UNIVERSITE DE NANCY I

Centre de Recherche en Informatique de Nancy

THESE

AMELIORATION ASSISTEE DE PROGRAMMES PAR OBJECTIFS

Présentée et soutenue publiquement le 8 Décembre 1989
pour l'obtention du titre de

Docteur de l'Université de Nancy I en informatique

par

Charles BARTHELEMY

Composition du Jury :

M. Jean Pierre FINANCE	*Professeur de l'Université de Nancy I*	*Président*
M. Jean Claude BOUSSARD	*Professeur de l'Université de Nice*	*Rapporteur externe*
Mme Odile FOUCAUT	*Professeur de l'Université de Nancy II*	*Rapporteur interne*
M. Jean Claude DERNIAME	*Professeur de l'Université de Nancy I*	*Examinateur*
M. Norbert HERSTCHUH	*Maître de conférence de l'Université de Nancy II*	*Invité*

UNIVERSITE DE NANCY I

Centre de Recherche en Informatique de Nancy

THESE

AMELIORATION ASSISTEE DE PROGRAMMES PAR OBJECTIFS

Présentée et soutenue publiquement le 8 Décembre 1989
pour l'obtention du titre de

Docteur de l'Université de Nancy I en informatique

par

Charles BARTHELEMY

Composition du Jury :

M. Jean Pierre FINANCE	*Professeur de l'Université de Nancy I*	*Président*
M. Jean Claude BOUSSARD	*Professeur de l'Université de Nice*	*Rapporteur externe*
Mme Odile FOUCAUT	*Professeur de l'Université de Nancy II*	*Rapporteur interne*
M. Jean Claude DERNIAME	*Professeur de l'Université de Nancy I*	*Examinateur*
M. Norbert HERSTCHUH	*Maître de conférence de l'Université de Nancy II*	*Invité*

A

Roseline,
Amélie,
Aloïs

Qu'il me soit permis d'adresser publiquement mes remerciements

à Jean-Pierre FINANCE, Professeur de l'université de Nancy I, directeur du C.R.I.N, pour l'honneur qu'il me fait de présider le jury

à l'ensemble des membres du jury pour avoir aussi été ceux qui, en juin 1987, avec Marion CREHANGE, Professeur de l'université de Nancy II, m'ont aidé à définir un objectif et donné la première impulsion à la dynamique qui m'a permis, aujourd'hui, de l'atteindre

à Jean-Claude DERNIAME, Professeur de l'université de Nancy I pour avoir supervisé mon travail, pour les conseils qu'il m'a prodigués, pour la confiance qu'il n'a cessé de m'accorder depuis longtemps déjà

à Odile FOUCAUT, Professeur de l'université de Nancy II, pour avoir accepté la responsabilité de rapporteur interne, pour ses critiques et propositions constructives déterminantes dans mon travail

à Monsieur Jean-Claude BOUSSARD Professeur de l'université de Nice (L.I.S.A.N.) d'avoir accepté la charge de rapporteur externe et donc pour l'honneur qu'il me fait d'être membre du jury

à Norbert HERTSCHUH, Maître de conférence de l'université de Nancy II, mon précieux collègue de l'I.U.T. pour la surcharge de travail qu'il a acceptée d'une part pour la mise au point d'un algorithme d'arborisation de graphe, d'autre part pour alléger ma charge d'enseignement ces deux dernières années

à Hassan AYOUB-BAHSOUN pour le soutien constant qu'il m'a apporté, en particulier pour les travaux publiés conjointement, pour ses compétences et pour les qualités de coeur dont il a fait preuve

à Jean-Luc REMY , Chargé de recherches (C.N.R.S.), pour l'aide précieuse qu'il m'a accordée lors de l'élaboration d'un algorithme d'assainissement du texte source d'un programme usant de son esprit critique et de ses compétences en matière de formulation et de formalisation des algorithmes

à Alain QUERE, Maître de conférence de l'université de Nancy I, pour sa collaboration à la mise au point d'un algorithme de recherche de modules dans le texte source d'un programme et pour le puits de connaissances qu'il a été pour moi

à Monsieur et Madame PARMENTIER sans qui la réalisation matérielle de grande qualité de cette thèse n'aurait pas été possible

à mes collègues de l'I.U.T. et en particulier à Hervé COILLAND, Maître de conférence de l'université de Nancy II, Directeur du Département Informatique de l'I.U.T. de Nancy, pour le soutien moral et logistique dont il ne s'est pas départi à mon égard et à Michèle CONRAD pour les responsabilités pédagogiques et administratives dont elle a accepté de me soulager pendant cette période de préparation de ma thèse

à tous et à beaucoup d'autres encore, merci et à ma femme aussi.

PLAN DE THESE

PARTIE II :

PROPOSITIONS POUR UN SYSTEME D'AMELIORATION DE LOGICIELS.
Les modèles, le processus

INTRODUCTION

CHAPITRE 1 :
FINALITE ET FONCTIONS D'UN SYSTEME D'AIDE A L'AMELIORATION DE LOGICIELS

1 1 : Finalité d'un système d'aide à l'amélioration de logiciels : SAAL
1 2 : Quelques caractéristiques du processus d'amélioration
1 3 : Les fonctions du SAAL
- 1 3 1 : Les fonctions de représentation et de gestion des objets définissant les programmes
- 1 3 2 : Les fonctions liées à la partie dynamique du processus
- 1 3 3 : Les fonctions mixtes

1 4 : L'aspect dynamique de l'amélioration du logiciel.
- 1 4 1 : Exemple de concepts, tâches et étapes d'un projet d'amélioration
  - 1 4 1 1 : Les objectifs généraux du projet
  - 1 4 1 2 : La cible d'amélioration
  - 1 4 1 3 : L'arbre d'amélioration
  - 1 4 1 4 : L'évaluation
  - 1 4 1 5 : La mise en oeuvre des fonctions d'amélioration
  - 1 4 1 6 : Conclusion : le processus et les modèles associés

CHAPITRE 2 :
UN MODELE CONCEPTUEL GENERIQUE DE REPRESENTATION DES PROGRAMMES

2 1 : Un modèle générique des données
- 2 1 1 : Modèles d'appui : le modèle structurel hiérarchique
  le modèle E R C.
  - 2 1 1 1 : Le modèle structurel (MS)
  - 2 1 1 2 : Le modèle E.R.C.

2 2 : Le modèle générique proposé : les concepts

- 2 2 1 : Approche générique du modèle
- 2 2 2 : La structure des composants du modèle générique de données
- 2 2 3 : Un modèle générique de données pour l'amélioration de programmes
- 2 2 4 : Représentation graphique d'un modèle de données

2 3 : Conclusion

CHAPITRE 3 :
UN MODELE GENERIQUE DE PROCESSUS D'AMELIORATION.

3 1 : Introduction
3 2 : Le processus d'amélioration
- 3 2 1 : Finalité de l'amélioration
- 3 2 2 : Différents niveaux de système d'amélioration
- 3 2 3 : Définition du modèle de processus d'amélioration
- 3 2 4 : Eléments de présentation et de mise en oeuvre du modèle

3 3 : La mise en œuvre du processus d'amélioration
3 4 : Conclusion

CHAPITRE 4 :
VERS LA REALISATION DU SYSTEME D'AMELIORATION DE LOGICIEL.

4 1 : Introduction
4 2 : L'amélioration de logiciels et les nouvelles fonctions des SGBD
- 4 2 1 : Les limitations des SGBD actuels
- 4 2 2 : Les fonctionnalités multimédia
- 4 2 3 : Les fonctionnalités déductives
- 4 2 4 : Les fonctionnalités dynamiques
- 4 2 5 : Les interfaces
- 4 2 6 : Conclusion

4 3 : La place d'un système d'amélioration de logiciels (SAAL) dans un A.I.G.L.
- 4 3 1 : Evolution des environnements de programmation
- 4 3 2 : Propriétés d'un atelier intégré de génie logilciel
- 4 3 3 : AGL et processus d'amélioration du logiciel

4 4 : Conclusion

CHAPITRE 5:

DE LA DIFFICULTE PRATIQUE A AMELIORER DES PROGRAMMES : de COBOL à ADA, un EXEMPLE

CONCLUSION ET PERSPECTIVES.

PARTIE III :

ANNEXES :

INTRODUCTION

Un effort important est fait actuellement pour la création d'ateliers permettant un développement moins "artisanal" plus "industriel" de logiciels. Une caractéristique essentielle du Génie Logiciel est la volonté de couverture de toutes les phases du cycle de vie du logiciel, c'est à dire l'intégration de tous les outils et de toutes les méthodes utilisées tout au long de ce cycle de vie.

Il n'en reste pas moins que le monde de l'informatique appliquée est confronté à deux problèmes essentiels; le coût toujours élevé du logiciel par opposition à l'accessibilité croissante du matériel et la difficulté d'adaptation des programmes existants aux nouveaux environnements informatiques.

L'étude des coûts relatifs du logiciel le long de son cycle de vie laisse apparaître que les étapes de maintenance et de test représentent approximativement 75 % du coût global.

Maintenir le logiciel (correction d'erreurs et modifications après le lancement) c'est assurer son évolution (changements dans les besoins du problème, dans l'ensemble des données ou des résultats attendus), son adaptation (changement de système d'exploitation ou de matériel), et son perfectionnement (diminution du temps d'exécution, du temps de réponse,...)

Le coût élevé de la maintenance du logiciel est généralement imputé à la qualité médiocre du texte source, à l'absence de documentation et à l'absence ou aux déficiences de méthodes de correction ou d' adaptation des programmes.

On constate également que la difficulté à adapter les programmes existants aux nouveaux environnements résulte, par surcroît, de l'absence de prise en compte des bons concepts de programmation (modularité, concepts orientés objet, type abstrait,...)

A l'absence de qualité des programmes, il y aurait trois raisons essentielles :

1) l'absence de mise en oeuvre d'une bonne méthode de développement de ces programmes (spécification et explicitation de l'énoncé avant codage).
2) le niveau insuffisant des langages de programmation couramment utilisés
3) les erreurs commises par les informaticiens tant au niveau du développement que de la maintenance des programmes lors de l'utilisation des outils, méthodes et langages.

L'une des conséquences de cette absence de qualité est l'impossibilité d'une réécriture immédiate des programmes ce qui laisse l'utilisateur face à l'alternative

suivante ; continuer à maintenir le texte source (on sait ce que cela coûte) ou reprendre tout le cycle de vie du programme (on craint ce que cela coûterait).

Cependant, si certaines qualités sont absentes de nombreux programmes, d'autres peuvent être présentes et il existe ainsi un très grand nombre de programmes non modulaires, non structurés, peu ou pas évolutifs mais qui sont corrects au sens où ils fournissent les résultats attendus. D'où l'intérêt que présente un système permettant d'améliorer certaines qualités (ou diminuer certains défauts) des programmes existants tout en préservant les qualités qu'ils ont.

Cette amélioration sélective des programmes, en fonction des objectifs de l'utilisateur, consiste essentiellement en la transformation assistée des textes sources correspondants, seule information disponible en général.

Mais là est en fin une quatrième raison essentielle à l'absence de qualité des programmes et à l'inertie des utilisateurs face à la mise en oeuvre de transformations c'est la difficulté qu'il y a à transformer ces programmes du fait de l'absence de bons environnements d'aide à l'amélioration de logiciels. Il y a encore loin de la théorie de la transformation des programmes à la mise en oeuvre des mécanismes qu'elle sous-tend.

L'utilisateur confronté à des problèmes d'amélioration se heurte actuellement non seulement à l'absence, sur le marché, de bons environnements (pratiques et accessibles) d'aide à l'amélioration mais également au manque d'ouvrages de réflexion synthétique sur le sujet.

Aussi dans le cadre de ce mémoire apportons nous notre contribution à un regroupement des principaux concepts d'intérêt dans le domaine de l'amélioration de programmes qu'il s'agisse du processus lui-même ou des données manipulées.

Indépendamment d'algorithmes concernant l'assainissement du texte d'un programme, la recherche et la définition de modules à partir de ce texte, cet effort de synthèse nous a conduit également à proposer une représentation des programmes à l'aide de modèles de bases de données et prenant en compte la notion de ressource de type quelconque (texte, graphe, diagramme).

Notre proposition va également dans le sens de l'aide à l'amélioration et au suivi du processus grâce à la notion d'arbre d'amélioration et à la prise en compte de mesures de qualité.

Ce mémoire traite d'une nouvelle phase du cycle de vie des programmes; la phase d'amélioration de programmes existants; phase que l'on peut définir par :

- la donnée : le programme source à améliorer essentiellement représenté par son texte source.
- le résultat : le programme cible qui correspond à une nouvelle version du programme source dont certaines qualités ont été améliorées.

- la relation entre la donnée et le résultat : essentiellement composition de fonctions d'aide à l'amélioration de certains critères de qualité.
- les objectifs en termes de facteurs de qualité à améliorer pour le programme source étudié (maintenabilité, portabilité,...etc.)

Nous abordons, dans ce mémoire, les problèmes concernant cette phase d'amélioration en deux parties principales :

1) la première partie constitue un bilan des principales fonctions et sous-fonctions terminales portant sur le produit à améliorer (appréhender, restructurer,...etc.)
2) la seconde partie aborde les notions de méthode et de processus d'amélioration avec modélisations.

Dans la première partie avant de traiter, une à une, dans les chapitres 3 à 7, les différentes fonctions terminales de l'amélioration, nous présentons dans le chapitre 1 l'état de l'art en matière de description de programme.

La démarche générale de l'amélioration fait l'objet du 2ème chapitre.

Dans la seconde partie le chapitre 1 présente les principales caractéristiques d'un processus d'amélioration et fait appel avec une certaine spécificité aux principales fonctions terminales pour la mise en oeuvre de ce processus en mettant l'accent sur la fonction "mesurer".

Dans le chapitre 5, nous insistons sur la nécessité d'un environnement de type atelier de génie logiciel pour l'implantation d'un système d'amélioration de programmes.

Les modélisations respectives de la représentation des programmes puis du processus d'amélioration sont envisagées successivement dans les chapitres 2 et 3. Si le modèle de données est défini en termes d'unités génériques, le modèle du processus quant à lui met en oeuvre les notions d'opération, d'activité et de méthode dans le cadre de stratégies.

Le chapitre 4 est le lieu de réflexions et de propositions concernant la réalisation d'un système d'amélioration de programmes.

Plutôt que d'émailler ce mémoire de petits exemples alors plus anecdotiques que significatifs, nous proposons dans le chapitre 6 un exemple d'amélioration de programme impliquant la distance de COBOL à ADA.

Nous terminons ce mémoire par différentes conclusions et perspectives à propos de notre travail.

PARTIE I

LES FONCTIONS D'AIDE A L'AMELIORATION DE PROGRAMMES

I-1 NOTRE HYPOTHESE EN MATIERE DE DESCRIPTION DE PROGRAMME : LE MODELE MODULAIRE

1-1 INTRODUCTION : NECESSITE DE LA MODELISATION

L'amélioration du texte source passe par sa compréhension et sa transformation, elle nécessite donc la mise en oeuvre d'opérations concernant à la fois les aspects syntaxiques et sémantiques du programme.

Dans la pratique, compte tenu du taux de rotation moyen des informations, la maintenance, la réutilisation et le portage d'un logiciel sont rarement l'oeuvre de celui qui l'a conçu et réalisé.

La méthode de conception, souvent personnelle, mise en oeuvre, les possibilités du langage de programmation utilisé, le choix des instructions pour le codage et le style de rédaction du programme sans parler des modifications liées aux tests et à la maintenance conduisent à un texte source difficilement exploitable par un utilisateur qui n'a pas lui-même réalisé ces différentes tâches.

L'un des moyens d'aide à la compréhension et à la modification d'un programme considéré comme un ensemble d'objets consiste à définir ces objets et les relations qui existent entre eux.

Cela exige pratiquement la définition d'une autre version du programme étudié plus proche des spécifications du problème résolu par ce dernier.

Une manière naturelle d'approcher un problème, de façon descendante et structurée, ainsi que la prise en compte dans cette démarche de la dualité données-traitements conduisent à envisager un programme à différents niveaux :

- le niveau modulaire où l'on peut considérer le programme comme un ensemble de composants modulaires liés entre eux par des relations (appel, inclusion,...).
- le niveau logique où chaque composant modulaire constitué de fonctions possède son histoire de calculs.
- le niveau de l'instruction comme moyen élémentaire d'expression du calcul et du contrôle.
- le niveau flot de données s'appuyant sur les trois niveaux précédents pour la composition de relations sur les données manipulées.

Les modèles proposés et rappelés permettent d'envisager un programme à l'un quelconque des niveaux précédents.

Nous rappelons quelques définitions concernant les deux premiers niveaux ; quant aux niveaux instructions et flot de données, ils seront abordés dans les chapitres de la partie I concernant les fonctions d'amélioration.

1-2 LE MODELE MODULAIRE

1-2-1 Définition

A un programme P on peut faire correspondre un couple (M, R_p) défini de la manière suivante :

M = {m_i / i = 1, n ; n entier ≥ 1} est l'ensemble des composants modulaires de P.

Rp = {rj / j = 1, p ; p entier ≥ 1 } est l'ensemble de toutes les relations binaires pouvant exister entre les composants de P.

Un composant modulaire désigné, ici, toute suite d'instructions identifiable sans ambiguité par un nom et activable par ce nom.

L'ensemble R_p appartient à P (R) ensemble des parties de R où R est l'ensemble de référence des relations binaires usuelles entre deux composants.

1-2-2 Quelques définitions et notations.

a) Ensemble de référence R

R = {r_k / k = 1,... m; m entier ≥ 1} avec par exemple :

- r1 : appel direct
- r2 : utilisation d'éléments communs (données, composants,.)
- r3 : modification de valeurs de données communes
- r4 : exportation
- r5 : importation
- r6 : inclusion
- r7 : communication

L'existence de redondances éventuelles entre les relations, pas nécessairement élémentaires, peut être intéressante tant du point de vue de l'utilisateur que du point de vue de l'implémentation.

b) Ensemble de référence transposé R^{-1}

C'est l'ensemble des relations transposées de R

R^{-1} {r_i / i = 1,...,n ri (p,q) = r_i^{-1} (q,p) , (p,q) E M x M}

où M est l'ensemble des composants modulaires d'une structure modulaire générale S.

On désigne par R_u l'union des relations de R et de son transposé : Ru = RUR^{-1}

c) Ensemble Ru * des relations composées de Ru

Un élément r de Ru * est une expression logique composées d'éléments appartenant à Ru reliés entre eux par les opérateurs classiques : NON : , ET : Λ, OU : V , OU exclusif : les

Remarque :

Dans la suite nous confondrons les notations R et Rp

1-2-3 Graphes des relations.

Etant donné une structure modulaire S une représentation classique de ce modèle est un graphe ayant pour noeuds les composants modulaires de S, chaque arc représentant l'existence d'une relation entre les noeuds extrémités.

1-2-3-1 Graphe de base d'une relation.

Etant donné une structure modulaire S et une relation ri appartenant à R, ensemble des relations binaires entre les composants de S on appelle graphe de base de ri et on note G (M, u_i) le graphe tel que :

- M représente l'ensemble des composants modulaires de S
- u_i l'ensemble des arcs de G tels que

  Etant donné p,q appartenant à M

  (p,q) E U_i <=> r_i (p, q) = 1

1-2-3-2 Graphe général G (M, U)

Ce multigraphe est la synthèse des graphes de base associés à toutes les relations.

Les arcs de U représentent les occurrences des relations modulaires R.

A tout couple (p,q) E M x M on fait correspondre le vecteur V (p,q) = (v_1, v_2,...,v_n) où n est le nombre de relations dans R et v_i (p,q) = si r_i (p,q) alors 1 sinon 0

Etant donné un couple (p,q) E M x M on a (p,q) E U <=> ∃ i = 1,--,n tq r_i (p,q) = 1

Un **chemin** dans le multigraphe est **homogène** par rapport à une relation r_i E R si tous les arcs vérifient r_i, dans le cas contraire le chemin est hétérogène.

1-2-3-3 Attributs associés à une relation de Ru *

A chaque relation binaire r E R entre les éléments d'une structure modulaire S sont associés des attributs. Ces attributs permettent l'expression de propriétés ou caractéristiques propres à chaque

relation ; ils peuvent varier avec les besoins de l'utilisateur et selon q
la relation exprime :

- la structuration,
- la communication (flot de contrôle, flot de données,...),
- la synchronisation,
- l'héritage,
- ---------- etc.

Exemple :

Nous avons développé un exemple dans (BARTH.,86) à propo
de la relation **d'appel.**

Ainsi si G (M, u_i est le graphe de base pour la relation d'appel r_1, éta
donné p,q E M x M tq r_1 (p,q) on peut associer à l'arc (p,q) de u_i le
attributs suivants :

- emplacements d'activation
- nombre d'activations
- conditions d'activation par emplacement
- alternatives modulaires
- éléments utilisés par l'appel (ou modifiés).

1-2-4 Apports du modèle modulaire.

La définition même du modèle modulaire permet une approche de celui-
ci en termes de **cible** et d'**impact.** De plus, au niveau modulaire, il
contient d'une part la "syntaxe" du programme (relations modulaires de
R), d'autre part la sémantique par propagation le long de tout sous-
graphe ou graphe partiel d'informations liées à la sémantique des
variables des instructions (PIDFAA, RTSM, ESSS) et du problème ;
nous aborderons ces mécanismes dans les chapitres suivants de la
1ère partie de ce mémoire.

Quant à l'implémentation d'un tel modèle, elle fait partie des travaux
abordés dans la IIème partie de ce mémoire.

1-2-4-1 Cible et impact.

La structure modulaire globale S d'un programme, représentée par le
multigraphe G (M, U), offre avec ses relations et attributs associés la
cible la plus vaste d'approche de ce programme que ce soit pour
l'observation, l'évaluation ou la transformation de celui-ci.

Il est, non moins important, de considérer comme cible candidate à l'une
des approches précédentes, à part les composants individuels de S,
toute sous-structure S' de S représentée par un sous-graphe ou
graphe partiel décoré.

Pour cerner toute sous-structure, on procède à une description détaillée
de celle-ci permettant l'examen de cette sous-structure en termes de
relations internes entre ses composants, de relations externes avec les
autres sous-structures ou d'occurrence d'une quelconque propriété
(attribut) de ces relations.

Généralement on spécifie une cible courante d'approche en fonction des
points de vues suivants :,

(1) Point de vue structurel : en termes ensemblistes, un composant
modulaire S' est une sous-structure de S et / ou un ensemble de
composants.

(2) Point de vue relationnel : en termes de types des relations qui
peuvent nous intéresser à un moment donné, s'agissant des relations
entre composants de la sous-structure S' considéré, ou entre S et
d'autres composants de son environnement.

(3) Point de vue qualitatif : en termes d'attributs (propriétés ou entités)
associés à chacun des types de relations choisis.

Certaines transformations envisagées sur le programme se traduisent
par un impact sur la structure modulaire.

Cet impact constitué de l'ensemble des modifications conséquentes
apportées à la structure peut être centré ou non, ou extérieur à une
cible prédéfinie par l'utilisateur; cet impact peut également définir sa
propre cible.

1-2-4-2 Opérations sur le modèle modulaire.

On peut aussi considérer le modèle modulaire comme un ensemble de
composants avec des opérateurs "ensemblistes"(réunion, division,
intersection,...).

La modularité (modules ou types) est l'un des critères essentiels de
qualité contribuant à améliorer la maintenabilité, la portabilité et la
réutilisation d'un programme.

Cette modularité peut être approchée par des opérations sur les
composants de la structure :

- factorisation d'un composant.
- intégration d'un composant à un autre.
- division d'un composant en plusieurs composants.
- ----- etc.

1-2-5 Evaluation du modèle.

Le diagnostic peut être établi pour la structure globale ou pour tout sous-structure et exprimé en termes de cible d'évaluation et d métriques associées.

Les métriques peuvent être classés en **catégories** selon la composition de leurs formules et en **niveaux** selon la partie de la structure modulaire dont elles visent l'évaluation.

1-2-5-1 Niveaux des métriques.

Une métrique donnée peut concerner la structure modulaire tout entière, ou une partie de cette structure. Parties définies par un ensemble de modules remplissant une fonction bien définie ou appartenant à un même niveau hiérarchique (de la composition) ou par un couple de modules ou simplement un module considéré individuellement.

Pour tous les détails concernant ces métriques de différents niveaux :

- structure modulaire globale
- sous-structure modulaire
- couple de modules
- module individuel

on peut consulter (AYOUB,86), (DAVCEV,84), (GILB,77), HALSTEAD,77), (MCCABE,76)

1-2-5-2 Catégories de métriques

Les métriques peuvent être classées selon le nombre des opérandes intervenant dans la métrique et la nature des opérations appliquées aux opérandes.

On distingue alors :

- les métriques élémentaires : exprimées par la cardinalité de l'un ou l'autre des attributs associés à une relation modulaire.
- les métriques simples : obtenues par application d'une fonction numérique à une métrique élémentaire ou par une formulation entre une métrique élémentaire et une constante.
- les métriques composées : composées d'au moins deux métriques élémentaires, elles sont dites de degré n où n est le nombre des opérateurs et / ou fonctions numériques dans la métrique.

1-3 LE MODELE LOGIQUE D'UN COMPOSANT MODULAIRE.

Un composant modulaire dont le texte source est rédigé dans un certain langage de programmation est la réalisation d'un ensemble d'opérations formant un "tout logique" consistant en la définition de fonctions et / ou d'objets liés au problème résolu.

Un composant modulaire peut être considéré de deux points de vue différents :

- point de vue syntaxique : c'est un **texte** rédigé dans un langage de syntaxe précise mais c'est aussi un **ensemble d'instructions** de ce langage liées entre elles par un certain ordre d'exécution (partie contrôle du composant modulaire).
- point de vue sémantique : c'est alors une **fonction** qui à des données et une suite d'identificateurs notant les résultats associe les valeurs des résultats correspondant à ces identificateurs. On s'intéresse alors au contenu des instructions, aux chemins d'exécution possibles et aux flux de données le long de ces chemins.

1-3-1 Le composant modulaire du point de vue syntaxique

Lorsqu'on étudie un composant modulaire ou une classe de composants modulaires un certain nombre de propriétés ne sont pas liées à la signification des fonctions (ni aux instructions, ni à la structure d'information) mais seulement à l'ordre des instructions.

Cette partie **contrôle** du composant modulaire permet d'obtenir des résultats (LIVERCY,78) sur la terminaison, l'isologie (propriété de deux composants qui calculent la même chose) et les transformations de composants modulaires.

Un autre ensemble de propriétés concerne la puissance ou l'efficacité relative d'une famille de **schémas** d'expression du contrôle par rapport à une autre.

Rappelons de manière succincte les principaux schémas et quelques définitions associées, sachant qu'un **schéma de programme** est un objet purement syntaxique; si on attribue une signification aux smboles un schéma de programme devient un programme.

1-3-1-1 Schéma de programme avec branchements.

1-3-1-1-1 Les symboles élémentaires et les instructions

Les symboles utilisés pour décrire les schémas avec branchements sont les suivants :

- les noms des variables, ensemble $X = \{x_1, \text{---}, x_n\}$
- les noms des fonctions, ensemble $F = \{f, g, \text{---}\}$
- les noms des prédicats, ensemble $P = \{p, q, \text{---}\}$
- un symbole : = représentant l'affectation.
- STOP notant l'arrêt
- les entiers naturels pour numéroter les instructions
- l'ensemble des symboles {,} , (,).

Les instructions sont de trois types :

- l'instruction d'arrêt : STOP
- les instructions d'affectation

  $x_i := x_j$

  $x_i := f(y_1,\text{--}, y_m)$ où $f \in F$, $ar(f) = m$ et $\forall i = 1,\text{--},m \quad y_i \in X$

  l'affectation simultanée notée :

  $x_1, x_2, \text{--} ; x_n := e1, e2, \text{--}, en$

  l'affectation conditionnelle notée :

  $(p_1 \text{--->} A_1 \mid p_2 \text{--->} A_2 \mid \text{---} \mid p_n \text{--->} An)$ où

  $\forall i = 1,\text{--},n \quad p_i \in P$ et A_i est une affectation.

Affectation conditionnelle qui correspond à l'affectation A_j associée au premier p_j évalué à vrai, ou qui est indéfini si tous les p_j sont évalués à faux.

- les instructions de test :

$p(y_1,\text{--}, y_n)\ h, k$ où $h, k \in N$, $p \in P$, $ar(p) = n$ et $\forall i = 1,\text{--},n \quad y_i \in X$.

Un schéma de programme avec branchement est alors une suite finie d'instructions numérotées telle que toute affectation est suivie d'une instruction.

1-3-1-1-2 Notion de graphe de contrôle.

De façon évidente, les schémas de programmes avec branchements peuvent être représentés par des organigrammes ou des graphes de contrôle.

Un graphe de contrôle est un graphe connexe orienté avec circuits noté $G = (N, A, n_o)$ tel que :

- N est un ensemble fini de noeuds.
- n_o est le seul noeud initial ($n_o \in N$)

- il n'y a qu'un seul noeud terminal (on peut toujours se ramener à ce cas).
- chaque noeud représente une instruction d'affectation (il s'agit alors du graphe de contrôle étendu) ou un bloc d'instructions (graphe de contrôle restreint) ou une instruction de test (simple si binaire, multiple sinon).
- A est un sous-ensemble de N x N, ensemble des arcs.
- V le noeud n_j du graphe il existe un parcours de n_o à n_j et de n_j au noeud terminal.

Un bloc est une suite d'instructions telles que le contrôle ne peut être donné qu'à la première instruction de la suite, cette dernière étant alors exécutée jusqu'à la dernière instruction.

Un parcours entre deux noeuds n_i et n_k est une suite de noeuds $(n_i,\text{---},n_k)$ tq pour $i \leq j < k$, $(nj, nj+1)$ est dans A.

Un chemin de contrôle est un parcours du noeud initial au noeud terminal sans répétition de cycle.

Un noeud simple correspond à une instruction d'affectation ou à un bloc ; il est à l'origine d'un seul arc.

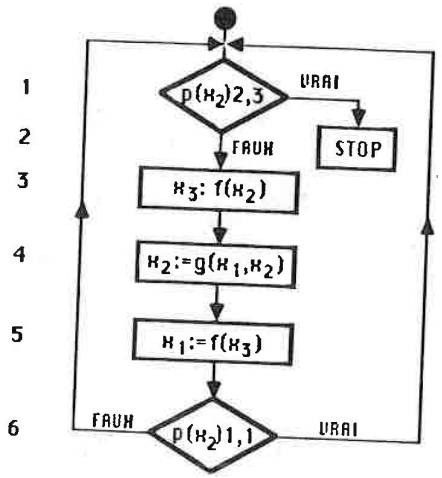
Un noeud de décision correspond à une instruction de test, il donne naissance à au moins deux arcs.

Un point de croisement correspond à l'existence de deux arcs non orientés (x,y) et (z,t) du graphe de contrôle, tels que $x < z < y < t$ pour l'ordre des noeuds dans le graphe.

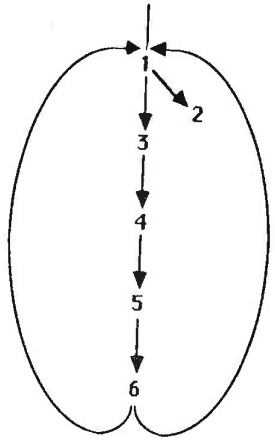
Exemple de programme :

1 $p(x_2)\ 2, 3$
2 STOP
3 $x_3 := f(x_2)$
4 $x_2 := g(x1, x_2)$
5 $x_1 := f(x_3)$
6 $p(x_2)\ 1, 1.$

schéma de programme.



ORGANIGRAMME ASSOCIE AU SCHEMA PRECEDENT.



GRAPHE DE CONTROLE ASSOCIE AU SCHEMA PRECEDENT

1-3-1-1-3 Intérêt des schémas avec branchements.

Les schémas de programmes avec branchements et les graphes associés constituent des représentations non "structurées" (au sens structures de contrôle) et interprétables pour un programme quelconque.

Le graphe de contrôle, dans cette forme primitive, permet l'évaluation de métriques pour la mesure de la qualité du logiciel (complexité du flot de contrôle, nombre de chemins, atteignabilité, nombre cyclomatique, linéarité, longueur des chemins de contrôle, nombre de points de croisement, proportion de noeuds de décision,...etc.) .

Les programmes de qualité médiocre, bien que corrects, objets de notre travail, peuvent toujours être considérés du point de vue de leur schéma avec branchements, ou graphes associés.

Les méthodes de transformation de programmes y compris en Reverse Engineering (Siemens) finissent par mettre "à plat" le programme, le schéma avec branchements (avec les graphes associés) devant un point obligé de représentation de ce programme avant toute modification.

1-3-1-2 Autres schémas et graphes associés.

Etant donné :

- un ensemble d'actions F (chaque élément de F représente une affectation).
- un ensemble de prédicats (ou tests) P
- un ensemble P tq on associe à chaque prédicat p E P sa négation p dans P ; on pose de plus p = p

La définition de différents types de compositions pour combiner les éléments de F et de P permet d'obtenir des assemblages appelés schémas de programmes; schémas que l'on peut construire en adjoignant à F et P des symboles auxiliaires (**si, alors, tantque, faire,** ---etc...)

Toute structure de contrôle consiste en le groupement de plusieurs types de compositions.

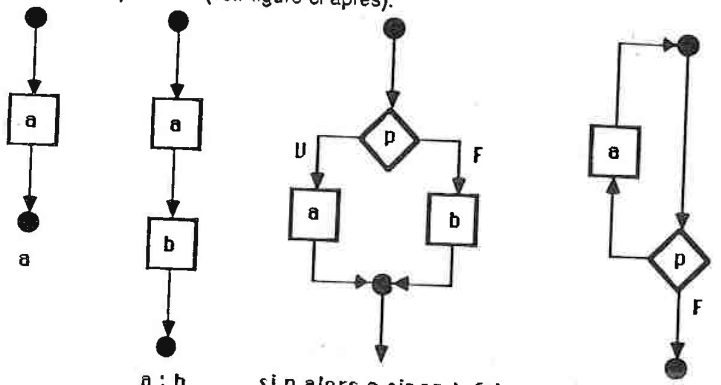
1-3-1-2-1 Le langage des D-schémas

Soit D la structure introduite par (DIJKSTRA,76)

Dans cette structure tout D-schéma admet un point d'entrée et un point de sortie, ce qui correspond à une analyse plus structurée du programme et à une conception plus systématique des preuves de programmes.

Intuitivement, on accepte pour la structure D, la composition séquentielle (;) la composition conditionnelle (si alors sinon fsi) et la composition itérative (tantque faire fait).

Les compositions précédentes peuvent être représentées à l'aide d'organigrammes associés à ces schémas de base ou schémas premiers (voir figure ci-après).



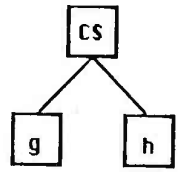
Il a été démontré que si les D-schémas sont aussi puissants que les organigrammes sur le plan fonctionnel (BOHM,66) ils sont sémantiquement moins puissants que ces derniers (KOSARAJU, R.74).

Un programme est dit propre (BASILI,82) TABOURIER) si son graphe associé possède un seul arc d'entrée et un seul arc de sortie et si de plus pour chaque noeud il existe un chemin passant par ce noeud.

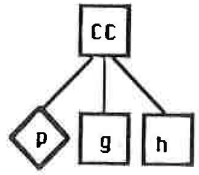
Un programme est dit structuré s'il s'exprime uniquement au moyen des graphes privilégiés précédents auxquels on ajoute éventuellement les structures suivantes **si alors fsi** et **repeter jqa**

Dans le cas où un programme est structuré, on peut lui associer une **arborescence** obtenue graphiquement en donnant par exemple (TABOURIER) aux schémas de base les représentations suivantes.

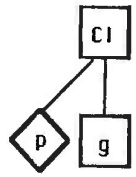
Composition séquentielle ; CS
g ; h



Composition conditionnelle ; CC
si p alors g sinon h fsi



Composition itérative ; CI
tantque p faire g fait



Les D-schémas, très utiles dans les processus de construction de programmes structurés, permettent également d'améliorer la compréhension de programmes existants par la réduction (si elle est possible ou rendre telle) du graphe associé en remplaçant progressivement chaque schéma de base identifié par un noeud (diviser pour comprendre); la fonction associée au programme devenant une composition de fonctions associées aux schémas intermédiaires.

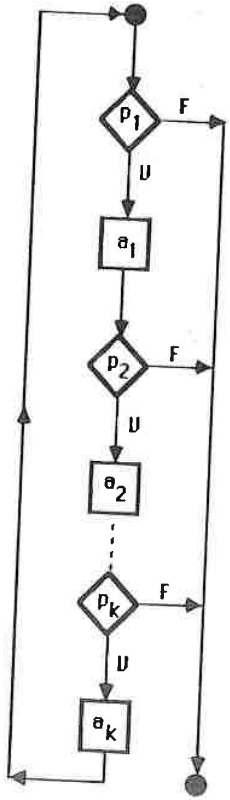
On trouve dans la littérature sur ce sujet un grand nombre de définition de la réductibilité d'un graphe en fonction des transformations considérées (ALLEN F.E. 76), HECHT M.S.74) et quelques méthodes de réduction mettant en oeuvre ces transformations.

La plupart des processus, automatiques ou manuels, de structuration de programmes (ou des graphes associés) consistent à faire apparaître ces schémas de base (ou leurs représentations graphiques).

1-3-1-2-2 Autres schémas

Si l'on accorde la non unicité du point de sortie dans une itération, on aborde la classe des BJn schémas (BOHM-JACOPINI); schémas construits avec la composition séquentielle a; b la composition conditionnelle **si** p **alors** a **sinon** b **fsi**, et la composition dite k-itération faire p_1 ---> a_1 ; -- ; p_k ---> a_k **fait**.

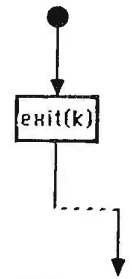
La composition k-itération peut être représentée à l'aide d
l'organigramme suivant :



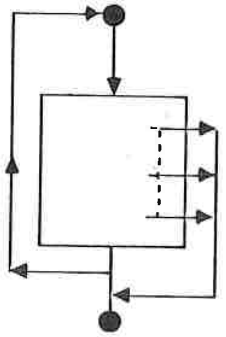
BJ_n - schémas

Enfin si l'on accorde de pouvoir sortir de k itérations englobantes (**exit** (k)) on aborde la classe des RE_n schémas (KOSARAJU R.,74); schémas construits avec la composition séquentielle, la composition conditionnelle, et la composition itérative (**faire a fait**) où a comporte éventuellement des instructions exit.

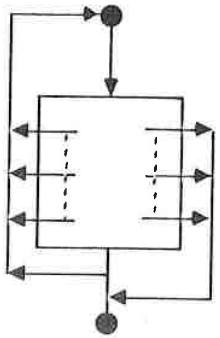
Les organigrammes associés sont les suivants :



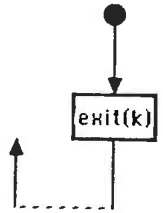
sortie de k itérations



RE_n - schémas



REC_n - schémas



retour au début de la k-ième itération englobante

Enfin l'adjonction de sauts en début d'itération (entrée (k)) conduit à la classe des REC_n-schémas.

Remarque :

Les schémas récursifs rendent compte des procédures récursives qui sont plus des définitions, à la limite axiomatiques, des fonctions que des descriptions de stratégies de calculs.

Or, de façon systématique, on sait passer d'une définition récursive à une procédure itérative décrivant la stratégie de calcul.

Aussi n'envisagerons-nous pas les schémas récursifs, ce qui n'est pas une restriction importante dans le cadre de l'amélioration de programmes "non scientifiques" existants (en COBOL en particulier).

1-3-2 Le composant modulaire du point de vue sémantique.

Les structures de contrôle permettent la construction des algorithmes, elles permettent également des transformations syntaxiques des

programmes associés mais avec une équivalence trop forte si l'on v
"accepter" les différentes versions possibles de texte sou
correspondant à un même énoncé (voir exemple de calcul
polynômes dans (ADAM A.,78)

Affaiblir cette équivalence entre programmes en prenant en compte
structure d'information (propriétés des objets manipulés par
programme) et le contenu des instructions (association de valeurs
des variables) c'est envisager le composant modulaire du point de v
sémantique.

La diversité des buts poursuivis par les utilisateurs (définir u
implémentation, construire des programmes corrects, concevoir d
nouveaux langages,...) explique en partie les différentes approches d
la sémantique.

Les méthodes de formalisation de la sémantique restent encore difficile
à appliquer que l'on considère l'ensemble données-programme tant d
point de vue interprétatif (la fonction sémantique associe au coupl
données-programme un calcul qui peut posséder un résultat) que d
point de vue dénotationnel (la fonction sémantique associe à u
programme sa dénotation ou fonction définie dans l'ensemble de
données et à valeurs soit dans l'ensemble des résultats, soit dans u
ensemble de calculs).

Quant à l'utilisation de la méthode des attributs ou des doubles
grammaires, elle se heurte encore à une certaine lourdeur.

Nous envisageons, dans notre contexte d'amélioration de programmes
existants représentés par leurs textes sources, d'aborder les notions
sémantiques à différents niveaux; le problème, l'instruction, le
composant modulaire.

1-3-2-1 La sémantique du problème.

Elle fait intervenir des informations qui ne sont pas contenues dans le
programme lui-même mais qui sont liées au problème résolu et à son
contexte.

Ces informations détenues par l'utilisateur qui peut les introduire sous
forme d'assertions ou de commentaires dans le texte source sont
presque toujours nécessaires dans les processus de compréhension et
d'amélioration de programmes existants.

Cela nous conduit à rappeler que les personnes confrontées à
l'amélioration de programmes sont non seulement l'informaticien de
service mais aussi l'utilisateur de ces programmes connaissant le

problème informatisé ainsi que les objets concernés et les relations
entre eux.

La participation active de cet utilisateur ne peut alors se concevoir que
dans un environnement techniquement convivial et permettant
l'expression sous des formes variées des fonctionnalités d'un
programme existant et des processus de transformation, à l'aide de
"langages" de plus haut niveau possible.

La modélisation d'un programme (objets et fonctions, ou structures de
données, composants modulaires et relations de contrôle et d'appel)
sous la forme d'un modèle Entité - Association ou Entité Relation
(E.R.C. de PARENT C.) ne peut que faciliter la collaboration
informaticien-utilisateur par exemple dans une perspective de
meilleure superposition des concepts de fonction et de composant
modulaire d'une part, de schémas de décision et de structures d'appel
et de contrôle d'autre part.

1-3-2-2 La sémantique des instructions.

Une instruction peut être considérée comme une suite de symboles et
c'est là son aspect syntaxique. On peut aussi s'intéresser à ce que fait
l'instruction lorsqu'elle est exécutée et ceci quelles que soient les
valeurs des variables qui y apparaissent. Pour cela il faut une
interprétation des instructions en fonction d'**actions** élémentaires et
d'**objets** caractérisant un langage de programmation.

Les objets élémentaires sont les noms de variables auxquels on associe
à certains moments une valeur; on appelle alors variable le couple
(nom de variable, valeur).

Les actions élémentaires sont :

- la lecture d'une valeur
- l'écriture d'une valeur
- le calcul d'une valeur (numérique ou booleenne)
- l'affectation d'une valeur à un nom de variable.

La sémantique des instructions permet certaines transformations comme
la simplification ou l'élimination des impuretés (X : = 1 + 1 = > X : = 2),
l'utilisation de propriétés de symétrie (X : = A + B = > X : = B + A ; Si c
alors b sinon a ; = > Si non C alors a sinon b ;)

La sémantique des instructions conduit à considérer du point de vue des
variables les actions d'utilisation, de définition et d'indéfinition.

Une variable fait l'objet d'une définition quand une valeur est associée à
son nom : le nom de la variable apparait comme membre gauche d'une

affectation ou dans un ordre de lecture ou dans un schéma de bou
implicite.

Une variable fait l'objet d'une utilisation quand sa valeur est prise
compte dans un calcul ; il existe dans l'instruction au moins u
occurrence du nom de la variable ne correspondant pas à u
définition.

Une variable est **indéfinie** hors de la portée de sa déclaration ou
dans le cadre de la portée de sa déclaration elle n'a pas encore f
l'objet d'une définition.

A chaque instruction I dans un composant modulaire, on peut toujou
associer l'ensemble D (I) des variables définies dans I, l'ensemble U
des variables utilisées dans I.

1-3-2-3 La sémantique du composant modulaire.

La sémantique d'un composant modulaire peut être envisagée de deu
façons différentes selon les objectifs visés :

• soit comme l'ensemble des chemins d'actions sur les variables ayar
une occurrence au moins dans le composant et c'est alors prendre e
compte la sémantique des instructions constituant ce composant.

• soit comme processus d'association de valeurs à des noms d
variables (et leur propagation) et c'est alors considérer les flux d
données dans le composant.

Dans un cas comme dans l'autre, l'expression de la sémantique d
composant modulaire est obtenue par propagation le long des chemin
du schéma du composant (ou du graphe associé) de l'information issu
des instructions.

On peut enfin considérer qu'un composant modulaire est un ensembl
de variables avec des relations entre elles comme :

• la dépendance élémentaire **de** (x,y) : il y a échange de valeur entre x
et y (affectation simple).

• la dépendance fonctionnelle **df** (x,y) : x est valorisée à l'aide du
résultat d'une fonction où y est une variable.

• la dépendance structurelle **ds** (x,y) : y est une sous-structure de x

• la dépendance physique **dp** (x,y) : y fait partie d'une redéfinition
mémoire de x , où il existe une synonymie.

1-4 CONCLUSION.

Les transformations de programmes en général, et donc l'amélioration de programmes, nécessitent la modélisation de ceux-ci d'une part à différents niveaux (programme, composant modulaire, instruction) et d'autre part des deux points de vue syntaxique et sémantique.

L'approche des concepts correspondant par les transformateurs doit rester conviviale quel que soit le profil de ces derniers, informaticiens ou gestionnaires. Le système d'amélioration proposé dans ce mémoire ainsi que les modèles associés s'efforcent d'intéresser à la fois les deux profils de transformateurs.

I-2 INTRODUCTION DES FONCTION PRINCIPALES DU PROCESSUS D'AMELIORATIO

2-1 INTRODUCTION

Aprés avoir rappelé au chapître précédent les principaux concepts su lesquels nous nous appuyons tout au long de ce mémoire pour décrire le programmes, nous introduisons dans ce chapître les principales fonctions d processus d'amélioration de programmes.

Nous nous appuyons pour ce faire sur les grandes caractéristiques de nouveaux langages de programmation .

En effet, la progression qui a été réalisée au niveau des langages d programmation est en partie issue du souci de construire de meilleurs programmes

2-2 QUELQUES CARACTERISTIQUES DES LANGAGES DE PROGRAMMATION ACTUELS

Constatons une fois de plus qu'entre des langages tels que FORTRAN, COBOL suivis d'ALGOL, PASCAL et leurs successeurs immédiats d'une part, et les langages tels que ADA, LTR 3, MODULA 2, il existe un gap technologique important, visible essentiellement par le passage d'un langage de progammation à un système de développement progressif de programmes et plus précisément :

- une évolution importante des structures de contrôle.
- une amélioration considérable du concept de type.
- une utilisation systématique du concept de module permettant d'exploiter les mécanismes de compilation séparée et de supporter une conception modulaire des programmes ou utilisant des types abstraits.
- une introduction du concept d'exception comme technique de programmation.
- l'introduction de l'indéterminisme.
- la possibilité d'écrire des modèles de programmes grâce à la généricité.

Cela ce traduit par le développement des qualités accessibles par ces langages.

Avec FORTRAN et COBOL la première qualiré visée était la correction, accompagnée d'un soucis de portabilité.

Avec PASCAL le gain en simplicité et en structuration a permi d'introduire ou d'améliorer la lisibilité , la solidité, la fiabilité et la portabilité.

Les notions de type et de procédure ont apporté avec elles la modularité, la reutilisabilité.

Enfin avec ADA, LTR3, MODULA 2 au niveau de la structuration de grands programmes le gain est essentiel en matière de réutilisabilité, de portabilité et d'évolutivité.

LA STRUCTURATION

Ce fut la préoccupation essentielle en programmation pratique des années 70 que d'améliorer les flux de contrôle négligeant ainsi les actions de base; il n'est que de citer les rudes échanges entre partisans et détracteurs du GOTO (KNUTH) Les langages se sont enrichis de bonnes structures de contrôle permettant au moins l'implantation propre des trois schémas de base (séquence, alternative, itération) dont BöHM et JACOPINI (BöH,66) (LIVERCY) ont démontré qu'ils étaient suffisants (mais à quel prix) pour construire tout programme.

L'unicité du point d'entrée et du point de sortie de ces schémas de base ai que pour tout "morceau" de schéma correspond à une analyse <<structurée>> permet une conception plus systématique des preuves de programmes.

L'éventail des structures de contrôle, fonctionnellement équivalentes à structure générale des organigrammes (D - schémas de DIJKSTRA (DAH,7 (LIVERCY), BJ_n - schémas de BöHM et JACOPINI, RE_n - schémas, REC_n schémas), correspond à la levée progressive de contraintes par rapport a organigrammes et conduit jusqu'à l'équivalence sémantique avec ces dernie (KOSARAJU R. 74, LIVERCY).

LE CONCEPT DE TYPE

Les langages recents fournissent une définition plus précise et plus complèt des types prédéfinis (réels en particulier), des mécanismes de construction de typ (énumération, dérivation, ...) de la notion de sous-type, et de type conditionnel. L contrôle statique de la conformité des actions avec les déclarations permet un programmation plus fiable. Par ailleurs, ces définitions permettent au langag d'échapper au besoin de disposer d'un algorithme complexe d'égalité des types.

De même la définition des modes de passage des paramètres est plu rigoureuse, complétant ainsi le fort typage.

LA MODULARITE

C'est le découpage des programmes en "morceaux" (ex.: module et classe de CIVA,(DERNIAME J.C.)) pour en faciliter la conception, la manipulation, et la modification.

Mais si la modularité se limitait à cela, elle poserait plus de problèmes (variables, globales, segmentation, définition de la notion de module) qu'elle n'en résoudrait (réduction du temps de recompilation, économie de place en mémoire centrale).

Un module doit avoir en fait d'autres qualités

1 être une unité de compilation séparée et de gestion de la bibliothèque

2 être protégé vis à vis de son environnement : **masquage**.Un utilisateur (ou un module) externe ne doit pas pouvoir perturber le fonctionnement du module en accédant aux données internes de celui-ci.

3 offrir une certaine garantie de **protection** de l'environnement vis à vis de lui-même. Un module ne doit pas pouvoir modifier des variables externes autres que celles précisées dans son interface.

4 doit pouvoir être défini fonctionnellement sans faire d'hypothèse sur les choix de son implantation (**abstraction**)

Ces règles conduisent à expliciter une partie visible du module, **son interface** et à développer séparement une partie cachée, **son corps** ce qui permet :

- à un module d'être utilisé avant d'être réalisé complètement
- plusieurs versions du corps d'un module (en bibliothèque) pour le même interface.

Une autre approche du concept de module est celle qui consiste a le considérer comme un élément de programme défini par ses entées, ses sorties et éventuellement ses "données modifiées". Dans ce cas le problème du découpage d' un programme est ramené à celui de la communication des données et de la définition d'une unité de programme pour gérer un ensemble de données utilisées par plusieurs éléments.

Cette notion d' "unité de données" par opposition à celle d' "unité de programme" abordée ci-dessus correspond à la définition d'un module de B. MEYER (MEYER B.,78) : des termes voisins sont ceux de "classe" en SIMULA, "forme" en ALPHARD, "grappe" (cluster) en CLU. La modularité est alors essentiellement liée aux structures de données, un module étant une structure de données, accessible de l'exterieur par le biais d'un ensemble de sous-programmes, et par eux seuls.

LES TYPES ABSTRAITS DE DONNEES

Un type abstrait de données est un module n'exportant que des procédures et des fonctions pour lesquelles l'effet de toute séquence d'appel peut être défini sans ambiguïté (complétude et cohérence). Un tel ensemble d'opérations et leurs définitions caractérise un ensemble d'objets (ceux sur lesquels peuvent s'exécuter ces opérations), c'est donc un type (DERNIAMe J.C.,79).

La programmation à l'aide de types abstraits permet une indépendance complète entre l'utilisation d'un objet (déclarer des variables du type, les passer en paramètres, ... en ne connaissant que l'interface du module) et les choix de sa réalisation (interchangeabilité des corps du module) ; d'où des avantages évidents:

- conception progressive sûre
- correspondance entre les spécifications et la réalisation (possiblités de preuve)
- maintenance plus facile (grande prorpiété de localité).
- réutilisation sûre depuis une bibliothèque de types.
- facilités d'interfaçage et de transport de logiciels.

Citons cependant un inconvénient : tous les liens entre l'invocation d'un variable et ses procédures d'accès et sa localisation sont dynamiques, ce qui e particulièrement inefficace.

Un compromis raisonnable (ADA) consiste à permettre la description du choi d'implantation dans l'interface du module en le rendant accessible à la complitatio des unités qui l'utilisent, mais inaccessible aux unités elles-mêmes : les liens son alors statiques, mais uniques dans un programme. L'intérêt de multireprésentations de type est atténué, dans ce cas, par une certaine lourdeur.

L'ENVIRONNEMENT DE PROGRAMMATION

L'évolution des environnements est nécessaire pour faire face à l'utilisation intensive et efficace de nouveaux formalismes de spécification et de programmation.

Les efforts portent actuellement par exemple dans les domaines de :

- l'intégration des différents outils sous un interface homogène et prédictible : non seulement l'adoption d'une représentation unique des objets manipulés avec vues multiples mais aussi intégration des environnements de spécification, de réalisation et de gestion des programmes (CS 84) (LH,85) (TELPGL)
- la mise au point et la maintenance : suivi de l'exécution des programmes sans interférer avec ceux-ci, à un niveau d'abstraction qui soit celui du langage.
- la description d'un projet et de toutes les informations qui le concernent dans une base de données : dossiers d'analyse, modules, dossiers de programme, d'exploitation,...

Les ateliers de génie logiciel (AGL, AIGL) qui vont de l'utilisation d'un dictionnaire de données géré par un SGBD ((EXCELLA RATOR) aux environnements de programmation générés par une structure d'accueil (PCTE, CAIS) constituent des moyens précis complets et souples pour capter la sémantique des projets, et des processus de développement et d'amélioration correspondants.

2-3 CONTEXTE ET MOTIVATION DE L'AMELIORATION DE PROGRAMMES

L'évolution significative des méthodes de conception de logiciels (CASE - Computer-Aided-Software-Engineering) et des langages de programmation (ADA,...) pose le problème de l'adaptation des programmes existants.

Un nombre impressionnant de programmes sont codés dans des langages de programmation dépassés (COBOL 74, FORTRAN IV, ...) et ont été conçus et réalisés pour la plupart sans application des principes de la programmation structurée. Il n'en reste pas moins qu'ils ont le mérite d'exister, de rendre service aux utilisateurs et souvent de représenter l'ensemble des opérations exécutées dans l'entreprise. Il en résulte un coût de maintenance très élevé, souvent plus de 50 % du budget de l'informatique, grevant ainsi sérieusement la part nécessaire aux nouveaux développements.

L'effort le plus important pour maintenir un logiciel est lié au temps qu'il faut passer pour comprendre un programme dont on n'est pas l'auteur en général.

La réutilisation d'un logiciel dans différents contextes reste toujours une exception dans un système non modulaire; à l'exception des constituants prédéfinis d'une bibliothèque de fonctions et sous-programmes.

Quant à la réécriture d'un programme dans une version améliorée, éventuellement dans un autre langage, cela reste une tâche délicate et coûteuse pour différentes raisons parmi lesquelles les deux suivantes :

- la re-conception d'un logiciel est impossible sans une compréhension complète de la version existante qui potentialise toutes les contraintes et toutes les décisions envisagées durant tout le cycle de vie?
  Très souvent, en effet, il n'existe pas de documentation des logiciels (dossiers, commentaires, ...) et les fonctionnalités sont bien "cachées" dans les textes des programmes.
- la re-conception d'un système correspond à un effort non négligeable dont le coût est difficile à justifier quand il n'est pas question de changer les fonctionnalités ce qui correspond au cas le plus fréquent.

2-4 LA DEMARCHE GENERALE DE L'AMELIORATION

Améliorer un programme nécessite l'exécution d'un certain nombre de tâch dans le cadre d'un processus qui peut être tour à tour séquentiel ou parallèl tâches dont les principales pourraient s'intituler :

- **APPREHENDER**
- **RESTRUCTURER**
- **MODULARISER**
- **ARBORISER**

Et à un niveau différent

- **LINEARISER**
- **DOCUMENTER**
- **CONTROLER**

APPREHENDER un programme existant, c'est retrouver les fonctionnalité attachées aux morceaux de texte source, c'est identifier de manière unique le objets manipulés et leur attacher ainsi une définition sémantique (leur type) Comprendre un logiciel existant, c'est enfin appréhender les liens entre fonctions e objets (l'algorithme) symbolisés par les flux de contrôle et flux de données.

La difficulté de cette tâche résulte essentiellement de la dispersion des flux e des fonctionnalités dans le texte source. Aussi toute compréhension de l'existan est souvent impossible sans un minimum de préparation du texte source.

Ainsi le texte source doit être assaini, par exemple, en éliminant le code mort les instructions inutiles, les variables redondantes, en extrayant les invariants ...

La structure du texte source peut être simplifiée et clarifiée, par exemple en améliorant l'indentation, en éliminant les branchements inutiles, les chaînages de contrôle intempestifs, les fausses itérations et conditionnelles, ...

En bref, aider à la compréhension c'est rapprocher le texte source d'une forme, utilisant diverses expressions possibles graphiques et textuelles, qui tient compte d'une qualité minimum et des "habitudes" de l'utilisateur; ce dernier n'étant que trés exceptionnellement à la fois le créateur et le "mainteneur" du programme concerné.

RESTRUCTURER un programme existant c'est améliorer son schéma et l'expression de son schéma. Cela suppose le choix d'une structure de contrôle et la réexpression du programme à l'aide des constructions permises par cette structure sur le vocabulaire associé.

Cette restructuration peut être envisagée à différents niveaux (intermodulaires, intramodulaire) et de façon globale (ensemble du programme) ou locale (morceau de progamme). Au dela des tentatives de transformations

automatiques (peu utilisées), essentiellement au niveau syntaxique les auteurs de la plupart des articles abordant le sujet conviennent de la nécessité de prendre en compte également l'aspect sémantique.

La restructuration redevient ce qu'elle aurait dû rester un processus de transformation guidé par l'utilisateur, et assisté par ordinateur.

L'utilisateur usant de sa connaissance de la sémantique du problème, des objets concernés, du langage utilisé et déterminant les choix au niveau de la structure donc des schémas et de leur représentation, l'ordinateur mettant en oeuvre des transformations au niveau syntaxique grâce à sa connaissance de la grammaire du langage et appliquant quelques règles liées à la sémantique (grammaires attribuées, procédures attachées aux règles de production, ...)

L'assistance est portée actuellement à un certain niveau d'efficacité grâce à des outils spécialisés, intégrés à l'environnement de programmation (éditeurs syntaxiques, graphiques,...)

Restructurer et Comprendre sont deux tâches dépendantes l'une de l'autre et que l'on ne peut réaliser purement séquentiellement.

MODULARISER un programme existant, c'est mettre en évidence le mieux possible les fonctionnalités. Il s'agit alors de regrouper des "morceaux" de texte source allant ensemble pour constituer le corps du module ; il s'agit également de séparer les variables concernées en variables locales définies et utilisées dans les textes regroupés et variables globales; ces dernières constituent alors un interface par remontée de leur définition dans le module le plus externe, ce qui se fait d'autant mieux que le graphe d'appel est sans circuit.

Modulariser c'est mettre en oeuvre (MYERS) l'un des deux processus classiques suivants :

- **processus descendant** s'appuyant sur la structure globale du programme existant (ce qui peut nécessiter une restructuration), il s'agit alors d'une **approche fonctionnelle** (un module est alors multi-blocs ou multi-procédures).
- **processus ascendant** dans le cas où le programme est considéré comme un ensemble de granules (blocs de base, actions, ...) avec des relations entre eux, il s'agit alors plutôt d'une approche couplage-cohésion ; un module correspond à un sous ensemble à cohésion maximum et à couplage minimum.

ARBORISER un graphe c'est le transformer en arbre.

Au niveau d'un programme existant, cette arborisation peut être envisagée au moins à deux niveaux ; celui du graphe d'appel, donc au niveau intermodulaire, et celui du graphe de contrôle au niveau intramodulaire.

Cette tâche nécessite quelques transformations du graphe concern
(modifications de conditions d'appel ou de branchement, duplication de noeuds,..
et exige pour cela la définition et la valorisation d'attributs caractéristiques de
noeuds et des arcs. Ces attributs (préconditions d'appel, expression conditionnel
d'un noeud alternatif, contexte d'un branchement, ...) sont utiles dans le processu
d'arborisation et dans la réalisation des modifications correspondantes au nivea
du texte source associé.

Si ce processus d'arborisation remet rarement en cause la définition de
modules, il est toutefois susceptible d'apporter quelques modifications dans leu
contenu.

LINEARISER ou plutôt RECODER un programme existant, c'est apporter au
niveau du texte source les modifications correspondant à la prise en compte des
résultats des tâches précédentes.

Selon le niveau de qualité du programme à améliorer, cette tâche peut aller
de quelques modifications du texte source à une réécriture du programme.

Dans tous les cas, il s'agit d'utiliser les instructions et structures de contrôle
du langage cible les plus appropriées à une bonne traduction des schémas
envisagés et de leur composition de façon à conserver le plus possible après
linéarisation du modèle les bonnes propriétés de ce dernier.

Cette tâche de codage doit être très automatisée mais il faut permettre à
l'utilisateur d'intervenir au coup par coup pour des situations délicates résultant de
l'insuffisance des instructions et structures de contrôle du langage cible à exprimer
les schémas composant le programme.

2-5 QUELQUES ASPECTS LOGIQUES ET PRATIQUES DU PROCESSUS D'AMELIORATION

Cette démarche générale de l'amélioration des programmes existants bien que lourde, est une nécessité actuellement dans la plupart des environnements. La mise en oeuvre d'une telle démarche n'est devenue possible (rentable) que depuis le développement d'outils adaptés aux tâches à réaliser.

Le processus d'amélioration d'un programme existant peut être conduit jusqu'à une profondeur correspondant à une spécification à postériori du problème résolu, il s'agit alors véritablement d'une reconception assistée par ordinateur (Reverse Engeneering,CARE(WAGNER J.)).

Cette formulation à postériori des spécifications par abstraction à partir du code source, tâche délicate s'il en est, doit être menée en parallèle avec la restructuration à l'appui d'un texte assaini et dégagé des impuretés.

Cette restructuration assistée par ordinateur doit pouvoir être composée, d'une part, de transformations systèmatiques et d'autre part de transformations spécifiques définies par l'utilisateur et éventuellement mises en oeuvre par lui-même.

Quand aux deux autres fonctions (DOCUMENTER et CONTROLER), leur portée n'est pas limitée au seul processus d'amélioration de programmes.

DOCUMENTER un programme c'est fournir et rendre facilement accessible toute l'information et seulement l'information utile tout au long du cycle de vie de ce programme et en particulier dans notre cas à la mise en oeuvre des autres fonctions du processus d'amélioration.

La documentation doit être complète pour permettre une approche des programmes selon différents points de vue d'un utilisateur ou d'un système ; elle ne doit pas être redondante (répétition ou inutilité de certaines informations).

Dans un environnement classique, il s'agit essentiellement de commentaires et elle est structurée en documentation **interne** ou **locale** (propriétés concernant **l'endroit** commenté) et documentation **externe** ou **globale** (propriétés du **programme** dans son ensemble (DONZEAU-GOUGE).Dans un environnement de type Atelier de Logiciel la documentation est intégrée aux modèles de représentation des programmes et du processus de développement, et ainsi rattachée aux entités documentées.

Quant à la forme, au niveau interne on peut distinguer la documentation formelle organisée pour être traitée mécaniquement et la documentation informelle destinée à aider l'informaticien dans une quelconque approche du programme.

La documentation formelle comprend toutes les représentations possibles du programme est l'ensemble des assertions ; elle a des applications variées comme :

- vérification de certaines contraintes sémantiques

- aide à la mécanisation de transformations complexes du programme.

La documentation informelle constituée de commentaires normalis introduits par le programmeur et de décompilations abrégées à un niveau de dét demandé (décompilateurs du système MENTOR).

Quant à la documentation externe elle peut être symbolisée par le dossier programme dont les composants peuvent être de nature trés divers; texte, graph diagramme, tableau, courbe, image d'état ou d'ecran, etc...

CONTROLER ou VALIDER un programme c'est essentiellement contrôler :

- sa qualité propre
- sa fidélité du point de vue sémantique, aux spécifications.

La qualité propre d'un logiciel s'exprime en termes de facteurs de qualité point de vue de l'utilisateur (ex maintenabilité, intégrité, portabilité,...)

A chaque facteur de qualité, on peut associer des critères de qualité, attributs internes du logiciel (point de vue du concepteur) (Mc CALL 77)

Un moyen de contrôler la qualité d'un programme consiste alors à définir calculer des métriques (mesures quantitatives des attributs internes du logiciel).

Dans le cadre de transformations de programmes dans notre object d'amélioration, il est utile de contrôler pour chaque modification du point de vu sémantique, la fidélité de la version résultat par rapport à la version donnée l'élaboration de mesures de complexité comme nous le verrons, est un moye d'assurer ces contrôles.

Dans une perspective d'amélioration d'un programme, il semble essentiel d procéder à un diagnostic de qualité avant et après transformation (élaboration d mesures de complexité).

Le diagnostic préalable donne une indication du niveau du programm compte tenu des possibilités du langage source et fournit ainsi une "mesure" d l'effort à accomplir (et donc du coût) pour l'améliorer. C'est donc un moyen d localiser les **programmes non récupérables**.

Le diagnostic à postériori permet d'apprécier le gain en matière de qualité essentiellement en fonction des choix de transformation spécifiques déterminés par l'utilisateur.

L'approche du logiciel à transformer doit pouvoir se faire, tant au niveau du flux de contrôle que du flux de données de manière modulaire (morceau pa morceau) sur toute représentation du programme et de façon progressive (approche descendante ou ascendante). Cela implique la visualisation et l balayage possible de tout ou partie d'un graphe avec accès en parallèle au text associé dans le programme concerné.

La disposition d'un environnement de type **atelier de logiciel** doi permettre la définition et la mémorisation progressive de la structure d'information

du site concerné, la prise en compte de l'ensemble des programmes de l'utilisateur conformément aux modèles de représentation des programmes et du processus d'amélioration définis par lui.

Cet environnement présente deux avantages essentiels :

- permettre une approche plus homogène et conviviale des fonctions et objets concernés par les différents types d'utilisateurs.
- exiger conformément aux modèles, la fourniture pendant le processus d'amélioration d'une certaine quantité et qualité d'information avec en contrepartie pour l'utilisateur la possibilité d'utiliser le calcul de métriques pour vérifier son niveau de compréhension du logiciel étudié et mesurer l'état d'avancement de son processus de redéfinition.

2-6 CONCLUSION

Le processus comprend donc un certain nombre d'étapes entre lesquelles un ordre préétabli ou imposé n'est pas nécessaire. Chaque étape correspond à un objectif accessible approximativement par l'application d'une fonction que nous venons de présenter.

La prise en compte des étapes est faite dans le cadre d'un cheminement propre à chaque utilisateur mais dépendant de différents facteurs :

- le niveau des langages de programmation utilisés (source et cible).
- le type des structures concernées (schémas de base).
- la qualité du programme existant et celle attendue du nouveau programme.
- la profondeur du processus d'amélioration (du traitement de texte à la reconstruction des spécifications).
- le type d'équivalence exigé entre programme source et programme cible.

Les moyens, en particulier l'environnement de programmation ou le système d'aide à l'amélioration, doivent permettre la mise en oeuvre des opérations et fonctions dont sont constituées les étapes (élaboration et modification d'ensembles de données, de graphes, de textes, ...) en vue de réaliser les tâches de la démarche d'amélioration.

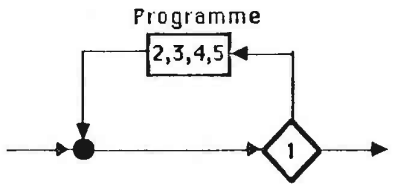
3-1 FINALITES DE LA FONCTION : APPREHENDER

L'objectif principal est la compréhension du programme existant et cel nécessite la recherche et la mise en évidence des principaux éléments constitutif que sont :

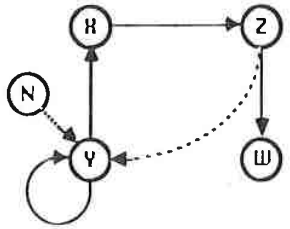
- **les actions** dont la sémantique est directement liée aux instructions d base du langage.
- **le flot de contrôle** dont la définition est liée à la composition des instructions d'appel, de branchement et aux structures de contrôle.
- **les données** dont la connaissance est liée partiellement aux déclarations.
- **le flot de données** expression de la propagation de relations (essentiellement de dépendance) entre les données le long du flot d'appel, du flot de contrôle ou de la structure du programme.

Ainsi à l'exemple de programme ci-dessous on peut associer son graphe de contrôle et son graphe de flux de données.

```
1    WHILE   Z < N DO
2            Z<----- H
3            H<----- Y
4            Y<----- Y+2
5            W<----- Z
             END WHILE
```

FLOT DE CONTROLE



FLOT DE DONNEES

Le but de ce chapitre est de rappeler l'intérêt qu'il y a à considérer à la fois les actions (flot de contrôle) et de données (flot de données) dans toutes approche d'un programme. Nous présentons deux manières différentes (ascendant et descendant) d'appréhender un logiciel existant, conduisant à la construction de structures de représentation qui vont permettre d'appliquer les autres fonctions du processus d'amélioration.

3-2 LES CONCEPTS DE FLOT DE DONNEES ET DE FLOT DE CONTRÔLE

Le graphe du flux de données met en évidence, non seulement comme le code, les flux directs dûs aux instructions d'affectation mais aussi le flux indirect entre les variables de l'expression conditionnelle (Z<N) et la variable Y.

Le double point de vue flot de contrôle flot de données est une nécessité au niveau de la compréhension des programmes de gestion en particulier. Cette dualité ne date pas d'aujourd'hui au niveau des méthodes d'analyse de projet (PSL:PSA, (TEICHROEW, 77), SADT, (ROSS)).

Le flot de contrôle a toujours été utilisé tout au long du cycle de vie du logiciel ; par exemple optimisation en compilation (BARRETT, 79), au niveau des tests, détermination des chemins utiles à tester, mesures de complexité directement liées au flot de contrôle (Mc Cabe, 76) ou s'appuyant sur lui.

Le flot de données a fait son apparation dans le domaine de la protection au cours de la transmission de l'information (COHEN 77) ; Une utilisation intéressante

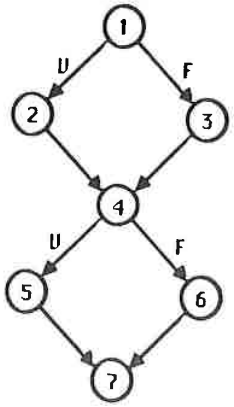
de ce flot de données est la décomposition de l'ensemble des variables d'u programme en sous-ensembles indépendants. Cette indépendance entre variable est utile de différentes manières : par exemple pour diminuer le nombre de tests effectuer sur un programme (HOWDEN, 80, MILLER, 80), en ceci que si de variables sont indépendantes l'approche combinatoire n'est pas utile, alors que s existe des interactions, elles doivent être testées;par exemple aussi pour contribue à la modularisation par la mise en évidence de variables locales.

Ainsi dans l'exemple suivant avec deux noeuds de décision il y a potentiellement quatre chemins distincts qui sont :

(1, 2, 4, 5, 7) , (1, 2, 4, 6, 7) , (1, 3, 4, 5, 7) , (1, 3, 4, 6, 7)

```
1  IF  A > B
       THEN
2      X = 7
       ELSE
3      X = B + 3
       END-IF
4  IF  C = D
       THEN
5      Y = C + 3
       ELSE
6      Y = 4
       END-IF
7
```

PROGRAMME



FLOT DE CONTROLE

L'approche combinatoire devrait tester ces quatres chemins.

Les graphes de flux de données ci-dessous montrent que l'ensemble { A,B,X, } est localement indépendant de l'ensemble {C, D, Y} et donc que les deux noeuds conditionnels sont indépendants l'un de l'autre; il suffit alors de tester les deux chemins **(1, 2, 4, 5, 7)** et **(1, 3, 4, 6, 7)**.

FLOT DE DONNEES

Un autre exemple d'utilisation du flux de données est celui de l'exécution symbolique (KING J., 75, CLARKE L.,76).

Le principe en est la construction d'une expression symbolique exacte qui définit les valeurs des variables en fonction des valeurs en entrée. Le défaut principal de l'expression symbolique est qu'elle n'est calculée à chaque fois que pour un chemin (ou un petit nombre de chemins) du programme.

En cours d'exécution symbolique à la rencontre d'une boucle, l'analyse du flot de données du corps de la boucle indique quel est le nombre maximum d'itérations utiles garantissant la prise en compte de tous les effets possibles du flot de données dans l'expression symbolique ; il s'agit du plus grand des deux nombres que sont :

- la longueur du plus long chemin
- la longueur du plus grand cycle.

L'étude de l'invariance, de l'indépendance entre variables de "contrôle" et variables de "calcul" fournit des résultats interessants pour faciliter la transformation des programmes.

Ces quelques applications concernant la prise en compte des objets et de leurs valeurs approchent le domaine de la sémantique des programmes.

Il y a longtemps déjà que la "théorie des programmes" (C.LIVERCY 78) a réuni syntaxe et sémantique des langages de programmation mais si l'unicité est réalisée au niveau de l'approche syntaxique d'un langage (et d'un programme), il n'en va pas de même au niveau sémantique où l'approche est différente (sémantique interprétative, calculatoire, fonctionnelle, axiomatique), selon le but poursuivi (implantation, programmation, transformation, conception des langages).

3-3 UNE METHODE D'APPREHENSION

L'approche d'un logiciel existant peut être faite soit de manière descendan avec appui sur une structuration en progressant du général au particulier soit manière ascendante en partant du niveau de l'instruction mais dans les deux c avec prise en compte de la dualité traitement - donnée.

3-3-1 La démarche descendante

Décomposer pour comprendre impose généralement la restructuratio partielle au moins, du logiciel concerné.

Cette démarche consiste à prendre en compte la structure logique existan la plus externe possible du texte source. Cette structure externe étant généralemer construite à l'aide de relations d'appel et / ou de contrôle entre élément modulaires du programme source.

Rappelons qu'un **élément modulaire** étant un morceau de texte sourc "appelable" et / ou "contrôlable" donc identifié par un nom, possédant au moins u point d'entrée et au moins un point de sortie et dont l'éxécution se termine pa l'accés à un point de sortie ou à la fin du programme.

En FORTRAN un élément modulaire peut être une séquence d'instruction comprises entre deux étiquettes ou un sous-programme, en PASCAL un bloc o une procédure, en COBOL un paragraphe ou une procédure ou une section ou u sous-programme, en ADA un sous-programme ou un package. L'utilisateur doi alors dans un contexte précis définir ce qui peut constituer un élément et donc les instructions de branchement et d'appel à prendre en compte.Une première difficulté résulte de l'utilisation abusive maladroite ou incorrecte des instructions d branchement alors considérées comme expressions de relations entre éléments modulaires, avec par exemple l'existence de points de croisement, de branchemen vers l'exterieur ou vers l'interieur d'un élément modulaire.

Aussi l'utilisateur doit-il pouvoir disposer des moyens de mettre en oeuvre tout ou partie des opérations suivantes :

- décomposition du programme en éléments modulaires
- analyse de chaque élément modulaire externe
- transformation de chaque élément externe reconnu comme tel en élément propre
- restructuration de chaque élément externe reconnu
- analyse de l'ensemble des éléments modulaires internes du programme.

3-3-1-1 Décomposition du progamme en éléments modulaires.

Sachant que les modules du programme cible seront constitués à partir des éléments modulaires propres du programme source il s'agit dans un premier temps de localiser parmi les éléments modulaires existants ceux qui présentent une singularité certaine en matière de fonctionnalité (cohésion fonctionnelle forte) et / ou en terme de relation avec le reste du programme (couplage léger).

Pour ce faire le programme est décomposé en deux classes d'éléments modulaires :

- L'ensemble des **éléments modulaires externes** ou appelés c'est à dire contrôlés par des instructions d'appel (passage du contrôle avec retour): il s'agit d'une sous-structure du modèle modulaire correspondant aux relations d'appel dont une représentation est le **graphe d'appel** du programme.
- l'ensemble des **éléments modulaires internes** c'est à dire controlés par des instructions de branchement (implicites ou explicites). L'un de ces éléments internes est dit principal il contient alors la première instruction du programme et aussi dans le cas d'un programme bien structuré, la dernière instruction : il sagit de la sous structure du modèle modulaire correspondant aux relations de branchement dont une représentation est le **graphe de branchement** du programme.

3-3-1-2 Analyse de chaque élément modulaire externe

L'analyse de chaque élément externe doit permettre de décider pour chacun d'entre eux s'il garde son statut d'élément externe ou s'il doit être réintégré au reste du programme par substitution pour être envisagé comme partie de l'ensemble des éléments internes.

Un élément externe est caracterisé par

- son nom
- le texte de son corps
- son interface
- ses relations avec le reste du programme (ex. partage des données)
- ses éléments modulaires externes avec un graphe d'appel
- ses éléments modulaires d'appel avec un graphe de branchement (ou graphe de contrôle du seul élément interne éventuel)

Si le texte de l'élément externe est court, sa lecture peut permettre d'en extraire la signification, dans le cas contraire, sa compréhension peut exiger de l'aborder au même titre qu'un programme comme ensemble d'éléments internes.

3-3-1-3 Transformation de chaque élément externe en **élément propre.**

Un élément propre (BASILI V.R., 82) est un élément modulaire ne possédant qu'un seul point d'entrée et tout point de sortie correspond à un retour du contrôle au point d'appel, de plus il n'existe pas d'instruction d'arrêt du programme dans un élément propre.

Généralement, les éléments externes d'un programme satisfont à cette définition (sous-programmes, procédures externes, ...). Cette transformation d'un élément externe en élément propre peut avoir des répercussions sur le texte du programme concerné.

3-3-1-4 Restructuration de chaque élément externe propre.

C'est le même processus que celui mis en oeuvre pour restructurer un programme en tant qu'ensemble d'éléments internes dans lequel on ne tient pas compte des appels d'éléments externes (voir RESTRUCTURATION, pl ch4).

3-3-1-5 Analyse de l'ensemble des éléments internes du programme.

Les deux buts principaux de cette analyse sont :

- mettre en évidence éventuellement de nouveaux éléments modulaires externes ; élément interne apparaissant plusieurs fois, regroupement d'éléments internes participant à la définition d'une même fonction ou utilisant un même sous-ensemble des variables du programme,....
- définir finalement la **sous-structure modulaire source** essentiellement par sa représentation sous forme de graphe de branchement (ou de contrôle dans le cas d'un nombre restreint d'éléments internes) sans tenir compte des éléments externes.

Ces deux buts principaux participent de la même démarche initiale: rapprocher le plus possible la structure physique du programme de sa structure logique en particulier diminuer le plus possible la distance textuelle entre deux éléments modulaires s'enchaînant par une instruction de branchement explicite (GO TO).

Un réordonnancement des éléments internes réduisant les instructions de branchement inutile (branchement direct au composé) résulte par exemple de l'élaboration du graphe de branchement à partir d'un arbre de recherche avec priorité à la profondeur (Depth First Search Tree).

La recherche de nouveaux éléments externes peut nécessiter la prise en compte du flot de données au niveau de l'ensemble des éléments modulaires internes.

Un élément interne est alors caractérisé par :

- son nom
- le texte de son corps
- le contexte d'appel (précondition d'exécution)
- la préassertion (invariant)
- l'ensemble des variables référencées
- l'ensemble des variables définies lors de la première ou de la dernière occurence de leur nom.
- l'ensemble des variables utilisées en première ou dernière occurence.
- les relations avec le reste du programme (couplage).
- le graphe de contrôle (cohésion).

La constitution d'éléments externes peut résulter, par exemple :

- d'un regroupement d'éléments internes ayant même graphe de contrôle, même texte source aux variables près (notion d'identité dans ARSAC BP).
- de l'isolement d'un sous-ensemble d'éléments internes utilisant ou échangeant des données correspondant à un sous-ensemble des variables du programme (**cohésion communicationnelle**).
- de l'isolement d'un sous-ensemble d'éléments internes concourant à la réalisation d'un ensemble de fonctions pour résoudre un problème précis (**cohésion procédurale**) ou à la réalisation d'une seule fonction sémantique (éventuellement sans variable d'entrée ou de sortie comme la génération d'un nombre aléatoire) (**cohésion fonctionnelle ou informationnelle**)
- de l'isolement d'un sous-ensemble d'éléments internes faiblement relié au reste du programme (couplage minimum) : les liaisons étant globalement de deux catégories
  - liaison ou couplage par l'environnement, couplage par zone commune ou par les externes ou par les paramètres et les données.
  - liaison ou couplage par le contrôle ou par le contenu.

La classification actuelle des types de cohésion en trois classes :

I : fonctionnelle, communicationnelle, séquentielle
II : procédurale
III : logique, coïncidentale

permet à l'aide de métrique appropriées de classer un sous-ensemble d'éléments modulaires dans l'une de ces trois classes (Thomas, J. Emerson, DMMC). A l'appui de ces métriques existent un certain nombre de mesures de

complexité du flot de contrôle (Mc Cabe's cyclomatic complexity, Halstead's software effort, Woodward et al's knots) dont la composition est souvent nécéssaire pour obtenir des résultats significatifs.

Dans tous les cas il est souhaitable quelque soit le moyen utilisé pour mettre en évidence un élément externe de procéder à l'assainissement du texte source par des transformations syntaxiques et / ou sémantiques locales.

3-3-2 La démarche ascendante :

C'est la seule approche possible pour diminuer la granularité d'un programme monolithique ou de code spaghetti, c'est à dire, non structuré (pas de structure de contrôle) et encombré d'instructions de branchements explicites, GO TO.

L'approche est alors nécessairement du type couplage-cohésion.

Le programme est alors considéré comme un ensemble de granules. Chaque **granule** est un ensemble éventuellement identifié d'instructions consécutives. La définition de ces granules et donc leur taille moyenne dépend des relations prises en compte entre granules; par exemple :

- une relation de précédence de nature sémantique entre granules peut être la suivante ;

  Soit g_1 et g_2 deux granules ; g_1 **précède** g_2 si l'exécution de g_1 appelle g_2 ou encore si g_2 a une occurrence dans g_1.

  Une telle relation définit des granules qui sont les actions de J.ARSAC;

  Le graphe de cette relation a un point d'entrée unique, le granule dont le nom est celui du programme et un point de sortie unique, le granule contenant l'instruction d'arrêt (on peut toujours se ramener à ce cas) ; ce graphe peut comporter des circuits correspondant aux boucles du programme.

- une relation de partage de données de nature syntaxique entre les granules peut être la suivante;

  On a **partage** (g_1, g_2) s'il existe au moins une variable ayant une occurrence à la fois dans g_1 et g_2.

  Cette relation de partage est symétrique, le graphe associé est non orienté et peut être décoré au niveau de chaque arc par la liste des variables partagées et au niveau de chaque noeud par le mode d'utilisation par variable. On peut envisager d'autres relations liées au flot de données du programme.

- Une autre relation syntaxique entre granules est celle du séquencement physique

  (structure du programme) de ces granules g_1 **est suivi de** g_2 si le texte de g_2 est placé immédiatement derrière g_1.

3-3-2-1 Premier type de démarche ascendante.

Ce premier type de démarche ascendante est celui défini par ARSAC dans (ARSAC,83).

Il consiste à associer au programme un système d'actions régulières sur lequel peuvent être effectuées des transformations (substitution, passage de la recursivité terminale à l'itération, identification,...)

Toute instruction structurée est décomposée en instructions élémentaires et branchements, puis en actions.

Une action est une suite d'instructions nommée. Le nom de la suite est lui-même une instruction.

L'action est dite régulière si :

- toute exécution de l'action se termine par l'appel d'une action (introduction d'une action terminant le programme).
- tout appel d'action dans le corps de la définition est en position terminale, c'est à dire qu'il n'est suivi d'aucune instruction dans le corps de l'action.

Le graphe associé à un tel système d'actions régulières définissant un programme n'est autre que le graphe de contrôle de ce programme.

3-3-2-2 Un second type de démarche ascendante.

Ce second type de démarche ascendante consiste à prendre en compte un ensemble de relations entre granules.

Le programme peut alors être représenté par un multi-graphe.

Trouver des composants modulaires, c'est trouver les sous ensembles de granules minimisant les arcs entre eux.

Dans le cas d'un graphe simple, il s'agirait de la recherche des points d'articulation ou d'isthmes et du partitionnement des sommets de ce graphe en composantes k-connexes - Voir les algorithmes de Tarjan (TAR, 72) pour la détermination des points d'articulation et des composants 2-connexes d'un graphe.

Dans le cas d'un multigraphe G connexe, on est amené à trouver des zones d'articulation, ensembles de points d'articulation ou d'isthmes, de taille minimum ou comprise dans un intervalle prédéfini.

Un intérêt de cette seconde démarche serait de pouvoir prendre en compte n'importe quel sous-ensemble de relations entre les granules et de comparer les partitionnements obtenus.

Cette approche de type couplage-cohésion devrait permettre à l'utilisateur le choix des relations entre granules et même un ordre de prise en compte de celles-ci de manière à refléter au mieux le type de couplage et le niveau de cohésion attendus.

3-4 CONCLUSION :

Au niveau de l'appréhension de l'existant,

- avec la première approche descendante, il y a prise en compte d'un certaine structuration du texte existant en terme d'éléments modulaires de cohésio au minimum coïncidentale;on procède alors par exemple à une reconstitutio fonctionnelle de l'actuelle version du programme.

- dans la deuxième approche, l'effet de décomposition du programme e granules parait, à priori, destructurant mais la représentation du programme sourc sous forme de multigraphe permet une étude de couplage (cohérence sur u ensemble de relations liées au contrôle, aux données et aux caractéristiques de granules (pré-assertion, variables utilisées, définies,....)

Toutes approches confondues, appréhender un logiciel c'est construire de structures de représentation qui vont permettre

- de mieux comprendre le logiciel candidat à l'amélioration

- d'appliquer éventuellement, selon le choix du transformateur, les autre fonctions du processus d'amélioration; l'utilisateur peut être assisté dans s décision par la mise en œuvre, en particulier, de la fonction CONTROLER (o DIAGNOSTIQUER).

I-4 RESTRUCTURATION DU TEXTE SOURCE : RESTRUCTURER

4-1 OBJECTIF :

La restructuration d'un programme est intéressante à différents points de vue. Elle participe à l'amélioration de sa lisibilité et donc de sa compréhension, elle permet l'amélioration de sa structure en ayant comme but l'acquisition de la propriété de reductibilité des graphes de flux associés. Elle permet et simplifie l'élaboration de mesures de complexité, liées au niveau de structuration, à la cohésion et au couplage des constituants dans l'optique de la modularisation.

L'approche précédente du texte source du logiciel a permis d'extraire de celui-ci des sous ensembles pour constituer des éléments modulaires externes (considérés comme "appelés") pour le programme.

Ceci ponctue la définition de la sous-structure du modèle modulaire correspondant à la relation d'appel..

Une dernière préoccupation concernant cette relation sera l'arborisation du graphe associé envisageable au moment où l'utilisateur considère comme définitive la sous-structure d'appel.

Les techniques d'arborisation sont indépendantes du type de la relation et seront envisagées à propos de la relation de branchement et/ou de contrôle dans ce chapitre concernant la fonction Arborisation).

Pour l'instant, le programme ou de la même façon tout élément modulaire externe peut alors être considéré du point de vue des sous-structures associées respectivement aux relations de branchement (GO TO graphes, Lyle Ramshasw) et de contrôle.

Pour ce qui est du flot de données, la nécessité d'une restructuration minimum correspond à un souci de simplicité et de qualité au niveau des mécanismes de propagation des données et de définition des contextes et assertions.

Si le programme est constitué d'un ensemble d'éléments modulaires internes, la restructuration fait appel à l'analyse et à la transformation du graphe de branchement associé.

Dans le cas contraire (il n'existe pas d'élément interne ou bien on décide de ne pas les prendre en considération comme tels) et de la même façon pour chaque élément modulaire c'est le graphe de contrôle associé qui peut avoir à subir directement des transformations pour faire acquisition de bonne propriétés de structuration.

Toute restructuration d'un programme existant ou d'un élément modulaire nécessite de tenir compte des niveaux de transformation possibles et de déterminer

- le type d'équivalence exigé entre le programme source et le programme cible.
- les instructions et structures de contrôle du langage cible et leur mode d'utilisation.

4-2 LES TYPES D'EQUIVALENCES POSSIBLES ENTRE PROGRAMME SOURCE ET PROGRAMME CIBLE

Le type d'équivalence entre programmes dépend non seulement des possibilités offertes par les langages utilisés pour l'implémentation, mais surtout de la nature de la conversion effectuée, c'est à dire des relations entre structures source et cible.

Etant données deux structures S_1 et S_2, les relations les plus couramment envisagées sont les suivantes (C.LIVERCY Théorie des programmes) :

- **conversion fonctionnelle** : pour toute interprétation et tout schéma s_1 de S_1, il existe un schéma s_2 de S_2 tel que $I[s_1] = I[s_2]$; intuitivement s_1 et s_2 définissent la même fonction.
- **conversion sémantique** : pour toute interprétation et tout schéma s_1 de S_1, il existe un schéma s_2 de S_2 construit sans introduire de nouvelles actions et tel que $I[s_1] = I[s_2]$
- **conversion par calcul** : pour toute interprétation et tout schéma s_1 de S_1 il existe un schéma s_2 de S_2 construit sans introduire de nouvelles actions ou de nouveaux tests et effectuant pour toute donnée le même calcul que s_1 .
- **conversion stricte** : alors intuitivement cette conversion impose qu'aucune recopie ne soit faite lors de la transformation.

Les programmes à améliorer ont été conçus avec comme moyen et même comme "méthode" l'organigramme (ou structure des schémas avec branchement). Quant aux programmes récents, résultats de l'application de bonnes méthodes de conception, ils font souvent usage de schémas de base de type RE (C) ∞ ou DRE(C) ∞ dont RAO-KOSARAJU (KOSARAJU,74) a prouvé l'équivalence sémantique à la structure des organigrammes.

Le problème est que les langages de programmation actuels n'offrent pas les moyens de traduction satisfaisants. Il n'est que de rappeler les délicates traductions dans certains langages des schémas itératifs à plusieurs points de sortie, ou des sorties en fin ou en début de la $i^{ième}$ itération englobante.

Tout ceci, ainsi qu'un certain niveau d'exigence en matière de structuration et de lisibilité des textes explique la nécessité d'utiliser le GO TO d'une façon et avec une fréquence liées au niveau du langage de programmation utilisé et à la nature du problème à résoudre (D.E.KNUTH).

L'élimination des GO TO dans le texte des programmes a fait couler beaucoup d'encre et peut se résumer au prix qu'il faut payer pour y parvenir.

On peut rappeler à ce propos les conditions nécessaires et suffisantes pour remplacer les constructions de contrôle source exprimées à l'aide de GO TO par des constructions cible sans GO TO. Ces conditions dépendent en particulier du type d'équivalence attendu entre les programmes (RAMSHAW).

Au niveau de **l'équivalence fonctionnelle** entre deux programmes où l'on permet l'introduction de variables, on peut toujours remplacer les structures de contrôle source, y compris le GO TO, par des boucles "tant que" incluant des instructions conditionnelles (Böhm et Jacopini).

Dès que l'on considère **l'équivalence sémantique** (path équivalence) ou **l'équivalence par calcul**, il a été démontré (Knuth et Floyd) que les seuls trois schémas de contrôle, la séquence (begin-end), l'alternative (if-then-else) et l'itération (while-do), ne suffisent plus à éliminer les GO TO.

Rao Kosaraju démontre même qu'une itération sans fin et une instruction de sortie (exit) de l'itération englobante n'y suffisent pas non plus et qu'il faut en arriver à une instruction de sortie multiniveaux pour pouvoir mettre fin à n'importe quelle itération englobante (Ada offre par exemple cette sortie multiniveaux).

Si l'on impose qu'aucune recopie ne soit faite, il s'agit alors d'une **équivalence stricte** (équivalence des graphes de flux), l'instruction de sortie multiniveaux ne peut permettre d'éliminer les GO TO que si le graphe de flux est réductible, c'est à dire qu'il n'existe pas de cycle à plusieurs points d'entrée (HECHT);

La reductibilité des graphes de flux correspond à la puissance de l'instruction de sortie multiple (BAKER).

L'équivalence structurelle impose non seulement la préservation du graphe de flux mais aussi de la structure du programme au niveau de la linéarisation des actions et tests (PETERSON). Cette équivalence exige, indépendamment de l'absence de GO TO entrant une structure de contrôle (GO TO inward ; en PASCAL et ADA il n'existe que des GO TO sortant (outward) alors qu'en C on dispose des deux types de GO TO) que le graphe de flux augmenté des arcs statiques, (augmented flow graph de (RAMSHAW L.,88) soit réductible.

Le processus d'élimination des GO TO dans ce cas impose en plus l'absence de points de croisement (type Head-to-Head) au niveau des graphes de branchement (GO TO graphs) de chaque "bloc" de programme.

Les contraintes imposées sont à la mesure du type d'équivalence attendu entre programme source et programme cible et donc des instructions disponibles dans le langage cible.

Bien que le langage de programmation utilisé puisse induire un type de structure déterminé facilitant ainsi la linéarisation des schémas correspondants, il est préférable que cette détermination soit le résultat de la nature du problème à traiter et du choix de l'utilisateur.

Plus la structure utilisée est riche plus elle est proche de celle des organigrammes, ce qui ne peut qu'élargir l'éventail des possibilités en matière de conception, en particulier au niveau des graphes ; nous revenons ainsi à la "méthode" des organigrammes mais maintenant avec une démarche de conception et l'utilisation de schémas de base précis qu'il suffit de composer.

En contrepartie, la construction de schémas de programme indépendamment du niveau du langage de programmation utilisé déplace les contraintes éventuelles au niveau de la linéarisation.

La réduction de ces contraintes peut-être obtenue de différentes manières :

- extension du langage de programmation utilisé avec un macro processeur de traduction du macro langage.
- systèmes de transformation syntaxique de type "source to source" utilisant une collection de primitives syntaxiques (pattern replacement rules) (MAHER, SLEEMAN).
- systèmes de transformation dirigés par la syntaxe utilisant des grammaires attribuées (KNUTH,68).
- utilisation d'un langage de programmation mieux adapté à la traduction des schémas introduits.

4-3 LES SCHEMAS DE BASE "PROPRES"

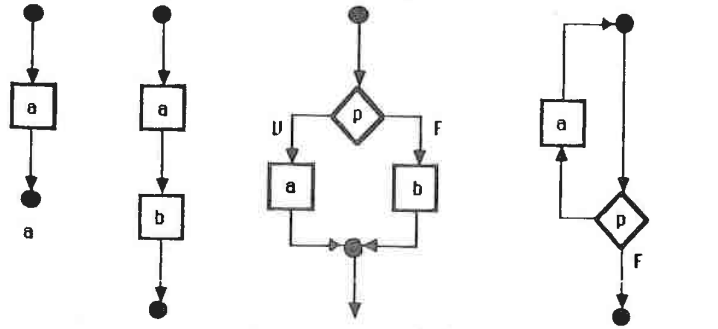
L'objectif essentiel en matière de restructuration est d'aboutir à l'expression du schéma de programme en terme de schémas de base propres.

Un **schéma de base propre** étant caractérisé par :

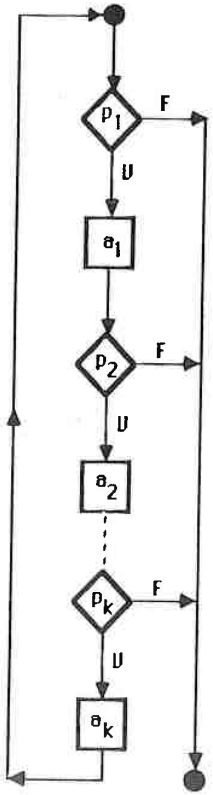
- l'unicité du **point** début ou **d'entrée**, point formel de passage obligatoire du contrôle pour accéder à la première instruction du schéma entré.
- l'unicité du **point** fin ou de **sortie**, point formel de passage obligatoire du contrôle après exécution de l'une quelconque des dernières instructions possibles du schéma concerné.
- quelle que soit le noeud (l'instruction) considéré du schéma il existe un chemin au moins ayant pour origine le point d'entrée et pour extrémité le point de sortie et passant par ce noeud.
- pour toute imbrication de n schémas itératifs il existe un niveau d'imbrication $k \leq n$ tel que tout point de sortie de l'une des k-1 itérations englobées réalise un saut au point d'entrée ou au point de sortie d'une itération englobante de niveau inférieur ou égale à k.

Ainsi quelle que soit la structure choisie par l'utilisateur, l'expression de l'algorithme est faite en terme de schémas propres facilitant ainsi les mécanismes d'analyse, de décomposition, de transformation et de décoration des graphes associés.

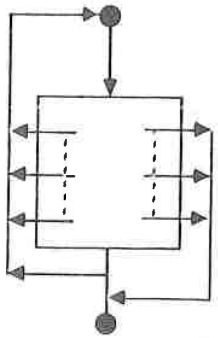
Rappel des principaux schémas de base :



D- schémas

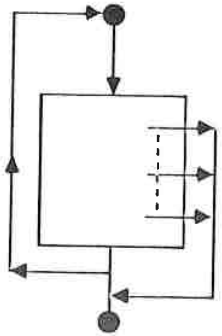


BJ_n - schémas

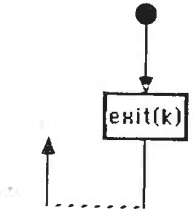


REC_n - schémas

sortie de k itérations



RE_n - schémas



retour au début de la k-ième itération englobante

4-4 LES ETAPES DE LA RESTRUCTURATION :

La restructuration peut nécessiter le passage par deux étapes principales dont la prise en compte dépend du volume du programme et de son niveau initial de structuration.

- -la décomposition du graphe de branchement en sous-graphes de branchement partitionnant ainsi l'ensemble des éléments modulaires internes en sous ensembles ; problèmes d'articulation ou de couplage.
- - la restructuration de chaque sous-graphe pour le rendre décomposable en schémas de base d'une structure choisie (D-schémas ou BJ ou RE schémas) : sous-graphe propre réductible.

4-4-1 Décomposition du graphe de branchement.

La donnée est un programme dont on a extrait les éléments modulaires externes (appelés par le programme)

Ce programme,à l'exclusion des éléments externes,est représenté par son graphe de branchement.

4-4-1-1 Elaboration du graphe de branchement.

Si le graphe de branchement n'a pas encore été élaboré (il a pu l'être au moment de l'étude des relations entre les éléments modulaire internes) on procède à sa construction.

Définition : un graphe de branchement est un graphe de flux où

- chaque noeud représente un élément modulaire interne
- chaque arc (ni, nj) représente le passage du contrôle au moins une fois du noeud ni au noeud nj (c'est un 1-graphe)
- on peut toujours se ramener à un noeud de sortie unique.

Le processus d'élaboration du graphe de branchement à partir du texte source peut consister en les étapes suivantes (voir l'exemple de construction du chapitre)

- relever dans le texte source la liste des noms des éléments modulaires internes.
- relever dans le texte source toutes les instructions de branchement implicites ou explicites et construire la liste des éléments extrémités pour chaque élément origine.
- construire alors la matrice d'adjacence du graphe recherché (matrice d'incidence sommet-sommet).
- génération du graphe de branchement par exploration de la matrice d'adjacence ; le parcours des chemins pouvant être fait avec priorité à la profondeur.

4-4-1-2 La décomposition en sous graphes propres

b1 - Définitions :

Le graphe de branchement est un graphe quasi-fortement connexe (qfc graphe)

Rappel : un graphe G est quasi-fortement connexe s'il existe un sommet r et un sommet a de G tels que pour tout sommet x de G il existe au moins un chemin (r...x) et un chemin (x...a)

On dit que r et a sont respectivement une **racine** et une **antiracine** de G

Soit un graphe G = (X,U) et un sous graphe F = (Y, V) de G

Soit x Y :

- x est dit **point d'entrée** de F s'il existe un chemin (y, x, z) avec y X-Y, z Y et z ≠ x
- x est dit **point de sortie** de F s'il existe un chemin (z, x, y) avec z Y, y X-Y et z ≠ x

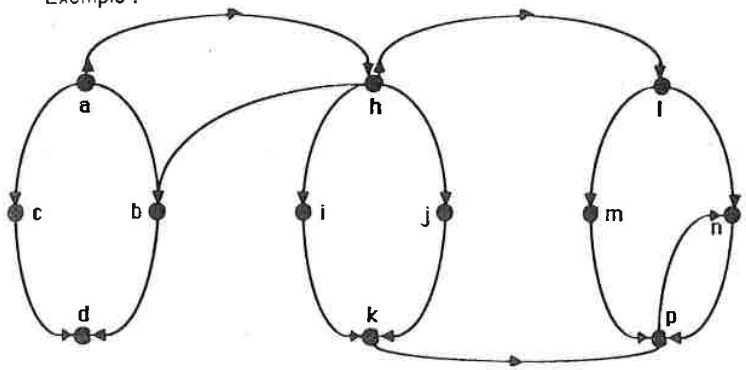
Définition d'un sous graphe propre.

Soit G un qfc graphe et F un sous-graphe de G.

On dit que F est propre s'il admet un **point d'entrée** unique et un **point de sortie** unique et si ces points sont respectivement **racine** et **antiracine** de F.

Il s'agit là de la notion de fuseau de Anne ADAM et J.P. LAURENT 78, et de schéma ou programme propre de V;R; BASILI et H.D. MILLS 82

Exemple :



Les sous-graphes quasi-fortement connexes {**a, b, c, d**} ou {**l, m, n, p**} ne sont pas propres, par contre le sous-graphe {**h, i, j, k**} est propre.

b2 - Le processus de décomposition

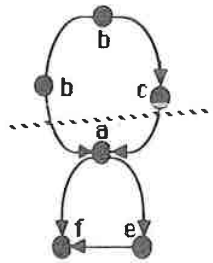
Définition :

Ensemble d'articulation et isthme;

Etant donné un graphe connexe G (X,U) un ensemble d'articulation A X du graphe connexe G est un ensemble A de sommets tel que le sous graphe G_{X-A} déduit de G par suppression des sommets de A ne soit plus connexe.

Etant donné un point d'articulation a d'un graphe connexe G on appelle sécante d'extremité a (c'est un isthme d'extrémité a) un ensemble d'arcs, ayant tous a pour extrémité, dont la suppression entraîne la non connexité du graphe et tel qu'aucun de ses sous-ensembles stricts ne possède la même propriété.

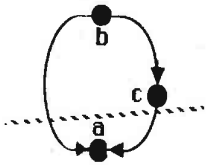
Exemple :



Les arcs (b,a) et (d,a) forment une sécante d'extrémité a

Remarque :
Un isthme n'est pas nécessairement une sécante

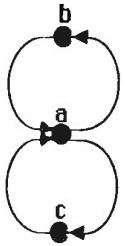
Exemple :



Le sommet a n'est pas un point d'articulation.

Inversement, étant donné un point d'articulation a , il n'existe pas nécessairement une sécante d'extrémité a.

Exemple :



De ces définitions découlent les résultats suivants :

1) Toute bipartition selon une sécante d'extrémité a conserve les circuits et si le graphe est quasi-fortement connexe cette sécante en a est unique et les sous-graphes sont quasi-fortement connexes.

2) Si le graphe est propre, sa bipartition selon une sécante le décompose en sous graphes propres.

C'est en cela que la bipartition est intéressante et aussi parce que les sous-graphes propres ainsi déterminés sont indépendants de l'ordre dans lequel on opère les bipartitions.

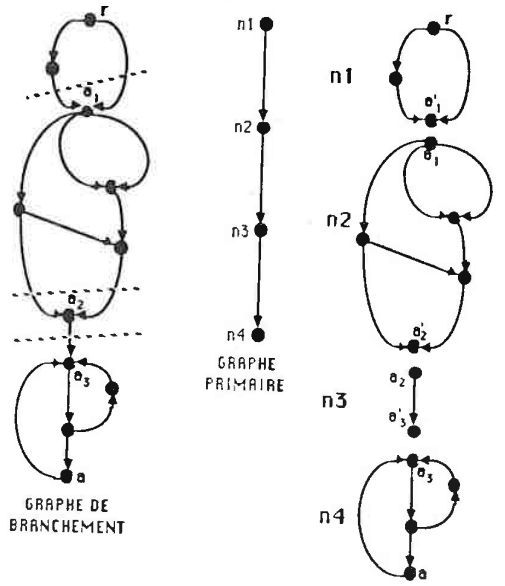
Le processus de décomposition du graphe de branchement propre G résulte de ces quelques définitions et résultats. On obtient ainsi un ensemble ordonné de sous-graphes propres constituant une structure de G représentable par le **graphe primaire de G.**

Définition :

Etant donné un graphe propre G, le graphe primaire de G est un graphe orienté tel que :

- les noeuds sont les sous-graphes propres de G
- il existe un arc de n_i à n_j, si les noeuds n_i et n_j résultent d'une bipartition.

Exemple :



Les points a'_i étant des points formels

Ce graphe primaire représente une relation d'ordre total qui est celle induite par les chemins élémentaires entre la racine et l'antiracine du graphe propre initial.

Ce graphe primaire, comme nous le verrons au niveau de la prise en compte des flux de données permet l'étude de la permutabilité entre sous graphes propres grâce à une relation de précédence basée sur les types (utilisation, définition) des références aux variables de ces sous graphes.

4-4-2 Restructuration d'un graphe de branchement propre.

4-4-2-1 Généralités et rappels :des principales approches :

Le problème de la restructuration est ramené à la dimension d'un morceau de programme correspondant à un sous-graphe propre non bipartitionable selon une sécante.

L'objectif est d'associer par application de transformations à chaque sous-graphe propre, un schéma réductible, planaire composé uniquement de schémas de base acceptés.

Compte tenu de la réduction du problème à un sous-ensemble des instructions du programme, on peut concevoir ici aussi une **participation active de l'utilisateur** dans la définition et la mise en oeuvre de stratégies d'amélioration par composition de tactiques prédéfinies ou construites par lui-même.

Depuis quelques années apparaissent de nombreux systèmes de transformation de programmes (PART (83).

En général, ces systèmes sont classés en fonction de leur but, de la classe des langages source et cible, des représentations utilisées du programme, des techniques, de l'organisation des transformations et de la forme des règles de transformation.

Pour certains systèmes (Leeds TS, MAHE 83) il s'agit de transformer automatiquement des programmes existants, pour d'autres (DEDALUS MAM 78) l'objectif est plutôt la programmation automatique.

Les programmes source et cible sont le plus souvent exprimés dans le même langage de haut niveau. Quant au volume, plusieurs milliards de lignes de PL 1 et COBOL sont actuellement à reconsidérer (Département de la défense américain).

Les deux tendances essentielles au niveau de l'organisation des transformations peuvent s'intituler :

- **l'approche catalogue**, avec un ensemble prédéfini complet de transformations utilisables.

- **l'approche générative**, avec un petit nombre initial de règles de transformation complété par un mécanisme de génération d'autres règles plus complexes.

Le programme source peut être représenté par son texte, mais également par des graphes associés (graphe de flux de contrôle, graphe de flux de données, graphe de structuration,...)

On peut envisager, utilement, de prendre en compte la dualité texte-graphe dans le cadre d'un processus assisté bénéficiant des techniques actuelles en particulier au niveau graphique.

On peut considérer qu'il existe deux classes de règles de transformation liées à la méthode de spécification des transformations.

- Dans le cadre de ce qu'on appelle les transformations symboliques, ce sont les règles de remplacement de motifs (pattern replacement rules).

- Dans le cadre des traductions dirigées par la syntaxe, ce sont les règles procédurales (semantic actions).

Notre objectif est limité à l'amélioration de programmes existants du point de vue de leur schéma (réductibilité) de leur modularité et des types des objets concernés pour contribuer à leur lisibilité et maintenabilité.

Le processus d'amélioration doit donc pouvoir être construit par l'utilisateur, facile à comprendre et modifiable par un autre donc réversible si nécessaire ; une difficulté classique est d'éviter de tomber au niveau des mécanismes de transformation dans le même écueil qu'au niveau des programmes à transformer.

Pour ce faire, la restructuration doit résulter de **stratégies** d'amélioration construites à l'aide de **tactiques** de transformations, ces dernières effectuant des modifications spécifiques sur les représentations du programme concerné.

Quelques étapes considérées comme essentielles, étapes pivots (application de la règle de pliage, élaboration d'un DFST,....) peuvent guider et aider l'utilisateur dans le processus d'amélioration. L'intérêt de telles transformations essentielles est d'éviter d'être confronté à la difficulté d'élaboration d'une liste exhaustive des transformations possibles. Un catalogue doit rester de dimension telle qu'il n'astreigne pas l'utilisateur à des recherches fastidieuses concernant l'éventuelle transformation applicable à chaque instant.

On classe les transformations en deux grands types :

Les **transformations syntaxiques** concernent directement les structures de contrôle et les **transformations sémantiques** concernant un deuxième aspect non moins important des langages celui de la structure d'information du langage c'est à dire les propriétés des objets manipulés par les programmes.

La prise en compte des objets manipulés permet non seulement des transformations sémantiques locales au niveau d'une instruction quelconque du programme (permutation, fusion de deux affectations, élimination de sélections inutiles ou redondantes, absorption,...) mais aussi des transformations plus globales (élimination d'instructions inutiles, de variables redondantes, sortie d'invariants de boucles,...) grâce à la propagation, le long du schéma du programme, de relations entre les variables associées à ces objets (relations de dépendance) ou entre variables et constructions syntaxiques (variables déclarées ou utilisées dans une branche d'alternative,...).

Au niveau des transformations de programmes, on distingue habituellement deux objectifs essentiels ; un objectif de construction pour lequel le but est d'améliorer le processus de programmation, et un objectif de compréhension dans le but d'aider à l'utilisation et à la modification du programme.

L'objectif de construction a inspiré la construction de systèmes d transformations utilisés dans le cadre de développements assités de programme de trois types essentiels, compilation étendue, métaprogrammation, synthèse d programmes. Par contre, l'objectif de compréhension a inspiré beaucoup plus le investigations "manuelles" pour tenter d'appréhender et d'expliquer les algorithme ou programmes existants, sans mettre l'accent sur la conception et la réalisation d systèmes automatisés.

Toutefois, depuis quelques années apparaissent des éléments de définition et des projets de reconception assistée de programmes (Reverse engineering, (ED ACLY, WAGNER J., 88, FUKUNAGA).

Dans la littérature, pour ce qui est de la compréhension et de l'amélioration de programmes, on trouve deux grandes catégories d'expression, de définition et de mise en oeuvre des transformations liées aux représentations utilisées de ces programmes ; l'approche **source à source** (ARSAC, CZEJDO, 85) et l'approche par les **graphes de flux.**

Dans l'approche **source à source**, la seule représentation envisagée des programmes est leur texte source.Chaque règle de transformation est composée d'une partie source et d'une partie cible. La partie source contient le schéma syntaxique à modifier, analysé à l'aide de la grammaire du langage source ainsi que la condition d'application de la règle. La partie cible contient le schéma syntaxique résultat analysable à l'aide de la grammaire du langage cible.

On trouve cette approche dans (ARSAC J.) ou le programme à analyser est exprimé par un système d'actions régulières mais ne sont pas envisagées dans ce contexte les transformations sémantiques globales qui nécessitent la prise en compte de la structure et de l'utilisation des données (flux de données).

Exemple :

Dans le fragment de programme PASCAL suivant on peut envisager de remplacer la boucle REPEAT par une bouche WHILE

```
REPEAT
    readln (elem);
    somme : = Somme + elem;
    I : = I + 1
UNTIL (elem = 0) OR (I = 100)
```

version initiale du fragment de programme.

```
readln (elem);
Somme : = Somme + elem;
I : = I + 1;
WHILE NOT ((elem = 0) OR (COUNT = 100) DO
    BEGIN
    readln (elem);
    Somme : = Somme + elem;
    I : = I + 1;
    END;
```

version finale du fragment de programme.

La transformation étant exprimée à l'aide des règles suivantes

```
Mode  : traduction avec remplacements multiples.
Indentation  : en sortie
Règle :
    Source : REPEAT suite-inst
                UNTIL condition
    Cible : suite-inst
                WHILE NOT (condition) DO
                BEGIN
                    suite-inst
                END;
```

4-4-2-2 Graphes de branchement ou de contrôle et bonnes propriétés

Dans l'approche **par les graphes de flux**, et il s'agit alors essentiellement du graphe de (flux de) contrôle, la restructuration du programme est ramenée à celle du graphe associé. Restructurer un graphe c'est lui procurer de bonnes propriétés : reductibilité, planarité,...(voir l'exemple traité CHn).

Une approche complète doit permettre d'envisager successivement les aspects syntaxiques et sémantiques du programme.

Sur le plan syntaxique, un objectif peut être d'aboutir à l'expression du schéma de programme en terme de schémas de base. Cette étape doit aussi préparer la suivante par le relevé d'informations utiles ; variables définies et utilisées, lieux de définition et d'utilisation, préconditions d'exécution des instructions ou contextes,....

Sur le plan sémantique, un objectif est l'optimisation du programme dont les retombées, en particulier lors de l'étude des flux de données, peuvent être essentielles au niveau de la modularisation.

Nous envisagerons ces deux aspects d'un programme au fur et à mesure des besoins au cours des étapes de notre processus d'amélioration de programmes.

Compte tenu du développement actuel des possibilités au niveau graphique (modifications sur l'écran de graphes) et des limitations toujours importantes dans les langages au niveau de l'expression syntaxique des schémas de base, il semble indispensable de pouvoir contrôler l'effet de toute modifcation d'une représentation du programme (son graphe de contrôle) sur une autre représentation (son texte).

L'apparente facilité de la transformation d'un graphe peut donc se traduire par de fâcheuses difficultés au niveau du texte du programme associé.

L'amélioration syntaxique d'un fragment de programme est envisagée dans un premier temps ; quant à l'amélioration "sémantique", elle sera abordée pour les besoins de la modularisation et de la linéarisation après l'étape d'arborisation ; cette dernière transformation du graphe de flux de contrôle ou de branchement donnent une représentation de ce dernier mieux adaptée à la propagation des données.

La dimension du fragment de programme étudié et son niveau de difficulté, l'intérêt de s'appuyer ou non sur sa structure actuelle peuvent argumenter son approche soit par le graphe de branchement soit par le graphe de contrôle.

4-4-2-2-1 Graphe de branchement ou de flux de contrôle et transformations syntaxiques

Définitions : Un graphe de flux de contrôle restreint (gcr) est un graphe de flux G = (N, E, n0) tel que :

1) Chaque noeud n de l'ensemble fini N représente un bloc de base c'est à dire une suite d'instructions impératives (non conditionnelles et non itératives) ne contenant aucune instruction de transfert du contrôle.

2) Chaque arc (n_i, n_j) caractérise le passage du contrôle du noeud ni au noeud n_j.

Un graphe de branchement (gb) est un graphe de flux G = (N,E,no) tel que :

1) Chaque noeud n de l'ensemble fini N représente une suite d'instructions comprises entre deux délimiteurs qui sont soit une étiquette, soit une instruction de branchement explicite.

2) Chaque arc (ni , nj) représente le passage du contrôle du noeud ni au noeud nj.

Le graphe de branchement (gb) est un sous-graphe du graphe de contrôle.

Un graphe de branchement restreint (gbr) est un graphe de branchement avec comme seul délimiteur l'étiquette.

Tous ces graphes possèdent un noeud initial (noeud d'entrée) et un noeud final (noeud de sortie) ; on peut toujours s'y ramener.

Il faut rappeler que l'utilisateur abordant un fragment de programme aura déjà éliminé les éléments modulaires externes du programme et avec eux identifié les instructions "d'appel" et par différence les instructions de branchement et/ou de contrôle.

4-4-2-2-2 Rappel de bonnes propriétés d'un graphe de flux de contrôle.

Un programme est bien structuré s'il est uniquement composé de schémas de base tous issus d'une structure déterminée par l'utilisateur et si le langage de programmation utilisé contient les instructions nécessaires à leur traduction propre.

Dans un programme existant, la difficulté est alors la localisation des constructions incorrectes et leur transformation; Au niveau du graphe de flux de contrôle associé cela consiste en l'étude de sa **réductibilité** et de sa **planarité**.

Apporter à un graphe les qualités de reductibilité et de planarité, c'est non seulement faciliter sa linéarisation et les mesures de sa complexité mais c'est aussi faciliter l'étude de la propagation des données dans le cadre de l'approche sémantique.

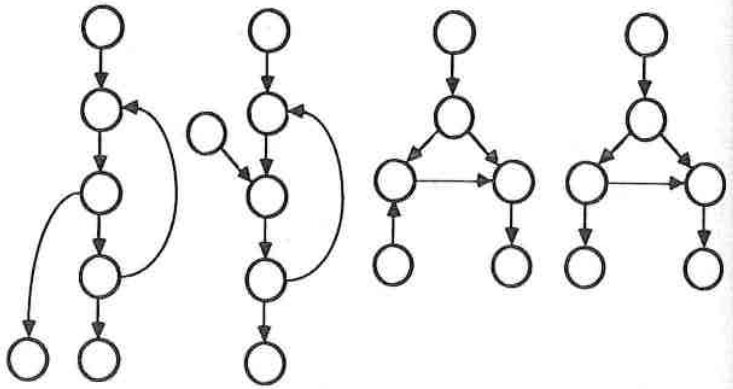
4-4-2-2-3 Planarité d'un graphe de flux : contribution à la structuration.

On dit q'un graphe G est planaire lorsqu'il admet une représentation sur un plan P par des points distincts figurant les sommets et des courbes simples figurant les arêtes, deux telles courbes ne se rencontrant pas en dehors de leurs extrémités.

Une approche duale de la planarité est celle de Kuratowski en théorie des graphes qui dit que tout graphe non planaire doit contenir au moins un des deux graphes non planaires spécifiques qu'il décrit.

De ce théorème de Kuratowski, Thomas J; Mc Cabe déduit une définition de la non structuration d'un programme qui est la suivante ;

Une condition nécessaire et suffisante pour qu'un programme (sans GO TO) soit non structuré (n'utilise pas seulement des schémas de base) est qu'il contienne comme sous graphe l'un des quatre suivants :



a b c d

En pratique Mc Cabe constate qu'une condition nécessaire et suffisante pour q'un programme soit non structuré est qu'il contienne au moins l'un des doublets (a,b) , (a,d) , (b,c) ou (c,d).

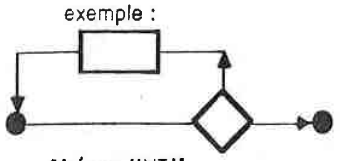
Ceci implique la possibilité d'écrire des proprammes structurés sans utiliser les branchements dans et/ou hors de boucles ou alternatives.

Une mesure de la structuration, liée à la réductibilité du graphe, (décomposition en sous-graphes propres avec noeud d'entrée unique et noeud de sortie unique) est celle de sa complexité essentielle notée ev = v - m

La complexité essentielle ev d'un programme structuré est 1 :

- où v = e - n + 2 p est la complexité cyclomatique du graphe associé avec e arcs, n noeuds et p composants connexes
- où m est le nombre de sous-graphes propres

exemple :

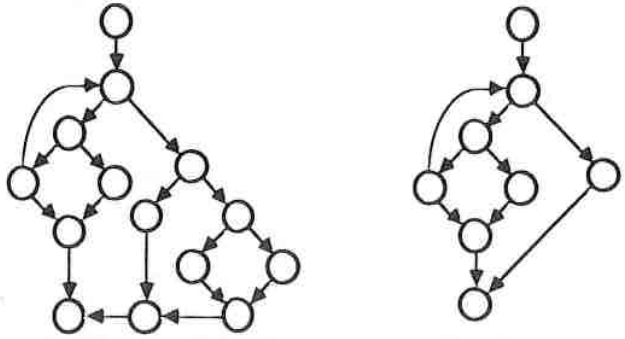


Shéma UNTIL

v = 3 - 3 + 2 = 2
ev =2 - 1 = 1
complexite cyclomatique

La différence entre la complexité essentielle et la complexité cyclomatique indiquant le degré de reductibilité du graphe.

exemple :



G : v (G) = 6 ev (G) = 4

G' : v (G) = 4

Remarque : complexité, tests et flux de données :

Sachant qu'elle représente le nombre de chemins linéairement indépendants, la complexité cylomatique d'un programme est utile, également, pour gagner du temps au niveau des tests à effectuer sur ce programme.

Etant donné un programme P de complexité cyclomatique v, soit cr le nombre de chemins testés (complexité réelle) ; si cr est plus petit que v alors l'une des conditions ci-dessous est vraie :

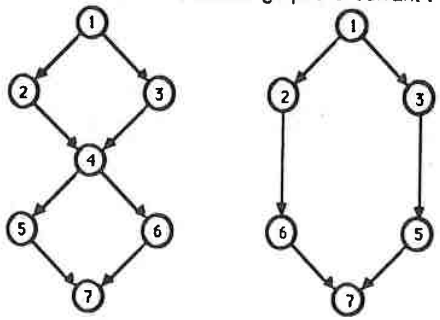
1) il y a d'autres chemins à tester;

2) le graphe de flux du programme peut être réduit v - cr fois (élimination de v-cr décisions);

3) des morceaux de programme peuvent être réduits à du code linéaire.

exemple :

Considérons à nouveau le graphe G suivant :



GRAPHE G

GRAPHE G'

Si l'on suppose que cr = 2 et que les deux chemins testés sont (1,2,4,6,7) et (1,3,4,5,7), les deux autres chemins (1,2,4,5,7) et (1,3,4,6,7) ne pouvant être exécutés on a cr < v et donc le graphe G peut être réduit en G' (condition 2 vraie)

4-4-2-2-4 Réductibilité d'un graphe de flux :

Quant à la réductibilité d'un graphe, il en existe plusieurs définitions essentiellement liées au processus de réduction.

Réduire un graphe consiste à chaque étape du processus à remplacer par un noeud un sous graphe ayant certaines propriétés.

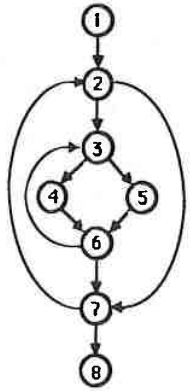
a) - Etant donné un graphe de flux G = (N,E,n_o) la définition de base de sa réductibilité de F.E. Allen et J. Cocke (ALLEN, 76) est exprimée en termes d'intervalles.

Etant donné un noeud n, un intervalle I (n) est le sous-graphe maximal à point d'entrée unique dans lequel n est ce seul point d'entrée et dans lequel tous les chemins fermés contiennent n ; n est appelé noeud tête (ou tête de l'intervalle).

La définition de l'ensemble des noeuds têtes d'un graphe permet donc de le décomposer en un ensemble unique d'intervalles disjoints (voir algorithme de F.E. Allen et J.Cocke)

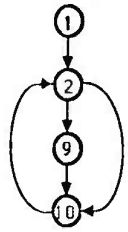
L'application répétitive de ce processus permet de passer successivement d'un graphe d'ordre j à un graphe d'ordre j + 1 ; le graphe initial étant **réductible** s'il existe un graphe d'ordre k réduit à un noeud.

exemple :



G1

I (1) = 1
I (2) = 2
I (3) = 3,4,5,6
I (4) = 7,8



G2

I (1) = 1
I (2) = 2,9,10

G3

I (1) = 1,11

G4

I (12) = 12

b) - Une définition équivalente de la réductibilité est celle de M.S. Hecht et J.D. Ullman

Etant donnés un graphe de flux G = (N,E,n_o) et les deux transformations suivantes d'un graphe de flux

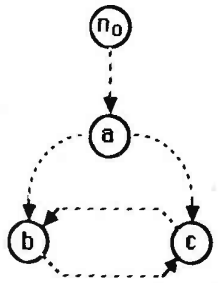
T1 : enlever un arc (n,n)

T2 : étant donné un arc (n1, n2) où n2 n'est pas le noeud initial et possède un prédécesseur unique n1, remplacer l'arc (n1,n2) par un noeud unique n.

On dit que G est réductible par intervalles si l'application répétitive de T1 et T2 conduit à un graphe composé d'un seul noeud.

Mais ce qui est intéressant, surtout, c'est la propriété de non réductibilité d'un graphe de flux de Hecht et Ulman.

Un graphe de flux est non réductible si et seulement si il contient le sous graphe désigné par (•)



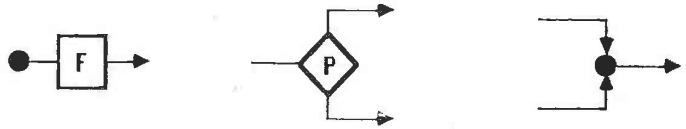
Le sous-graphe (*)

Ainsi de la même façon que par le biais de la planarité, on arrive à une caractérisation rapide du manque de structuration d'un graphe de flux; on parvient ici à une approche immédiate de la non réductibilité de ce graphe.

Un autre intérêt de la démarche de Hecht et Ulman (HECHT, 74) est de relier la propriété de réductibilité d'un graphe de flux , G, à sa définition à partir d'un arbre de parcours avec priorité à la profondeur (DFST;depth-first spanning tree) de G ; l'algorithme de Tarjan (TARJAN, 72) dans ce cas, définit un autre ordonnancement des noeuds que celui du processus de détermination des intervalles.

Ces deux mécanismes d'ordonnancement des noeuds d'un graphe sont utiles au niveau de l'étude des flux de données ; ainsi dans les algorithmes d'atteignabilité la rapidité de stabilisation s'accomode de la méthode par intervalles tandis que pour la facilité et une certaine normalisation on peut préférer l'ordonnancement de Tarjan et c'est là notre choix.

c) - Une dernière approche de la réductibilité de V.R. Basili et H.D. Mills (BASILI V.R.) est liée à la représentation du programme à analyser sous forme de diagramme de flux (organigramme) composé uniquement de trois types de noeuds :

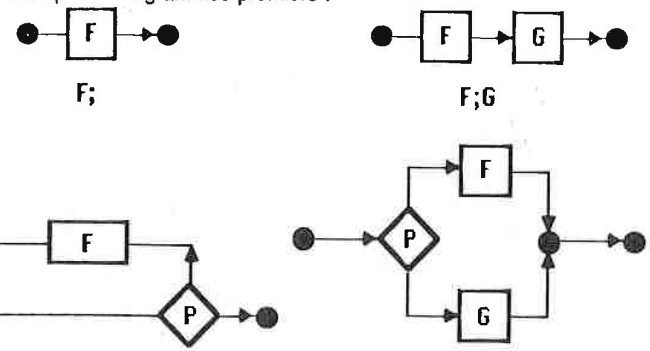


noeud fonction noeud prédicat noeud de regroupement

Un programme est dit propre si son diagramme de flux possède un seul point d'entrée ,ne, un seul point de sortie ,ns, et si pour chaque noeud ,n, il existe un chemin de n_e à n_s passant par n.

La décomposition d'un programme propre en programmes premiers (décomposition non unique) correspond à la réduction du diagramme de flux associé à l'aide de diagrammes premiers.

Exemple de diagrammes premiers :

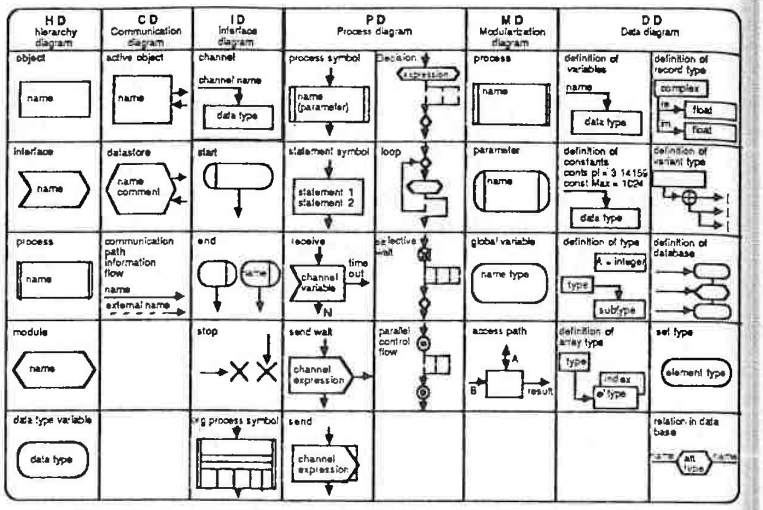


s'agit là des diagrammes associés aux schémas de base d'une structure déterminée.

Cette approche de Basili et Mills est "manuelle" tant au niveau de l'élaboration du diagramme de flux que de sa transformation (essentiellement par recopie de noeuds) pour le rendre réductible mais leur permet ensuite une décomposition du programme en fonctions et l'aboutissement à sa compréhension grâce à l'élaboration d'une abstraction fonctionnelle par examen direct des morceaux de programme les plus petits, par mise en évidence d'invariants pour les boucles et la définition de fonctions pour les morceaux les plus importants (voir U.D.P. de V.R. BASILI et H.D. MILLS (BASILI, V.R.)).

Compte tenu du niveau croissant des moyens de traitement graphiques et de l'intérêt qu'a toujours représenté l'organigramme (ou ordinogramme) au niveau des méthodes d'analyse, cette dernière approche de la transformation de programmes permet d'envisager une certaine homogénéité des représentations tout au long du processus d'automatisation d'une application ; on peut citer, par exemple ici le tableau ci-dessous des symboles du langage de conception GRAPES 86 du projet CARE de J.Wagner /Siemens AG (WAGNER J.).

Pour peu que le langage de programmation associé (si encore nécessaire) possède les structures de contrôle qui correspondent à chaque diagramme premier la décompilation est immédiate.



SYMBOLES DU LANGAGE DE CONCEPTION GRAPES 86 (Projet CARE).

4-4-2-3 Acquisition des propriétés de planarité et de réductibilité.

Acquérir les propriétés de planarité et de réductibilité pour un graphe nécessite essentiellement la mise en oeuvre de transformations modifiant les circuits de ce graphe.

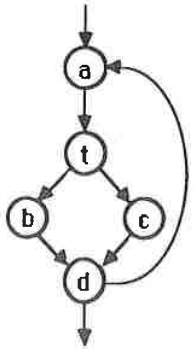
4-4-2-3-1 Quelques définitions et transformations syntaxiques classiques

a) Circuits et boucles

En théorie des graphes, on appelle boucle un arc ayant origine et extrémité confondues.

En programmation, on appelle **boucle** l'ensemble des circuits d'un graphe ayant mêmes points d'entrée et mêmes points de sortie.

exemple :

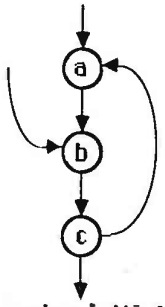


Les deux circuits de ce graphe (a,t,b,d) et (a,t,c,d) ayant mêmes points d'entrée et mêmes points de sortie constituent une seule boucle.

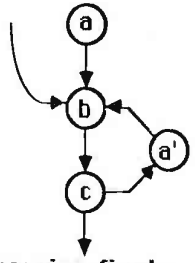
b) Points d'entrée dans les boucles

Lorsqu'une boucle a plusieurs points d'entrée, il est intéressant dans un but de structuration du graphe de la décomposer en boucles simples ayant un seul point d'entrée ; cette transformation est toujours possible par recopie des noeuds situés entre les différents points d'entrée.

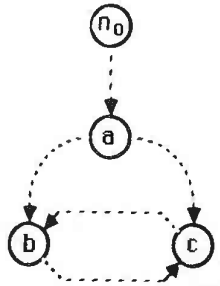
exemples :

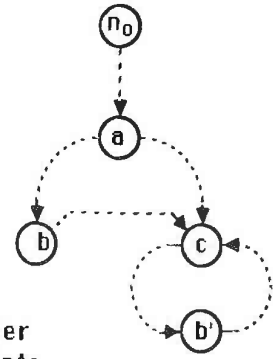


version initiale



version finale

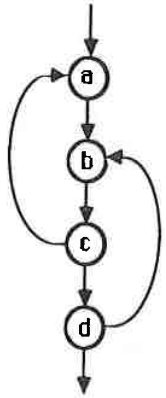


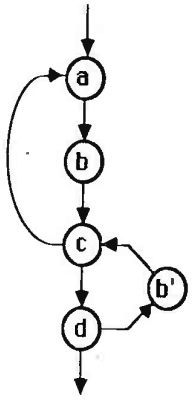


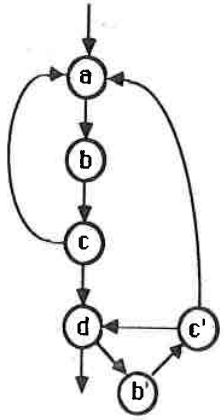
Le sous-graphe (*)
où chaque arc peut représenter
un ensemble de noeuds disjoints.

Cas des boucles enchevêtrées :

exemple :







version initiale
avec trois circuits

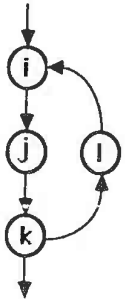
recopie de b

recopie de c

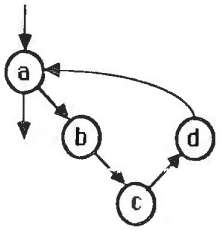
c) Boucles à un seul point de sortie

Il existe plusieurs formes de boucles ayant un seul point d'entrée et un seul point de sortie.

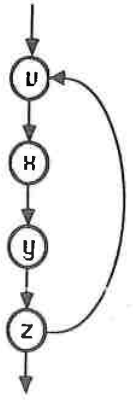
exemple :



forme générale



boucle WHILE

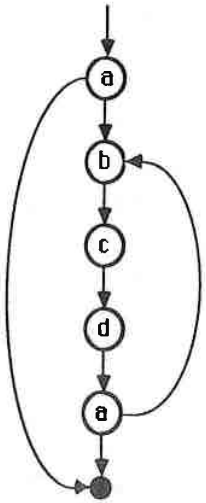


boucle DO

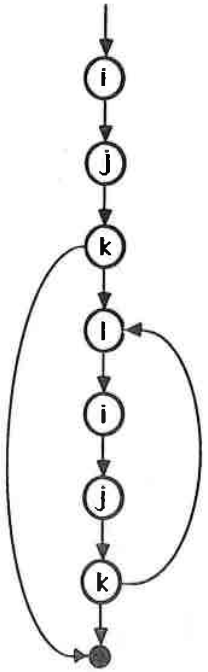
On peut toujours se ramener à une boucle DO pour les deux raisons suivantes (Anne ADAM) :

- les transformations mises en oeuvre sont celles qui nécessitent le moins de recopie de points.
- l'ensemble des instructions d'une boucle DO constitue un fuseau ; intuitivement construit à l'aide d'un chemin avec priorité à la profondeur.

exemples :



transformation en boucle DO de la boucle WHILE ci-dessus (recopie de a)



transformation en boucle DO de la boucle générale ci-dessus (recopie de i, j, k)

d) Boucles à plusieurs points de sortie.

Si la recopie de points permet de résoudre le problème des points d'entrée multiples, il n'en va pas de même pour les points de sortie multiples (D.E. Knuth et R.W. Floyd)

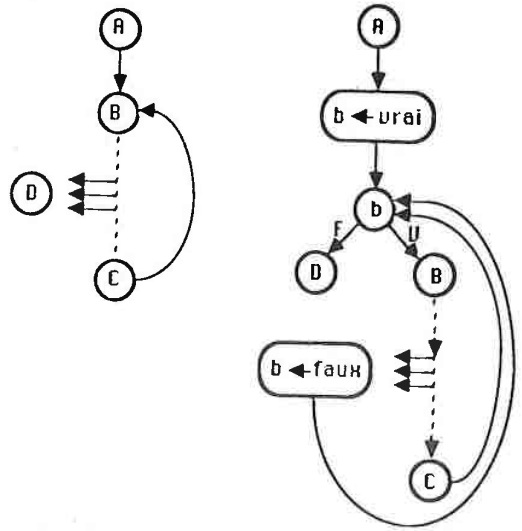
Cette dernière transformation intéressante au niveau de la recherche de planarité et de reductibilité est nécessaire pour peu que le langage cible ne possède pas les instructions et structures de contrôle correspondant aux BJn et

REn schémas et permettant leur traduction sans ajouter tests et branchements explicites (exit (i), entrée (j), return (k),....)

Dans ce cas, la structuration de la boucle nécessite l'utilisation de variables supplémentaires;

- soit des variables de travail pour mémoriser des valeurs de variables du programme aux points critiques de l'exécution du corps de la boucle.
- soit des variables booleennes pour mémoriser la réalisation d'une condition de sortie.

exemple :

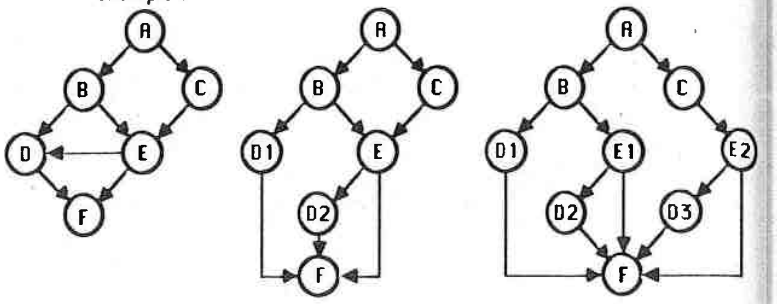


Introduction de la variable booléenne b

e) Treillis et recopie de points.

Dans le cadre d'un morceau de programme sans boucle, la recopie de points de façon ordonnée permet toujours de ramener un treillis à une composition d'arbres simples ; l'algorithme STRUCTURE de A.L. Baker utilisant les notions de noeud dominateur d'un graphe et de noeuds suivants d'un noeud (B.S. Baker) assure cette transformation.

exemple :



Treillis initial

Toute autre transformation, par exemple la traversée d'un test par un noeud, la fusion et l'éclatement de boucles, nécessite de posséder de l'information sur le contenu des instructions et on aborde alors le domaine des transformations sémantiques.

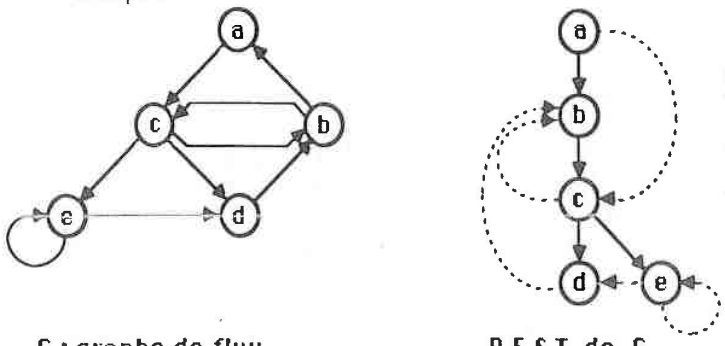
4-4-2-3-2 Une représentation "normalisée" du graphe de flux.

a) Notion de DFST (depth-first search tree).

Une bonne représentation d'un graphe de flux est celle issue d'un arbre d'analyse avec priorité à la profondeur (DFST).

Le DFST d'un graphe de flux G est un arbre d'analyse orienté de G construit par l'algorithme de R.E. Tarjan.

exemple :



G : graphe de flux **DFST de G**

Si (u, v) est un arc du DFST alors u est le prédécesseur immédiat (pi) de v, v est le successeur immédiat (si) de u, les relations prédécesseur et successeur étant les fermetures transitives de p_i et s_i.

Le graphe de flux est alors représenté par un DFST additionné d'arcs de trois classes possibles :

1) les arcs en avant (u, v) tels que u et v sont des noeuds du DFST et u est un prédécesseur de v ; arc (a, c) dans l'exemple.
2) les arcs en arrière (u, v) tels que u et v sont des noeuds du DFST et u est un successeur de v ; arcs (d, b) et (c, b) dans l'exemple.
3) les arcs en travers dont l'origine et l'extrémité ne sont pas dans le même sous arbre du DFST ; arc (e, d) dans l'exemple.

b) Réductibilité et DFST.

On doit à M.S. Hecht et J.D. Ullman l'élaboration de quelques propriétés intéressantes des graphes de flux réductibles représentés à partir d'un DFST;

Le DFST permet l'expression simple et visuelle de propriétés caractéristiques des graphes de flux réductibles, propriétés utiles au niveau des étapes d'arborisation et de modularisation de même qu'au niveau de l'analyse des flux de données.

L'élaboration d'un graphe de flux, à l'aide d'un DFST , permet non seulement une représentation graphique "normalisée" de ce graphe mais aussi une approche de l'étude de sa réductibilité plus pratique, l'application des transformations syntaxiques étant plus "visuelle". Quant aux transformations liées à la sémantique, elles bénéficient de cette représentation en particulier au niveau de l'analyse des flux de données (localisation des boucles et de leurs points d'entrée, adjacence, couplage,.....)

Etant donné G un graphe de flux réductible (par exemple au sens des intervalles), en abrégé gfr ;

G est décomposable de façon unique en un sous graphe acyclique maximum (son **dag**) et un ensemble d'arcs en arrière.

Ce dag est constitué d'un DFST de G complété par les arcs en avant et les arcs en travers.

Relation de domination :

Etant donné un graphe de flux $G = (N,E,n_0)$ et deux noeuds distincts de G, soit m et n ; le noeud m **domine** le noeud n dans G si tout chemin dans G de n_0 à n passe par m.

Cette relation de domination est schématisée par le dag de G.

Tout graphe de flux ne possédant que des cycles à un seul point d'entrée est réductible.

Chaque arc en arrière d'un graphe de flux réductible définit une boucle avec un seul point d'entrée ; ce point dominant tout autre point de la boucle (à rapprocher de la notion de graphe propre).

4-5 CONCLUSION

Ce chapitre constitue une synthèse des quelques caractéristiques et propriétés essentielles des graphes (et sous-graphes) les plus utiles en programmation.

Faire acquérir aux graphes représentatifs d'un programme (issus de la fonction **appréhender**) par l'un des mécanismes de restructuration envisagés dans ce chapitre certaines caractéristiques ou propriétés c'est par exemple contribuer **à :**

• faciliter l'application de la fonction ARBORISER et contribuer à l'amélioration de la "lisibilité" du graphe et du programme associé (LINEARISER)..

• faciliter la maintenance future par l'utilisation et la conservation de qualités des graphes, comme leur réductibilité, leur planarité.

Mais au-delà de cette restructuration, l'utilisateur non spécialiste peut souhaiter passer à une représentation arborescente de son programme, représentation peut être plus facilement linéarisable et correspondant nécessairement à un modèle bien **typé** (opérateurs bien connus) dans un espace à deux dimensions (plan).

I 5 ARBORISATION D'UN SOUS -GRAPHE PROPRE : ARBORISER

Dans ce chapitre, nous présentons la fonction arboriser; il s'agit en fait d'une démarche assez répandue que l'on trouve, par exemple, dans le passage d'une syntaxe concrète (représentation linéaire) à une syntaxe abstraite (langage pivot dans lequel les programmes sont des arbres) ou dans la définition de la sémantique d'une langue naturelle par CHOMSKY en termes de structure de surface et de structure profonde.

Nous posons tout d'abord le problème de l'arborisation d'un programme, c'est à dire sa représentation sous forme arborescente, et différemment des solutions rencontrées dans la littérature pour le résoudre, nous exposons ensuite notre propre solution.

Cette solution "paramétrable"

• privilégie au maximum l'aspect graphique ("visuel") des représentations du programme.

• permet à l'utilisateur de passer d'une représentation d'un programme sous forme de graphe de flux normalisé à une représentation arborescente grâce à une décomposition s'appuyant sur le concept de schéma de base.

5-1 LE PROBLEME DE L'ARBORISATION : LES HYPOTHESES .

Le but est ici de donner d'un programme une représentation arborescente facilitant une certaine modularisation et la linéarisation.

La donnée est un (morceau de) programme propre représenté syntaxiquement par son graphe propre réductible ; le résultat est une arborescence dont la racine contient le noeud d'entrée du graphe, les niveaux correspondant à une certaine imbrication et une hiérarchisation des constituants du (morceau de) programme.

Cette arborisation peut être envisagée pratiquement à deux niveaux :

- au niveau modulaire par l'arborisation du (sous) graphe de branchement propre si l'on envisage de respecter (en partie) la structuration existante en éléments modulaires.

- au niveau élémentaire par l'arborisation du (sous-) graphe de contrôle restreint associé au (morceau de) programme étudié.

Le graphe concerné par le processus d'arborisation est supposé posséde
les propriétés de reductibilité et de planarité (voir section 2)

Le graphe étudié est également supposé construit à partir d'un Depth Firs
Search Tree (DFST) dont le processus d'élaboration permet un ordonnancemen
des noeuds dont on peut ainsi constituer une liste L = {n_1, n_2,...n_p} où p est l
nombre de noeuds du graphe (algorithme de nomérotation de TARJAN (TARJAN
72)),.

Cette liste L est utilisée par la suite au niveau de l'arborisation,mais auss
pour faire en sorte que tout branchement explicite (GO TO) introduit dans l
programme cible soit un branchement en avant et enfin pour s'assurer que tous les
noeuds seront bien pris en compte.

5-2 LE PROCESSUS D'ARBORISATION DU GRAPHE DE FLUX NORMALISE.

5-2-1 Introduction.

On appelle graphe de flux normalisé un graphe de flux propre reductible et
planaire construit à partir d'un DFST dont les noeuds sont numérotés par
l'algorithme précédent.

On trouve dans la littérature différents processus de linéarisation de graphes
s'appuyant sur la réductibilité de ceux-ci.

Ainsi l'algorithme de B.S. Baker propose une linéarisation d'un graphe de
flux, représenté à l'aide de son DFST, graphe supposé réductible.

Cet algorithme n'utilise pas de recopie de code, ni de nouvelles variables, ni
de création de procédures mais en contrepartie doit permettre l'introduction
systématique de branchements explicites (GO TO) entrant dans le corps d'une
boucle (REPEAT) ou dans une branche (THEN ou ELSE) d'une instruction
conditionnelle. Mais si ces branchements n'altèrent pas la lisibilité très localement,
par contre, ils ne contribuent pas à améliorer l'approche globale du programme.

On peut préférer laisser à l'utilisateur une certaine liberté de choix en ce qui
concerne l'aspect du code cible, l'utilisation des instructions du langage cible et
donc partir d'hypothèses moins restrictives au niveau des transformations
possibles.

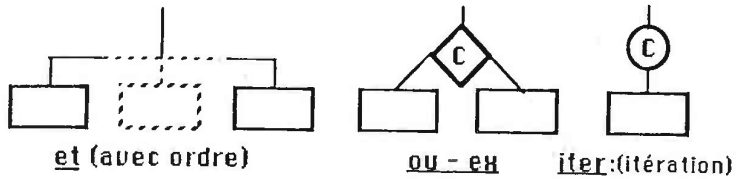
Notre processus d'arborisation et donc de linéarisation n'implique pas un
choix unique au niveau des transformations, permettant ainsi d'envisager
différentes possibilités de modularisation.

5-2-2 Le processus d'arborisation.

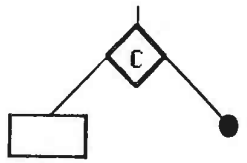
La donnée est un graphe de flux normalisé G = (N,E,n_o); par exemple un graphe de branchement normalisé où chaque noeud correspond à un élément modulaire du programme source, chaque arc (n_i, n_j) entre deux éléments modulaires correspond à au moins un passage du contrôle de n_i à n_j.

Le résultat à obtenir est un arbre généralisé A (G) permettant de représenter à la manière des arbres déductifs les grands types de schémas de base;

Les éléments constituants les plus courants (D-schémas) d'un tel arbre étant représentés de la manière suivante ;



Chaque rectangle représente à son tour un arbre généralisé ou l'arbre vide alors noté • ; ainsi une alternative avec une branche vide est représentée par :



Mais on peut envisager pour un tel arbre généralisé des constituants correspondant à des schémas de type BJ_n ou RE_n , si le langage cible s'accommode bien de leur traduction.

Cet arbre généralisé constitue une partie de la spécification, à un certain niveau, des traitements effectués par ce programme.

Une partie complémentaire de la spécification, utile au moment de la modularisation et de la linéarisation, est la documentation (décoration) associée à cet arbre concernant le flux de données, les contextes, etc...(voir 3.3)

Etant données la représentation de G à l'aide d'un DFST et la liste L ordonnée de ses noeuds, on sait classer les arcs en trois catégories ; les arcs en avant d'un prédécesseur à un successeur dans L, les arcs en arrière d'un successeur à un précédesseur dans L et les arcs en travers (voir exemple CH 3).

L'extrémité d'un arc en arrière étant, pour un graphe normalisé, le noeud d'entrée dans une boucle, peut aussi être, par exemple, un noeud conditionnel; aussi pour faciliter la décomposition du graphe et distinguer les différents rôles d'un même noeud nous introduisons un noeud formel comme prédécesseur immédiat, sur le DFST, de chaque extrémité d'un arc en arrière.

Quelques définitions :

Demi-degré :

Etant donnés un graphe G et un sommet i de G le demi-degré exterieur (intérieur) de i, noté d^+_G (i) (d^-_G (i)), est le nombre d'arcs ayant i comme extrémité initial (terminale).

Noeud impératif :

On appelle ainsi tout noeud n de G de demi-degré extérieur $d^+_G(n) = 1$

Noeud alternatif :

On appelle ainsi tout noeud n de G de demi-degré extérieur $d^+_G(n) > 1$

Noeud itératif :

On appelle ainsi tout noeud n de G de demi-degré extérieur $d^+_G(n) = 1$ et qui est l'extremité d'au moins un arc en arrière.

Noeud formel itératif :

Etant donné un noeud n de G extrémité d'un arc en arrière, on introduit un noeud formel p, dit noeud formel itératif, tel que tout arc (q,n) est remplacé par la succession des arcs (q,p) et (p,n) dans le graphe étendu G'

Noeud de terminaison :

Etant donné un sous-graphe g de G et un noeud n de g.

Si n est un noeud alternatif, le noeud de terminaison de l'alternative est le premier noeud commun à tous les chemins dans g d'origine n en supposant un seul parcours des circuits éventuels.

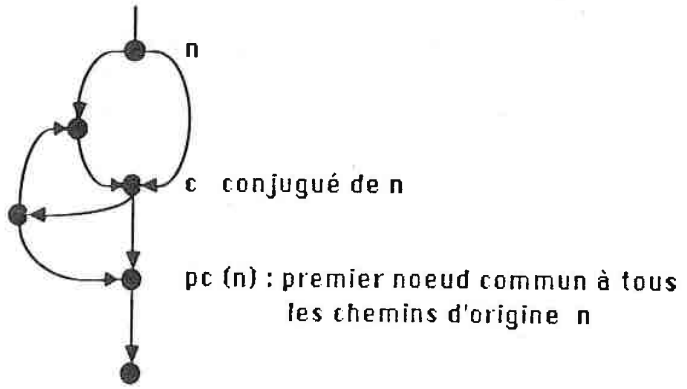
Si n est un noeud itératif, le noeud de terminaison de l'itération est le noeud successeur immédiat dans le DFST de G du noeud origine de l'arc en arrière caractérisant l'itération concernée.

Dans le cas particulier où n est un noeud impératif, on peut considérer que le noeud de terminaison est le premier successeur de n.

On note pt_g (n) le noeud de terminaison dans g associé à n.

Remarque :

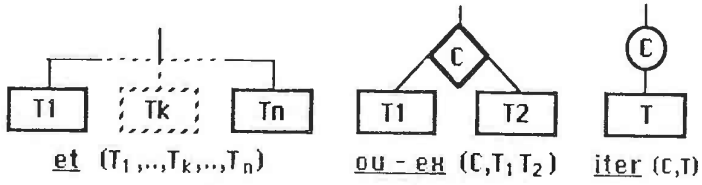
Cette notion de noeud de terminaison est différente de celle de "point conjugué d'un noeud" de PITRAT (PITRAT J., 71) sauf dans le cas où le graphe est sans circuit comme le montre l'exemple suivant :



Notations :

1 - Etant donnés un graphe G, son arbre généralisé AG (G) et un sous-graphe g de G, on note noeud $_{AG}$ (g) le noeud de AG (G) associé au sous-graphe g.

2 - Constituants de base de l'arbre généralisé associé à un graphe G



Chaque T_i symbolise un ensemble de noeuds d'un sous-graphe de G ; C symbolise une expression conditionnelle ou désigne le noeud de G contenant l'expression .

5-2-3 L'algorithme d'arborisation

Construction de l'arbre généralisé associé à G soit AG (G) :

Considérer le noeud racine de G, premier élément de la liste L associée à G et un sous-graphe g de G initialisé à G. La racine de AG (G) est noeud AG (G)

Pour chaque sous-graphe g de G considérer :

le noeud racine de g soit n_r, premier noeud dans L du sous-ensemble de L associé à g c'est à dire qui contient tous les noeuds de g.

si n_r est un noeud impératif

alors - *décomposer L en premier (L) et reste (L)*

- *attacher à noeud AG (g) un composant*

(et, premier (L), reste (L))

- *prendre pour L, reste (L) soit L = reste (L)*
- *prendre pour g, g dont on enlève nr soit g = g - (n_r)*

si n_r est un noeud alternatif

alors - *décomposer L en premier (L) et reste (L)*

- *rechercher l'élément de coupure ec de reste (L) tq ec est le premier élément r de reste (L) tq tous les chemins issus de premier (L) passent par r*

% ec représente dans L le pt_g (nr)

- *attacher à noeud AG (g) un composant (et, premier (L), pg (reste (L)), pd (reste(L))).*
- *décomposer pg (reste (L)) en deux sous-listes non nécessairement disjointes;*
- *% l'une des sous-listes peut être vide, sauf s'il existe un point formel de regroupement.*
- *slv : sous-liste des éléments correspondant aux noeuds de g accessibles depuis n_r par la branche "vrai".*
- *slf : sous-liste des éléments correspondant aux noeuds de g accessibles depuis n_r par la branche "faux".*
- *attacher à noeud AG (pg(reste (L))) un composant*

(ou-ex, premier (L) , slv , slf)

- *prendre pour g successivement les sous-graphes associés à slv et slf.*

si n_r est un noeud itératif (noeud formel)

alors - *rechercher l'élément de coupure ec de L, tq ec est l'élément de L qui correspond dans g au successeur immédiat sur le DFST (G) de l'origine de l'arc en arrière caractérisant l'itération considérée.*

% ec représente dans L le pt_g (n_r)

- *attacher à noeud AG (g) un composant*

(et , (iter (C, pg (L))) , pd (L))

- *prendre pour g successivement les sous-graphes associés à pg (L) et à pd (L)*

Remarques :

1 - Voir en annexe l'application de cet algorithme à un programme COBOL;

2 - La possibilité d'avoir à raisonner en terme de sous-listes de L (et non en terme de sous-ensembles) résulte d'un choix correct de DFST de G; en particulier la branche contenant le noeud d'arrêt du programme ou de sortie doit être parcourue (recherche en profondeur) la première.

Cas particuliers :

1) Dans le cas où pour un noeud n d'un sous-graphe g de G on a d^+_g (n) >2, on peut envisager deux solutions :

- l'alternative multiple généralisée est décomposée en alternatives simples ce qui nécessite l'analyse de l'élément modulaire associé à n et éventuellement sa décomposition en éléments modulaires plus petits, voir en blocs de base et on est ramené au graphe de flux de contrôle de cet élément modulaire.
- l'alternative multiple généralisée fait partie des constituants possibles de l'arbre généralisé (c'est par exemple une alternative multiple correspondant à une instruction CAS) et est analysée comme telle.

2) Dans le cas où il existe plusieurs noeuds impératifs consécutifs, l'utilisateur peut les regrouper en un seul noeud impératif au niveau de l'arbre généralisé, réduisant ainsi le nombre de décomposition de ce dernier.

3) L'utilisateur peut accepter ou non comme structure (type de noeud) à envisager non seulement les D-schémas, mais aussi les BJ_n - et Re_n - schémas si le langage cible offre les structures de contrôle et instructions correspondantes (voir les extensions de FORTRAN, PL/1, les possibilités de ADA,...)

Conclusion :

La fonction ARBORISER permet à l'utilisateur d'associer à tout graphe de flux d'un programme une arborescence grâce à un processus de décomposition dont le résultat dépend des choix effectués par cet utilisateur;

- choix des propriétés attendues du graphe de flux initial (application éventuelle de la fonction RESTRUCTURER)
- choix des schémas de base, paramètres de la décomposition.

5-3 DE L'ARBORISATION A LA MODULARISATION : TRANSFORMATIONS SEMANTIQUES.

Les moyens mis en oeuvre dans les étapes précédentes concernent essentiellement l'aspect syntaxique du programme. Il n'en va pas de même pour la modularisation où les critères pris en compte, en dehors de la taille du morceau de programme et de sa complexité cyclomatique ou essentielle, nécessitent de l'information sémantique (nature des traitements, flux de données,....)

De plus, tout processus de constitution de modules pour le programme cible ne peut s'appuyer que sur du code assaini (sans code mort, sans redondance,...) et amélioré (permutations, fusions, absorptions,...); un tel code ne peut résulter que de l'application de transformations sémantiques.

Aussi après quelques généralités sur la sémantique en programmation, nous rappelons les deux classes principales d'analyse du flux de données rencontrées dans la littérature et précisions à propos de la définition et de la construction d'une relation de dépendance générale une troisième classe d'analyse avec appui sur une représentation du programme en terme de composition de schémas de base.

Nous terminons en rappelant quelques unes des transformations sémantiques qui peuvent résulter de l'analyse du flux de données.

5-3-1 La sémantique en programmation

Pour ne pas aborder une formalisation de la sémantique dans notre travail, mais sans exclure cette possibilité ultérieurement, nous introduisons de façon pratique la sémantique en programmation à plusieurs niveaux;

- la sémantique des données
- la sémantique des instructions
- la sémantique des programmes
- la sémantique du problème

5-3-1-1 la sémantique des données

Au-delà des **structures de contrôle** permettant l'écriture d'algorithmes, un autre aspect important des langages est celui qui concerne la structure d'information, c'est à dire les propriétés des objets manipulés par le programme.

Ces propriétés peuvent être relevées dans le programme, il s'agit alors d'une partie (statique) de la sémantique du programme, ou découler de la connaissance des objets du monde réel (l'entreprise) et des relations qui existent entre eux.

La première approche considère la sémantique des données comme contenue, en partie, dans l'ensemble des déclarations d'identificateurs

de types, de types abstraits,...(d'objets simples ou complexes), de procédures,... et prend appui sur la structure syntaxique du programme pour l'étude des **flux de données** associées.

Ainsi on peut relever dans les déclarations des relations entre les variables associées aux objets comme par exemple : la relation de dépendance hiérarchique (structurelle), la relation de concaténation entre variables au sein d'une même structure, la relation de redéfinition (synonymie) entre variables partageant de la mémoire.

La seconde approche consiste en la représentation des objets réels de l'entreprise.

Ainsi dans le cadre de la gestion d'une entreprise si la transformation des programmes est imposée par le passage d'un SGF (système de gestion de fichiers) à un SGBD relationnel l'expression de la sémantique des données nécessite un **modèle conceptuel** de données (REMORA, MERISE,...).

5-3-1-2 La sémantique des instructions :

Du point de vue sémantique, chaque instruction peut être interprétée à l'aide d'un petit nombre **d'actions élémentaires** sur les variables dont elle contient une occurrence au moins ; chaque variable étant en réalité un couple (nom de variable, valeur).

On peut citer les actions élémentaires suivantes :

- lecture d'une valeur
- écriture d'une valeur
- calcul d'une valeur à l'aide d'une expression (ensemble de valeurs de variables, de constantes, d'opérateurs) arithmétique ou logique.
- affectation d'une valeur.

La sémantique des instructions permet un certain nombre de transformations; par exemple élimination des impuretés, substitution d'une constante à une expression, permutations d'opérandes dans une expression,...

Les notions de variable utilisée et de variable définie contribuent également à la définition sémantique des instructions.

- une **variable est définie** à l'endroit de l'association d'une valeur à son nom.

C'est le cas d'une variable membre gauche d'une affectation, d'une variable dans un ordre de lecture. Pour ce qui est des appels de procédures, il faut tenir compte des effets de bord ; au niveau des procédures non récursives, cela nécessite le calcul de l'ensemble des entrées et de l'ensemble des sorties de cette procédure, et dans le cas des procédures récursives il est nécessaire d'utiliser une

méthode itérative (A.V. AHO et J.D. ULLMAN) ou (B.K. ROSEN), ou (P.FAIRFIELD et M.A. HENNEL).

De plus, on impose habituellement certaines hypothèses simplificatrices comme l'interdiction de paramètres de type étiquette ou procédure, et un seul point d'entrée à une procédure.

- une **variable est utilisée** à l'endroit où sa valeur intervient dans un calcul.

C'est le cas d'une variable dans le membre droit d'une affectation ou ayant une occurrence dans une expression logique.

On a le même problème d'effets de bord avec les procédures.

A une instruction x on peut donc associer :

D_X : ensemble des variables définies par x

U_X : ensemble des variables utilisées dans x

5-3-1-3 La sémantique du programme :

Sans aller jusqu'à considérer un programme comme une fonction ou un calcul, on peut envisager sa sémantique comme constituée non seulement de la sémantique des données et des instructions mais aussi essentiellement :

- de son flux de contrôle
- de son éventuel flux d'appel
- de son flux de données voir (3-3-1)

5-3-1-4 La sémantique du problème :

Elle est constituée d'informations non contenues dans le programme (sauf partiellement s'il est très bien documenté) et liées au problème résolu par ce programme, voir à l'environnement, au processus de conception, à la culture de l'utilisateur... Ces informations pourraient être introduites à l'aide d'assertions rattachées, par exemple, aux variables (intervalle de variation) aux éléments modulaires (contexte sémantique),....

5-3-2 Notion de flux de données :

Considérer le flux de données, c'est s'intéresser aux échanges d'information entre les variables du programme.

De façon plus générale, c'est considérer des relations statiques entre les variables, relations représentées par un graphe de flux de données.

Un graphe de flux de données est un graphe orienté dont les noeuds sont les variables du programme et les arcs représentent les flux d'informations entre elles.

Ainsi, David A.Gustafson, considère sur un tel graphe deux types de flux ; les flux directs entre variables dans les instructions d'affectation (généralisées éventuellement) et les flux indirects par exemple entre les variables d'une

expression conditionnelle définissant le nombre d'itérations d'une boucle et les variables ayant une occurrence dans le corps de cette boucle.

```
WHILE Z < N DO
      Z<----- N
      K<----- Y
      Y<----- Y+2
      W<----- Z
END WHILE
```

MORCEAU DE PROGRAMME

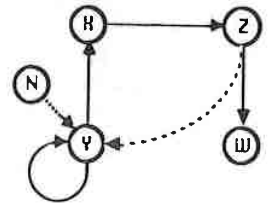


DIAGRAMME DES FLUX DE DONNEES

——— Flux direct

- - - - - - - Flux indirect

L'association à ce graphe de la matrice d'adjacence permet par fermeture transitive d'envisager tous les chemins possibles et **inimaginables** d'où la difficulté qu'il y a à manipuler de telles matrices (donc les graphes associés) sauf dans une perspective d'étude de la conservation de l'information ce qui n'est pas exactement notre propos.

Une démarche plus classique est celle qui consiste à considérer le flux de données comme résultat d'un processus de propagation de relations entre les variables du programme le long d'un graphe représentant ce programme.

Parmi les relations possibles entre variables d'un programme, on peut citer par exemple le flux direct, le flux indirect de D.A. Gustafson, la redéfinition, la hiérarchie, l'influence spéciale, l'influence arithmétique et l'influence d'affectation de M.C. Portmann.(PORTMANN M. C.,74).

On peut envisager de façon plus générale (voir le modèle conceptuel de programme dan pII ch2) des relations structurelles, contextuelles, fonctionnelles, arithmétiques ou logiques entre variables.

Quant aux techniques d'analyse du flux de données existantes, elles sont de deux types principaux :

- l'analyse **itérative** dont le principe est la propagation de l'information le long du graphe de flux, de noeud en noeud, jusqu'à stabilisation.

Ainsi pour le problème de la substitution au niveau d'une variable s'il y a une seule définition d'une variable v qui atteint le noeud p et si cette définition est une constante alors s'il y a une référence à v en p elle peut être remplacée par cette constante.

Cela nécessite l'étude de **l'atteignabilité** en un noeud p des définitions de variables et donc la résolution itérative des équations de flux de données dont il a été prouvé qu'elle était finie.

- l'analyse par "**intervalles**" :

Le graphe du flux est supposé réductible et donc décomposable à différents niveaux (graphes d'ordre n) par intervalles (F.E. Allen et J.Cocke).

L'étude de l'atteignabilité et donc la résolution des équations de flux de données sont effectuées en deux phases en travaillant successivement dans le sens croissant puis décroissant des ordres des graphes de décomposition.

5-3-3 Une autre approche de l'analyse du flux de données :

Contrairement aux méthodes précédentes, nous proposons la propagation des relations entre variables le long des graphes de contrôle, de branchement et d'appel à l'appui des constructions de base.

Les résultats de cette analyse contribuant en partie à l'enrichissement du modèle modulaire d'un ensemble de programmes (voir pl ch1).

Ce mécanisme de propagation permet alors de réaliser progressivement la définition de relations entre les variables entre deux points de programme mais offre aussi un moyen de procéder à l'assainissement et à la documentation du code concerné.

L'analyse du flux de données peut être envisagée de deux points de vue différents ; l'analyse statique qui concerne en général les **variables** est effectuée sur un programme **hors de son exécution** et l'analyse dynamique qui concerne en général les **données** est l'étude des résultats d'un programme **lors de son exécution**

- L'analyse **dynamique** introduit par exemple et avec elle l'indéterminisme, naturel hors de toute interprétation, la définition du programme en termes de schémas relationnels.

L'outil qu'est le calcul relationnel contient les divers types de schémas, permet d'étudier les preuves de correction de programmes et constitue un support formel dans la propagation de l'information.

Dans ce cadre les actions de base du programme (opérations de base du langage) sont réduites aux relations qu'elles induisent sur les variables manipulées et seules sont prises en compte les constructions fondamentales et leur interprétation relationnelle.

- L'analyse **statique** prend appui en général sur une définition du programme en terme d'ensemble de schémas (constructions) de base constituant un schéma du programme. Dans notre cas, le programme ayant été restructuré et arborisé on dispose d'un schéma propre arborescent.

Les objets concernés par cette analyse sont les **variables** ou leurs **occurrences** dans le texte concerné, objets auxquels on peut attacher des

caractéristiques comme le nom de la variable, le type de l'occurrence (utilisation, définition), le lieu de l'occurrence,...

Les objets concernés sont reliés entre eux par des relations de dépendance.

Selon le but poursuivi lors d'une analyse statique, on précisera le type de relation de dépendance utile ainsi que les caractéristiques nécessaires des objets.

Les relations de dépendance entre objets pouvant aller de la relation la plus générale qui lie tous les objets entre eux, solution calculable mais souvent trop simplificatrice, aux relations les plus fines comme la dépendance en calcul, dépendance entre contrôle, dépendance hiérarchique,.....

En facteur des exemples précis avec d'autres contextes d'utilisation de l'analyse statique (voir en annexes notre algorithme d'élimination des définitions redondantes dans un texte, et dans pl ch6 un procédé de modularisation d'un programme) et pour une **relation de dépendance générale** rappelons ici le mécanisme de calcul de cette relation pour un schéma de programme (BOUFRICHE Z.) (BELAID A.,87).

5-3-3-1 Définition de la relation de dépendance générale :

La relation de dépendance notée R, est définie comme étant un ensemble de couples (u, v) tels que u **dépend de** v.

On note dom (R) = {u / ∃ v, (u,v) R } le domaine de la relation R

Par exemple si R désigne la dépendance entre un objet modifié (u) et un objet (v) participant à la modification;

- u est membre gauche d'une affectation, v appartient au membre droit.
- u est un paramètre passé par adresse dans une fonction v alors dom (R) est l'ensemble des objets modifiés.

L'union de deux relations R1 et R2 est définie comme étant l'union des couples qui les composent et on note :

R1 U R2 = {(u,v) / (u,v) R1 ou (u,v) R2}

Le calcul de la relation de dépendance pour un schéma de programme est effectué en deux étapes.

- calcul de la relation de **dépendance directe** R pour les différents types de constructions de base à savoir,

1) l'affectation
2) la séquence
3) la conditionnelle
4) les itérations conditionnelles
5) les appels de procédures.

Pour les constructions non prévues, on peut prendre la relation générale de dépendance.

- la relation cherchée est alors la fermeture transitive de R (notre R^*) c'est à dire la plus petite relation reflexive et transitive contenant R.

a) Cas de l'affectation :
Soit α l'instruction d'affectation suivante :
$x := f(x_1, x_2,...,n_x)$; alors $R_\alpha = U_i \{(x, x_i)\}$

exemple :
$x := y + z$; $R_\alpha = \{(x,y), (x,z)\}$;

b) Cas de la séquence :
Soit S la séquence $\alpha_1 ; \alpha_2 ; \ldots ; \alpha_n$; alors $R_S = R_{\alpha 1} U R_{\alpha 2} U...U R_{\alpha n}$

c) Cas de la conditionnelle :
Soit C la conditionnelle suivante :
si p $(x_1, x_2,...,x_n)$ alors α;

On peut supposer que tout objet x dépendant de α est également dépendant des objets x_i opérandes dans l'expression p alors
$R_c = R_\alpha U \{(x,y)/x$ dom (R_α) et y $\{x_1, x_2,...,x_n\}\}$
Dans le cas général si C est la conditionnelle suivante :
si p $(x_1, x_2,..., x_n)$ alors α sinon β ; alors
$Rc = R_\alpha U R_\beta U \{(x,y) / x$ dom $(R_\alpha U R_\beta)$ et y $\{x_1, x_2,.., x_n\}\}$

exemple :
si a alors x : = x - u
sinon y : = u - v; $Rc = \{(x,a),(x,u),(y,a),(y,u),(y,v)\}$

d) Cas de l'itération conditionnelle :
Soit I l'itération conditionnelle suivante :
tant que p $(x_1, x_2,...,x_n)$ faire α ; alors
$R_I = R_\alpha U \{(x,y) /x$ dom (R_α) et y $\{x_1, x_2,.., x_n\}\}$

La relation de dépendance directe associée à un morceau de texte source M étant notée R_M la relation de dépendance D cherchée est alors la fermeture transitive de R_M soit $R [M] = R_M^*$

exemple :
x :=a * b
tant que x > y faire
si x > O alors
z : = z+x-y;
fsi
x : = x - 1;
f tant que;
RM = {(x,a),(x,b),(z,x),(z,y)}
R [M] = {(x,a),(x,b),(z,x),(z,y),(z,a),(z,b)}

e) Cas des appels non récursifs de procédures :
Soit f $(e_1, e_2, - ,e_n, r_1, r_2, \ldots r_p)$ un appel à la procédure f où les e_i (i = 1..,n) sont les expressions associées aux paramètres "données" et rj (j = 1,..,p) celles correspondant aux paramètres "résultats" alors
$R (f) = R_{appel} U R^*_f U R_{retour}$
où R^*_f désigne la fermeture transitive de la relation induite par le corps de f, R_{appel} est celle induite par l'affectation des données e_i aux paramètres d^f_i R_{retour} celle induite par l'affectation du paramètre r^f_j au résultat r_j.

La notation à double-indices correspond à la nécessité de qualifier chaque nom de variable par le bloc qui la contient. Ce mécanisme de **séparation des variables** est aussi intéressant pour différencier les utilisations d'une même variable globale à différentes fins dans un programme sans structure de bloc, ou dans un bloc et peut être utile pour la modularisation d'un programme.

Ce mécanisme considère pour une variable X donnée l'ensemble $\mathbb{D} = \{d_1, d_2,..., d_n\}$ des définitions de X et l'ensemble $\mathbb{U} = \{u_1, u_2,..., u_p\}$ des utilisations de X.

L'étude pour chaque d_i de son ensemble **d'atteignabilité**, atteint (d_i), permet de rechercher une partition de $\mathbb{D}$ en sous-ensembles $\mathbb{D}1, \mathbb{D}2,.., \mathbb{D}n$ tels que les ensembles U1, U2,..,Un correspondants forment une partition de U. Si une telle partition existe on peut séparer la variable X.

Remarques :

1) Le processus de modularisation avec définition des interfaces des modules confirmera à quel point la relation de dépendance envisagée ci-dessus est "grossière", mais elle permet la présentation de façon générale d'un mode de calcul.

2) Le mécanisme de définition de relation de dépendance est réalisé par B.Maher et D.H.Sleeman sous forme de calcul sur des matrices de correspondance et des matrices complètes par fermeture transitive, dans le but d'assainir du texte source.

L'analyse du flux de données et donc la prise en considération de relations entre les variables d'un programme permettent d'effectuer des transformations de ce programme.

Ces transformations peuvent être des simplifications et/ou des optimisations.

On distingue souvent les **optimisations de bas niveau** (propagation de constante, élimination de sous-expressions communes, extraction d'instructions invariantes d'une boucle,...) des **optimisations de haut niveau** concernant globalement les structures de contrôle (fusion ou éclatement de boucles, intéraction entre boucles et instructions conditionnelles,...) , optimisations d'autant plus faciles à mettre en oeuvre sur une structure arborescente du programme.

On doit admettre que de n'envisager les transformations d'un programme que d'un seul point de vue, celui d'un graphe ou celui du langage, ne peut que présenter des inconvénients, ainsi,

- une transformation unique au niveau d'un graphe peut prendre de nombreuses formes dans le langage (duplication de noeuds par exemple).

- de nombreuses transformations au niveau du langage n'ont pas de conséquence sur le graphe (suppression d'un GO TO, différentes linéarisations d'un circuit,...)

Aussi tout environnement de transformation doit, pour être efficace, permettre de manipuler simultanément un graphe et le texte source du programme dans un processus nécessairement conversationnel permettant à l'utilisateur le choix à chaque instant du support de sa transformation et la vérification sur la représentation complémentaire de l'effet de cette transformation ; cette configuration pouvant être soutenue par un optimiseur proposant d'une part pour les transformations les plus classiques le mode de représentation du programme le plus adéquat et d'autre part pour une représentation donnée d'un morceau de programme la liste des transformations envisageables.

5-3-4 Simplification ou assainissement du programme

La simplification d'un programme peut consister en la prise en compte, essentiellement localement, d'un petit nombre d'instructions de ce programme . Parmi les transformations les plus courantes dans la littérature à ce niveau on peut citer :

- l'élimination des impuretés au niveau d'une instruction
  - expressions non réduites

  x : = x + y - y ---> x : = x ---> vide
  - ambiguïté d'un opérande utilisé à différentes fins

  x : = y * y ; x : = z + 1 + x

  ---> u : = y * y ; x : = z + 1 + u
  - sous-expressions communes

  x : = (y + z) * (y + z) ---> u = (y + z) ; x : = u * u
  - simplifications ou explosions liées aux propriétés de certaines fonctions

  $x : = y^2 + 2 * y * z + z^2$ ---> $x : = (y + z)^2$

  $x : = y + 1$ ---> $x : = y + \sin^2 x + \cos^2 x$
  - calcul de constantes

  x = 2 + 1 ---> x : = 3
  - fausse conditionnelle

  si vrai alors α sinon β fsi ; ---> α

  si C alors α sinon β fsi ;---> α
  - réordonnancement d'une conditionnelle

  si non p alors sinon α fsi ;

  ---> si p alors α fsi;
  - boucle vide

  tantque c faire ftantque; ---> vide

Ces transformations doivent être manipulées avec précaution ; d'une part, certaines règles sont symétriques et donc ne doivent pas être appliquées deux fois à la même instruction, d'autre part ces transformations dépendent souvent de la conception pratique qu'à l'utilisateur dans son contexte de l'identité de deux instructions ou expressions.

- suppression d'instructions
  - branchements inutiles

    GO TO E;

    E :
  - instructions inaccessibles (code mort)
  - affectations redondantes ou inutiles (voir en annexe n un algorithme d'élimination de définitions inutiles).
- composition des instructions (fusion de deux affectations)

Outre la composition de fonctions, il peut être intéressant d'introduire une variable intermédiaire pour éviter d'avoir à effectuer plusieurs fois le même calcul ; à l'inverse pour aller vers une certaine normalisation, on peut remplacer chaque occurrence d'une variable par l'expression correspondant à son mode de calcul.

- permutation de deux instructions en séquence impérative

Deux instructions i_j, i_k sont en séquence impérative si

- i_j a un seul successeur immédiat i_k.
- i_k a un seul prédécesseur immédiat i_j

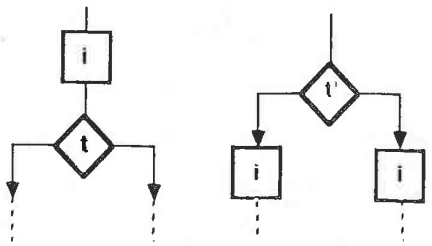
Rappelons que $\mathbb{D}(i_n)$ (respt $\mathbb{U}(i_n)$) est l'ensemble des variables définies par (respt utilisées dans) l'instruction i_n.

Les deux instructions i_j et i_k sont permutables si :

1) i_k a un seul successeur immédiat
2) $\forall x \quad D(i_j), x \quad D(i_k)$ et $x \quad U(i_k)$
3) $\forall y \quad D(i_k), y \quad D(i_j)$ et $x \quad U(i_j)$

Il y a alors conservation, après permutation, de l'état des variables à la sortie des deux instructions (A. ADAM)

cas particulier : la traversée d'un test (absorption gauche)



Où t' est identique à t si i et t sont permutables sinon t' est obtenu par composition si la variable définie dans i est utilisée dans t.

- recopie d'instructions ou fusion de deux occurences d'une même instruction.

Opérations toujours possibles avec quelques précautions en recopie pour éviter les arbres infinis.

La recopie et la fusion sont très utilisées pour faire apparaître comme composants d'un programme uniquement des schémas de base, de plus elles ne modifient pas l'histoire des calculs.

- Sélections inutiles ou redondantes

Ainsi [t = vrai] si t alors f sinon g fsi;
équivaut sémantiquement à f.
[t = faux] si t alors f sinon g fsi;
équivaut sémantiquement à g.
Ainsi si t alors si t alors f sinon g fsi sinon h fsi;
équivaut sémantiquement à si t alors f sinon h fsi;
Cas particulier, celui de la tautologie.
Ainsi si t alors f sinon f fsi;
équivaut sémantiquement à f.

5-3-5 Optimisation du programme.

Une optimisation peut avoir comme objectif, soit un gain de temps, soit un gain de place, soit un gain en qualité de structuration ; les transformations mises en oeuvre sont plus globales que lors des simplifications et parmi les plus courantes de ces transformations on peut citer :

- sortie des invariants d'une boucle.

Cette extraction peut concerner des variables et des instructions.

Les variables intervenant dans le contrôle de l'exécution de la boucle sont considérées comme "variantes".

Une instruction est un invariant de boucle si toutes les variables ayant une occurrence dans cette instruction sont invariantes.

Il existe différents processus de localisation des invariants de boucle liés aux techniques d'analyse du flux de données (analyse itérative, technique des intervalles, matrices de correspondance complètes et leur fermeture transitive,..)

La localisation d'une instruction invariante est une chose, son extraction en est une autre et nécessite de tenir compte de la sémantique du programme. Il y a des cas où l'on ne peut sortir l'instruction invariante :

1) Si le corps de la boucle peut ne pas être exécuté.
2) Si la variable définie dans l'instruction invariante est référencée dans le corps avant sa définition.
3) Si l'instruction invariante est dans une branche d'alternative dont la condition utilise des variables variantes.

Remarque :
les précautions précédentes à prendre lors de l'extraction d'invariants d'une boucle sont de même nature que celles qui concernent l'élimination d'affectations inutiles dans une boucle (voir l'algorithme en annexe).

- l'inversion propre pour une boucle.
Elle consiste à remplacer une séquence s d'instructions en tête du corps de la boucle par une copie de s devant la boucle et une copie de s en queue du corps de la boucle.
Aussi FAIRE s ; f BOUCLER est équivalent à
s ; FAIRE f ; s BOUCLER si et seulement si s est propre (J. ARSAC).
Une application classique de cette inversion est la notion de lecture initiale (ou d'avance) d'un article d'un fichier avant l'itération qui traite chaque article et la lecture de l'article suivant en fin de module itéré.

- fusion de boucles consécutives, séparation en boucles consécutives.
Il s'agit en fait de constituer le corps d'une boucle par concaténation du corps de deux boucles consécutives ou au contraire d'éclater le corps d'une boucle en les corps de deux boucles consécutives.
En dehors du problème des effets de bord (par exemple le problème des variables indicées) la condition essentielle pour réaliser ces opérations étant qu'aucune des variables définies dans un des deux corps en présence ne soit utilisée dans l'autre ou dans la condition d'itération.

- permutation de boucles consécutives
Une permutation de deux boucles B_i et B_j est possible si cette permutation respecte toutes les relations de précédence suivantes (A. ADAM);

1 - $\mathbb{D}(B_i) \quad \mathbb{D}(B_j) \neq \emptyset$
2 - $\mathbb{D}(B_i) \quad \mathbb{U}(B_j) \neq \emptyset$
3 - $\mathbb{U}(B_i) \quad \mathbb{D}'B_j) \neq \emptyset$
4 - $\exists B_k$ tel que B_i précède B_k et B_k précède B_j

5-4 CONCLUSION

Dans ce chapitre, nous avons envisagé la fonction ARBORISER qui permet de passer d'un graphe (de flux dans notre contexte) à une arborescence dont les noeuds correspondent à des schémas de base syntaxiques.
L'arborescence "syntaxique" ainsi construite bénéficie des bonnes propriétés des arborescences, et sert de support pour la construction progressive de relations entre les objets du programme correspondant.
L'application de la fonction ARBORISER exige du graphe de départ certaines propriétés accessibles grâce à la fonction RESTRUCTURER et permet par exemple la mise en oeuvre simple de la fonction MODULARISER et de la fonction LINEARISER.
Une représentation arborescente d'un programme est également un excellent support pour la documentation de ce dernier grâce à la décoration possible (attributs) des noeuds et à la propagation de l'information documentaire le long des branches (héritage et synthèse).

I 6 MODULARISATION DU PROGRAMME : MODULARISER

En pratique, cette modularisation peut être recherchée dans l'ensemble des éléments internes et/ou dans chaque élément externe.

Modulariser un texte source, c'est trouver une décomposition en morceaux (modules) de celui-ci la plus proche possible de celle qui aurait pu résulter de l'application d'une bonne méthode de conception lors de la prise en charge du problème à résoudre. L'aspect à priori peu modulaire des programmes étudiés (structure modulaire non apparente) peut résulter de l'absence de mise en oeuvre d'une méthode de conception structurée (raffinements successifs, décomposition fonctionnelle, machines abstraites...) de l'absence dans le langage de programmation de concepts propres à exprimer la notion de module, d'une utilisation incorrecte des possibilités du langage lors de la linéarisation ou codification de l'algorithme.

Quant on songe à la dualité données-traitements que constitue un programme, on peut considérer le concept de modularité par la description respectivement des traitements et des informations.

Après quelques rappels sur les concepts de **module** et de **type**, nous envisageons quelques mécanismes de modularisation classiques et proposons notre algorithme de décomposition du texte d'un programme à l'aide de sous-ensembles de variables à dépendance limitée.

6-1 MODULARITE DES TRAITEMENTS : MODULES.

La finalité de la modularisation (diviser pour régner) est essentiellement de réduire la complexité d'un programme pour en faciliter la compréhension, la maintenance et la réutilisation mais en pratique le but n'est pas toujours atteint.

Liskov (1-MYERS p10 9) a mis en évidence trois raisons essentielles de cet échec :

- Les modules regroupant trop de fonctions différentes même s'il existe un lien quelconque entre elles.
- L'apparition d'une même fonction dans différents modules.
- Les modules inter-agissant sur des données cachées ou communes de manière inattendue.

Une démarche de modularisation de traitements peut avoir pour buts de créer un module par fonction (module de haute cohésion) et de minimiser les relations entre modules par l'utilisation de passage de paramètres formels (couplage minimum entre modules).

Mais si la cohésion fonctionnelle est souhaitable , on dénombre en réalité sept types de cohésion souvent regroupés en trois classes :

1 - cohésion fonctionnelle, séquentielle, communicationnelle
2 - cohésion procédurale, temporelle.
3 - cohésion logique, coïncidentale

La classe 1 correspond à des modules dont les actions contribuent à l'élaboration d'une fonction particulière.

La classe 2 correspond à des modules dont les actions sont à exécuter en séquence.

La classe 3 correspond à des modules constitués de sous-ensembles d'actions sélectionnables à l'aide d'un paramètre passés à l'évocation du module.

Une mesure de cohésion utilisant un graphe de flux réduit du module et pour chaque variable v_i de ce module, l'ensemble des noeuds du graphe contenant une référence à v_i permet de situer ce module dans l'une des trois classes (Thomas J. Emerson) (DMMC).

Quant aux couplages entre modules, on en dénombre six;

- le couplage par le **contenu**

  C'est le cas avec un langage de haut niveau de deux modules d'une même unité compilable se partageant leur contenu ("procédures PERFORMEES" en COBOL).

- le couplage par une zone **commune** (ou plus simplement par les **externes**).

C'est le cas d'un ensemble de modules PL/1 referençant une structure de données déclarées EXTERNAL, ou du COMMON en FORTRAN, de la COMMON-STORAGE SECTION en COBOL.

- le couplage par le **contrôle**

C'est le cas d'un module pouvant déterminer un chemin d'exécution dans l'autre (arguments de type éléments de contrôle ou noms de fonction).

- le couplage par les paramètres

En distinguant les paramètres globaux (couplage **par tampons**) (par exemple tout un enregistrement d'un fichier) de celui des paramètres spécifiques (couplage **par les données**) où l'on ne transmet que les seules données utiles au module.

Le concept de module toujours au niveau du programme peut être l'objet d'autres approches, parmi lesquelles :

- la taille du module

C'est une condition nécessaire que la taille d'un module soit petite mais non suffisante pour une bonne modularité ou une simplicité correcte

- la complexité d'un module

La complexité cyclomatique (Thomas J.Mc Cabe) d'un module semble être une bonne mesure de ce qui correspond intuitivement au niveau de difficulté de ce module sauf dans certains cas particuliers (instruction CAS très longue). De plus, elle est très facile à calculer ; ainsi pour un programme structuré elle est égale au nombre de test plus un, pour un ensemble de modules constituant un programme elle est la somme des complexités des modules.

- module à fonctionnalité prévisible

Sans modification de son état d'une exécution à l'autre donc indépendant de l'environnement.

- module en temps que procédure interne

En général, type de module à éviter car non appelable en dehors du module le contenant (il faut le recopier) et difficile à isoler pour le tester

- minimisation des données accédées

Réduire l'ensemble des données auxquelles un module peut accéder, c'est déjà s'éloigner des concepts des variables globales, des COMMON, des EXTERNAL, des tampons.

- masquage des informations

Le module accède à des données non rendues accessibles à l'extérieur (cachées).

- minimisation de l'interface
- structure de décision

Il est souhaitable que la hiérarchie entre modules reflète la structure de décision du programme (éviter les paramètres,elément de décision)

On peut citer dans quelques langages la prise en compte du concept de module :

- CIVA : (DERNIAME 74)

un module est une procédure compilée séparément, un métamodule permet par affectation d'arguments aux métavariables la génération d'un module

- MODULA : (WIRTH 76)

Les modules de MODULA permettent la description de processus parallèles à la manière des moniteurs de (HOARE 74) avec exécution en exclusion mutuelle.

- CLU : (LISKOV 74)

L' "abstraction fonctionnelle" n'est autre q'une procédure compilable séparément (module à un seul point d'entrée).

- PASCAL :

La procédure de PASCAL

- ADA :

La notion de module est caractérisée par le package et le sous-programme.

- ATM :

L'unité machine implantable grâce à l'unité module.

6-2 MODULARITE DES DONNEES : TYPES.

Une unité modulaire de description d'informations est une suite de déclarations d'objets qui peuvent être créés, consultés et/ou modifiés depuis l'extérieur. Ainsi deux modules de traitement peuvent se partager une unité modulaire décrivant des informations et donc ne pas utiliser le mécanisme du passage de paramètres.

Ainsi en est-il des "classes" du langage CIVA et de la possibilité des "modules d' "utiliser" la même classe.

Ainsi en LIS (ICHBIAH 76) les DATA SEGMENT et DATA PARTITION permettent le partage d'objets et de descriptions d'objets (définitions de types).

6-2-1 Notion de type

"Un type peut être considéré comme un ensemble d'objets distincts représentables, et ayant en commun un certain nombre de propriétés sémantiques" (BOUSSARD 77).

"Par type, nous entendons une collections d'objets (les instances de ce type) possèdant certaines caractéristiques, et dont le nom constitue le nom du type (KOSTER 77)."

Cette notion est celle utilisée dans de nombreux langages de programmation par l'intermédiaire des déclarations de variables.

L'utilisateur dispose ainsi de types prédéfinis (INTEGER, REAL, BOOLEAN,...) et peut élaborer des types construits en général par composition de types prédéfinis à l'aide de structures disponibles dans les langages (tableau, structure, ensemble,...)

Les avantages et inconvénients des types construits sont liés aux faits suivants :

- les objets d'un type ont plus en commun la même représentation que les mêmes propriétés sémantiques.
- on ne peut modifier ni la sémantique des opérations, ni le nombre d'opérations d'un type.
- les objets sont réalisés immédiatement (en mémoire).

6-2-2 Notion de type abstrait.

"On décrit un type d'objets en précisant les propriétés des fonctions associées" (J.P. FINANCE 79)

La notion de type abstrait est apparue historiquement (LIS,75), (WUL,75) comme l'analogue au niveau des structures de

données de la notion de module au niveau des structures de contrôle (DERNIAME 74)

L'introduction des types abstraits au niveau des langages de programmation présente des avantages :

- la distinction entre structure logique et structure physique d'un objet permet à l'utilisateur de ne pas avoir à manipuler la représentation des objets.
- la définition de mécanismes de protection et de confidentialité interdisant certains accès à certains utilisateurs (DER,79) thèse FINANCE.
- la conservation et le partage d'objets entre utilisateurs deviennent possible.

On retrouve cette notion de type abstrait dans les langages algorithmiques récents comme ADA, ATM,.... et dans des extensions de langages plus classiques comme PASCAL (K.PROCH J.C. DERNIAME), PL/1 (M.V. ZELKOWITZ).

6-3 LE PROCESSUS DE MODULARISATION :

Ce processus doit conduire progressivement à la mise en évidence de modules et donc à leur spécification.

6-3-1 Les spécifications d'un module :

On peut distinguer les spécifications abstraites liées aux fonctionnalités et permettant une définition et une utilisation correctes du module, des spécifications concrètes concernant la définition et le fonctionnement des opérations concrètes réalisant les fonctions (opérations abstraites).

6-3-1-1 Spécifications abstraites :

Au niveau abstrait, le module est caractérisé par sa partie visible c'est-à-dire son interface, il est alors utile de préciser le processus d'appel des opérations (spécifications syntaxiques) et l'effet "visible" des fonctions (spécifications sémantiques).

6-3-1-1-1 Spécifications syntaxiques : profil

Dans les langages récents, la distinction est faite entre les paramètres formels utilisés comme données dans une procédure, les paramètres utilisés comme résultats et les paramètres "modifiables" (à la fois données et résultats).

Cette distinction, sans expliquer clairement le fonctionnement de l'opération aux utilisateurs, a le mérite d'éviter des confusions à l'utilisation de l'opération et permet au compilateur de contrôler que les arguments donnés à l'appel sont bien conformes aux spécifications.

Les spécifications syntaxiques d'une opération devraient fournir :

- le nom de l'opération
- le mécanisme d'appel de l'opération (procédure, fonction)
- la nature de l'opération (consultation, modification,...)
- le type des paramètres
- la nature des paramètres (donnée, résultat, modifiable,...)

6-3-1-1-2 Spécifications sémantiques : description formelle

Le profil de l'utilisateur et la nécessité de vérifier la cohérence (consistance) et la fidélité (adéquation de la spécification au problème) peuvent conduire à des spécifications sémantiques allant d'un ensemble structuré ou non de commentaires (optionnels ou obligatoires avec mots clés) HELOÏSE : ROUSSEAU 77) (MINOT) à un système de description formelle dans un langage de spécification (utilisation de prédicats pré/post exécution de l'opération) général (projet CIP, MEDEE, PROLOG, ATM,...) ou spécifique d'un domaine particulier (langages liés aux méthodes d'analyse en informatique de gestion, ARIANE, PROTEE, MERISE, ... ou aux bases de données, SOCRATE, SQL,...)

6-3-1-2 Spécifications concrètes :

La spécification à ce niveau concerne l'implantation des modules, la réalisation des objets, et correspond à la description en termes du langage de programmation (structures de contrôle, structures de données) des opérations composantes et des représentations d'objets.

6-3-2 Quelques mécanismes de modularisation :

La donnée est un programme ou un morceau de programme, soit un ensemble d'éléments internes, soit un élément externe. Ce morceau de programme, appréhendé, peut déjà avoir été restructuré et aussi arborisé.

Les mécanismes de modularisation sont liés aux représentations disponibles du morceau de programme et aux moyens mis en oeuvre.

6-3-2-1 Décomposition "fonctionnelle" :

L'approche ici consiste plus à comprendre et à vérifier le programme par la mise en évidence de fonctions composantes comme dans V.R. Basili et H.D. Mills.

6-3-2-1-1 Concepts et techniques utilisés

- Diagramme de flux (flowchart)

C'est un graphe de flux orienté dont les noeuds représentent les tests et actions sur les données, les arcs symbolisant le flux de contrôle. Pour le diagramme de flux d'un programme, il existe un arc d'entrée (dont l'extrémité est le noeud tête du programme) et un arc de sortie (dont l'origine est le noeud queue du programme)

- Affectation généralisée

Toute fonction transformant des données en de nouvelles données peut être considérée comme une forme généralisée de l'instruction d'affectation.

Les formes les plus courantes de l'affectation généralisée étant :

- l'affectation classique où une seule variable est modifiée ;

x : = e où e est une expression pouvant contenir des fonctions, alors l'instruction d'affectation définit une fonction mais en général ne sont pas pris en compte les effets de bord dans les définitions d'affectations et de tests.

- l'affectation parallèle notée,

x1, x2,...xn : = e1, e2,..,en où simultanèment la valeur de ei est affectée à xi.

- l'affectation conditionnelle notée,

(p1--->A1/p2--->A2/...../pn--->An) où $\forall_i$ p_i est un prédicat et A_i une affectation généralisée.

Affectation dont la définition est la première A_j telle que p_j est vrai, sinon indéfinie.

- Fonction d'un programme propre

La fonction d'un programme propre P est la fonction qui correspond à toutes les exécutions possibles de P commençant avec son point d'entrée et se terminant par son point de sortie.

On étend cette definition de fonction aux sous-programmes et aux morceaux propres du programme et on dit que deux programmes sont équivalents si leurs fonctions de programme sont identiques.

Un morceau de programme est dit premier (le schéma associé est dit schéma premier) s'il est propre et s'il ne contient pas de sous-programme propre sauf lui-même et des noeuds fonction.

6-3-2-1-2 La méthode utilisée :

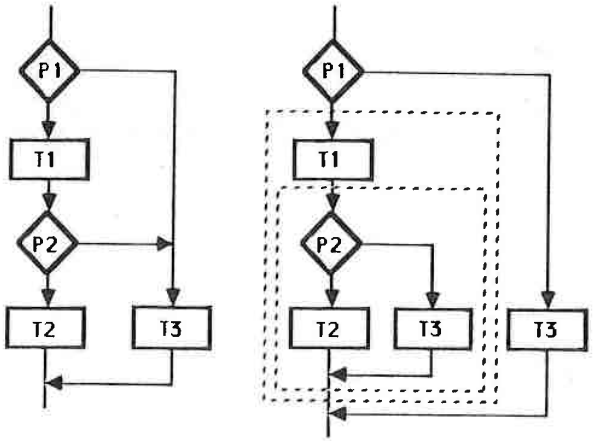
Il s'agit essentiellement d'utiliser le schéma du programme analysé avec comme support le diagramme de flux et la table des références croisées pour toutes les variables avec les étapes suivantes :

• a) - décomposition du programme en morceaux premiers

Cette décomposition consiste en la transformation du diagramme de flux en un diagramme réductible au sens de "l'absorption" des schémas premiers.

Ces transformations "manuelles" utilisent essentiellement la recopie de noeuds du diagramme pour faire apparaître des schémas premiers à un seul noeud prédicat.

exemple :



VERSION INITIALE VERSION FINALE

• b) - élaboration de la table de références des variables.

Cette table donne pour chaque variable la liste des instructions où elle est définie et la liste des instructions où elle est utilisée.

Cette table après quelques réflexions, à en croire les utilisateurs, permet le découpage fonctionnel du programme. En réalité, l'étude du comportement des variables permet de confirmer ou d'infirmer le choix d'un morceau premier comme fonction. Chaque morceau premier ainsi choisi calcule les valeurs d'une fonction. Les entrées (arguments) de cette fonction sont définies par les valeurs initiales de tous les identificateurs qui sont des entrées (arguments) pour les instructions constituant le morceau premier. Les sorties (valeurs) de cette fonction sont définies par les valeurs finales de tous les identificateurs qui sont des sorties (valeurs) pour les instructions constituant ce morceau de programme.

• c) - élaboration d'abstractions fonctionnelles équivalentes

A chaque morceau premier, il s'agit alors d'élaborer une spécification fonctionnelle exprimée à l'aide des affectations généralisées.

exemple :

```
1 IF p  =   1  THEN write ('message')FI;
2 a :   =   ax
3 b :   =   bx
4 fa :  =   f(a)
5 fb :  =   f(b)
6 c :   =   a
7 d :   =   b-a
8 e :   =   d
morceau premier
```

Ce morceau premier est décomposé en une instruction d'écriture et une affectation multiple ; la fonction équivalente est alors f.1-8 = f.1-1 ; f.2-8

où f.1-1 = IF p = 1 THEN write ('message') FI;

f.2-8 = a,b,c,d,e,fa,fb : = ax,bx,ax,ax-bx,ax-bx,f(a), f(b)

Dans le cas d'itérations, il s'agit essentiellement de l'étude des invariants de boucle.

En conclusion, ce mécanisme exigeant sur le plan logique et formel n'est intéressant que pour des programmes de type scientifiques, de plus il exige un gros investissement homme (démarche bottom-up au sens de chaque morceau premier) ; plusieurs semaines pour un programme en FORTRAN de 100 lignes (V.R. Basili et H.D. Mills) UDP. Aussi ne le retiendrons nous pas..

6-3-2-2 Décomposition "structurelle" :

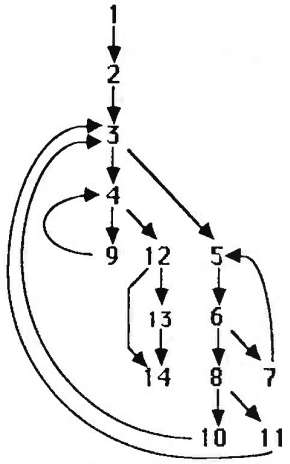
Le mécanisme consiste à s'appuyer d'une part sur la structure actuelle du programme ou morceau de programme en termes d'éléments internes (paragraphes, blocs étiquetés,...) et d'autre part sur l'expression de cette structure sous la forme d'un DFST de son graphe d'enchaînement :

6-3-2-2-1 Elaboration d'un DFST du graphe d'enchaînement

Processus déjà considéré lors de la restructuration et de l'arborisation d'un programme.

6-3-2-2-2 constitution de modules provisoires

Chaque élément modulaire externe est un module provisoire. Dans la représentation graphique du DFST (voir l'exemple traité dans le châpitre suivant) on peut considérer que chaque arc 'vertical' représente une dépendance forte entre les noeuds origine et extrémité.



EXEMPLE DE DFST

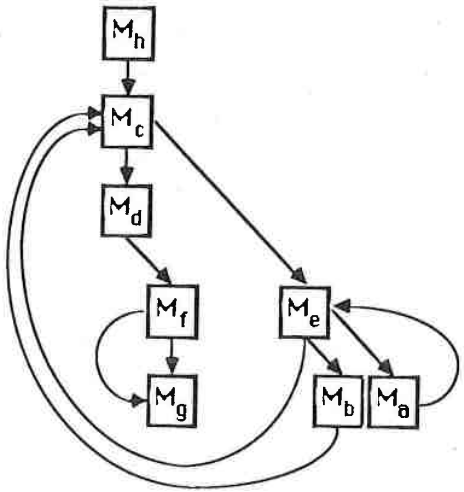
Par contre, chaque noeud non relié verticalement à au moins un autre noeud peut constituer un module provisoire ; ainsi en est-il pour les noeuds 7 et 11.

Chaque noeud atteint par au moins un arc autre que vertical peut constituer un 'début' de module ; c'est le cas des noeuds 3 , 5 , 4 , 12 , 14 ; on particularise ainsi les points de branchement ; on ajoute le noeud d'un point de branchement initial.

La décomposition en modules provisoires liée à l'exemple de DFST ci-dessus serait alors :

Ma = (7)	Md = (4,9)	Mg = (14)
Mb = (11)	Me = (5,6,8,10)	Mh = (1,2)
Mc = (3)	Mf = (12, 13)	

La représentation modulaire provisoire du programme serait alors la suivante :



6-3-2-2-3 Constitution des modules définitifs.

Ayant atteint la limite d'exploitation de la syntaxe et du graphe d'enchaînement (ou de contrôle), il devient nécessaire de faire appel aux informations sémantiques plus précises liées au problème au programme, et aux instructions.

Les préoccupations à ce niveau sont celles que l'on rencontre aussi lors de l'étape de linéarisation.

Il faut considérer en particulier la signification des arcs, l'homogénéité des modules provisoires et procéder à une démarche bottom-up.

Ainsi dans l'exemple ci-dessus :

- seul le module M_f passe le contrôle au module M_g, on peut donc faire de M_f et M_g un seul module M_{fg} qui correspond probablement à des opérations terminales du programme ; on peut de même envisager la création de M_{dfg} à partir de M_d et M_{fg}, mais seule la connaissance de leur contenu permet la prise de décision.
- les modules M_e et M_b ont la même origine, M_e, et la même extrémité, M_c ; l'étude de leurs contenus permettrait de savoir s'il existe des traitements "symétriques" et de décider alors soit la création d'un module M_{eb} soit la décomposition de M_e et la mise en évidence d'un module $M_i = (10)$ de même poids que $M_b = (11)$
- quant à M_a sa situation avec branchement depuis M_e et retour à M_e peut correspondre à une situation précise dans la logique du problème et aussi rester un module ou dans le cas contraire être réintégré à M_e.

En conclusion, cette démarche moins formelle surtout au niveau de la constitution des modules définitifs est intéressante pour des programmes de type gestion ; elle nécessite une approche sémantique détaillée, mais sur des portions du programme correspondant aux sous-graphes d'enchaînement successivement considérés.

Remarque : une démarche semblable consiste à exploiter l'arbre généralisé et fait l'objet d'un exemple dans le chapitre suivant.

6-3-2-3 Décomposition par les données (variables).

Une autre démarche est celle qui consiste à utiliser le flux de données, soit pour décomposer un programme monolithique, soit pour confirmer ou infirmer un découpage modulaire existant ou réalisé par un autre moyen : il y a deux approches possibles, la première a comme origine une certaine littérature (D.A. Gustalson,CFDFDI), la seconde est notre approche.

6-3-2-3-1 Les sous-ensembles de variables indépendants

Dans ce cas, la représentation du programme ou morceau de programme est le diagramme de flux de données, symbolisé par le graphe orienté de flux de données associé, traduction des relations statiques entre les variables. Ce graphe est indépendant de l'ordre d'exécution des instructions et représente le flux de données avec une seule exécution de chaque instruction, boucle ou bloc.

Un exemple de diagramme de flux a déjà été donné figure X

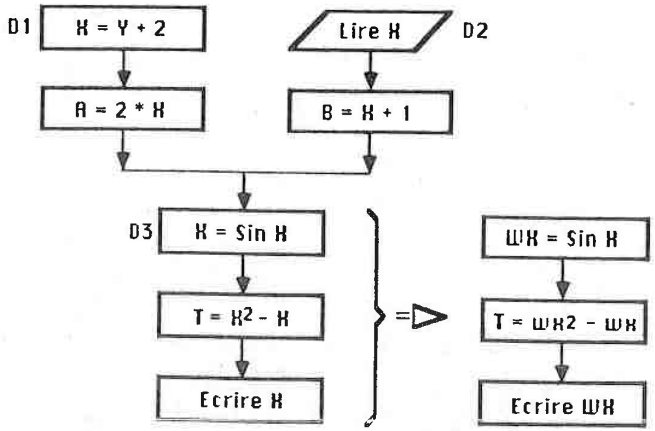
Les relations entre variables étant essentiellement non symétriques et transitives, l'élaboration de la matrice d'adjacence associée au graphe et le calcul de sa fermeture transitive permettent de déterminer le partitionnement de l'ensemble des variables en sous-ensembles indépendants.

Ce partitionnement est utile, par exemple, au niveau des processus de test des programmes où l'approche combinatoire des cas possibles est simplifiée si l'on a analysé les interactions entre variables.

Dans le but ici de modulariser un programme, cela permet d'isoler des morceaux de programme indépendants du point de vue des variables utilisés.

Mais si certains programmes volumineux "implicitement" multiprocédures ou multifonctions c'est à dire conçus comme tels mais réalisés sans l'utilisation en clair de ces concepts du fait du programmeur ou du fait du langage, peuvent être ainsi décomposés il en va rarement de même avec la plupart des programmes où l'on trouve souvent une répartition assez homogène des occurrences de chaque variable. Pour compenser au moins l'abus, de la part du programmeur, de réutilisation d'une même variable à différentes fins (économie de place mémoire) on peut avant de partitionner l'ensemble des variables procéder à la séparation des variables (voir 3-3-2) évitant ainsi à une même variable de jouer des rôles différents selon les occurrences rencontrées dans différentes parties du programme.

Exemple :



Ce processus s'accomode fort mal des variables indicées, car une définition de variable indicée n'est pas nécessairement une nouvelle définition du tableau associé.

En conclusion, cette démarche qui permet d'isoler des morceaux de programmes très indépendants (modules sans paramètre ou dont toutes les variables sont paramètres) peut être utile pour décomposer de très volumineux programmes.

6-3-2-3-2 Les sous-ensembles de variables à dépendance limitée.

L'objectif est alors de faire de certains morceaux de texte un module (procédure ou fonction) sans avoir à mettre en paramètre toutes les variables ayant une occurrence dans ce texte et de distinguer les paramètres données et résultats.

Ceci impose de localiser les variables intermédiaires et permet d'en faire des variables locales au module.

1- Le contexte et les hypothèses :

Un pseudo langage L défini par ses grammaires concrètes et abstraites (voir annexe A1) sert d'illustration à notre mécanisme de modularisation.

Ce langage possède la notion de variable et les instructions et structures de contrôle de base classiques.

La donnée de notre problème est un programme P (ou composant de programme) supposé correct et assaini, (élimination des redondances, extraction des invariants de boucles,etc...) écrit en L et même plus précisément un morceau M de P. Ce morceau M peut être constitué d'un ensemble de blocs ou d'une partie d'un bloc, production terminale d'un non terminal de la grammaire. M peut être un élément modulaire candidat au statut de module.

Etant donné un morceau M de P, on lui associe :

- l'ensemble V de toutes les variables ayant une occurrence au moins dans M
- l'ensemble VE des variables de V visibles de l'extérieur de M, c'est à dire ayant une occurrence au moins de type utilisation à l'extérieur de M.

Le problème consiste alors à déterminer dans V deux sous ensembles non nécessairement disjoints qui seront appelés respectivement les données et les résultats de M, en évitant de faire un paramètre de toute variable ayant une occurrence dans M.

On associe ainsi à M une proposition de module soumise à l'appréciation de l'utilisateur.

Ce module a comme interface la réunion des données et résultats.

Les paramètres sont déterminés quant à leur nature, pour ce qui est de leur type c'est celui des "variables" correspondantes.

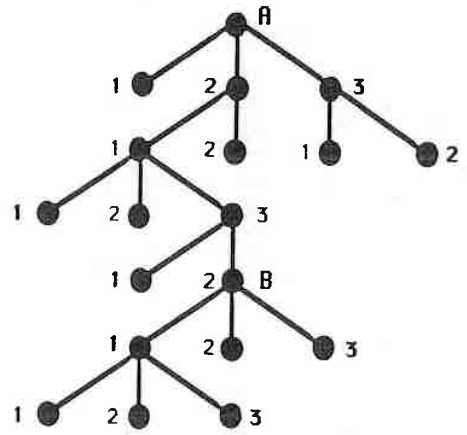
Si ce morceau M est accepté comme module, le rôle de l'utilisateur est alors de spécifier sémantiquement ce module soit sous la forme d'axiomes (ALPHARD (WULF, 75)), soit sous la forme de commentaires structurés (HELOÏSE (ROUSSEAU))

Notre problème consiste donc essentiellement à mettre en évidence les variables intermédiaires.

Ainsi dans le morceau de programme symbolisé par la séquence ;
y : = x + 1 ;
z : = y + 2
si VE = {x, z} alors on peut éliminer y des paramètres du module associé, et dans ce cas :
x est un paramètre donnée
z un paramètre résultat;
si VE = {x, y, z} alors y doit figurer comme paramètre et dans ce cas x est une donnée, y et z des résultats.

Remarque :
Les ambiguïtés pouvant exister au niveau de la définition de la dépendance entre deux variables, constituant un grand nombre de possibilités, sont difficilement formalisables.
Aussi l'assainissement du programme (élimination des impuretés, des redondances ou inutilités) doit il être effectué au préalable de façon intéractive.
Ainsi face à l'instruction suivante, x : = x + k + y - y, l'utilisateur peut seul décider de la dépendance de x vis à vis de y.
En réalité, le concept manipulé est celui **d'occurrence** d'une variable dans un texte et si l'on considère le programme non pas comme une liste mais du point de vue de son arbre abstrait une occurrence d'une variable n'est autre que l'unique chemin dans l'arbre qui va d'un noeud étiqueté par le nom de la variable jusqu'à la racine.

Notations et définitions :
Etant donné un programme P , on note G (P) l'arbre de représentation interne de P (arbre abstrait de P).
Un sous-arbre B d'un arbre A peut être désigné par le chemin qui mène de la racine de A à la racine de B : une façon de désigner ce chemin consiste à numéroter les fils d'un noeud (cf-dessin).



B est désigné par le chemin (2, 1, 3, 2,)

Si α désigne un nœud de l'arbre G (P) d'un programme P ou, ce qui revient au même, le sous-arbre de G (P) situé sous ce noeud on note :

- 𝔻 [α] l'ensemble des variables ayant au moins une occurrence de type, définition dans α
- 𝕌 [α] l'ensemble des variables ayant au moins une occurrence de type utilisation dans α

Etant donné un morceau de texte t on note :
- D [t] l'ensemble des variables ayant au moins une occurrence dans t considérées comme données du module (partiel) associé à t
- R [t] l'ensemble des variables ayant au moins une occurrence dans t considérées comme résultats du module (partiel) associé à t

Soit ℝ la relation associée à un morceau de texte t telle que *domaine* (ℝ) = D [t] et *image* (ℝ) = R [t]

2 - Construction récursive de ℝ.
Nous envisagerons successivement la construction de ℝ donc de R et de D pour les types d'instructions de base que sont ;
- l'affectation
- l'alternative
- l'itération conditionnelle
- l'appel de procédure

La relation recherchée est alors la fermeture transitive de ℝ c'est à dire la plus petite relation reflexive et transitive qui contient ℝ.

a) Cas de l'affectation.

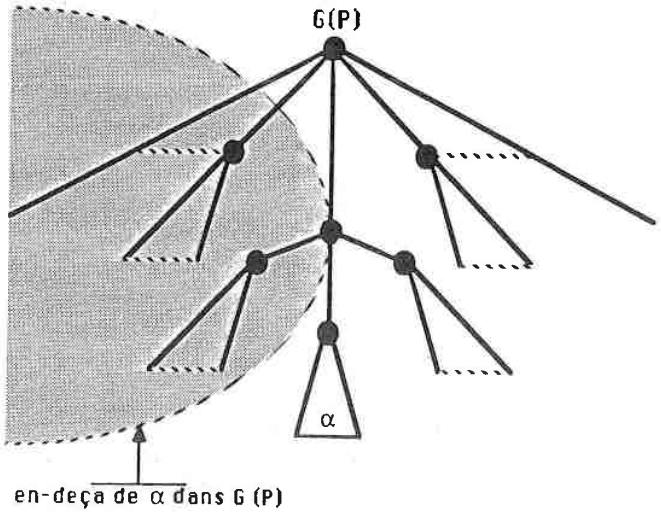
Soit α l'instruction d'affectation, simple ou généralisée, suivante :

(α) $y := f(x_1,...,x_n)$ où x_i $i = 1,..,n$ sont les variables distinctes en partie droite de l'affectation.

Alors $\mathbb{D}[\alpha] = \{y\}$ et $\mathbb{U}[\alpha] = \{x_1,...,x_n\}$

Définitions :

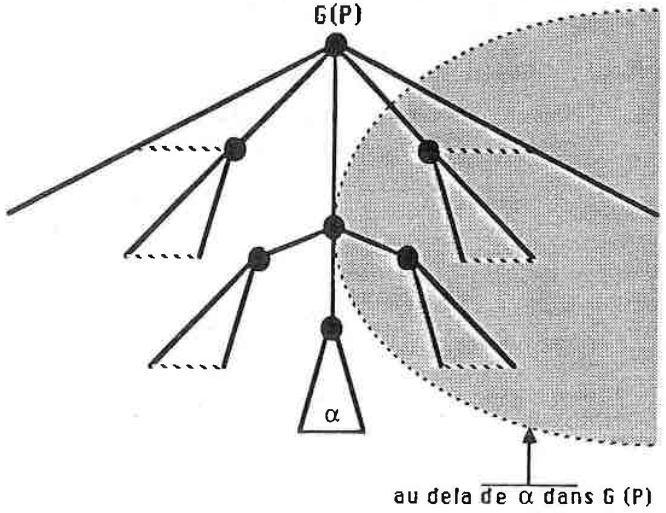
Etant donné le programme P soit $\mathbb{A}nt[\alpha,P]$ l'ensemble des variables v C $\mathbb{D}[\alpha]$ U $\mathbb{U}[\alpha]$ telles qu'il existe au moins une occurrence de v de type définition en-deça de α dans P; soit G(P) la représentation graphique de l'arbre abstrait associé à P.



On note $\mathbb{P}os[\alpha, P]$ l'ensemble des variables v C $\mathbb{D}[\alpha]$ U $\mathbb{U}[\alpha]$ telles que :

a) il existe au moins une occurrence de v de type utilisation au-delà de α dans P.

b) la première occurrence est une occurrence d'utilisation.



Remarque :

1) On fait l'hypothèse que $\mathbb{U}[\alpha]$ C $\mathbb{A}nt[\alpha, P]$

Dans le cas contraire on se trouve, par exemple, dans l'une des situations suivantes :

- les cas d'erreurs, comme celui d'une variable utilisée avant d'être définie, supposés éliminés dans notre programme ce qui évite de faire une analyse dynamique pour vérifier que l'on passe bien dans $\mathbb{A}nt[\alpha,P]$ par une occurrence de v de type définition donc avant d'atteindre α. Il faut remarquer toutefois que dans certains langages ce type d'utilisation est possible ; c'est le cas en **C**.

```
Main ( ) {
int x ;
x = x + 1;
printf (" % d \ n" , x) ;}
```

peut renvoyer le résultat 1.

- Exemple de programme "correct" dans lequel x n'est pas initialisé.

Soit le morceau de programme suivant :

(1) SI C ALORS x : = 1 SINON y : = 2 FSI;

(2) SI C' ALORS z : = x + 1 SINON z : = y FSI;

:

(α) u : = f (...,x,...);
:
(n) SI C" ALORS W : = z + 1 SINON y : = 3 FSI;

Dans ce cas x est une donnée du module associé à α si le programme est correct, c'est à dire si tout chemin de contrôle passant par α passe par la branche ALORS de (1) ; l'étude des flux de données et de la complexité du programme aura permis de réduire le graphe de contrôle aux seuls chemins exécutables.

Pour la définition de Fos [α , P] le niveau d'exigence n'est pas le même ; ainsi dans l'exemple ci-dessus, même avec pour C la condition x ≠ x, z pourrait être considéré comme un résultat.

Etant donné un sous-arbre α de G (P) notons R [α/α] ou R [α] l'ensemble des résultats de α dans un contexte limité à α et R [α/P] ces mêmes résultats dans un contexte P qui est le programme (ou composant de programme) contenant α.

Alors avec notre hypothèse la relation attachée à α est définie par :

D [α] = U [α]

R [α/P] = D [α] Fos [α, P] avec une condition nécessaire de "propreté" soit

D [α] C Ant [α, P].

Si on réduit le contexte de α à elle-même on a R [α] = D [α]

Exemple :

(α) x : = x + y + z + 5

D [α] = {x,y,z}

Si Fos [α,P] = Ø alors R [α] = Ø et on peut supprimer α

Si Fos [α,P] = {y,z} alors R [α] = Ø , et (α) est toujours inutile.

Si Fos [α,P] {x,y,z} alors R [α] = {x} et dans ce cas y et z peuvent être considérés comme paramètres en entrée, x comme paramètre en entrée-sortie.

Remarque :

L'environnement d'un morceau de programme α constitué de Fos [α,P] et de Ant [α,P] peut être défini de manière récursive sur l'arbre abstrait du programme.

b) Cas de la séquence :

Soit la séquence d'instructions α ; β

L'ensemble D [β] doit être défini pour calculer R [β] or, D [β] est défini si

D [β] C D [α] U R [α] U Ant [β,P\α]

Alors la relation attachée à la séquence α ; β est définie par :

D [α;β] = D [α] U (D [β] \ R [α])

R [α;β] = R [α] U R [β]

ou dans contexte élargi à P

D[α;β/P] = D[α;β]

R[α;β/P] = R[α;β] Fos[(α;β),P] avec Fos[β,P] C Fos[(α;β),P]

Exemple :

Soit la séquence

(α) y : = x + 1;

(β) z : = y + 2;

Considérons quelques contextes possibles :

•Si Ant [(α;β),P] = {x} et Fos [(α;β),P] = {z} alors

D [α] = {x} D [β] = {y}

R [α] = {y} R [β] = {z}

Au total D [(α;β)] = {x} et R [(α;β)] = {z} et y peut donc être ramené au rang de variable locale, et même disparaître éventuellement par composition.

•Si Ant [(α;β),P] = {x} et Fos [(α;β),P] = {y} alors

D [α] = {x} D [β] = {y}

R [α] = {y} R [β] = Ø : on applique ici le fait que Fos[(α;β),P]=Fos[β,P]

Au total D [(α;β)] = {x} et R [(α;β)] = {y} ; z peut être ramené au rang de variable locale et même disparaître éventuellement avec l'instruction (β).

•Si Ant [(α;β),P] = Ø et Fos [(α;β)] = {y,z} alors

D [α] = Ø D [β] = Ø

R [α] = {y} R [β] = {z}

Au total D [(α;β)] = Ø et R [(α;β)] = {y,z};

il n'y a pas de paramètre en entrée, x peut être considérée comme locale à (α;β) ; le fait que la première occurrence de x ne soit pas une définition n'est pas nécessairement, comme on l'a vu, pour le langage C, une erreur.

Remarque :
Pour la plupart des langages on a D[α] C Ant [α,P]

•Si Ant [(α;β),P] = {x, y, z} et Pos [(α;β),P] = {x,y,z} alors
D [α] = {x} D [β] = {y}
R [α] = {y} R [β] = {z}
Au total D [(α;β)] = {x} et R [(α;β)] = {z} ; y peut être considérée comme locale à (α;β) ou comme paramètre en entrée-sortie.

c) Cas de l'alternative
• si l'alternative est simple, soit de la forme suivante :

SI C ALORS α ;
Cette alternative est assimilable à la séquence C ; α car au niveau statique on ne connait pas la valeur de C et l'on est ramené à la séquence.

• si l'alternative est complète, soit de la forme suivante :

SI C alors α SINON ß ;
Cette alternative est assimilable à la séquence C ; (α U ß) où l'opérateur U est entendu au sens du parallélisme dans l'ordre d'exécution.
Alors la relation attachée à l'alternative complète

SI C ALORS α SINON β; est définie par :
D [C; (α Uβ)] et R [C; (α; (αUβ)] avec
D [αUβ] = D [α] U D [β]
R [αUβ] = R [α] U R [β]

exemple :
Soit l'alternative

```
(γ) si x > a   alors  y := x + 1;   |
                      z := x + 5;   | (α)
                      w := y + 3;   |

                sinon y := x + 4;   |
                      z := y + 2;   | (β)
```


Dans le contexte le plus lourd où toutes les variables appartiennent à
Ant [(γ),P] et à Pos [(γ),P] on a
D [x>a] = {x,a} D [α] = {x} D[β] = {x}
R [x>a] = Ø R [α] = {w,z,y} R [β] = {z}
Au total D [γ] = {x,a} et R [γ] = {w,z} et y peut être remplacé par une variable locale et considéré comme paramètre en sortie.

Soit l'alternative

```
(γ) si x > a   alors  y := x + 1;   |
                      z := x + 5;   | (α)
                      w := y + 3;   |

                sinon x := x + 1;   |
                      z := y + 3 ;  | (β)
```

Dans le même contexte, on a cette fois
D [β] = {x,y} et R [β] = {z,x}
Au total D [γ] = {x, a, y} et R [γ] = {w, z, x} ; x est devenu un paramètre en entrée-sortie et y un paramètre en entrée.

Remarque :
Si C est une expression alors R [c] = Ø, alors que si C est une affectation R [c] ≠ Ø
Ainsi si γ est l'instruction SI C ALORS α SINON β ;
où C est une affectation :
D [γ] = D [c] U (D [αUβ] \ R [c])
R [γ] = R [αUβ] U (R [c] \ D [αUβ])
ou dans un contexte élargi à P
D [γ|P] = D [γ]
R [γ|P] = R [αUβ] U (R [c] Pos [(αUβ),P])

d) Cas de l'itération
Soit l'instruction γ suivante
(γ) tantque C faire
..α
fait ;
Cette itération γ assimilée à une séquence C ; α ou nous supposons R [c] = Ø ce qui nous donne :
D [γ] = D [C] U D [α]
R [γ] = R [α]

Remarques :

-- Il y a des langages ou l'on peut avoir $R[c] \neq \emptyset$

-- Au niveau d'une itération, il est intéressant éventuellement de faire disparaître le (s) compteur (s) non utilisé (s) postérieurement ou si la première utilisation postérieure est une occurrence de définition explicite. Dans le cas contraire, on peut garder le compteur, à moins d'interdire toute utilisation à d'autres fins d'un compteur dans une itération. Il est possible de maintenir une liste des noms de compteur.

On a souvent envie de considérer $R[\alpha]$ $\mathbb{P}os[\alpha, P]$ que l'on peut noter $RP[\alpha]$ il y aurait alors deux façons de faire pour "modulariser" :

1) modulariser pour utiliser uniquement α dans son contexte initial : il suffit alors de considérer les résultats $RP[\alpha]$.

2) modulariser pour réutiliser α éventuellement dans un autre contexte : il peut être alors intéressant de conserver tous les résultats de α c'est à dire $R[\alpha]$.

exemples :

1- si un compteur d'itération n'est pas utilisé ailleurs, il est inutile de le considérer comme résultat.

2- si on veut utiliser l'itération ailleursP il peut être intéressant, dans d'autres contextes de considérer que le compteur est un résultat : on obtiendra en effet, grosso-modo, un "module" permettant de traiter aussi les problèmes où le nombre de passages dans l'itération est utilisé.

Exemple :

Soit l'itération (δ) tantque y < b

```
        faire
        si x > a    alors α   |
                    sinon ß   |  (γ)
        fait ;
```

où l'alternative (γ) est celle du dernier exemple.

Toujours dans le contexte le plus large on trouve :

$D[\delta] = \{x, a, y, b\}$ et $R[\delta] = \{w, z, x\}$

Cas particulier du compteur dans une itération.

Soit l'itération (λ)

```
s := o ; i := o;  (λ1)
tantque i ≤ n                |
    faire                    |
        s := s + I           |  (λ2)
        i := i + 1           |
    fait ;                   |
```


Le compteur i doit disparaître surtout si la première occurrence de i au-delà de (λ) est une occurrence de définition ; mais on peut souhaiter garder i si cette occurrence est du type i : = f (i)

Dans un contexte général on a ici :

$D[\lambda_2] = i, n, s$ et $R[\lambda_2] = i, i$

$D[\lambda] = n$ et $R[\lambda] = i, s$

I peut donc être considéré selon le contexte au-delà de (λ) comme un paramètre en sortie ou comme une variable locale.

Remarques :

1 - Nous avons supposé que les variables étaient globales.

Dans le cadre d'une structure de bloc, les déclarations modifient l'ensemble des variables ; il suffit alors d'indicer les noms des variables par le numéro du bloc contenant la déclaration.

2 - La relation de dépendance envisagée lors de la présentation du processus de définition par induction structurelle sur la grammaire prenait en considération les dépendances vraies et les autres. Ici au contraire on veut éliminer les variables intermédiaires non visibles de l'extérieur.

3 - Quant aux instructions de branchement, elles ne devraient pas poser de problème si leur utilisation n'est pas destructurante ; éviter les branchements vers l'intérieur d'une instruction de base ou vers l'extérieur depuis une instruction de base. Ce sont des hypothèses classiques que nous avons rencontrées précédemment.

Autres utilisations possibles de la relation $\mathbb{R}$:

En général au niveau des transformations de programmes.

Au niveau de l'analyse statique par exemple :

- la permutation de morceaux de texte (remplacement de la séquence α ; ß par ß ; α) peut être considérée du point de vue parallélisme entre α et ß.
- la composition d'affectation n'est possible que si la variable substituée n'est pas dans $\mathbb{P}os[\alpha,P]$ ou α est l'ensemble des deux affectations considérées.

Au niveau de l'analyse dynamique, on peut envisager, à la manière de l'interprète PROLOG, de modifier le texte source (de α) en fonction du chemin exécuté et donc en fonction des occurrences activées de $\mathbb{A}nt[\alpha,P]$ et de leur type.

6-4 MODULARISATION ET "TYPAGE"

Faire apparaître un type lors de l'analyse d'un texte source d'un programme, c'est parvenir à sa définition en construisant les trois ensembles de base suivants :

a) **Les générateurs :**

Ce sont les opérations fondamentales à l'aide desquelles sont construits tous les objets du type correspondant.

Ainsi toutes les listes sont construites à partir des générateurs liste-vide et adj avec liste-vide : ---> liste

adj : V, liste ---> liste

où V est le type des éléments de la liste.

b) **Les caractéristiques :**

Ce sont les opérations qui permettent d'utiliser les objets du type concerné.

Ainsi les caractéristiques définissant le type liste sont elles tête et reste avec

tête : liste ---> V

reste : liste ---> liste

c) **Les axiomes :**

Ils définissent les valeurs des caractéristiques des résultats des générateurs et donc de tout objet du type concerné.

Par exemple pour le type liste l'ensemble des axiomes exprimant que tête et reste sont les opérations inverses de adj

tête (adj (x,l)) = x

reste (adj (x,l)) = l

tête (liste-vide) et reste (liste-vide) ne sont pas définis

vide (adj (x,l)) = faux

vide (liste-vide) = vrai

Toute autre opération sur des objets du type concerné pouvant être définie à partir des opérations de base de ces ensembles : ainsi adjq qui admet une liste en donnée et une liste en résultat est définie par :

adjq (l,x) =si vide (l) alors adj (x,liste-vide)

sinon adj (tête (l), adjq (reste (l),x))

L'approche de la définition d'un type peut être envisagée de deux points de vue complémentaires le point de vue statique et le point de vue dynamique.

- le type **statique**

L'analyse des déclarations du programme permet de définir la partie statique du type.

Une déclaration de variable permet de relever localement les valeurs d'un certain nombre d'attributs ; nature des caractères, longueur, édition.

L'étude du contexte de la déclaration permet de définir des relations statiques entre la variable considérée et d'autres variables comme la dépendance structurelle (hiérarchie dans un arbre, article d'un fichier) , de localisation (redefines, common...).

L'utilisateur peut attacher à cette variable des attributs sémantiques connus de lui, où des contraintes.

Cette approche permet d'envisager le type recherché du point de vue statique comme la composition de types connus avec synthétisation ou héritage d'attributs le long de la relation de dépendance structurelle par exemple.

- le type **dynamique** :

L'analyse des instructions du programme permet de définir la partie dynamique du type (les opérations).

Pour chaque variable étudiée on peut relever :

- pour chaque occurrence dans une expression (arithmétique ou logique), l'opérateur (arithmétique ou logique) agissant sur elle, l'opérande éventuellement associé pour cette opération. Dans le cas d'une expression arithmétique, on recense des utilisations possibles d'une variable du type étudié, dans le cas d'une expression logique on relève des contraintes ou des contextes d'utilisation de la variable.
- pour chaque occurrence en partie gauche d'une affectation la partie droite associée, l'affectation étant un processus d'échange de valeur.

Outre la mise en évidence d'opérateurs sur la variable, cette approche doit permettre de constituer des classes de variables à proposer à l'utilisateur comme éléments de réflexion pour la constitution d'un type.

La mise en évidence d'un ensemble de modules mono- ou multi-fonctionnels ayant comme paramètres essentiellement des éléments d'une même classe de variables peut contribuer à la découverte des opérations sur le type à définir.

Cette manière intuitive d'envisager la reconstruction de types à partir de textes sources dans le cas d'un système interactif mériterait quelques réflexions ultérieures.

Ce problème n'a pas à ma connaissance été déjà sérieusement envisagé alors que le moyen d'implémenter des types abstraits dans des langages classiques a déjà fait l'objet de travaux comme ceux de M. V. ZELKOWITZ avec PL/1, CELENTANO avec PASCAL, où l'on procède à la modification du langage existant, de K. PROCH et J.C. DERNIAME avec PASCAL où l'on utilise un langage externe hybride d'ATM et de PASCAL préprocessé en PASCAL.

Toute tentative d'abstraction a le mérite de rendre le programme indépendant des choix d'implémentation pour les objets d'un type donné, améliorant ainsi l'évolutivité du programme.

6-5 LES CONSEQUENCES DE LA MODULARISATION.

6-5-1 Concrétisation de la modularité :

La définition de modules dans un programme existant a des répercussions sur le passage du contrôle entre les différents constituants de ce programme (voir exemple châpitre III)

Au niveau du modèle modulaire du texte analysé, il y a disparition d'occurrences de la relation "passer le contrôle", au profit de l'apparition d'occurrences de la relation d'appel ; l'ensemble avec des paramètres effectifs correspondant à l'ensemble des variables visibles de l'extérieur du texte.

Au niveau du module mis en évidence, il s'agit de passer du morceau de texte source analysé à une spécification de module, ce qui nécessite quelques transformations;

- donner un nom au module
- spécifier l'interface

Les variables en entrée (sortie ou entrée-sortie) du module doivent faire l'objet d'une substitution par des paramètres formels en entrée (sortie ou entrée-sortie).

- spécifier le corps du module.

Les variables intermédiaires mises en évidence sont transformées en variables locales et doivent donc être déclarées dans le corps du module.

La définition de niveau logique (spécification des traitements) du module dépend des caractéristiques du texte analysé.

Si ce dernier a de bonnes propriétés (celles de ses représentations) et a déjà fait l'objet de transformations dans le cadre du processus d'amélioration mis en oeuvre localement ou dans un cadre englobant sa définition logique en termes de flux de contrôle, flux de données est déjà connue au moins partiellement.

Dans le cas contraire, approche intuitive d'un morceau de texte monolithique, on peut mettre en oeuvre à postériori les autres modules du processus d'amélioration pour "normaliser" ce nouveau module.

Remarque :

Les difficultés dans la mise en place de la modularisation résultent essentiellement de l'incapacité éventuelle du langage cible à gérer ce concept de module ; c'est le cas en particulier des langages sans structure de bloc.

6-5-2 Appréciation de la modularité.

La modularisation du programme peut être faite par approches successives surtout dans le cas d'un texte source non structuré.

L'aide à l'utilisateur à ce niveau, indépendamment de l'appréciation qu'il peut avoir des fonctionnalités des modules mis en évidence peut être envisagée sous la forme de mesures concernant le module lui-même et le module dans ses relations avec le reste du programme.Cette approche est envisagée dans le module MESURER où les métriques considérées seront placées dans un contexte plus large que celui de l'appréciation de la modularité.

I-7 AUTRES FONCTIONS DU PROCESSUS D'AMELIORATION DE PROGRAMME.

Nous abordons de façon générale trois fonctions :

CONTROLER
LINEARISER
DOCUMENTER

sur lesquelles nous reviendrons dans la deuxieme partie de ce rapport

Ces fonctions, en plus de leurs finalités respectives, ont également un rôle de service dans le processus d'amélioration.L'application d'une des fonctions du processus peut nécessiter

• de vérifier à l'aide de la fonction CONTROLER (MESURER) le gain obtenu ainsi dans le passage à une nouvelle version du programme.

• d'observer l'aspect linéaire de la version de programme résultat (fonction LINEARISER) offrant ainsi à l'utilisateur la dualité texte-autre représentation.

Toute modification du programme en cours d'amélioration exigeant la mise à jour de la documentation associée (DOCUMENTER).

7-1 LES VERIFICATIONS EN COURS D'AMELIORATION : CONTROLER

Dans la littérature on propose, dans le cas de la conception des logiciels et de l'amélioration de leur qualité, l'élaboration de mesures de complexité dans les buts suivants.

- décomposer un programme en modules, ou plus précisément évaluer la modularité d'un programme.
- déterminer les morceaux à risque d'un programme par une simple analyse statique.
- aider au contrôle de qualité du logiciel.
- documenter les constituants du logiciel à l'aide de mesures caractéristiques (attributs métriques).

Actuellement tout système de conception de logiciels ou de restructuration de logiciels possède une composante pour la mesure de la qualité des résultats (atelier ALIS de BULL-SEMS), les mesures évaluées font partie de la documentation des objets manipulés (unités fonctionnelles, données).

Toute la difficulté actuelle des outils proposés et/ou réalisés de mesure de la complexité des logiciels (compleximètre, qualimètre, modulamètre,...) (ALIS,MILESI) (TOULOUSE) (HAB) réside dans le choix de bonnes métriques.

Dans le cas d'un processus d'amélioration de programme, processus nécessairement intéractif, l'écueil à éviter est de proposer une liste exhaustive de toutes les mesures possibles ; il semble préférable, comme pour les transformations de programme, de proposer une liste réduite de métriques de base pour chaque type d'objet à mesurer.

Nous proposons les trois classes de mesures de complexité suivantes :

- les mesures liées aux données : **la complexité informationnelle**
- les mesures liées à un module : **la complexité intramodulaire**
- les mesures liées à un ensemble de modules : **la complexité extramodulaire.**

7-1-1 La complexité informationnelle :

Parmi les mesures les plus courantes liées aux déclarations, il y a :

- le volume, le nombre des variables déclarées
- la profondeur d'un type (nombre des types intermédiares pour arriver aux types de base)
- le nombre de champs d'une structure
- la complexité du graphe des déclarations conditionnelles (M_C Cabe).

7-1-2 La complexité intramodulaire :

La complexité au niveau d'un module est envisagée de deux points de vue, textuel et logique;

- la complexité textuelle

Il s'agit alors de dénombrements d'occurrences de mots d'un vocabulaire dans le texte ; ce sont essentiellement les mesures de Halstead :

- corrélation entre variables
- volume et longueur du module (IBM-50 lignes, TRW-2 pages)
- le vocabulaire (nombre d'opérandes, d'opérateurs)
- niveau de langage
- mesure de la difficulté à lire un programme (K.Christensen PSS)
- fréquence des impuretés

- la complexité logique

Elle est relative au graphe de contrôle du module.

Il s'agit principalement des mesures suivantes :

- le nombre cyclomatique du graphe soit V (G), de Mc Cabe tel que $V(G) = \pi + 1$ où π est le nombre de noeuds alternatifs si le programme est structuré (Formule de Mills).
- le niveau d'imbrication des conditionnelles
- le nombre d'instructions non structurantes (GO TO)
- le software effort de Halstead
- le program-Knots de Woodward (points de croisement).

Remarque : des analyses plus statistiques que théoriques ont été faites sur l'efficacité de ces mesures à apprécier l'amélioration d'un logiciel lors de transformations classiques telles que la linéarisation d'un graphe, le dédoublement de noeuds, la structuration de boucles à sorties multiples (CMCFC).

- la complexité mixte : mesure de la cohésion du module. Thomas J.Emerson propose une métrique prenant en compte les relations entre le flux de contrôle et les références aux données. Cette métrique a un rôle discriminant en ce qui concerne les classes de cohésion possibles pour un module.

On peut rappeler brièvement que les trois classes de cohésion sont

I la classe des modules monofonctionnels
II la classe des modules, séquences multi-actions
III la classe des modules, ensemble d'actions exécutables séparément.

7-1-3 la complexité extramodulaire

Au niveau d'un ensemble de modules, il s'agit de mesurer les relations de chacun d'entre eux avec les autres ou avec les données.

Pour les relations entre modules on peut citer :

- les complexités hiérarchiques et d'inter-connexion des modules liées au graphe d'appel (T.GILB.) HAB.
- les mesures de la découpe d'un logiciel concernant soit les communications statiques entre composants (importation, exportation, visibilité,...) soit la composition de chaque composant, en composants de niveaux inférieurs (nombre de procédures par module, nombre d'instructions par procédure,...)

Pour les relations mettant en jeu les données, on peut citer :

- la complexité du flux d'information de Dennis Kafura (ESSS) avec la mesure et la réduction des chemins de flux de données entre modules grâce à l'introduction de niveaux d'abstraction supplémentaires.
- les mesures de stabilité des opérandes, c'est à dire mesures de l'indépendance et de la visibilité des opérandes en fonction des lieux de définition et d'utilisation (MILESI) CA.

Au cours du processus d'amélioration d'un programme, l'utilisateur doit pouvoir demander, à chaque instant, l'élaboration des mesures les plus significatives vis à vis de l'objectif du moment et de la finalité du module d'amélioration mis en oeuvre.

Le système d'aide à l'amélioration (défini dans la deuxieme partie de ce rapport) doit pouvoir proposer à l'utilisateur, pour chaque module mis en oeuvre, une liste des métriques les plus appropriées à l'appréciation de la réalisation de l'objectif correspondant.

Il est difficilement concevable de travailler à l'amélioration interactive d'un programme de façon modulaire par objectifs sans avoir à chaque instant la possibilité de mesurer non seulement la convergence vers l'objectif du moment (appréhender, comprendre) mais également le respect de la convergence sur l'ensemble des objectifs (modulariser, assainir,...).

7-2 LA REECRITURE DU TEXTE SOURCE : LINEARISER

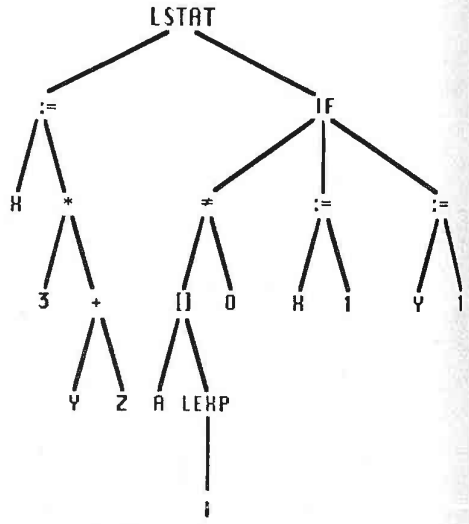
Dans le cadre d'un processus d'amélioration de programmes, linéariser c'est passer, pour un composant quelconque de programme, d'une représentation intermédiaire ("compilée") à une représentation sous la forme d'un texte, syntaxiquement correct, rédigé dans le langage cible, en préservant au moins l'équivalence fonctionnelle.

Cette fonction **linéariser** peut être mise en oeuvre à différents moments du processus d'amélioration et dans le sens d'objectifs variés.

Ainsi on peut souhaiter "voir" à chaque instant (noeud de l'arbre d'amélioration) l'aspect linéaire du composant de programme en cours de transformation, il s'agit alors essentiellement d'une **décompilation** ; ainsi Mentor permet une décompilation abrégée à différents niveaux de l'arbre abstrait du composant de programme.

Citons ici un exemple de décompilation par Mentor issu de (DONZEAU-GOUGE).

```
begin
X := 3*(Y+z);
if A [I] ≠ 0 then X := 1
             else Y := 1
end
```



Fragment de programme — Représentation abstraite

```
N = 2    begin
         #;#;
         end
```

```
N = 3    begin
         X := #;
         if # then # else #;
         end
```

Décompilations abrégées de niveau N

Dans ce cas, le seul objectif est l'aspect linéaire du composant. C'est une possibilité offerte actuellement dans tout environnement intéractif convivial que cette dualité graphe-texte à toutes les étapes du cycle de vie du logiciel.

Mais on peut aussi souhaiter linéariser en contribuant à la réalisation de certains objectifs fixés pour le processus d'amélioration en cours. Considérons quelques exemples d'objectifs et leurs conséquences au niveau linéarisation.

1- Si l'objectif est l'élimination des GO TO dans le texte du programme (ASHCROFT 75, KNUTH 74, PETERSON 73) la linéarisation dépend du type d'équivalence attendu entre les différentes représentations du composant de programme et des propriétés de la représentation intermédiaire.

Au niveau de l'équivalence fonctionnelle (pour des données identiques, des résultats identiques) on peut toujours remplacer les GO TO par la composition de WHILE et IF.

Au niveau de l'équivalence des chemins (équivalence forte : pour des données identiques, des chemins (tests et actions) identiques on peut toujours remplacer les GO TO par la composition de REPEAT et EXIT multiniveaux.

Mais à partir du moment où l'équivalence est du type très forte (même graphe de flux), ou structurelle (très forte et sans modification ou réarrangement des instructions autre que GO TO) il est nécessaire que le graphe de flux ("augmenté" dans le second cas) soit réductible pour pouvoir remplacer les GO TO par la composition de REPEAT et EXIT multiniveaux (RAMSHAW 88).

Force est de constater par exemple que ces transformations ne conduisent pas l'augmentation de la lisibilité pas plus d'ailleurs que d'autres méthodes d'élimination du GO TO comme l'introduction de variables de contrôle, la recopie de code ou la création de sous-programmes sans paramètre (DE BALBINE).

2- Si l'objectif est l'amélioration de la lisibilité, c'est la définition même de la notion de programme lisible qui va guider la linéarisation.

Ainsi BAKER B.S propose, en considérant l'équivalence fonctionnelle :

- dans le cas d'une représentation intermédiaire réductible du composant, les notions de programme réductible (le graphe de contrôle associé est réductible) et de programme proprement imbriqué (utilisation "propre" de REPEAT et IF).

- dans le cas d'un graphe de contrôle irréductible la préservation locale de la structuration : à la nécessité d'un second point d'entrée dans une boucle (REPEAT) est opposée la définition de situations possibles pour ce point dans le corps de la boucle (par exemple pas dans une clause d'un IF).

Avec le même objectif de lisibilité auquel s'ajoute en général la bonne structuration WILLIAMS D.O. propose la généralisation de certaines structures de contrôle en permettant l'exécution (optionnelle) d'un bloc de code :

- à chaque itération dans une boucle (ON-REPEAT)
- à la fin d'une boucle (ON-EXIT)
- à la fin d'exécution du corps d'un sous-programme ou d'une procédure (ON-EXIT-CODE).

La lisibilité du texte cible dépend alors essentiellement de la façon d'introduire ces structures de contrôle généralisées ; adjonction d'instructions au langage cible ou simulation à l'aide d'autres instructions ou mécanismes existants (extension des mécanismes d'exception par exemple).

On peut également au moment de la linéarisation contribuer à améliorer la simplicitén, la rapidité d'exécution du texte cible.

Au total la fonction linéariser contribue à la réalisation des objectifs du processus d'amélioration mais reste dépendante dans sa définition et sa mise en oeuvre d'une part des moyens offerts par le langage cible d'autre part de l'état de la représentation intermédiaire du composant dont on part et donc du type et du mode d'exécution des fonctions d'amélioration précédemment mises en oeuvre.

7-3 LA DOCUMENTATION DU PROGRAMME : DOCUMENTER

Dans le cadre du processus d'amélioration d'un programme **documenter** ce programme c'est apporter l'information complémentaire au code source pour faciliter sa compréhension et la mise en œuvre de modifications le concernant. Toute opération de maintenance ne devrait pas conduire à spécifier le problème ou à reconcevoir l'algorithme pour comprendre avant de modifier le programme.

Dans notre contexte, où la seule donnée concernant le programme est son code source, améliorer du logiciel implique nécessairement de le documenter.

La documentation d'un programme peut être développée à deux niveaux constituant ainsi la documentation interne ou locale du programme ou sa documentation externe ou globale. On distingue habituellement aussi à chacun de ces niveaux la documentation formelle (assertions, tables, graphes) organisée pour être utilisée mécaniquement et la documentation informelle (texte). Quant à la mise en oeuvre du processus d'amélioration, elle nécessite la gestion d'un stock documentaire des représentations de programme manipulées.

7-3-1 La documentation interne ou locale.

Dans son aspect informel, la documentation interne est constituée essentiellement de commentaires éventuellement normalisés. Ainsi on peut générer des identifications dans le code source : rappel du nom d'une procédure en fin de procédure, rappel du nom du module d'origine pour une variable externe, terminaison d'une structure de contrôle par un identificateur significatif, etc...

De même, l'utilisateur dans le cadre d'un système de question--réponse doit pouvoir introduire des commentaires localisés, explicatifs ou informatifs : partie du problème concernée, objets du monde réel manipulés, etc...(Mills H D 70).

Dans son aspect formel, la documentation interne peut être obtenue par ventilation de la documentation externe (vérification d'assertions locales) ou introduite directement par l'utilisateur en général sous forme d'assertions locales (propriétés, invariants (BASILI V.R. 82), pre et post-conditions).

7-3-2 La documentation externe ou globale.

La documentation externe peut se présenter sous forme, de modèles ,de graphes, de textes, de tables ou de représentations particulières du programme mettant en relief certaines de ses caractéristiques.

La documentation externe est habituellement physiquement séparée du programme.

Dans son aspect informel la documentation externe peut être constituée de fichiers de textes, de décompilations abrégées (Mentor) à différents niveaux, etc...

Dans son aspect formel la documentation globale peut être constituée d'un modèle conceptuel des données, du graphe d'appel, du graphe de contrôle, du diagramme des flux de données, de tout ou partie de l'arbre de syntaxe concret (W. REPS) ou de l'arbre abstrait (occurrences définissantes des identificateurs dues à la procédure Mentol. DOC)

Ces représentations partielles du programme constituant elles-mêmes une étape utile pour la constitution de tables par exemple :

- tables des effets d'alias possibles,
- tables des effets de bord possibles,
- références croisées intermodulaires, intra-modulaires
- matrice de dépendance dynamique des procédures.

7-3-3 Le stock documentaire de l'amélioration.

Améliorer un programme c'est définir et construire un arbre d'amélioration à la manière de l'arbre d'analyse de (GRAM A. 86). Aussi non seulement faut-il mémoriser et maintenir cet arbre mais aussi et surtout toutes les informations gérées par les fonctions mises en oeuvre dans le cadre de l'application de tactiques d'amélioration conformes à la stratégie choisie.

Toutes ces informations, parmi lesquelles les différentes versions des diverses représentations du programme à améliorer, nécessitent pour leur gestion un environnement du type base de données si non base de connaissances comme nous l'envisageons dans la IIème partie de ce rapport.

Actuellement la mémorisation et l'exploitation de la documentation sont très variables en fonction de l'environnement de programmation. D'une génération à l'autre des ateliers de logiciel on est passé, par exemple, de l'éditeur de texte, à l'éditeur syntaxique (ou éditeur basé sur des schémas syntaxiques) et enfin à des éditeurs multisyntaxiques (passage d'un langage à un autre, parmi lesquels un langage de documentation). La documentation bénéficiant de ce fait des représentations de programmes sous forme d'arbres attribués, aux attributs de plus en plus structurés et calculables.

7-3-4 Conclusion

La fonction DOCUMENTER est complémentaire à toutes les autres fonctions.

Lors de la mise en oeuvre du processus d'amélioration d'un programme la fonction APPREHENDER permet la constitution d'une documentation initiale et à chaque fois qu'on modifie le programme la fonction DOCUMENTER doit permettre de mettre à jour cette documentation.

La structuration de la documentation d'un programme et sa maintenance s'accommodent fort bien d'une représentation arborescente de ce dernier.

I - CONCLUSION

Dans cette première partie, nous avons abordé quelques unes des fonctions utiles dans le cadre de la mise en oeuvre d'un processus d'amélioration de programmes : on pourrait envisager aussi ASSAINIR (transformations sémantiques), MODELISER le programme (voir P II,ch 2), etc.

On peut considérer qu'il s'agit globalement de fonctions "stratégiques" dont l'application consiste en l'exécution d'une ou plusieurs fonctions terminales.

La spécification plus formelle de ces fonctions à la manière d'Anna GRAM (GRAM A.86). est un travail intéressant envisagé dans un proche avenir.

Chaque fonction, indépendamment de ses données et de son résultat peut être caractérisé par ses objectifs et la ou les qualités essentielles à atteindre grâce à son application au programme (ou à ses représentations).

Nous proposons un résumé de ces fonctions dans le tableau ci-aprés :

FONCTIONS	OBJECTIFS / QUALITE	DONNEES	RESULTATS	COMMENTAIRES
APPREHENDER	Faire un DIAGNOSTIC initial COMPRENDRE la version initiale du programme CONSTRUIRE des structures de représentation permettant d'appliquer les autres fonctions Tentative de SPECIFICATION	Le texte source	Mesure de qualité Documentation initiale Représentations du programme (graphes) des données (modèles).	Il s'agit de décomposer le programme de manière ascendante ou descendante pour initialiser le rapprochement entre la structure physique des représentations et la structure logique de ce programme. APPREHENDER nécessite pour repérer et décrire les composants pertinents pour l'amélioration, la mise en œuvre de fonctions terminales parmi lesquelles on peut citer LOCALISER (fonctions et objets) , DESIGNER, TYPER, REPRESENTER.
RESTRUCTURER	Faire acquerir certaines propriétés aux structures de représentations du programme (réductibilité planarité) Changer de niveau d'abstraction	graphe de flux	graphe de flux avec de nouvelles propriétés (réductibilité, planarité)	Il s'agit d'un rappel de quelques mécanismes de décomposition de graphes en sous-graphes propres et d'acquisition de propriété de réductibilité et planarité La restructuration peut-être syntaxique et/ou sémantique ; RESTRUCTURER c'est mettre en œuvre par exemple des fonctions terminales comme ASSAINIR, OPTIMISER.
(ASSAINIR)	Simplifier et optimiser le programme	graphe + texte	graphe + texte	Il s'agit essentiellement de transformations sémantiques
ARBORISER	Passer d'un graphe à une arborescence Faciliter la mise en œuvre de MODULARISER, DOCUMENTER, LINEARISER.	graphe	arborescence	Il s'agit d'associer à un graphe normalisé (représenté à l'aide de son DFST) une arborescence, support intéressant pour l'application des autres fonctions. Nous proposons un algorithme d'arborisation ARBORISER c'est: essentiellement: DECOMPOSER une structure de représentation (graphe) et la REPRESENTER à l'aide d'une imbrication arborescente de schémas de base (analyse syntaxique).
MODULARISER	Faire apparaître des modules et des types Réduire la complexité Améliorer la réutilisabilité, la fonctionalité, le niveau d'abstraction	graphe ou texte	graphe ou texte correspondant à une spécification de module ou de type	Il s'agit de construire des modules (traitements) et des types (données). Selon la technique utilisée le programme est représenté par un graphe ou par son texte source. nous proposons un algorithme de recherche de modules. MODULARISER c'est: DECOMPOSER de manière fonctionnelle, structurelle ou par les données (notre algorithme), SPECIFIER, TYPER, GENERALISER (PARAMETRER et/ou REPRESENTER (abstraction)).
LINEARISER	Réécrire le texte à partir d'une structure de représentation issue de l'application des autres fonctions. Optimiser l'utilisation des possibilités de langage cible. Améliorer la clarté, la lisibilité conserver la structuration.	structures de représentation	texte dans le langage de programmation cible	Il s'agit du "recodage" du programme dans le langage de programmation cible dans le respect des règles de traduction et d'objectifs de clarté, de lisibilité et de conservation de la structuration LINEARISER c'est essentiellement REPRESENTER sous forme textuelle.
CONTROLER	Vérifier par des mesures la qualité de la représentation du programme Contribution à la construction de l'arbre d'amélioration	structures de représentation ou texte	Mesures de qualité	Il s'agit d'élaborer des mesures de complexité liées aux modules permettant d'apprécier le résultat de l'application de chaque fonction au cours du processus d'amélioration
DOCUMENTER	Mettre à jour la documentation du programme. Améliorer la compréhensibilité, la lisibilité, la maintenabilité.	structures de représentation (arborescence décorée) stock documentaire	nouvelles structures de représentations (modification des attributs associés) nouveau stock documentaire interne et externe	Il s'agit, partant de la documentation initiale constituée par la fonction APPREHENDER de mettre à jour cette documentation à chaque modification des structures de représentation du programme alimentant ainsi le stock documentaire interne et/ou externe.

RECAPITULATION DES FONCTIONS GLOBALES D'AMELIORATION

Mais ces fonctions ne présentent d'intérêt pour l'utilisateur que dans le cadre de la définition d'un processus d'amélioration de programme qui permette leur mise en oeuvre **efficace et conviviale**. Ces deux dernières qualités ne peuvent résulter que de la définition de **modèles simples** tant au niveau des **données manipulées** que des **objets et étapes du processus**, ainsi que de la définition de moyens de réalisation **multi média** du système d'amélioration prenant en compte toutes les représentations du programme et des données attendues à la fois par l'informaticien et l'utilisateur. Ce sont les objectifs de la partie II de notre rapport.

PARTIE II

PROPOSITIONS

POUR UN SYSTEME D'AMELIORATION DE LOGICIELS

Les modèles, le processus.

II - INTRODUCTION

Dans la seconde partie de ce rapport, nous abordons la spécification d'un Système d'Aide à l'Amélioration de Logiciels (SAAL)

Pour permettre la mise en oeuvre de ce SAAL outre ses fonctions principales (pII ch1) nous proposons une modélisation d'une part des données par lui manipulées, objets, relations, fonctions concernés par le programme (pII ch 2), d'autre part du processus d'amélioration (pII ch 3).

Ces modèles génériques compatibles avec tout environnement, tout langage de programmation, tout mode de spécification, tout type de problème permettent à l'utilisateur (informaticien ou non) de définir ses propres modèles de programme et de processus d'amélioration.

En matière de réalisation, notre propostion (pII ch 4) va dans le sens de l'utilisation des SGBD de 3ème génération (en cours d'étude actuellement) dans un environnement évolué de type Structure d'Accueil.

Cette seconde partie se terminant par un travail de réflexion (pII ch 5) sur un cas pratique d'amélioration d'un programme de gestion avec passage de COBOL à ADA.

II-1 FINALITE ET FONCTIONS D'UN SYSTEME D'AIDE A L'AMELIORATION DE LOGICIELS

1-1 FINALITÉ D'UN SYSTEME D'AIDE À L'AMÉLIORATION DE LOGICIELS : SAAL

La mise en oeuvre d'un SAAL nécessite une collaboration entre utilisateurs finals et informaticiens sur des sites informatiques où il existe un grand nombre de programmes opérationnels utiles à l'entreprise concernée, ces programmes restant de maintenance difficile et coûteuse et de réutilisation délicate.

La finalité d'un SAAL est de seconder l'informaticien dans sa tâche de transformation des programmes existants en des programmes cible pour parvenir à :

- une maintenance plus rapide et fiable des programmes;
- une réutilisation croissante des éléments du parc de programmes
- une contribution certaine à la modélisation de l'univers concerné.

Les objectifs et les fonctions d'un système d'amélioration sont comparables à ceux d'un système de développement ; aussi peut-on envisager tout un continuum de solutions, allant d'une méthode entièrement manuelle, irréaliste au moins du point de vue de son coût, à un automate dont on voit mal comment partant du seul texte source des programmes il pourrait atteindre tous les objectifs.

Actuellement il existe, dans différents contextes d'aide au développement, des outils (analyseurs, éditeurs, documenteurs, transformateurs, optimiseurs,...) permettant de réaliser tout ou partie de certaines fonctions du processus d'amélioration.

Notre projet est celui d'un système :

- indépendant du domaine des problèmes concernés (gestion, informatique industrielle,...)
- indépendant du langage de programmation (source et/ou cible) utilisé : (possibilité de passer d'un langage à un autre)
- indépendant des types de composants et de la structuration des programmes.
- indépendant de la méthode de conception des programmes.

L'amélioration assistée que nous proposons associe l'automatisation apportée par des outils à la connaissance du monde réel et à la possibilité de certains choix laissés à l'homme.

1-2 QUELQUES CARACTÉRISTIQUES DU PROCESSUS D'AMÉLIORATION :

Il nous semble que l'amélioration de logiciels ne peut être réalisée que de façon modulaire, par couches successives, et que le processus associé peut être mis en oeuvre par plusieurs "transformateurs" ayant différents points de vue et objectifs successifs.

La modularité :

La taille moyenne des programmes existants et la dimension des problèmes de l'organisation pris en compte nécessitent la décomposition des programmes en modules. Le processus de décomposition, la définition de la notion de module sont autant de difficultés à résoudre.

La démarche progressive :

La réalisation du programme cible en partant uniquement du texte du programme source ne peut être faite en une seule fois mais par couches successibles dans le temps.

Ainsi le "transformateur" crée des versions successives du programme à améliorer chacune d'entre elles correspondant à des objectifs partiels précis et donc à la mise en oeuvre de certaines fonctions du processus d'amélioration.

La multiplicité des transformateurs :

Le taux de rotation des informaticiens et des utilisateurs nécessite d'envisager un système multitransformateurs, avec les problèmes liés à la reprise du processus de transformation en cours (état d'avancement, dernière version, objectifs définis et atteints,...), à la variété des points de vue et à l'ordre de prise en compte par chacun des objectifs partiels envisagés. (stratégie et tactique)

1-3 LES FONCTIONS DU SAAL :

Le niveau des objectifs du SAAL et les problèmes soulevés au paragraphe précédent ont conduit à la mise en évidence de fonctions que doit assurer ce système.

Les principales classes de fonctions sont :

- les fonctions liées à la représentation et à la gestion des objets participant à la définition des programmes à améliorer.
- les fonctions définies dans la première partie du présent mémoire liées à la partie statique du processus d'amélioration.
- les fonctions liées à la partie dynamique du processus d'amélioration et qui permettent de définir des comportements possibles de ce processus et l'assistance à l'utilisateur.
- les fonctions mixtes, utiles à la fois au niveau statique (qualité propre du logiciel) et au niveau dynamique (état d'avancement du processus).

1-3-1 Les fonctions de représentation et de gestion des objets définissant les programmes

Nous abordons successivement les fonctions

REPRESENTER et **GERER**

1-3-1-1 La représentation des programmes : la fonction : REPRESENTER

Cette fonction doit permettre une modélisation du programme constituant une description complète, homogène du programme indépendamment de sa qualité, du langage de programmation utilisé et de son état d'amélioration donc des transformations appliquées.

Cette fonction de construction du modèle du programme recouvre deux tâches :

- la reconnaissance des composants du modèle et la valorisation de leurs attributs.
- l'insertion au cours du temps de nouveaux éléments avec mise à jour des attributs et relations.

La définition du modèle générique de représentation des programmes et quelques exemples d'instanciation font l'objet du pII ch2.

1-3-1-2 Les fonctions de gestion des (versions de) programmes : la fonction : GERER

Une approche "base de données" permet de représenter et de gérer tous les objets concernés par notre processus d'amélioration.

Ces objets semblent appartenir à trois classes de données nécessaires (GOD 87, BEN) :

- des données sémantiquement dépendantes d'un projet d'amélioration particulier.

  Ces données représentent les objets créés et modifiés par les outils d'amélioration ; elles décrivent le logiciel en cours de transformation.

  Voir à titre d'exemple une instanciation du modèle générique de programme dans le chapître 2
- des données décrivant les objets que sont les outils du processus d'amélioration.

  Ces objets de type outil peuvent être considérés du point de vue de leurs relations avec différents types d'objets pour

définir des modèles de processus d'amélioration sorte de représentations de tactiques d'amélioration.

- des données historiques

Elles permettent la conservation des états successifs d'un logiciel en cours d'amélioration.

Elles offrent au transformateur la possibilité, d'une part, de comprendre le pourquoi et le comment des transformations successives d'un logiciel, d'autre part, de reprendre le processus d'amélioration à partir d'un état intermédiaire quelconque.

Des études menées sur les bases de données historiques (KEBAILI A., 84,TIGRE, 85) apportent des éléments de réponse à la gestion de l'histoire d'un système d'information.

1-3-2 Les fonctions liées à la partie dynamique du processus

Le pilotage de l'utilisateur ou l'assistance en amélioration :
la fonction : **PILOTER**

La fonction PILOTER a pour but de guider l'utilisateur dans la mise en oeuvre du processus d'amélioration.

Si la modélisation de la partie statique (ressources du processus) du processus d'amélioration permet le contrôle et l'aide à la gestion de cette partie statique, la modélisation de la partie dynamique (comportement des composants) apparait comme son complément pour l'assistance à l'amélioration de logiciels.

La nécessité d'une modélisation du processus d'amélioration résulte :

- du nombre, de la complexité et de l'interdépendance des fonctions utiles au cours de l'amélioration de logiciels.
- du fait qu'une description formelle de processus permet l'identification et la définition de l'état courant de celui-ci ; base de compréhension et de communication entre "transformeurs " et utilisateurs des logiciels.
- de l'intérêt d'une structuration formelle pour le contrôle de la cohérence de l'amélioration : structuration indispensable à la réutilisation du processus d'amélioration.

Dans le pII ch3, nous proposons un modèle générique de processus d'amélioration.

1-3-3 Les fonctions mixtes :

Nous présentons successivement les fonctions
DOCUMENTER et **CONTROLER;**

1-3-3-1 La fonction : **DOCUMENTER :**

C'est un ensemble d'actions permettant de renseigner le "transformateur" à tout instant sur l'état du programme qu'il améliore sur le problème résolu par le programme en vue d'une réutilisation.

Les deux types d'utilisateurs du système documentaire étant le "transformateur" qui est l'intervenant extérieur, et le SAAL qu'on peut qualifier d'interlocuteur interne.

Pour mettre en oeuvre la fonction documentation, il faut :

- définir le stock documentaire
- disposer de moyens de gestion et d'interrogation de ce stock

Le stock documentaire est essentiellement constitué des instanciations du modèle conceptuel générique de programme qui est abordé dans le pII ch2.

Dans un environnement classique, il pourrait être constitué à deux niveaux :

- le modèle modulaire réduit la plupart du temps au sous-graphe d'appel des composants modulaires.
- le modèle intramodulaire souvent réduit :
  - au graphe de contrôle
  - aux références croisées

  ce qui est encore trop si l'on en croit MYERS (SR 146) qui suggère de réduire la documentation d'un module à son code et à des commentaires concis,mais il est vrai pour un objectif différent du nôtre.

Dans notre cas, le stock documentaire comporte, d'une part, les modèles modulaire et intramodulaire représentant le programme à améliorer (syntaxe et sémantique "internes"), d'autre part, le modèle "relationnel" représentant les objets du monde réel et le problème résolu (sémantique "externe" vis-à-vis du programme).

Ce stock documentaire est enrichi :

- de manière automatique par les retombées de la mise en oeuvre de certains outils d'aide à l'amélioration.
- de manière manuelle par le "transformateur" fournissant de l'information essentiellement sémantique pendant le

déroulement du processus (valorisation d'attributs du modèle)

Les moyens de gestion et d'interrogation du stock sont liés au choix d'implémentation de ce stock ; nous abordons brièvement les niveaux possibles de réalisation dans le pII, ch 4.

1-3-3-2 Le contrôle de qualité : la fonction **CONTROLER**

Le processus d'amélioration consiste en la construction d'un programme cible comme dernier terme d'une suite de versions d'un programme source. Le passage d'une version vi à une version vi + 1 dans la suite résulte de la transformation de l'énoncé que constitue la version origine vi.

Les deux préoccupations essentielles du "transformateur" sont alors d'assurer :

- un contrôle sémantique de fidélité de chaque version à la version initiale : toute solution fournie par l'exécution d'une version intermédiaire doit être correcte et équivalente au programme source.
- un contrôle de qualité

L'obtention d'un programme cible de qualité suppose une politique d'assurance et de contrôle de la qualité du programme adaptée aux spécificités du domaine d'application concerné et donnant les moyens (procédures d'intervention) de respecter les objectifs de qualité du site.

1-3-3-2-1 Définition et construction de la qualité

Pour être efficace, la construction de la qualité doit être intégrée à l'amélioration du logiciel permettant le contrôle et la correction tout au long du processus de transformation du logiciel.

Le contrôle de qualité peut être assuré par le "transformateur" lui-même et doit réaliser les fonctions suivantes :

- contrôle du processus d'amélioration

il s'agit de s'assurer que l'amélioration du logiciel respecte une logique possible dans l'ordonnancement de mise en oeuvre des fonctions d'amélioration (par exemple, éviter de boucler inutilement); ce rôle peut être dévolu au pilote du système d'aide à l'amélioration.

- contrôle de la qualité propre du logiciel.

il s'agit d'évaluer la qualité du logiciel pour vérifier qu'à l'instant **t** le niveau de qualité est conforme à celui défini à

priori par l'assurance qualité, et de s'assurer que la version de programme produite à une étape donnée demeure cohérente avec les versions précédentes,mais surtout qu'il s'agit bien d'une amélioration, c'est à dire d'un progrès mesurable au niveau des critères de qualité envisagés.

Les contrôles sont plus efficaces s'ils s'appuient sur les mesures de la qualité.

Depuis quelques années, l'étude de la spécification de la qualité a permis de répertorier facteurs (caractérisation de la qualité du point de vue utilisation) et critères (caractéristiques quantifiables pour estimer les facteurs) de qualité et de réaliser des outils de mesure.

Un certain nombre de métriques sont actuellement proposées mais la recherche dans ce domaine a plutôt produit des modèles de mesure concernant un aspect restreint de la qualité du logiciel (cas de la mesure de complexité) appliquant généralement des théories mathématiques (théorie des graphes) ou des théories de la physique (entropie).

Chaque modèle proposé ne permet de saisir qu'un aspect particulier de la qualité du logiciel : lisibilité, modularité, complexité, fiabilité,...

Ceci ne nuit en rien à notre processus d'amélioration, dans lequel le "transformateur", à chaque instant, travaille dans le sens d'un objectif partiel lié à un nombre restreint de facteurs et de critères de qualité.

En contre partie notre système doit intégrer les différents modèles auxquels on est susceptible de faire appel ; et c'est là que l'utilisateur doit déterminer le sous-ensemble de métriques"signifiantes" pour son parc de logiciel compte tenu des objectifs globaux de l'amélioration.

L'approche quantitative de la qualité du logiciel la plus courante est la mesure de la complexité du logiciel issue de l'analyse statique des programmes (mesures statiques).

1-3-3-2-2 Les mesures de complexité :

Notion de complexité.

Les définitions "académiques" de la complexité font toutes allusions à un assemblage, difficile à comprendre, de plusieurs éléments.

Ces définitions peuvent s'appliquer à tout niveau :

- un logiciel est un assemblage de programmes
- un programme est un assemblage de modules
- un module est un assemblage de blocs
- un bloc est un assemblage d'instructions
- une instruction est un assemblage d'opérateurs et opérandes

La complexité est toujours, au sens étymologique, une estimation de la difficulté de compréhension de ces assemblages.

Les différents modèles de complexité qui concourrent à la complexité propre du logiciel sont :

- la complexité textuelle

  Le programme est considéré comme une succession d'opérateurs et d'opérandes (ex. métriques de Halstead), BASILI, BOWEN,...)
- la complexité structurelle

  Le programme est considéré comme un ensemble de modules :métriques relatives au graphe d'appel, structure modulaire (hiérarchie, interconnexion).
- la complexité fonctionnelle

  C'est la complexité de la fonction du logiciel.
- la complexité logique

  C'est la complexité de la structure du flot de contrôle du programme ou de ses modules (ex : métriques de Mc Cabe, CHEVALLIER,...)
- la complexité stylistique

  C'est la complexité du code source.
- la complexité relative aux données

  C'est la complexité des structures de données et du flot de données (ex : mesures d'OVIEDO, WITWORTH,...)

1-3-3-2-3 Le contrôle de qualité et l'amélioration de programme :

Les relations entre facteurs et critères de qualité ont été définies de la manière suivante par Mc Cabe pour deux facteurs principaux intéressant l'amélioration de programme :

Facteurs	Critères
Maintenabilité :	Cohérence, Simplicité, Concision Modularité, Autodescrpition.
Réutilisabilité :	Modularité, Autodescription
Portabilité :	Indépendance / Machine et Système.

L'intérêt des mesures de complexité est essentiel vis à vis de ces facteurs, de plus ces mesures ont des caractéristiques et propriétés intéressantes pour notre processus d'amélioration :

- ce sont des mesures **statiques**. Or pour la réutilisabilité et la maintenabilité, il est nécessaire d'avoir des estimations prédictives ne nécessitant pas l'exécution des programmes.
- ce sont des mesures **indépendantes du langage** utilisé ; ce qui permet des vérifications sans encombre quand on profite du processus d'amélioration pour changer de langage de programmation.
- ce sont des mesures, en général, faciles à automatiser.

Remarque :

A ces contrôles s'ajoutent naturellement tous les contrôles syntaxiques et sémantiques classiques dans un environnement de programmation.

1-4 L'ASPECT DYNAMIQUE DE L'AMELIORATION DU LOGICIEL

Comme pour la description de la dynamique de tout système à comportement déterministe, quel que soit le modèle d'exécution appliqué, à tout projet d'amélioration correspondent les ensembles suivants :

- un ensemble d'**objets**
- un ensemble d'**activités**
- un ensemble de **règles**

La production d'objets, le déroulement d'activités, et l'application de règles ont lieu dans le cadre d'un ensemble de **tâches** du projet, tâches auxquelles participent des **personnes** à travers une ou plusieurs **fonctions** de la **phase** d'amélioration du **cycle de vie** du logiciel.

Les activités et /ou sous-fonctions se déroulent en appliquant (souvent) des méthodes et/ou des techniques éventuellement à l'aide d'outils.

Ce modèle de processus d'amélioration fait l'objet d'une définition plus complète dans le chapitre 3 mais envisageons ici quelques-unes des étapes possibles de l'exécution d'un projet d'amélioration.

1-4-1 Exemples de concepts, tâches et étapes d'un projet d'amélioration

1-4-1-1 Les objectifs généraux du projet

Le transformateur définit son projet en termes d'objectifs généraux pour un ensemble de programmes concernant une même classe de problèmes (scientifiques, de gestion,...) et rédigés dans un même langage.

Ces objectifs généraux peuvent inclure :

- le changement de langage de programmation.
- le changement de matériel, et/ou de système d'exploitation
- la prise en compte de nouveaux concepts de programmation (type, coucurrence,...)
- le niveau d'amélioration envisagé, par exemple sons la forme des facteurs de qualité essentiels à faire acquérir aux programmes avec l'indication des moyens de vérification de cette acquisition (taille maximum d'un module, résultats attendus des mesures de complexité,...)
- la création d'une nouvelle structure de données (passage d'un SGF à un SGBD).

De tels objectifs permettent de définir globalement le projet d'amélioration, de donner une idée des grandes étapes à venir et du poids relatif probable de l'homme par rapport aux outils dans le déroulement du processus.

1-4-1-2 La cible d'amélioration

L'utilisateur définit le morceau de texte source, composant du programme, qui constitue la cible d'amélioration.

Pour un composant d'un certain niveau de granularité (voir figure 1) l'utilisateur peut procéder à une amélioration individuelle, puis à une amélioration contextuelle en prenant en compte les relations de ces composants avec les composants de même granularité (relations "horizontales") et les composants de granularité différente (relations "verticales").

Une cible d'amélioration peut être spécifiée en fonction des points de vues suivants :

(1) Point de vue structurel qui découle de son niveau granulaire (hiérarchie de composants; composant modulaire, bloc, instruction,...)

(2) Point de vue relationnel dans les types de relations (verticales ou horizontales) intéressant le transformateur (cible homogène ou hétérogène).

(3) Point de vue qualitatif dans la définition des critères de qualité que l'on souhaite améliorer - conformément aux objectifs généraux.

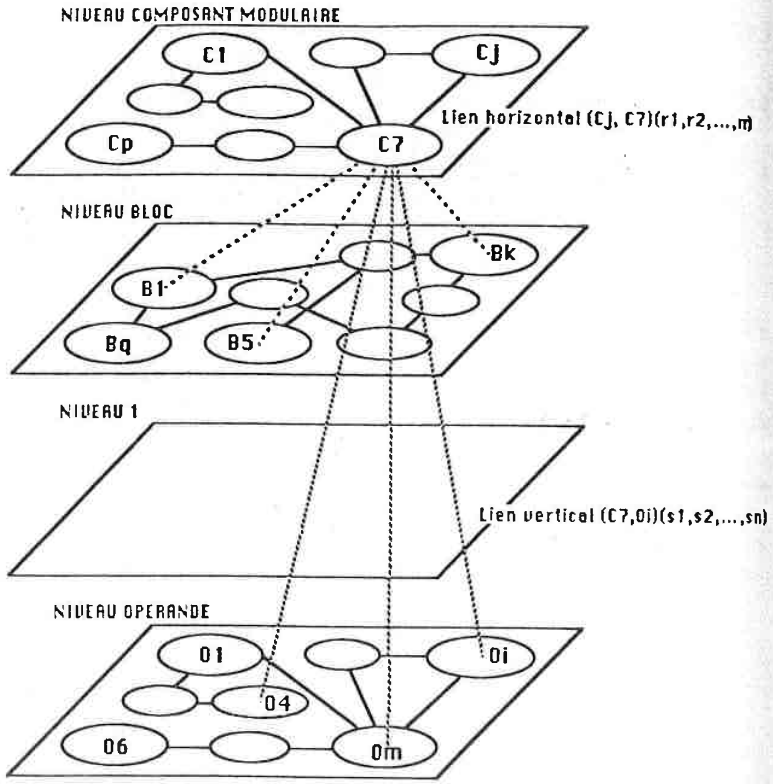


FIGURE 1

1-4-1-3 L'arbre d'amélioration

Reprenant l'excellente idée d'arbre d'analyse introduite dans (GRAM,86) pour mémoriser les perspectives et le plan de développement de logiciel nous proposons la notion d'arbre d'amélioration.

A chaque étape de l'amélioration, l'arbre d'amélioration permet de présenter l'exploration raisonnée et hiérarchisée des différentes solutions par niveaux successifs de spécification et de résolution de sous-problèmes. Chaque étape apparait comme une transformation appliquée à tout ou partie de l'énoncé du sous-problème considéré.

Cet arbre d'amélioration est un support agréable et précis, dans le cadre d'un processus intéractif, pour définir, mémoriser et modifier le plan d'amélioration.

Dans le but de modéliser (voir systématiser) la démarche d'amélioration, nous proposons de particulariser les différentes phases d'amélioration que sont :

- Les **stratégies** permettant d'expliciter différents modes d'amélioration de programme. Elles caractérisent des règles de comportement général guidant les choix du transformateur en tenant compte des objectifs généraux du projet par exemple : privilégier la fonction MODULARISER.
- Les **tactiques** décrivant les étapes logiques d'amélioration conduisant à un composant possédant certaines propriétés caractérisant les critères de qualité envisagés. Elles mettent en oeuvre fonctions primaires et activités conduisant à la satisfaction, par le composant, aux critères de qualité. Par exemple : élaborer le DFST du composant puis linéariser.
- Les **fonctions primaires**, les **activités** sont les étapes élémentaires de l'amélioration d'un programme. Par exemple l'activité DESIGNER qui consiste à expliciter une entité par un nom et une définition informelle peut être une étape des fonctions APPREHENDER ou DOCUMENTER.

Remarque :

Il s'agit ici, nous en mesurons pleinement les limites, de propositions qui ne cherchent qu'à définir un cadre schématique aux efforts actuels visant à mieux maîtriser l'amélioration de logiciels.

Le contenu de l'arbre d'amélioration peut être défini par :

(1) La **racine** : elle représente le texte source à améliorer.

(2) Les **noeuds intermédiaires** : chaque noeud représente une étape de l'amélioration, il lui est associé une version du composant en cours d'amélioration. Les fonctions primaires ou activités applicables à ce noeud déterminent un ou plusieurs noeuds fils.

(3) Les **feuilles** de l'arbre sont de deux sortes :

- les unes correspondent à des voies abandonnées soit par manque d'intérêt (pas de gain de qualité) soit par impossibilité.
- les autres correspondent à une solution soit sous la forme d'une version originale du composant, soit sous la forme de l'identification avec un composant existant (réutilisation).

L'intérêt d'un tel arbre d'amélioration est multiple.

- Permettre la définition de stratégies et tactiques d'amélioration.
- Permettre le suivi du processus d'amélioration
- Permettre de considérer certains noeuds de l'arbre comme des cibles d'évaluation de la qualité du composant associé.
- Conserver un historique des tactiques envisagées.
- Eviter de refaire un travail déjà fait.
- Eviter d'en faire plus que demandé, autrement dit arrêter l'amélioration dès que les objectifs sont atteints.

1-4-1-4 L'évaluation :

Il est nécessaire de pouvoir évaluer le niveau de qualité du composant de programme objet du traitement d'amélioration. Cette évaluation peut intervenir à différentes étapes de l'amélioration.

- Dès le début dans le cadre de la mise en oeuvre de la fonction APPREHENDER pour établir un **diagnostic de qualité** du composant de programme en effectuant des mesures caractéristiques des critères de qualité concernés par les objectifs ; il s'agit de la racine de l'arbre d'amélioration.
- Avant la mise en oeuvre d'une fonction (primaire) pour établir le niveau de qualité de la version de composant de programme cible de l'amélioration correspondant à un noeud de l'arbre d'amélioration.
- Après la mise en oeuvre d'une fonction (primaire) pour apprécier le gain de qualité selon les critères visés par l'application de la fonction en particulier au niveau de chaque feuille de l'arbre d'amélioration.
- En fonction du triggering ou déclenchement automatique de mise à jour de certains attributs relatifs au code (OQUENDO 89)

On trouve actuellement, dans la littérature sur le sujet, un grand nombre de métriques destinées à l'évaluation d'un composant modulaire individuel (Mc CABE 76) (HALSTEAD 77), etc... mais aussi des approches destinées à des évaluations inter-modulaires (MYERS,74), (GILB 77), (YIN 87) etc...

Dans le cadre de l'évaluation de la qualité du logiciel dans un environnement intégré de génie logiciel nous proposons dans (AYOUB 89.3) une approche d'évaluation flexible du code source. Nous visons la conception et l'implémentation d'un évaluateur basé sur un modèle partant d'un noyau évolutif des métriques de base.

Sur ce noyau, nous proposons de définir :

- un ensemble de fonctions de gestion du noyau et de son évolutivité.
- un ensemble de fonctions pour l'expression de nouvelles métriques, soit définies par l'utilisateur, soit composées d'autres métriques.

Quant à la conservation de l'information nécessaire à l'évaluateur ou résultat des évaluations, nous l'envisageons essentiellement sous la forme d'un enrichissement du profil des schémas de données de l'amélioration (développés dans les chapitres 2 et 3) par les attributs, entités et relations utiles à l'élaboration des métriques calculées (AYOUB 89.0)

1-4-1-5 La mise en oeuvre des fonctions (primaires) d'amélioration

L'exécution de chaque fonction ou activité résulte soit d'une décision de l'utilisateur, soit de la mise en oeuvre d'un chaînage avant (ou arrière puis avant) d'un moteur d'inférence sous-jacent.

Les informations destinées à l'amélioration et à l'évaluation des textes sources peuvent avoir différentes origines :

1) L'analyse statique et/ou dynamique du code source.
2) L'interrogation des personnes censées être "bonnes sources" d'informations sur le code à améliorer (utilisateur) et sur le processus d'amélioration (informaticien).
3) L'expertise des informations extraites par analyse ou interrogation
4) Le triggering avec mise à jour d'attributs relatifs au code
5) Les outils d'amélioration élaborant certaines fonctions.

1-4-1-6 Conclusion : le processus et les modèles associés

Nous venons d'aborder quelques concepts, supports et étapes liés à la démarche d'amélioration.

Dans les chapitres suivants, nous précisons les modèles d'appui respectivement des données (p II ch 2) et du processus d'amélioration (p II ch 3).

Améliorer du logiciel nécessite l'utilisation d'outils de mise en oeuvre des fonctions du processus, outils qu'il est souhaitable de centraliser sur un même site pour constituer un atelier d'amélioration de logiciels.

Ce système d'amélioration n'est-il pas un outil possible pour un Atelier Intégré de Génie Logiciel (AIGL) construit sur une structure d'accueil ?

La réponse à cette question est l'objet du p II ch 4.

II 2 UN MODELE CONCEPTUEL GENERIQUE DE REPRESENTATION DES PROGRAMMES.

2-1 UN MODELE GENERIQUE DES DONNEES

Les données sont constituées ici des différentes versions des textes sources des programmes à transformer et de la documentation éventuelle les accompagnant à un momemt quelconque du processus d'amélioration.

Nous envisageons successivement les modèles de référence, à la fois simples et propres au niveau formel, puis le modèle proposé.

2-1-1 Les modèle d'appui : Le modèle structurel hiérarchique et Le modèle ERC

Dans notre contexte d'amélioration de textes sources existants, le modèle doit être assez général pour permettre la description, la compréhension et la transformation d'un programme quelque soient le style et la structure du programme, quelque soient les caractéristiques et particularités du langage utilisé.

Les critères fixés pour définir le modèle de données servant à décrire et à manipuler les objets concernés par le processus d'amélioration de programmes sont ceux-là même fixés par l'équipe Génie Logiciel (KIM 86) au niveau du développement d'un logiciel.

- **puissance de représentation** : représentation à la fois des aspects structurels et sémantiques du monde de la programmation.
- **évolutivité** : l'adjonction ou le retrait d'objets ou de relations ne doit pas remettre en cause tout le schéma précédemment construit de la base.
- **simplicité** : le modèle doit être à la fois facile à comprendre et facile à utiliser (même pour un utilisateur non spécialiste de la programmation).
- **base formelle** : le modèle doit avoir une base théorique solide pour éviter toute ambiguïté d'application et d'interprétation et pour permettre éventuellement une vérification formelle.

Le modèle Entité-Association (E-A de Chen 76) a un pouvoir descriptif assez riche sans être complexe et les diagrammes de représentation des schémas sont très lisibles.

Le modèle E-A souffre toutefois de ne pas être fondé sur une base théorique solide contrairement au modèle relationnel, et surtout il est mal adapté à la représentation d'objets à structure complexe comme des programmes.

En revanche, une extension récente de ce modèle proposée par C. PARENT, le modèle ERC , permet une meilleure représentation de ce type d'objet.

Quant aux approches orientées objet ou relationnelle, elles ne sont pas sans inconvénient.

L'approche orientée objet permet le traitement d'objets complexes identifiés, offre une extensibilité facile avec les classes, le concept d'héritage et le stockage dans le même formalisme des données et des programmes. En contre partie, l'approche orientée objet (à la manière E-A) manque de cadre formel; les Langages de Manipulation de Données (LMDs) des Systèmes Orientés Objet (SOOs) ne sont pas simples, ni complètement indépendants des mécanismes physiques de stockage (conflit entre l'encapsulation et les possibilités d'un bon langage de requête).

De plus les SOOs sont monoutilisateurs, sans contrôle de concurrence sur les données et ont des performances médiocres de gestion de grandes quantités de données (absence de buffers et de clustering : stockage dans un même endroit des objets ayant des relations entre eux).

En revanche l'approche relationnelle est simple, formelle, avec peu de concepts.

Les LMD_S relationnels sont simples, puissants, indépendants du stockage physique des données. Les systèmes relationnels contrôlent la concurrence et permettent ainsi le portage correct des données, ils sont multiutilisateurs et optimisent le traitement de quantités énormes de données.

En contrepartie, les systèmes relationnels manipulent des objets simples (les tuples), avec des **clés** d'identification.

Seules les données sont gérées par le SGBD correspondant, les programmes étant stockés dans un système différent.

L'absence du concept d'héritage entraine la redondance des informations.

La solution retenue consiste à représenter les programmes et donc les objets constituants à l'aide de deux modèles :

- le modèle structurel (MS) issu de la notion de modèle sémantique hiérarchique étendu (SHM +,(BRODIE M.,84)) qui permet la définition, l'abstraction et la représentation des objets en termes, de classes, d'entités, de types d'entités et d'attributs.
- le modèle Entité-Relation-Complet (ERC de Parent 87) proche du modèle E-A avec une algèbre de manipulation des objets décrits par ce modèle.

Ce modèle ERC et son algèbre ont fait l'objet d'une définition formelle ce qui à la fois permet d'éviter toute ambiguïté et offre une base théorique à des développements ultérieurs.

2-1-1-1 Le modèle structurel (MS) :

Le modèle structurel permet la construction et la spécification des entités essentiellement en termes d'attributs associés et de hiérarchisation des objets.

Le modèle est décomposé en une hiérarchie d'abstractions. Une abstraction d'un système est un modèle de ce système dans lequel certains détails sont délibérément omis.

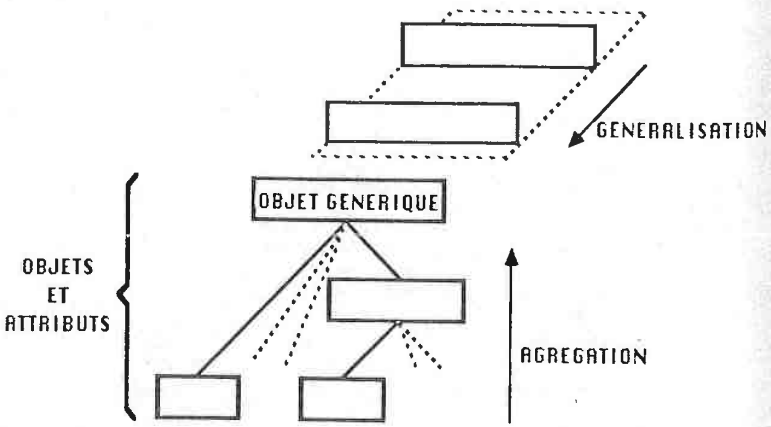
Cette hiérarchie d'abstractions permet l'accès au modèle à différents niveaux d'abstraction facilitant l'approche selon différents points de vue de l'utilisateur.

On distingue habituellement deux formes d'abstractions : **l'agrégation** et la **généralisation**.

L'agrégation (SMITH, 77a) est l'abstraction utilisée pour manipuler comme un tout (appelé objet complexe agrégat) le regroupement de divers objets (qui sont les objets composants).

La généralisation (SMITH, 77 b) est l'abstraction qui permet de concidérer un groupe d'objets complexes appelés objets spécialisés, comme un objet complexe unique qui est l'objet générique.

Ces deux processus d'abstraction peuvent être mis en oeuvre séparément ce qui conduit J.M. et DCP Smith à proposer une représentation graphique où agrégation et généralisation sont introduites de façon orthogonale.



REPRESENTATION GRAPHIQUE DE LA STRUCTURE MS.

L'agrégation est supposée représentée dans le plan de la page, le niveau d'agrégation allant en croissant du bas vers le haut de la page.

La généralisation est supposée représentée dans un plan perpendiculaire à la page, le niveau des objets génériques allant en croissant de l'arrière vers l'allant de la page.

1) L'agrégation

L'agrégation est l'abstraction qui regroupe en un seul attribut complexe identifié plusieurs attributs simples ou complexes d'une même entité ayant un lien entre eux.

Il y a ainsi création d'objets agrégés, avec comme cas particulier l'entité elle-même qui peut être considérée comme l'agrégat de tous ses attributs.

2) La généralisation

La généralisation est l'abstraction qui regroupe en un seul type entité particulier, différentes classes d'entités spécifiques partageant des attributs communs ; ces classes peuvent être considérées comme sous-catégories du type d'entité associé. Les attributs communs devenant des attributs du type entité. Il y a création ainsi d'objets génériques constituant une hiérarchie.

L'intérêt de ces objets génériques est triple :

1) L'application à ces objets génériques d'opérateurs définis sur les entités des classes généralisées.
2) La spécification d'attributs généraux pour ces objets génériques.
3) La spécification de relations impliquant des objets génériques.

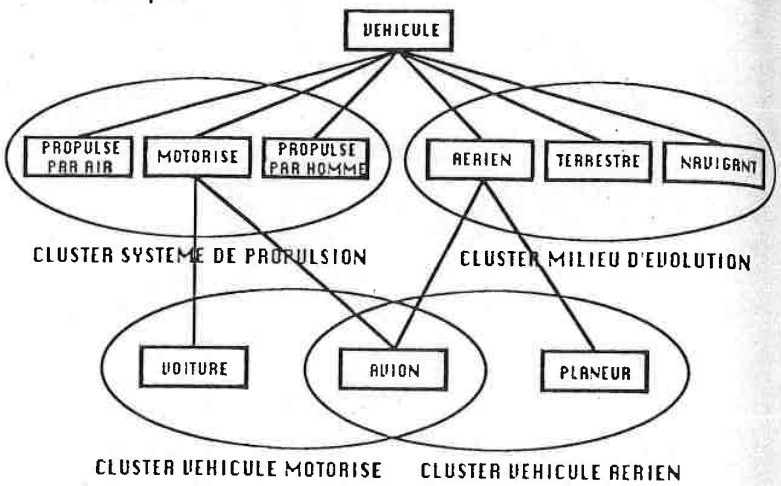
Cette hiérarchie ne peut être une arborescence que si la décomposition de chaque objet générique est faite à l'aide de classes mutuellement exclusives (cluster).

Toutefois, ce partitionnement en cluster n'est pas toujours possible en particulier au bas de la hiérarchie au niveau du regroupement des instances d'entités où le processus de généralisation est parfois appelé **classification**.

La classification est l'abstraction qui consiste à percevoir comme un objet complexe (classe) une collection d'éléments de même nature (les instances d'objet).

De façon plus générale si la généralisation est basée sur l'abstraction de propriétés spécifiques de sous-classe, la classification par contre sert à collecter des instances ayant les mêmes propriétés

Exemple :



Remarque :

J.M.. Smith et DCP Smith montrent la simplicité qu'il y a à traduire ou representer les abstractions dans le modèle relationnel de Codd. Il suffit pour cela d'associer une relation à chaque objet générique de la hiérarchie à condition de respecter certaines conditions sémantiques nécessaires.

2-1-1-2 Le modèle E-R-C

2-1-1-2-1 Présentation du modèle ERC

Les caractéristiques principales différenciant le modèle ERC du modèle E-A sont les suivantes :

a) Le concept de Type Association (TA)

Les propriétés du TA sont les mêmes dans les deux modèles.

Les TA_S peuvent lier un nombre quelconque de Types Entités (TE).

Chaque lien entre un TE et un TA est identifié par un nom exprimant le rôle joué par le TE dans l'association.

Une association peut donc lier plusieurs fois le même TE avec des rôles différents.

Les TA_S peuvent avoir des attributs.

b) Le concept de Type Entité.

- Dans le modèle E-A, chaque TE a une clé, donc deux entités ayant la même clé ne peuvent exister dans le système.

- Dans le modèle ERC, les TEs n'ont pas de clé et les duplications sont permises.

c) Le concept d'Attribut.

- Dans le modèle E-A les attributs sont obligatoires, monovalués et simples.
- Dans le modèle ERC un attribut peut être :
  . obligatoire ou optionnel
  . monovalué ou multivalué
  . simple ou complexe.

Ces différences essentielles entre le modèle EA et le modèle ERC résultent directement des définitions des concepts de base du modèle ERC.

2-1-1-2-2 Définition du modèle ERC

1) Le concept de DOMAINE :

Le domaine d'un attribut peut être élémentaire ou complexe . Un domaine élémentaire est un ensemble V nommé de valeurs élémentaires avec des opérateurs relationnels sur V^2.

Un domaine complexe, reflétant complexité et multivaluation, permet de générer des amas (multiensembles) de valeurs pour les attributs correspondants.

2) Le concept de STRUCTURE :

Ce concept sert à décrire les caractéristiques (domaine, cardinalités, composition) d'un attribut indépendamment de l'objet (TE, TA ou Attribut complexe) qu'il décrit.

3) Le concept de TE :

Un TE, E est un quadruplet : E = (nom, sch, pop, env) tel que :

nom (E) est le nom de E

sch (E) est le schéma de E; ensemble non vide des structures composant E.

pop (E) est la population de E ou occurrences d'entités de E

env (E) est l'ensemble des TAs auxquels participe E avec pour chaque TA, R, le rôle de E dans R.

4) Le concept de TA

Un TA, R est un quadruplet : R = (nom, env, sch, pop)

tel que :

nom (R) est le nom de R

env (R) est l'ensemble des quadruplets décrivant les TEs liés par R

sch (R) est le schéma de R : ensemble éventuellement vide des structures composant R.

pop (R) est la population de R ou occurrences d'associations de R.

5) Le concept d'attribut

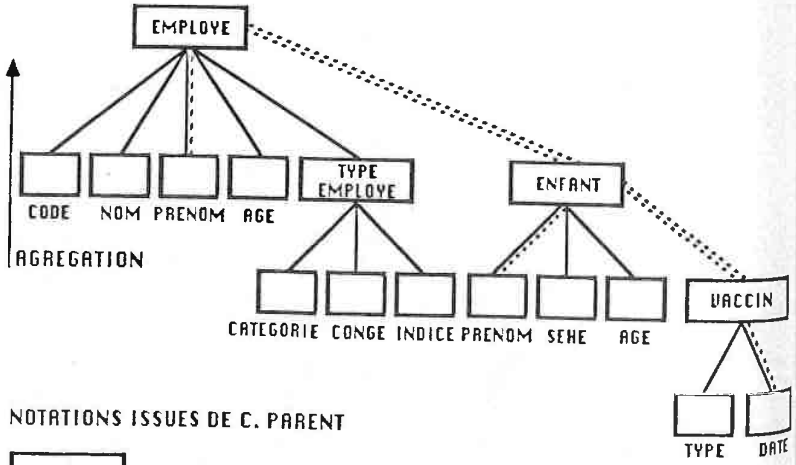
Un attribut, A, est un triplet : A = (obj, str, rea)

tel que :

obj (A) est l'objet dont A est attribut (TA, TE ou attribut complexe)

str (A) est la structure de A

rea (A) est la réalisation de l'attribut, sa valeur.



NOTATIONS ISSUES DE C. PARENT

R	Représente une entité, un attributsimple ou complexe
———	Dénote une cardinalité 1 : 1
- - - - - - - - -	Dénote une cardinalité 1 : n
- - - - - - - - -	Dénote une cardinalité 0 : n
- - - - - - - - -	Dénote une cardinalité 0 : 1

UNE REPRESENTATION GRAPHIQUE DE L'AGREGATION

2-1-1-2-3 Présentation de l'algèbre E R C.

Toute requête sur le schéma conceptuel de données est exprimée par une combinaison d'opérateurs qui forment une expression algèbrique.

L'algèbre E R C. est une algèbre d'entité : ses opérateurs ont pour opérandes des T E s et produisent en resultat un nouveau T E n'existant pas dans le schéma conceptuel de données, nouveau T E qui peut servir à sont tour d'opérande.

Les opérateurs dont les définitions s'appuient sur les régles d'**héritage** et de **compatibilité** (PARENT, 87) sont

- des opérateurs n-aires : la r-jointure, le produit, la différence, l'union
- des opérateurs unaires : la selection, la projection, la réduction, la compression, le renomage et la simplification.

Ce modèle E R C est donc doté d'un calcul (ce qui n'est pas le cas pour le modèle E-A) et de plus cette algèbre a été démontrée complète.

Remarque :

Les concepts de base du modèle ERC supportent le mécanisme d'agrégation sur les attributs d'une entité, en revanche ils ne supportent pas dans la publication de 1987 le mécanisme de généralisation, son implantation dans le modèle est en cours d'étude actuellement (Software Process Workshop,88). Aussi pour des raisons de clarté, il est souhaitable qu'un modèle séparé (MS) puisse prendre en compte ces types d'abstraction.

2-2 LE MODELE GENERIQUE PROPOSE : LES CONCEPTS

2-2-1 Approche générique du modèle

Dans toute approche d'un programme en vue de son amélioration, l'un des aspects les plus importants est sa **structure** en termes d'**unités conceptuelles.**

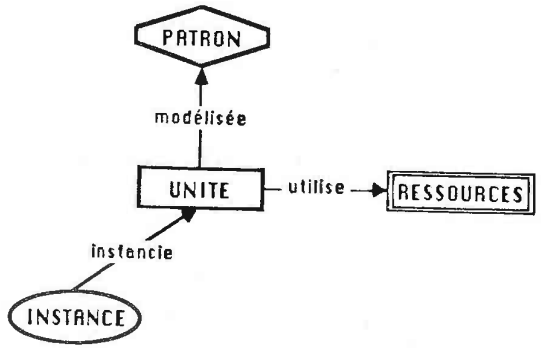
Ces unités conceptuelles qui sont les constituants de base dans la construction des programmes sont reliées entre elles par des relations de type **structurel** (membre de, précédé de, contient,...) et/ou **fonctionnel** (appel, créé, modifié,...)

Chaque unité conceptuelle peut posséder des **instances** et constituer un objet (une classe) dans des processus taxonomiques.

La métaconnaissance nécessaire à la définition de chaque unité peut constituer ce qu'on appelle le **patron** de l'unité, sorte de modélisation de l'unité (type de l'unité).

Toute information rattachée à une unité conceptuelle permettant une meilleure définition et compréhension de son contenu constitue un élément des **ressources utilisées** par l'unité.

Un modèle générique de données dans un environnement quelconque est alors le suivant :



L'intérêt d'un tel modèle (R;Molsching-Pitrik CSL-SDKS) réside dans sa simplicité et donc sa généralité.

Proposé pour la génération de modèles de données lors de processus de développement et de maintenance de logiciel, il est tout aussi intéressant au niveau de l'amélioration de programmes existants.

La recherche d'un niveau unique d'expression des concepts de structuration sous-tendus par les langages et les méthodes quelque soit le système de traitement et de gestion de l'information concerné est

une démarche devenue essentielle devant la multiplicité des étapes et des outils mis à la disposition d'un nombre toujours plus restreint d'informaticiens dans le cadre des nouveaux ateliers.

Uniformiser par exemple les concepts d'agrégation (au niveau des bases de données),; de structure d'article, de séquence d'article (au niveau des systèmes classiques), de relation "partie de" (au niveau des bases de connaissances)) ne peut que faciliter l'intégration des approches et quoi de plus utile pour un système d'amélioration.

2-2-2 La structure des composants du modèle générique de données

2-2-2-1 La structure des unités

Une Unité est identifié par son **nom** et consiste en deux parties l'**interface** et le **corps.**

1) L'interface est composée de trois sections :

- l'unité de composition des sections est le **trait.**

La notion de trait est proche de celle d'attribut dans les systèmes Base de données (relationnels) mais de conception plus générale ; alors qu'un attribut sert à décrire une propriété d'une relation un trait peut contenir une procédure, une règle de production, une contrainte d'intégrité,...

- la **section administrative** est constituée de **traits** caractérisant l'**auteur,** la **date** (de création, de modification), la **version,** la **phase,** l'**état** (complétude ou non) de l'unité.
- la **section structurelle,** chargée de la définition de toutes les relations structurelles de l'unité avec son environnement (autres unités, patrons, ressources, instances)

Elle comporte une liste de traits prédéfinis :

- **antécédents** : liste des unités la précédant dans les phases antérieures, ce qui permet une analyse de la suite des transitions et constitue une aide aux transformations comme la fusion ou la recopie d'unités.
- **patron** : trait qui fait référence à la métaconnaissance concernant l'unité décrite ; pour une unité décrivant un type abstrait le patron associé spécifie un trait représentation et un trait opérations.
- **sur-unité (sous-unité)** sont des traits permettant de construire des hiérarchies au sens généralisation /spécialisation et avec elles les mécanismes d'**héritage** parmi les unités.

- **membre de** : trait qui permet le rattachement de l'unité à un ensemble la contenant, au sens d'un critère quelconque considéré comme important dans un but donné.
- **utilise** ; trait qui permet d'énumérer les noms des ressources utilisées dans l'unité ; ressources non fournies par l'unité, ni héritées par elle ; moyen aussi d'établir des dépendances entre unités.
- **perspectives** : trait qui permet d'énumérer différentes vues de l'unité en masquant certains aspects à l'utilisateur.

- la section **traits spécifiques** : cette section regroupe un ensemble de traits spécifiant des ressources visibles pour l'environnement (ressources qui peuvent être nommées dans les traits **"utilise"** d'autres unités). Ces traits définis par l'utilisateur peuvent appartenir à au moins une catégorie de traits (par exemple membre, propriété, procédure, donnée, méthode) en fonction du rôle joué par les constituants du trait ou la portée d'un trait.

2) Le corps est essentiellement destiné à raffiner la spécification donnée dans l'interface et son contenu est directement lié à l'application concernée.

2-2-2-2 Les patrons et les instances

Un patron détient la métaconnaissance sur les contenus des unités, comme la définition de l'unité, le nombre et les catégories de traits fournis, des prédéfinitions de catégories de traits et de catégories de valeurs de traits.

Chaque instance est constituée essentiellement de son texte et d'un trait prédéfini **"instance-de"** qui la relie à l'unité dont elle est une instanciation.

Quant aux ressources, elles sont reliées à certaines unités, sont d'un certain type (texte, fichier, outils, graphe,...) et ont une certaine représentation qui constitue leurs corps.

2-2-3 Un modèle générique de données pour l'amélioration de programmes

Partant du modèle très général précédent on peut proposer un modèle générique plus directement destiné à représenter des programmes dans un contexte d'amélioration de ces derniers.

Les unités introduites doivent être indépendantes du langage de programmation, des types de schémas de programmes utilisables, de la méthode de conception et de toute convention de programmation.

2-2-3-1 Les unités du modèle de programme

En réalité, l'objet global considéré est une version d'un programme à un moment donné au cours du cheminement qui conduit du programme source au programme cible.

Nous proposons comme patron les unités génériques de base suivantes :

U-PROG	unité de programmation identifiable compilable exécutable (tout programme principal).
U-MOD	unité de programmation identifiable, compilable séparément, exécutable grâce à un "appel" dans une autre unité de programmation (sous programme, procédure, fonction, module).
U-LOG	partie d'unité de programmation, éventuellement activable par passage du contrôle), ensemble d'instructions délimité par un point d'entrée E et un point de sortie S tel qu'il existe au moins un chemin de E à S, ou encore ensemble de lignes d'une unité de programmation correspondant à la description d'une entité logique (déclaration de fichier logique).
U-PHY	morceau connexe du texte d'une unité de programmation délimitable physiquement à l'aide, par exemple, de numéros de lignes, d'étiquettes, de séparateurs propres à un langage ou de mots clés, élément éventuel d'une bibliothèque.
U-BLOC	suite d'instructions contrôlée par une structure de contrôle (partie ALORS ou SINON d'une instruction conditionnelle, corps d'une instruction itérative).
U-INST	une quelconque instruction d'un langage de programmation, assimilable à une affectation généralisée.
U-VAR	unité désignant un objet quelconque valorisable.

2-2-3-2 Les traits principaux d'un modèle de programme

En faisant référence au modèle précédent, parmi les traits qui nous paraissent intéressants, et ce ne sont pas ceux de la section administrative, citons ici, en particulier, les types de relations que l'on peut envisager entre les patrons dans la section structurelle.

Une bonne approche de l'ensemble des relations classiques entre composants d'un système logiciel est celle de B.MEYER (MEY-SKB) qui propose une classification des relations les plus importantes. Ces relations pourraient être prédéfinies et constituer une bibliothèque.

• **relations entre objets de types différents :**

a **contient** b

L'objet b est un constituant de a , par exemple un U-MOD; a peut contenir un U-LOG ; b. La relation inverse peut être notée

partie de ;

a **contient** b <==> b **partie de** a

exemple :

b est un champ de a dans une structure.

a **modélise** b

L'objet a contient alors une description de ce que b fait, ce qui revient à dire aussi que b est une manière de faire ce qui est décrit par a.

à l'inverse, on note b **instance de** a

exemple :

un type abstrait peut modéliser un paquetage ADA.

Cette relation permet de construire une hiérarchie de spécifications.

• **relations entre objets de même type :**

a **complète** b

Les objets a et b coopèrent à un même but, par exemple les différentes procédures d'un paquetage peuvent coopérer à la définition d'un type.

a **particularise** b

Tout ce qui est décrit par a l'est aussi par b.

La relation inverse est notée b **généralise** a

exemples :

le mécanisme de préfixation de Simula et Smalltalk

une unité module généralise sous-programme

a **fait-référence-à** b

exemples :

a et b sont des U-MOD tels que a **appelle** b

a est un U-MOD et accède à une variable b d'un autre U-MOD (partage de données, structure de bloc,...)

La relation inverse étant notée b **est-référencée-par** a

a **nécessite** b

b est nécessaire à la compréhension, à l'exécution ou à la définition de a.

a **est-déclaré-dans** b

cas des langages à structure de bloc.

a **partage-avec** b

Relation symétrique dans laquelle a **et** b peuvent accéder à une information commune.

• **relations entre U-MOD et U-LOG :**

a **appelle** b

cas classique d'appel d'une procédure par une autre.

a **créé** b

Relation qui existe dans certains langages ou un traitement peut en déclencher un autre (tâches en ADA ou PL/1).

C'est le cas aussi de l'allocation dynamique de données (new en PASCAL).

a **active** b

Relation qui correspond à des "coroutines" ou des processus parallèles (ADA).

a **envoit-à** b

L'objet a passe de l'information à b , soit directement (paramètres) soit indirectement (zone commune modifiée).

La relation inverse est notée b **reçoit de** a.

• **relations entre divers objets d'un programme :**

Ces relations couvrent plus directement les besoins en matière d'analyse statique du programme.

a **suit** b

L'exécution de l'instruction a peut être suivie immédiatement de celle de l'instruction b (flot de contrôle).

a accède-à b

La valeur de l'objet b est nécessaire à l'exécution de a

exemple :

si a est une affectation, elle nécessite l'accès à tous les objets,b,du membre droit.

a modifie b

La valeur de b peut être modifiée pendant l'exécution de a.

exemple :

si a est une instruction d'affectation, elle modifie la variable b membre gauche de cette affectation.

a nécessite valeur de b

La valeur de a peut être modifiée par un calcul utilisant la valeur de b.

exemple :

a membre gauche et b dans le membre droit d'une affectation.

Remarquè :

Une approche plus sémantique au niveau du modèle nécessite (B.Meyer SKB) l'expression de **contraintes** (conditions logiques sur les relations et attributs) à satisfaire pour la définition et la conservation de la cohésion logique de la base.Ces contraintes de dépendance ou d'intégrité contribuent à la définition des états cohérents du modèle.

Ces contraintes, auxquelles peuvent être associées des **actions** en cas de violation, ainsi introduites au niveau du modèle, allègent la programmation et sont plus fiables : c'est d'ailleurs la façon actuelle d'assurer l'intégrité d'une base dans les récents SGBD relationnels (SYBASE).

Dans les travaux du groupe Génie Logiciel concernant le développement et la gestion de logiciels, on retrouve ce doublet **contraintes-actions** sous la forme de concepts plus généraux que sont **caractéristique** et **expression.**

Ces deux concepts englobent les aspects statiques et dynamiques du processus de développement de logiciel et permettent sous forme d'expressions logiques de décrire les états sains ou particuliers du système (Bénali K.,89).

2-2-3-3 Instanciation du modèle générique de données

Un modèle de données spécifique à un langage, une méthode et un type de problème dans un quelconque environnement résulte alors d'une instanciation du modèle générique de données pour l'amélioration.

Cette instanciation résulte de la mise en oeuvre d'un opérateur NOUV un peu à la manière de NEW d'ADA ou de l'INSERT des SGBD mais aussi de l'utilisation des relations prédéfinies sous forme de traits des unités, la généricité pouvant s'exprimer en partie à l'aide des abstractions comme la généralisation, la spécialisation, l'agrégation, la classification.

Aussi dans un environnement défini par le langage COBOL, et un système de gestion de fichiers on a par exemple :

sous-programme	NOUV U-MOD
procédure	NOUV U-LOG
fich-descr	NOUV U-PHY

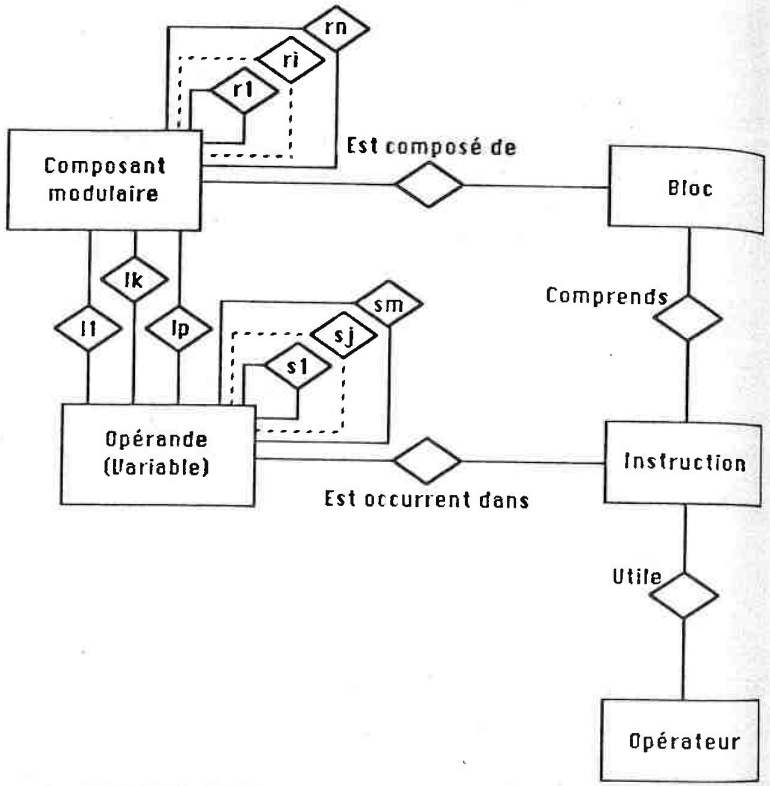
l'U-VAR, variable peut avoir pour valeur du trait sous-unité tableau report, fichier ou simple.

Un exemple plus complet est développé dans le p II ch 6 et en annexe.

2-2-4 Représentation graphique d'un modèle de données

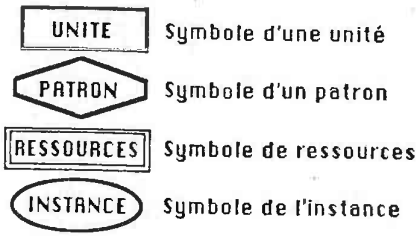
Dans un contexte d'amélioration de programme, où l'homme est un acteur essentiel dans le processus de transformation, il est nécessaire d'envisager la dualité des représentations textuelle et graphique avec maintien de la cohérence logique entre les deux représentations et possibilités de passage immédiat de l'une à l'autre.

Ainsi le schéma conceptuel des données de l'amélioration prend en compte différents types d'unités constituant un programme, les types de relations ainsi que les attributs correspondants. Un exemple de profil de schéma, où pour des raisons de clarté ne figurent pas les propriétés éventuellement complexes des unités, est donné ci-dessous.



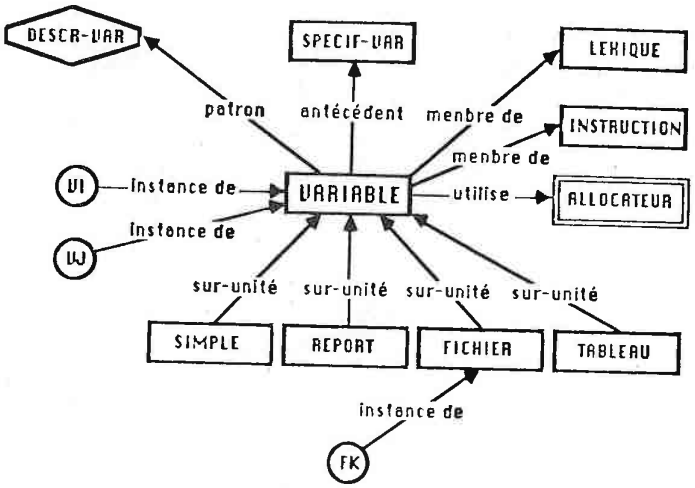
UN PROFIL DE SCHEMA DE DONNEES DE L'AMELIORATION

Le modèle de données sorte de multigraphe n'échappant pas à la règle, sa représentation graphique peut utiliser un symbolisme classique :



Les arcs reliant ces symboles sont étiquetés du nom du trait caractérisant la relation.

La figure ci-dessous représente par exemple partiellement l'unité variable.



MORCEAU DE GRAPHE AUTOUR DE L'UNITE VARIABLE

L'intérêt d'une telle représentation n'est pas tant de mettre en évidence le multigraphe que d'offrir à l'utilisateur l'accès à n'importe quel sous graphe caractérisé par la prise en compte d'une seule relation.

Ainsi le sous graphe constitué des noeuds reliés par des arcs étiquetés sur-unité donne par transitivité la hiérarchie de généralisation/spécialisation entre unités.

Dans une démarche de type Reverse Engineering, le sous-graphe des arcs étiquetés antécédent permet de parcourir les niveaux (abstractions) de spécifications.

2-3 CONCLUSION

Le modèle générique de représentation de logiciel proposé prend appui sur la notion de modèle sémantique. Sa simplicité et sa généralité permettent de réaliser toute instanciation correspondant à un contexte précis d'amélioration de programmes.

Mais il s'agit ici de la modélisation d'un système causal et ce ne sont pas les mêmes concepts, comme nous allons le voir dans le p II ch 3, qui peuvent permettre la modélisation du processus d'amélioration essentiellement indéterministe.

II-3 UN MODELE GENERIQUE DE PROCESSUS D'AMELIORATION :

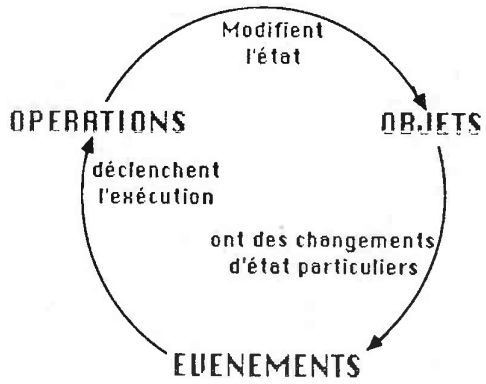
3-1 INTRODUCTION

On aborde alors la modélisation de la **dynamique** du système d'amélioration.

Au-delà de l'aspect **statique** que constitue le modèle de données de nombreux travaux ont abordé dans ce cadre la dynamique des **systèmes d'information** (SI). Une première modélisation de cette dynamique est la prise en compte des **contraintes d'intégrité** qui permettent de valider les changements d'état du système d'information (voir par exemple HOW 86 (Bénali).

Une modélisation plus formelle et complète de cette dynamique des SI est réalisée dans des projets comme REMORA (FOU 82) (ROL 88) ou MERISE (TAR 84). Ce type de modélisation permet une expression de la dynamique des traitements dans les SI à l'aide de concepts spécifiques symbolisant les stimulis auxquels peut être soumis un SI et les réactions qui peuvent en découler. Pour le projet MERISE, il s'agit des **évènements, opérations, résultats** et **synchronisations**, pour le projet REMORA de **c-évènements, c-opérations, c-objets, c-déclenches.**

Ces modèles (MERISE, REMORA, ...) sont bien adaptés à la modélisation de la dynamique de systèmes causals comme ceux que l'on rencontre dans les entreprises. Les interactions avec le monde extérieur sont considérées comme des évènements externes. Le système d'information dynamique va réagir à ces évènements en exécutant des opérations qui pourront , à leur tour, être la cause d'autres évènements (internes).



La modélisation de la dynamique associée au processus de développement non déterministe est plus récente.

Une des premières approches, critiquée pour le côté "prédéfinition" qu'elle offre d'un processus de développement plutôt "créatif", est l'approche "Process Programming" dont l'idée de base est l'utilisation d'un langage de programmation pour décrire un processus de développement.

Une autre approche est celle qui consiste à utiliser un système à base de règles pour la modélisation et l'exécution de processus de développement (projet MARVEL, DSKBEE).

Dans cette approche un modèle de développement de logiciel est décrit par un ensemble de règles, du type précondition ---> postcondition, déduites d'un ensemble d'activités schématisées par le triplet : précondition - activité - postcondition.

L'utilisateur, dans sa demande d'exécution d'une activité, est alors assisté par le système grâce à un moteur d'inférence en avant qui fonctionne implicitement mais aussi , dans le cas où la précondition n'est pas vérifiée, grâce à un moteur à chaînage arrière, sur les règles précondition - postcondition, pour déterminer et déclencher une séquence d'activités rendant possible l'exécution de l'activité demandée.

L'utilisateur a même la possibilité d'activer et de désactiver des stratégies (ensembles de règles) différentes pour un même ensemble d'activités.

Dans le cadre du projet ALF l'équipe de Génie Logiciel (BENALI K.,89) a défini un modèle de développement et de gestion assité de logiciel, le MASP (Model for Assisted Software Process) qui décrit la façon dont les activités (soltware process) doivent être exécutées et l'organisation des objets manipulés par ces activités

C'est sur ce dernier modèle que nous nous appuyons, dans notre travail, pour modéliser la dynamique du système d'amélioration..

3-2 LE PROCESSUS D'AMELIORATION

3-2-1 Finalités de l'amélioration

Le but du processus d'amélioration d'un programme, partant d'une version initiale d'un programme est d'aboutir à une version finale satisfaisant à un certain nombre de critères et/ou d'objectifs définis par l'utilisateur parmi lesquels on peut citer dans une liste non exhaustive :

- améliorer la modularité
- assainir le texte source (éliminer les instructions et variables inutiles).
- éliminer certaines instructions ou structures de contrôle
- aboutir à une traduction n'utilisant que des schémas de base d'un type déterminé.
- changer de langage
- documenter le programme
- améliorer la qualité du programme par rapport aux mesures de complexité, caractéristiques de certains critères ou facteurs de qualité.
- passer d'un SGF à un SGBD ou d'un SGBD hiérarchique ou réseau à un SGDB de type relationnel (changer d'environnement).

La donnée du processus est un programme dont le niveau est jugé insuffisant de certains points de vue (lisibilité, modularité, maintenabilité,...) par l'utilisateur de façon intuitive ou objective par exemple à l'aide de mesures de complexité.

Alors que le processus de développement correspond à une application de l'ensemble des énoncés dans l'ensemble des programmes, le processus d'amélioration est une application de l'ensemble des programmes sur lui-même.

Mais si le processus de développement est généralement "descendant" en matière de niveaux de spécifications allant d'un énoncé informel à un programme réalisé dans un certain langage pour un environnement donné, le processus d'amélioration, quant à lui, est d'un niveau de complexité qui dépend essentiellement des objectifs assignés.

3-2-2 Différents niveaux de système d'amélioration

Parmi les grandes classes de processus d'amélioration, on peut citer :

- L'amélioration de la structure du programme.

  Il s'agit alors essentiellement de mettre en oeuvre des transformations syntaxiques ou sémantiques du programme. Les objectifs sont alors par exemple;

  passer d'un type de schéma de programme à un autre (éliminer les instructions de branchement), assainir le texte (éliminer le code mort), optimiser (éliminer les invariants, les redondances, fusionner des itérations).

  Ce type d'amélioration nécessite habituellement différentes représentations du programme (graphes, matrices) sur lesquelles peuvent agir des opérateurs de transformations et des outils de l'atelier concerné.

- Le changement de langage de programmation.

  Dans le cas où le langage source et le langage cible sont de même niveau, il s'agit pratiquement d'une traduction (trans codification).

  Par contre, et c'est le cas le plus fréquent si le langage cible est de plus haut niveau que le langage source, il devient nécessaire de modifier la structure des traitements (introduction par exemple des concepts de bloc, de procédure, de package) et éventuellement la structure des données (passage du concept de fichier au concept d'objet, mise en évidence de types abstraits).

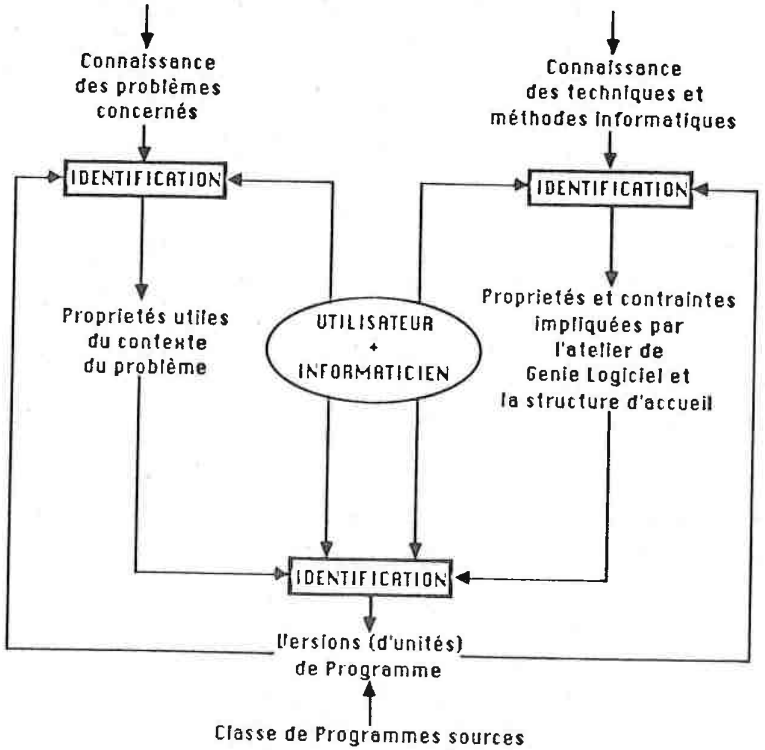
- Le changement d'environnement.

  Lorsqu'il ne s'agit que de la modification de quelques caractéristiques de l'environnement d'exploitation, les seuls moyens d'adaptation du langage utilisant en général la séparation des propriétés "abstraites" et "concrètes" d'un programme (pour le langage ADA, package SYSTEM, pragmas, spécifications de représentation de données ou de description de traitements) suffisent en général à résoudre le problème.

  Mais la modification du système de gestion des données peut devenir un difficile problème et nécessiter de remonter dans les niveaux de spécification jusqu'à la constitution éventuelle d'un modèle conceptuel, il s'agit alors de spécification à postériori d'une application (Reverse Engineering).

  Cette spécification à postériori exige un code source d'une certaine qualité (nécessité de restructurer, modulariser,...), différentes représentations du programme suffisamment conviviales (graphes et

diagrammes de flux) (projet GCARE de Siemens AG), et la collaboration étroite entre l'utilisateur (connaissance du problème) et l'informaticien (connaissance des techniques et méthodes informatiques) pour procéder aux abstractions nécessaires.



SCHEMA GENERAL D'AMELIORATION DE LOGICIEL

Citons dans ce domaine des projets et outils comme PROMPTER (création pas à pas de façon ascendante de la description abstraite d'un programme dans une approche orientée objet), SNEED sur PL/1 (étude de la finalité de gros programmes) et PROUST sur PASCAL (aide à la compréhension de programmes), un atelier de génie logiciel sur PC comme IEW utilisé par exemple pour le passage d'un modèle réseau (IDS2) à un SGBD relationnel avec comme étape charnière l'élaboration d'un schéma E/A et sa normalisation.

3-2-3 Définition du modèle de processus d'amélioration

A l'image de tout traitement de logiciel, le processus d'amélioration nécessite une modélisation dont les conséquences fondamentales essentielles sont :

1) Améliorer la **communication** entre les différents utilisateurs concernés par le processus, en facilitant la définition donc la compréhension de l'ensemble du processus.

2) Faciliter la **reprise** d'un processus en cours avec la notion d'état courant.

3) Faciliter l'**apprentissage** et l'**évolution** du processus d'amélioration.

4) Faciliter la **gestion** de l'amélioration en attachant à chaque **état caractéristique** du processus des critères de définition (par exemple objets résultats) et des mesures.

Le modèle doit être assez souple pour :

- être compatible avec l'actuelle manière d'améliorer les programmes sur le site concerné (pour une période transitoire éventuelle).
- permettre de travailler à un niveau quelconque de spécification (dans un contexte de REVERSE ENGINEERING), ou sur une quelconque forme de représentation (graphes, diagrammes, texte).
- permettre un caractère dynamique à la définition d'un processus (éviter les prédéfinitions rigides).
- pour être indépendant de l'environnement de programmation (système de gestion de données, langages, méthodes, conventions de programmation,...)

Toutes ces exigences nous conduisent à introduire différents concepts essentiels comme ceux de stratégie, méthode, activité, arbre d'amélioration.

3-2-3-1 Les principaux concepts du modèle

Nous abordons successivement les aspects statiques puis dynamiques du processus d'amélioration pour introduire des notions inspirées dans leurs définitions des travaux du groupe Génie Logiciel .

3-2-3-1-1 Aspects statiques : notions d'**état** et de **version**

Améliorer un programme P, instanciation d'un modèle de représentation M, c'est construire un programme Pf dernier élément d'une suite de représentations intermédiaires de ce programme.

Ce programme final Pf est représenté par un modèle Mf qui symbolise les objectifs à atteindre lors de la mise en oeuvre du processus d'amélioration.

Chaque représentation intermédiaire correspondant pour le programme à un certain état de représentation à conserver à la demande du système ou de l'utilisateur est une **version** du programme.

Une version est donc un état stable du programme, sachant qu'un **état** du programme est une représentation intermédiaire du programme aboutissement d'une **opération** de modification portant sur un autre état.

Un état du programme est dit **stable** si la représentation associée est cohérente et constitue le résultat de la mise en oeuvre d'une **activité** ; dans le cas contraire, l'état est dit instable et ne peut être mémorisé durablement dans la base de données associée (protection).

3-2-3-1-2 Aspects dynamiques : notions d'opération, d'activité, de méthode, de stratégie.

L'unité d'intervention minimum pour l'homme à l'aide des outils de l'atelier ou pour le système est l'opération.

Une **opération** est donc une étape élémentaire de l'amélioration d'un programme. Aucune d'entre elles n'est la composition d'autres opérations (indépendance).

Les opérations permettent d'obtenir par composition les **activités** qui servent à décrire les étapes logiques d'amélioration de programmes et ne sont qu'un des éléments d'une méthode d'amélioration.

Une **méthode** est un ensemble :

- d'activités
- de modèles de représentation de l'information
- de modèles d'enchainement d'outils
- de règles ordonnant les actions à mettre en œuvre

Les actions peuvent être des opérations, des activités, ou des méthodes

Une méthode est caractérisée par des objectifs globaux ou partiels symbolisés par une version du programme ayant certaines propriétés , devant respecter certaines **contraintes.**

Sachant que plusieurs chemins peuvent mener à une version particulière, la méthode est là pour guider l'amélioration et assurer que l'état obtenu a un sens.

Chaque stratégie définie par l'utilisateur est alors un cadre pour la définition et l'application d'un enchaînement de méthodes, enchaînement éventuellement réduit à l'application d'une seule.
En pratique, la notion de méthode est une notion récursive : une méthode est une combinaison logique de méthodes s'intéressant chacune à un problème spécifique ; ainsi dans un processus d'amélioration d'un certain niveau, il y aura combinaison de méthodes s'intéressant aux différentes phases de RE et de "redéveloppement" du logiciel.

Une **stratégie** s'exprime en termes d'objectif global, de niveau de système d'amélioration et de conduite générale du processus.
Le suivi du déroulement d'un processus d'amélioration peut être matérialisé par un **arbre d'amélioration** (directement inspiré de l'arbre d'analyse d'Anna GRAM (Bou-RPP)

L'arbre d'amélioration est défini par les conventions suivantes :

(1) La **racine** de l'arbre représente l'état du programme à améliorer ; il lui est associé la représentation du programme en terme d'instanciation du modèle correspondant (modèle initial).

(2) Chaque **noeud intermédiaire** représente une étape de l'amélioration : il lui est associé un élément de représentation du programme ou d'une partie du programme correspondant à un état stable, éventuellement à une version.
En fonction des opérations que l'on choisit d'appliquer à un noeud intermédiaire, ce dernier est relié à un ou plusieurs noeuds fils. Chacun de ceux-ci correspond à une solution envisagée à ce niveau de l'amélioration.

(3) Les **feuilles** de l'arbre correspondent pour les unes à des versions du programme abandonnées soit par manque d'intérêt (par exemple pas de gain au niveau des mesures de complexité) soit du fait de l'impossibilité de poursuivre dans la même direction (par exemple niveau de restructuration insuffisant pour permettre de modulariser) pour les autres à des versions solutions.
Les différents constituants de l'instanciation réalisant le programme correspondant à une feuille sont répartis sur la séquence des noeuds conduisant de la racine à cette feuille.

L'arbre d'amélioration est un ensemble structuré de textes, de graphes, de diagrammes produits par la mise en oeuvre du processus d'amélioration. Ses noeuds sont étiquetés par les noms d'opérations ou activités, ses schémas sont qualifiés par les méthodes appliquées.

3-2-4 Eléments de présentation et de mise en oeuvre du modèle.

On aborde successivement les éléments d'une linéarisation du modèle puis les concepts et mécanismes de la mise en oeuvre.

3-2-4-1 Linéarisation du modèle

Le but est ici de proposer les caractéristiques principales des différents composants du modèle de processus d'amélioration à savoir les opérations, les activités, les méthodes.

- les **opérations**

Dans le processus de transformation du programme, l'opération est une étape élémentaire intéressant tout ou partie du programme et correspondant à la transition entre deux états non nécessairement stables étant caractérisée par la satisfaction des **contraintes** attachées à cet état. Comme nous le verrons au niveau de la mise en oeuvre du modèle, ces contraintes sont impliquées d'une part dans des mécanismes de déclenchement d'actions de type message ou de récupération d'erreurs (**triggers**), d'autre part dans le processus de gestion des taches d'amélioration (**règles**).

Une opération peut être décrite par :

1) une **définition** en clair
2) **antécédent** état pris en compte par l'opération
3) **conséquent** état résultant de la mise en oeuvre de l'opération
4) **condition** d'application de l'opération
5) **contextes** éventuels de mise en oeuvre
   précède : liste d'opérations
   succède à : liste d'opérations
   dans : liste d'activités
6) **mesures** de qualité (complexité) significatives
7) **outils** nécessaires à la mise en oeuvre de l'opération

• Les **activités**

Il s'agit alors d'une étape logique du processus d'amélioration aboutissant si non à une version du programme du moins à un état stable.

Une activité peut être décrite par :

1) une **définition**	en clair
2) **antécédent**	état stable (ou version) pris en compte
3) **conséquent**	état stable (ou version) résultat
4) **condition**	d'application de l'activité
5) **schéma**	expression où les opérandes sont des noms d'opérations et les opérateurs **et/ou** avec le parenthèsage.
6) **mesures**	de qualité significatives au niveau activité.

• Les **méthodes**

Dans notre contexte d'amélioration de programme, une méthode peut être décrite par :

1) la **définition formelle** ou technique d'une activité ou d'un processus

2) la **liste des activités** et/ou opérations englobées.

3) les **modèles de données**, instanciations du modèle générique de données vu précédemment.

4) des **modèles d'enchainement** d'activités et d'opérations: un modèle d'enchainement est un véritable schéma à la manière des schémas de programmes.

Un modèle est composé dans le cas le plus simple à l'aide de règles de production qui peuvent être :

• **impératives** : il s'agit d'un "**bloc**" d'opérations à exécuter séquentiellement.

• **conditionnelles** : de la forme

si (condition) **alors** liste-d'opérations

Les conditions de déclenchement sont des expression ou équations logiques portant sur les traits des unités concernées comportant des contraintes caractéristiques d'états.

• **itératives** : de la forme

tant que (condition) **répéter** opération

Dans ce cas, le résultat de l'opération est nécessairement du même type que sa donnée et donc l'opération est exécutée un certain nombre de fois sur un ou plusieurs éléments du programme.

5) La liste des outils ou **modèle d'enchainement** des outils (de l'atelier) concernés par les opérations en jeu.

6) **Mesures** : de qualité significatives au niveau d'une méthode.

Remarque :

Dans certains modèles le concept de méthode n'englobe pas celui de règle.

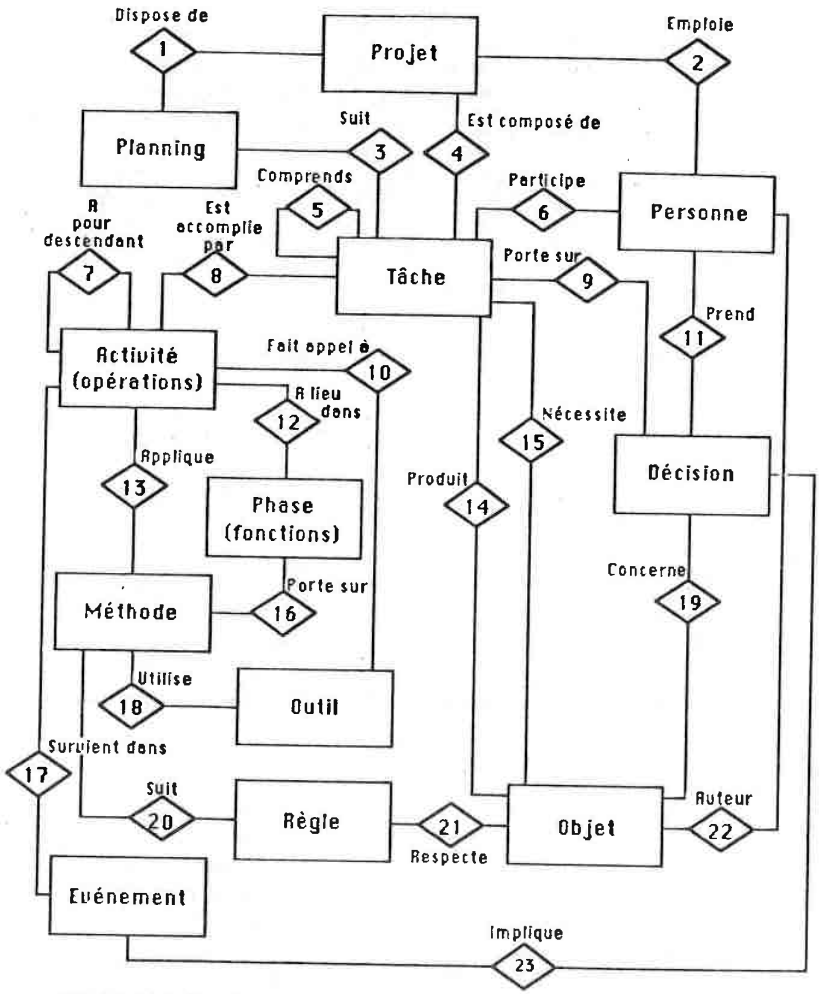
Dans la littérature actuelle, nombreux sont les articles et ouvrages abordant la modélisation du traitement de logiciel; citons au passage H.TARDIEU (TARD 86) où l'on trouve une expertise d'un Atelier de Génie Logiciel en termes d'entités et d'associations, ainsi que Takuya Katayama (KAT 81, KAT 89) où l'on trouve un modèle fonctionnel hiérarchique de traitement de logiciel avec les notions de décompositions d'activités et d'arbre d'exécution ; citons enfin W.S. HUMPHREY et KELLNER (HUM 89), ainsi que dans le cadre du projet ALF, DEITERS, GRÜHN et SCHÄFER (DEIT 89).

3-3 LA MISE EN OEUVRE DU PROCESSUS D'AMÉLIORATION.

Dans un contexte d'apparition d'ateliers et d'environnements intégrés de génie logiciel où des outils, comme les analyseurs, partagent des données avec d'autres outils de l'environnement à travers une base de données (ou base d'objets) commune, l'identification d'un profil de schéma conceptuel, prenant en compte, d'une part, les données du projet d'amélioration, et d'autre part, les données nécessaires pour effectuer des évaluations relatives à la qualité, nous semble d'un intérêt significatif.

Le profil exprimé à l'aide d'un modèle ERC restant suffisamment général, dans le choix des types d'entités, types de relations et attributs, pour être réutilisable dans un quelconque environnement d'amélioration, reprenant ainsi les travaux de (PENEDO 87) et du projet REQUEST (projet ESPRIT) (DALE 88) concernant la définition des schémas conceptuels généraux des bases de données pour la gestion, l'observation et l'évaluation du développement de logiciels.

Les ensembles d'objets, d'activités, d'opérations, de méthodes et / ou techniques, de règles de personnes sont inclus dans l'ensemble des types d'entités du schéma (figure suivante). Mais on trouve également le concept de tâche lié à l'organisation et aux responsabilités (suivi du projet, gestion des versions, assurance du contrôle de qualité,...)



PROFIL DE L'AMELIORATION : une instanciation du processus

Les objets de ce schéma sont les unités génériques du modèle de données précédent.

Une **phase** correspond à une période de mise en oeuvre du projet et se compose d'une ou plusieurs fonctions d'amélioration. Chaque **fonction** est composée **d'activités** et / ou de **sous-fonctions** et au niveau terminal **d'opérations**.

Ces fonctions et ces opérations peuvent correspondre à tout ou partie d'une fonction présentée dans la première partie de ce mémoire.

L'opération correspond à l'unité d'intervention minimum pour l'homme ou pour le système à l'aide des outils de l'atelier ; c'est une fonction primaire sur les objets.

A un instant donné du déroulement du processus d'amélioration le transformateur dispose d'un état de l'objet en cours d'amélioration caractérisé, d'une part par le chemin dans l'arbre d'amélioration exécuté allant de la racine au noeud correspondand à l'état de l'objet, d'autre part, par l'instanciation schématique ou linéaire du modèle de données associé à cet instant à l'objet et enfin du fait de la stratégie et de la tactique en cours (symbolisées par l'arbre d'amélioration prévisionnel) d'un ensemble de règles applicables.

C'est une définition qui correspond à l'utilisation d'un moteur d'inférence.

Le système peut déduire :

- un ensemble d'opérations qui **doivent** être réalisées, celles qui correspondent à des règles dont les conditions sont conséquemment vraies.
- un ensemble d'opérations qui **peuvent** être réalisées;
  - celles dont la condition d'application est vraie.
  - celles qui correspondent à la méthode en cours d'application en conformité avec l'ordonnancement.

Le processus consiste au total en un mécanisme de transitions d'états, états dont la non satisfaction des contraintes peut provoquer la mise en oeuvre de **triggers** (déclencheurs de messages ou de mécanismes de récupérations d'erreurs, règles à actions spécifiques.

L'utilisateur contrôle le déroulement du processus en conformité avec la stratégie préalablement définie ; il choisit les méthodes et activités à appliquer compte tenu des grandes lignes de la stratégie, des objectifs partiels à l'instant et des opérations applicables à chaque instant.

Une image de la progression dans l'amélioration lui est disponible sous la forme de l'arbre d'amélioration; cet arbre met en évidence parmi les états stables (les états non stables étant visualisables dans des fenêtres spécifiques) les différentes versions dérivées successives du programme et permet d'effectuer des choix concernant des versions alternatives avec entre autres éléments d'appréciation les valeurs fournies par les mesures de complexité élaborées.

La mise en oeuvre d'un tel processus sur un parc "homogène" de programmes (même langage de programmation, même méthode de programmation, même environnement,...) nécessite une phase d'initialisation.

Au cours de cette phase d'initialisation, il peut être nécessaire :

- de générer les modèles de programme source et cible prenant en compte les langages, méthodes et conventions de programmation, les types de schémas de programmes.
- d'élaborer des règles prédéfinies, par exemple, par des spécialistes des environnements, langages et méthodes respectivement sources et cibles.
  Règles prédéfinies qui sont entrées dans le système et constituent un fond commun constamment alimenté par les utilisateurs.
- de prédéfinir un ensemble d'opérations, d'activités, voir de méthodes spécifiques à un site d'amélioration, à moins de disposer et d'accepter une version standard de cet ensemble.

3-4 CONCLUSION

Dans un environnement d'amélioration, l'utilisateur doit pouvoir, à chaque instant, accéder à toute donnée caractérisant soit un état du programme, soit une étape dans le déroulement d'un processus d'amélioration, et cela quelque soit le type de cette donnée (texte, graphe, diagramme); mais si la plupart des problèmes sont résolus au niveau conceptuel, la réalisation d'un **système d'amélioration réaliste et efficace** est une tâche encore difficile aujourd'hui, ce que nous confirmons dans le p II ch 4.

II-4 VERS LA REALISATION DU SYSTEME D'AMELIORATIONDE LOGICIEL.

4-1 INTRODUCTION

Dans ce chapitre, nous nous interrogeons sur le type de logiciel qu'il faudrait utiliser pour réaliser un système d'aide à l'amélioration de logiciel (SAAL).

Les fonctionnalités assignées à un SAAL : représenter, gérer, maintenir, retrouver, etc... des objets volumineux à structure complexe suggèrent deux voies possibles pour réaliser un SAAL.

1- Partir d'un SGBD (Système de Gestion de Base des Données).

Un SGBD pourrait servir de point de départ au développement d'un SAAL. Nous avons même envisagé pendant quelque temps la réalisation partielle d'une maquette de SAAL avec un SGBD relationnel.

Mais cette pré-étude a mis en évidence l'inadaptation de ce type de SGBD pour la mise en oeuvre d'un SAAL et nous avons constaté que le type de logiciel qui pourrait servir de base au SAAL est assez proche des **SGBD de 3ème génération** en cours d'étude actuellement.

Nous présentons donc dans le paragraphe 4.2 les limitations des SGBD actuels et les extensions qui sont en cours de développement en montant pour chacune d'elles en quoi elle serait nécessaire à la réalisation d'un SAAL.

2- Partir d'un AGL (Atelier de Génie Logiciel)

Mais si un SGBD de 3ème génération peut constituer un bon point de départ pour un SAAL la spécificité du domaine : **le logiciel** nécessite d'étudier une autre approche pour la réalisation d'un SAAL : les environnements de programmation et en particulier les **Ateliers de Génie Logiciel** (AGL) ce qui est l'objet du paragraphe 4.3

4-2 L'AMELIORATION DE LOGICIELS ET LES NOUVELLES FONCTIONS DES SGBD

4-2-1 Les limitations des SGBD actuels.

Les SGBD commercialisés actuellement et disponibles sur de gros ordinateurs reposent tous sur le modèle relationnel et ont été conçus pour gérer des données de gestion classique à structure assez simple.

Le développement des technologies de stockage optique des informations ainsi que la diminution des temps d'accès à ces mêmes informations permettent maintenant d'envisager le traitement automatique de données beaucoup plus complexes : graphes, images, etc...

Ensuite, les SGBD actuels, s'ils permettent de retrouver des données stockées dans la base, ne sont pas capables de déduire de nouvelles informations à partir de données stockées dans le base et de règles générales connues.

Pour cela, il faut étendre les SGBD actuels en les dotant d'une fonction supplémentaire : une fonction de déduction.

Enfin, les SGBD actuels souffrent d'une certaine lourdeur au niveau des interfaces malgré la fourniture d'outils de développement appelés "Langage de 4ème génération" rares sont les SGBD offrant par exemple une interface graphique.

Ces trois limitations des SGBD relationnels ont entraîné tout un courant de recherches afin de définir une nouvelle génération de SGBD permettant la gestion de données multimédia, la prise en compte de la déduction et offrant des interfaces riches et variées (GARDARIN,86) (PRC,89).

Aussi, pour lever les limitations des SGBD actuels, la future génération de SGBD offrira quatre classes de fonctionnalités : fonctionnalités multimédia, déductibes, dynamiques et d'interfaçage.

4-2-2 Les fonctionnalités multimédia

L'extension des SGBD actuels doit leur permettre de gérer des objets complexes tels que graphes, diagrammes, images (vastes ensembles de cellules), textes, séries statistiques de mesures, etc...

Dans le cas de notre SAAL, les objets à manipuler sont ceux du développement de programmes pour les entreprises.

Parmi les objets de type texte, le transformateur peut souhaiter manipuler

• le dossier de programme dont les constituants sont formalisés à des degrés divers pour le programme.

• des morceaux de code en différentes circontances par exemple :

- pour comprendre le programme à améliorer (APPREHENDER)
- pour passer après transformation d'une représentation schématique à une représentation linéaire (LINEARISER).
- pour apprécier, entre autres qualités, la lisibilité et la clarté des constituants du programme cible : en permettant par exemple différentes traductions linéaires d'une même représentation graphique.

Quant aux objets graphiques (graphes ou diagrammes), le transformateur doit pouvoir non seulement les visualiser pour apprécier par exemple leur complexité ou l'effet d'une transformation, mais aussi intervenir directement à l'écran sur ces objets pour permettre une certaine souplesse vis à vis de transformations prédéfinies systématiques (retouches artisanales de graphes).

Actuellement, la description et la manipulation d'objets complexes sont activement abordés au niveau de la recherche en informatique (GEODE, INRIA), (EXPRIM, NANCY).

Un objet complexe est envisagé soit comme un assemblage quelconque construit à partir de séquences, d'ensembles et d'agrégats soit comme une fonction pour exprimer des liens inter-objets ou certains aspects dynamiques comme les déclencheurs (triggers). (LE MAITRE, 88).

Un objet complexe est de type complexe (structure hiérarchique obtenue par application progressive de constructeurs sur des types de base (GARDARIN, 88)), et peut posséder différentes versions (ou états stables) courante, dérivées (historique de l'objet) ou alternatives (représentant différents choix).

4-2-3 Les fonctionnalités déductives

Dans le cadre de notre SAAL, le transformateur doit pouvoir :

1 Au niveau des données, définir des objets en extension et en intension Dans le second cas, l'utilisateur peut obtenir à la demande l'élaboration d'une représentation d'un objet ou de l'objet résultat d'une transformation appliquée à l'objet initial ou la mise en oeuvre d'opérateurs d'abstraction.

Ainsi dans le passage d'un état à un autre d'un morceau de programme si les transformations appliquées sont formalisables (outils disponibles

aux procédures définies) l'utilisateur peut souhaiter ne conserver qu'en intension l'état final.

Ceci peut aussi constituer un moyen de distinguer un **état** d'un morceau de programme, d'une **version** de ce même morceau. Une version d'un morceau de programme correspondant à un état "permanent" donc intéressant, par exemple, au niveau de l'historique de l'amélioration du programme auquel il appartient.

Lors de la mise à jour de la base de données, des règles sont utiles pour générer des données déduites (voir le prototype BDGEN (NICOLAS,83). Ainsi l'introduction d'une instruction d'un morceau de programme dans la base doit entraîner la création des variables concernées (si elles ne sont pas encore répertoriées) et des relations associées.

Enfin lors des interrogations, les données déduites doivent être manipulables mais non nécessairement présentes dans la base (interface déductive sur SABRE (MADELAINE,84)).

2 Au niveau des traitements, un modèle de processus d'amélioration est un guide de l'amélioration pour atteindre des états stables et en temps que tel il se doit de comporter :

- la description des types d'objets et leurs liens
- la spécification syntaxique (profil des opérateurs) et sémantique (ce que fait l'opérateur) des opérations de chaque type d'objet et le lien entre opérations et outils.
- la définition des états stables du processus d'amélioration essentiellement sous la forme de contraintes.
- un ensemble de règles de production décrivant les opérations à entreprendre en cas de non respect d'une contrainte avec comme objectif d'atteindre un état stable.
- un ensemble d'enchaînements possibles des opérations.

Le système de pilotage et d'assistance à l'amélioration ne doit autoriser que des opérations assurant une transition entre états stables.

La base d'objets doit donc être augmentée des règles décrivant un processus d'amélioration et peut être vue, au moins de ce fait, comme une base de connaissances ; quant au système d'assistance, ce n'est pas autre chose qu'un système à base de connaissances.

4-2-4 Les fonctionnalités dynamiques (projet SHERPA de NANCY) :

Il s'agit là de faciliter la mise en oeuvre de la dynamique dans les bases de données en particulier en ce qui concerne les notions de type évolutif de données et de schémas dynamiques de base de données.

Le retard des études dans ce domaine tient peut être à la simplicité du modèle relationnel qui a longtemps implicité des structures de données stables.

Mais force est de constater aujourd'hui, au niveau de la CAO, de la Bureautique et du Génie Logiciel pour ce qui est de notre processus d'amélioration de logiciel, que si l'on construit des objets on procède ensuite à leur interrogation et à leur modification.

Ainsi il est nécessaire tout au long de l'amélioration d'un programme, de voir évoluer le modèle de représentation de ce dernier de l'état source (version source) à l'état cible (version cible) ce qui s'accomode fort bien des qualités de généricité et d'évolutivité des modèles. Les historiques et les versions sont des supports utiles pour la dynamique mais ne sont que des outils.

Le problème est aussi celui des conséquences de la création et de la modification dynamique des schémas sur :

- l'appartenance des informations à ces schémas
- les opérateurs de modèles sémantiques comme l'agrégation, la généralisation,...

La manipulation dynamique de schémas d'objets donne une dimension nouvelle à la notion d'**héritage** entre les classes d'objets devant l'évolution conséquente de ces classes et de leurs instances.

NGUYEN (NGU 87) et RIEU (RIEU 87) modélisent toute manipulation d'un schéma d'objet par une séquence finie d'opérations de réduction, d'augmentation, de connexion et de produit.

Au niveau du processus d'amélioration, la dynamique est envisagée actuellement (CAUVET C,87) sous l'angle de la description des actions en termes d'abstractions de contrôle (ODIENNE P,87), (BRODIE M.84), que sont la séquence, le choix et la répétition équivalents comportementaux de l'agrégation, de la généralisation et l'association.

4-2-5 Les interfaces

Les interfaces des SGBD actuels offrent un certain nombre d'outils parmi les suivants :

- un langage de commande assertionnel.
- un langage de programmation interactif (4ème génération)
- un éditeur plein écran (saisie et correction de tuples)
- des interfaces langage de programmation (COBOL, PASCAL, PL 1)
- un langage d'interrogation par grilles
- un générateur de formes (masques de saisie)
- un éditeur de rapports
- un utilitaire de génération de courbes, histogrammes
- un utilitaire de génération de statistiques
- un tableur géré sur écran

Les outils sont nombreux et variés mais on peut reprocher la technicité des langages (devoir exprimer les jointures), l'apparition tardive d'interface avec des langages récents (LISP ou ADA), et un niveau d'intégration insuffisant en ce qui concerne les interfaces langage de programmation (sauf PASCAL/R et ses extension (SCHMIDT 84)).

Dans le cadre d'un SAAL, le transformateur non nécessairement spécialiste de l'informatique doit pouvoir disposer d'interfaces plus commodes d'emploi.

Les interfaces utiles dans le cadre de l'amélioration de logiciel peuvent être classées en deux catégories :

- les **interfaces d'aide** au **déroulement du processus** d'amélioration, permettant par exemple l'édition graphique ou textuelle de l'arbre d'amélioration avec une **interface d'apprentissage** pour pouvoir réutiliser des mécanismes d'amélioration déjà envisagés.
- l'appel des fonctions d'amélioration et donc l'accès aux objets concernés.
- l'acquisition, la modification et l'execution de règles dans le cadre des fonctionnalités déductives.
- les **interfaces de communications** permettant la définition et la manipulation de vues simplifiées de la base des données de l'amélioration. L'utilisateur doit pouvoir disposer de différentes vues d'un même objet, élément constituant d'un état ou d'une version d'un morceau de programme.

Il y a longtemps que le problème des représentations multiples d'un objet a été abordé par exemple avec le concept de "**boite**" (WILSON,83, COUTAZ J.86) (formalisation de la représentation visuelle d'un objet),

sorte de filtre pour le flot de données qui circule entre l'utilisateur et la base des applications.

L'utilisateur doit pouvoir afficher ces vues de nature quelconque (graphe, diagramme, texte, tableau,...) et les modifier de façon interactive.

Parmi les difficultés qui surgissent, citons le problème de l'équivalence sémantique des différentes vues d'un même univers.

Les deux catégories d'interfaces précédentes nécessitent, pour laisser une certaine liberté au transformateur au niveau de la définition et de la modification éventuellement "artisanales" des graphes aux schémas, une **interface graphique.**

Cette dernière, intégrant un éditeur graphique, doit pouvoir générer les spécifications associées à un graphe et leur représentation dans la base de données.

4-2-6 Conclusion

Actuellement, nombreux sont les travaux de recherche portant sur la conception et la réalisation d'intefaces très évoluées pour satisfaire aux exigences des nouvelles fonctionnalités attendues des SGBD.

Toutefois, une autre approche de ces fonctionnalités est la tendance à les prendre en compte dans des langages à objet (SMALLTALK), (EXPRIM, HALIN G.89)

Après les langages de manipulation de données qui sont en général des algèbres ou des calculs, apparaissent des langages de programmation logique comme COL ("Complexe Object Language") d'Abiteboul et Grumbach (ABITEBOUL,89) avec des données traitées qui peuvent être relationnelles ou fonctionnelles.

Ces derniers langages, contrairement aux langages classiques de manipulation de données permettent :

- la gestion des objets structurés (nuplets, amas, ensembles,...)
- l'intégration de données relationnelles et fonctionnelles
- le traitement de la récursion.

Au niveau de la déduction :

- dans les bases de connaissances on voit se développer des langages de règles comme RDL 1 (SIMON,89 INRIA).
- pour les bases de données plus classiques avec des langages comme COL on réalise aussi des couplages entre des mécanismes de déduction et ces bases (ELLUL A.87) par exemple le couplage PROLOG / ORACLE via le langage C, et le couplage Méthode d'Alexandre / ORACLE.

4-3 LA PLACE D'UN SYSTEME D'AMELIORATION DE LOGICIELS (SAAL) DANS UN ATELIER INTEGRE DE GENIE LOGICIEL (AIGL).

4-3-1 Evolution des environnements de programmation

Un environnement de programmation classique est constitué d'un système opératoire et d'une collection d'outils élémentaires : un éditeur, un compilateur, un chargeur, un système exécutif et parfois un gestionnaire de versions et de configurations.

Ces outils permettent la création et la transformation de fichiers. Ils ne permettent pas la structuration, ni la conservation, ni le traitement de toutes les informations nécessaires tout au long du cycle de vie du logiciel.

Les environnements de programmation plus récents sont basés sur un dictionnaire de données ou une base de données qui structure et conserve tous les objets créés par les différents outils. De tels environnements appelés **Atelier de Génie Logiciel** (AGL, ou PSE pour Programming Support Environnement) peuvent être vus comme un ensemble d'outils coopérant pour capturer, représenter, contrôler, raffiner, transformer, ... les informations relatives à un projet.

On parle même d'**Atelier Intégré de Génie Logiciel** (AI GL) lorsqu'il s'agit d'un atelier de génie logiciel (AGL) cohérent.

Les outils d'un AIGL doivent permettre la conception et le développement cohérents de logiciels et leur maintenance.

La notion d'intégration est une notion progressive et le degré d'intégration d'un AIGL dépend des propriétés (décrites en 4-3-2) vérifiées par cet AIGL.

Les ateliers actuels étant faiblement intégrés constituent des boites à outils ; outils évoluant indépendamment les uns des autres.

Si on se place enfin à un niveau plus générique, on envisage alors la notion de **Structure d'Accueil** (SA)

Une structure d'accueil est un ensemble de mécanismes de base permettant le développement d'ateliers logiciels. Le niveau et la généralité des concepts de la SA doivent être suffisants pour pouvoir développer des ateliers logiciels plus ou moins sophistiqués.

Les structures d'accueil actuelles (il s'agit des systèmes d'exploitation), en assurant la mise en oeuvre du seul concept de fichier, ne permettent que difficilement le développement d'ateliers logiciels intégrés.
Une structure d'accueil est un métasystème qui correspond probablement à un système d'exploitation du futur paramétrable avec par exemple, comme paramètres le système, le langage, la méthode.
Une méthode étant un ensemble d'outils, de modèles et de règles; modules et règles définissant l'ensemble des séquences logiques possibles d'opérations pour passer d'un état cohérent du logiciel à un autre état cohérent.
Une méthode peut être, par exemple, développer sous GCOS, en Merise avec le langage COBOL.

4-3-2 Propriétés d'un atelier intégré de génie logiciel

Ces propriétés sont celles qui sont actuellement considérées comme nécessaires pour qu'une collection d'outils constitue un atelier de génie logiciel totalement intégré.(GODART C.,87)

4-3-2-1 Structuration de la collection d'outils

La collection d'outils est **structurée**.
Une sous-collection d'outils peut être associée à une méthode de développement , à un utilisateur, à un type d'information,...
Un outil peut appartenir à une ou plusieurs collections. Tout outil ou toute sous-collection doit être **identifiable**.

4-3-2-2 Complétude

Il existe toujours une méthode possible de mise en oeuvre des outils pour assurer un développement complet.

4-3-2-3 Cohérence

Tout résultat d'un outil doit être dans une forme utilisable par tout autre outil, dans le cadre d'une méthode quelconque de développement.

4-3-2-4 Conservation de l'information

Toutes les données générées pendant le développement et la maintenance du logiciel doivent être conservées durant la vie du logiciel (protection contre la modification).
En particulier, ces informations incluent les noms des outils et l'ordre de leur invocation (tactique) ayant permis d'aboutir à l'état d'avancement du logiciel.

4-3-2-5 Structuration des données

La collection des données doit être structurée poour intégrer les différentes vues utiles.

4-3-2-6 Extensibilité

Il doit être possible d'ajouter ou de modifier un outil, ou une sous-collection d'outils.

4-3-2-7 Distribution

Partage des informations entre différents postes de travail d'une équipe assurant la cohérence dans la coopération à la production et à la maintenance de logiciels.

4-3-2-8 Portabilité

L'indépendance de l'atelier vis-à-vis de la structure d'accueil sur laquelle il est développé est importante.

Remarques :

1) Ces propriétés que l'on ne retrouve pas toutes (c'est le cas de l'extensibilité et de la portabilité) dans les ateliers existants, sont celles que devrait posséder un Atelier d'Amélioration de Logiciel (AAL)

2) Autant la structure d'accueil est assez générale pour être indépendante du domaine d'application concerné, autant chaque AGL

ou AIGL est le résultat du choix d'un domaine d'application, d'un langage, d'un système support, de méthodes possibles.

3) Les propriétés d'un AIGL font appel aux techniques "bases de données" tant du point de vue de la structuration des données (modèles E-A) des outils et collections d'outils que de la conservation des informations (bases de données historiques).

4-3-3 A G L et processus d'amélioration du logiciel

L'écueil à éviter est le manque d'intégration entre le processus de développement et le processus d'amélioration. C'est pourtant le risque encouru si le niveau des techniques et concepts utilisés pour l'amélioration de logiciel est insuffisant.

Dans la pratique, on connait déjà le problème que pose le manque d'intégration de la maintenance de logiciels avec les étapes de développement.

Les utilisateurs sont obligés de travailler dans deux environnements différents :

- l'environnement de développement de nouvelles applications où l'on entrevoit la possibilité de maintenir ces dernières au niveau des spécifications logiques voir conceptuelles.
- l'environnement de maintenance des applications existantes où l'on travaille au niveau du code source.

Un environnement d'amélioration de logiciel est une organisation, c'est à dire qu'il a la propriété de relier, de maintenir, et de produire des objets. Un modèle de processus d'amélioration de logiciels est un ensemble d'outils, de modèles de représentation de l'information, de modèles d'enchainements d'outils et de règles de gestion.

Un modèle de processus d'amélioration de logiciel est là pour guider l'amélioration et contrôler la cohérence des états successifs du logiciel.

La définition d'un modèle d'amélioration doit comporter :

- la description des types d'objets et leurs liens

- la spécification syntaxique (profil des opérations) et sémantique des opérations de chaque type d'objet.
- la description des contraintes à respecter pour l'amélioration de logiciel

(ex. un élément modulaire ne peut être arborisé qu'après avoir été restructuré) avec pour le cas contraire des règles de production décrivant les opérations à exécuter pour retrouver un état stable de l'amélioration.

- un ensemble d'enchainements possibles d'opérations (ordre partiel) : **tactiques** prédéfinies d'amélioration.

Ces objets et opérations nécessaires à l'amélioration trouvent leur place dans la base d'objets du développement de logiciel augmentée de quelques opérations spécifiques (ex : opérations liées aux fonctions APPREHENDER, RESTRUCTURER, MODULARISER) et de règles décrivant un modèle de processus d'amélioration (ordonnancement et contraintes).

Une telle base d'objets peut être vue comme une **base de connaissances** et le système d'assistance comme un système à base de connaissances.

Réaliser une structure d'accueil à gestion d'objets, paramètrable par les modèles de processus de développement et d'amélioration peut consister en l'intégration des fonctionnalités d'un système à base de connaissances et d'un système d'exploitation.

4-4 CONCLUSION

L'amélioration de logiciel processus aboutissant à une sorte de réincarnation du logiciel dans un nouvel environnement constitue, en fait, une nouvelle phase du cycle de vie étendu.

L'étude des besoins en matière d'amélioration de logiciel montre qu'ils sont de même nature que ceux du développement de logiciel lorsqu'on envisage tout le cycle de vie :

Ces nouveaux besoins exigent de nouvelles fonctionnalités des SGBD en cours de développement mais aussi des fonctionnalités qui sont celles des bases de connaissances. On assiste alors par exemple à une démarche (VERNADA F.87) qui consiste à décrire en terme de base de données relationnelles les modes classiques de représentation des connaissances (règles de production, schémas et réseaux sémantiques), avec adjonction de moteurs d'inférence travaillant sur ces structures relationnelles.

Quant à l'approche de réalisation d'un environnement d'amélioration passant par un AGL, elle nécessite des bases de données. Il suffit, pour s'en convaincre, de passer en revue les propriétés attendues des ateliers de demain, propriétés nécessaires à la prise en compte du cycle de vie étendu du logiciel.

L'intégration souhaitable de tous les outils et de toutes les méthodes du cycle de vie étendu est grandement catalysée par le passage de l'informatique des **fichiers** à l'informatique des **objets**.

C'est dans cette informatique des objets que convergent les bases de données et le génie logiciel (GODART C.,87)

Aussi la réalisation d'un système d'amélioration de logiciel (SAAL), difficile dans le cadre d'un objectif à court terme, reste dans l'attente de la mise en oeuvre d'environnements en cours de conception.

II-5 DE LA DIFFICULTE PRATIQUE A AMELIORER DES PROGRAMMES : de COBOL à ADA, un EXEMPLE.

5-1 INTRODUCTION

Améliorer un programme c'est lui faire acquérir un certain niveau de qualité qu'il ne possède pas et cela conformément aux objectifs définis par les utilisateurs.

Une première difficulté résulte de la spécification de la qualité du logiciel.

La définition IEEE de la qualité d'un logiciel est la suivante :

" La qualité d'un logiciel est l'ensemble des caractéristiques d'un produit logiciel, relatives à son aptitude à satisfaire des besoins donnés".

Cette définition suggère une évaluation unique, à postériori, de la qualité; évaluation liée à l'utilisation du produit et exprimée par le degré de satisfaction de l'utilisateur.

Dans la pratique, la définition du niveau de qualité doit passer par une spécification quantitative de la qualité du produit à réaliser.

Cette définition du niveau de qualité ne pouvant être issue que de la concertation entre utilisateurs et informaticiens, dans notre cas, avant de lancer tout processus de transformation.

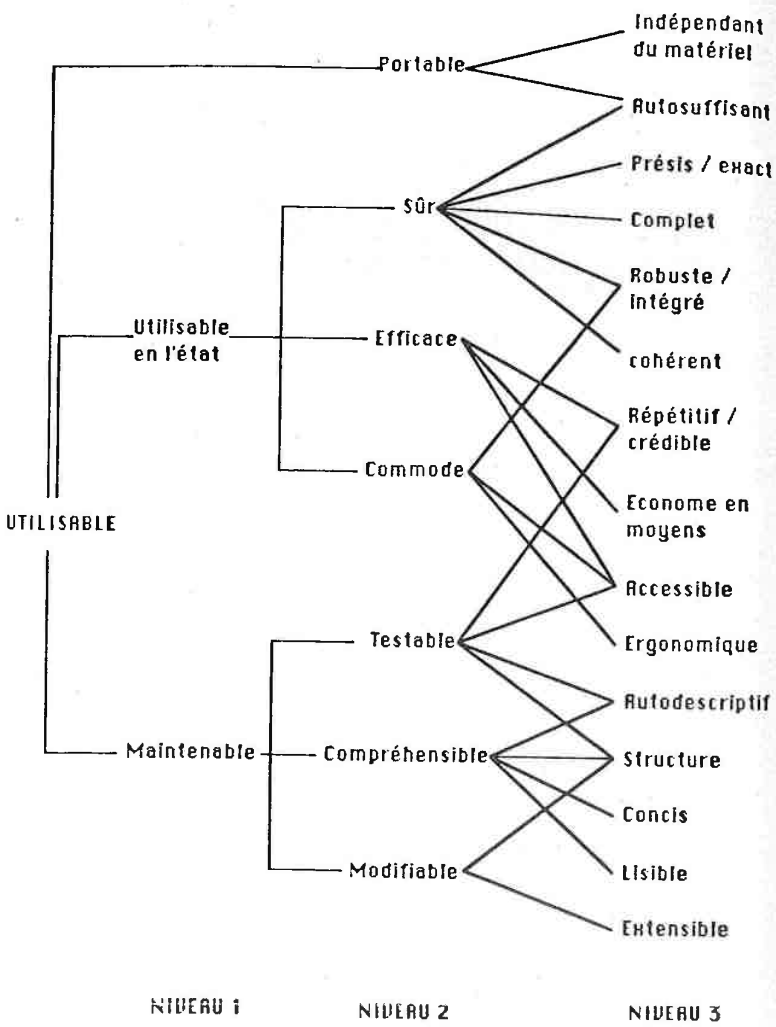
C'est dans cette optique qu'ont été réalisées, ces dernières années, deux études essentielles sur les caractéristiques de la qualité du logiciel.

• La première a été conduite par Barry W.BOEHM sous contrat du National Bureau of Standards (BOEHM 73).

BOEHM propose une décomposition arborescente de la qualité du logiciel à l'aide de paramètres sur trois niveaux (voir figure 1).

Par exemple, un programme est compréhensible s'il est lisible, concis, autodescriptif et structuré.

Mais on a reproché à BOEHM la caractérisation insuffisante de chaque niveau ainsi que l'absence de métriques d'évaluation des paramètres.



GRAPHE DE DECOMPOSITION DE LA QUALITE DE BOEHM.
FIGURE 1

• La seconde a été conduite par James A.McCALL, sous contrat du U.S. Air Force Electronic Systèms Division et Rome Air Dévelopment Center (McCALL 77)

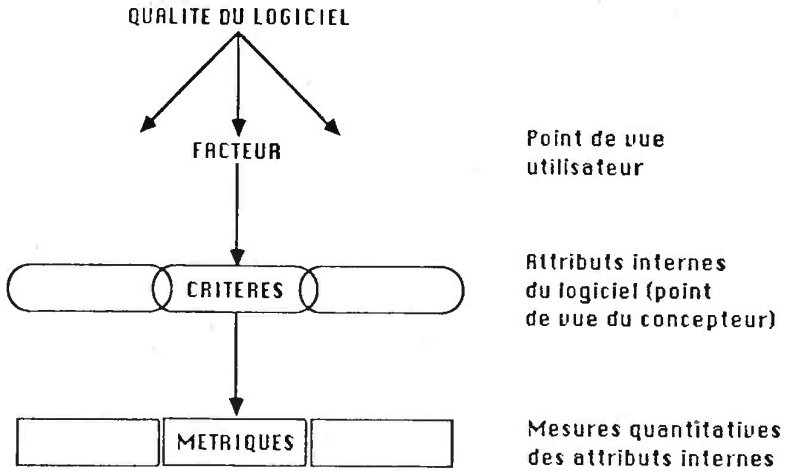
Contrairement à BOEHM, qui avait tendance à définir de manière absolue les paramètres de la qualité, McCALL a essayé d'établir un mécanisme extensible et adaptable, pour spécifier et mesurer la qualité d'un produit logiciel.

Mc CALL tenant compte des différents points de vue propose une décomposition de l'ensemble des paramètres en deux types :

• les paramètres externes ou facteurs de qualité caractérisant la qualité d'utilisation du logiciel.

• les paramètres internes ou critères de qualité, caractéristiques quantifiables permettant d'estimer les facteurs.

La décomposition de la qualité du logiciel selon Mc CALL reste une arborescence à trois niveaux (voir figure 2).



NIVEAUX D'ABSTRACTION DES PARAMETRES DE LA QUALITE (Mac CALL 77)

Figure 2

Mc CALL a ainsi défini 11 facteurs et 23 critères.

Les relations entre ces facteurs et critères ont été étudiées (ZAOUCHI 82); un facteur étant complètement défini par l'ensemble de ses critères; les relations entre facteurs étant décrites par les critères communs à ces facteurs (voir figure 3).

Figure 3 : RELATIONS ENTRE FACTEURS ET CRITERES

FACTEURS	CRITERES
CORRECTION	Traçabilité. Cohérence. Complétude.
FIABILITE	Tolérance aux erreurs. Cohérence précision. Simplicité.
EFFICACITE	Efficacité de mémorisation. Efficacité d'exécution.
INTEGRITE	Contrôle d'accès. Contrôle de manipulations des données.
UTILISABILITE	Formation. Communication. Exploitabilité.
MAINTENABILITE	Cohérence. Simplicité. Concision. Modularité. Autodescription.
FLEXIBILITE	Modularité. Généralité. Extensibilité. Autodescription.
TESTABILITE	Simplicité. Instrumentation. Modularité. Autodescription.
PORTABILITE	Modularité. Autodescription. Indépendance / machine. Indépendance / système.
REUTILISABILITE	Modularité. Autodescription. Indépendance / machine. Généralité. Indépendance / système.
INTEROPERABILITE	Modularité. Données communes. Facilité de communication.

Il est alors essentiel pour définir un objectif en matière d'amélioration de tenir compte de ces relations car, par exemple, un même critère peut avoir un effet positif sur un facteur et un effet négatif sur un autre. Ainsi, augmenter l'efficacité grâce aux critères associés, c'est diminuer la portabilité.

Quant à la mesure de la qualité, approche rationnelle s'il en est de la qualité, les chercheurs ont abouti à des modèles de mesure de différents aspects restreints de la qualité (mesure de la complexité par exemple) appliquant des théories mathématiques (graphes) ou de la physique (entropie).

Chaque modèle ne saisissant donc qu'un aspect particulier de la qualité du logiciel : lisibilité, modularité, fiabilité, complexité,...etc.

Les objectifs en termes de facteurs étant précisés, le processus d'amélioration consiste à agir de manière optimum sur les critères correspondant à ces facteurs.

Rappelons ici que notre travail nous a conduit (IIème partie, chapitre 2) à modéliser les données à manipuler à l'aide d'entités et de relations.

5-2 LE MODELE ET LA MESURE DE LA QUALITE

L'information sémantique apportée par le modèle doit permettre non seulement le suivi et la mémorisation des différentes versions du logiciel entre son état source et, finalement, son état cible mais également doit constituer un support convivial et précis des informations liées à la mesure quantitative du degré d'amélioration résultant des différentes transformations.

L'utilisateur ayant défini la cible d'évaluation (par exemple un certain composant modulaire) et fait choix des facteurs de qualité qu'il veut améliorer il en résulte un ensemble de fonctions et procédures à mettre en oeuvre.

Nous proposons alors par objectif d'évaluation un enrichissement du modèle (AYOUB 89) de l'ensemble des informations utiles pour la définition de ces fonctions et procédures. Ces informations pouvant apparaitre dans le modèle sous le forme de nouvelles entités ,de nouvelles relations ou de nouveaux attributs.

Ce processus doit permettre à l'utilisateur d'apprécier l'intérêt de ces informations et de contribuer à leur valorisation. Cet enrichissement est aussi un support intéressant pour un changement de tactique ou de stratégie en matière d'évaluation.

Il restera à l'utilisateur à apprécier l'intérêt de la mémorisation, permanente éventuelle, non seulement de ces informations de base pour effectuer les mesures mais aussi des résultats de ces mesures.

Pour améliorer du logiciel comme pour le construire, il s'agit bien de prévoir en termes de stratégies et d'agir en termes de tactiques.

Le contexte dans lequel nous nous situons est limité au passage d'un langage source de haut niveau (lhn-s) à un langage cible de haut niveau (lhn-c).

Dans le cas où il s'agit d'un seul et même langage, le processus d'amélioration est essentiellement caractérisé par une meilleure utilisation des éléments du langage (opérations et structures de contrôles par exemple) et par l'apport "manuel" et automatique d'informations complémentaires.

Dans le cas où le lhn-c est différent du lhn-s il faut également aborder les problèmes liés à la traduction parmi lesquels :

- la mise en correspondance des opérations, en particulier les entrées-sorties.
- la mise en correspondance des structures de données et des opérations associées.
- la prise en compte de la différence entre opérations mathématiques ou sur chaînes.
- la prise en compte de la différence entre les structures de contrôle...etc

Dans ce chapitre, nous n'envisageons que les problèmes de gestion et donc les langages permettant l'écriture des programmes correspondants.

5-3 QUELQUES CARACTERISTIQUES DES LANGAGES DE "GESTION"

Parmi les exigences auxquelles doit satisfaire un langage de gestion, on peut citer :

- le nombre et la complexité des opérations concernant les fichiers.

En dehors des opérations locales permettant l'accès séquentiel ou direct à des adresses logiques sur mémoires secondaires (lecture, écriture, modification, effacement d'articles), l'utilisateur doit pouvoir disposer d'opérations globales comme :

- le tri d'un fichier dans un ordre précisé
- la fusion de fichiers
- l'interclassement de fichiers triés sur le même critère
- l'accès aux articles d'un même fichier selon des ordres différents en fonction de clés d'accès définies à priori...etc
- la facilité et la lisibilité des opérations de recherche dans les tableaux
- une arithmétique décimale efficace et lisible.

Les opérandes doivent pouvoir contenir un grand nombre de chiffres significatifs (18 à 20 chiffres), et leur partie "décimale" doit pouvoir subir arrondis et troncatures à la manière comptable.

- le langage doit posséder un certain nombre d'opérations et fonctions mathématiques ou sur chaînes de caractères.
- un programme écrit dans ce langage doit pouvoir être lu par l'utilisateur gestionnaire.

Si le problème de l'amélioration de la qualité d'un programme est doublé de celui de la traduction dans un autre langage, il est important de s'assurer que le langage cible (lhn-c) permet de satisfaire à ces exigences et si possible d'une meilleure façon que le langage source (lhn-s).

Nos travaux pratiques se sont effectués avec COBOL comme lhn-s et ADA comme lhn-c.

5-4 DE COBOL à ADA

Compte tenu de notre limitation aux problèmes de gestion, et dans une perspective de traduction, notre choix a été COBOL et ADA pour diverses raisons.

• Sur le plan économique le langage COBOL (Common Business Oriented Langage), apparu dès le début des années 60, a connu un grand succès surtout sur les gros systèmes.

De nos jours, il est bon de remarquer que 50 % des emplois informatiques nécessitent la connaissance de ce langage soit pour maintenir les quelque milliards de lignes de COBOL existantes dans le monde, soit pour réaliser de nouveaux programmes. On dispose actuellement de nombreuses versions micro (comme MS-COBOL de MICROSOFT) de COBOL.

•La définition de ce langage a beaucoup évolué depuis sa création, tout d'abord pour tenir compte des besoins des utilisateurs (ce qui est assez rare en matière de définition de langage) puis finalement pour introduire très récemment les concepts indispensables pour un langage moderne. Une norme 8X doit remplacer les normes dépassées 68 et 74.

• COBOL et ADA sont deux langages créés sous contrat du DOD (Department of Defense)

• ADA est généralement considéré comme meilleur que COBOL pour différentes raisons :

- ADA bénéficie de sa nouveauté
- les concepts d'ADA permettent une programmation plus modulaire et structurée.
- l'environnement de mise au point est meilleur
- la possibilité de l'exécution parallèle de fragments de code
- ADA offre une bonne portabilité (COBOL ayant été jusque là un des langages les plus "portables").

• Comme l'écrit HILL (HILL 86) en dépit de l'opinion de Sammet (Sammet 81) la plupart des scientifiques de l'informatique pensent qu'ADA contient les caractéristiques essentielles de COBOL et plus encore.

Pour entrer un peu plus dans les détails envisageons successivement ces deux langages dans leurs caractéristiques essentielles puis quelques exemples de difficultés de traduction.

Nous proposerons ensuite un canevas général pour le processus d'amélioration et enfin quelques éléments concernant le traitement d'un petit programme COBOL.

5-5 QUELQUES CARACTERISTIQUES SYNTAXIQUES ET SEMANTIQUES IMPORTANTES EN COBOL

La matière de ce paragraphe est essentiellement issue de (ANSI,74) à l'exception de quelques références faites à (ANSI,85)

5-5-1 Structure d'un programme COBOL :

Un programme COBOL (voir exemple de programme en annexe) est constitué toujours dans le même ordre, de 4 divisions :

- l'IDENTIFICATION DIVISION dont le but est de donner un nom au programme et de permettre l'introduction de commentaires généraux.
- l'ENVIRONMENT DIVISION qui contribue à la portabilité (lien programme-machine), à l'indépendance du programme (liens objets logiques, objets concrets) et à l'optimisation et à la sécurité de l'exécution du programme.
- la DATA DIVISION qui permet la description de **toutes les données** du programme qui, de ce fait, sont **globales**. Elle est constituée essentiellement de deux sections :
  - la FILE SECTION qui contient toutes les descriptions des fichiers utilisés dans le programme. Chaque description apportant le nom logique du fichier et la description de ses types d'articles.
  - la WORKING-STORAGE section qui contient les descriptions de toutes les variables simples ou structurées utilisées dans le programme.
- la PROCÉDURE DIVISION qui correspond à l'ensemble des traitements définis sur les données décrites dans la division précédente.

Elle est constituée de sections et/ou de paragraphes.

Un **paragraphe** est formé d'une étiquette suivie d'une suite de phrases COBOL.

Une **phrase** COBOL étant une suite d'**instructions** COBOL (éventuellement une seule) terminée par un point (souvent celui qui délimite une instruction conditionnelle).

Une **section** est formée d'une étiquette de section suivie d'une suite de paragraphes.

5-5-2 Déclarations et types :

Un identificateur est une suite d'au plus 30 caractères pris parmi les lettres, les chiffres et le tiret et doit contenir, au moins une lettre. Il est qualifiable à l'aide d'identificateurs de niveau supérieur dans la structure considérée.

En COBOL, une déclaration associe un identificateur à un objet qui peut être variable ou constant (CONSTANT SECTION de la DATA DIVISION;)

5-5-2-1 Les types numériques

Pour résumer, on peut les classer grâce à l'attribut PICTURE (PIC en abrégé) en trois catégories :

1• **Display unedited** : décimal en base 10 mémorisé sous la forme d'une chaîne de caractères numériques avec ou sans signe;
exemple : PIC S999 V99
L'occupation mémoire correspondante est de un caractère mémoire par chiffre décimal (en dilaté) ou à la demande de l'utilisateur d'un caractère mémoire pour deux chiffres décimaux (en condensé).
Un attribut supplémentaire (ROUNDED permet lors de la troncature éventuelle de la partie décimale d'assurer un arrondi sur la dernière décimale conservée.

2.• **Display, édited** : décimal en base 10 édité dont les caractéristiques d'édition sont spécifiées à l'aide de symboles spéciaux;
exemple : PIC + FFF 9•9CR
La mise en forme imprimable d'une quantité numérique résultant d'un transfert en mémoire de cette valeur d'une variable de catégorie 1 dans une variable de catégorie 2.
Dans le cas des variables de type 2 un attribut supplémentaire (BLANK WHEN ZERO) peut, par exemple, donner une forme imprimable à la variable qui correspond à une chaîne de blancs dans le cas où sa valeur est zéro.

3 • **Computational'** La valeur est alors mémorisée en mode binaire (souvent complémentée à 2).
L'occupation mémoire correspondant au nombre entier de caractères de mémoire nécessaires pour mémoriser la valeur.
exemple : PIC S99V99 COMPUTATIONAL
Des variables de catégorie 1 et 3 peuvent échanger des valeurs et être utilisées indifféremment dans des calculs.

5-5-2-2 Les types simples non numériques

L'utilisateur dispose des types chaînes de caractères de longueur fixe, soit alphabétiques, soit alphanumériques, éventuellement éditées avec

le processus de cadrage à gauche implicite sauf indication contraire (attribut JUSTIFIED RIGHT).

5-5-2-3 Les types structurés

Il s'agit des types article et tableau

L'article est la racine d'un arbre dont les noeuds sont des éléments structurés et les feuilles des éléments simples.

La description linéaire utilisant identificateurs et nombres niveaux.

exemple :

```
01 ARTICLE.
     02 NUMÉRO PIC 9 (5).
     02 DÉSIGNATION PIC X (30).
     02 HORS-TAXE PIC 9 (4)V99.
     02 DATE-FIX.
          03 JOUR PIC 99.
          03 MOIS PIC X (12).
          03 ANNEE PIC 99.
```

L'élément structuré, dont l'article est un cas particulier, est considéré comme une chaîne de caractères dont la longueur est la somme des longueurs de ses composants; une affectation mettant en oeuvre un élément structuré s'effectue sans tenir compte des types spécifiques des composants de cette structure. Ceci peut être source d'erreurs et nuire à la portabilité.

Le **tableau** est un élément structuré déclaré avec l'attribut OCCURS pour définir chaque dimension

exemple :

```
02 T.
     03 ELEM PIC s9v9 OCCURS 5.
```

Le type des éléments d'un tableau peut être quelconque. Un tableau peut être de longueur variable (attribut DEPENDING ON). Les éléments d'un tableau sont accessibles soit à l'aide d'indices à valeurs entières soit à l'aide de pointeurs (attribut INDEXED BY).

Remarque :

Tout élément structuré peut être redéfini (attribut REDEFINES) à l'aide de variantes en général de même longueur mais sans définition de sélecteur.

5-5-3 Instructions de traitement et structures de contrôle

Rappelons que non seulement COBOL ne permet pas la récursivité mais de plus n'a pas été conçu pour respecter (et encore moins renforcer) l'esprit de la programmation structurée comme en témoigne la permissivité de la plupart des compilateurs.

5-5-3-1 Séquentialité

La séquentialité est indiquée à l'aide de séparateurs (espace, point-virgule, virgule ou point).

Une suite d'instructions peut être regroupée dans une phrase délimitée par un point, la phrase étant l'élément de composition des paragraphes.

5-5-3-2 Instructions hors contrôle

• MOVE [ALL]

Instruction d'affectation assurant automatiquement toutes les conversions légales nécessaires pendant le transfert.

ALL impliquant la répétition de la valeur émise pour remplir la variable réceptrice.

exemple :

```
MOVE ALL "XY" TO R
```

• SET

Instruction d'affectation (calcul) d'une valeur à une variable de type pointeur (index) pour accéder à un élément de tableau.

exemple :

```
SET INDEX 1 TO 5 , SET INDEX 1 UP BY 3
```

• COMPUTE [ON SIZE ERROR]

Instruction arithmétique permettant le calcul d'une expression arithmétique et l'affectation du résultat à une ou plusieurs variables avec conversions automatiques et éditions en mémoire si nécessaire.

exemple :

```
COMPUTE X ROUNDED, Y = exparithm ON SIZE ERROR PERFORM
TRAIT.
```

Le programmeur dispose également des instructions arithmétiques simples (ADD, SUBTRACT, MULTIPLY, DIVIDE).

L'option ROUNDED correspond à une demande d'arrondi sur le dernier chiffre conservé de la partie décimale ; l'option ON SIZE ERROR donne de l'instruction arithmétique une forme conditionnelle permettant d'indiquer un traitement à effectuer en cas de troncature envisagée de la partie entière dans la zone résultat.

• Option CORRESPONDING

Cette option permet pour l'instruction d'affectation et les instructions arithmétiques simples impliquant des opérandes structurés de mettre en oeuvre l'opération correspondante autant de fois qu'il y a de composants de ces opérandes de même nom et de même qualification.

• INSPECT

Instruction de parcours d'une chaîne de caractères permettant :

- le comptage des caractères satisfaisant à certaines contitions liées au positionnement dans la chaîne.
- le remplacement de ce même type de caractères par d'autres

exemple :

```
INSPECT ZONE TALLYING COMPTEUR FOR ALL "X" AFTER
      SPACE REPLACING BY SPACES;
INSPECT ZONE REPLACING LEADING
      ZEROES BY SPACES.
```

• SORT, RELEASE, RETURN

SORT est l'instruction de tri (appel d'un utilitaire de tri) d'un ensemble d'article d'un fichier de travail temporaire sur un ou plusieurs critères avec un ordre spécifié par critère.

L'instruction SORT permet de préciser la composition avant tri du fichier de travail et la constitution de fichiers résultats du tri à partir du fichier de travail trié.

Ces précisions sont apportées à l'aide de paramètres (noms de fichiers).

exemple :

```
SORT FICH-TRAV ASCENDING VAR1 DESCENDING VAR2 VAR3
USING FICH-DON OUTPUT PROCEDURE SECT.
```

Les instructions RELEASE et RETURN sont respectivement les instructions d'écriture d'un article (dans l'éventuelle procédure avant tri : INPUT PROCEDURE) et de lecture d'un article (dans l'éventuelle procédure après tri : OUTPUT PROCEDURE) du fichier de travail du tri.

• MERGE RETURN

De schéma semblable à l'instruction SORT, l'instruction MERGE assure l'interclassement des articles de plusieurs fichiers triés. Le fichier de travail trié peut être exploité par une procédure (OUTPUT PROCEDURE) après fusion ou être envoyé dans un fichier permanent.

• SEARCH [ALL]

Cette instruction assure une recherche séquentielle dans un tableau avec pointeur jusqu'à ce que l'une des conditions soit satisfaite et fournit alors la valeur du pointeur repérant l'élément correspondant ; l'une des conditions pouvant être le débordement en fin de tableau (attribut AT END).

exemple :

```
SEARCH ELEMENT AT END MOVE "NON" TO REPONSE
                WHEN val > ELEMENT (INDEX 1) MOVE "OUI" TO
                REPONSE
```

• STRING, UNSTRING

Ces instructions permettent, pour la première, la concaténation de sous-chaînes issues de différentes chaînes de caractères, pour la seconde, l'éclatement d'une chaîne de caractères en plusieurs sous-chaînes. Les sous-chaînes pouvant être délimitées par les bornes de la chaîne correspondante, par l'occurrence d'un caractère donné ou par une adresse sous la forme d'un pointeur.

exemple :

```
STRING A DELIMITED BY SIZE
       B DELIMITED BY "X"  INTO C
       ON OVERFLOW PERFORM DEBORD-C
```

En cas de débordement de la zone réceptrice, le programmeur peut indiquer un traitement à effectuer.

• ACCEPT, DISPLAY

Ces instructions permettent le transfert de petites quantités d'informations entre une zone de mémoire d'une part et un support externe géré par un terminal à vitesse lente (clavier-écran, imprimante,....) d'autre part.

exemple :

```
ACCEPT VAR FROM console
DISPLAY "DATE DU JOUR" DATE ON papier
```

• EXIT

Utilisable uniquement comme seul contenu d'un paragraphe qui devient paragraphe vide.

Le paragraphe vide est utilisé par exemple pour terminer une procédure interne (voir PERFORM).

5-5-3-3 Instructions et structures de contrôle

Les instructions et structures de contrôle du COBOL (ANSI 74) sont insuffisantes mais comme nous le verrons par la suite; la norme (ANSI 8X) apporte quelques corrections.

• GO TO

Instruction "bien connue" de transfert impératif du contrôle à la première instruction suivant l'étiquette désignée (première instruction d'un paragraphe ou d'une section en COBOL).

• GO TO... DEPENDING ON

Sorte de GO TO calculé qui correspond à un branchement parmi une liste d'étiquettes données, à celle dont le rang figure à l'instant comme valeur entière du sélecteur.

exemple :

```
GO TO étiq 1 étiq 2 DEPENDING ON S
```

Pour une valeur du sélecteur ne correspondant pas à un rang possible dans la liste des étiquettes citées l'instruction GO TO n'est pas exécutée et le contrôle passe à l'instruction suivante.

Cette instruction permet une simulation de l'instruction **CAS**.

• ALTER

Cette instruction permet de modifier l'objet d'un GO TO constituant à lui seul un paragraphe

exemple :

```
X.  GO TO Y.
P.
  |  -------
Y.
  |  -------
     ALTER X TO PROCEED TO Z
  |  -------
     GO TO X
  |  -------
Z    STOP RUN.
```

• IF... [ELSE...]

La syntaxe correspondante peut se définir par,

IF condition [THEN] (liste d'instructions / NEXT SENTENCE)

[ELSE (liste d'instructions / NEXT SENTENCE)].

où [ ] décrit une partie optionnelle et (/) décrit un choix possible.

Chaque liste d'instruction peut contenir à son tour un IF mais se termine par un point (qui marque aussi la fin de l'instruction IF composée) ou le mot ELSE;

L'absence de ENDIF en COBOL (ANSI 74) est une grave lacune enfin comblée (ANSI 8X) jusqu'alors contournable par l'utilisation de l'instruction PERFORM.

• PERFORM [UNTIL]

La forme non itérative, PERFORM étiq 1 [THRU étiqn], correspond à la demande d'exécution d'un morceau de code à la manière d'une procédure interne sans paramètre ; toutes les variables sont globales.

Le morceau de code concerné est le contenu du paragraphe d'étiquette étiq 1 si l'option THRU n'est pas précisée.

Dans le cas contraire, ce morceau de code est constitué de l'un quelconque des ensembles d'instructions situées sur un chemin de contrôle allant de la première instruction du paragraphe étiq 1 à la dernière instruction du paragraphe étiqn.

Les PERFORM recursifs ne sont pas autorisés.

Remarque :

Les combinaisons malheureuses d'instructions GO TO et PERFORM sont une des causes du manque de structuration des programmes COBOL et constituent l'un des moyens de mettre à l'épreuve la permissivité du compilateur.

Dans la forme itérative du PERFORM, le nombre d'itérations peut être indiqué à l'aide d'une valeur entière ou résulte d'une condition d'arrêt.

Les schémas syntaxiques de ces deux variantes sont :

- PERFORM étiq1 [THRU étiqn] (entier/variable) TIMES

  Cela correspond à une boucle POUR avec un compteur (muet) interne.
- PERFORM étiq1 [THRU étiqn] [VAYING index1 FROM init1 BY pas1] UNTIL condition 1 [AFTER indexj FROM initj BY pasj UNTIL condition j]*

Les index d'itération pouvant être des variables simples de catégorie 1 ou 3 ou de type pointeur.

Dans le cas où il y a un seul index un pseudo-code équivalent à ce PERFORM peut être :

```
index 1 = init 1
Tantque non condition 1
    executer étiq1 à étiqn
    index 1 = index1 + pas 1
fintantque
```

5-5-4 La modularité

L'unité de programmation en COBOL est le programme ou le sous-programme. Il n'existe pas de découpage possible d'un programme en unités compilables séparément. Les seuls ensembles de "modules" sont les "JOBs" qui sont des suites de programmes avec communication d'informations sous la forme de fichiers de données transmis d'un programme à un autre.

Il est toutefois possible de gérer des bibliothèques de fichiers de texte source, fichiers insérables dans le texte d'un programme avant compilation à l'aide d'une instruction COPY fichier. L'option REPLACING de cette dernière instruction permet d'effectuer des modifications du texte copié pour l'adapter à chaque appel.

5-5-4-1 Les pseuso-procédures internes.

Il s'agit en fait de morceaux de texte source, ensembles de paragraphes (ou de sections) contrôlés par un PERFORM et travaillant sur une partie des variables globales du programme.

Ces unités "logiques" sans variables locales et sans paramètres sont des "blocs" étiquetés non réutilisables si ce n'est comme fichiers de texte source insérables avec modifications dans différents programmes (COPY).

5-5-4-2 Les sous-programmes.

Il s'agit de procédures externes appelables par l'instruction CALL (passage de paramètres par référence) d'appel de sous-programmes. La recursivité n'est pas permise en matière d'appel de sous-programmes.

Les paramètres ne peuvent être que de type article ou variable simple et de plus aucun contrôle de compatibilité de type n'est effectué entre paramètres formels et réels.

Un fichier et, à fortiori, un sous-programme ne peut pas être paramètre.

Dans un sous-programme (de même structure qu'un programme) les paramètres sont regroupés dans une section spéciale de la DATA DIVISION d'entête LINKAGE SECTION.

Le sous-programme appelé reste en mémoire sauf indication contraire du programme appelant (CANCEL) et son exécution se termine à la rencontre de l'instruction EXIT PROGRAM.

5-5-5 Les exceptions, la mise au point.

Le traitement des exceptions en COBOL consiste en l'exécution d'une procédure définie par l'utilisateur à la rencontre d'une erreur en cours d'exécution du programme. Il s'agit essentiellement des erreurs d'entrée-sortie concernant les fichiers gérés par le programme.

Le système d'aide à la mise au point dont la mise en oeuvre est conditionnelle consiste aussi en procédures définies par l'utilisateur et qui ne sont exécutées que lorsque certaines conditions sont réalisées, par exemple :

- à chaque exécution d'un certain type d'instruction d'entrée-sortie concernant les fichiers.
- à chaque exécution d'une "procédure" COBOL ou d'une instruction ALTER concernant une "procédure GOTO"
- à chaque référence faite à une certaine variable

• à chaque modification de la valeur d'une variable référencée dans une instruction PERFORM au niveau de l'option VARYING et/ou UNTIL.

Toutes les procédures ainsi définies par l'utilisateur accompagnées de leurs conditions d'exécutions sont regroupées dans une entête spéciale de la PROCEDURE DIVISION délimitée par le parenthèsage DECLARATIVES, END DECLARATIVES.

exemple :

```
DECLARATIVES.
EXTENSION SECTION.• USE AFTER ERROR PROCEDURE ON fichier 1.
ETIQ1•          .....
                texte de la procédure à exécuter en cas d'erreur d'entrée-
                sortie concernant fichier 1
PROC1 SECTION •  USE FOR DEBUGGING proc1•
                .....
                texte de la procédure à exécuter juste avant chaque
                exécution de la procédure proc1
                .....
END DECLARATIVES.
```

5-5-6 Les entrées-sorties, les fichiers.

Il y a deux niveaux d'entrées-sorties :

• Un niveau des textes pour les communications entre l'homme et la machine permettant les échanges d'informations de faible volume entre la mémoire centrale et les supports externes correspondant à des périphériques lents comme l'imprimante, le clavier-écran. Ces entrées-sorties correspondent en COBOL aux instructions ACCEPT et DISPLAY.

Exemple :

ACCEPT VAR1 pour l'entrée à la console d'une valeur à donner à la variable VAR1 du programme COBOL correspondant
DISPLAY "RESULTAT DU TIRAGE" RES pour l'impression d'une chaîne de caractère suivie du contenu de la variable RES à l'instant.

• Un niveau des fichiers

Il s'agit alors d'échanger entre la mémoire centrale et les supports externes de gros volumes d'informations sous la forme d'ensembles d'articles constituant des fichiers.

5-5-6-1 Les fichiers; organisations.

Les organisations de fichier accessibles en COBOL sont :

1) l'organisation **séquentielle**

Seul l'accès séquentiel est possible aux articles d'un tel fichier dans l'ordre où ils sont mémorisés sur le support externe.

2) l'organisation **relative**

En plus de l'accès séquentiel, ce type de fichier permet l'accès direct à chaque article grâce à son rang (adresse relative) parmi les articles du fichier correspondant.

3) l'organisation **indexée**

Ce fichier est caractérisé par l'existence d'une clé d'accès primaire et de zéro ou plusieurs clés d'accès secondaires. A chaque clé est associée une table d'index permettant l'accès aux articles du fichier correspondant soit de manière ordonnée en fonction des valeurs de la clé (croissantes ou décroissantes), soit de manière directe en indiquant une valeur précise de la clé.

5-5-6-2 Les fichiers : description

Les informations nécessaires pour décrire un fichier et pouvoir accéder à ses articles se situent essentiellement à deux endroits dans le programme COBOL :

- Dans l'ENVIRONMENT DIVISION, où le paragraphe FILE-CONTROL de l'INPUT-OUPUT SECTION contient une phrase SELECT par fichier permettant de préciser;
  - le nom du fichier logique, ensemble des articles du point de vue du programmeur.
  - le nom du fichier physique, ensemble des articles et des blocs du point de vue du système de gestion de fichiers ou le nom du périphérique concerné.
  - l'organisation du fichier
  - le type d'accès aux articles choisi par le programmeur.
  - si nécessaire, le nom des clés d'accès utilisées dans le programme.
  - éventuellement le nom d'une zone de réponse (mot d'état) qui contiendra à chaque instant pendant l'exécution du programme une valeur significative du type d'erreur éventuellement commise lors des traitements mis en oeuvre concernant ce fichier.

- Dans la DATA DIVISION où chaque description de fichier (indicateur FD) de la FILE SECTION permet de préciser après avoir rappelé le nom du fichier logique;
    - l'existence ou non d'un bloc étiquette pour ce fichier et si ce bloc est de format standard ou non pour le système courant.
    - éventuellement le facteur de blocage ou la taille des blocs, la taille de l'article.
    - éventuellement le type de codage des caractères utilisé dans le fichier (EBCDIC ou ASCII).
    - pour les fichiers d'édition (éditeur COBOL) la taille de la page, le haut et le bas de page, le débordement en bas de page.
    - la description (niveau 01) du ou des types d'articles du fichier (redéfinition implicite de la zone d'entrée-sortie associée au fichier.

5-5-6-3 Les fichiers : opérations d'entrée-sortie

Un tableau résume les opérations essentielles. Chaque instruction est désignée à l'aide d'un ou plusieurs mots COBOL significatifs pour éviter de donner les définitions syntaxiques complètes.

DESIGNATION DES INSTRUCTIONS	RÔLE DES INSTRUCTIONS
OPEN	Ouverture d'un fichier en entrée, en sortie ou en modification.
CLOSE	Fermeture d'un fichier
READ... AT END...•	Lecture de l'article suivant dans un fichier à organisation séquentielle ou dans un fichier d'une autre organisation mais pour lequel la méthode d'accès choisie est séquentielle : s'il n'y a plus d'article à lire le code COBOL de l'option AT END est exécuté.
READ...INVALID KEY...•	Lecture directe d'un article caractérisé par une valeur de clé pour les fichiers à organisation relative ou indexée. Si l'article correspondant n'existe pas le code COBOL de l'option INVALID KEY est exécuté.
START	Permet pour un fichier indexé ou relatif le positionnement d'un pointeur de lecture en fonction de la valeur de clé donnée pour une séquence d'articles commençant par l'article associé à cette valeur de clé plutôt que par le premier article du fichier.
WRITE	Permet l'écriture d'un article dans un fichier en cours de création (ouverture en sortie).
WRITE...INVALID KEY...•	Permet l'écriture d'un article dans un fichier relatif ou indexé existant (ouverture en modification). Si l'insertion ne peut être faite le code COBOL de l'option INVALID KEY est exécuté.

REWRITE-[INVALID KEY]	Permet la réécriture d'un article dans un fichier d'une organisation quelconque. L'option INVALID KEY permet d'exécuter le code associé lorsque l'article à remplacer n'existe pas.
DELETE...[INVALID KEY...]	Permet l'effacement d'un article dans un fichier non séquentiel. Le code associé à l'option INVALID KEY est exécuté lorsque l'article à effacer n'existe pas.

5-6 CONCLUSION : DE COBOL 74 à COBOL 8X

Le langage COBOL c'est aussi un préprocesseur de tables de décision, un éditeur d'états permettant :

• d'une part de laisser le soin à un préprocesseur de linéariser la table de décision en générant le texte COBOL correspondant.

• d'autre part au programmeur de décrire un état à imprimer (description de fichier état) et d'en demander l'impression, en cours d'exécution du programme, non plus ligne à ligne mais page par page ou bloc logique par bloc logique (entête de commande, bas de facture,...)

Mais enfin et surtout COBOL aujourd'hui c'est COBOL 8X en cours de "standardisation", dont nous pouvons citer quelques caractéristiques (NBSI 83) (CUGI 82), (FIOR 83).

• L'allègement des descriptions COBOL comme la possibilité de ne pas identifier un composant d'une structure ; l'initialisation statique d'un tableau sans avoir à effectuer une redéfinition (REDEFINES).

• L'introduction d'une instruction EVALUATE approchant le schéma d'une instruction CAS

```
EVALUATE      expression
WHEN          valeur 1        PERFORM trait 1
-------
-------
WHEN          valeur j        PERFORM traitj
-------
END EVALUATE
```

• Le bloc PERFORM où la procédure à exécuter est localisée avec le PERFORM

L'option WITH TEST du PERFORM permet de plus de réaliser des boucles TANTQUE et JQA

exemple :

```
PERFORM WITH TEST AFTER UNTIL  X > = 100
        ADD 1 TO X
        MOVE T (X)  TO T (X + 1)
END-PERFORM
```

Un indice peut être une expression arithmétique et tous les opérateurs de relation classiques sont autorisés

• Généralisation du parenthèsage permettant entre autres choses d'imbriquer proprement des instructions conditionnelles; citons par exemple :

IF ... THEN... [ELSE]... END-IF
READ ... AT END ... END-READ
ADD...END-ADD

• Référence directe possible à des portions de variables, considérées comme chaînes de caractères sans déclarer ces portions.

MOVE VAR (4:3) TØ RES où 4 est le rang du premier caractère de VAR à transférer et 3 la longueur de la chaîne de caractères à transférer.

• La possibilité d'imbriquer des programmes avec les notions de variables globales partagées (celles spécifiées comme telles dans le programme le plus externe) et de variables locales.

La mise au point actuelle et la commercialisation de compilateurs COBOL norme 8X pour ordinateurs mini et micro permet de penser que COBOL est un langage loin de disparaître;

De plus, le processus d'amélioration des programmes existants étant aussi lourd pour passer de COBOL 68 ou 74 à COBOL 8X que pour passer à ADA, le langage cible n'étant qu'un des éléments pris en compte dans ce processus, la tâche des entreprises est sensiblement la même dans les deux cas.

5-7 QUELQUES CARACTERISTIQUES SYNTAXIQUES ET SEMANTIQUES ESSENTIELLES EN ADA POUR LA TRADUCTION DE COBOL VERS ADA.

Nouveau langage conçu comme COBOL à l'instigation du Ministère de la Défense des U.S.A. (DoD), ADA est pour certains Le Langage de la fin du siècle qui va supplanter tous les autres, pour d'autres un langage raté manquant d'"orthogonalité". Pour aider le professionnel à se forger une opinion, un manuel d'évaluation de ADA a été rédigé par le "Groupe Ada de l'AFCET" (VERR 82) dans le même esprit que celui qui avait amené un groupe de travail d'évaluation d'Algol 68 quelques années auparavant (BOUSSARD 71).

Sa vocation vise une grande variété d'applications mais pour notre part seule sa capacité à traiter des problèmes de gestion nous intéresse dans ce chapitre.

5-7-1 Structure d'un programme ADA

Un programme ADA est constitué d'un ensemble d'**unités de programmation** qui correspondent à un découpage logique du programme et peuvent être compilées séparément ou non.

Les deux principaux types d'unités étant les sous-programmes et les paquetages.

5-7-1-1 Les sous-programmes

Les sous-programmes correspondent à la réalisation bien différenciée des fonctions et des procédures :

• les **fonctions** retournent une valeur de type quelconque et ont des paramètres non modifiables

• les **procédures** peuvent modifier leurs paramètres mais ne retournent pas de résultat au point d'appel.

Chaque sous-programme est décomposé en deux parties :

- la **spécification** ou entête du sous-programme.
- le **corps**

Les deux parties d'un sous-programme peuvent être séparées à condition de répéter intégralement la spécification devant le corps; toutefois un sous-programme ne doit pas être appelé pendant l'élaboration d'une partie déclarative si le corps de ce sous-programme se trouve après l'appel.

Les sous-programmes peuvent être définis récursifs. A la manière d'une macro il est possible d'obtenir la copie du texte du sous-programme à chaque appel en utilisant :

pragma incline (nom ssprog1 {,nom ssprog2});

La partie formelle contient la liste des déclarations des paramètres formels (leur type est quelconque sauf procédural) et leur mode de passage (**in, out, in out**)

5-7-1-2 Les paquetages.

Les paquetages sont des unités qui permettent de regrouper un ensemble d'entités (types, variables, sous-programmes,etc.)

Un paquetage se compose de deux parties :

- la **spécification** regroupant, uniquement sous la forme de déclarations, toutes les informations utiles à la bonne utilisation des entités exportées.

Cette spécification est composée de deux parties :

  - la partie visible (accessible à l'utilisateur)
  - la partie privée éventuelle (accessible au compilateur).

- le **corps du paquetage** qui comporte aussi deux parties :
  - une partie déclarative; objets nécessaires au fonctionnement interne du paquetage et la réalisation concrète des unités de programmes décrites dans la spécification du paquetage (sous-programmes, paquetages,etc.)
  - une partie initialisation.

Remarque :

A priori le parallélisme offert au niveau exécution par les unités **tâches**, est de peu d'intérêt pour nos préoccupations immédiates.

5-7-2 Déclarations et types

En Ada une déclaration associe un identificateur à une entité qui peut être un objet, un nombre ou un type.

5-7-2-1 Les déclarations d'objet

Un objet est une entité ayant les propriétés d'un type précisé. Les objets peuvent être déclarés variables ou constants (attribut **constant**) et initialisés. Ce type de déclaration correspond sensiblement à ce que l'on trouve dans la plupart des langages.

Remarque :

La déclaration d'un **nombre** constitue un cas particulier de la déclaration d'objet constant. La déclaration d'un nombre ne permet que des valeurs numériques entières ou réelles sous la forme d'expressions littérales, donc statiques (évaluation à la compilation).

5-7-2-2 Les déclarations de type

Parmi les principaux types définis en Ada et utiles dans la perspective d'une traduction COBOL-ADA citons :

- les types **scalaires** prédéfinis décomposés en types **discrets** (énumérés ou entiers) et en types **réels** (flottants ou fixes).

Une variable entière est habituellement implémentée sous forme d'un élément binaire de longueur fixe avec complément à 2. La spécification d'entier ne précise pas la taille.

Pour les variables réelles fixes est utilisée une implémentation binaire point fixe spécifiée par la donnée d'une plage de valeurs (dont est déduite la taille de la partie gauche) et la limite d'erreur absolue (dont est déduite la taille de la partie droite).

- les types **composés** qui sont les **tableaux** et les **articles** et qui sont construits par l'utilisateur.

5-7-2-3 A propos des types construits

Outre les types prédéfinis, l'utilisateur peut définir de nouveaux types ou sous-types (**is new**) à l'aide de :

  - l'énumération
  - la dérivation avec ou sans contrainte
  - les constructeurs tableau, article.

5-7-2-3-1 Les **articles**

Un type article est une suite de composants initialisables par défaut.

exemple :

```
type   DATE      is
  record
       JOUR  :  INTEGER range  1..31;
       MOIS  :  STRING (1..10);
       AN    :  INTEGER range 0..3000 := 1980;
  end record;
```

Il est possible d'introduire dans une description de type article un discriminant qui est une contrainte utilisée pour :

  - établir une contrainte d'indice sur un composant (de type tableau) de l'article.
  - choisir une variante de l'article (**case**)
  - discriminer un composant (de type article) de l'article.

exemple :

```
type modpaie is (espèce, chèque)
...
type paiement (stylepaie : modpaie)
  record
        responsable : string (1..30);
        bureau : string (1..30);
        case stypepaie
              when espèce = > agent payeur : string (1..30)
              when chèque = > centre : string (1..30);
                              numéro : string (1..12);
              when others  = > null;
        end case
  end record;
```

5-7-2-3-2 Les tableaux

Les tableaux classiques connus dans la plupart des langages sont dits avec contrainte en Ada.

exemple :

```
type GROUPE is array (1..20) of ELEVE;
G1, G2 : GROUPE;
ECOLE : array (1..50) of GROUPE;
```

Un tableau sans contrainte est une sorte de modèle de tableau dans lequel le domaine de valeurs des indices reste à définir. Les contraintes d'indices étant nécessairement introduites à la déclaration d'un objet sur un type tableau sans contrainte.

exemple :

```
type RANG is range  1..1000;
type GROUPE is array (RANG range <>) of ELEVE;
G1, G2 : GROUPE (1..20);
```

5-7-3 Instructions et structures de contrôle :

Les instructions en Ada contrairement à COBOL permettent de respecter, sinon de renforcer l'esprit de la programmation structurée (sauts contrôlés, blocs...)

5-7-3-1 Séquentialité :

La séquentialité est marquée par le point-virgule terminant chaque instruction.

Une **suite** d'instructions peut être regroupée dans un **bloc** pouvant contenir une partie déclarative ; blocs utilisables en particulier en Ada pour le traitement des exceptions.

5-7-3-2 Affectation :

L'ordre d'évaluation des parties, droite ou gauche en premier, n'est pas défini par le langage. Il n'y a **pas de conversion de type implicite**, même pas pour les types numériques.

Il n'y a pas d'affectation multiple.

5-7-3-3 Structures de contrôle non itératives :

On trouve en Ada toutes les structures classiques pour contrôler la séquence d'exécution des instructions d'un programme.

• GO TO

Permet le transfert du contrôle à une instruction étiquetée. Les étiquettes sont déclarées implicitement.

Il est possible d'étiqueter le **end** d'une instruction composée par le biais d'une instruction vide :

<<FIN>> **null ; end.**

Les sauts en Ada ne sont autorisés que s'il n'y a aucun problème d'environnement : par exemple on ne peut pas entrer dans une instruction composée par un **goto** ni passer d'une partie à une autre d'une même structure de contrôle. Par exemple, on ne peut transférer le contrôle entre les parties **alors** et **sinon** d'une instruction conditionnelle.

• IF

Tout ce qu'il y a de plus classique avec le paranthèsage **if ... end if** et toujours un point-virgule devant un else et devant un end if.

A noter la forme condensée de l'instruction if (avec **elsif**) simplifiant l'écriture des if en cascade (réduction des end if) et l'existence des courts-circuits pour l'élaboration séquentielle des conjonctions.

• CASE

De schéma syntaxique classique pour une instruction à choix multiples exclusifs.

case expression **is**
{ when cond 1 { | condj} = > suite d'instructions}
end case;

où condj peut être une expression, une liste de valeurs ou **others** et dans ce dernier cas condj représente tous les cas non cités.

exemple :

```
déclare n : character;
case n is
        when 'A' .. 'z'        = > trait_lettre;
        when '0' .. '9'        = > trait_chiffre;
        when ';' | '.' | ':' | ',' = > trait_sépar;
        when others            = > trait_qcq;
        end case;
```

5-7-3-4 Les boucles

La boucle de base (**loop**... **end loop**) traduisant l'itération sans fin est à l'origine de la construction des deux autres boucles possibles en Ada : la boucle **while** et la boucle **for**.

• FOR

Dans ce cas, une variable de contrôle locale (non modifiable par le programmeur et de portée limitée à la boucle) prend ses valeurs dans un ensemble discret de valeurs ou dans un intervalle.

exemple :

for I **in** PLAGE **loop**... **end loop;**

A noter l'absence de pas

• WHILE

Il s'agit de la boucle tantque avec test en début de boucle.

exemple :

```
read (FICHIER);
while not FIN_LECT (FICHIER)
        loop
                traitement d'un article
                read (FICHIER);
        end loop;
```


Remarque :

Il n'existe pas plus qu'en COBOL (ANSI 74)

• de boucle avec test en fin de boucle (disponible en COBOL 8 X)

• d'instruction de branchement en fin de boucle pour passer à l'itération suivante.

• EXIT

Le but de **exit** est de permettre d'indiquer, n'importe où dans une boucle, une sortie impérative ou conditionnelle de la boucle implicitement englobante sinon de celle citée.

exemple :

```
loop << celle_là>>
        I := I + 1
        exit celle_là when I > 10;
        ...
end loop celle_là ;
```

5-7-4 La modularité :

Elle s'exprime en termes d'unités de programmation et de généricité.

Paquetages et sous-programmes sont décomposables en unités autonomes (de bibliothèques) et unités séparées avec les problèmes que cela pose aux niveaux de l'ordre de compilation et de l'ordre d'élaboration de ces unités.

Le **paquetage** permet de regrouper un ensemble d'entités (types, variables, sous-programmes,etc.) en tenant compte de critères de nature logique.

Le concept de paquetage ne peut donc que faciliter :

• une démarche descendante de conception d'un programme (par exemple emboîtement d'un paquetage dans le corps d'un paquetage).

• un rassemblement d'entités à partager de la même manière (emboîtement d'un paquetage dans une spécification).

• la description abstraite d'un type ou d'un objet : le type est complété dans la partie privée de la spécification mais en revanche la dissimulation de la structure du type n'est pas totale.

La **généricité** permet d'éviter de réécrire pour des applications différentes des unités de programmation presque semblables. Par exemple, des types abstraits définis dans des paquetages et paramètrés par des types ; le type pile muni de ses fonctions "empiler", "dépiler" etc... où le type des éléments est un paramètre. Par exemple aussi l'écriture en Ada de "routines" d'entrée-sortie générales pour

exploiter les fichiers en utilisant une fonction générique de conversion sans tests de compatibilité entre paramètres formels et paramètres réels.

5-7-5 Les exceptions :

En Ada, une exception peut être déclenchée (raise) lors de l'exécution d'un (sous-) programme P. Ce déclenchement est un branchement immédiat à une séquence de traitement appelée récupérateur (**handler**) puis le (sous-) programme P est considéré comme terminé et le contrôle est rendu à un bloc englobant le récupérateur (pas de reprise du contrôle derrière le point de déclenchement).

5-7-6 Les entrées-sorties, les fichiers :

On considère trois niveaux d'entrées-sorties; niveau des fichiers, niveau des textes et niveau des périphériques;

Le **niveau des périphériques** concerne les seules primitives qui servent à échanger de l'information de commande avec les périphériques.

Le **niveau des textes** est celui qui concerne les entrées-sorties lisibles par l'être humain et pour les communications. Un texte est considéré comme une suite de caractères regroupables en lignes et en pages, entités gérables explicitement à l'aide de primitives appropriées.

6-7-6-1 Le niveau des fichiers

La définition des fichiers et des opérations possibles sur ceux-ci est la partie du langage Ada la plus récente et la plus évolutive.

Un fichier est aussi considéré en Ada comme constitué de deux entités : le **fichier interne** (logique ou abstrait) et le **fichier externe** (concret), tous deux visibles par le programme.

Les accès aux éléments d'un fichier en Ada peuvent être faits :

- de manière strictement séquentielle (**fichier séquentiel**)
- de manière séquentielle ou sélective (**fichier direct**)

Le traitement des fichiers se fait à l'aide de primitives contenues dans les paquetages génériques SEQUENTIAL_IO et DIRECT_IO à raison d'un exemplaire de ces paquetages par type d'élément distinct de fichier.

Les primitives disponibles sont de trois catégories :

• les primitives d'association entre fichier interne et fichier externe :

CREATE et OPEN pour établir une association

CLOSE et DELETE pour terminer une association

RESET pour réinitialiser l'état d'un fichier sans rompre l'association en cours.

• les primitives d'examen de l'état d'un fichier. Un fichier pouvant être passé en paramètre d'une procédure, celle-ci peut ignorer l'état de ce fichier transmis; Les primitives d'examen apportent cette information à la procédure de façon plus agréable que la récupération des exceptions signalant des interdictions.

Parmi les primitives citons :

MODE qui rend le mode d'ouverture courant.

NAME qui rend le nom du fichier externe associé;

IS-OPEN qui indique si une liaison est établie;

END_OF_FILE qui indique si la lecture séquentielle en cours est achevée;

et pour les fichiers directs

SIZE qui indique le nombre d'éléments du fichier externe;

INDEX qui indique la valeur de l'index courant.

• les primitives d'accès aux éléments d'un fichier.

Ce sont les primitives :

READ lecture d'un élément

WRITE écriture d'un élément

L'accès ne peut être que séquentiel pour un fichier séquentiel suivant le modèle d'une suite, mais peut être séquentiel (index courant) ou sélectif (index explicite) pour un fichier direct suivant le modèle d'un tableau à une dimension.

Constatons qu'il n'existe pas dans le langage de base le type fichier séquentiel indexé. On peut trouver un paquetage de définition de ce type de fichier dans une version écrite par Kurbel et Pietsch (Kurb, 86) utilisant le paquetage DIRECT-IO.

On trouve également dans (VERR 82) la définition d'une procédure de fusion de fichiers séquentiels ordonnés et la définition d'un paquetage générique SEQUENTIEL_INDEXE.

Remarque :

L'utilisation du paquetage DIRECT _IO est inefficace, du fait des nombreux appels de procédures que contient conséquemment chaque primitive du paquetage SEQUENTIEL_INDEXE, du fait en particulier de l'absence de primitive d'effacement d'article.

De plus, l'utilisation de la généricité pour définir le type des articles d'un fichier interdit les articles de longueur variable car le type des articles fourni lors d'une génération de DIRECT_IO doit être contraint et donc tous les articles auront la même variante.

5-8 QUELQUES ELEMENTS POUR LA TRANSFORMATION DE PROGRAMMES COBOL.

Notre objectif ici n'est pas d'être exhaustif ce qui nécessiterait un travail considérable, mais d'aborder quelques problèmes liés à l'amélioration de programmes COBOL.

5-8-1 Instructions Indésirables

1) Réduction des GO TO et ALTER

- Eliminer les GO TO inutiles par réorganisation du texte source. Dans certains cas, il sera nécessaire d'introduire des GO TO par exemple pour ramener à l'unicité du point de sortie dans un composant modulaire de la PROCEDURE DIVISION.
- Transformer le schéma ALTER-GO TO

Chaque instruction ALTER est remplacée par l'affectation d'une constante à une variable de travail, variable à ajouter dans les déclarations. Le GO TO associé est alors remplacé par une instruction conditionnelle IF ou GOTO -- DEPENDING.

Ainsi étant donné le schéma ALTER-GOTO suivant :

```
P1.
  -----
  ALTER     P3   TO   PROCED   TO   P5.
  -----
P2.
  -----
  ALTER     P3   TO   PROCED   TO   P8.
  -----
P3.     GO TO.
P4.
  -----
P5.
  -----
P8.
  -----
```


Il peut être remplacé par :

```
P1.
  -----
  MOVE 1     TO    VAR-SELECT.
  -----
P2.
  -----
  MOVE 2     TO    VAR-SELECT.
  -----
P3.
  IF VAR-SELECT = 1    THEN       GO TO P5,
     ELSE IF VAR-SELECT = 2       GO TO P8.
P4.
  -----
P5.
  -----
P8.
  -----
```

Ce schéma impose d'ajouter VAR-SELECT PIC 9 dans les déclarations.
Le paragraphe P3 peut être écrit :

```
P3.
  GO  TO  P5  P8        DEPENDING      ON   VAR-SELECT.
```

- transformer le GOTO calculé en structure CAS.

Le GO TO calculé peut aussi permettre d'éviter l'imbrication des tests (IF complexe) dans le cas où toutes les conditions imbriquées portent sur des valeurs possibles d'une seule variable.

Cette dispersion du contrôle par le GO TO calculé nécessite souvent son regroupement (par des GO TO) en un point, celui de sortie d'une "structure CAS".

Dans l'exemple ci-dessus, on peut par exemple, si le contexte s'y prête, créer un paragraphe vide P5-8, point de regroupement du contrôle.

```
P5.
  -----
  GO TO P5-8.
P8.
  -----
P5-8.    EXIT.
```

• statuer sur le PERFORM simple.

Eliminer un PERFORM simple et le remplacer par le contenu du paragraphe "PERFORMe' " si ce dernier est, par exemple, de petit volume.

Pour tout paragraphe exécuté tantôt par un PERFORM simple tantôt du fait de sa place dans le texte source (en séquence) faire choix d'un seul mode de passage du contrôle à ce paragraphe dans le programme correspondant.De ce fait, soit remplacer les PERFORM par le texte du paragraphe, soit remplacer l'occurrence du texte exécutée en séquence par un PERFORM.

5-8-2 Contributions à la modularité.

La modularité peut être recherchée soit au niveau des traitements, il s'agit de construire des modules, soit au niveau des données, il s'agit alors d'approcher la définition de types abstraits.

5-8-2-1 Modularité des traitements

Chaque paragraphe (ou ensemble de paragraphes) PERFORMe' est candidat au statut de procédure interne ou externe.

D'un point de vue pratique, on accepte quelques règles :

- un module doit correspondre à une entité logique du problème (l'utilisateur est seul juge) et à une entité discrète du traitement.
- du point de vue exécution, il ne possède qu'un seul point d'entrée et en général un seul point de sortie (aucun GO TO vers une étiquette extérieure au corps du module).
- l'activation du module est commandée depuis l'extérieur du module (accès depuis un point quelconque du programme).

En COBOL toutes les variables sont globales avec les conséquences suivantes :

- plusieurs composants modulaires peuvent utiliser les mêmes variables ce qui peut favoriser l'utilisation d'une même variable à différentes fins.
- le traitement codé dans un composant modulaire est figé à l'ensemble des variables utilisées dans les instructions qui le composent ; le composant est de réutilisabilité nulle.

Pour qu'un composant modulaire accède au statut de procédure (réutilisable) il faut définir une interface spécifique.

Pour définir les paramètres composant cette interface, on peut procéder en deux temps :

1) classification des variables ayant une occurrence au moins dans le composant en variables important une valeur, variables exportant une valeur, variables ayant leur valeur modifiée dans le composant (méthodes d'analyse de flux de données de Aho et Ullman (AHO 79), notre méthode des sous-ensembles de variables à dépendance limitée) chapitre I,6).

2) détermination des variables locales au composant et par différence des paramètres "réels" avec leur sens (entrée, sortie ou entrée-sortie).Il ne reste plus alors qu'à définir l'interface du composant modulaire à l'aide d'un ensemble de paramètres formels compatibles avec les paramètres réels mis en évidence.

Chaque appel de cette nouvelle procédure interne est alors :

- précédé d'une phase d'affectation de l'interface (paramètres entrée et entrée-sortie)
- suivi d'une phase d'affectation de l'interface (paramètres sortie et entrée-sortie).

Les difficultés rencontrées sont de deux sortes :

1) les variables étudiées peuvent être simples ou structurées. Certaines instructions opèrent sur des variables structurées (READ) et donc altèrent le contenu de tous les champs composants, d'autres instructions modifiant un champ altèrent de ce fait toute la structure englobante.

2) La détermination des classes de variables (donc des paramètres) dans les composants modulaires ne peut résulter que d'un processus ascendant partant des composants feuilles du graphe de contrôle avec propagation le long des structures de contrôle. (Kildall 73, Kam, Ullman 76, AHO-Ullman 73, Allen-Cocke 76, Maher-Sleeman 83).

L'une des difficultés est celle de déterminer si une variable exporte une valeur (utilisée après le composant modulaire) dans le cas d'un composant impliqué directement ou indirectement dans le corps d'une boucle (processus de fermeture transitive sur la relation d'utilisation d'une variable sans problème car COBOL n'autorise pas la récursivité).

Remarque :

Les procédures externes en COBOL correspondent aux sous-programmes dont les paramètres ne peuvent être que des variables simples indépendantes (niveau 77) ou structurées (niveau 01 uniquement).

Une procédure interne peut être transformée sans difficulté en procédure externe si l'entité logique de traitement qu'elle constitue présente un intérêt au-delà du programme englobant.

5-8-2-2 Le typage

Un certain nombre de langages permet de réaliser des abstractions de données (et de contrôle) comme ALPHARD (WULF-75-78), CLU (LISKOV-75-77), ATM (CHABRIER,79), SMALLTALK (GOLDBERG,76), EUCLID (LAMPSON,77), ADA (VERRAND (LE),82, STRATFORD,82), etc...

Dans ces langages étudiés dans (DERNIAME 79, HANSON 79) l'abstraction est introduite soit grâce aux notions de base, soit en offrant des mécanismes de restriction de visibilité.COBOL n'offre aucune de ces deux possibilités aussi l'introduction de types abstraits en partant de programmes COBOL ne peut être envisagée que de l'une des manières suivantes :

- modifier légèrement la définition du langage COBOL pour introduire les notions nécessaires à la manière de ZELKOWITZ dans (ZELKOWITZ,78) pour le langage PL 1 ou de PROCH (PROCH,80) pour PASCAL; cela en général au détriment des performances du programme objet (pointeurs, gestion dynamique des objets "abstraits").
- profiter de la traduction de COBOL à ADA et utiliser ainsi les possibilités d'ADA en matière de définition et de construction de types abstraits.

Mais dans un cas comme dans l'autre, une première difficulté est de mettre en évidence des types en partant uniquement du texte source des programmes COBOL.

Représenter un type cela revient à créer, parmi toutes les catégories de module (voir Modularité dans CHAK 80) un module "type" définissant une classe d'objets en même temps que les opérations applicables.

Il s'agit donc de modulariser partiellement un texte source en utilisant les relations contribuant à la mise en évidence d'objets appartenant à une même classe et dans le même temps des opérations les concernant.

Ainsi en COBOL on peut approcher le concept de type de façon progressive en passant par des spécifications partielles.

- du point de vue statique dans la DATA DIVISION on peut relever pour chaque variable :
  - son identificateur
  - la nature des caractères (alphabétique, numérique, alphanumérique) issue des symboles de la clause PIC.
  - le type de mémorisation (clause USAGE)
  - la longueur
  - si elle est éditée ou non (symboles d'édition dans la clause PIC).
  - si elle est structurée, ses composants immédiats au niveau suivant de l'arborescence et/ou la variable dont elle est composant (relation de dépendance hiérarchique) : avec comme cas particuliers :
    1) le cas de l'article d'un fichier, association dans une description de fichier de deux variables de niveau de description respectifs 01 et FD.
    2) le cas du tableau, association de la variable à indicer (clause OCCURS) avec le nom global du tableau.
  - si elle partage de la mémoire directement ou indirectement avec d'autres variables (relation de dépendance en mémorisation ou synonymie).

- du point de vue dynamique dans la PROCEDURE DIVISION, on peut relever :
  - pour chaque occurrence d'une variable l'opérateur (arithmétique ou logique) agissant sur elle et les opérandes associés (relations de dépendance arithmétique ou logique).
  - pour chaque variable, les variables échangeant des valeurs avec elle (instructions MOVE, COMPUTE) (relation de dépendance en valeur).
  - pour chaque variable son ou ses modes de calcul ou de définition (COMPUTE, READ) (relation de dépendance en définition) : une variable est définie par une expression arithmétique (COMPUTE), par une donnée obtenue par lecture (READ) ou par une expression logique (niveau 88 en DATA DIVISION).
  - pour chaque relation relevée entre variables, on peut préciser sa localisation (numéro de ligne, nom de paragraphe ou de section) et son contexte.

Le contexte peut être composé essentiellement de deux éléments :

1) le niveau dans la structure logique des données : structure à la base de certaines méthodes de conception et de transformation de programme (WARNIER,75) et sous-jacente dans d'autres méthodes (DUCRIN 85).

2) l'expression logique composée qui définit l'ensemble des conditions auxquelles satisfont à l'endroit du texte source où se situe l'occurrence de la relation relevée les éléments constituant la structure logique des données.

Le second élément peut être obtenu par composition des conditions de base attachées à chaque structure de contrôle (condition d'un IF, condition d'arrêt (UNTIL) d'un PERFORM itératif), le long du graphe de contrôle ou plutôt de la branche concernée de l'arbre abstrait du programme.

Par contre, le premier élément est plus difficile à préciser car on ne bénéficie pas en général au niveau du texte source, en tête de chaque paragraphe ou section, comme dans le texte d'analyse, en tête de chaque module de définition d'un objet, d'une part de l'identification de l'objet défini, d'autre part de sa définition générale (conditionnelle ou itérative) permettant d'établir un lieu précis avec les données utiles à cette définition.

Il devient alors nécessaire de tenir compte de la structure hiérarchique des composants modulaires et de la position des instructions d'entrée-sortie dans les modules.

La seule aide qu'on puisse apporter à l'utilisateur dans sa recherche de variables et d'opérateurs "allant ensemble" c'est de lui fournir en plus de la liste des éléments statiques et dynamiques indiqués précédemment un multigraphe à la manière du graphe des noeuds de dépendance (ou hypergraphe ou graphe des arêtes de l'hypergraphe) de PORTMANN M.C. 74 et une décomposition de ce multigraphe en sous-multigraphes connexes au sens de la recherche d'ensembles d'articulations minimum (CHAK,80).

On appelle ensemble d'articulations de deux sous-graphes E' et E" d'un même graphe E, l'ensemble des arcs qui relient des points E' à E" ou inversement.

Remarque :

D'un point de vue pratique, on peut envisager de simuler le concept de type à l'aide de la clause COPY. Cette clause permet la copie avant compilation à l'endroit de l'appel (par COPY), d'un morceau de texte source issu d'une bibliothèque de textes sources. Il est même possible pendant cette recopie de modifier le texte recopié. Cette possibilité augmentée de la **redéfinition** et du **renommage** permet de limiter les transformations mais n'offre qu'une approche grossière de la notion de type.

5-8-3 Approches pour la traduction de COBOL en ADA.

Loin d'être une traduction ligne à ligne (et encore moins mot à mot), le passage de COBOL à ADA nécessite de nombreuses transformations COBOL origine et parfois du développement en ADA. Nous donnons ici quelques exemples de difficultés abordées, difficultés liées au manque de structuration du programme COBOL ou aux exigences ou possibilités du langage ADA.

5-8-3-1 Autour du GO TO

En COBOL, aucune restriction n'est imposée concernant l'utilisation du GO TO, de plus il n'existe pas d'instruction de sortie multiniveaux (multilevel exit).

Aussi si la transformation doit aboutir à une autre version en COBOL du programme initial, l'élimination des GO TO ne peut être réalisée que si l'on se contente d'une équivalence fonctionnelle (pour des données identiques, des résultats identiques) entre les deux versions de programme.

Le langage ADA possède une instruction EXIT (**exit** [nom-boucle] [**when** condition];) permettant la sortie d'une boucle englobante à un quelconque niveau.

Il devient alors possible (RAMSHAW L. 88) d'éliminer les GO TO dans le passage de COBOL à ADA à condition selon le niveau d'équivalence souhaité de rendre le graphe de flux de contrôle (éventuellement **augmenté**) réductible (un seul point d'entrée par cycle. RAMSHAW propose des règles d'élimination locale des GO TO à condition que chaque bloc de texte source étudié ne possède pas de **point de**

croisement du contrôle. La notion de bloc étant liée ici à la portée des étiquettes et correspond dans notre cas à un programme COBOL entier.

A noter que si le langage ADA permet les GO TO vers l'extérieur d'une instruction structure, il interdit les GO TO vers l'intérieur ce qui est une condition nécessaire pour conserver l'équivalence structurelle des deux versions du programme lors d'une élimination des GO TO.

L'utilisation des exceptions est aussi un moyen d'éliminer certains GO TO. A noter toutefois que si le mécanisme de **déclenchement** et de **récupération**

existe en COBOL et en ADA le retour après exécution du récupérateur n'est pas le même dans les deux langages.

- En COBOL, il s'agit d'un schéma "correction" : il y a essai de reprise d'exécution immédiatement après le point de déclenchement de l'exception (AT END du READ, fin d'exécution d'une "procédure").
- En ADA, il s'agit d'un schéma "terminaison" : le (sous-) programme en cours d'exécution est considéré comme terminé et le contrôle est rendu à un bloc englobant le récupérateur. Il existe une instruction de déclenchement d'exception (RAISE).

Les GO TO calculés sont transformés en instruction CASE.

5.8.3.2 Autour du IF

Selon les versions de COBOL il existe ou non le **endif** ; il n'existe pas de **elsif**;

L'expression de la condition peut faire appel à la factorisation d'opérandes et d'opérateurs.

En ADA on dispose du **end if** du **elsif** et de l'élaboration collatérale ou séquentielle (courts-circuits) au niveau des conditions composées.

Un IF imbriqué COBOL peut être traduit en un IF ADA ou en un CASE, les difficultés apparaissent essentiellement au niveau de la traduction des conditions (type des variables, des opérateurs).

exemple :

Si la variable typmodif est déclarée comme chaîne d'un caractère alphabétique :

Le texte COBOL suivant :

```
IF typmodif =     " C "
THEN PERFORM TCREATION
ELSE  IF typmodif =     " A "
        THEN PERFORM TANNULATION
        ELSE        IF typmodif = " M "
                THEN PERFORM TMODIFICATION
                ELSE PERFORM TERREURMODIF.
```

devient en ADA :

```
CASE typmodif IS
    WHEN ' C ' =     >     PERFORM TCREATION;
    WHEN ' A ' =     >     PERFORM TANNULATION;
    WHEN ' M ' =     >     PERFORM TMODIFICATION;
    WHEN OTHERS = >     PERFORM TERREURMODIF;
END CASE :
```

où il reste à transformer les PERFORM par exemple en appels de procédures.

5.8.3.3 Autour de l'expression mixte IF avec GO TO.

Quant à une expression non structurée comme " IF A = B GO TO C. " on peut avoir comme premier réflexe de la transformer en l'expression structurée " IF A = B PERFORM C." à condition de vérifier l'équivalence et d'obtenir une amélioration des critères de qualité envisagés.

J. Cris Miller rappelle bien dans son article (MILLER,87) que ce n'est pas là la seule transformation envisageable compte tenu des questions que l'on peut se pose : par exemple,

- Quel est le contenu du paragraphe contenant l'expression ?
- Le paragraphe C contient-il un GO TO, un CALL éventuellement sans retour, un PERFORM d'une procédure de niveau logique inférieur ?
- Le paragraphe C est-il référencé par ailleurs ?
- Le paragraphe C est-il lourd, contient-il des expressions conditionnelles à son tour ?
- Où passe le contrôle après exécution du paragraphe C ?

Les réponses à ces questions peuvent conduire à différentes possibilités comme

" IF A = B PERFORM C PERFORM D." si le paragraphe C consiste simplement en une suite d'instructions impératives et/ou d'appels de procédures,

" IF A = B PERFORM GO TO D." si le paragraphe C renvoit au paragraphe D.

" IF A = B ADD X Y ." dans le cas où le paragraphe C est très court.

Il n'y a que si le paragraphe C ne contient ni branchement ni instruction d'arrêt d'exécution, si la liste d'instructions dans C n'est ni trop longue ni trop courte et si le contrôle revient de C à l'instruction suivant l'expression à transformer que cette dernière peut être remplacée par " IF A = B PERFORM C ".

Pour répondre à toutes ces questions et d'autres encore, le seul texte source est très difficile à déchiffrer, aussi l'utilisateur doit-il avoir accès au graphe de contrôle et/ou d'appel (à différents niveaux, modulaire, logique) à condition que ces derniers soient décorés d'informations précisant les contextes de chaque noeud, les instructions d'entrée-sortie, de terminaison).

Mais en règle générale, il est souhaitable de passer du GO TO au PERFORM; cette dernière instruction allant dans le sesns de la modularité, réutilisabilité.

5-8-3-4 Autour du PERFORM.

Pour ce qui est du PERFORM simple, impératif deux cas peuvent se présenter :

- le PERFORM est remplacé par la suite d'instructions qu'il contrôle("remontée" d'une suite d'instructions dans la structure logique du programme).
- la "procédure COBOL" PERFORMée est transformée en une procédure ADA avec une liste de paramètres déterminée par l'analyse des variables concernées comme on l'a vu précédemment.

Dans le cas d'un PERFORM itératif, on peut distinguer les deux formes possibles

- PERFORM procédure N times qui peut se traduire par une boucle FOR en ADA à condition dans le cas où N est une variable non COMPUTATIONAL d'assurer la modification de sa représentation pour être utilisable en ADA.

- PERFORM procédure [VARYING] UNTIL condition qui peut se traduire par une boucle WHILE avec NOT (condition).

La seule vraie difficulté une fois de plus est la modification du mode de représentation des variables.

5-8-3-5 Autour des instructions complexes de COBOL.

Pour ce qui est des instructions complexes, on trouve dans (HILL,86) et dans (VERRAND,82) des propositions de traduction.

Pour SORT et MERGE la traduction sous la forme, probablement concurrente, d'une tâche avec différents points d'entrée liés à l'initialisation, aux clés, aux instructions d'entrées-sorties et à la terminaison.

Pour SEARCH INSPECT, STRING et UNSTRING on peut envisager des procédures regroupées en packages.

Pour les fichiers indexés et les opérations associées on peut (VERRAND,82) définir un paquetage générique SEQUENTIEL-INDEXE, utilisant DIRECT-IO.

Toutes ces instructions complexes COBOL nécessitent l'écriture de packages ADA volumineux ce qui alourdit les programmes ADA et n'améliore pas la lisibilité.

5-8-3-6 Autour des représentations de variables et des calculs.

Nous avons classé précédemment les variables COBOL en trois catégories selon le mode de représentation; Contrairement à ADA le langage COBOL permet le traitement de données numériques sous forme de liste de caractères dans les opérations arithmétiques. Cette forme offre une grande souplesse au niveau :

- du nombre de caractères pour une variable numérique (non lié à la taille des mots mémoire comme en format interne binaire).
- des mécanismes d'arrondi et de troncature à la façon comptable ou 1.000 000 est très différent de 0.999 999.
- de la possibilité de traitements comparatifs avec des chaînes de caractères.

De plus COBOL offre les conversions automatiques des représentations de variables dans le cas où les représentations sont hétérogènes au sein d'une instruction arithmétique.

Nous abordons là, les nécessaires et longues conversions de représentations de variables dans le passage de COBOL à ADA. Ces problèmes sont essentiellement ceux traités dans (HILL,86) et force est de constater qu'en matière de description de variables le texte source ADA est beaucoup plus long et difficile à lire que le texte COBOL;

exemple issu de (HILL,86)
texte COBOL :

```
01    Ligne-courante.
      02 numéro  PIC   zzz 9.
      02 FILLER  PIC   X (5) VALUE SPACES.
      02 prix    PIC   zz,zzz.99  OCCURS 5.
      02 FILLER  PIC   X (7) VALUE SPACES.
      02 message       PIC X (10).
```

texte ADA :

```
DECLARE  Ligne-courante IS
  RECORD
      numéro      :     STRING (1••4);
      FILLER_001  :     STRING (1••5) : = (1••5 = ' ');
      prix        :     ARRAY (1••5) OF STRING (1••9);
      FILLER_002  :     STRING (1••7) : = (1••7 = ' ') ;
      message     :     STRING (1••10);
  END RECORD;
```

avec des informations complémentaires concernant "numéro" et "prix" que sont

```
NUMERIC_PICTURE : = (DERIVED_LENGTH = > 4, PICTURE_LENGTH
= > 4, PIC_STRING = > (1••4= > "zzz9", others = > '-' );
NUMERIC_PICTURE : = (DERIVED_LENGTH = > 7, PICTURE_LENGTH
= > 9, DECIMAL_LOCATION = > 2, PIC_STRING = > (1••9 = >
"zz, zzz•99", others = > '-");
```

en supposant tous les types nécessaires déclarés.

5-9 UN EXEMPLE CONCRET D'AMELIORATION ET DE TRADUCTION

Parmi les différents programmes COBOL étudiés, citons ici, l'un des plus simples; il consiste en une mise à jour d'un fichier des produits vendus par une entreprise.

Les textes sources, graphes et schémas auxquels il est fait référence dans la suite sont placés en annexes.

5-9-1 L'énoncé informel

5-9-2 Les objectifs

5-9-3 Quelques étapes d'amélioration de l'exemple

5-10 CONCLUSION

CONCLUSION ET PERSPECTIVES

Dans le cadre de l'amélioration des programmes dans l'optique de leur réécriture pour de nouveaux environnements logiciels et compte tenu, d'une part, jusqu'à une date récente du manque d'intérêt porté à ce problème par les concepteurs de logiciels, d'autre part de l'échec des tentatives de transformations et/ou traductions automatiques (traductions) sur le terrain nous défendons la thèse suivante :

- il n'y a pas d'amélioration absolue d'un programme, celle-ci doit être envisagée vis à vis d'objectifs précis correspondant à une certaine qualité à atteindre pour une nouvelle version de ce programme;

- cette amélioration ne peut se concevoir sans la possibilité d'une part, pour le transformateur de choisir parmi un ensemble de fonctions d'amélioration, celles qu'il convient d'appliquer à chaque instant en fonction du contexte, et de l'état de la version courante du programme, d'autre part de mesurer, à la demande, le gain obtenu, dans le sens de l'approche des objectifs, par l'application de ces fonctions.

- il est nécessaire d'assister le transformateur, dans sa tâche d'amélioration, par un logiciel spécifique intégré dans un atelier de Génie Logiciel offrant en autres possibilités, l'interactivité, le traitement de données complexes multi-média, la modélisation à la fois des données et du processus de modélisation.

Ce mémoire consistant en réflexions et propositions concernant l'amélioration de programme doit trouver son prolongement dans différentes perspectives comme :

- le développement d'une spécification formelle des activités et fonctions de l'amélioration de programmes.

- la conception et la réalisation d'une maquette de système d'amélioration inspirée des modèles génériques proposés et appliquée dans un environnement précis à un langage de programmation déterminé, prolongement par exemple, de nos réflexions dans le passage de COBOL à ADA;

- la participation à la conception et à la réalisation d'un atelier de génie logiciel futur dans le sens de l'intégration des fonctionnalités caractérisant l'amélioration des programmes ; ceci semble bien

correspondre aux très récentes élucubrations intitulées Reverse Engineering;

- l'adaptation d'éditeurs très évolués, pour l'aide à la génération et à l'instanciation de modèles de données et de modèles de processus d'amélioration.

Cette liste non exhaustive montre l'ampleur de la tâche à accomplir pour satisfaire aux besoins considérables des entreprises dans le domaine de l'amélioration de programmes. Nous formulons l'espoir que cette attente ne sera pas déçue en comptant sur une prise de conscience par les concepteurs et chercheurs, de la "noblesse" de la tâche d'amélioration de programmes, tâche enfin réalisable de nos jours grâce à l'évolution rapide des moyens dans le domaine du Génie Logiciel.

PARTIE III

ANNEXES

UN MINI-LANGAGE D'APPLICATION : L

Définition de la syntaxe concrète en format BNF-like ; c'est l'automate capable de décider de l'appartenance d'une phrase donnée au langage.

```
<programme>   ::=   <module>*
<module>      ::=   <entête><corps>
<entête>      ::=   <ident>(<list decla)|<ident>()
<corps>       ::=   DEBUT<list decla><listbloc>FIN
<listdecla>   ::=   <decla>|<listdecl ><decla>
<decla>       ::=   <mode><listident>;
<mode>        ::=   entier|chaîne
<listident>   ::=   <ident>|<listident>,<ident>
<listbloc>    ::=   <bloc>|<listbloc>•<bloc>
<étiquett >   ::=   % ID
<listinstru > ::=   <instruc>|<listinstruc>;<instruc>
<instruct>    ::=   <affect>|<alter>|<iter>|<trans>|<saut>|<exec>
<affect>      ::=   <ident>:=<expres>
<alter>       ::=   SI<test>ALORS<listinstruc>IS|
                    SI<test>ALORS<listinstruc>SINON<listinstruc>IS
<iter>        ::=   TANTQUE<test>FAIRE<listinstruc>FAIT
<trans>       ::=   LIRE<ident>|ECRIRE<expres>
<saut>        ::=   ALLERA<étiquette>
<ident>       ::=   % ID
<expres>      ::=   <op1expres><expres>|<elemexpres>|
                    <expres><opexpres><elemexpres>
<op1expres>   ::=   +|-
<opexpres>    ::=   +|-|•|/
<elemexpres>  ::=   <elem><opexpres><elem>
<elem>        ::=   <ident>|<const>|(<expres>)
<test>        ::=   <elemtest>|<op1log><test>|
                    <test><oplog><elemtest>
<opcomp>      ::=   >|<|=|≠
<elemtest>    ::=   <elem><opcomp><elem>
<oplog>       ::=   ET|OU
<op1log>      ::=   NON
<exec>        ::=   EXECUTER<etiquette>
<const>       ::=   %NOMBRE
```


Les identificateurs précédés du symbole "%" tel que % ID désignent des unités lexicales génériques définies par des expressions régulières dans l'analyseur lexical du langage.

DEFINITION DE LA SYNTAXE ABSTRAITE :

C'est la description arborescente des programmes dans ce langage.La syntaxe abstraite est composée d'opérateurs et de phyla :

- Les opérateurs sont les noeuds de l'arbre ; ils sont d'arité fixe (inférieure à 3) ou d'arité variable (noeuds de liste).

Les opérateurs d'arité nulle sont les feuilles de l'arbre et correspondent aux atomes du langage.

Les opérateurs d'arité fixe non nulle peuvent avoir des fils de types différents, tandis que les fils d'un noeud de liste doivent tous être de même type. Le type d'un noeud est le phylum auquel ce noeud appartient.

- Les phyla (Phylum : du grec phulon «race, tribu») sont simplement des ensembles, non vides, d'opérateurs.

A chaque occurrence d'un arbre de syntaxe abstraite est associé un phylum qui indique quels sont les opérateurs autorisés à cet emplacement. Le phylum associé à un emplacement donné ne dépend que du père de cet emplacement.

Définir la syntaxe abstraite c'est :

1 - Décider quels sont les opérateurs avec, pour chacun d'eux, leur arité et les phyla de leurs fils;

2 - Définir les phyla.

Les operateurs : définition des arbres

programme	--->	MODULE +...;
module	--->	ENTETE CORPS
entête	--->	IDENT DECLARATION *...;
corps	--->	LISTDECLA LIST BLOC
listedécla	--->	DECLARATION +...;
déclaration	--->	MODE LISTIDENT
listident	--->	IDENT +...;
entier	--->	;
chaîne	--->	;
listbloc	--->	BLOC +...;
bloc	--->	ETIQUETTE LISTINSTRUC
étiquette	--->	;
listinstruc	--->	INSTRUCTION+...;
affect	--->	IDENT EXPRES
alter	--->	TEST LISTINSTRUC
altersinon	--->	TEST LISTINSTRUC LISTINSTRUC
iter	--->	TEST LISTINSTRUC
lire	--->	IDENT
écrire	--->	EXPRES
saut	--->	ETIQUETTE
exec	--->	ETIQUETTE
plus	--->	EXPRES EXPRES
moins	--->	EXPRES EXPRES
mult	--->	EXPRES EXPRES
dire	--->	EXPRES EXPRES
1plus	--->	EXPRES
1moins	--->	EXPRES
et	--->	TEST TEST
ou	--->	TEST TEST
non	--->	TEST
égal	--->	ELEM ELEM
dif	--->	ELEM ELEM
sup	--->	ELEM ELEM
inf	--->	ELEM ELEM
ident	--->	;
nombre	--->	;
chaîne	--->	;
comment	--->	;
listcomment	--->	COMMENT*...;

Les phyla : définition des ensembles

MODULE	::=	module;
ENTETE	::=	entête;
CORPS	::=	corps;
LISTEDECLA	::=	listedecla;
LISTEBLOC	::=	listebloc;
LISTINSTRUC	::=	listinstruc;
DECLARATION	::=	déclaration;
MODE	::=	entier chaîne;
LISTIDENT	::=	listident;
BLOC	::=	bloc;
INSTRUCTION	::=	affect alter altersinon iter lire écrire saut exec;
IDENT	::=	ident;
EXPRES	::=	plus moins mult dire 1plus 1moins IDENT CONST;
CONST	::=	nombre chaîne;
TEST	::=	et ou non;
ELEMENTEST	::=	><=≠;
COMMENT	::=	comment;

L'ELIMINATION DES DEFINITIONS REDONDANTES DANS DU TEXTE SOURCE

Le processus d'élimination de définitions redondantes proposé utilise la propagation le long de la structure logique du texte source. Ce dernier est supposé constitué à l'aide des schémas de base classiques pour un programme structuré. Cette approche d'un programme a déjà été envisagée au niveau de la fonction MODULARISER.

1) Quelques définitions préalables :

• Bloc de base :

Dans un texte source, un bloc de base est un ensemble d'instructions, simples ou assimilables à une affectation généralisée, telles que le contrôle passe implicitement de l'une à l'autre dans l'ordre physique d'apparition de ces instructions dans le texte.

Etant donné un bloc de base B on affecte à chaque instruction un entier qui correspond au rang de cette instruction dans B dans l'ordre physique d'apparition des instructions.

Notation :

On abrège l'expression "instruction de rang n dans "B" sous la forme "Instruction n dans B".

• Définition d'une variable dans une instruction.

On dit qu'une variable x fait l'objet d'une définition dans une instruction i ou encore que l'instruction i définit x et on note d (x, i) si la valeur de x est modifiée à l'exécution de l'instruction i.

Par exemple x appartient à la partie gauche d'une affectation, ou x est le résultat d'une instruction de lecture.

• Référence à une variable dans une instruction.

On dit qu'une variable x fait l'objet d'une référence dans une instruction i ou encore que l'instruction i fait référence à x et on note r (x, i) s'il existe au moins une occurrence de x dans l'instruction i qui n'est pas une définition de x dans i.

Par exemple x est en partie droite d'une affectation ou x est l'objet d'une instruction de sortie.

• Ensemble des définitions d'une variable x dans un bloc de base B.

Etant donné un programme $\mathbb{P}$ soit $\mathbb{V}$ l'ensemble des variables de $\mathbb{P}$.

Etant donnée une variable x $\mathbb{V}$, visible dans B, l'ensemble $\mathbb{D}$ (B, x) des définitions de x dans B est l'ensemble des instructions qui dans B, définissent x.

On note $\mathbb{D}$ (B, x) = {instruction i ; i B et d (x, i)}.

• Ensemble des instructions qui, dans B, font référence à x.

On note $\mathbb{R}$ (B, x) = {instruction i ; i B et r (x, i)}

2) Une méthode d'élimination des définitions redondantes :

Nous proposons un algorithme d'élimination des définitions redondantes dans un bloc de base.

Etant donné un bloc de base B, pour chaque variable x visible dans B, éliminer les définitions redondantes de x dans B c'est :

1) Construire l'ensemble $\mathbb{U}$ (B, x) des couples d'instructions (j, k) dans B tels que j ≤ k,est une définition de x dans B, k est la première instruction qui fait référence à x dans B au dela de j ;

On note $\mathbb{U}$ (B, x) = { (j, k) ; j, k instructions dans B ,j ≤ k, d (x, j) et r (x, k) et ∃ d'instruction n dans B avec j < n < k telle que d (x, n) ou r (x, n) }.

2)Construire l'ensemble à annuler, $\mathbb{A}$ (B, x) , des instructions de défintion redondantes de x dans B en rapprochant finalement $\mathbb{U}$ (B, x) et $\mathbb{D}$ (B, x).

-Elément de coupure et parties d'ensemble ordonné.

Etant donné un ensemble ordonné $\mathbb{E}$, un élément de coupure ec de $\mathbb{E}$ est un élément de $\mathbb{E}$ qui partitionne $\mathbb{E}$ en deux sous-ensembles appelés respectivement partie gauche de $\mathbb{E}$, soit pg ($\mathbb{E}$), et partie droite de $\mathbb{E}$, soit pd ($\mathbb{E}$);

On note : pg ($\mathbb{E}$) = { e ; e $\mathbb{E}$, e < ec}

pd ($\mathbb{E}$) = { e ; e $\mathbb{E}$, e ≥ ec }

Remarques :

a) Un élément de coupure est défini par une proprieté ou une fonction.

b) Si l'élément de coupure existe il détermine une partie gauche éventuellement vide, une partie droite non vide.

c) Si l'élément de coupure n'existe pas alors

- $\mathbb{E}$ est vide ou
- pg ($\mathbb{E}$) est vide; il y a coupure à gauche de $\mathbb{E}$ ou
- pd ($\mathbb{E}$) est vide; il y a coupure à droite de $\mathbb{E}$.

2-1 Construction de U (B,x).

Cas d'une variable x visible dans le bloc B, avec la liste des références en majeur.

Résultat :

U (B, x) = { (δ, ρ); δ D (B, x), ρ R (B, x), ρ est immédiatement inferieur à ρ ou si δ' est le dernier élément de D (B, x) et il n'existe pas ρ' R (B,x), ρ' ≥ δ' alors (δ', ∞) }

Données :

D (B, x) = { δ; δ entier > o, instruction δ B, d (x, δ) }

R (B, x) { ρ;ρ entier > o, instruction ρ B, r (x, ρ) }

% On confond une instruction et son numéro dans B

Définition :

D' (B, x) = D (B, x)

R' (B, x) = R (B, x)

% Les intermédiaires D' et R' permettent de sauvegarder D et R.

U (B, x) = ensemble vide

Si R' (B, x) est vide alors si D' (B, x) non vide ajouter à U (B, x) le couple (m,n) avec pour m le dernier élément de D' (B, x) et pour n la valeur ∞ is.

sinon

tantque D' (B, x) non vide et R' (B, x) non vide

soit ρ le premier élément de R' (B, x) recherche de l'élément de coupure δc de D (B, x)

tel que δc est le premier élément δ de D' (B, x) tel que δ ≥ ρ

si δc existe alors si pg (D'(B, x))non vide alors ajouter à U (B, x) le couple (m, n) avec pour m le dernier élément de pg (D (B, x)) pour n l'élément ρ de R' (B, x) is

% si δc existe et pg (D' (B, x)) est vide alors il n'existe pas de définition

% précédant ρ dans B (ρ est un référence à x à risque dans B), ou

% il 'nexixte pas de définition immédiatement inférieure à ρ

si δc n'existe pas (coupure à droite de D' (B, x)) alors

pd (D' (B, x)) = vide

ajouter à U (B, x) le couple (m, n) avec pour m le dernier élément de D' (B, x) pour n l'élément ρ de R' (B, x) is

prendre pour D' (B, x) la partie droite de D' (B, x)

soit D' (B, x) = pd (D'(B, x))

retirer ρ de R' (B, x)

fintantque :

si D' (B, x) non vide (=>R' (B, x) est vide) alors ajouter à U (B, x) le couple (m, n) avec pour m le dernier élément de D'(B, x) pour n la valeur ∞

% si D' (B, x) est vide le problème est résolu.

2-2 Construction de A (B, x)

Définition de la liste à gauche de U (B, x) soit Lg (U)

Lg (U) = { m ; ∃ n, (m, n) U}

on à donc Lg (U) D

Algorithme

% Calcul de la différence Dif = D (B,x) - Lg (U (B,x))

% parcours de droite à gauche

Dif = ensemble vide

tantque D (B, x) non vide

soit δ le premier élément de D (B, x)

si δ n'appartient pas à Lg (U) alors ajouter δ à Dif

retirer δ de D (B, x)

fintantque

% Rapprochement entre Dif et U (B, x), construction de A (B, x).

A (B, x) = ensemble vide

tantque Dif non vide

soit dif le premier élément de Dif

ajouter dif à A (B, x)

recherche de (mc, nc), premier élément (m, n) de U tel que n ≤ dif

% (mc, nc) est un élément de coupure de U (B, x)

si (mc, nc) existe alors si nc = dif alors

ajouter mc à A (B, x)

soit smc = mc

% smc est une sauvegarde de mc

prendre élément suivant (m, n) dans U (B, x)

tantque U (B, x) non vide et smc = n

ajouter m à A (B, x)

soit smc = m

prendre élément suivant (m, n) dans U (B, x)

fintantque

retirer dif de Dif

fintantque

Remarques :

a) L'un des principes de la solution est le suivant : si une définition récursive est redondante et si en temps que référence à x , elle utilise une définition précédente ne servant qu'à cela alors on peut éliminer cette précédente définition.

b) Relever toutes les références utilisant une même définition peut être utile pour relever toutes les références à risque (en tête). On peut aussi relever les définitions à risque.

c) Le style de définition (avec commentaires) permet une vérification cas par cas, en particulier éviter les incohérences liées aux cas limites (premier et dernier éléments).

d) Une seconde solution consiste en plusieurs passes dans la deuxième partie de l'algorithme, éliminant à chaque fois une définition redondante de x par imbrication de niveau supérieur ou égal au rang de la passe : solution peut-être un peu plus longue mais plus simple et ne cherchant pas à traiter des cas trop particuliers (niveau d'imbrication grand des définitions récursives de x).

On peut aussi limiter le nombre de passes à une estimation moyenne (3 par exemple) et s'arrêter et prévenir l'utilisateur au-delà.

e) Un autre algorithme d'élimination : principe

Définition :

Etant donné un bloc de base B, soit $\mathbb{A}$ (B), l'ensemble des définitions à supprimer dans B et soit $\mathbb{T}$ (B) l'ensemble des variables à traiter dans B

$\forall$ x $\mathbb{T}$ (B) initialisé à $\mathbb{D}$ (B)

$\forall$ k $\mathbb{D}$ (B)

si [k + 1, k'] $\mathbb{R}$ (B) = Ø alors
  ajouter k à $\mathbb{A}$ (B)
  si k $\mathbb{R}$ (B) alors
    ajouter à $\mathbb{T}$ (B) les yi tels que
    r (yi, k)

Optimiser :

$\mathbb{R}$ (B) étant ordonné on peut utiliser $\mathbb{R}$ (B, a) sous-liste de $\mathbb{R}$ (B) commençant à l'instruction a, où a est le plus grand plus petit que k' sinon @ a.

La condition devient [k + 1, k'] $\mathbb{R}$ (B, a) = Ø donc a [k + 1, k']

f) Quelque soit l'algorithme, le coût est sensiblement le même, c'est la somme des longueurs de $\mathbb{D}$ et de $\mathbb{R}$.

g) La suppression de définitions redondantes peut conduire à la mise en évidence de variables localement "inutiles" objets de déclarations à risque.

Exemple :

Illustration partielle du problème de l'élimination de définitions redondantes à l'aide d'une représentation matricielle ou graphique du bloc de base.

Soit la séquence d'instructions B suivante :

```
1 a := b + 1
2 c := a + b
3 d := e + b
4 a := e + b + 1
5 c := a
```

Représentation par la matrice de dépendance (en valeur).

Les relations entre les variables ayant une occurrence au moins dans B sont mémorisées sous la forme d'une matrice de dépendance entre variables ; soit MDV cette matrice. Alors MDV (i, j) = {n/n est un numéro d'instruction dans B n définit v_i , n fait référence à la variable v_j ou à la constante c_j }

Pour l'exemple, la matrice MDV est la suivante :

	1	a	b	c	d	e
1						
a	1,4		1,4			4
b						
c		2,5	2			
d			3			3
e						

matrice initiale

	1	a	b	c	d	e
1						
a	4		4			4
b						
c		5				
d			3			3
e						

matrice finale

Parcours de la matrice ligne par ligne (variable définie par variable définie).

- 2ème ligne : variable a

a est défini en 1 et 4, or a est utilisé en 2 (1 < 2 < 4),(voir colonne de a), donc la définition de a en 1 est utile à priori ; a est également utilisé en 5 (4 < 5) donc la définition de a en 4 est utile à priori.

• 3ème ligne : variable b

b n'est pas défini dans B et pourtant utilisé en 1, 2, 3, et 4 (voir colonne de b) donc b doit être défini à l'entrée de B : ce sont des utilisations à risque à priori.

• 4ème ligne : variable c

c est défini en 2 et 5, or il n'existe pas d'utilisation de c dans B (donc pas entre 2 et 5) donc la définition de c en 2 est inutile ; on barre tous les 2 dans la 4ème ligne, donc a et b ne sont plus référencés dans 2.
On peut alors reprendre l'algorithme sur l'ensemble des variables ainsi (partiellement) libérées précédant celle qui a provoqué cette libération.
Ici on reprend l'ensemble {a, b}
a n'est plus référencé en 2 donc la définition de a en 1 devient inutile ; on barre les 1 dans la ligne de a (2ème ligne).
b reste utilisé en 3 et 4.

• 5ème ligne : variable d

il y a une seule définition de d en 3 ; d n'est pas utilisé dans B, donc sa définition est à risque mais ne peut être supprimée.

Le principe de l'élimination sur matrice peut s'écrire :

Pour chaque ligne l_i de MDV, ligne associée à la variable v_i, pour tout couple d'entiers consécutifs (n_p, n_r) apparaissant dans { MDV (i, j), j = 1,k } tel que dans la colonne de v_i il n'existe pas d'entier n_q, $n_p < n_q < n_r$ alors :

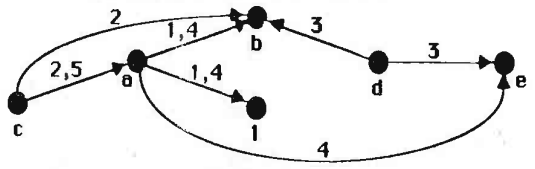
- supprimer dans l_i toutes les occurrences de n_p
- relever les variables associées aux colonnes c_k (partiellement) libérées
- reprendre l'analyse avec la première de ces variables libérées.

• représentation par le graphe de dépendance entre variables.

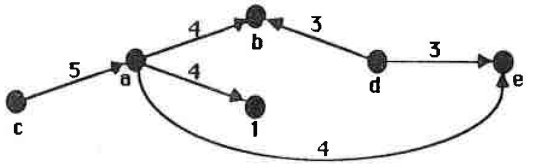
Le graphe de dépendance (en valeur) entre variables, soit GDV, est tel que :

- chaque noeud représente une variable (ou 1 constante)
- chaque arc (v_i, v_j) est tel que la variable v_i (origine de l'arc) dépend en valeur de la variable v_j (ou constante c_j) (extrémité de l'arc)
- chaque arc est décoré par l'ensemble des numéros d'instructions où existe une dépendance en valeur entre l'origine et l'extrémité.

Pour l'exemple, le graphe GDV est le suivant :



GRAPHE INITIAL



GRAPHE FINAL

Le graphe GDV peut être parcouru par variable définie c'est à dire noeud origine d'arc par noeud origine d'arc.

Pour chaque noeud n du GDV, origine d'un arc au moins,

pour tout couple d'entiers consécutifs (n_p, n_r), apparaissant sur l'ensemble des arcs d'origine n, tel qu'il n'existe pas sur les arcs d'extrémité n, un entier n_q, $n_p < n_q < n_r$ alors :

- supprimer les occurrences de n_p sur les arcs correspondants.
- supprimer les arcs qui ne sont plus décorés
- reprendre l'analyse des noeuds extrémités d'arcs (partiellement) libérés de décoration.

Le bloc de base final est donc :

d := e + b
a := e + b + 1
c := a

3) L'élimination de définitions redondantes dans les principales structures de contrôle : séquence, alternative, itération.

3-1 Cas de la séquence de blocs

Ce cas peut se présenter quand il y a des étiquettes inutiles dans le texte source ou dans le cadre de concaténation de blocs dans le processus d'élimination de définition par exemple dans une structure alternative.

Considérons le cas élémentaire de deux blocs consécutifs B1 et B2.

Dans le cas de n blocs consécutifs, il suffit, dans le cadre d'un

processus itératif, ayant terminé le traitement des i premiers blocs réduits à un seul bloc assaini de rapprocher ce dernier du bloc B_{i+1} à la manière du cas élémentaire de deux blocs.

Hypothèses :

1) Les blocs B1 et B2 sont supposés assainis localement.
2) Pour chaque variable x ayant une occurrence dans B1 ou B2 il est nécessaire de connaître le type (définition ou référence) de sa dernière occurrence dans B1 et de sa première occurrence dans B2 : cela revient à prendre en considération références et définitions à risque entre B1 et B2.

L'élimination d'une définition redondante en fin de bloc B1 nécessite alors la relance du processus d'élimination :

exemple :

B1 1 x : = y
2 x : = x + y + 1
3 y : = 2
B2 4 x : = y + 3
5 z : = 5

Dans B1 la derniere occurrence de x correspond à une définition recursive de x à risque.

Dans B2 la premiere occurrence de x est une définition

Donc pour la séquence B1, B2 l'instruction 2 est redondante, et l'itétration du processus révèle la redondance de 1 également.

3-2 Cas de la structure alternative.

Soit le schéma de l'alternative suivant :

A :: = si p (x_1,...,x_n) alors BV
sinon BF

où p (x_1,...,x_n) est un prédicat, BV et Bf sont des blocs de base assainis c'est à dire dont on a éliminé les définitions localement redondantes.

Il est nécessaire de supposer que toutes les instructions de B1 et B2 font référence à x_1,...,x_n (dépendance en mode de calcul ou en contexte).

Le bloc de synthèse associé à l'aternative ne conserve alors que les instructions

-de type de définition de x s'il existe au moins une définition de x dans l'un des deux blocs B1 ou B2

-de type de référence à x s'il existe au moins une référence à x dans l'un des deux blocs B1 ou B2

Un tel bloc de synthèse peut alor subir le mécanisme d'élimination et être concaténé aux autres blocs du programme à condition pour chaque variable d'indiquer les références (définitions) à risque sur chaque branche de l'alternative (B1 et B2).

3-3 Cas de la structure itérative

soit le schéma de l'iteration suivant :

I ::= tantque p (x_1,...,x_n) faire
B
fintantque

Il est nécessaire également de supposer que les variables x_1,...,x_n sont référencées dans chaque instruction de **B**

On considère alors le bloc B ainsi modifié et on procéde à l"elimination sur le bloc (B,B) (concaténation de B avec lui-même) soit B^2

La composition de l'itération avec les autres blocs du programme utilise la représentation par B^2 de cette itération y compris pour les références (définitions) à risque.

CONCLUSION

A condition d'éffectuer sur un programme structuré les transformations précédentes au niveau de chaque structure de contrôle notre algorithme d'élimination peut ainsi operer par propagation de bloc en bloc dans un programme.

Ce processus peut-être ensuite réalisé dans le cadre d'une représentation matricielle (ensemblede MDV) ou graphique (ensemble de GDV).

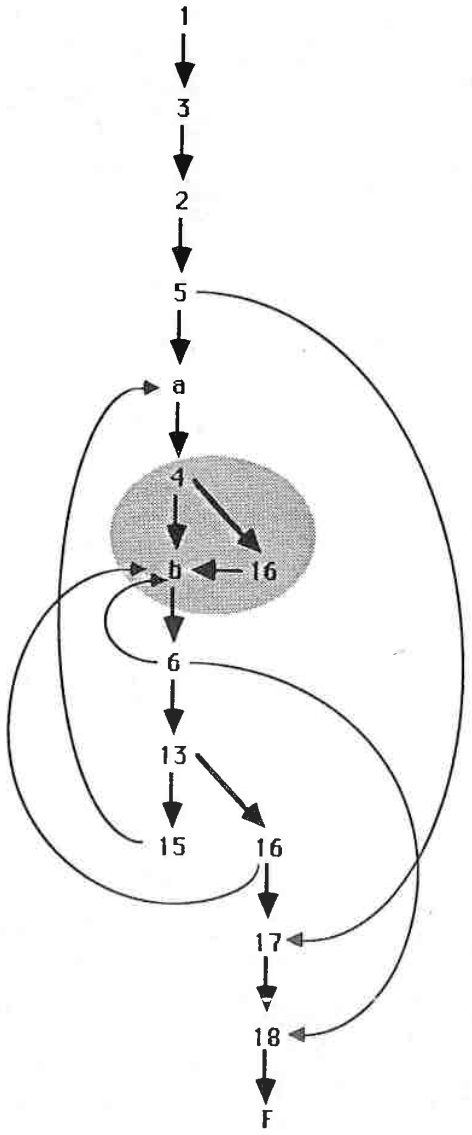
EXEMPLE D'AMELIORATION DE PROGRAMME

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PMAJSR.
AUTHOR. LUI.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. DPS-6.
OBJECT-COMPUTER. DPS-6.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
      SELECT FPRIX ASSIGN DISK.
      SELECT FMODIF ASSIGN DISK.
      SELECT FPRIXMAJ ASSIGN DISK.
DATA DIVISION.
FILE SECTION.
FD FPRIXMAJ LABEL RECORD STANDARD.
01 ARTPRIXMAJ.
      02 NMAJ PIC 9(5).
      02 DESMAJ PIC X(30).
      02 PHTMAJ PIC 9(5)V9.
      02 TAXEMAJ PIC 9(3)V9.
      02 DMAJ.
            03 J PIC 99.
            03 M PIC 99.
            03 A PIC 99.
FD FMODIF LABEL RECORD STANDARD.
01 ARTMODIF.
      02 CC PIC 9.
      02 C PIC 9(5).
      02 CREA.
            03 DESCREA PIC X(30).
            03 PHTCREA PIC 9(5)V9.
            03 TAXECREA PIC 9(3)V9.
            03 DCREA.
                  04 J PIC 99.
                  04 M PIC 99.
                  04 A PIC 99.
      02 MODIF REDEFINES CREA.
            03 PRIX PIC 9(5)V9.
            03 FILLER PIC X(40).
FD FPRIX LABEL RECORD STANDARD.
01 ARTPRIX.
      02 N PIC 9(5).
      02 DES PIC X(30).
      02 PHT PIC 9(5)V9.
      02 TAXE PIC 9(3)V9.
      02 D.
            03 J PIC 99.
            03 M PIC 99.
            03 A PIC 99.
WORDKING-STORAGE SECTION.
77 FINFPRIX PIC 9 VALUE O.
77 FINFMODIF PIC 9 VALUE O.
77 DJ PIC 9(6).
```

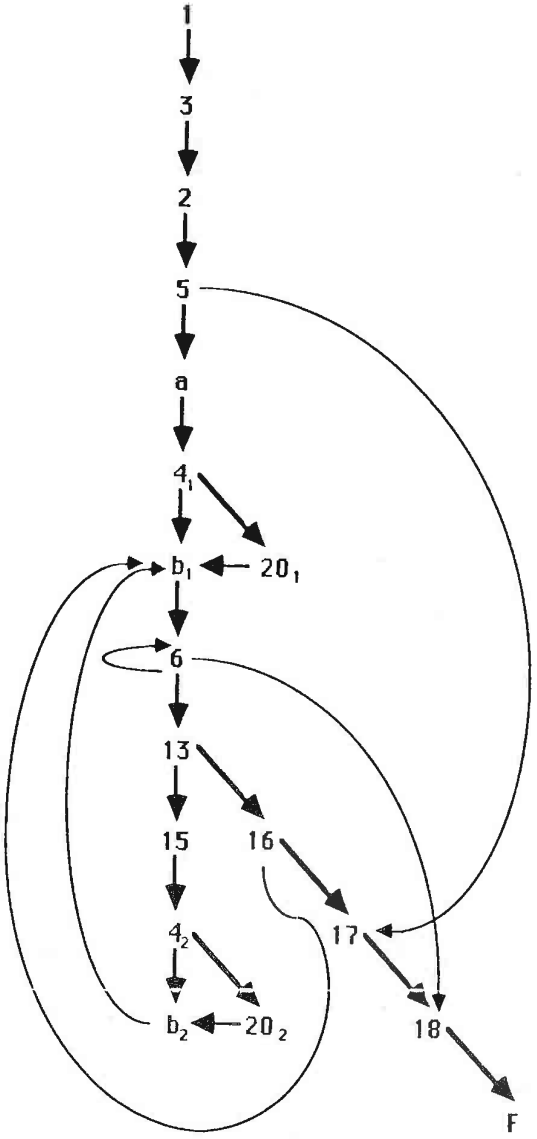


```
PROCEDURE DIVISION.
1     TP.
2           MOVE 0 TO FINFPRIX.
3           MOVE 0 TO FINFMODIF.
4           GO TO OUVERT2.
5     OUVERT1.
6           OPEN INPUT FPRIX OUTPUT FPRIXMAJ.
            ----
7           GO TO LECT 2.
8     OUVERT2.
9a9b        OPEN INPUT FMODIF GO TO OUVERT1.
10    LECT1.
11          READ FPRIX AT END GO TO FINAL.
            ----
12          MOVE ARTPRIX TO ARTPRIXMAJ.
            ----
13          GO TO TEST1.
14    LECT2.
15          READ FMODIF AT END MOVE 99999 TO C GO TO
            FINMODIF3.
            ----
16          GO TO LECT1.
17    TEST1.
18          IF N = 99999 AND C = 99999 GO TO FINTRAITMAJ.
19          IF C= N PERFORM TMODSUP THRU FINTMODIF
            GO TO TEST1
20          ELSE GO TO TRECREA.
21    TMODSUP.
22          IF CC = 2 GO TO TMODIF.
23          READ FPRIX AT END MOVE 99999 TO N.
24          READ FMODIF AT END MOVE 99999 TO C GO TO
            FINMODIF3.
25          GO TO FINTMODIF.
26    TMODIF.
27          GO TO AFFECTMODIF.
28    RTMODIF.
29          PERFORM ECRIRE
30          READ FPRIX AT END MOVE 99999 TO N.
31          READ FMODIF AT END MOVE 99999 TO C
            GO TO FINMODIF3.
32    FINTMODIF. EXIT.
33    AFFECTMODIF.
34          MOVE ARTPRIX TO ARTPRIXMAJ.
35          MOVE PRIX TO PHTMAJ.
36          ACCEPT DJ.
37          MOVE DJ TO DMAJ.
38          GO TO RTMODIF.
39    ECRIRE.
40          WRITE ARTPRIXMAJ.
41    TRECREA.
42          IF C < N ALTER REOUCREA TO PROCEED TO
      CREA ELSE
43          ALTER REOUCREA TO PROCEED TO RECOP.
44    REOUCREA.
45          GO TO.
```

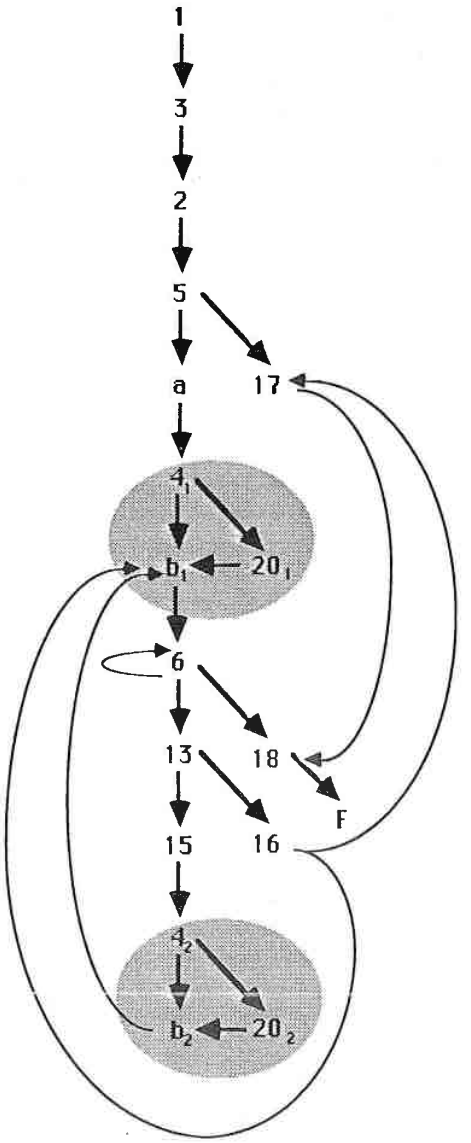
```
46   RECOP.
47        MOVE ARTPRIX TO ARTPRIXMAJ.
48        WRITE ARTPRIXMAJ.
49        GO TO LECT1.
50   CREA.
51        MOVE C TO NMAJ.
52        MOVE DESCREA TO DESMAJ.
53        MOVE PHTCREA TO PHTMAJ.
54        MOVE TAXECREA TO TAXEMAJ.
55        MOVE DCREA TO DMAJ.
56        READ FMODIF AT END MOVE 99999 TO C
          GO TO FINMODIF3.
57        GO TO TEST1.
58   FINMODIF3.
59        PERFORM FINALRECOP UNTIL N = 99999.
60   FINTRAITMAJ.
61        CLOSE FPRIX FPRIXMAJ FMODIF.
62        STOP RUN.
63   FINALRECOP.
64        MOVE ARTPRIX TO ARTPRIXMAJ.
66        WRITE ARTPRIXMAJ.
67        READ FPRIX AT END MOVE 99999 TO N.
68   FINAL.
69        MOVE 99999 TO N.
70        GO TO TEST1.
```

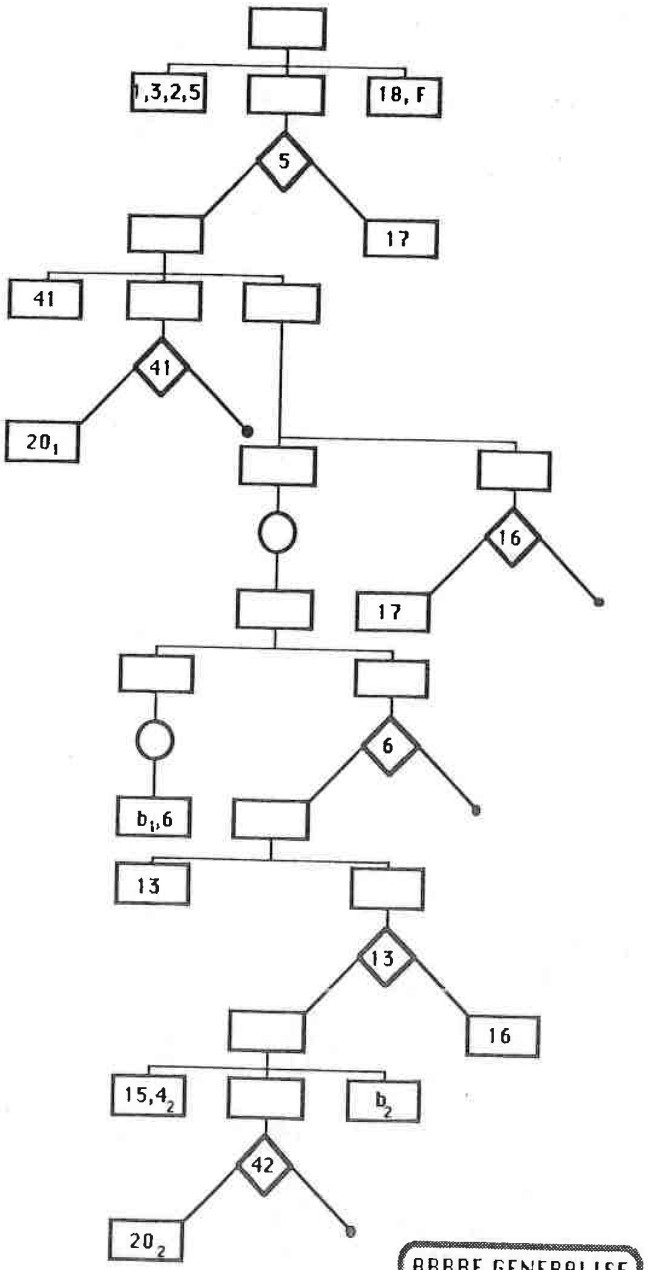
D F S T du graphe de branchement



REPRESENTATION DU GRAPHE
à l'appui d'un D F S T



GRAPHE planaire réductible



ARBRE GENERALISE

BIBLIOGRAPHIE

• ABITEBOUL S.88, GRUMBACH S. "COL a Logic-based Language for Complexe Objets" PRoc. Int. Conf. Extending Database Technology, Venice,88

• ADAM A. 78 Utilisation des transformations sémantiques pour la correction automatique des programmes. Thèse d'état Université Pierre et Marie Curie (Paris,6).

• AHO Alfred V. et Ullman, Jeffrey D. 1979.Principales of Compiler Design. Addison-Wesley Publishing Company 408-517

• AHO, A.V. et Ullmann J.D. The Théory of Parsing, Translation, and Compiling vol 2 : Compiling. Prentice-Hall Englewood Cliffs 1973. 845-963

• ALLEN F.E. 76 et COCKE J. A program Date Flow Analysis Procédure. Comm. of ACM 76 Vol 19,3 137, 147.

• ALLEN F.E. et COCKE J. A program data flow analysis procédure. Comm. ACM 19,3-1976. 137-147

• ALLEN, F.E. 72, COCKE J. Graph-théoretie constructs for program contral flow analyses IBM R.R. RC 3923 T.J. Watson Research Center Yorkstown Heights 72

• ARSAC

• ASHCROFT E. et MANNA Z. Translating program schémas to while schémas SIAM J. 4, 2, 75, 125-46

• AYOUB BAHSOUN HASSAN, 86 MODULOMETRE : Outil d'évaluation de la modularité. Conception préliminaire et choix dans le développement. Rapport N° 86-R-134-CRIN 545O6 Vandoeuvre-les-Nancy.

• AYOUB, Bahsoun Hassan 86, BARTHELEMY C. DERNIAME J.C. Vers une évaluation plus précise de la structure modulaire : application sur la relation d'appel. CRIN.

• BAKER B.S. 77 An Algorithm for Structuring Flowgraphs Bell Laboratories, Muray Hill, New Jersey. Journal of ACM 24, 1, 77, 98-120.

• BAKER B.S. et KOSARAJV S.R. A comparison of multilevel break and next statements. J of ACM 26, 79, p555-66

• BARRETT W.A. 79 et J.D. COUCH. "Compiler Construction : Théory and Practive". C 1979 Science Research Associates.

• BARTHELEMY C. 86, AYOUB BAHSOUN H. DERNIAME J.C. "Vers une évaluation plus précise de la structure modulaire, application à la relation d'appel". Rapport CRIN I

• BASILI V.R 82 et MILLS H.D. Understanding and Documenting Programs IEEE Trans. on S.E. 8, 3, 82.

• BASILI VICTOR R. 82 et MILLS H.D. Understanding and Documenting Programs IEEE T. on Software Engineering SE-8, 3, 82

• BELAID A.87 Rapport de fin de convention CRIN-ETCA. Perception multicapteurs. Langage de traitement d'images haut niveau. Rapport 87-R-114.

• BENALI Khalid 89 Assistance et pilotage dans le développement de logiciel. Thèse de l'Université de Nancy I (à paraître).

• BERGE C. 70 Graphes et Hypergraphes DUNOD,70

• BOCHMANN 73 Multiple exits from a loop without the GOTO CACM 16 (7), 73

• BOHM. C. 66 , JACOPINI G. Flow diagrams turing machines and languages writh only two formation rules. Com. ACM 9, 5, 366-371.

• BOUSSARD J.C. et DUBY J.J. Rapport d'évaluaion d'Algol 68 R.I. R.O. 5ème année 71 p 15-106.

• BRODIE,M. 84, MYLOPOULOS, J; SCHMIDT Y. "On conceptual modelling", SHM + Springer-Verlag, New-York, 84.

• CAUVET C. 87 et PROIX C, TRECOURT D. "Une base de règles pour la modélisation des aspects dynamiques des systèmes d'information". Université PARIS 1 IAEFP 87

• CHABRIER J. 79 , PROCH K. "Manuel d'utilisation d'A.T.M. Rapport CRIN 79-R.081 NANCY.

• CHAK SOK Sakhonn 80 Evolutivité du Logiciel. Thèse de doctorat de 3ème cycle informatique NANCY.

• CHEN P.P. 76 The entity-relationship model : Toward an unified view of data ACM-TODS 1,1 76 p 9-36

• CLARKE L.76 "A system to Générate Fast Data and Symbolically Exécute Programs" IEEE Trans SE, 76, p215-22.

• COHEN E. "Information Transmission in Computational Systems" Proc. of ACM Symp. on O.S. Principles, 77, p133-39

• COUTAZ J. 86, "Abstractions pour Interfaces Interactives TSI vol 5, 4, 86.

• CZEJDO Bogdan 85 et RUSINKIEWICZ Marck Program Transformations and their application in teaching procedural and non procedural Languages; UNIVERSITY of HOUSTON ACM !(,3

• DAVCEV D. 84 Software Métrics for Modular Programming Systems", Séminaire sur les méthodes quantitatives en Génie Logiciel Sophia-Antipolis 84 p 19-24.

• DE BALBINE G. Using the Fortran structuring engine In Proc. of Comp. Sci. and Stat. : Symp. on. Interface Los Angeles 75 297-305

• DERNIAME J.C. 67 "Algorithme de construction du graphe réduit d'un graphe" Congrès AFCET, 67

• DERNIAME J.C. 74 "Le projet CIVA" Thèse de doctorat d'Etat 74

• DERNIAME J.C. 79, FINANCE J.P. "Types abstraits de données : Spécification, utilisation et réalisation". Ecole d'été AFCET. MONASTIR Juillet 79.

• DIJK STRA E.W. 76 A discipline of programming. Prentice Hall séries in Automatic Computation, New Jersey.

• DIJKSTRA 68 GOTO Statement Considered Harmful CACM 11 (3), 147-48, 68

• DONZEAU-GOUGE V. et G, KAHN G., LANG B. Introduction au système MENTOR et à ses applications IRIA-LABORIA

• ED ACLY Initiatives in Reverse Engineering : The Impossible, the Essential, and the Disappointing. SOFTWARE Technologie Service IDC

• ELLUL A.87, Couplage entre mécanismes de déduction et base de données. Centre de Recherche Bull. Des bases de données aux bases de connaissances; AFCET.

• FUKUNAGA Koïchi

• GARDARIN G. 88 PUCHERAL P., THEVENIN J.M., STEFFEN H. BELLOSTA M.J. BIGAGLI D. DARRIEUMERLOU C. FARGEST. GEODE "Un serveur d'objets extensibles pour SGBD en mémoire primaire" Journées BDD et BDOO, AFCET, PARIS, 88

• GARDARIN, 86 "Nouvelles perspectives des bases de données" Eyrolles,86

• GILB Tom 77 "Software Métrics" Winthrop publishers Inc.77 Cambridge, Massachussetts-p 88-90, 182-188.

• GODART C. 87, BENALI,K, BOUDJLIDA,N. CHAROY F, DERNIAME J.C. Les bases de données sur le chemin du Génie Logiciel. CRIN-NANCY - DES bases de données aux bases de connaissances (AFCET).

• GOLDBERG, 76 and KAY. "Smalltalk 76 Instruction Manual" XEROX..76 SSL 76-6.

• GRAM. A., 86 Groupe de travail A.F.C.E.T. (GROPLAN). Raisonner pour programmer. BORDAS

• HALSTEAD M.H. 77 "Elements of Software Science" Elsevier North Hollan, New York 77.

• HECHT M.S. 74 et ULLMAN J.D. University of Mariland, Princeton University Charactérizations of Réducible Flow Graphs. Journal of ACM 21, 3, 74, 367-75.

• HECHT M.S. 74 et ULLMANN J.D. Characterizations of Reducible Flow Graphs. Journal of the ACM Vol 21, 3, 74, 367-375.

• HILL, Jerome Anthony 86 Translation of one high-level language to another : COBOL to ADA; U.M.I. Dissertation Information Service.

• HOWDEN W.E. 80 "Functional Program Testing" IEEE Trans. on S.E. 6, 80, p162-69

• KEBAILI A. 84 Une contribution à l'étude du problème de l'évolution de structure dans les systèmes d'informations historiques Projet REMORA. Thèse de troisième cycle, Nancy, 84

• KIM M.J 86, TOUNSI, N, DERNIAME J.C. Cohérence d'un schéma de base de données de type Entité-Association INFOCOM-86 PARIS 86.

• KING J.C.75 "A new approach to program testing" Proc. Int. Conf. Reliable Software 75, p228-33.

• KNUTH 74 Structured Programming with GO TO statements ACM Computing Surveys 6 (4) 74.

• KNUTH D.E. et FLOYD R.W. Notes on avoiding "go to" statements Infor. Proc. Letters 1, 71, 23-31

• KOSARAJU RAO, S. 74 Analysis of structured programs, J. Comput. System Sci 9, 232-255.

• LAKHOTIA 83 An improvement over "An improvement over decply nested IF-THEN-ELSE control structures" ACM SIGPLAW Notices 18 (5), 83

• LAMPSON B.W. and al. 77 "Euclid Report" Xerox Research Center Palo Alto 76. SIGPLAN Notices 12, 2-77.

• LE MAITRE J., 88 "Le langage de manipulation de base de données GRIFFON JISI,88 Tunis, 88.

• LISKOV B.H 75 , ZILLES S. "Spécification techniques for Data Abstractions". International Conférence ou Reliable Software. Los Angeles 75. Sigplan Notices 10, 6 72-87, 75.

• LISKOV B.H 77, ZILLES S. "Introduction to formal spécifications of Data abstractions" Current Trends in Programming Méthodologig. T.T. Yeh. ed. Vol 1 Prentice Hall 77.

• LIVERCY C. 78 Théorie des programmes; schémas, preuves, sémantique CRIN-UNIVERSITES NANCY I et II avec le concours du CNRS, DUNOD.

• MADELAINE J.,84 "Deductive Data Bases in SABRE" Workshop on Data-Base Machines, Minnowbrook (New York), 84.

• MADELAINE, 84 Interface déductive sur SABRE

• MAHER B. et SLEEMAN D.H. A data driveu system for syntactic transformations SIGPLAN Notices (ACM) 1981 50-52.

• MAHER B. et SLEEMAN D.H. Automatic Program Improvement : Variable Usage Transformations. The University of Leeds ACM TPL S Vol 5 No 2 1983. 236-264.

• Mc CABE T.J. 76 "A complexité Measure" IEEE Trans. On software Engineering, vol SE-2, N° 4, 76.

• MEYER B. 78 et BAUDOIN C. "Méthodes de programmation" Editions EYROLLES Recherches EDF 78

• MILLER E.F. "Program Testing Technology in The 1980s" Oregan Report Proc of The Conf. of Computing in The 1980s p405.

• MILLER J.CRIS 87 et BURTON M.Strauss III Implications of Automated Restructuring of COBOL. Peat Marwick / The Catalyst Group - Chicago. SIGPLAN NOTICES V22, 6 June 87.

• MILLS H.D. 70 Syntax-Directed Documentation For PL 360 IBM, Gaithersburg, Maryland. Comm. ACM 13, 4, 70.

• MYERS G.J. SOFTWARE Reliability John Wiley and Sons

• NEWMAN 83 IF-THEN-ELSE, again ACM SIG PLAN Notices 18 (12) 83

• NGUYEN et RIEU "Manipulation d'objets dynamiques dans les bases de données" INRIA:GRENOBLE, Laboratoire GI, 87.

• NGUYEN, 86 "Quelques fonctionnalités des bases de données avancées" Thèse d'Etat. Université de Grenoble,86 Manipulation des schémas d'objets

• NICOLAS, 83 et YAZDANIAN K. "An outline of BDGEN : a déductive DBMS; In Proc. of IFIP,83 North Holland,84

• NICOLAS, 83 Prototype BD GEN (génération de données dans les BD)

• ODIENNE P. "Sémantique des objets et implémentation relationnelle pour gérer la présentation et la dynamique des applications" Sema-metra/Cerci Isem PARIS-SUD, 87

• PARENT C., 87 et SPACCAPIETRA S. "Un modèle et une algèbre pour les bases de données de type entité-association". TSI vol 6 n° 5, 87

• PETERSON W.W. et KASAMI T. et TOKURA N. On the capacilities of while, repeat and exit statements Comm. ACM 16, 8, 73, 503-12

• PITRAT J. A Général Game Playing Program. Artificial Intelligence and Heuristie Programming. Edinburg University Press 71.

• PORTMANN M.C. 74 thèse de spécialité. UTILISATION des hypergraphes pour l'évaluation automatique des tailles de données (REMORA) 74.

• PORTMANN M.C. 74 Utilisation des hypergraphes pour l'évaluation automatique des tailles de données. Thèse de spécialité NANCY I.

• PRC Rapport 89 "Rapport d'activité du PRC Base de Données 3ème génération " Juin 89

• PROCH K 80 "Introduction des types abstraits en PASCAL". Rapport CRIN 80 P 013.

• RAMSHAW LYLE 88 Eliminating go to's while Preserving Program Structure Digital Equipment Corporation Systems Research Center Palo Alto California. J of ACM Vol 35 N° 4, 88, 893-920.

• RIEU, 85 "Modèle et fonctionnalités d'un SGBD pour les applications CAO" Thèse Doctorat INP de Grenoble,85

• ROSS D. T,77 et K.E. SCHOMEN "Structured Analysis for Requirements Définition" IEEE Trans. on SE vol 3, 77 p6-15

• ROUSSEAU

• SIMON E.88, PORTIER M.A., PARTHER M. "RDL 1, manuel de programmation" Rapport ESPRIT,88

• SMITH J.M. et C.P., 77 a Database Abstractions : Aggrégation. Communications of ACM, 77

• SMITH J.M. et C.P., 77 b Database Abstractions : Aggrégation and Généralization ACM Transactions on Databse Systems, 77.

• SOFTWARE Process Workshop 88. Proceedings of the 4 th International SPW ACM Press.

• STRATFORD M.J. 82-COLLINS ADA : A Programmer's Conversion Course; Ellis Horwood-Publishers. COMPUTERS and Their APPLICATIOWS.

• TABOURIER

• TARJAN, R.E. 72 Depth-first search and linear graph algorithms SIAM J.Computing 1, 2, 72, 146-60.

• TEICHROEW D.77 et E.A. HERSHEY III; "PSL/PSA : A computer-Aited Technique for structured Dodumentation and Analysis of Information Processing Systems" IEEE Trans.on. S.E. vol 3, 77, p41-48.

• TIGRE, 85 Gestion des historiques pour la base de données généralisées TIGRE Rapport TIGRE 29-85.

• VERRAND (LE), 82 Le langage ADA Manuel d'évaluation; AFCET; Agence de l'Informatique; DUNOD BORDAS 82 ISBN 2-04-015499-X

• WAGNER J. 88 Graphic Computer-Aided Reverse Engineering (CARE) Siemens AG Application Programs.

• WILLIAM D.O. Structured transfer of control DATA Handling Division CERN CH-1211 Geneva 23.

• WILSON A.G. 83, E.A. DOMESHECK, E.L. DRASCHER, J.S. DEAN "The Multipurpose presentation system" 9 th Conference on Very Large DB Proceedings, 83

• WULF 75. WULF W., LONDON R.L. , SHAW M. "Abstraction and Vérification in Alphard". In New Directions in algorithmic Languages. Schumann ed. IFIP. W G 2.1 Working Conference Munich 75-IRIA 76.

• WULF 78 WULF W. "An Informaf Définition of ALPHARD (Préliminary)". Computer Science Dpt. CMU-CS - 78-105. Carnegic Mellon U 2/78.

• YUEN 84 Further comments on the premature loop exit problem ACM SIGPLAN Notices 19 (1) 84.

• ZELKOWITZ M.V. 78 "Implémentation of a capability-based Data Abstraction". IEEE Trans. Soft. Eng. V. SE 4 n° 1 Jan.78

INDEX

NOM DE L'ETUDIANT : Monsieur BARTHELEMY Charles

NATURE DE LA THESE : Doctorat de l'Université de NANCY I en Informatique

VU, APPROUVE ET PERMIS D'IMPRIMER

NANCY, le - 5 DEC. 1989 n° 2414

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

UNIVERSITE DE NANCY I LE PRESIDENT

M. BOULANGE

Amélioration assistée de programmes par objectifs

L'amélioration assistée de programmes concerne tous les programmes existants dont on souhaite augmenter l'une ou l'autre des qualités. L'utilisateur, dans le cadre de stratégies, définit les objectifs à atteindre essentiellement en termes de facteurs de qualité à améliorer.

Le programme subit alors un certain nombre de changements d'états, le conduisant d'une version à l'autre, grâce à la transformation de structures de représentations de celui-ci choisies par l'utilisateur. Ce dernier disposant d'un ensemble de fonctions d'amélioration met en œuvre des tactiques dont il peut suivre l'évolution intrinsèque ou comparative sur un arbre d'amélioration et dont il peut mesurer les effets grâce à l'élaboration de métriques "signifiantes". L'assistance à l'amélioration est décrite comme dans ALF par des règles et des contraintes. Une maquette de système d'amélioration et sa mise en œuvre pour faire du Reverse-Engineering ou de la traduction assistée de programmes (de COBOL vers ADA par exemple) complète la présentation de ce travail.

Mots-cles Génie Logiciel, Reconstruction de logiciel, Transformation de programmes, Modèle, Qualité