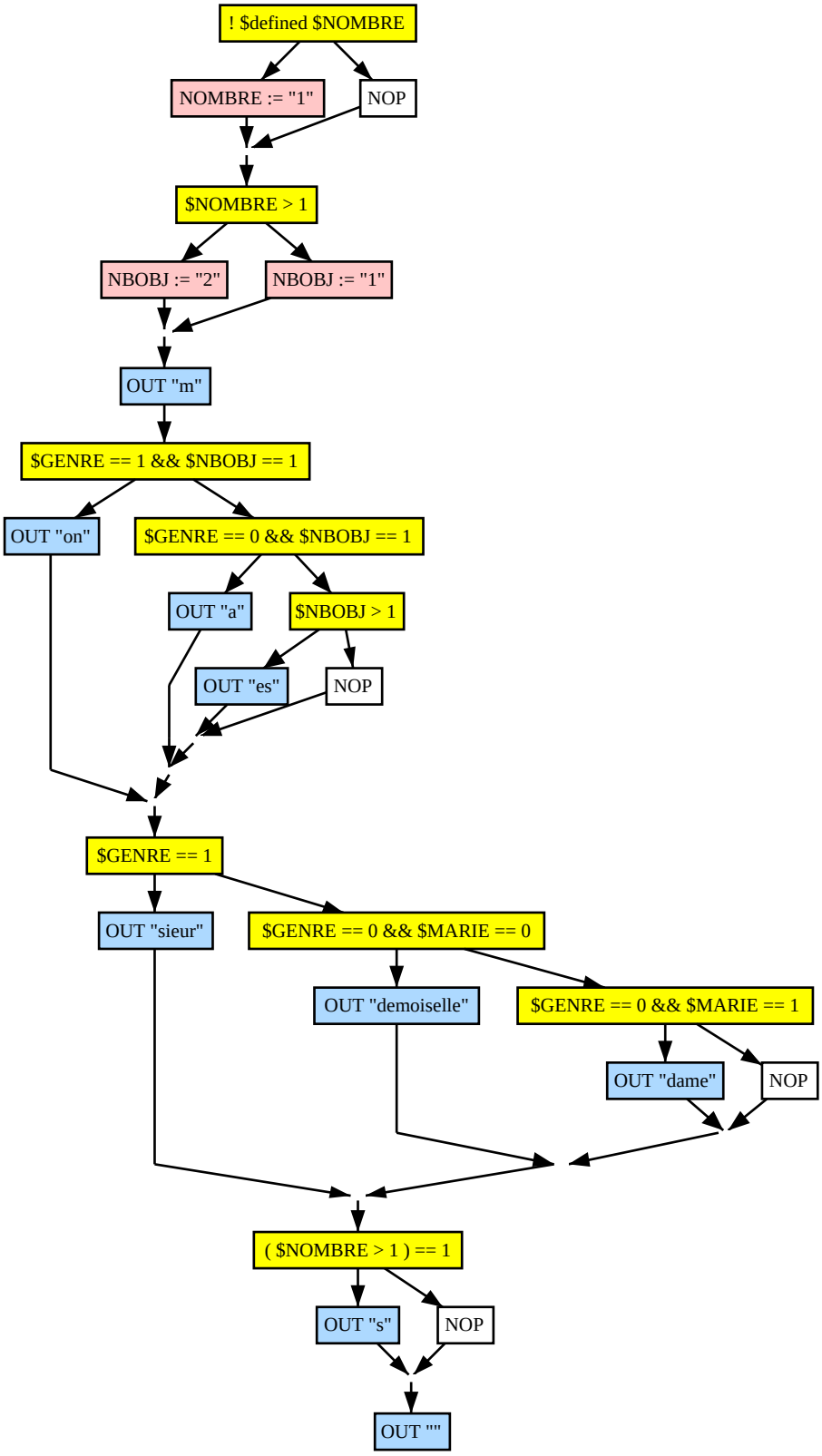
La phase de spécialisation génère des définitions inutiles. Dans la colonne de gauche celles-ci sont marquées en italiques. Elles ont été éliminées dans la procédure ci-dessous.

La page ci-contre montre le graphe de flot de contrôle après spécialisation et élimination du code mort. Il est considérablement plus simple que celui de la page précédente !

```
PROCEDURE g_titre_personne_etendu IS
 BEGIN
  IF ! $defined $NOMBRE
   NOMBRE := "1"
  ELSE
   NOP


  IF $NOMBRE > 1
   NBOBJ := "2"
  ELSE
   NBOBJ := "1"
  OUT "m"
  IF $GENRE == 1 && $NBOBJ == 1
   OUT "on"
  ELSE
   IF $GENRE == 0 && $NBOBJ == 1
    OUT "a"
   ELSE
    IF $NBOBJ > 1
     OUT "es"
    ELSE
     NOP
  IF $GENRE == 1
   OUT "sieur"
  ELSE
   IF $GENRE == 0 && $MARIE == 0
    OUT "demoiselle"
   ELSE
    IF $GENRE == 0 && $MARIE == 1
     OUT "dame"
    ELSE
     NOP

  IF ( $NOMBRE > 1 ) == 1
   OUT "s"
  ELSE
   NOP
  OUT ""
  OUT ""
 END
```


! $defined $NOMBRE
NOMBRE := "1"
NOP
$NOMBRE > 1
NBOBJ := "2"
NBOBJ := "1"
OUT "m"
$GENRE == 1 && $NBOBJ == 1
OUT "on"
$GENRE == 0 && $NBOBJ == 1
OUT "a"
$NBOBJ > 1
OUT "es"
NOP
$GENRE == 1
OUT "sieur"
$GENRE == 0 && $MARIE == 0
OUT "demoiselle"
$GENRE == 0 && $MARIE == 1
OUT "dame"
NOP
( $NOMBRE > 1 ) == 1
OUT "s"
NOP
OUT ""

### B.1.6 *Graphe de dépendance de g_titre_personne étendue*

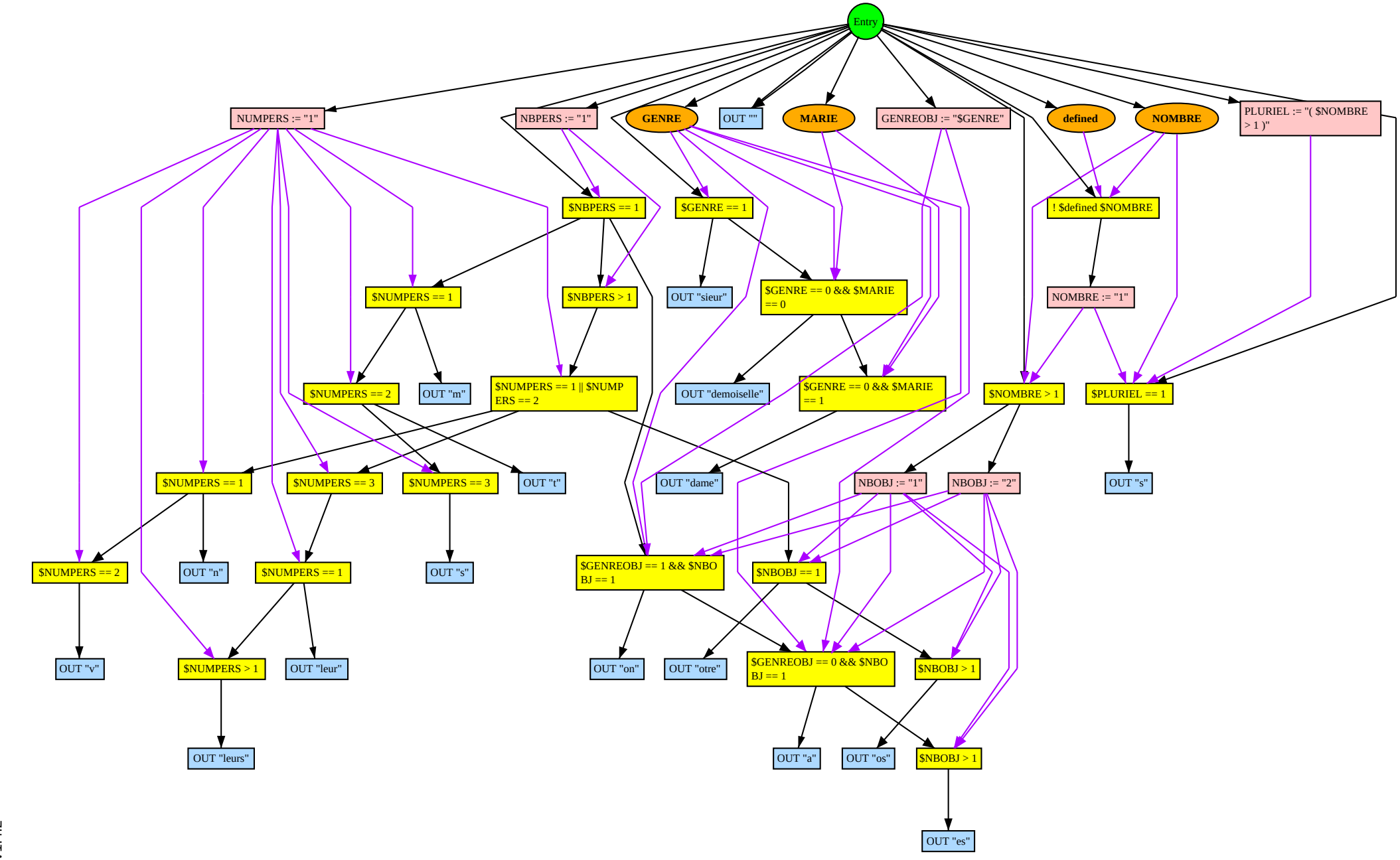
Ci-contre le graphe de dépendance de la procédure g_titre_personne étendue. La dépendance de sortie n'est pas représentée.

On remarquera tout de suite les paramètres qui sont représentés par des ovales oranges.

L'oeil est attiré par la partie gauche du graphe. Toute une partie ne dépend que des définitions *NUMPERS := "1"* et *NBPERS := "1"* . On peut ainsi s'apercevoir visuellement que le code correspondant est inutile.

La page suivante montre la même procédure mais après élimination du code mort via une spécialisation.

```
PROCEDURE g_titre_personne_etendu IS
 BEGIN
  IF ! $defined $NOMBRE
   NOMBRE := "1"
  ELSE
   NOP
  NUMPERS := "1"
  NBPERS := "1"
  GENREOBJ := "$GENRE"
  IF $NOMBRE > 1
   NBOBJ := "2"
  ELSE
   NBOBJ := "1"
  IF $NBPERS == 1
   IF $NUMPERS == 1
    OUT "m"
   ELSE
    IF $NUMPERS == 2
     OUT "t"
    ELSE
     IF $NUMPERS == 3
      OUT "s"
     ELSE
      NOP
   IF $GENREOBJ == 1 && $NBOBJ == 1
    OUT "on"
   ELSE
    IF $GENREOBJ == 0 && $NBOBJ == 1
     OUT "a"
    ELSE
     IF $NBOBJ > 1
      OUT "es"
     ELSE
      NOP
  ELSE
   IF $NBPERS > 1
    IF $NUMPERS == 1 || $NUMPERS == 2
     IF $NUMPERS == 1
      OUT "n"
     ELSE
      IF $NUMPERS == 2
       OUT "v"
      ELSE
       NOP
     IF $NBOBJ == 1
      OUT "otre"
     ELSE
      IF $NBOBJ > 1
       OUT "os"
      ELSE
       NOP
    ELSE
     IF $NUMPERS == 3
      IF $NUMPERS == 1
       OUT "leur"
      ELSE
       IF $NUMPERS > 1
        OUT "leurs"
       ELSE
        NOP
     ELSE
      NOP
   ELSE
    NOP
  IF $GENRE == 1
   OUT "sieur"
  ELSE
   IF $GENRE == 0 && $MARIE == 0
    OUT "demoiselle"
   ELSE
    IF $GENRE == 0 && $MARIE == 1
     OUT "dame"
    ELSE
     NOP
  PLURIEL := "( $NOMBRE > 1 )"
  IF $PLURIEL == 1
   OUT "s"
  ELSE
   NOP OUT "" OUT "" END
```

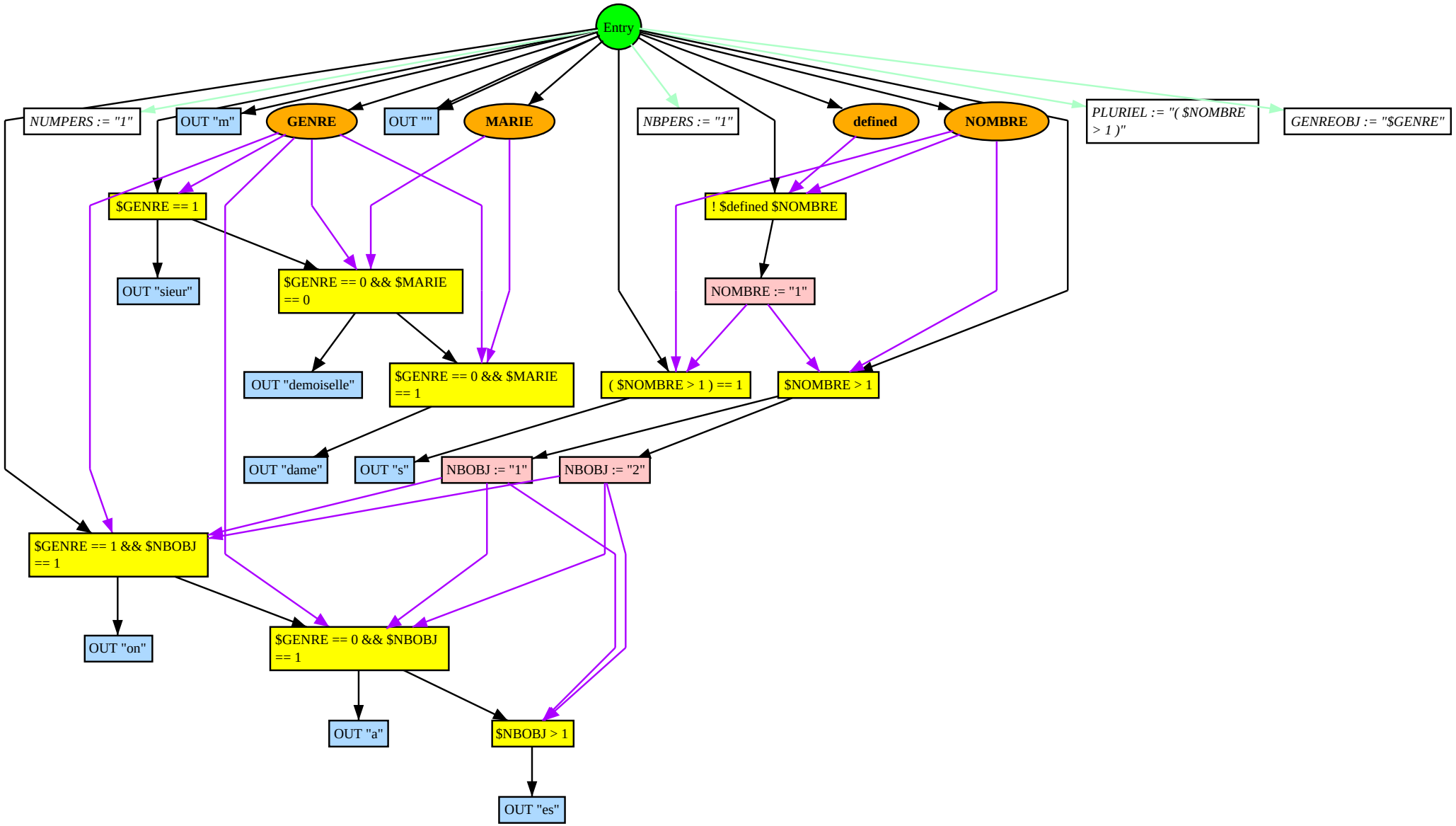

Entry
NUMPERS := "1"
NBPERS := "1"
GENRE
OUT ""
MARIE
GENREOBJ := "$GENRE"
defined
NOMBRE
PLURIEL := "( $NOMBRE > 1 )"
$NBPERS == 1
$GENRE == 1
! $defined $NOMBRE
$NUMPERS == 1
$NBPERS > 1
OUT "sieur"
$GENRE == 0 && $MARIE == 0
NOMBRE := "1"
$NUMPERS == 2
OUT "m"
$NUMPERS == 1 || $NUMPERS == 2
OUT "demoiselle"
$GENRE == 0 && $MARIE == 1
$NOMBRE > 1
$PLURIEL == 1
$NUMPERS == 1
$NUMPERS == 3
$NUMPERS == 3
OUT "t"
OUT "dame"
NBOBJ := "1"
NBOBJ := "2"
OUT "s"
$NUMPERS == 2
OUT "n"
$NUMPERS == 1
OUT "s"
$GENREOBJ == 1 && $NBOBJ == 1
$NBOBJ == 1
OUT "v"
$NUMPERS > 1
OUT "leur"
OUT "on"
OUT "otre"
$GENREOBJ == 0 && $NBOBJ == 1
$NBOBJ > 1
OUT "leurs"
OUT "a"
OUT "os"
$NBOBJ > 1
OUT "es"

### B.1.7 *Graphe de dépendance après l'élimination du code mort*

Ci-contre le graphe de dépendance est présenté après l'élimination du code mort via spécialisation mais avant l'élimination des définitions inutiles et des chemins vides.

Les noeuds dessinés en blanc correspondent aux définitions inutiles. Effectivement aucune dépendance de données ne sort de ces noeuds. Ils seront éliminés dans une deuxième phase.

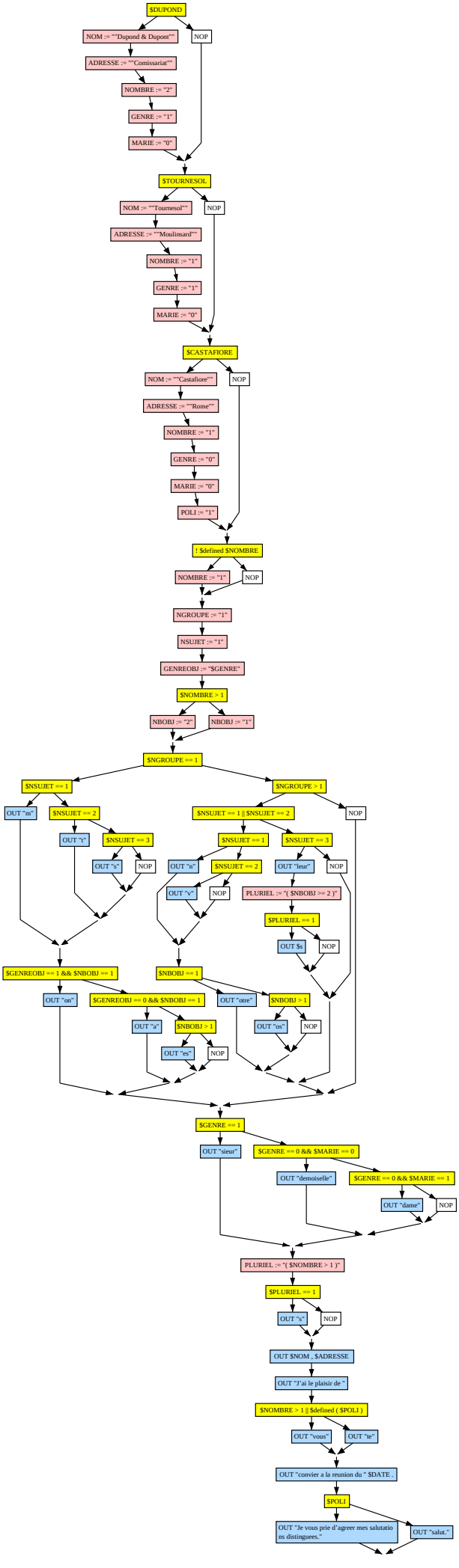
```
PROCEDURE g_titre_personne_etendu IS
 BEGIN
  IF ! $defined $NOMBRE
   NOMBRE := “1”
  ELSE
   NOP
  NUMPERS := “1”
  NBPERS := “1”
  GENREOBJ := “$GENRE”
  IF $NOMBRE > 1
   NBOBJ := “2”
  ELSE
   NBOBJ := “1”
  OUT “m”
  IF $GENRE == 1 && $NBOBJ == 1
   OUT “on”
  ELSE
   IF $GENRE == 0 && $NBOBJ == 1
    OUT “a”
   ELSE
    IF $NBOBJ > 1
     OUT “es”
    ELSE
     NOP
  IF $GENRE == 1
   OUT “sieur”
  ELSE
   IF $GENRE == 0 && $MARIE == 0
    OUT “demoiselle”
   ELSE
    IF $GENRE == 0 && $MARIE == 1
     OUT “dame”
    ELSE
     NOP
  PLURIEL := “( $NOMBRE > 1 )”
  IF ( $NOMBRE > 1 ) == 1
   OUT “s”
  ELSE
   NOP
  OUT ““
  OUT ““
 END
```


Entry
NUMPERS := "1"
OUT "m"
GENRE
OUT ""
MARIE
NBPERS := "1"
defined
NOMBRE
PLURIEL := "( $NOMBRE > 1 )"
GENREOBJ := "$GENRE"
$GENRE == 1
! $defined $NOMBRE
OUT "sieur"
$GENRE == 0 && $MARIE == 0
NOMBRE := "1"
OUT "demoiselle"
$GENRE == 0 && $MARIE == 1
( $NOMBRE > 1 ) == 1
$NOMBRE > 1
OUT "dame"
OUT "s"
NBOBJ := "1"
NBOBJ := "2"
$GENRE == 1 && $NBOBJ == 1
OUT "on"
$GENRE == 0 && $NBOBJ == 1
OUT "a"
$NBOBJ > 1
OUT "es"

### B.1.8 *Graphe de contrôle de la procédure g_lettre étendue*

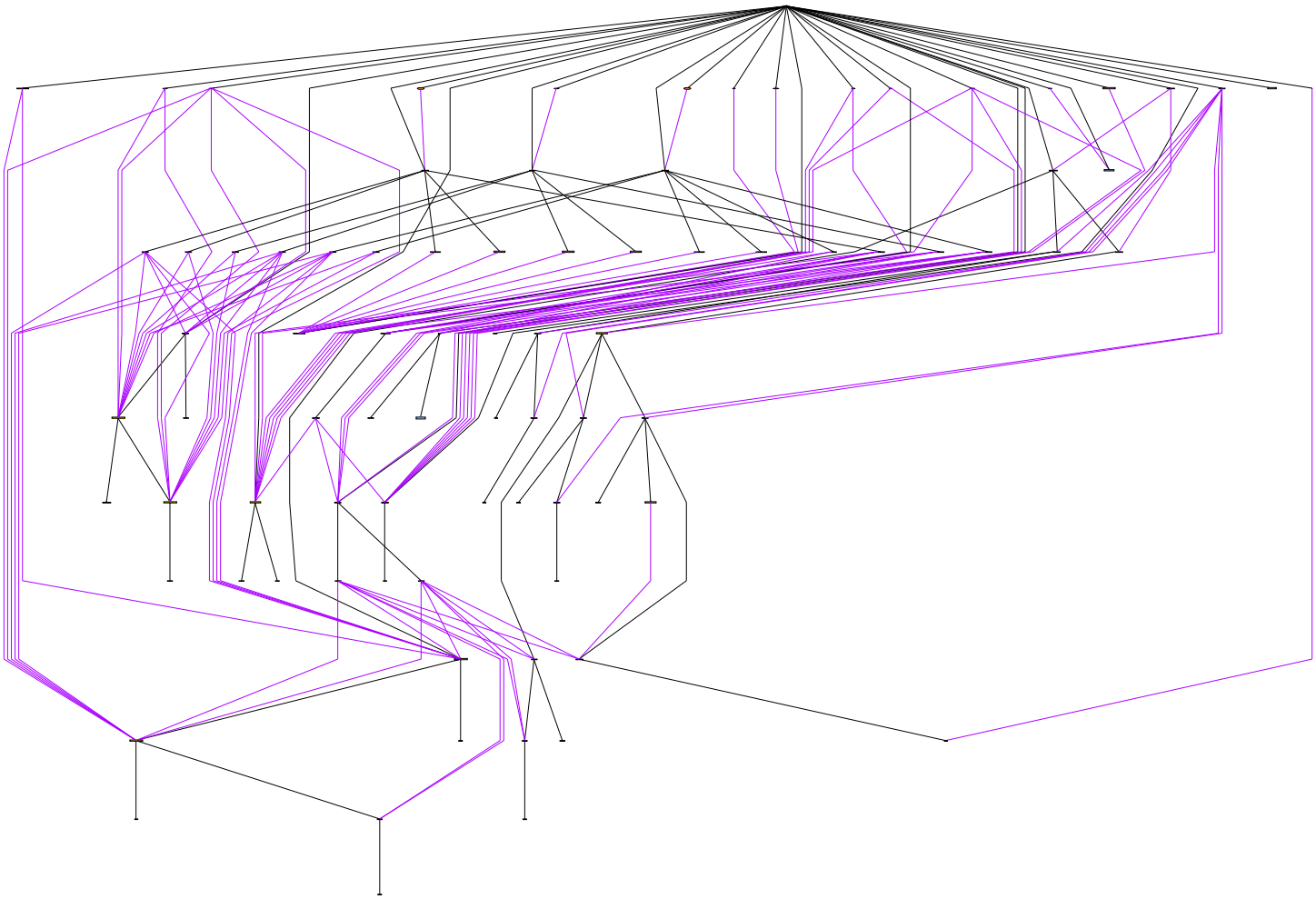
La procédure suivante correspond à la procédure g_lettre étendue. Dans les sections qui suivent elle va être spécialisée pour le cas où CASTAFIORE vaux 1.

```
PROCEDURE g_lettre_etendu IS
 BEGIN
  IF $DUPOND
   NOM := ""Dupond & Dupont""
   ADRESSE := ""Commissariat""
   NOMBRE := "2"
   GENRE := "1"
   MARIE := "0"
  ELSE
   NOP
  IF $TOURNESOL
   NOM := ""Tournesol""
   ADRESSE := ""Moulinsard""
   NOMBRE := "1"
   GENRE := "1"
   MARIE := "0"
  ELSE
   NOP
  IF $CASTAFIORE
   NOM := ""Castafiore""
   ADRESSE := ""Rome""
   NOMBRE := "1"
   GENRE := "0"
   MARIE := "0"
   POLI := "1"
  ELSE
   NOP
  IF ! $defined $NOMBRE
   NOMBRE := "1"
  ELSE
   NOP
  NGROUPE := "1"
  NSUJET := "1"
  GENREOBJ := "$GENRE"
  IF $NOMBRE > 1
   NBOBJ := "2"
  ELSE
   NBOBJ := "1"
  IF $NGROUPE == 1
   IF $NSUJET == 1
    OUT "m"
   ELSE
    IF $NSUJET == 2
     OUT "t"
    ELSE
     IF $NSUJET == 3
      OUT "s"
     ELSE
      NOP
   IF $GENREOBJ == 1 && $NBOBJ == 1
    OUT "on"
   ELSE
    IF $GENREOBJ == 0 && $NBOBJ == 1
     OUT "a"
    ELSE
     IF $NBOBJ > 1
      OUT "es"
     ELSE
      NOP
  ELSE
   IF $NGROUPE > 1
    IF $NSUJET == 1 || $NSUJET == 2
     IF $NSUJET == 1
      OUT "n"
     ELSE
      IF $NSUJET == 2
       OUT "v"
      ELSE
       NOP
     IF $NBOBJ == 1
      OUT "otre"
     ELSE
      IF $NBOBJ > 1
       OUT "os"
      ELSE
       NOP
    ELSE
     IF $NSUJET == 3
      OUT "leur"
      PLURIEL := "( $NBOBJ >= 2 )"
      IF $PLURIEL == 1
       OUT $s
      ELSE
       NOP
     ELSE
      NOP
   ELSE
    NOP
  IF $GENRE == 1
   OUT "sieur"
  ELSE
   IF $GENRE == 0 && $MARIE == 0
    OUT "demoiselle"
   ELSE
    IF $GENRE == 0 && $MARIE == 1
     OUT "dame"
    ELSE
     NOP
  PLURIEL := "( $NOMBRE > 1 )"
  IF $PLURIEL == 1
   OUT "s"
  ELSE
   NOP
  OUT $NOM , $ADRESSE
  OUT "J'ai le plaisir de "
  IF $NOMBRE > 1 || $defined ( $POLI )
   OUT "vous"
  ELSE
   OUT "te"
  OUT "convier a la reunion du " $DATE .
  IF $POLI
   OUT "Je vous prie d'agreer mes salutations distinguees."
  ELSE
   OUT "salut."
END
```


$DUPOND
NOM := ""Dupond & Dupont""
NOP
ADRESSE := ""Comissariat""
NOMBRE := "2"
GENRE := "1"
MARIE := "0"
$TOURNESOL
NOM := ""Tournesol""
NOP
ADRESSE := ""Moulinsard""
NOMBRE := "1"
GENRE := "1"
MARIE := "0"
$CASTAFIORE
NOM := ""Castafiore""
NOP
ADRESSE := ""Rome""
NOMBRE := "1"
GENRE := "0"
MARIE := "0"
POLI := "1"
! $defined $NOMBRE
NOMBRE := "1"
NOP
NGROUPE := "1"
NSUJET := "1"
GENREOBJ := "$GENRE"
$NOMBRE > 1
NBOBJ := "2"
NBOBJ := "1"
$NGROUPE == 1
$NSUJET == 1
OUT "m"
$NSUJET == 2
OUT "t"
$NSUJET == 3
OUT "s"
NOP
$NGROUPE > 1
$NSUJET == 1 || $NSUJET == 2
$NSUJET == 1
OUT "n"
$NSUJET == 2
OUT "v"
NOP
$NSUJET == 3
OUT "leur"
NOP
PLURIEL := "( $NBOBJ >= 2 )"
$PLURIEL == 1
OUT $s
NOP
NOP
$GENREOBJ == 1 && $NBOBJ == 1
OUT "on"
$GENREOBJ == 0 && $NBOBJ == 1
OUT "a"
$NBOBJ > 1
OUT "es"
NOP
$NBOBJ == 1
OUT "otre"
$NBOBJ > 1
OUT "os"
NOP
$GENRE == 1
OUT "sieur"
$GENRE == 0 && $MARIE == 0
OUT "demoiselle"
$GENRE == 0 && $MARIE == 1
OUT "dame"
NOP
PLURIEL := "( $NOMBRE > 1 )"
$PLURIEL == 1
OUT "s"
NOP
OUT $NOM , $ADRESSE
OUT "J'ai le plaisir de "
$NOMBRE > 1 || $defined ( $POLI )
OUT "vous"
OUT "te"
OUT "convier a la reunion du " $DATE .
$POLI
OUT "Je vous prie d'agreer mes salutatio ns distinguees."
OUT "salut."

### B.1.9 *Graphe de dépendance de g_lettre étendue*

Cette figure montre que les graphes de dépendance ne sont visualisables que pour des exemples simples. Ils ne sont pas fait pour cela ; ils servent à réduire les programmes (via une spécialisation ou une découpe).



### B.1.10 *Spécialisation de g_lettre étendue pour le cas CASTAFIORE = 1*

La figure suivante correspond au graphe de dépendance spécialisé avec la définition CASTAFIORE=1. L'étape d'élimination des définitions n'a pas encore été appliquée. On constate effectivement que de nombreuses définitions sont inutiles (elles sont marquées en blanc).

Il est intéressant de remarquer que la suppression de définitions peut impliquer la suppression d'instructions conditionnelles lorsque l'ensemble des instructions qu'elles contrôle devient vide.

La procédure ci-dessous est obtenue juste après la spécialisation.

```
PROCEDURE g_lettre_etendu_spec IS
 BEGIN
   IF $DUPOND
    NOM := ""Dupond & Dupont""
    ADRESSE := ""Commissariat""
    NOMBRE := "2"
    GENRE := "1"
    MARIE := "0"
  ELSE
    NOP
  IF $TOURNESOL
    NOM := ""Tournesol""
    ADRESSE := ""Moulinsard""
    NOMBRE := "1"
    GENRE := "1"
    MARIE := "0"
  ELSE
    NOP
  NOM := ""Castafiore""
  ADRESSE := ""Rome""
  NOMBRE := "1"
  GENRE := "0"
  MARIE := "0"
  POLI := "1"
  IF ! $defined 1
    NOMBRE := "1"
  ELSE
    NOP
  NGROUPE := "1"
  NSUJET := "1"
  GENREOBJ := "$GENRE"
  NBOBJ := "1"
  OUT "m"
  OUT "a"
  OUT "demoiselle"
  PLURIEL := "( $NOMBRE > 1 )"
  NOP
  OUT "Castafiore" , "Rome"
  OUT "J'ai le plaisir de "
  IF $NOMBRE > 1 || $defined ( 1 )
    OUT "vous"
  ELSE
    OUT "te"
  OUT "convier a la reunion du " $DATE .
  OUT "Je vous prie d'agreer mes salutations distinguees."
 END
```

Cette procédure est obtenue après l'élimination des définitions inutiles et des chemins vides.

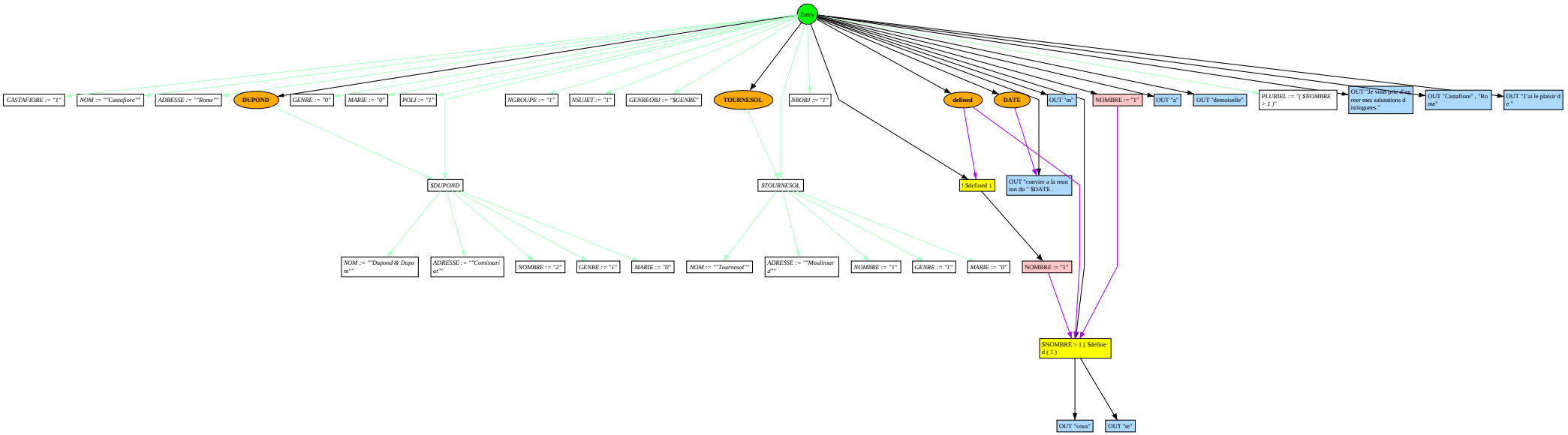
Les parties notées en italiques auraient dues être éliminées. Si elles ne l'ont pas été c'est parce que la version actuelle du prototype n'intègre que des règles triviales de simplification d'expressions. Dans les versions futures de telles simplifications devront être prises en compte.

```
PROCEDURE g_lettre_etendu_spec IS
 BEGIN
   NOMBRE := "1"
   IF ! $defined 1
     NOMBRE := "1"
   ELSE
     NOP
   OUT "m"
   OUT "a"
   OUT "demoiselle"
   OUT "Castafiore" , "Rome"
   OUT "J'ai le plaisir de "
   IF $NOMBRE > 1 || $defined ( 1 )
     OUT "vous"
   ELSE
     OUT "te"
   OUT "convier a la reunion du " $DATE .
   OUT "Je vous prie d'agreer mes salutations distinguees."
 END
```

On arrive alors à la procédure suivante. La spécialisation au cas où CASTAFIORE=1 permet de passer de 116 lignes à 11 lignes ! La procédure réduite est bien plus facile à comprendre.

```
PROCEDURE g_lettre_etendu_spec IS
 BEGIN
   OUT "m"
   OUT "a"
   OUT "demoiselle"
   OUT "Castafiore" , "Rome"
   OUT "J'ai le plaisir de "
   OUT "vous"
   OUT "convier a la reunion du " $DATE .
   OUT "Je vous prie d'agreer mes salutations distinguees."
 END
```

Remarquons que l'on obtient encore un objet générique puisqu'il est paramétré par la date de la réunion. Le graphe de dépendance obtenu à la fin de ce processus peut être utilisé pour générer une signature. Ce qui permet l'exploration interactive des signatures.


Entry
CASTAFIORE := "1"
NOM := ""Castafiore""
ADRESSE := ""Rome""
DUPOND
GENRE := "0"
MARIE := "0"
POLI := "1"
NGROUPE := "1"
NSUJET := "1"
GENREOBJ := "$GENRE"
TOURNESOL
NBOBJ := "1"
defined
DATE
OUT "m"
NOMBRE := "1"
OUT "a"
OUT "demoiselle"
PLURIEL := "( $NOMBRE > 1 )"
OUT "Je vous prie d'ag reer mes salutations d istinguees."
OUT "Castafiore" , "Ro me"
OUT "J'ai le plaisir d e "
$DUPOND
$TOURNESOL
! $defined 1
OUT "convier a la reun ion du " $DATE .
NOM := ""Dupond & Dupo nt""
ADRESSE := ""Comissari at""
NOMBRE := "2"
GENRE := "1"
MARIE := "0"
NOM := ""Tournesol""
ADRESSE := ""Moulinsar d""
NOMBRE := "1"
GENRE := "1"
MARIE := "0"
NOMBRE := "1"
$NOMBRE > 1 || $define d ( 1 )
OUT "vous"
OUT "te"

# ANNEXE B

# Bibliographie

## B.1 Introduction

Cette bibliographie reflète la diversité des domaines abordés dans notre travail. Comme il l'est expliqué tout au long de cette thèse cette diversité est liée à l'utilisation d'abstractions : alors que la programmation globale a été pendant longtemps un domaine à part avec ces concepts et ses notions propres, il apparait chaque jour plus clairement que les connexions avec d'autres domaines peuvent être fructueux, pourvu que se l'on place à un niveau d'abstraction suffisant. Développer cette idée est justement l'un des objectifs de cette thèse.

Les références bibliographiques citées ci-dessous font partie d'une base bibliographique plus conséquente développée par l'auteur au cours des dernières années. Cette base rassemble plus de 350 pages de bibliographie commentée et indexée par mots clés ainsi que des informations sommaires relatives à plus de 240 systèmes, 500 auteurs, 60 pages de définitions et de citations triées par thémes, etc.
Ces informations sont disponibles en interne sous la forme d'un hypertexte et sous la forme de document HTML exploitable sous le Web. Un langage de requête interactif permet également d'exploiter cette base en proposant des recherches par mots clés. Une partie de ces informations sera prochainement disponible à l'adresse "http://www-lsr..imag.fr/users/Jean-Marie.Favre". Pour plus d'informations ne pas hésiter à contacter l'auteur (jmfavre@imag.fr).

Note: cette bibliographie ne contient que peu de références concernant les langages de programmation modernes, ces derniers étant supposés connus. Le lecteur trouvera dans [Favr89] une étude plus détaillée des relations entre ces langages et la programmation globale.

# B.2 Références bibliographiques


[Abac89]

***A. Abacus***; "**Parameterizing C Code at Compile and Run Time**", in Structured Programming, Vol. 10, N. 4, Springer Verlag, 1989, pp. 209-214.

[Abri77]

***J.R. Abrial***; "**Manuel du langage Z**", in Technical Report, Notes Z1 à Z15 non publiées, EDF, Paris, (France), 1977.

[AebiLarg94]

***D. Aebi***, ***R. Largo***; "**Methods and Tools for Data Value Reengineering**", in Lecture Notes in Computer Science, No. 819, 1st International Conference on Applications of Databases, ADB-94, Vadstena (Sweden), June 1994, pp. 400-411

[AdamWT89]

***R. Adams***, ***A. Weinert***, ***W.F. Tichy***; "**Software Change Dynamics, or Half of All Ada Compilations are Redundant**" , in Lecture Notes in Computer Science, No. 387, Second European Software Engineering Conference, Warwick, England, September 1989, pp. 203-221.

[Ahme94]

***M. Ahmed-Nacer***; "**Gestion et évolution de schémas pour les bases de données de génie logiciel**", PhD. dissertation, Université Joseph Fourier, Grenoble (France), juillet 1994.

[AhoSU89]

***A. Aho***, ***R. Sethi***, ***J. Ullman***; "**Compilateurs - Principes, techniques et outils**", ISBN 2-7296-0295-X, InterEditions, Paris, 1989, 875 pages.

[AikeMR94]

***P. Aiken***, ***M. Munro***, ***R. Richards***; "**DoD Legacy Systems Reverse Engineering Data Requirements**", in Communications of the ACM, May, 1994, pp. 26-41.

[Alzo92]

***R. Al-Zoubi***; "**Attributed Graph-Based Representation for Software View Generation and Impact-of-change Analysis**", PhD dissertation, University of Michigan, 1992., 174 pages.

[AmbrOday88]

***J. Ambras***, ***V. O'Day***; "**MicroScope: A Knowledge-Based Programming Environment**", in IEEE Software, May 1988, pp. 50-58.

[Arno86]

***R.S. Arnold***; "**Tutorial on Software Restructuring**", IEEE Computer Society Press, ISBN 0-8186-0680-0, 1986, 365 pages.

[ArnoMart86]

***R.S. Arnold***, ***R. J. Martin***; "**Software Maintenance. Guest Editors' Introduction**", in IEEE Software, May 1986, pp. 4-5.

[Arno89]

***R.S. Arnold***; "**Software Restructuring**", in Proc. IEEE, Vol. 77, No. 4, April 1989, pp. 607-617.

[Arno92]

***R.S. Arnold***; "**Common Risks of Reengineering**", in Reverse Engineering Newsletter, April, 1992.

[Arno93]

***R.S. Arnold***; "**Software Reengineering**", IEEE Computer Society Press, ISBN 0-8186-3272-0, 1993, 675 pages.

[Arno94]

***R.S. Arnold***; "**Software Reengineering: A Quick History**", in Communications of the ACM, May, 1994, pp. 13-14.

[ArnoFrak91]

***R.S. Arnold***, ***W.B. Frakes***; "**Software Reuse and Reengineering**", in CASE Trends, February, 1991.

[ArseSpra94]

***J.B. Arseneau***, ***T. Spracklen***; "**Reengineering Software Modularity using Artificial Neural Networks**" , in International Conference on Artificial Neural Networks, 1994.

[Babe91]

***R.L. Baber*** ; "**Software Engineer's Reference Book Epilogue: future developments**", in Software Engineer's Reference Book, Butterworth-heinemann, ISBN 0-7506-1040-9, 1991, Chapter 63, pp. 1-15.

[BakeEick94]

***M.J. Baker***, ***S.G. Eick***; "**Visualizing Software Systems**", in Proc. 16th International Conference on Software Engineering, Sorrento, Italy, 1994, pp. 59-67.

[BankDKZ93]

***R.D. Banker***, ***S.M. Datar***, ***C.F. Kemerer***, ***D. Zweig***; "**Software Complexity and Maintenance Costs**", in Communications of the ACM, November 1993, pp. 81-94.

[BallGV90]

***R. Ballance***, ***P. Giannini***, ***M. Van De Vanter***; "**The Pan Language-Based Editing System For Integrated Development Environments**", in Proc. 4th SIGSOFT Symposium on Software Development Environments, Ivrine (California), 1990.

[BeckEich93]

***J. Beck***, ***D. Eichmann***; "**Program and Interface Slicing for Reverse Engineering**", in Proc. 15th International Conference on Software Engineering, Baltimore, Maryland, May 1993, pp.. 509-518.

[Beck93]

***J. Beck***; "**Interface slicing: a static program analysis tool for software engineering**", PhD Dissertation, Dept. Statistics & Computer Science, West Virginia University, 1993.

[Belk88]

***N. Belkhatir***; "**NOMADE : un noyau d'environnement pour la programmation globale**", PhD. dissertation, Université Joseph Fourier, Grenoble (France), 214 pages, july, 1988.

[Benn91]

***K.H. Bennett*** ; "**Automated support of software maintenance**" , in Information and Software Technology, Vol. 33, No. 1, Jan/Feb 1991, pp. 74-85.

[Benn&al91]

***K.H. Bennett***, ***B.J. Cornelius***, ***M. Munro***, ***D.J. Robson***; "**Software maintenance**", in Software Engineer's Reference Book, Butterworth-heinemann, ISBN 0-7506-1040-9, 1991, Chapter 20, pp. 1-18.

[Benn95]

***K.H. Bennett***; "**Legacy Systems: Copying with Success**", in IEEE Software, January 1995, pp. 19-23.

[Bers&al88]

***B. Berstein***, ***V. Kruskal***, ***N. Sarnak***; "**Creation and Maintenance of Multiple Versions**" in Proc. International Workshop on Software Version and Configuration Control, Grassau (Germany), January, 1988, pp. 264-275.

[BigeRile87]

***J. Bigelow***, ***V. Riley***; "**Manipulating Source Code in DynamicDesign**" in Proc. of Hypertext '87, Chappel Hill, (North Carolina), 1987, pp. 397-408.

[BiggMW94]

***T.J. Biggerstaff***, ***B.G. Mitbander***, ***D.E. Webster***; "**Program Understanding and the Concept Assignment Problem**", in Communications of the ACM, May, 1994, pp. 72-83.

[BohmJaco66]

***C. Bohm***, ***G. Jacopini***; "**Flow diagrams, Turing machines, and languages with only two formation rules**", in Communications of the ACM, May 1966, pp. 366-371.

[Boeh76]

***B.W. Boehm***; "**Software Engineering**", in IEEE Transactions on Computers, December 1976, pp. 1226-1241

[Borr&al88]

***P. Borras***, ***D. Clement***, ***T. Despeyroux***, ***J. Incerpi***, ***G. Kahn***, ***B. Lang***, ***V. Pascual***; "**CENTAUR: The System**", in Proc. SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston (Mass.), 1988.

[BourA94]

***P. Bourque***, ***A. Abran***; "**An innovative software reengineering tools workshop - a test of market maturity and lessons learned**", in ACM SIGSOFT, Software Engineering Notes, Vol. 19, No. 3, July 1994, pp. 30-34.

[Broo87]

***F.P. Brooks***; "**No Silver Bullet. Essence and Accidents of Software Engineering**", in IEEE Computer, April 1987, pp. 10-19.

[Brow74]

***P.J. Brown***; "**Macro processors and techniques for Portable Software**", Wiley series in computing, ISBN 0 471 11005 1, 1974, 244 pages.

[Brow77]

***P.J. Brown***; "**Software Portability**", ISBN 0-521-21485-8, Cambridge University Press, Cambridge, 1977.

[Brow80]

***P.J. Brown***; "**Why Does Software Dies?**", Pergamon Infotech State of the Art Report, "Life Cycle Management", Pergamon Infotech Ltd., 1980.

[BrowLevi93]

***M.R. Brown***, ***R. Levin***; "**Bridges: Tools to Extend the Vesta Configuration Management System**", Technical report 108, System Research Center, Digital, June 1993, 60 pages.

[BursKM90]

***S. Burson***, ***G.B. Kotik***, ***L. Markosian***; "**A Program Transformation Approach to Automating Software Re-engineering**", in Proc. 14th International Computer Software & Applications Conference, 1990.

[Buss&al94]

***E. Buss***, ***R. DeMori***, ***W. M. Gentleman***, ***J. Henshaw***, ***H. Johnson***, ***K. Kontogiannis***, ***E. Merlo***, ***H.A. Muller***, ***J. Mylopoulos***, ***S. Paul***, ***A. Prakash***, ***M. Stanley***, ***S.R. Tilley***, ***J. Troster***, ***K. Wong***; "**Investigating Reverse Engineering Technologies: The CAS Program Understanding Project**", in IBM Systems Journal, Vol. 33, No. 4, 1994.

[Call91]

***F.W. Calliss***; "**A Comparison of Module Constructs in Programming Languages**" in SIGPLAN Notices, Vol. 26, N. 1, January, 1991, pp. 38-46.

[CartShep95]

***M. Cartwright***, ***M. Shepperd***; "**Maintenance: The Future of Object Orientation**" in Proc. of the 9th European Workshop on Software Maintenance, Durham (England), September 1995.

[Casa96]

***R. Casallas***; "**Objets historiques annotés dans les environnements guidés par les procédés de fabrication de logiciels**", Thèse, Université Joseph Fourier, Grenoble, février 1996.

[CeriCres83]

***S. Ceri***, ***S Crespi-Reghizzi***; "**Relational Data Bases in the Design of Program Construction Systems**" in SIGPLAN Notices, Vol. 18, N. 11, November 1983, pp. 34-44.

[CheaHT79]

***T.E. Cheatham***, ***N. Habermann***, ***J.A. Townley***; "**Symbolic Evaluation and the Analysis of Programs**" in IEEE Transactions on Software Engineering, Vol. 5, N. 4, July 1979, pp. 402-417.

[Chea85]

***T.E. Cheatham*** editor; Proc. of the "**Workshop on Software Engineering Environments for Programming-in-the-Large**", GTE Laboratories Incorporated, Harwichport, Massachusetts, 27-29 June, 1985.

[Chen95]

***Y.F. Chen*** ; "**Reverse engineering**", Chapter 6 of "Practical Reusable Unix Software", John Wiley & Sons, pp. 177-208.

[ChenRama89]

***Y.F. Chen***, ***C.V. Ramamoorthy***; "**The C Information Abstractor**" in Proc. 13th International Computer Software and Applications Conference, Orlando, Florida, September, 1989.

[ChikCros90]

***E.J. Chikofsky***, ***J.H. Cross***; "**Reverse Engineering and Design Recovery : A Taxonomy**", in IEEE Software, January 1990, pp. 54-58.

[ChiuLevi93]

***S.Y. Chiu***, ***R. Levin***; "**The Vesta Repository: A File System Extension for Software Development**", Technical report 106, System Research Center, Digital, June 1993, 60 pages.

[ChoiScac89]

***S. C. Choi***, ***W. Scacchi***; "**Assuring the Correctness of Configured Software Description**", in Proc. of the 2nd International Workshop on Software Configuration Management, Princeton, (New Jersey), October, 1989.

[Clev89]

***L. Cleveland***; "**A program understanding support environment**" in IBM Systems Journal, Vol. 28, No. 2, 1989, pp. 324-344.

[ClifCrok93]

***J. Clifford***, ***A. Croker***; "**The Historical Relational Data Model (HRDM) Revisited**", Chapter 1, pp. 6-27, in [Tans&al93].

[ClifCT93]

***J. Clifford***, ***A. Croker***, ***A. Tuzhilin***; "**On the Completeness of Query Languages for Grouped and Ungrouped Historical Data Models**", Chapter 20, in [Tans&al93]..

[CoenD91]

***A. Coen-Porisini***, ***F. DePaoli***; "**SESADA : An Environment Supporting Software Specialization**" in Lecture Notes in Computer Science, No. 550, Springer Verlag, ESEC'91, Proc. 3rd European Software Engineering Conference, Milano, Italy, October 1991, pp. 266-289.

[CoenDGM91]

***A. Coen-Porisini***, ***F. DePaoli***, ***C. Ghezzi***, ***D. Mandrioli***; "**Software Specialization Via Symbolic Execution**" in IEEE Transactions on Software Engineering, Vol, 17, N. 9, September 1991, pp. 884-899.

[ConrDW86]

***R. Conradi***, ***T.M. Didriksen***, ***D.H. Wanvik***, editors; "**Advanced Programming Environments**", LNCS, Vol. 244, Springer Verlag, ISBN 3-540-17189-4, Proc. of an International Workshop, Trondheim, Norway, June 1986.

[Cons94]

***M. Consens***; "**Creating and Filtering Structural Data Visualizations using Hygraph Patterns**", PhD thesis, Technical report CSRI-302, Department of Computer Science, University of Toronto, February 1994, 144 pages.

[ConsMR92]

***M. Consens***, ***A. Mendelzon***, ***A. Ryman***; "**Visualizing and Querying Software Structures**" in Proc. 14th International Conference on Software Engineering, Melbourne, Australia, May, 1992.

[Corb89]

***T.A. Corbi***; "**Program understanding: Challenge for the 1990s**", in IBM Systems Journal, Vol. 28, No. 2, 1989, pp. 294-306.

[CowaInce91]

***P.D. Coward***, ***D.C. Ince***; "**The Role of Symbolic Execution in Software Maintenance**" in Journal of Software Maintenance: Research and Practice, Vol. 3, 1991, pp. 183-192.

[CurtKSI87]

***B. Curtis***, ***H. Krasner***, ***V. Shen***, ***N. Iscoe***; "**On Building Software Process Models Under the Lamppost**", in Proc. 9th International Conference on Software Engineering, Monterey, (California), April, 1987, pp. 96-103.

[Dart91]

***S. Dart***; "**Concepts in Configuration Management Systems**", in Proc. of the 3rd International Workshop on Software Configuration Management, Trondheim, (Norway), June 1991.

[Dart92]

***S. Dart***; "**The Past, Present, and Future of Configuration Management**", Technical Report CMU/SEI-92-TR-8, ESC-TR-92-8, Software Engineering Institute, Carnegie Mellon University, 31 pages.

[DartCB93]

***S. Dart***, ***A.M. Christie***, ***A.W. Brown***; "**A Case Study in Software Maintenance**", Technical Report CMU/SEI-93-TR-8, ESC-TR-93-185, Software Engineering Institute, Carnegie Mellon University, 51 pages.

[DartEFH87]

***S. Dart***, ***B. Ellison***, ***P.H. Feiler***, ***N. Habermann***; "**Software Development Environments**", in IEEE Computer, Vol. 20, No. 11, November 1987, pp. 18-28.

[Davi92]

***A.M. Davis***; "**Why Industry Often Says 'No Thanks' to Research**" in IEEE Software, November 1992, pp. 97-99.

[DeloAdib82]

***C. Delobel***, ***M. Adiba***, "**Bases de données et systèmes relationels**", Edition Seuil, 1982

[Dean93]

***G. Dean*** ; "**A Survey of Configuration Languages**", technical report SE.5.93, Lancaster University, England, 1993, 9 pages.

[Dene93]

***E. Denert***; "**Software Engineering in Business and Academia, How Wide is the Gap ?**" in Lecture Notes in Computer Science, No. 717, Springer Verlag, Software Engineering ESEC'93, 4th European Software Engineering Conference, Germany, September 1993, pp. 769-783.

[DereKron76]

***F. DeRemer***, ***H. Kron***; "**Programming-in-the-Large vs. Programming-in-the-Small**" in IEEE Transactions on Software Engineering, Vol. 2, N. 2, February 1976 pp. 80-86.

[DietCall92]

***S.W. Dietrich***, ***F.W. Calliss***; "**A Conceptual Design for a Code Analysis Knowledge Base**" in Journal of Software Maintenance: Research and Practice, Vol. 4, N. 1, March 1992, pp. 19-36.

[Dijk68]

***E.W. Dijkstra***; "**Go To Statement Considered Harmful**", in Communications of the ACM, Vol. 11, No. 3, 1968, pp. 147-148.

[DionAB93]

***B. Dion***, ***L. Angeli***, ***A. Bravo Lastra***; "**ParaGraph: An interactive environment for**

**parallelizing FORTRAN programs**", rapport de recherche INRIA no. 1929, Mai 1993

[Donz&al84]

***V. Donzeau-Gouge***, ***G. Kahn***, ***B. Lang***, ***B. Mélèse***; **"Document structure and modularity in Mentor",** in Proc. SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, Pittsburgh (PA), 1984, pp. 141-148.

[EmmeSW93]

***W. Emmerich***, ***W. Schafer***, ***J. Welsh***; "**Databases for Software Engineering Environments**: **The Goal has not yet been attained**" in Lecture Notes in Computer Science, No. 717, Springer Verlag, Software Engineering ESEC'93, 4th European Software Engineering Conference, Germany, September 1993, pp. 145-162.

[EickSS92]

***S.G. Eick***, ***J.L. Steffen***, ***E.E. Sumner***; "**Seesoft - A Tool For Visualizing Line Oriented Software Statistics**" , in IEEE Transactions on Software Engineering, Vol. 18, N. 11, November 1992, pp. 957-968.

[Estu85]

***J. Estublier***; "**A Configuration Manager : The Adele Data Base of Programs**" in Proc. of the Workshop on Software Engineering Environments for Programming-in-the-Large, Harwichport (Massachusetts), 1985.

[EstuDeni88]

***J. Estublier***, ***J.P. Denier***; "**Software Maintenance: A Survey**" in Workshop on Software Engineering & its Applications, Toulouse (France), 1988.

[EstuCasa94]

***J. Estublier***, ***R. Casallas***; "**The Adele Software Configuration Manager**", Chapter 4 of [Tichy94], pages 99-139.

[EstuCasa95]

***J. Estublier***, ***R. Casallas***; "**Three dimensional versioning**" in Proc. of the 5th International Workshop on Software Configuration Management, Seattle, USA, April 1995.

[EstuFavr89]

***J. Estublier***, ***J.M. Favre***; "**Structuring Large Versioned Software Products**" in Proc. 13th International Computer Software and Applications Conference, Orlando, Florida, September, 1989, pp. 404-411.

[Favr88]

***J.M. Favre***; "**Représentation multi-langages des programmes pour la programmation globale**", Rapport de DEA, Laboratoire de Génie Informatique, Institut National Polytechnique de Grenoble, 1988.

[Favr89]

***J.M. Favre***; "**Olga : Un Noyau Multi-langage Pour Nomade**" in 2nd International Conference On Software Engineering & its Applications, Toulouse, (France),1989, pp. 717-730.

[Favr93]

***J.M. Favre***; "**Vers une representation multi-langages et multi-versions des programmes**" in 6th International Conference On Software Engineering & its Applications, Paris, (France), 1993, pp. 459-468.

[Favr94a]

***J.M. Favre***; "**Reengineering-In-The-Large vs. Reengineering-In-The-Small**" in 1st SEI Workshop on Reengineering, Software Engineering Institute, Carnegie Mellon University, Pittsburgh (Pennsylvania) 3-5 May, 1994.

[Favr94b]

***J.M. Favre***; "**Support for Reengineering-In-The-Large**", Proc. of the CAiSE'94 Doctoral Consortium, International Conference on Engineering Information System, 6-10 june, utrecht (Netherlands), Memoranda Informatica 94-24, University of twente, 1994.

[Favr95a]

***J.M. Favre***; "**The CPP Paradox**" in Proc. of the 9th European Workshop on Software Maintenance, Durham (England), September 1995.

[Favr95b]

***J.M. Favre***; "**Maintenance et ré-ingénierie globale en présence de préprocesseurs**", 8ième conférence internationale, le génie logiciel et ses applications. Novembre 1995.

[Favr96a]

***J.M. Favre***; "**Reverse Engineering and Configuration Management: Concepts and Perspectives**", in Proc. of Software Conf'96, Paris (France), 11-12 June, 1996.

[Favr95b]

***J.M. Favre***; "**Preprocessors from an abstract point of view**", in International Conference on Software Maintenance-1996, Monterey (California), November 1996.

[Fram88]

***M. Frame***; "**A Lexical Comparaison Program to Simplify the Maintenance of Portable Software**" in International Conference on Software Maintenance-1988, Phoenix, (Arizona), October 1988, pp. 348-350.

[Feil91a]

***P.H. Feiler***; "**Configuration Management Models in Commercial Environments**", Technical Report CMU/SEI-91-TR-7, ESD-91-TR-7, Software Engineering Institute, Carnegie Mellon University, March 1991, 58 pages.

[Feil91b]

***P.H. Feiler*** editor; Proc. of the "**3rd International Workshop on Software Configuration Management**", Trondheim, Norway, June 12-14, 1991, ISBN 0-89791-429-5, ACM-Press, 1991.

[Feld79]

***S.I. Feldman***; "**Make - A Program for Maintaining Computer Programs**", in Software - Practice and Experience, 9:4, April, 1979, pp. 255-265.

[FielHarr88]

***J. Field***, ***P.G. Harrison*** ; "**Functional Programming**", ISBN 0-201-19249-7, Addison-Wesley, 1988., 602 pages.

[Fort92]

***G. Forte***; "**Tools Fair: Out of the Lab, Onto the Shelf**", in IEEE Software, May 1992, pp. 70-79.

[Fowl90]

***G.S. Fowler***; "**A Case for make**", in Software - Practice and Experience, 20, Issue No. S1, June, 1990, pp. 35-47.

[Fowl&al95a]

***G.S. Fowler***, ***D. Korn***, ***S.C North***, ***H. Rao***, ***K.P. Vo***; "**Libraries and File System Architecture**", Chapter 2 of "Practical Reusable Unix Software", John Wiley & Sons, pp. 25-88.

[Fowl&al95b]

***G.S. Fowler***, ***D. Korn***, ***H. Rao***, ***R. Snyder***, ***K.P. Vo***; "**Configuration Management**", Chapter 3 of "Practical Reusable Unix Software", John Wiley & Sons, pp. 91-120.

[FrasMyer87]

***C.W. Fraser***, ***E.W. Myers***; "**An Editor for Revision Control**" in ACM Transactions on Programming Languages and Systems, Vol. 9, N. 2, April 1987, pp 277-295.

[Fugg93]

***A. Fuggetta***; "**A Classification of CASE Technology**", in IEEE Computer, December 1993, pp. 25-38.

[GadiNair93]

***S. Gadia***, ***S.S. Nair***; "**Temporal Databases: A Prelude to Parametric Data**", Chapter 2, pp. 28-66. in [Tans&al93].

[Gall89]

***B. Galler***; "**Toughts on Software Engineering**", in Proc. 11th International Conference on Software Engineering, 1989, pp. 97.

[GallKlos93]

***H. Gall***, ***R. Klosch***; "**Capsule Oriented Reverse Engineering for Software Reuse**" in Lecture Notes in Computer Science, No. 717, Springer Verlag, Software Engineering ESEC'93, 4th European Software Engineering Conference, Germany, September 1993, pp. 418-433.

[Gent89]

***W. M. Gentleman***; "**Managing Configurability in Multi-installation Realtime Programs**", in Proc. of the Canadian Conference on Electrical and Computer Engineering, Vancouver (British Columbia), November 1989, pp. 823-827.

[GentMSC89]

***W. M. Gentleman***, ***S.A. McKay***, ***D.A. Stewart***, ***W. Cowan***; "**Commercial Realtime Software Needs Different Configuration Management**", in Proc. of the 2nd International Workshop on Software Configuration Management, Princeton, (New Jersey), October, 1989, pp. 152-161.

[GlasNois81]

***R.L. Glass***, ***R.A. Noiseux***; "**Software Maintenance Guidebook**", Prentice-Hall, ISBN 0-13-821728-9, 1981, 193 pages.

[Gold86]

***R. Goldberg***; "**Software Engineering: An emerging discipline**", in IBM Systems Journal, Vol. 25, No. 3, 1986, pp. 334-353

[GrasChen90]

***J.E. Grass***, ***Y.F. Chen***; "**The C++ Information Abstractor**" in USENIX, C++ Conference Proc., San Francisco (CA), April, 1990.

[Gras92a]

***J.E. Grass***; "**Object-Oriented Design Archeology with CIA++**" in USENIX, Computing

Systems, Vol. 5, N. 1, 1992.

[Gras92b]

***J.E. Grass***; "**Cdiff: A Syntax directed Differencer for C++ Programs**" in USENIX, C++ Conference Proc., Portland (Oregon), August, 1992.

[GrosSnelt93]

***F.J. Grosch***, ***G. Snelting***; "**Polymorphic Components for Monomorphic Languages**" in Proc. of the 2nd International Workshop on Software Reusability, pages 47-55, Lucca, Italy, March 1993, pp. 47-55.

[Gull92a]

***B. Gulla***; "**A Browser for Versioned Entity-Relationship Database**" , in Proc. International Workshop on Interfaces to Database Systems, Glasgow, Scotland, July 1992.

[Gull92b]

***B. Gulla***; "**Improved Maintenance Support by Multi-Version Visualizations**" , in International Conference on Software Maintenance-1992.

[Gull93]

***B. Gulla***; "**The constraint diagram: an approach to visualizing the version space**" , in Proc. of the 4th International Workshop on Software Configuration Management, Baltimore, Maryland, USA, May 1993, pp. 112-122.

[GullKY91]

***B. Gulla***, ***E.A. Karlsson***, ***D. Yeh***; "**Change-Oriented Version Descriptions in EPOS**", in Software Engineering Journal, 6(6), November 1991, pp. 378-386.

[HabeNotk86]

***N. Habermann***, ***N. Notkin***; "**Gandalf: Software Development Environments**", in IEEE Transactions on Software Engineering, Vol. 12, N. 12, December 1986.

[HaleHawo88]

***D.P. Hale***, ***D.A. Haworth***; "**Software Maintenance: A Profile of Past Empirical Research**", in International Conference on Software Maintenance-1988, Phoenix, (Arizona), October 1988, pp. 236-240.

[HammCham93]

***M. Hammer***, ***J. Champy***; "**Le reengineering**", Dunod, 1993.

[Hann91]

***M.A. Hanna***; "**Getting Back to Requirements Proving to Be a Difficult Task**" in Software Magazine, October 1991.

[HannLevi93]

***C.B. Hanna***, ***R. Levin***; "**The Vesta Language for Configuration Management**", Technical report 107, System Research Center, Digital, June 1993, 60 pages.

[HendDG91]

***J. Henderson***, ***R. Dowsing***, ***D. Graham***; "**Practical program development issues**", Chapter 46 in Software Engineer's Reference Book, Butterworth-heinemann, ISBN 0-7506-1040-9, 1991, pp. 1-16.

[HorgMoor84]

***J.R. Horgan***, ***D.J. Moore***; "**Techniques for Improving Language-Based Editors**", in Proc. SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments,

Pittsburgh (PA), 1984, pp.7-14.

[Horw85]

***S. Horwitz***; "**Generating language-based editors: a relationally-attributed approach**", Technical Report TR85-696, PhD dissertation, Computer Sciences Department, Cornell University, 1985.

[Horw90a]

***S. Horwitz***; "**Identifying the Semantic and Textual Differences Between Two Versions of a Program**", in SIGPLAN Notices, Vol. 25, N. 6, June, 1990, pp. 234-245.

[Horw90b]

***S. Horwitz***; "**Adding Relational Query Facilities to Software Development Environments**" in Theorical Computer Science, 73:2, 1990.

[HorwReps92]

***S. Horwitz***, ***T. Reps***; "**The Use of Program Dependence Graphs in Software Engineering**", in Proc. 14th International Conference on Software Engineering, Melbourne, Australia, May, 1992, pp. 392-411.

[HorwTeit87]

***S. Horwitz***, ***T. Teitelbaum***; "**Generating Editing Environments Based on Relations and Attributes**" in ACM Transactions on Programming Languages and Systems, Vol. 9, N. 2, April 1987, pp. 577-608.

[Ince84]

***D.C. Ince***; "**The Provision of Procedural and Functional Interfaces for the maintenance of Program Design Language and Program Language Notations**" in SIGPLAN Notices, Vol. 19, N. 2, February 1984, pp. 68-74.

[IngrBW93]

***P. Ingram***, ***C. Burrows***, ***I. Wesley***; "**Configuration Management Tools: a Detailed Evaluation**", ISBN 0-903960-82-3, Ovum Ltd., London, 1993.

[Jaco94]

***G. Jacob***; "**Le reengineering**", Hermes, Paris, 137 pages

[Jaco91]

***I. Jacobson***; "**Re-engineering of old systems to an object-oriented architecture**" in Communications of the ACM, May, 1991, pp. 340-350.

[JarzTan94]

***S. Jarzabek***, ***C.L. Tan***; "**Modeling multiple views of common features in software reengineering for reuse**", in Lecture Notes in Computer Science, No. 811, 6th International Conference on Advanced Information Systems Engineering, CAiSE'94, Utrecht (Netherlands), June 1994, pp. 268-282.

[Jone90]

***C.F. Jones*** ; "**Systematic Software Development Using VDM**", Prentice Hall, 1990.

[JoneGS93]

***N.D. Jones***, ***C.K. Gomard***, ***P. Sestoft***; "**Partial Evaluation and Automatic Program Generation**", Prentice Hall, ISBN 0-13-020249-5, 1993, 415 pages.

[Kame87]

***R.F. Kamel***; "**Effect of Modularity on System Evolution**", in IEEE Software, January

1987, pp. 48-54

[Katz90]

***R. Katz***; "**Toward a Unified Framework for Version Modeling in Engineering Databases**" in ACM Computing Survey Vol. 22, N. 4, December 1990, pp. 375-408.

[KeabRM88]

***J. Keables***, ***K. Roberson***, ***A. Mayrhauser***; "**Data Flow Analysis and its Application to Software Maintenance**", in International Conference on Software Maintenance-1988, Phoenix, (Arizona), October 1988, 335-347.

[King81]

***J.C. King***; "**Program Reduction using Symbolic Execution**" in ACM SIGSOFT, Software Engineering Notes, Vol. 6, N. 1, January 1981, pp. 9-14.

[KozaLN91]

***W. Kozaczynski***, ***S. Letovsky***, ***J. Q. Ning***; "**A Knowledge-Based Approach to Software System Understanding**", in Proc. 6th Annual Knowledge-Based Software Engineering Conference, 1991, pp. 162-170.

[Krak87]

***S. Krakowiak*** ; "**Principes des systèmes d'exploitation des ordinateurs**", Dunod informatique, Seconde édition, ISBN-2-04-018632-8, 1987, 486 pages.

[Kras91]

***P. Krass***; "**Building A Better Mousetrap**", in Information week, March 1991, pp. 24-30.

[Kris95]

***B. Krishnamurthy***, editor; "**Practical Reusable Unix Software**", John Wiley & Sons, ISBN 0-471-05807-6, 1995, 370 pages.

[KronSnel94]

***M. Krone***, ***G. Snelting***; "**On the Inference of Configuration Structures from Source Code**" in Proc. 16th International Conference on Software Engineering, Sorrento, Italy, 1994.

[Krus83]

***V. Kruskal***; "**Managing Multi-Version Programs with an Editor**" in Research Report, RC 10217 (#45003), IBM Thomas J. Watson Research Center, P.O. box 218, Yorktown Heights, New York 10598, August 83.

[Lafu90]

***G.M.E. Lafue***; "**Panel on Software Re-Engineering**", in Proc. 12th International Conference on Software Engineering, Nice (France), May 1990.

[LakeBlan95]

***T. Lake***, ***T. Blanchard***; "**Reverse Engineering of Assembler Programs using a TDF-based Intermediate Language**" in Proc. of the 9th European Workshop on Software Maintenance, Durham (England), September 1995.

[Lako93]

***A. Lakhotia***; "**Understanding someone else's code: Analysis of experiences**", in Reverse Engineering Newsletter, a publication of the Subcommittee on Reverse Engineering of the IEEE Computer Society, January, 1993.

[LampSchmi83]

***B.W. Lampson***, ***E.E. Schmidt***; "**Practical Use of a Polymorphic Applicative Language**", in Proc. 10th Symposium on Principles of Programming Languages, ACM , January, 1983.

[LangSchw91]

***R. Lange***, ***R.W. Schwanke***; "**Software Architecture Analysis : A Case Study** " , in Proc. of the 3rd International Workshop on Software Configuration Management, Trondheim, (Norway), June 1991, pp. 19-28.

[LanoHaug92]

***K. Lano***, ***H. Haughton***; "**Extracting Design and Functionality form Code**" in Proc. 5th International Workshop on Computer Aided Software Engineering, Montréal (Canada), July 1992, pp. 74-82.

[LiRama85]

***C.H. Li***, ***J. Ramanathan***; "**Beyond Isolated Systems for Programming-in-the-Small and Development-in-the-large**" in Proc. of the Workshop on Software Engineering Environments for Programming-in-the-Large, Harwichport (Massachusetts), 1985.

[LeblChas87]

***D.B. Leblang***, ***R.P. Chase***; "**Computer-aided Software Engineering in a Distributed Workstation Environment**" in Proc. 2nd SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environment, Palo Alto, California, December 1986. In ACM SIGPLAN Notices Vol. 22, N. 1, January 1987.

[LehmBela85]

***M.M. Lehman***, ***L. Belady***; "**Program Evolution. Processes of Software Change**", Academic Press, London, 1985.

[Lehm80]

***M.M. Lehman***; "**Programs, life cycles, and laws of software evolution**" in Proc. IEEE, Vol. 68, No. 9, September 1980, pp. 35-41.

[LeluSalo86]

***P. Leluc***, ***Y. Salomon***; "**Enquête sur les coûts de la maintenance**", in Génie logiciel, N. 5, Juillet 1986, pp. 68-71.

[LeviMcjo93]

***R. Levin***, ***P.R. McJones***; "**The Vesta Approach to Precise Configuration of Large Software Systems**", Technical report 105, System Research Center, Digital, June 1993, 38 pages.

[LetjMR92]

***M. Letjer***, ***S. Meyers***, ***S.P. Reiss***; "**Support for Maintaining Object-Oriented Programs**" in IEEE Transactions on Software Engineering, Vol. 18, N. 12, December 1992, pp. 1045-1063.

[Lewe88]

***C. Lewerentz***; "**Extended Programming in the Large in a Software Development Environment**" in Proc. SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston (Mass.), 1988, pp. 173-182.

[Lie90]

***A. Lie***; "**Versioning in Software Engineering Database**" , PhD dissertation, Division of Computer Systems and Telematics, The Norwegian Institute of Technology, 1990.

[Lint84]

***M. Linton***; "**Implementing Relational Views of Programs**" in Proc. SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, Pittsburgh (PA), 1984, pp. 132-140.

[Litm93]

***A. Litman***; "**An Implementation of Precompiled Headers**" in Software - Practice and Experience, 23:3, March 1993, pp. 341-350.

[LivaAlde93]

***P. E. Livadas***, ***S.D. Alden***; "**A Toolset for Program Understanding**", in Technical Report SERC-TR-64-F, Software Engineering Research Center, University of Florida, Available via ftp : ftp.cs.purdue.edu:pub/serc, June 1993.

[LivaCrol92]

***P. E. Livadas***, ***S. Croll***; "**Program Slicing**", in Technical Report SERC-TR-61-F, Software Engineering Research Center, University of Florida, Available via ftp : ftp.cs.purdue.edu:pub/serc, October 1992.

[LivaSmal94]

***P. E. Livadas***, ***D.T. Small***; "**Understanding Code Containing Preprocessor Construct**", in IEEE Third Workshop on Program Comprehension, Washington, November 1994.

[Lore93]

***N.A. Lorentzos*** ; "**The Interval-extended Relational Model and Its Application to Valid-time Databases**", Chapter 3, pp. 67-91. in [Tans&al93].

[Luba86]

***M. Lubars***; "**Code Reusability in the large versus Code Reusability in the Small**", in ACM SIGSOFT, Software Engineering Notes, Vol. 11, N. 1 January 1986.

[Lyle94]

***J.R. Lyle***; "**Program Slicing**", in Encyclopedia of Software Engineering, J.J. Marchiniak Editor, John Wiley and Sons, 1994, pp. 873-877.

[LyleGall88]

***J.R. Lyle***, ***K.B. Gallagher***; "**Using Program Decomposition to Guide Modifications**", in International Conference on Software Maintenance-1988, Phoenix, (Arizona), October 1988, pp. 265-269.

[Lyon81]

***M.J. Lyons*** ; "**Salvaging your software asset (tools based maintenance)**", in AFIPS Conference Proceedings, Vol. 50, 1981, pp. 337-341.

[MafnAM93]

***B. Magnusson***, ***U. Asklund***, ***S. Minor***; "**A model for Semi-(a)Synchronous Collaborative Editing**" , in Proc. of ACM SIGSOFT'93 Symposium on the Foundations of Software Engineering, Los Angeles (California), 1993.

[Mahl94]

***A. Mahler***; "**Variants: Keeping Things Together and Telling Them Apart**" , in [Tich94], Chapter 3, 1994.

[MahlLamp88]

***A. Mahler***, ***A. Lampen***; "**An Integrated Toolset for Engineering Software Configurations**" , in Proc. SIGSOFT/SIGPLAN Software Engineering Symposium on

Practical Software Development Environments, Boston (Mass.), 1988, pp. 191-200.

[Marc94a]

***J.J. Marciniak*** Editor; "**Software Engineering Encyclopedy**", John Wiley and Son's, 1994

[Marc94b]

***J.J. Marciniak***; "**Software Engineering, A Historical Perspective**", in Encyclopedia of Software Engineering, J.J. Marchiniak Editor, John Wiley and Sons, 1994, pp. 1176-1183.

[MarkBK94]

***L. Markosian***, ***R. Brand***, ***G.B. Kotik***; "**Customized Software Evaluation Tools: Application of an Enabling Technology for Reengineering**", in Proc. of the Fourth Systems Reengineering Technology Workshop, Monterey, (California), February 1994.

[MarkNBBK94]

***L. Markosian***, ***P. Newcomb***, ***R. Brand***, ***S. Burson***, ***T. Kitzmiller***; "**Using an Enabling Technology to Reengineer Legacy Systems**", in Communications of the ACM, May, 1994, pp. 58-70.

[Mcde91a]

***J.A. McDermid*** Editor; "**Software Engineer's reference book**", Butterworth-HeinemannLtd, ISBN 0-750-61040-9, 1991

[Mcde91b]

***J.A. McDermid*** ; "**Software Engineer's reference book, Introduction to Part II**", in Software Engineer's Reference Book, Butterworth-heinemann, ISBN 0-7506-1040-9, 1991, 15 pages.

[McdeDenv91]

***J.A. McDermid***, ***T. Denvir***; "**Software Engineer's reference book, Introduction to Part I**", in Software Engineer's Reference Book, Butterworth-heinemann, ISBN 0-7506-1040-9, 1991, 15 pages.

[Mcle83]

***B.J. McLennan***; "**Overview of Relational Programming**", in SIGPLAN Notices, Vol. 18, N. 3, March 1983.

[Melo93]

***W. Melo***; "**Un Environnement de Développement Logiciel Centré Procédés de Fabrication**", PhD. dissertation, Université Joseph Fourier, Grenoble (France), 202 pages, october 1993.

[Mend93]

***A. Mendelzon*** (editor); "**Declarative Database Visualization: Recent Papers From the HY+/Graphlog project**", technical report CSRI-285, Computer Systems Research Institute, University of Toronto, Canada, June 1993, 157 pages.

[MerkDC92]

***E. Merks***, ***j. Dyck***, ***R. Cameron***; "**Language Design for Program Manipulation**" in IEEE Transactions on Software Engineering, Vol. 18, N. 1, January 1992.

[Meye85]

***B. Meyer***; "**The Software Knowledge Base**" in Proc. 8th International Conference on Software Engineering, London, (United Kingdom), 1985, pp. 158-165.

[Meye92]

***B. Meyer*** ; "**Effeil, The Language**", Prentice-Hall, ISBN-0-13-247925-7, 1992.

[Meye93]

***S. Meyers***; "**Representing Software Systems In Multiple-View Development Environments**". Ph.D. dissertation, Brown University Department of Computer Science, may 1993.

[MIL-srah94]

"**Software Reengineering Assessment Handbook**", MIL-HDBK-SRAH, Version 1.0, Software Technology Support Center, US Air Force, February 1994.

[Morg84]

***H.W. Morgan***; "**Evolution of a Software Maintenance Tool**" in Proc. of the 2nd National Conference on EDP Software Maintenance, 1984, pp. 268-278.

[Mull86]

***H.A. Muller***; "**Rigi - A Model for Software System Construction, Integration and Evolution based on Module Interface Specifications**", PhD Dissertation, Rice University, Houston Texas, 182 pages, 1986

[MullTOC92]

***H.A. Muller***, ***S.R. Tilley***, ***M.A. Orgun***, ***B.D. Corrie***; "**A Reverse Engineering Environment Based on Spatial and Visual Software Interconnection Models**" in ACM SIGSOFT, Software Engineering Notes, Vol. 17, No. 5, December 1992, Proc. of the 5th Symposium on Software Development Environments, pp. 88-98.

[Munc93]

***B.P. Munch***; "**Versioning in Software Engineering Database : the Change Oriented Way**" , PhD dissertation, Division of Computer Systems and Telematics, The Norwegian Institute of Technology, 1993.

[Munc&al93]

***B.P. Munch***, ***J.O. Larsen***, ***B. Gulla***, ***R. Conradi***, ***E.A. Karlsson***; "**Uniform Versioning: The Change-Oriented Model**" , in Proc. of the 4th International Workshop on Software Configuration Management, Baltimore, Maryland, USA, May 1993.

[Murr92]

***B. Murray***; "**A Statically Typed Abstracted Representation for C++ Programs**" in USENIX, C++ Conference Proc., Portland (Oregon), August, 1992, pp. 83-97.

[Nara88a]

***K. Narayanaswamy***; "**Version Control in the Common Lisp Framework**" in Proc. International Workshop on Software Version and Configuration Control, Grassau (Germany), January, 1988.

[Nara88b]

***K. Narayanaswamy***; "**Static Analysis-Based Program Evolution Support in the Common Lisp Framework**" in Proc. 10th International Conference on Software Engineering, 1988, pp. 222-230.

[Nara89]

***K. Narayanaswamy***; "**A Text-Based Representation for Program Variants**" in Proc. of the 2nd International Workshop on Software Configuration Management, Princeton, (New Jersey), October, 1989, pp. 30-33.

[Newb93]

***F. Newbery Paulisch***; "**The Design of an Extendible Graph Editor**" , in Lecture Notes in Computer Science, No. 704, Springer Verlag, June 1993.

[Nick91]

***P.J. Nicklin***; "**Managing Multi-Variant Software Configurations**" in Proc. of the 3rd International Workshop on Software Configuration Management, Trondheim, (Norway), June 1991, pp. 53-57.

[NingEK92]

***J. Q. Ning***, ***A. Engberts***, ***W. Kozaczynski***; "**Program Concept Recognition and Transformation**", in IEEE Transactions on Software Engineering, Vol. 18, N. 12, December 1992, pp. 1065-1075.

[NormC92]

***R.J. Norman***, ***M. Chen***; "**Working Together to Integrate CASE**", in IEEE Software, March 1992, pp. 12-16

[Oman90]

***P. Oman***; "**Maintenance Tools**" in IEEE Software, Vol. 7, N. 3, May 1990, pp. 59-66.

[OtteOtte84]

***K.J. Ottenstein***, ***L.M. Ottenstein***; "**The Program Dependence Graph in a Software Development Environment**", in SIGPLAN Notices, Vol. 19, N. 5, May 1984, pp. 177-184.

[OtteEllc92]

***K.J. Ottenstein***, ***S.J. Ellcey***; "**Experience Compiling Fortran to Program Dependence Graphs**", in Software - Practice and Experience, 22:1, January 1992, pp. 41-62.

[Paga91]

***F.G. Pagan***; "**Partial Computation and the Construction of Language Processors**", Prentice Hall, ISBN 0-13-651415-4, 1991, 166 pages.

[Pari86a]

***G. Parikh***; "**Handbook of Software Maintenance**", John Wiley & Sons, New York, 1986.

[Pari86b]

***G. Parikh***; "**Software Maintenance Notes (1)**",in ACM SIGSOFT, Software Engineering Notes, Vol. 11, N. 2, April 1986, pp. 49-57.

[Pari86c]

***G. Parikh***; "**Software Maintenance Notes (2)**",in ACM SIGSOFT, Software Engineering Notes, Vol. 11, N. 5, October1986, pp. 86-91.

[PariZveg83]

***G. Parikh***, ***N. Zvegintzov***; "**Tutorial on Software Maintenance**", Los Alamitos, California, 1983.

[Parn72]

***D.L. Parnas***; "**On the criteria to be used in decomposing systems into modules**" in Communications of the ACM, December, 1972, pp. 1053-1058.

[Parn94]

***D.L. Parnas***; "**Software Aging**", in Proc. 16th International Conference on Software Engineering, Sorrento, Italy, 1994, pp. 279-287.

[Paul95]

***S. Paul***; "**Theory and Design of Source Code Search Systems**", PhD thesis, University of Michigan, 1995, 147 pages.

[PaulPrak94]

***S. Paul***, ***A. Prakash***; "**Supporting Queries on Source Code: A Formal Framework**", Technical report CSE-TR-209-94, University of Michigan, Computer Science and Engineering Division, Department of Electrical Engineering and Computer Science, April 1994, 20 pages.

[PaulCCW93]

***M.C. Paulk***, ***B. Curtis***, ***M.B. Chrissis***, ***C.V. Weber***; "**Capability Maturity Model, Version 1.1**", in IEEE Software, July 1993, pp. 16-27.

[PremBlah94]

***W.J. Premerlani***, ***M. Blaha***; "**An Approach for Reverse Engineering of Relational Databases**", in Communications of the ACM, May, 1994, pp. 42-49.

[Perr87]

***D. Perry***; "**Software Interconnections Models**", in Proc. 9th International Conference on Software Engineering, Monterey, (California), April, 1987, pp. 61-69.

[PerrStie93]

***D. Perry***, ***C.S. Stieg***; "**Software Faults in Evolving a Large System : a Case Study**" in Lecture Notes in Computer Science, No. 717, Springer Verlag, Software Engineering ESEC'93, 4th European Software Engineering Conference, Germany, September 1993, pp. 48-68.

[PetrWind94]

***M. Petre***, ***R. Winder***; "**Programming Languages: Models and Programming Styles**", in Encyclopedia of Software Engineering, J.J. Marchiniak Editor, John Wiley and Sons, 1994, pp. 892-900.

[Pigo94]

***T.M. Pigoski***; "**Maintenance**", in Encyclopedia of Software Engineering, J.J. Marchiniak Editor, John Wiley and Sons, 1994, pp. 617-636.

[PlaiWadg93a]

***J. Plaice***, ***W. Wadge***; "**A New Approach to Version Control**" in IEEE Transactions on Software Engineering, Vol. 19, N. 3, March 1993, pp. 268-276.

[PlaiWadg93b]

***J. Plaice***, ***W. Wadge***; "**A Unix Tool for Managing Reusable Software Components**" in Software - Practice and Experience, 23:9, September 1993, pp. 933-948.

[Pigo94]

***R. Prieto-Diaz*** ; "**Module Interconnection Languages**", in Encyclopedia of Software Engineering, J.J. Marchiniak Editor, John Wiley and Sons, 1994, pp. 703-706.

[PietNeig89]

***R. Prieto-Diaz***, ***J.M. Neighbors***; "**Module Interconnection Languages - a survey**", in Proc. of the 2nd International Workshop on Software Configuration Management, Princeton, (New Jersey), October, 1989.

[Phil83]

***J.C. Philips***; "**Creating a Baseline for an Undocumented System - or What Do You Do**

**With Someone Else Code**", in [Arno86], pp. 244-246

[Quil94]

***A. Quilici***; "**A Memory-Based Approach to Recognizing Programming Plans**" in Communications of the ACM, May, 1994, pp. 84-93.

[RamaGP86]

***C.V. Ramamoorthy***, ***V. Garg***, ***A. Prakash***; "**Programming in the large**" in IEEE Transactions on Software Engineering, Vol. 12, N. 7, July 1986, pp. 769-783.

[RamaPU84]

***C.V. Ramamoorthy***, ***A. Prakash***, ***Y. Usuda***; "**Software Engineering : Problems and Perspectives**", in IEEE Computer, October 1984, pp.191-207.

[Raul95]

***J.C. Rault***; "**Maintenance, rénovation et renouveau**", in Génie logiciel, N. 35, Mars 1995.

[RedwRidd85]

***S.T. Redwine***, ***W.E. Riddle***; "**Software Technology Maturation**", in Proc. 8th International Conference on Software Engineering, London, (United Kingdom), 1985, pp. 189-200.

[Rebo94]

"**Software Reuse : A Holistic Approach**", Wiley & Sons, 1994.

[Reis90]

***S.P. Reiss***; "**Interacting with the FIELD Environment**" in Software - Practice and Experience, 20, Issue No. S1, June, 1990, pp. 89-115.

[Reko85]

***M.G. Rekoff***; "**On Reverse Engineering**", in IEEE Trans. on Systems, Man, and Cybernetics, March-April, 1985.

[RepsHP88]

***T. Reps***, ***S. Horwitz***, ***J. Prins***; "**Support for Integrating Program Variants in an Environment for Programming in the Large**", in Proc. International Workshop on Software Version and Configuration Control, Grassau (Germany), January, 1988, pp. 197-216.

[RepsT84]

***T. Reps***, ***T. Teitelbaum***, "**The Synthesizer Generator**" in Proc. SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, Pittsburgh (PA), 1984.

[Robe87]

***P. Robert***, "**Petit Robert. Dictionnaire de langage Française**", ISBN 2-85036-066-X, 1987.

[RobsBCM91]

***D.J. Robson***, ***K.H. Bennett***, ***B.J. Cornelius***, ***M. Munro***; "**Approaches to Program Comprehension**", in Journal of Systems and Software, Vol. 14, Feb. 1991, pp. 79-84.

[Roch74]

***M.J. Rochkind***; "**The Source Code Control System (SCCS)**" in IEEE Transactions on Software Engineering, Vol. 1, December 1974, pp. 370-376.

[RomaCox93]

***G.C. Roman***, ***K.C. Cox***; "**A Taxonomy of Program Visualization Systems**", in IEEE Computer, December 1993, pp. 11-23.

[Romb87]

***H.D. Rombach***; "**A Controlled Experiment on the impact of Software Structure on Maintainability**" in IEEE Transactions on Software Engineering, Vol. 13, N. 3, March 1987, pp. 344-354.

[Rose95]

***D. Rosenblum***; "**Self-Checking Programs and Program Instrumentation**", Chapter 5 of "Practical Reusable Unix Software", John Wiley & Sons, pp. 159-176

[RoseWolf91]

***D. Rosenblum***, ***A.L. Wolf***; "**Representing Semantically Analyzed C++ Code with Reprise**" in USENIX, 4rd C++ Conference Proc., Washington (DC), April, 1991.

[Ross86]

***G. Ross***; "**Integral C - A Practical Environment for C Programming**", in Proc. 2nd SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environment, Palo Alto, California, December 1986. In ACM SIGPLAN Notices Vol. 22, N. 1, January 1987, pp. 42-48.

[Scac94]

***W. Scacchi***; "**Technology Transfer**", in Encyclopedia of Software Engineering, J.J. Marchiniak Editor, John Wiley and Sons, 1994, pp. 1323-1327.

[Schn87]

***N.F. Schneidewind***; "**The State of Software Maintenance**" in IEEE Transactions on Software Engineering, Vol. 13, N. 3, March 1987, pp. 303-310.

[Schr89]

***U. Schroeder***; "**Incremental Variant Control**" in Proc. of the 2nd International Workshop on Software Configuration Management, Princeton, (New Jersey), October, 1989, pp. 145-148.

[SchwStra91]

***R.W. Schwanke***, ***V.A. Strack***; "**An Intelligent Tool For Re-engineering Software Modularity**" , in Proc 13th International Conference on Software Engineering, 1991, pp. 83-92

[SchwStra93]

***R.W. Schwanke***, ***V.A. Strack***; "**Configuration Management Problems and Architectural Integrity**" , in Proc. of the 4th International Workshop on Software Configuration Management, Baltimore, Maryland, USA, May 1993, pp. 225-228.

[SchwPlat89]

***R.W. Schwanke***, ***M.A. Platoff*** ; "**Cross References are Features**" , in Proc. of the 2nd International Workshop on Software Configuration Management, Princeton, (New Jersey), October, 1989, pp. 86-95.

[Scio91]

***E. Sciore***; "**Multidimensional Versioning for Object-Oriented Databases**" in Proc. Second International Conference on Deductive and Object-Oriented Databases, 1991.

[Scio94]

***E. Sciore***; "**Versioning and Configuration Management in an Object-Oriented Data Model**" , in VLDB Journal, Vol 3, N. 1, January 1994.

[ShanSnod89]

***K. Shannon***, ***R. Snodgrass***; "**Mapping the Interface Description Language Type Model into C"**, in IEEE Transactions on Software Engineering, Vol. 15, N. 11, November 1989, pp. 1333-1346.

[Shaw84]

***M. Shaw***; "**Abstraction techniques in Modern Programming Languages**", in IEEE Software, October 1984, pp. 10-26.

[Shaw86]

***M. Shaw***; "**Beyond Programming-in-the-large : The Next Challenges for Software Engineering** " in Lecture Notes in Computer Science, No. 244, Springer Verlag, Proceedings of the International Workshop on Advanced Programming Environments, Trondheim (Norway), June 1986, pp. 520-535.

[Shaw89]

***M. Shaw***; "**Remembrances of a Graduate Student**", in Proc. 11th International Conference on Software Engineering, 1989, pp. 99-100.

[Shaw90]

***M. Shaw***; "**Prospects for an Engineering Discipline of Software**", in IEEE Software, November 1990, pp. 15-24

[Sing92]

***P. Singleton***; "**A Case for Declarative Programming-in-the-Large**", PhD dissertation, Technical Report TR94-05, Department of Computer Science, University of Keele, Keele, Staffs, ST5 5BG, 1992, 335 pages.

[SingBrer92]

***P. Singleton***, ***O.P. Brereton***; "**Building Software by Deduction: Why and How**", Technical Report TR92-17, Department of Computer Science, University of Keele, Keele, Staffs, ST5 5BG, December 1992, 17 pages

[SingBrer93]

***P. Singleton***, ***O.P. Brereton***; "**A Case for Declarative Programming-in-the-Large**", in Proc. 5th International Conference on Software Engineering and Knowledge Engineering, San Francisco, California, 1993.

[StasPatt91]

***J.T. Stasko***, ***C. Patterson***; "**Understanding and Characterizing Program Visualization Systems**", Technical Report GIT-GVU-91-17, College of Computing, Georgia Institute of Technology, pp. 412-420, 1991.

[Stef85]

***J.L. Steffen***; "**Interactive examination of C program with Cscope**", in USENIX, Winter 1985 Conference, January, 1985, pp. 170-175

[Smds90]

***SMDS***; "**Aide-De-Camp, Product Overview**" , Technical Report, SMDS, P.O. Box 555, MA 01742, USA, 1990.

[SmitMulaSmit90]

***M.C. Smith***, ***D.E. Mularz***, ***T.J. Smith***; "**Case Tools Supporting Ada Reverse Engineering: State of the Practice**", in Eighth National Conference on Ada Technology, 1990, pp. 157-164

[SmitOman90]

***D.B. Smith***, ***P. Oman***; "**Software Tools In Context**", in IEEE Software, Vol. 7, N. 3, May 1990, pp. 15-19.

[SneeJand87]

***H.M. Sneed***, ***G. Jandrasics***; "**Software Recycling**", in International Conference on Software Maintenance-1987, pp. 82-86.

[Snel95]

***G. Snelting***; "**Reengineering of Configurations Based on Mathematical Concept Analysis**", Computer science report 94-02,Technical University of Braunschweig, Germany, January 1995, 28 pages.

[Snod84]

***R. Snodgrass***; "**Monitoring in a Software Development Environment: A Relational Approach**" in Proc. SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, Pittsburgh (PA), 1984, pp. 124-131.

[Snod93]

***R. Snodgrass***; "**An Overview of TQuel**", Chapter 6, pp. 141-182, in [Tans&al93].

[SoniNH95]

***D. Soni***, ***R.L. Nord***, ***C. Hofmeister***; "**Software Architecture in Industrial Applications**", in Proc. 17th International Conference on Software Engineering, Seattle, Washington, 1995, pp. 196-207.

[SpenColly92]

***H. Spencer***, ***G. Collyer***; "**#ifdef Considered Harmful, or Portability Experience With C News**" in USENIX, Summer 1992 Technical Conference, San Antonio (Texas), June, 1992, pp. 185-197.

[Spie76]

***M. J. Spier***; "**Software Malpractice - A Distasteful Experience**", in Software - Practice and Experience, 6, 1976, pp. 293-299.

[Spul90]

***D. Spuler***; "**Check: A Better Checker for C**", Thesis, James Crook University, Australia 1990, 106 pages.

[SpulSaje92]

***D. Spuler***, ***A.S.M. Sajeev***; "**Static Detection of Preprocessor Macro Errors in C**", Technical report 92-7, James Crook University, 1992, 18 pages.

[Stan84]

***T.A. Standish***; "**An essay on software reuse**", in IEEE Transactions on Software Engineering, Vol. 10, N. 5, 1984, pp. 494-497.

[Stal92]

***R. Stallman***; "**The C Preprocessor**", GNU Project, Free software foundation, July 1992, 52 pages.

[Stei95]

***A.A. Steiner***; "**Rompre avec le train-train de la maintenance**", in Le monde Informatique, N. 643, juillet 1995, pp. 22-25.

[Stro91]

***B. Stroustrup*** ; "**The C++ Programming language**", ISBN 0-201-53992-6 Addison-Wesley, 1991, 695 pages.

[Stro94]

***B. Stroustrup*** ; "**Design and Evolution of C++**", .ISBN 0-201-54330-3, ATT, 1994

[Swan76]

***E.B. Swanson***; "**The Dimensions of Maintenance**", in Proc. 2nd International Conference on Software Engineering, 1976, pp. 492-497.

[Tans&al93]

***A.U. Tansel***, ***J. Clifford***, ***S. Gadia***, ***S. Jajodia***, ***A. Segev***, ***R. Snodgrass*** ; Editors. "**Temporal Databases : Theory, Design, and Implementation**", The Benjamin/Cummings Publishing Company, Inc. , 390 Bridge Parkway, Redwood City, California 94065, 1993, 635 pages.

[Tich85]

***W.F. Tichy***; "**RCS - A System for Version Control**" in Software - Practice and Experience, 15:7, July 1985, pp. 637-654.

[Tich88]

***W.F. Tichy***; "**Tools for Software Configuration Management**", in Proc. International Workshop on Software Version and Configuration Control, Grassau (Germany), January, 1988

[Tich92]

***W.F. Tichy***; "**Programming-in-the-Large: Past, Present, and Future**" in Proc. 14th International Conference on Software Engineering, Melbourne, Australia, May, 1992, pp. 362-366.

[Tich94]

***W.F. Tichy***, Editor; "**Configuration Management**", Trends in Software 2, ISBN 0-471-94245-6, John Wiley & Sons, 1994.

[TilbCroo92]

***D. Tilbrook***, ***R. Crook***; "**Large Scale Porting through Parametrization**" in USENIX, Summer 1992 Technical Conference, San Antonio (Texas), June, 1992, pp. 209-216.

[Till93]

***S.R. Tilley***; "**Documenting-in-the-large vs. Documenting-in-the-small**" in Proc. of the IBM CAS Conference, CASCON'93, October 1993, pp. 1083-1090.

[TillMull93]

***S.R. Tilley***, ***H.A. Muller*** (Editors); "**The Rigi Compendium, Volume II**", University of Victoria, 1993, 131 pages.

[TillWMS93]

***S.R. Tilley***, ***M.J. Whitney***, ***H.A. Muller***, ***M.A.D. Storey***; "**Personalized Information Structures**" in 11th International Conference on Systems Documentation, Waterloo (Ontario), October 1993, pp. 325-337.

[Tip94]

***F. Tip***; "**A Survey of Program Slicing Techniques**", technical report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994, 58 pages.

[Toma94]

***J.E. Tomayko***; "**Milestones in Software Engineering**", in Encyclopedia of Software Engineering, J.J. Marchiniak Editor, John Wiley and Sons, 1994, pp.687-697.

[TrygCG93]

***E. Tryggeseth***, ***R. Conradi***, ***B. Gulla***; "**Software Configuration Management in PROTEUS**" , in Proc. of the 4th International Workshop on Software Configuration Management, Baltimore, Maryland, USA, May 1993.

[TrygGull95]

***E. Tryggeseth***, ***B. Gulla***; "**Comprehensive Variability Modelling with the Proteus Configuration Language**" , in Proc. of the 5th International Workshop on Software Configuration Management, Seattle, USA, April 1995.

[Ulri90]

***W.M. Ulrich***; "**The Evolutionary Growth of Software Reengineering and the Decade Ahead**" in American Programmer, Vol.3, No. 10, October 1990, pp. 14-20.

[Ulri93]

***W.M. Ulrich***; "**Formal Method Needed to Assist Reengineering, Redevelopment Effort**" in CASE Trends, November, 1993.

[Unif]

"**unifdef**", Unix man pages

[Vand93]

***M. Van De Vanter***, "**Algebras for Object-Oriented Query Languages**", Ph.D. dissertation, University of Wisconsin, Madison, 1993, 194 pages

[Venk95]

***G.A. Venkatesh*** ; "**Experimental Results from Dynamic Slicing of C Programs**", in ACM Transactions on Programming Languages and Systems, Vol. 17, N. 2, 1995, 197-216.

[VoChen92]

***K.P. Vo***, ***Y.F. Chen***; "**Incl: A Tool to Analyze Include Files**" in USENIX, Summer 1992 Technical Conference, San Antonio (Texas), June, 1992, pp. 199-208.

[VottPort95]

***L.G. Votta***, ***A. Porter***; "**Experimental Software Engineering: A Report on the State of the Art**", in Proc. 17th International Conference on Software Engineering, Seattle, Washington, 1995, pp. 277-379.

[Waru66]

***A. Warusfel***, "**Dictionnaire Raisonné de Mathématiques**", Edition Seuil, 1966.

[WattWF87]

***D.A. Watt***, ***B.A. Wichmann***, ***W. Findlay***; "**ADA, Language and Methodology**", Prentice-hall international, series in computer science, ISBN 0130040789, 1987, 518 pages.

[Wegn84]

***P. Wegner***; "**Capital-Intensive Software Technology. Part 2 : Programming in the large**" in IEEE Software, July 1984, pp. 24-32.

[WeidHH95]

***B.W. Weide***, ***W.D. Heym***, ***J.E. Hollingsworth***; "**Reverse Engineering of Legacy Code Exposed**", in Proc. 17th International Conference on Software Engineering, Seattle,

Washington, 1995, pp. 327-331.

[Wein83]

***G.M. Weinberg***; "**Kill That Code!**", Infosystems, August 1983.

[WeinCG92]

***M. Wein***, ***W. Cowan***, ***W. M. Gentleman***; "**Commercial Realtime Software Needs Different Configuration Management**" , in Proc. of the 1992 ACM/SIGAPP Symposium on Applied Computing (SAC'92), Kansas City (Kansas), March 1992.

[Weis84]

***M. Weiser***; "**Program Slicing**", in IEEE Transactions on Software Engineering, Vol. 10, N. 4, July 1984, pp. 352-357.

[WeisCrew93]

***D. Weise***, ***R. Crew***; "**Programmable Syntax Macros**", in ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, 1993, pp. 156-165.

[West89]

***B. Westfechtel***; "**Revision Control in an Integrated Software Development Environment**", in Proc. of the 2nd International Workshop on Software Configuration Management, Princeton, (New Jersey), October, 1989, pp. 96-105.

[West91]

***B. Westfechtel***; "**Structure-Oriented Merging of Revisions of Software Documents**" in Proc. of the 3rd International Workshop on Software Configuration Management, Trondheim, (Norway), June 1991, pp. 68-79.

[West92]

***B. Westfechtel***; "**A Graph-Based Approach to the Construction of Tools for the Life Cycle Integration between Software Documents**" in Proc. 5th International Workshop on Computer Aided Software Engineering, Montréal (Canada), July 1992, pp. 2-13.

[WinkStof88]

***J.F.H. Winkler***, ***C. Stoffel***; "**Program-Variations-in-the-Small**", in Proc. International Workshop on Software Version and Configuration Control, Grassau (Germany), January, 1988, pp. 175-196.

[Wink87]

***J.F.H. Winkler***; "**Version Control in Families of Large Programs**", in Proc. 9th International Conference on Software Engineering, Monterey, (California), April, 1987, pp. 150-161.

[Wink86]

***J.F.H. Winkler***; "**The Integration of Version Control into Programming Languages**", in Lecture Notes in Computer Science, No. 244, Springer Verlag, Proceedings of the International Workshop on Advanced Programming Environments, Trondheim (Norway), June 1986, pp. 230-250.

[Wink86a]

***T.C. Winkler*** editor; Proc. of the "**International Workshop on Software Version and Configuration Control**", Grassau, FRG, 27-29 January, 1988.

[Whit91]

***D. Whitgift***; "**Software Configuration Management: Methods and Tools**", John Wiley and Sons, West Sussex, England, July 1991.

[WolfCW88]

***A.L. Wolf***, ***L.A. Clarke***, ***J.C. Wilden***; "**A Model of Visibility Control**" in IEEE Transactions on Software Engineering, Vol. 14, N. 4, April, 1988, pp. 512-520.

[Word92]

***J.B. Wordsworth*** ; "**Software Development with Z**", ISBN 0-201-62757-4, Addison-Wesley, 1992.

[Yang91]

***W. Yang***; "**Identifying syntactic differences between two programs**" in Software - Practice and Experience, 21:7, July 1991, pp. 739-755.

[Your89]

***E. Yourdon***; "**Re-3 : Re-engineering, Restructuring, Reverse Engineering**" in American Programmer, Vol.2, No. 4, April 1989, pp. 3-10.

[Yu91]

***D. Yu***; "**A View on Three R's (3Rs): Reuse, Re-engineering, and Reverse Engineering**", in ACM SIGSOFT, Software Engineering Notes, Vol. 16, No. 3, July 1991, pp. 69

[Zell94]

***A. Zeller***; "**Configuration Management with Feature Logics**", technical report 94-01, Technical University of Braunschweig (Germany), March, 1994, 48 pages.

[ZellSnel94]

***A. Zeller***, ***G. Snelting***; "**Incremental Configuration Management Based on Feature Unification**", technical report 94-04, Technical University of Braunschweig (Germany), April, 1994, 13 pages.

# TABLES DES MATIERES

# INDEX

## #



## A



## C



## D

# E



# F



# G



# I

# L



# M



# O

# P



# R



# S

## T



## V

# LISTE DES FIGURES

De nombreuses figures illustrent le discours tenu dans cette thèse (mises bout à bout elles représentent plus de 50 pages...). La plupart de ces figures représentent des transformations d'information. Dans certains cas la distinction entre données et traitement et particulièrement importante, notamment dans le cas où l'on décrit des processus d'évaluation partielle. Des conventions précises ont été prises pour les notations utilisées dans ces figures. Celles-ci sont décrites dans la figure 142 (p.296).

*¡ Colorín, colorado,*
*este cuento, se ha acabado !*

## RESUME

Alors que l'informatique est résolument tournée vers l'avenir, cette thèse se concentre sur le passé ; non pas par nostalgie mais plutôt parce que le futur des logiciels âgés est une question d'actualité. Plus particulièrement trois thèmes sont abordés : la *maintenance*, la *ré-ingénierie* et la *programmation globale*. L'objectif de cette étude est d'explorer l'intersection entre la ré-ingénierie et la programmation globale, domaine que nous avons baptisé *ré-ingénierie globale*. L'idée principale est de réutiliser des concepts et des techniques de programmation détaillée. En fait nous proposons de définir la programmation globale en distinguant 4 aspects du logiciel : l'*architecture*, la *manufacture*, la *variation* et l'*évolution*. Un modèle abstrait basé sur des concepts ensemblistes est proposé pour modéliser les différentes entités de programmation globale. La technologie relative à ce domaine est ensuite décrite en considérant aussi bien l'état de l'art que l'état de la pratique. La différence entre ces deux aspects souligne l'intérêt de la ré-ingénierie globale. A titre d'illustration, nous étudions le cas des *préprocesseurs* en tant qu'outils de programmation globale. Ces outils de bas niveau provenant des années 70 sont traditionnellement utilisés pour décrire des *familles de programmes*. Pour faciliter la compréhension de tels artefacts nous proposons d'utiliser des techniques comme l'*évaluation partielle*, l'*analyse inter-procédurale de flot de données* ou encore le *découpage*. Ces techniques, définies de manière rigoureuse, s'appuient sur la *sémantique dénotationnelle* du préprocesseur utilisé. Un prototype montrant leur utilité a été réalisé. Il prouve qu'une approche basée sur des fondements théoriques issus de la programmation détaillée est applicable pour résoudre des problèmes pratiques de programmation globale.

## MOTS CLES

Génie Logiciel, Maintenance, Ré-ingénierie, Retro-ingénierie, Programmation globale, Gestion de configurations, Préprocesseur, Macro-processeur

## ABSTRACT

While computing turns to the future, this thesis focuses on the past; legacy software systems are and will continue to be maintained. Three themes are essential in the context of this study: Software Maintenance, Software Reengineering and Programming-In-The-Large (PITL). The main goal of this thesis is to explore the intersection between Reengineering and Programming-In-The-Large, a domain that we coin "Reengineering-In-The-Large". The key idea is to prove that Programming-in-the-small concepts can be used to solve practical Programming-In-The-Large problems. We studied PTIL according to 4 different views: architecture, manufacture, variation and evolution. We also defined an abstract model based on set theory concepts to formalise PITL entities. The PITL technology is studied. The differences between the State-Of-The-Art and the State-Of-The-Practice emphasizes the need for Reengineering-In-The-Large. Preprocessors are studied as special cases of PITL tools. These low-level tools are traditionally used to describe program families. In order to make their comprehension easier, we use techniques like partial evaluation, inter-procedural data-flow analysis and slicing. A prototype which combines these different techniques was implemented to demonstrate that an approach based on theoretical Programming-In-The-Small concepts can be used to solve practical Programming-In-The-Large problems.

## KEYWORDS

Software Engineering, Maintenance, Reengineering, Reverse-Engineering, Programming-In-The-Large, Configuration Management, Preprocessor, Macro-processor