Titre: Title:	Définition d'un modèle pour conserver l'information des programmes sources (modèle DATAGRAPH) pour le logiciel DATRIX
Auteurs: Authors:	Mario Simoneau, André Beaucage, & Pierre N. Robillard
Date:	1989
Type:	Rapport / Report
Référence: Citation:	Simoneau, M., Beaucage, A., & Robillard, P. N. (1989). Définition d'un modèle pour conserver l'information des programmes sources (modèle DATAGRAPH) pour le logiciel DATRIX. (Rapport technique n° EPM-RT-89-21). https://publications.polymtl.ca/9611/

Institution:	École Polytechnique de Montréal
Numéro de rapport: Report number:	EPM-RT-89-21
URL officiel: Official URL:	
Mention légale: Legal notice:	

Définition d'un modèle pour conserver l'information des programmes sources (modèle DATGRAPH) pour le logiciel DATRIX

Mario Simoneau, Ing. jr,
Étudiant M. Sc. A.

André Beaucage, Ing.,
Associé de recherche

Pierre N. Robillard, Ph. D., Ing.,
Directeur du laboratoire de Recherche en Génie Logiciel

ÉCOLE POLYTECHNIQUE DE MONTRÉAL
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
SECTION INFORMATIQUE
LABORATOIRE DE GÉNIE LOGICIEL

SEPTEMBRE 1989

Ce document a pu être publié grâce à une subvention du Conseil de recherches en sciences et en génie du Canada (CRSNG) ainsi qu'un contrat de recherche et de développement de BELL CANADA.

Dépôt légal, 4^{e} trimestre 1989
Bibliothèque nationale du Québec
Bibliothèque nationale du Canada

Pour se procurer une copie de ce document, s'adresser au:

Service de l'édition
École Polytechnique de Montréal
Case Postale 6079, Succursale A
Montréal (Québec) H3C 3A7
(514) 340-4000

Compter 0,10 $ par page (arrondir au dollar le plus près) et ajouter 3,00 $ (Canada) pour la couverture, les frais de poste et la manutention. Régler en dollars canadiens par chèque ou mandat-poste au nom de l'École Polytechnique de Montréal. Nous n'honorerons que les commandes accompagnées d'un paiement, sauf s'il y a eu entente préalable dans le cas d'établissements d'enseignement, de sociétés ou d'orgamismes canadiens.

TABLE DES MATIÈRES

1. Introduction 1

2. Composantes du modèle DATGRAPH 2
2.1 Théorie des graphes 2
2.2 Définition des termes généraux 6
2.3 Description de l'ensemble du modèle 6

3. Graphe du flux de contrôle 11
3.1 Représentation graphique du graphe du flux de contrôle 11
3.2 Caractéristiques conceptuelles de la construction du graphe 13
3.3 Formalisation de la construction du graphe du flux de contrôle 14
3.3.1 Règles de traduction (Production du graphe initial) 14
3.3.2 Règles de réduction 16
3.3.3 Règles de concaténation 18
3.3.4 Règles d'association 20
3.4 Exemples de construction de graphe pour les structures de base 22
3.4.1 Structures de base du Fortran 22
3.4.2 Structures de base du Pascal 24
3.4.3 Structures de base du langage C 26
3.5 Définition du contenu d'un point d'attache 31
3.6 Définition du contenu d'un segment 32
3.6.1 Définition du contenu des segments primaires 34
3.6.2 Définition des opérations utilisées sur le contenu 34
3.6.3 Définition des éléments conservés selon les règles de réduction 36
3.6.4 Définition des éléments conservés selon les règles de concaténation 36
3.6.5 Définition des éléments conservés selon les règles d'association 37

4. Graphe de construction 38
4.1 Construction du graphe 39
4.2 Éléments du graphe de construction 45
4.3 Définition du contenu d'un sommet 46
4.4 Définition du contenu d'un arc 48

5. Graphe du flux de données 50
5.1 Caractérisation du transfert de données 50
5.2 Construction et représentation interne du graphe 51
5.3 Définition du contenu d'un sommet 52
5.4 Définition du contenu d'un arc 53
5.5 Exemple de construction du graphe du flux de données 53

6. Graphe d'appel 54
6.1 Construction du graphe 54
6.2 Définition du contenu d'un sommet 55
6.3 Définition du contenu d'un arc 55

7. Graphe des déclarations **56**
7.1 Construction du graphe **56**
7.2 Définition du contenu d'un sommet **57**
7.3 Définition du contenu d'un arc **57**

8. Conclusion **59**

9. Références **60**

TABLE DES FIGURES

Figure 1 : Utilisation du modèle DATGRAPH 1
Figure 2 : Exemple général d'un graphe 3
Figure 3 : Tableau et matrice associée d'un graphe 5
Figure 4 : L'analyse d'un source vis-à-vis le modèle DATGRAPH 7
Figure 5 : Liens possibles dans le graphe des déclarations 9
Figure 6 : Relations possibles lors de l'appel d'une sous-routine (A appelle B) 10
Figure 7 : Représentation du graphe de contrôle pour les entrées et les sorties 16
Figure 8 : Code source et graphe de contrôle initial correspondant 16
Figure 9 : Énoncé des règles de réduction 17
Figure 10 : Exemple d'utilisation des règles de réduction 18
Figure 11 : Énoncé des règles de concaténation 19
Figure 12 : Exemple d'utilisation des règles de concaténation 20
Figure 13 : Énoncé des règles d'association 21
Figure 14 : Exemple d'utilisation des règles d'association 22
Figure 15 : Code source pour le graphe de construction 40
Figure 16 : Exemple de graphe de construction 40
Figure 17 : Code source pour le graphe de construction 42
Figure 18 : Exemple de graphe de construction 42
Figure 19 : Code source pour le graphe de construction 44
Figure 20 : Exemple de graphe de construction 44
Figure 21 : Schématisation des entrées-sorties d'une routine 51
Figure 22 : Exemple de graphes de flux de données 52
Figure 23 : Exemple de graphe de construction 53
Figure 24 : Exemple de graphe de flux de données 53

TABLE DES TABLEAUX

Tableau 1 : Définition du symbolisme de la théorie des graphes 3
Tableau 2 : Positionnement des graphes du modèle DATGRAPH 7
Tableau 3 : Énumération des règles de réduction 17
Tableau 4 : Caractéristiques de concaténation des segments primaires 18
Tableau 5 : Énumération des règles de concaténation 19
Tableau 6 : Énumération des règles d'association 21
Tableau 7 : Regroupement des éléments composant une expression booléenne 33
Tableau 8 : Contenu des segments primaires 34
Tableau 9 : Définition des opérateurs sur le contenu des segments 35
Tableau 10 : Exemple de l'utilisation des opérateurs sur le contenu 35
Tableau 11 : Contenu des segments après l'application des règles de réduction 36
Tableau 12 : Contenu des segments après l'application des règles de concaténation 36
Tableau 13 : Contenu des segments après l'application des règles d'association 37
Tableau 14 : Exemples d'utilisation des champs des sommets du graphe de construction .. 48

1. Introduction

Dans le cadre d'un projet de recherche ayant comme but le développement d'un logiciel d'évaluation de code source, un modèle général devant permettre de conserver l'information du code source de langages procéduraux a été élaboré.

L'information contenue dans les fichiers de code source est extraite par différents analyseurs spécifiques aux langages (voire même aux différents dialectes d'un langage) analysés. Cette information doit être transformée de façon à obtenir une seule représentation pour tous les langages de programmation. À partir de cette représentation, différentes requêtes pourront être effectuées sans avoir à considérer la syntaxe, la grammaire et les divers caractéristiques des langages analysés. La figure 1 permet de représenter ce cheminement.

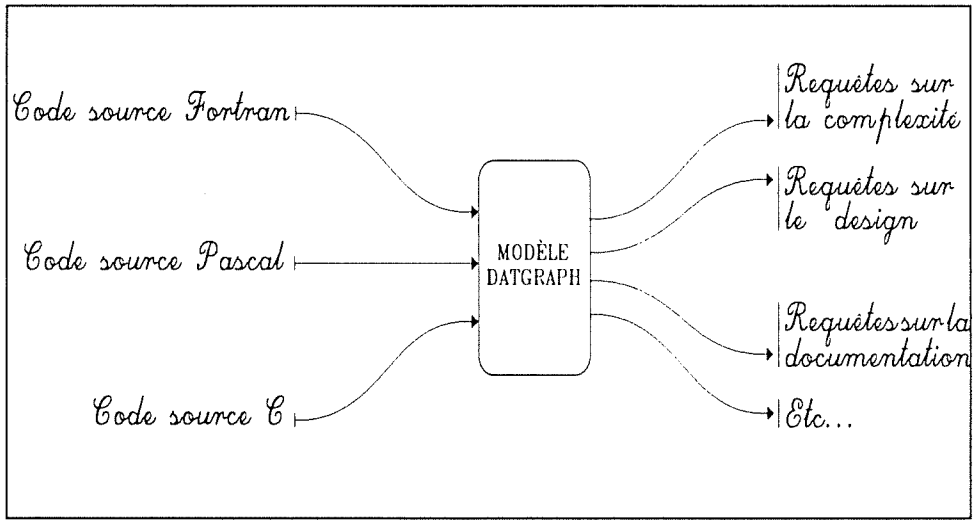


Figure 1 : Utilisation du modèle DATGRAPH

L'information contenue dans ce document part d'un niveau conceptuel général et descend jusqu'au niveau précédent le design d'implantation. Cette recherche ayant été effectuée en étroite relation avec certains concepts déjà implantés, il apparaît raisonnable de considérer un niveau de détail assez précis.

De façon à aider le lecteur à comprendre les divers notions exposées, des exemples concrets accompagnent les explications.

2. Composantes du modèle DATGRAPH

Le modèle DATGRAPH est composé de quatre graphes principaux qui permettent de conserver toute l'information relative au flux de contrôle à l'intérieur d'une routine et global au projet, au flux de données local aux routines et global au projet, ainsi qu'aux liens déclaratifs existant entre les identificateurs déclarés dans les routines du projet. Un cinquième graphe est utilisé temporairement pour la construction des différentes déclarations, permettant ainsi d'identifier les symboles rencontrés lors de l'analyse.

Le but de regrouper cette information est de pouvoir plus facilement analyser et visualiser certaines caractéristiques du code source et de rendre cette analyse et cette représentation indépendante du langage de programmation utilisé. Entre autre, à partir des informations de ce modèle, divers métriques de complexité peuvent être calculées et un certain niveau de documentation peut être extrait.

Avant de continuer l'élaboration du modèle, il convient de définir certains termes utilisés ainsi que certaines considérations conceptuelles générales.

2.1 Théorie des graphes

Le modèle présenté se sert principalement d'une représentation connue sous le nom de **graphe**. L'utilisation de cette représentation est bien justifiée par ce qu'a écrit R. Pellet dans *l'article liminaire* de son livre sur la théorie des graphes [PELL66]:

> *" Pour mieux comprendre ou mieux faire comprendre certaines réalités complexes, on est parfois amené à les simplifier et à les réduire à l'essentiel de ce qu'il convient de considérer en elles. Cet essentiel peut souvent se représenter par un schéma graphique où l'on distingue: d'une part des points qui traduisent l'existence d'éléments de base dans la situation étudiée, d'autre part, des lignes reliant ces points et qui expriment les relations existant entre les éléments de base. "* *Roger PELLET*

Il convient de définir certains termes tirés de la théorie des graphes. Bien qu'il ne s'agit que d'un résumé du vocabulaire descriptif de R. Pellet sur la théorie des graphes [PELL66], ces termes seront suffisants pour comprendre l'approche suggérée. Certains termes seront également présentés à titre informatif même s'ils ne seront pas nécessairement utilisés ultérieurement alors que d'autres seront quelque peu vulgarisés pour les besoins de ce travail.

Définitions des termes de la théorie des graphes[1]

Depuis le mathématicien hongrois Koening, on nomme graphe tout schéma pouvant s'analyser en un ensemble de points ou sommets, et un ensemble de lignes orientées ou non, reliant entre eux tout ou partie de ces sommets. De tels schémas peuvent être étudiés indépendemment de la signification particulière des sommets et des lignes qui les constituent. [PELL66]

Le terme **graphe,** couramment employé tout au long de ce document pour identifier les différentes formes servant à conserver les données, désigne une représentation de ce type.

1: Ces définitions sont tirées de [PELL66] avec une nomenclature adaptée aux besoins du présent travail.

Le tableau 1 permet de définir le symbolisme utilisé. Il est à remarquer que ces symboles sont empruntés à la théorie des ensembles à l'exception du symbole "<" qui se voit donner une autre signification que celle habituellement vue en algèbre élémentaire.

Symbole	Signification
Γ	Symbolise l'application qui définit le graphe
$\forall$ x	Quelque soit x ..., pour tout x ...,
$\exists$	Il existe ...
$x \in X$	L'élément x fait partie de l'ensemble X
$x \notin X$	L'élément x ne fait pas partie de l'ensemble X
$X \subset Y$	L'ensemble X est contenue dans l'ensemble Y
$Y \supset X$	L'ensemble Y contient l'ensemble X
$X \cup Y$	Réunion de l'ensemble X et de l'ensemble Y
$X \cap Y$	Intersection de l'ensemble X et de l'ensemble Y
$\emptyset$	Ensemble vide
$\lvert X \rvert$	Nombre d'éléments de l'ensemble X
$P_1 \Rightarrow P_2$	La proposition P_1 entraîne la proposition P_2
$x < y$	x est un ancêtre de y (définie plus loin)

Tableau 1 : Définition du symbolisme de la théorie des graphes

Considérons un ensemble X et un autre ensemble distinct Y. À tout élément de X on peut faire correspondre un ou plusieurs éléments de Y. On définit ainsi ce qu'on appelle une **application** de X dans Y.

Lorsque nous avons une application de X dans lui-même, la figure obtenue s'appelle un **graphe**. Un graphe G est donc défini par un ensemble X et une application Γ de X dans lui-même. On notera G par (X , Γ). La figure 2 montre un exemple de graphe.

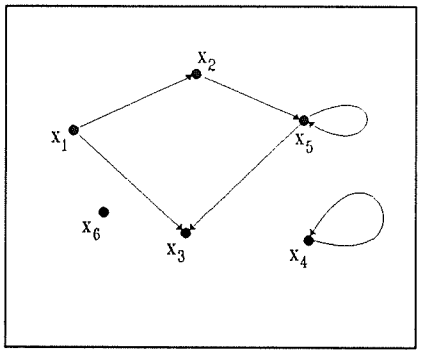


Figure 2 : Exemple général d'un graphe

Un **arc** (a, b) relie a à b si $b \in \Gamma a$. a est l'**extrémité initiale**, b est l'**extrémité terminale**. a et b sont dits **adjacents**. Deux arcs sont dits **arcs adjacents** lorsqu'ils ont une extrémité commune. Un arc ayant x pour extrémité terminale est dit **incident intérieurement** à x. Un arc ayant x pour extrémité initiale est dit **incident extérieurement** à x. Ces deux définitions s'appliquent également aux arêtes.

Une **arête** est un ensemble de deux sommets x et y tels que $x \in \Gamma y$ ou $y \in \Gamma x$. C'est en quelque sorte un arc non orienté.

Une **boucle** est un arc qui fait correspondre un sommet à lui-même.[2]

Un **chemin** est une séquence d'arcs tous parcourus dans le même sens. Un chemin est **simple** si tous ses arcs sont différents, sinon il est **composé**. Il est **fini** ou **infini** selon le nombre d'arcs qui le composent.

Un **circuit** est un chemin fini et fermé. L'extrémité terminale du dernier arc coïncide avec l'extrémité initiale du premier. C'est un cycle donc tous les arcs sont parcourus dans le même sens.

Une **chaîne** est une séquence d'arêtes dont une extrémité de chacune coïncide avec une extrémité d'une autre. Une chaîne est **simple** si toutes les arêtes sont différentes, sinon elle est **composée**.

Un **cycle** est une chaîne dont le sommet initial et le sommet terminal coïncident, et qui n'emprunte pas deux fois le même arc. Un **cycle élémentaire** n'emprunte qu'une fois le même sommet.

Dans un graphe fortement connexe, il existe au moins deux arcs incidents à chaque sommet. S'il y en a plus de deux, le sommet est dit **noeud**, sinon c'est un **anti-noeud**. Une **branche** est un chemin dont seules les extrémités sont des noeuds.

Le sommet x est un **ancêtre**, un **ascendant** du sommet y, ou encore, y est un **descendant** de x, s'il existe un chemin allant de x à y. On dit que x précède y ou que y suit x. Tel que vu précédemment on dénote cette relation par: $x < y$.

Dans un multigraphe (graphe pouvant avoir plusieurs arêtes reliant le même couple de sommets) le **degré** ou la **valence** d'un sommet est le nombre d'arêtes adjacentes à ce sommet.

Dans un graphe orienté, le **demi-degré intérieur** (**extérieur**) d'un sommet x est le nombre d'arcs incidents intérieurement (extérieurement) à ce sommet. On les note: $d^-(x)$ et $d^+(x)$.

Un sommet x d'un graphe est une **entrée** si son demi-degré intérieur est nul, c'est-à-dire si: $d^-(x) = 0$ avec $d^+(x) \neq 0$.

Un sommet est **isolé** s'il n'a aucune descendance et n'est la descendance d'aucun autre.

Un point d'un graphe est une **sortie** s'il n'est pas isolé et s'il n'a pas de descendant(s). On dit aussi qu'il est **terminal**.

Un sommet est **pendant** s'il n'a qu'une arête incidente.

On appelle **sommets irrelatés** deux sommets d'un même centre, mais qui ne sont pas descendants l'un de l'autre. Dans un arbre généalogique deux frères sont irrelatés.

2: Il ne faut pas confondre le terme *boucle* utilisé pour les graphes et celui utilisé dans les langages de programmations.

L'**ordre** d'un graphe est le nombre de ses sommets.

Un **sous-graphe** est un graphe issu de *G* par suppression de sommets et, bien entendu, des arcs adjacents à ces sommets. Un **graphe partiel** est un graphe issu de *G* par suppression d'arcs (les sommets restent même s'ils ne sont plus reliés).

On dit qu'un **graphe** est **planaire** lorsque sa représentation sur un plan, dite alors planaire topologique, ne comporte que des arêtes ne se rencontrant pas en dehors de leurs extrémités.

On appelle **graphe simple** le graphe défini par une application d'un ensemble *X* dans un autre ensemble *Y* strictement distinct de *X*.

Un **graphe** est **symétrique** si chaque couple de sommets reliés dans un sens l'est aussi dans l'autre.

Un **arbre** est un graphe connexe (formé d'une seule composante) sans cycle. Il est aisé de voir:

- qu'un arbre a autant d'arêtes que de sommets moins un;
- que si on supprime une arête quelconque, il cesse d'être connexe;
- et que si on relie deux sommets quelconques par une arête, on crée un cycle.

Une **arborescence** de racine *a* est un arbre dans lequel:

- aucun arc ne se termine en *a*;
- un arc et un seul se termine en tout sommet différent de *a*.

Les applications représentées par des graphes peuvent se représenter sous forme de tableau en convenant qu'on marquera, ou non, la croisée des axes issus des éléments des ensembles s'il y a correspondance, ou non, entre ces éléments. On peut remplacer ce tableau par une matrice carrée où le chiffre 1 marquera l'existence d'une correspondance, le chiffre 0 l'absence de correspondance. Cette matrice est appelée la **matrice associée** au graphe. La figure 3 donne un exemple de ce tableau et de cette matrice pour le graphe montré à la figure 2. Si on a affaire à un multigraphe le chiffre 1 sera remplacé par le nombre de liaisons correspondantes. Par exemple, une application de cette matrice est obtenue en élévant cette matrice à une puissance entière. Le coefficient p_{ij} de $[A]^{ß}$ représente alors le nombre de chemins distincts de longueur ß allant de x_i à x_j dans G.

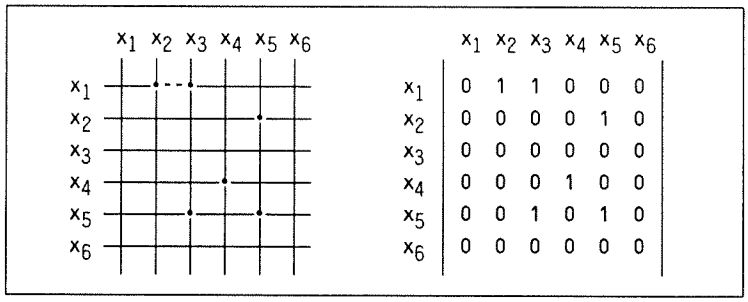


Figure 3 : Tableau et matrice associée d'un graphe

2.2 Définition des termes généraux

L'unité de base utilisée pour la dissociation des informations d'un projet est appelée une **unité fonctionnelle**. Elle correspond à la plus petite unité (d'un langage de programmation) ayant ses propres définitions et pouvant être appelée de façon indépendante. Il s'agit, en général, des fonctions, des procédures, des sous-routines et de toutes les autres formes de ce genre. Pour cette raison, le terme **routine** sera communément employé pour référer à l'unité de base.

Le terme **identificateur** est utilisé dans le même sens qu'il est défini dans la syntaxe des langages de programmation. C'est-à-dire qu'il représente un nom (mot) utilisé pour identifier un élément tel qu'une variable ou une fonction par exemple.

Le graphe temporairement utilisé pour la construction des différentes déclarations porte le nom de **graphe de construction**.

Les quatre graphes permettant de conserver et de représenter l'information portent les noms suivants:

- **graphe du flux de contrôle**, aussi appelé (en abrégé) graphe de contrôle;
- **graphe du flux de données**, aussi appelé (en abrégé) graphe de données;
- **graphe d'appel**;
- et **graphe des déclarations**.

2.3 Description de l'ensemble du modèle

Les sections qui vont suivre vont permettre de définir de façon plus exacte la construction et le contenu des différents graphes du modèle DATGRAPH. Pour l'instant, une brève description de l'utilité de chacun des graphes et de l'inter-relation entre ceux-ci, permettra de mieux en justifier la présence.

L'analyse du code source permet de créer les graphes précédemment mentionnés d'une façon bien définie. L'analyse s'effectue par fichier comme c'est le cas pour les compilateurs. Lors de cette analyse est généré un *graphe de construction* qui ne sert que temporairement[3] pour la construction des informations relatives aux déclarations et à la disponibilité des différents identificateurs rencontrés. Ce graphe est dynamique dans le sens où une partie de l'information qu'il contient peut être retirée au cours de l'analyse lorsqu'elle devient inutile.

Lorsqu'elle est analysée, chaque routine du fichier produit directement un graphe de contrôle. De plus, elle ajoute au graphe de construction certains liens qui vont permettre de compléter les autres graphes. Ces liens peuvent être vus comme représentant un second niveau d'information. À la fin de l'analyse de cette routine, un graphe de contrôle complet à été produit et, à partir du second niveau d'information du graphe de construction, un graphe du flux de données est généré pour cette routine. Les liens du second niveau d'information du graphe des déclarations servent également à construire de façon cumulative (à chaque routine, pour le projet) le graphe d'appel et le graphe des déclarations. La figure 4 permet de résumer ce cheminement.

3: L'information du graphe de construction n'est pas conservée après l'analyse du fichier.

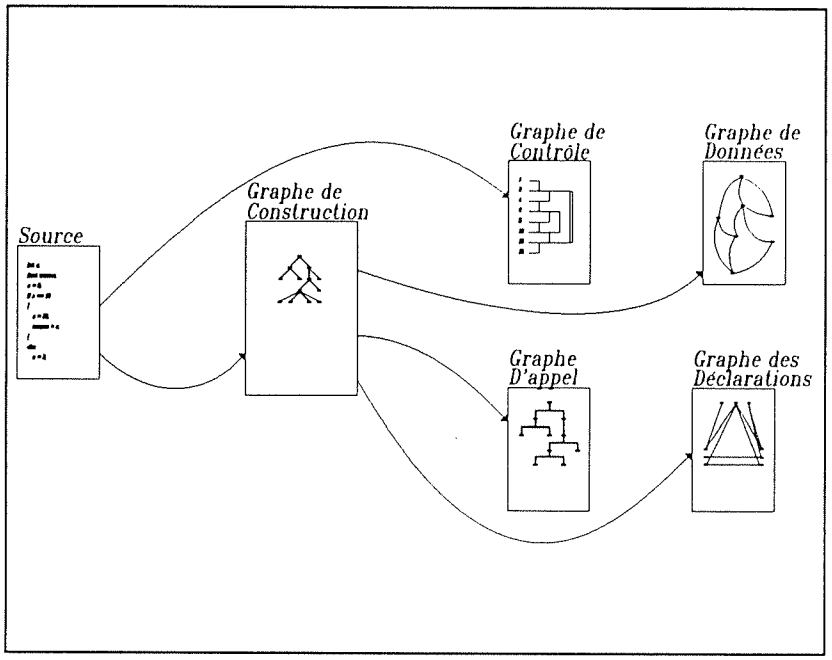


Figure 4 : L'analyse d'un source vis-à-vis le modèle DATGRAPH

Donc, chaque routine analysée possède un graphe du flux de contrôle et un graphe du flux de données. Le graphe des déclarations est associé au projet au même titre que le graphe d'appel. De ces graphes, seul le graphe du flux de contrôle est directement utilisé sous sa forme de graphe pour sa représentation. Les autres graphes ne servent qu'à la représentation interne dans le but de conserver et de traiter l'information plus efficacement.

Les caractéristiques du design et des structures de programmation employées seront principalement représentées par le graphe de contrôle et le graphe d'appel. Pour sa part, l'information relative à la définition et l'utilisation des données sera principalement contenue dans le graphe du flux de données et le graphe des déclarations. Aussi, cette vue peut se faire autant du point de vue des unités fonctionnelles que d'un point de vue global au projet. Le tableau 2 permet de mieux positionner les différents graphes utilisés. Bien que les utilisations de chacun des graphes semblent assez distinctes, il n'en demeurent pas moins très inter-reliés.

		Régions d'application	
		Routine	Projet
Caracté-ristiques	Structure (Design)	Flux de contrôle	Graphe d'appel
	Données	Flux de données	Déclarations

Tableau 2 : Positionnement des graphes du modèle DATGRAPH

Le **graphe de construction** permet de conserver le nom et le type des différents identificateurs rencontrés. Sa forme d'arbre permet aussi de conserver la portée de leur accessibilité. Ce niveau d'information est nécessaire afin de pouvoir identifier correctement les

différents identificateurs rencontrés lors de l'analyse. Le second niveau d'information est nécessaire pour la construction des autres graphes. Il est constitué de tous les liens existant entre les identificateurs. D'un point de vue général, les sommets de ce graphe seront associés aux différents identificateurs alors que les arcs démontreront, d'une part, les différents statuts déclaratifs et, d'autre part, le type des relations entre les identificateurs.

Le **graphe du flux de contrôle** représente les caractéristiques relatives au flux de contrôle d'une routine. Il est constitué de segments (représentant les arcs) et de points d'attache (représentant les sommets). L'utilisation de cette terminologie tient du fait que toute l'information relative au contenu du graphe de contrôle est placée dans les arcs, les sommets ne servant que de points de liaison pour ceux-ci. Ceci respecte le fait que le graphe représente par ses sommets les *éléments de base* (le contrôle) et par ses arcs les *relations* qui existent entre ces décisions de contrôle, ce qui satisfait la définition de R. Pellet précédemment citée. L'information conservée permet entre autre de montrer la nature du segment (son nom), la position de son contenu dans le code source, et certaines autres caractéristiques permettant de retrouver l'information pertinante du code source. Outre ses caractéristiques vis-à-vis du calcul des métriques et de la représentation de tous les cheminements de contrôle possible dans le source, ce graphe permet de faire le lien entre les éléments conservés dans les autres graphes et le code source.

Le **graphe du flux de données** contient les informations nécessaire pour retrouver l'information véhiculée à l'intérieur d'une routine. Cette information est principalement véhiculée par les variables et les fonctions (paramètres et valeurs de retour). Ce graphe conserve tous les liens entre les différents véhicules d'information utilisés dans une routine (y compris l'utilisation locale des variables globales). Les liens qui sont présents dans le graphe du flux de données sont caractérisés par l'utilisation ou l'affectation des composantes concernées. De plus, la position de ces relations est conservée de façon à permettre d'associer l'information relative au flux de données local en relation étroite avec le flux de contrôle. Tout comme pour le cas du graphe de construction, les sommets représentent les identificateurs[4] déclarés ou utilisés. Les arcs, par leur orientation, permettent d'indiquer quel identificateur a servi à affecter ou à modifier quel autre identificateur. Leur contenu permet de retracer l'endroit où ce flux de données s'est effectué.

Le **graphe d'appel** représente les liens entre les différentes unités fonctionnelles (routines) du projet. Les sommets sont la représentation des différentes unités. Par leur orientation, les arcs permettent d'identifier quelle unité appelle (donne le contrôle à) quelle autre unité. Leur contenu permet de conserver l'endroit des appels par rapport au graphe de contrôle. Ce graphe permet de conserver la structure de la hiérarchie des appels entre les routines. Cette structure peut tout aussi bien servir pour le calcul de métriques, la représentation arborescente (ou par référence) des appels, ou pour une aide à la documentation. Dans le cadre précis du développement du logiciel d'évaluation de code source, les sommets de ce graphe servent également de référence de base car chacun d'eux représente une unité de base tel que défini précédemment. Il s'agit donc de la composante principale de DATRIX du point de vue de l'organisation interne.

Le **graphe des déclarations** permet de conserver les identificateurs déclarés dans le projet. Il contient aussi bien les identificateurs locaux que ceux qui sont directement partagés par plus d'une routine (globaux). Les sommets représentent trois ensembles bien distincts. Le premier groupe est celui représentant les routines. Le second est celui représentant les variables. Alors que le troisième est celui représentant les différents types[5] qui ont été définis. Par leur présence, les arcs permettent d'indiquer quelle variable et quelle définition de type est disponible pour

---

4: En réalité, les sommets du graphe du flux de données sont un sous-ensemble de ceux du graphe de construction.

5: Le terme **type** est utilisé pour représenter les éléments définis comme étant de nouveaux types possibles. Par exemple, un nom de RECORD en Pascal ou un nom de structure en langage C.

quelle routine. Et elles permettent aussi d'indiquer le type des variables. La figure 5 montre les liens possibles dans ce graphe ainsi que leurs orientations. Selon les définitions vues précédemment, il peut être extrapolé qu'il s'agit d'un graphe simple à trois composantes[6]. Par leur contenu, les arcs permettent de savoir quelles routines ont déclaré, ou utilisé quelles variables ou quels types. Entre autre, cette structure permet de représenter les liens entre les données disponibles et possiblement utilisées dans les différentes routines, et donne une idée de l'utilisation des types.

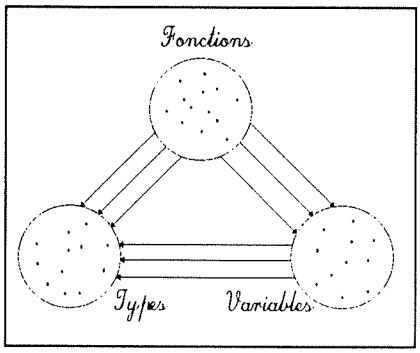


Figure 5 : Liens possibles dans le graphe des déclarations

Principales utilisations

L'information conservée permet donc d'effectuer des requêtes très diversifiées et ce, indépendemment du langage de programmation analysé.

Du point de vue du contrôle, il est possible d'obtenir des informations sur la complexité du code ou le design implanté par exemple. Ceci concerne autant le niveau interne de chacune des unités fonctionnelles que celui du projet.

L'étude des données peut permettre de déceller certaines informations par rapport à la cohésion interne des routines. Les inter-relations entre les routines peuvent aussi être facilement visualisées et quantifiées. La figure 6 montre les différentes relations pouvant exister entre deux routines. Il est possible de voir que toutes ces relations se retrouvent dans les graphes précédemment définis.

6: Il s'agit de l'application d'un ensemble X (les fonctions) dans un ensemble Y (les variables) strictement distinct de X, de l'application d'un ensemble X dans un ensemble Z (les types définis) strictement distinct de X, et de l'application d'un ensemble Y dans un ensemble Z strictement distinct de Y.

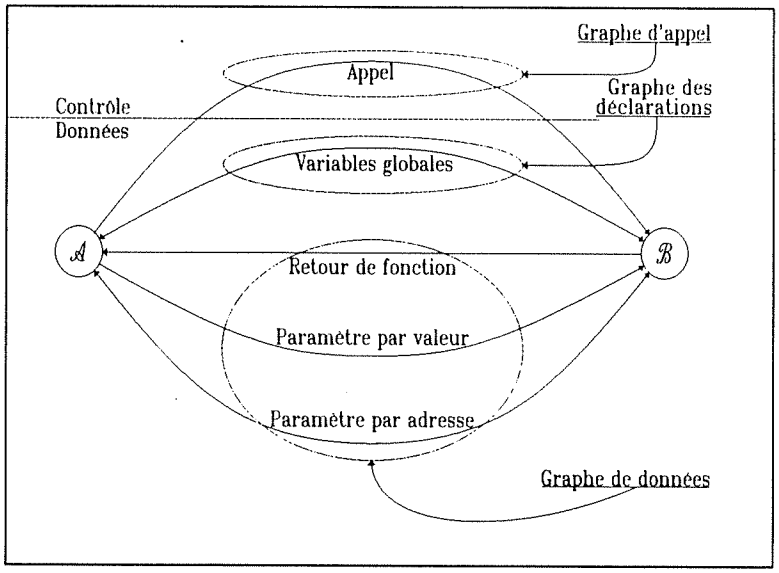


Figure 6 : Relations possibles lors de l'appel d'une sous-routine (A appelle B)

De plus, l'information recueillie est suffisamment complète pour permettre d'en tirer une documentation pertinente relativement au code source. Par exemple:

- les liens entre les routines peuvent permettre de mieux gérer les modifications apportées au projet;
- les identificateurs déclarés (variables, types) qui ne sont pas utilisés peuvent être identifiés;
- une liste des utilisations des éléments peut être construite, de cette façon:
  - les implications des modifications apportées à une définition d'un type de données sont facilement identifiables;
  - les effets d'une modification sur une variable disponible par plus d'une routine peuvent être visualisés et quantifiés.

Une étude plus approfondie des possibilités d'utilisation de l'information recueillie peut faire l'objet d'une recherche ultérieure. Pour l'instant, les sections suivantes vont définir de façon plus exacte la méthode à employer pour la construction de chacun des graphes. Les mentions référant à des possibilités d'utilisation de l'information n'auront d'autres implications que de permettre de s'assurer de conserver suffisamment d'information pertinente.

3. Graphe du flux de contrôle

L'analyse statique du code source repose encore sur les travaux initiaux de Halstead [HALS77] et de McCabe[MCCA76]. Les objectifs initiaux de cette analyse consistaient à déterminer les paramètres du code source de façon à obtenir certaines mesures. Ces métriques visent à mesurer la complexité, voire même la qualité, du code source.

Les métriques textuelles proviennent de la *science du logiciel*, qui est basée sur le décompte du nombre d'opérateurs et d'opérandes. Cette approche ne tient pas compte de la structure du programme. Malgré un bon acceuil de la part de la communauté scientifique, les métriques basées sur la science du logiciel sont l'objet de critiques de plus en plus nombreuses. Certaines attaques portaient directement sur le modèle psychologique de base [COUL83] [SHEN83]. La solidité de plusieurs validations empiriques a été mise en doute [HAME82] [LASS81].

Pour leur part, les métriques de graphe proviennent de la représentation du flux de contrôle. Cette représentation est principalement composée de deux éléments qui peuvent être appelés les *sommets* et les *arcs* par association à la théorie des graphes. Les sommets sont principalement des points de décisions alors que les arcs représentent les différentes alternatives originant de ces points.

La métrique de la complexité cyclomatique a reçu un haut degré d'acceptation de la part de la communauté scientifique. Cette dernière et ses dérivées ont fait l'objet de plusieurs recherches, expériences et implantations pour le calcul automatique. Cependant, les critiques s'élèvent autant du côté de la validité du fondement théorique que de la validité des études empiriques. Une de ces critiques porte sur le manque de distinction entre une décision qui a ou n'a pas de branche ELSE. Il y a aussi des doutes concernant la relation entre les difficultés d'effectuer des tests et la valeur de la métrique. Une proposition a été faite à l'effet que ceci pourrait être dû à la transformation ambiguë de McCabe entre le flux de contrôle du programme et le graphe généré [WHIT87].

Le problème fondamental reste que sans un modèle explicite, la validation empirique est sans signification [SHEP88].

Le graphe du flux de contrôle tel que présenté dans cette section suit un modèle formel pour la représentation du code source. Le modèle permet une représentation d'ensemble facilement visualisable du programme. Le programme est divisé en segments qui sont des regroupements d'instructions. L'organisation de ces segments permet d'obtenir le flux de contrôle du programme. L'avantage principal du modèle est qu'il permet une représentation complète et universelle du code source. Le représentation du graphe de ce modèle est indépendante du langage de programmation et s'applique à n'importe quel environnement. Son formalisme permet de définir de façon stricte des métriques et produit une représentation unique du code source.

3.1 Représentation graphique du graphe du flux de contrôle

Les techniques basées sur une représentation graphique du flux de contrôle sont très populaires. Par exemple, la schématisation du flux de contrôle est encore très utilisée dans la phase du design par l'utilisation de méthode comme l'organigramme (flowchart[7]) [BOHL71]

7: La méthode du 'flowchart' fût l'une des premières techniques de ce genre.

[FERS78] [SHNE77], les graphes N-S [LIND77] [SHAW82], le pseudocode schématique [ROBI85] [ROBI86] et plusieurs autres[8].

Les représentations graphiques sont très utiles parce qu'elles représentent beaucoup plus d'information qu'un simple chiffre. Le nombre d'éléments, leurs inter-relations, leurs organisations et plusieurs autres caractéristiques peuvent être visualisées et évaluées d'un simple coup d'oeil. Avec un peu d'expérience, il devient facile de détecter rapidement des problèmes potentiels.

Il est important de mentionner que les graphes ne sont pas strictement nécessaires pour calculer les métriques. N'importe quelle métrique basée sur les graphes peut être évaluée directement à partir du code source. La représentation graphique est uniquement requise pour la visualisation nécessaire à l'humain. Le graphe devrait donc être dessiné de façon à être immédiatement utilisable, et les informations pertinentes devraient être contenues pour respecter ce fait.

Le manque de standards, ou de méthodes formelles, pour générer une représentation graphique du code source est un problème. Par exemple, même si les symboles utilisés avec la technique de l'organigramme de IBM sont dans une certaine mesure standardisés, la façon de transcrire l'organigramme en code est laissée à l'imagination du programmeur. Malheureusement, les graphes de flux de contrôle existant ne sont pas rigoureusement définis. La valeur des métriques et leurs interprétations peuvent donc varier de façon significative selon la nature de l'algorithme utilisé pour obtenir le graphe du flux de contrôle à partir du code source.

En résumé, la représentation du graphe devrait posséder les caractéristiques qui suivent.

- Être dédiée à la visualisation humaine.
- Être une représentation exacte et formelle du code source. La transformation devrait être indépendante du langage de programmation utilisé.
- Être organisée de façon à fournir l'information de façon efficace.

Plusieurs représentations graphiques du graphe de flux de contrôle peuvent être retrouvées dans la littérature. La représentation de McCabe [MCCA76] fut l'une des premières. Cependant, par sa simplicité elle dépossède le graphe de certaines informations pertinentes à sa complexité comme par exemple l'élimination de croisements entre les cheminements de contrôle et l'élimination (par le fait même) des bris de structures. D'autres, parmi ses successeurs, ont introduit dans leurs représentations une différence entre les arcs séquentiels et les arcs de branchements [SCHN79]. La considération des chemins du graphe a été effectuée par Schneidewind [SCHN83] dans sa métrique de complexité.

Le graphe du flux de contrôle est construit de façon à mettre en évidence la physionomie du programme. Chaque segment (appelé *arc* dans la terminologie des graphes) a sa propre identification qui est basée sur son contenu. L'ordonnancement de ces segments, imposé par les points d'attache (appelés *sommets* dans la terminologie des graphes), fournit l'information relative aux possibilités du flux de contrôle. Le modèle de représentation du graphe utilisé subséquemment dans ce travail est une systématisation de celui utilisé dans [SCHN83] et [SCHN79].

8: Pour une discussion plus complète et une comparaison de ces méthodes, le lecteur est prié de consulter les références suivantes: [BERG81], [MART85], [RAMS83].

Un arc du graphe est un segment de programme qui contient un type spécifique d'instruction. Un arc montre la façon d'aller d'un sommet à un autre en exécutant un groupe d'instructions (possiblement vide). Les segments du graphe sont alignés verticalement, ce qui limite la représentation du flux de contrôle à deux directions. La direction vers le bas représente un flux d'exécution avant (même direction que lors de la lecture normale du code source) et est représentée par des traits verticaux simples (|). La direction vers le haut représente un flux d'exécution arrière (retour de boucle, par exemple) et est représentée par des traits verticaux doubles (‖). Les segments consécutifs d'un même point d'attache sont distingués par un déphasage horizontal (⊣). Tous les segments sont parallèles et originent de points d'attache numérotés et disposés sur l'axe vertical. La longueur d'un segment est indépendante de son contenu. Elle ne représente que la distance entre le début et la fin de la portée des énoncés l'ayant généré.

Les sommets du graphe sont associés aux points d'attache. Ils sont donc les points où s'attachent les segments. Ils sont représentés par des lignes horizontales. Le numéro d'un point d'attache correspond au numéro de la ligne (dans le code source) ayant généré le(s) segment(s) qui part(ent) de ce point. En général, ces numéros sont ordonnés en ordre croissant tout comme le sont les numéros des lignes du code source.

Certains points d'attache du graphe sont identifiés par des symboles spéciaux. Une flèche entrant dans le graphe (->) indique que l'exécution peut commencer à partir du segment partant de ce point, il s'agit d'un point d'entrée. Une flèche sortant du graphe (<-) indique que l'exécution se termine à l'intérieur du segment arrivant à ce point, il s'agit d'un point de sortie.

L'ordonnancement des segments selon l'axe horizontal dépend de la distance de leur point d'attache de destination. Le segment de gauche est celui ayant le point d'attache de destination le plus près. Les segments ayant le même point d'attache de destination sont disposés en ordre croissant de longueur en partant de la gauche.

Le graphe du flux de contrôle est dessiné dans un demi-plan. Seulement le côté droit de l'axe vertical est utilisé pour disposer les segments. Ceci est nécessaire afin de permettre d'identifier les croisements pouvant survenir entre les segments. Un croisement survient lorsqu'un segment (vertical) croise la ligne horizontale d'un point d'attache. Deux types de croisements peuvent être identifiés. Les premiers sont ceux permis par les règles de la programmation structurée. Ils sont simplement appelés *croisements* et sont représentés par deux lignes qui se croisent (╫, ┼). Le deuxième type est composé des croisements qui ne respectent pas les règles de la programmation structurée. Ils sont appelés *bris de structure* et sont représentés par des blocs (▪) à l'endroit du croisement.

3.2 Caractéristiques conceptuelles de la construction du graphe

Les caractéristiques principales que doit posséder le graphe final vis-à-vis la correspondance source-graphe sont à trois niveaux. Premièrement, la représentation des éléments doit être aussi claire que possible puisque le graphe est une représentation qui est utilisée principalement pour augmenter la compréhension de l'humain. Ce qui implique que les points d'attache non nécessaires ne doivent pas apparaître lors de la version finale du graphe. Deuxièmement, chaque segment du graphe ne doit contenir que des *énoncés de contenu* apparaissant séquentiellement dans le code source. Les énoncés de contenu sont ceux n'apportant aucune information aux différentes possibilités de flux de contrôle (les énoncés séquentiels et les expressions des booléens). Finalement, les portées des chemins arrières doivent être conservées afin de ne pas biaiser la correspondance source-graphe. Ce qui implique qu'il ne faut pas modifier leur point de départ ni leur point d'arrivée.

3.3 Formalisation de la construction du graphe du flux de contrôle

Les étapes de base pour la construction du graphe consistent principalement à générer un graphe initial sur lequel certaines règles de simplifications sont appliquées pour obtenir un graphe final. Ce graphe doit respecter les caractéristiques précédemment établies. Les sous-sections qui suivent expliquent chacune des étapes de la construction du graphe. Un exemple sera construit par étape afin de bien comprendre l'influence des règles.

3.3.1 Règles de traduction (Production du graphe initial)

La correspondance entre les énoncés des langages de programmation et les composantes du graphe du flux de contrôle est la base de la construction. Selon la façon dont le graphe est généré à partir du code source, les métriques reliées au graphe peuvent varier. Il faut donc une transformation rigoureuse entre le code source des différents langages de programmation et le graphe qui se doit d'être indépendant des langages.

À partir du code source d'un langage procédurel, un graphe de base est construit à l'aide de *segments primaires*. Les sommets du graphe représentent les points d'attache des segments. Le numéro de la ligne où l'information générant le segment est apparue dans le fichier source est associé au point d'attache de départ. Ce numéro ne sert que pour référence visuelle au source puisqu'il peut y avoir plus d'un point d'attache ayant le même numéro de ligne. Les segments primaires possibles sont les suivants:

Segments primaires:

- Segment séquentiel (nommé **S** pour *Sequential*). Produit par les énoncés qui ne participent pas au flux de contrôle. Les énoncés commentaires ne sont pas considérés comme étant des énoncés valides. Le segment possède un contenu qui représente les actions associées au code séquentiel.

- Segment étiquette (nommé **L** pour *Label*). Produit par un point de référence pour les autres segments du programme. Ce point de référence peut être réel (généré par une étiquette, un énoncé ELSE, etc.) ou trivial (comme pour le cas du cheminement séquentiel d'une conditionnelle).

- Segment de branchement avant (nommé **F** pour *Forward*) conditionnel ou inconditionnel. Produit par un énoncé de contrôle qui fait référence à une étiquette dans le sens du flux de contrôle. Le segment F branche toujours sur un segment L.

- Segment de branchement arrière (répétitif) (nommé **B** pour *Backward*) conditionnel ou inconditionnel. Produit par un énoncé de contrôle qui fait référence à une étiquette dans le sens inverse du flux de contrôle. Le segment B branche toujours sur un segment L.

- Segment conditionnel (nommé **C** pour *Conditional*). Produit par un énoncé de contrôle qui permet un choix de N directions (N > 1). N segments F (ou B) partent de ce segment. Un segment L est ajouté entre le segment F et le début du cheminement qui n'est pas référencé (partie vrai d'un IF-THEN-ELSE, par exemple) s'il existe. Le segment C possède le contenu associé à la condition (e.g. le booléen).

La représentation graphique permet donc de déterminer que les segments S, C, L et F sont représentés par des lignes verticales simples. Alors que les segments B sont représentés par des lignes verticales doubles.

Les règles de **traduction** sont les seules qui dépendent du langage de programmation. Ce sont elles qui donnent la transparence du graphe de contrôle vis-à-vis le langage. Ces règles permettent de traduire les différents langages en un ensemble de segments bien définis. Grâce aux différents types de segments énumérés précédemment, le processus de traduction devient relativement simple et peut suivre les grandes lignes suivantes:

- Chaque énoncé séquentiel (affectation, déclaration, etc.) produit un segment séquentiel (**S**).

- Chaque énoncé conditionnel (énoncé IF, condition de sortie de boucle, etc.), ayant N alternatives, produit un segment conditionnel (**C**) duquel partent N segments F (ou B) représentant les cheminements possibles. Un segment L est ajouté entre le segment F et le début du cheminement qui n'est pas référencé (partie vrai d'un IF-THEN-ELSE, par exemple) s'il existe.

- Chaque point de référence (étiquette, début de boucle, énoncé ELSE, énoncé END IF, etc.) produit un segment étiquette (**L**).

- Chaque saut avant inconditionnel produit un segment de branchement avant (**F**).

- Chaque saut arrière inconditionnel produit un segment de branchement arrière (**B**).

- Des segments S sont ajoutés pour certains éléments qui sont implicites à un énoncé. Par exemple, un énoncé du type FOR possède une initialisation et un incrément. Ces actions seront représentées par un segment S avant le segment C pour représenter l'initialisation, et par un segment S précédent le segment B de retour pour représenter l'incrément.

En général, l'extrémité terminale d'un segment est reliée au même point d'attache que l'extrémité initiale du segment auquel ce dernier branche. Deux cas se distinguent de ce fait. Le premier cas concerne le segment S généré par un énoncé permettant de commencer l'exécution d'une routine à ce point (énoncé d'entrée). Ce segment a comme extrémité initiale un point d'attache sur lequel ne peut partir et ne peut arriver aucun autre segment. Selon la terminologie des graphes il s'agit d'un *sommet d'entrée* puisqu'il a un demi-degré intérieur égal à zéro[9]. Il s'en suit que les segments qui se dirigaient sur le point d'attache de l'énoncé d'entrée seront reliés à l'extrémité terminale du segment le représentant. Le second cas concerne un segment S généré par un énoncé provoquant l'arrêt de l'exécution de la routine (énoncé de sortie). Ce segment a comme extrémité terminale un point d'attache sur lequel ne peut partir et ne peut arriver aucun autre segment. Selon la terminologie des graphes il s'agit d'un *sommet de sortie* puisqu'il n'a aucune descendance[10]. La figure 7 permet de représenter ces deux cas.

9: Il faut faire attention. Pour qu'un sommet du graphe du flux de contrôle représente réellement un point d'entrée du point de vue du contrôle, il faut qu'il satisfasse deux conditions. Il doit évidemment être un sommet d'entrée tel que défini par la théorie des graphes et, de plus, il doit posséder l'attribut de la flèche entrant (->) qui démontre l'accès possible de ce sommet par l'extérieur. Si la seconde condition n'est pas respectée, il s'agit alors d'un sommet dont l'arc (ou les arcs) incident extérieurement ne peuvent jamais faire parti d'un chemin (plus concrètement il s'agit de code mort).

10: Comme pour le cas du sommet d'entrée deux conditions doivent être respectées pour qu'un sommet du graphe du flux de contrôle représente réellement un point de sortie du point de vue du contrôle. Il doit évidemment être un sommet de sortie tel que défini par la théorie des graphes et, de plus, il doit posséder l'attribut de la flèche sortant (<-). Cependant, ces deux conditions seront toujours présentes simultanément puisqu'aucun énoncé peut *ne pas transférer* le flux de contrôle et du même coup *ne pas l'arrêter*.

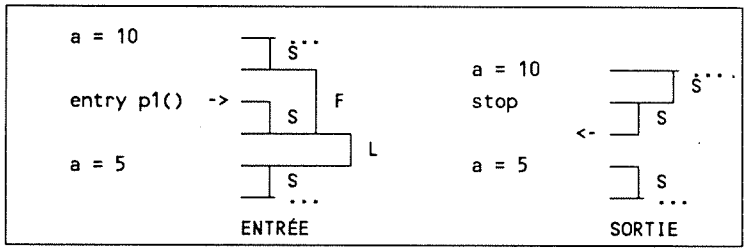


Figure 7 : Représentation du graphe de contrôle pour les entrées et les sorties

Suite à cette traduction, le code source d'une routine peut être représenté sous forme schématique. Il sera constitué d'un ensemble des segments primaires définis précédemment. Le graphe obtenu après cette étape est appelé le *graphe initial*.

Par exemple, le code source présenté à la figure 8 permet de générer le graphe du flux de contrôle correspondant.

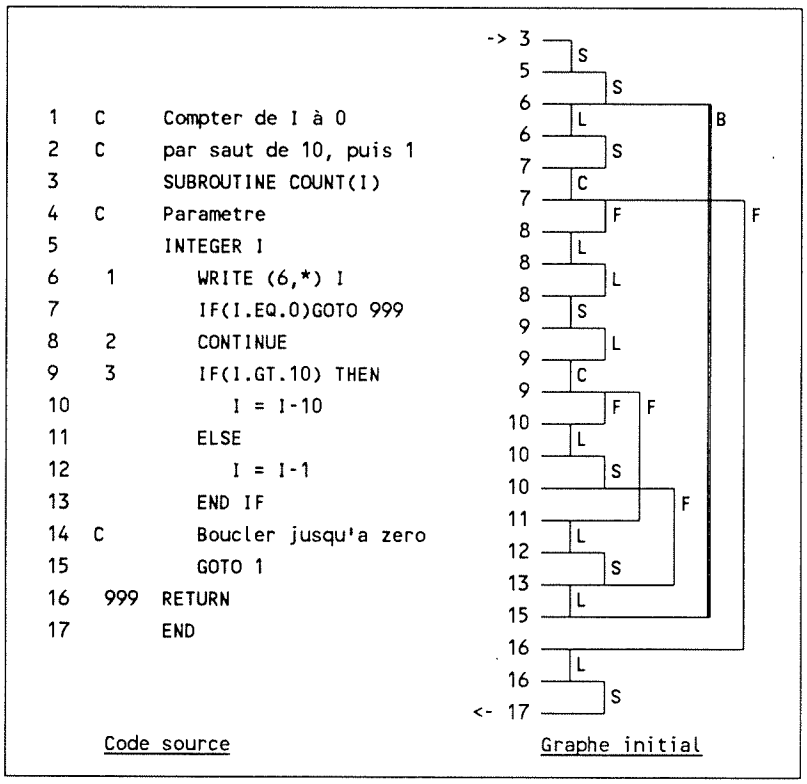


Figure 8 : Code source et graphe de contrôle initial correspondant

3.3.2 Règles de réduction

La première étape de simplification du graphe consiste à réduire les segments qui n'apportent aucune information au flux de contrôle. Les règles régissant la **réduction** permettent de réduire en un seul segment les segments primaires **S** qui se suivent et les segments primaires **L**

qui ne sont pas explicitement référencés par d'autres segments (L ou B). Ceci permet d'alléger la représentation car les segments réduits peuvent être vus comme étant des blocs fermés représentés par un seul segment. Formellement, les règles sont celles présentées à la figure 9.

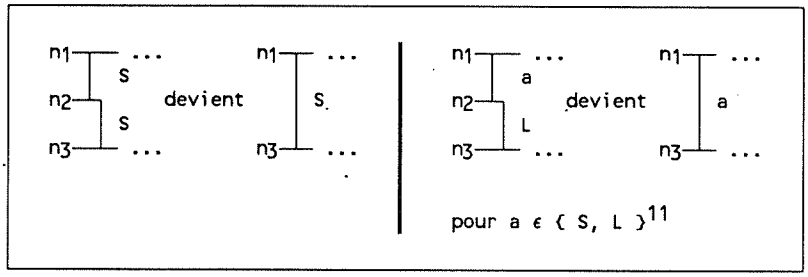


Figure 9 : Énoncé des règles de réduction

L'expansion des règles de réduction donne les possibilités présentées au tableau 3.

▪ S + S -> S	▪ S + L -> S[12]	▪ L + L -> L[13]

Tableau 3 : Énumération des règles de réduction

Le graphe obtenu à la fin de l'application de ces règles de réduction est appelé le *graphe réduit*. Sa caractéristique par rapport au graphe initial est qu'il est allégé de certains points d'attache qui ont été éliminés par le regroupement de segments. Il n'y a donc aucune information disparue suite à cette transformation.

L'application de ces règles à l'exemple de la section précédente donne le graphe de contrôle présenté à la figure 10.

11: Il est à remarquer que le segment *a* ne peut être un des éléments F ou B car ils sont des références explicites (branchement) au segment S. De plus, le segment *a* ne peut être l'élément C car ce dernier implique plusieurs arcs partant de n_2, et le segment C n'est suivi que de segments F ou B.

12: Cas d'une étiquette inutile.

13: Cas lorsqu'un artefact de construction a généré plusieurs étiquettes successives.

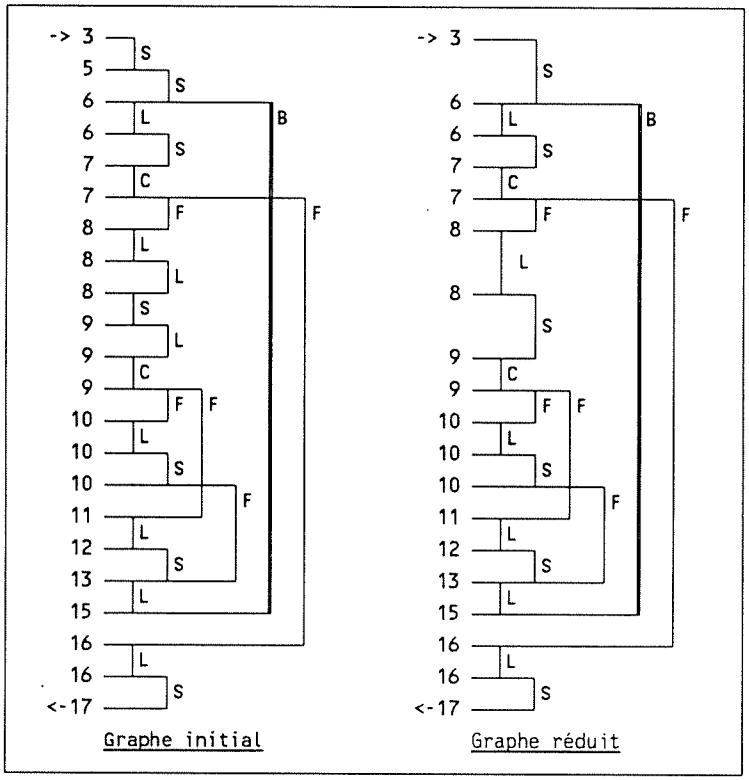


Figure 10 : Exemple d'utilisation des règles de réduction

3.3.3 Règles de concaténation

La deuxième étape de simplification du graphe consiste à concaténer les segments primaires afin d'obtenir des segments secondaires. La **concaténation** est effectuée afin d'éliminer certains points d'attache qui n'apportent pas d'information relativement au flux de contrôle. Ces points d'attache ont été générés dans le graphe initial simplement parce que chaque élément était vu indépendemment du contexte (des segments l'entourant). Cette étape a donc comme but de concaténer les segments avec leurs voisins immédiats si possible. La concaténation s'effectue selon les caractéristiques de concaténation associées à chaque élément. Ces caractéristiques représentent la faculté qu'a un élément à pouvoir être pré-concaténé ou post-concaténé. Cette possibilité repose sur le fait que le segment concerné ne doit pas impliquer de modification au flux de contrôle (saut, séparation, branchement possible, etc.) et doit conserver les portées des chemins arrière (segments B). En représentant ces possibilités par une étoile (*) avant et après l'élément, les caractéristiques peuvent se représenter de la façon présentée au tableau 4.

*C	: pré-concaténable (non *post-* car une séparation du flux suit ce segment)
*F	: pré-concaténable (non *post-* car ce segment produit un saut)
S	: pré et post-concaténable
L*	: post-concaténable (non *pré-* car un branchement est possible à l'avant)
B	: non concaténable (non *pré-* et non *post-* car il faut conserver la portée)

Tableau 4 : Caractéristiques de concaténation des segments primaires

L'application de la concaténation suit les règles définies par les caractéristiques. Deux segments consécutifs qui sont reliés par leur étoile respective sont concaténables. Ces règles doivent être appliquées selon l'ordre chronologique des points d'attache du graphe. Alors le résultat de la concaténation est unique. La raison de cet ordre chronologique ne repose que sur le fait de la simplification du nombre de règles de concaténation possible. Par exemple, les permutations du genre LS+F et L+SF sont éliminées. Plus formellement, ceci mène aux règles de concaténation présentées à la figure 11.

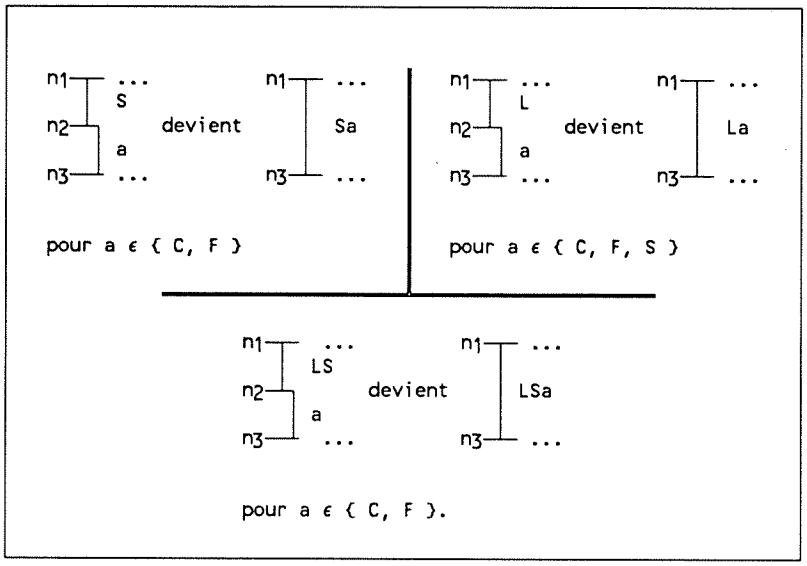


Figure 11 : Énoncé des règles de concaténation

L'expansion des règles donne les possibilités présentées au tableau 5.

■ S + C -> SC	■ L + C -> LC	■ LS + C -> LSC
■ S + F -> SF	■ L + F -> LF	■ LS + F -> LSF
	■ L + S -> LS	

Tableau 5 : Énumération des règles de concaténation

L'application des règles de concaténation donne de nouveaux segments. Les *segments secondaires* possibles sont les suivants: *SC, *SF, LS*, LC, LF, LSC et LSF.

Il est possible de voir que la concaténation est complète car les seuls cas de concaténation possibles restants (LS* avec *SC, par exemple) ne peuvent se produire puisque l'application des règles de réduction a précédemment éliminé ces cas.

Le graphe obtenu à la fin de cette étape est appelé le *graphe concaténé*. Il est simplifié du point de vue de la représentation par rapport au précédent (graphe réduit). Ceci est apporté par l'ajout des segments secondaires qui permettent de relier l'information qui est forcément séquentiel dans le code source et qui n'influence pas le cheminement du flux de contrôle.

L'application de ces règles à l'exemple de la section précédente donne le graphe de contrôle présenté à la figure 12.

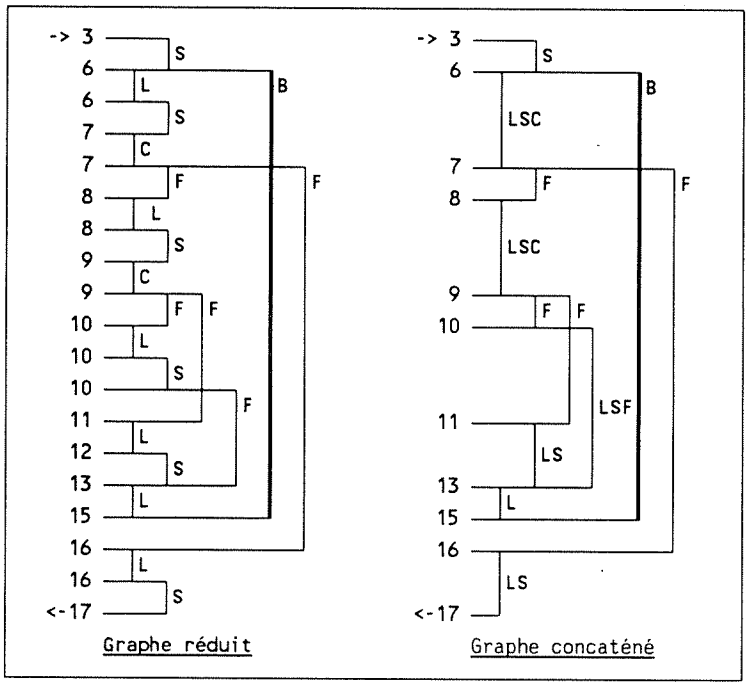


Figure 12 : Exemple d'utilisation des règles de concaténation

3.3.4 Règles d'association

La troisième étape de simplification du graphe consiste à associer les segments primaires et secondaires résultants des étapes précédentes. Les règles d'**association** permettent de simplifier la représentation du graphe en éliminant les références uniques tout en respectant le principe que deux séquences de code générant du contenu (segments S et C) doivent apparaître successivement dans le code source pour être dans le même segment. Sinon, les deux segments doivent rester distincts. Le respect de cette contrainte est facilité par le fait que tous les rassemblements possibles ont été effectués lors des étapes de réduction et de concaténation[14]. Donc, lors de l'application des règles d'association il ne reste plus qu'à associer les segments qui ne provoquent pas de rassemblement de code. Pour ce faire, il suffit d'éviter d'associer deux segments ayant chacun une partie de contenu[15]. Pour les sauts arrière il faut conserver la portée de la boucle. Le seul cas d'association possible pour ce type de saut est le cas d'un retour de boucle effectué en deux étapes ou plus.

Ces règles peuvent être appliquées simultanément, sans préséance. La seule priorité conservée est le parcours du graphe en partant du haut (sommet correspondant à la première ligne de code) vers le bas de façon à conserver les points d'attache de façon cohérente.

Formellement, les règles d'association respectant les critères sont celles présentées à la figure 13.

14: Le seul cas non associé pour un code consécutif dans le source est celui d'un saut sur la ligne suivante. Ce fait est normal compte tenu du fait que la séquence de code est coupée par un saut.

15: Les segments ayant du contenu sont les segments ayant une partie *S* ou une partie *C*.

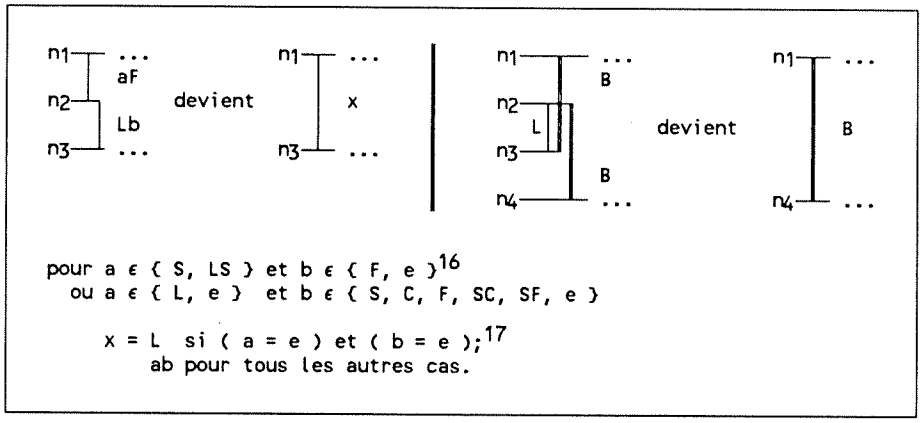


Figure 13 : Énoncé des règles d'association

L'expansion des règles donne les possibilités présentées au tableau 6.

■ SF + LF -> SF	■ LF + LS -> LS	■ F + LS -> S
■ SF + L -> S	■ LF + LC -> LC	■ F + LC -> C
■ LSF + LF -> LSF	■ LF + LF -> LF	■ F + LF -> F
■ LSF + L -> LS	■ LF + LSC -> LSC	■ F + LSC -> SC
	■ LF + LSF -> LSF	■ F + LSF -> SF
	■ LF + L -> L	■ F + L -> L
Et;		
	■ B + L + B -> B	

Tableau 6 : Énumération des règles d'association

Après l'application de ces règles, le **graphe final** est obtenu. L'application des règles étant formelle, le graphe généré a une représentation unique. Ce graphe permet une représentation simplifiée du code source et chacun des arcs le composant ne contient que des énoncés d'action (pas de contrôle) séquentiels dans le code source. Les énoncés participant au contrôle ne sont pas dans le contenu car ils participent à la création des segments.

L'application de ces règles à l'exemple de la section précédente donne le graphe du flux de contrôle présenté à la figure 14.

16: Le symbole *e* représente une chaîne vide. Autrement dit, *aF* avec *a* = *e* donne *F*.
17: Ce cas ne survient que lorsque le segment *L* est suivi immédiatement d'un segment *B*.

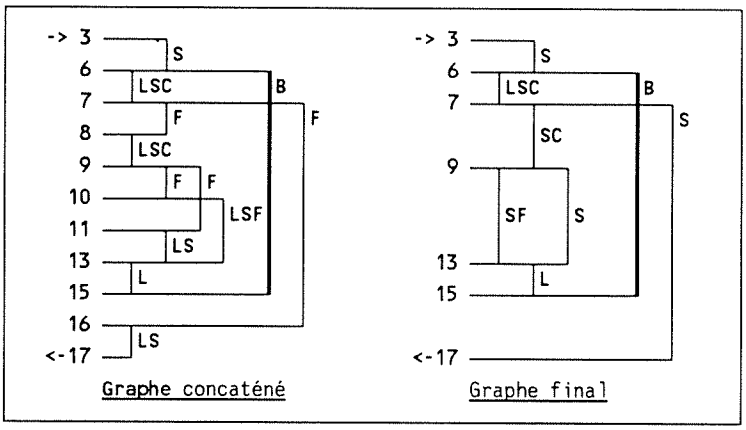
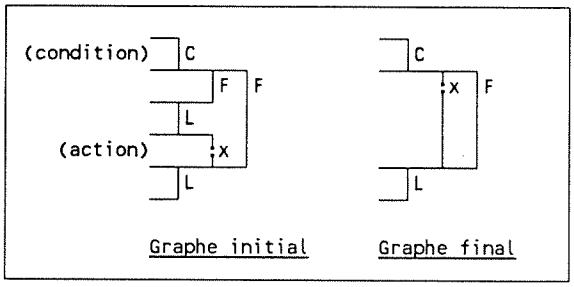


Figure 14 : Exemple d'utilisation des règles d'association

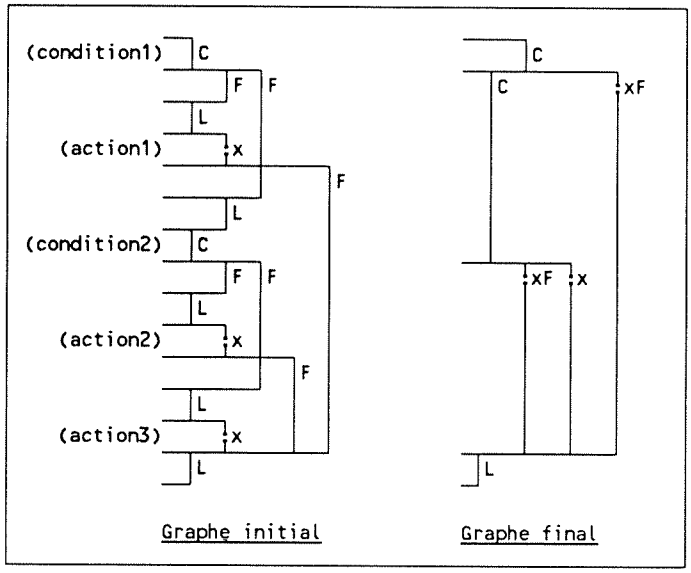
3.4 Exemples de construction de graphe pour les structures de base

L'application des règles de construction du graphe de contrôle à l'égard des structures de base du PASCAL, du langage C et du FORTRAN donne les résultats présentés dans les sections qui suivent. Les graphes initiaux et finaux sont présentés avec certains segments ayant le symbole *x* dans leur nom. Ce symbole représente qu'un sous-graphe peut être contenu à cette endroit. Ce sous-graphe étant inconnu (infinité de possibilités), une certaine interprétation du graphe final doit être faite. De plus, une certaine limitation d'interprétation doit être faite pour la représentation des croisements (en particulier pour le langage C).

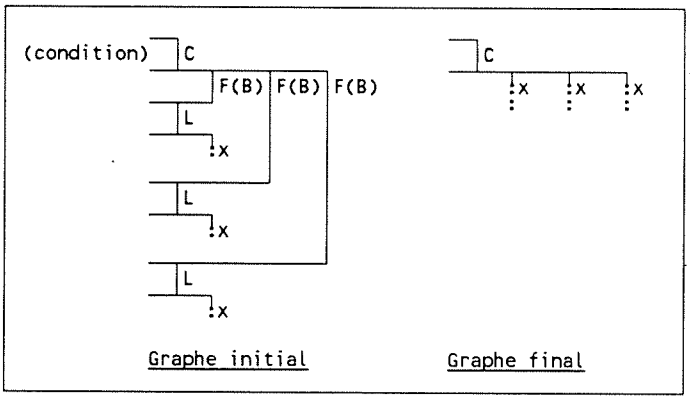
3.4.1 Structures de base du Fortran



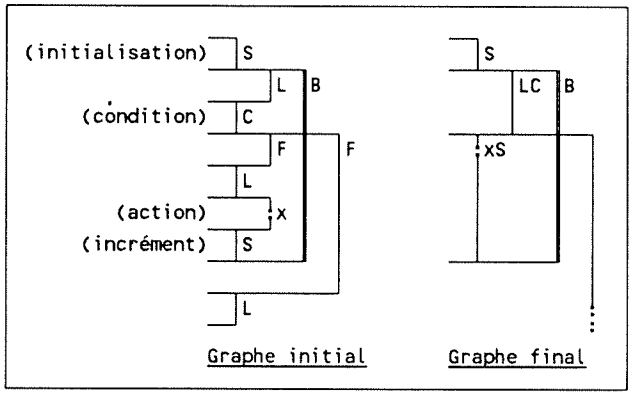
IF (condition) THEN (action)



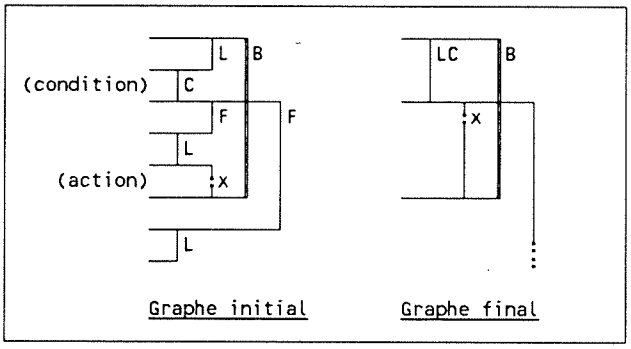
IF (condition1) (action1)
ELSEIF (condition2) (action2)
ELSE (action3)



IF (expression) GOTO (N directions)

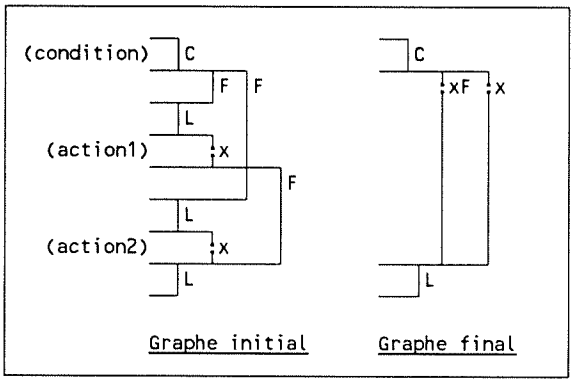


DO (initialisation) (condition) (incrément) (action)

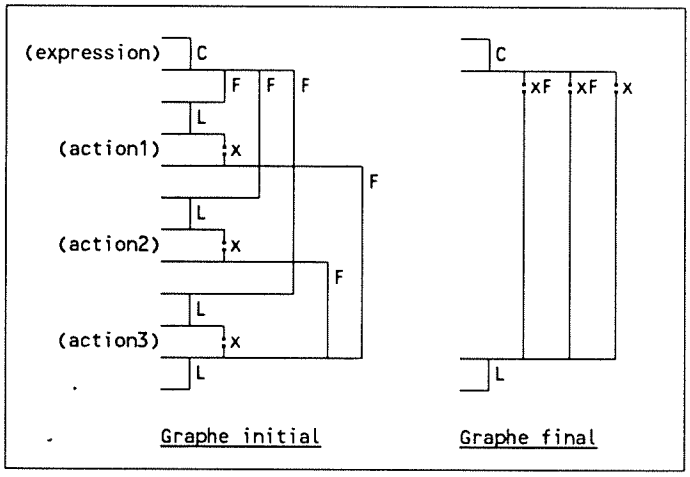


DO WHILE (condition) (action)

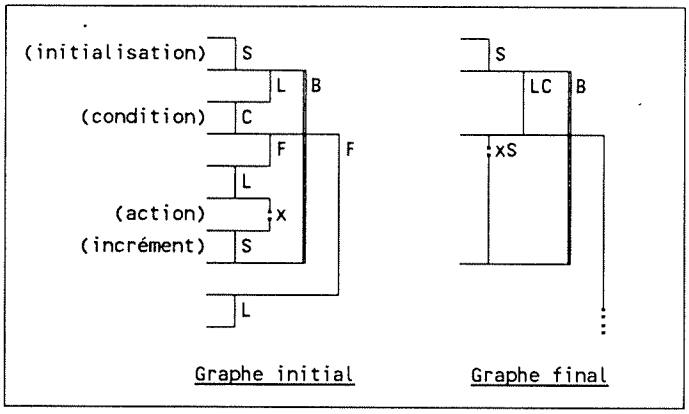
3.4.2 Structures de base du Pascal



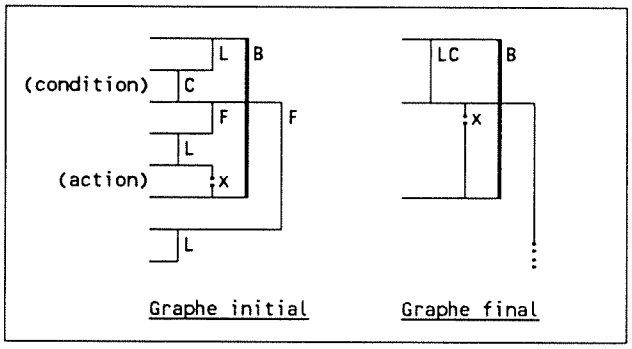
IF (condition) THEN (action1) ELSE (action2)



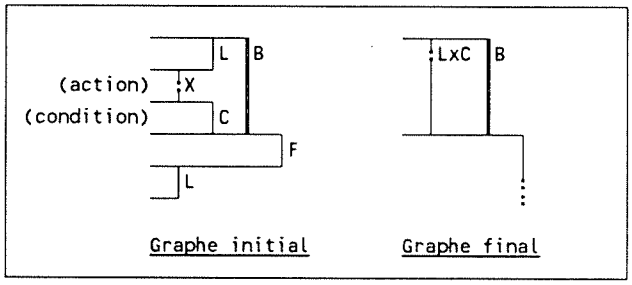
CASE (expression) OF
(cas1) : (action1)
(cas2) : (action2)
(cas3) : (action3)



FOR (initialisation) TO (condition) DO (action)

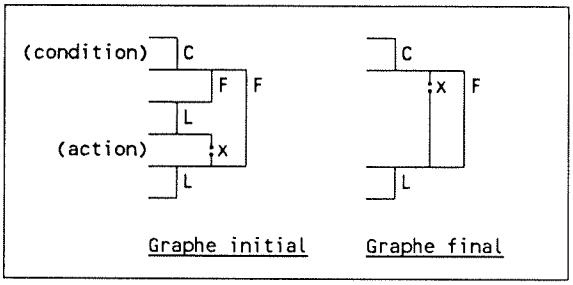


WHILE (condition) DO (action)

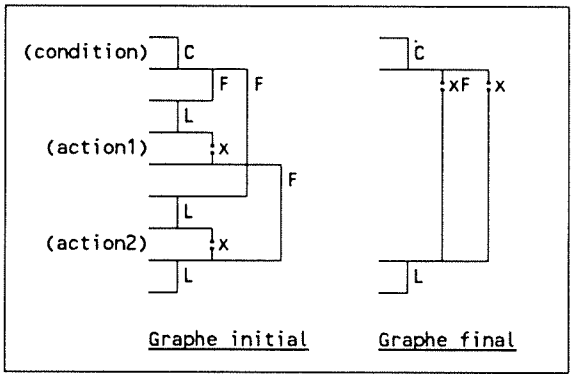


REPEAT (action) UNTIL (condition)

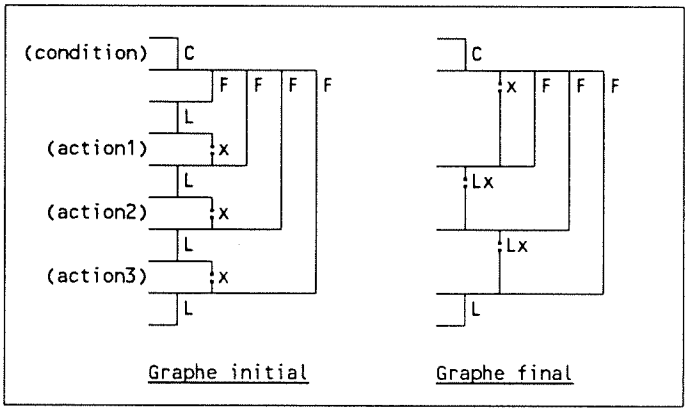
3.4.3 Structures de base du langage C



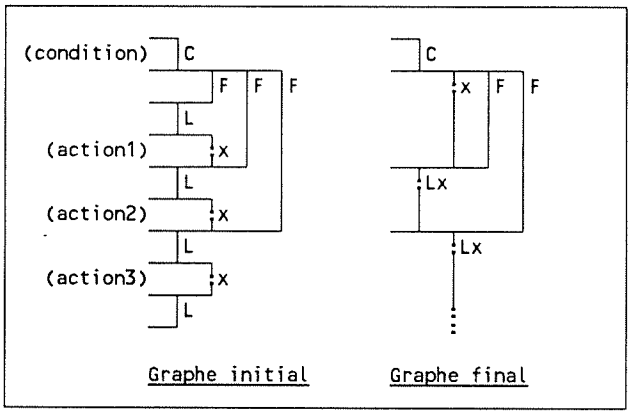
IF (condition) {action}



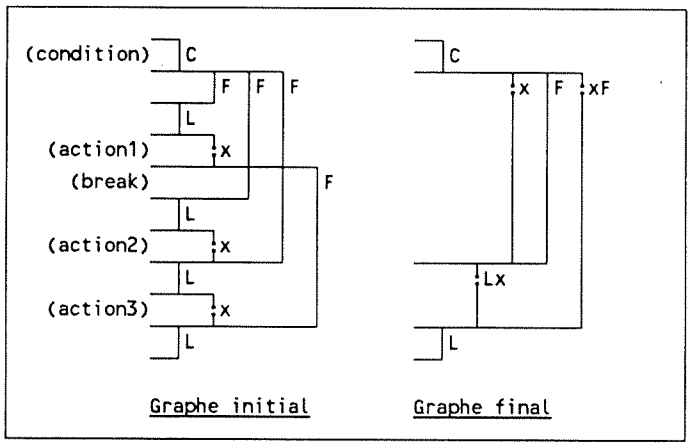
IF (condition) {action1} ELSE {action2}



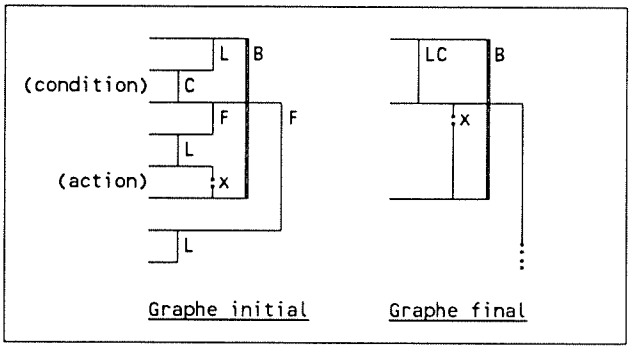
SWITCH (condition) { case1: action1; case2: action2; case3: action3; }



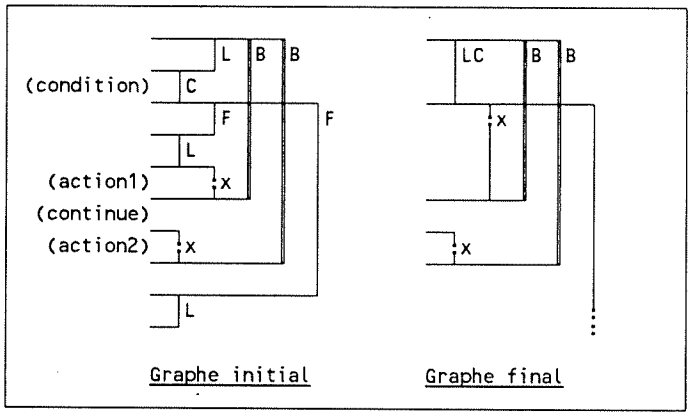
SWITCH (condition) { case1: action1; case2: action2; default: action3; }



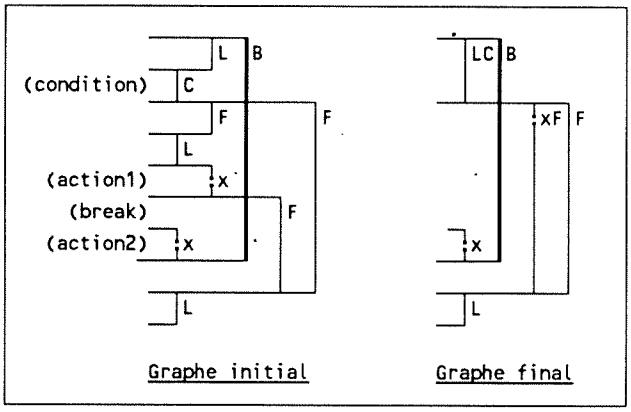
SWITCH (condition) { case1: action1; break; case2: action2; default: action3; }



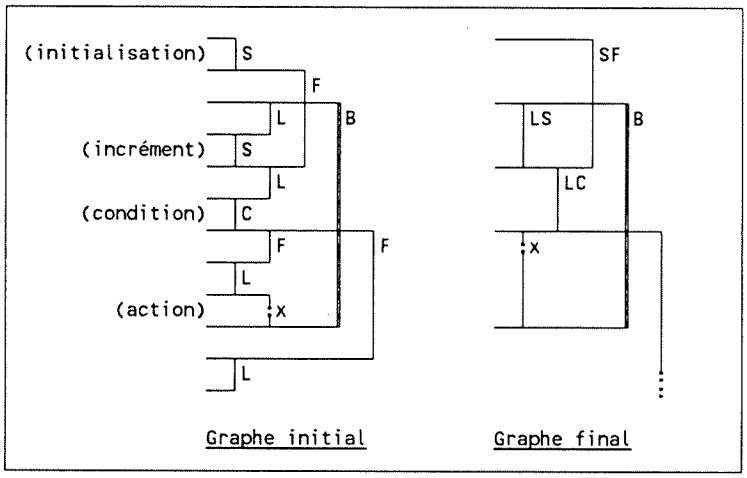
WHILE (condition) {action}



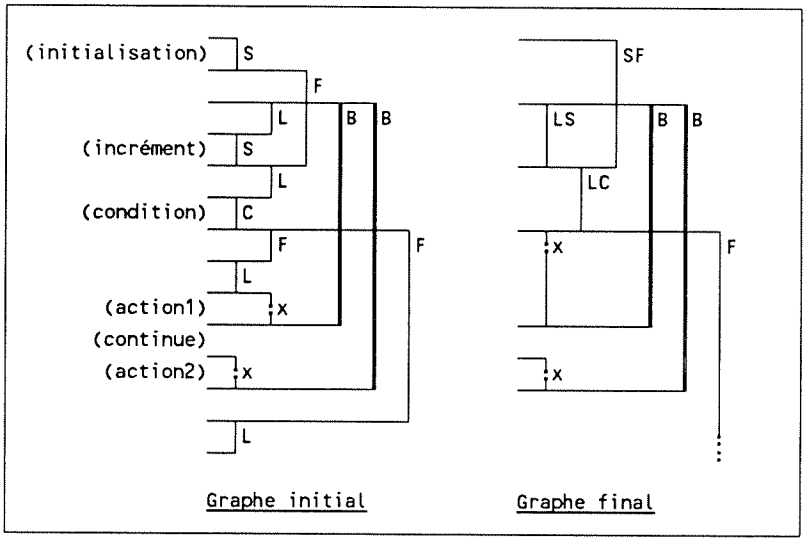
WHILE (condition) { action1; continue; action2 }



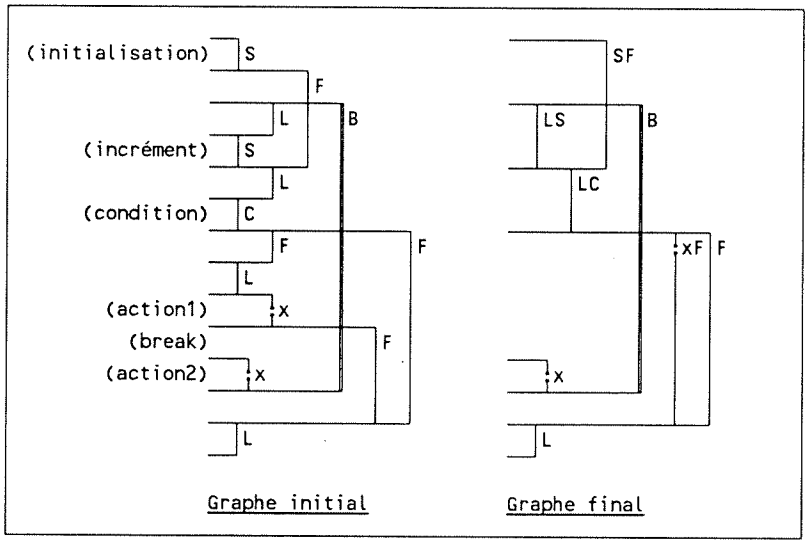
WHILE (condition) { action1; break; action2 }



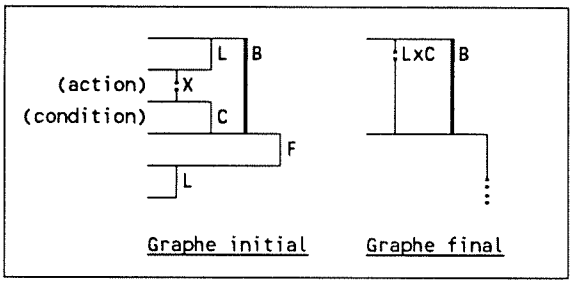
FOR (initialisation, condition, incrément) {action}



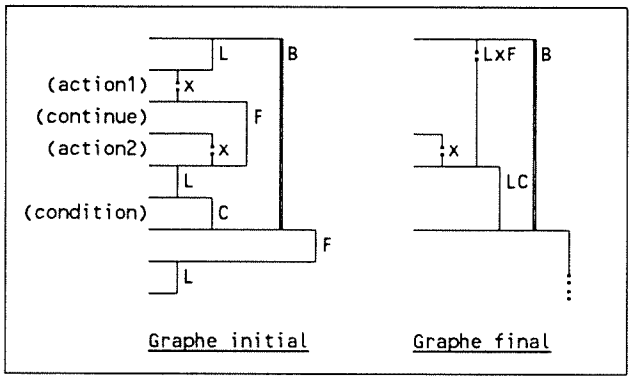
FOR (initialisation, condition, incrément) { action1; continue; action2; }



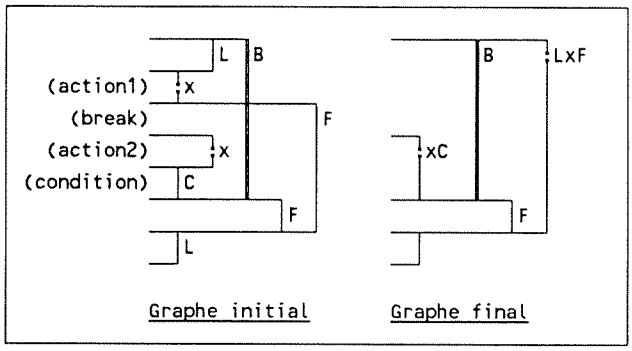
FOR (initialisation, condition, incrément) { action1; break; action2; }



DO {action} WHILE (condition)



DO { action1; continue; action2 } WHILE (condition)



DO { action1; break; action2 } WHILE (condition)

3.5 Définition du contenu d'un point d'attache

Les points d'attache servent à relier les segments entre-eux. Le seul élément qui soit contenu dans un point d'attache du graphe de flux de contrôle est le numéro de la ligne du code

source où ont été générés les segments de ce point. L'application des règles n'influence pas ces numéros. Les points d'attache qui restent conservent leur numéro original[18].

3.6 Définition du contenu d'un segment

Pour une implantation de la construction du graphe du flux de contrôle il est important de mentionner quelle est l'information qui est contenue dans les segments. Cette information doit permettre de pouvoir retracer toute l'information utile pour reconstruire les différentes caractéristiques des langages procéduraux. De façon générale, le contenu du segment doit posséder l'information suivante:

- son **nom**, afin de pouvoir identifier le type d'opération qui peut être effectué sur ce segment.

- la **position** (dans le source) du code contenu dans ce segment et du code formant ce segment. Chaque segment ne doit contenir que du code (celui associé au contenu) successif dans le code source. Par contre, le code formant le cheminement peut être distinct. Donc, la position du début et la position de la fin vont permettre de retracer tout le code du contenu qui est inclu dans le segment, alors qu'une liste sera nécessaire pour conserver les positions du code de contrôle.

- le **nombre d'énoncés** du type séquentiel contenu dans ce segment. Chaque segment peut (en langage C par exemple) avoir une section de déclaration et une section de structure (code réellement exécutable). De plus, deux types d'énoncés sont distingués. Les énoncés commentaires et les autres énoncés séquentiels. Donc, ceci implique qu'il faut cumuler quatre informations pour chaque segment. C'est-à-dire le nombre d'énoncés de chaque type pour chaque section.

- la **conditionnelle**. Cette information doit permettre de pouvoir retracer le type de l'énoncé conditionnel (IF expression, IF arithmétique, etc.), le texte de son booléen[19] et les éléments nécessaires pour calculer sa complexité. Ces éléments sont regroupés en deux principaux groupes (les opérateurs et les opérandes) et servent au calcul des métriques. De plus, les segments qui originent d'un segment ayant une composante conditionnelle (C) doivent avoir un numéro indiquant la séquence d'apparition de ceux-ci selon l'ordre défini par le type de l'énoncé conditionnel. Ceci est nécessaire afin de pouvoir retracer la bonne alternative selon l'évaluation du booléen. Lorsque le numéro de séquence ne s'applique pas directement (le segment précédent n'a pas de composante C) il est mis à 1.

- l'identificateur qui sert d'**étiquette**. Le nom de l'étiquette sera conservé pour référence au source. Entre autre, les noms des étiquettes non référencées pourront être placés dans une liste. Les étiquettes générées automatiquement (ELSE, END IF, etc.) ont NULL comme nom associé et ne peuvent pas se retrouver dans la liste.

- les diverses **caractéristiques** qu'un segment peut posséder. Un indicateur doit permettre de savoir s'il y a un énoncé d'entrée dans le segment. De même, un indicateur doit permettre de savoir s'il y a un énoncé de sortie dans le segment. Des indicateurs devront montrer si certaines composantes des segments sont commentées. Les composantes possiblement commentées sont les parties correspondant à un segment primaire du type C, F, L ou B.

18: Par exemple, voir les indicateurs dans les définitions des règles présentées dans les sections précédentes.

19: Le texte du booléen est conservé pour une reconstruction éventuelle. L'utilisation de ce texte ne se fait que sous la forme d'une manipulation de la chaîne complète puisque l'étude des parties de cette chaîne est dépendante du langage de programmation.

Voici la formalisation de la liste des éléments que doivent contenir les différents segments du graphe de contrôle selon les besoins énumérés précédemment:

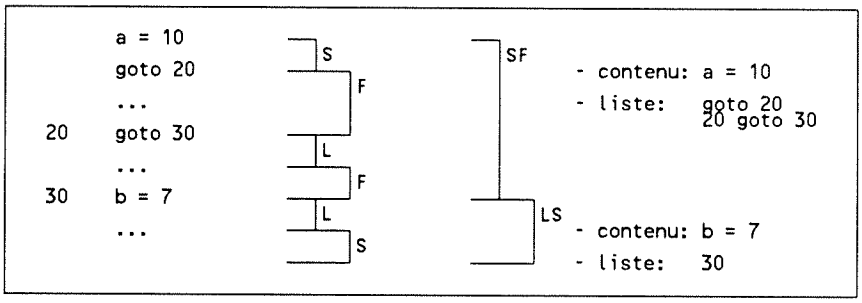
Nom:

1) Nom du segment.

Position:

2) Adresse de départ du contenu du segment. Cette adresse est constituée du nom du fichier source, du numéro de la ligne correspondante du fichier source et d'un index. L'index est le nombre de caractère lu dans le fichier (en tenant compte des fichiers inclus).
3) Adresse de fin du contenu du segment (Nom du fichier, numéro de ligne et index).
4) Pointeur à une liste des positions du code de contrôle (tout ce qui n'est pas du contenu et qui a généré le segment) et/ou du code des éléments qui sont simplifiés lors des règles de réduction et d'association.

   Par exemple:



Nombre d'énoncés:

5) Nombre d'énoncés contenus dans la section déclaration du segment.
6) Nombre d'énoncés commentaires contenus dans la section déclaration du segment.
7) Nombre d'énoncés contenus dans la section structure du segment.
8) Nombre d'énoncés commentaires contenus dans la section structure du segment.

Conditionnelle:

9) Expression booléenne complète dans le langage de programmation, le type de l'énoncé de contrôle et le nombre de chacun des éléments présenté au tableau 7. Cette information ne s'adresse qu'aux segments qui ont un membre C.

Types	Éléments	Exemples
Opérateurs	logique comparaison arithmétique	ou, et <, >, >= +,-,*,/,(),affectation
Opérandes	constante variable évaluation	constante, 10, "allo" variable appel à fonction

Tableau 7 : Regroupement des éléments composant une expression booléenne

10) Séquence des segments qui suivent le segment C. Cette séquence est définie pour chaque ensemble de segments qui suivent un segment C.

Étiquette:

11) L'identificateur qui sert d'étiquette. Valable seulement pour les segments ayant une composante L.

Caractéristiques du segment:

12) Attribut du segment: (les indicateurs de l'attribut sont indépendants)

- N_i si un énoncé d'entrée est présent, NN_i sinon.
- N_e si un énoncé de sortie est présent, NN_e sinon.
- C_c si la partie C du segment est commentée, NC_c sinon.
- C_l si la partie L du segment est commentée, NC_l sinon.
- C_f si la partie F du segment est commentée, NC_f sinon.
- C_b si la partie B du segment est commentée, NC_b sinon.

3.6.1 Définition du contenu des segments primaires

Selon les règles de construction du graphe du flux de contrôle, il y a 12 segments différents qui peuvent être rencontrés dans le graphe final. Certains de ces segments ne possèdent pas toute l'information décrite précédemment. Le contenu des segments secondaires sera établi lors des règles de concaténation. La liste des éléments que contiennent chacun des cinq segments primaires est présentée au tableau 8.

Segment \ Numéro de l'élément	1	2	3	4	5	6	7	8	9	10	11	12
S	√	√	√	*[20]	√	√	√	√		√		√
C	√	√	√	*					√	√		√
L	√			√						√	√	√
B	√			√						√		√
F	√			√						√		√

Tableau 8 : Contenu des segments primaires

3.6.2 Définition des opérations utilisées sur le contenu

Les prochaines sections montrent les éléments d'information résultant de l'application des règles de simplification du graphe. Les tableaux permettent de montrer l'opération à effectuer à partir des éléments des segments concernés pour obtenir les éléments du segment résultant.

20: Le symbole * indique que le segment primaire ne contient pas directement l'information mais que de l'information peut se retrouver dans un segment S ou un segment C après l'application des règles de réduction et d'association pour conserver l'information des segments simplifiés.

Certaines opérations sont nécessaires pour obtenir le contenu du segment résultant de l'application d'une des règles. La définition des opérateurs utilisés dans les sections qui suivent est présentée dans le tableau 9.

+ : Additionner les éléments,

| : Effectuer l'opération OU binaire (bit à bit),

« : Affecter le bit associé au segment à la valeur de la constante.
ex: β « NC_l affecter le bit associé au segment L
à la valeur de NC_l dans le segment β.

^ : Concaténer les deux listes d'éléments.

Tableau 9 : Définition des opérateurs sur le contenu des segments

Par exemple (fictif), le tableau 10 montre que lorsque les segments S et L sont réduits:

- les segments à réduire sont Sα et Lβ,
- l'élément 1 prend la valeur du nom du nouveau segment, le nouveau segment est un segment S,
- l'élément 2 (position de début du contenu) prend la valeur de l'élément 2 du segment α,
- l'élément 3 (position de la fin du contenu) prend la valeur de l'élément 3 du segment β,
- l'élément 4 pointe sur une liste qui contient une concaténation des listes de Sα et Lβ,
- l'élément 5 (nombre d'énoncés de la section déclaration)prend la valeur de la somme des éléments 5 des segments α et β,
- ...
- l'élément 9 (expression booléenne) n'est pas utilisé,
- ...
- l'élément 12 (attribut) prend la valeur de l'opération suivante: l'indicateur relié à Lβ est affecté par la valeur NC_l puis l'opération OU binaire est effectuée avec l'élément 12 de Sα.

Segments à réduire (exemple) — Numéro de l'élément

	1	2	3	4	5	6	7	8	9	10	11	12
S_α et L_β	S	α	β	α^β	$\alpha+\beta$	$\alpha+\beta$	$\alpha+\beta$	$\alpha+\beta$		α		$\alpha \mid (\beta \ll NC_l)$

Résultat de la réduction (exemple)

Tableau 10 : Exemple de l'utilisation des opérateurs sur le contenu

3.6.3 Définition des éléments conservés selon les règles de réduction

Cette section montre les éléments d'information résultant de l'application des règles de réduction de deux segments. Le tableau 11 permet de montrer l'opération à effectuer à partir des éléments des deux segments à réduire pour obtenir les éléments du segment résultant.

Segments à réduire[21] — Numéro de l'élément

	1	2	3	4	5	6	7	8	9	10	11	12
S_α et S_β	S	α	β	$\alpha\hat{}\beta$	$\alpha+\beta$	$\alpha+\beta$	$\alpha+\beta$	$\alpha+\beta$		α		$\alpha+\beta$
S_α et L_β*	S	α	α	$\alpha\hat{}\beta$	α	α	α	α		α		$\alpha \mid (\beta \ll NC())$
L_α et L_β*	L			$\alpha\hat{}\beta$						α	α	$\alpha \mid (\beta \ll NC())$

Résultat de la réduction

Tableau 11 : Contenu des segments après l'application des règles de réduction

3.6.4 Définition des éléments conservés selon les règles de concaténation

Cette section montre les éléments d'information résultant de l'application des règles de concaténation de deux segments. Le tableau 12 permet de montrer l'opération à effectuer à partir des éléments des deux segments à concaténer pour obtenir les éléments du segment résultant. Les opérations utilisées ont été définies précédemment.

Segments à concaténer — Numéro de l'élément

	1	2	3	4	5	6	7	8	9	10	11	12
S_α et C_β	SC	α	β	$\alpha\hat{}\beta$	α	α	α	α	β	α		$\alpha \mid \beta$
S_α et F_β	SF	α	α	$\alpha\hat{}\beta$	α	α	α	α		α		$\alpha \mid \beta$
L_α et C_β	LC	β	β	$\alpha\hat{}\beta$					β	α	α	$\alpha \mid \beta$
L_α et F_β	LF			$\alpha\hat{}\beta$						α	α	$\alpha \mid \beta$
L_α et S_β	LS	β	β	$\alpha\hat{}\beta$	β	β	β	β		α	α	$\alpha \mid \beta$
LS_α et C_β	LSC	α	β	$\alpha\hat{}\beta$	α	α	α	α	β	α	α	$\alpha \mid \beta$
LS_α et F_β	LSF	α	α	$\alpha\hat{}\beta$	α	α	α	α		α	α	$\alpha \mid \beta$

Résultat de la concaténation

Tableau 12 : Contenu des segments après l'application des règles de concaténation

21: Les étiquettes des segments L_β identifiés par un astérisque (*) sont non référencées. Avant d'être éliminées elles peuvent être mises dans une liste avec leur attribut qui indique si elles sont commentées. Le code associé à ces étiquettes appartient au nouveau segment généré.

3.6.5 Définition des éléments conservés selon les règles d'association

Cette section montre les éléments d'information résultant de l'application des règles d'association de deux segments. Le tableau 13 permet de montrer l'opération à effectuer à partir des éléments des deux segments à associer pour obtenir les éléments du segment résultant. Les opérations utilisées sont définies dans une des sections précédentes.

Segments à associer[22] — Numéro de l'élément

	1	2	3	4	5	6	7	8	9	10	11	12
SF_α et L^*F_β	SF	α	α	$\alpha\hat{}\beta$	α	α	α	α		α		$(\alpha\ll NC_f)\mid(\beta\ll NC_l)$
SF_α et $L^*\beta$	S	α	α	$\alpha\hat{}\beta$	α	α	α	α		α		$(\alpha\ll NC_f)\mid(\beta\ll NC_l)$
LSF_α et L^*F_β	LSF	α	α	$\alpha\hat{}\beta$	α	α	α	α		α	α	$(\alpha\ll NC_f)\mid(\beta\ll NC_l)$
LSF_α et $L^*\beta$	LS	α	α	$\alpha\hat{}\beta$	α	α	α	α		α	α	$(\alpha\ll NC_f)\mid(\beta\ll NC_l)$
LF_α et L^*S_β	LS	β	β	$\alpha\hat{}\beta$	β	β	β	β		α	α	$(\alpha\ll NC_f)\mid(\beta\ll NC_l)$
LF_α et L^*C_β	LC	β	β	$\alpha\hat{}\beta$					β	α	α	$(\alpha\ll NC_f)\mid(\beta\ll NC_l)$
LF_α et L^*F_β	LF			$\alpha\hat{}\beta$						α	α	$(\alpha\ll NC_f)\mid(\beta\ll NC_l)$
LF_α et L^*SC_β	LSC	β	β	$\alpha\hat{}\beta$	β	β	β	β	β	α	α	$(\alpha\ll NC_f)\mid(\beta\ll NC_l)$
LF_α et L^*SF_β	LSF	β	β	$\alpha\hat{}\beta$	β	β	β	β		α	α	$(\alpha\ll NC_f)\mid(\beta\ll NC_l)$
LF_α et $L^*\beta$	L			$\alpha\hat{}\beta$						α	α	$(\alpha\ll NC_f)\mid(\beta\ll NC_l)$
F_α et L^*S_β	S	β	β	$\alpha\hat{}\beta$	β	β	β	β		α		$(\alpha\ll NC_f)\mid(\beta\ll NC_l)$
F_α et L^*C_β	C	β	β	$\alpha\hat{}\beta$					β	α		$(\alpha\ll NC_f)\mid(\beta\ll NC_l)$
F_α et L^*F_β	F			$\alpha\hat{}\beta$						α		$(\alpha\ll NC_f)\mid(\beta\ll NC_l)$
F_α et L^*SC_β	SC	β	β	$\alpha\hat{}\beta$	β	β	β	β	β	α		$(\alpha\ll NC_f)\mid(\beta\ll NC_l)$
F_α et L^*SF_β	SF	β	β	$\alpha\hat{}\beta$	β	β	β	β		α		$(\alpha\ll NC_f)\mid(\beta\ll NC_l)$
F_α et L_β	L			$\alpha\hat{}\beta$						α	β	$(\alpha\ll NC_f)\mid\beta$
$B_\alpha$et L_β*et B_σ	B			$\alpha\hat{}\beta\hat{}\sigma$						α		$\alpha\mid(\beta\ll NC_l)\mid(\sigma\ll NC_b)$

Résultat de l'association

Tableau 13 : Contenu des segments après l'application des règles d'association

22: Les segments L identifiés par un astérisque (*) peuvent être accumulés pour obtenir la liste de ceux simplifiés. Certains d'entre eux sont non nécessaires.

4. Graphe de construction

De par sa nature, le graphe de contrôle était relativement indépendant des différents éléments déclaratifs du langage analysé. Seul l'identification des énoncés de contrôle était nécessaire pour assurer une construction exacte du graphe. La nature des identificateurs n'était pas nécessaire dans la mesure où ces éléments pouvaient être identifiés. Cependant, les informations conservées dans les autres structures d'information nécessitent plus de rigueur dans l'identification précise des types et des caractéristiques des identificateurs.

Au même titre que le compilateur, l'analyseur de code source doit pouvoir tenir trace des différentes déclarations et des portées de validité des identificateurs déclarés. L'utilité de cette information n'est nécessaire qu'au point de vue conceptuel afin de pouvoir construire de façon exacte les différents liens permettant de conserver l'information.

Le graphe de construction doit contenir un premier niveau d'information bien précis. Il doit conserver la définition des genres et des appartenances (l'accessibilité) des différents identificateurs. Cette information est nécessaire afin de pouvoir déterminer quel identificateur est déclaré par qui et pour obtenir le lieu de validité de ces identificateurs. Pour certains langages le nom des identificateurs doit être unique, ce qui ne pose pas d'ambiguïté concernant leur utilisation. Cependant, d'autres langages (le langage C, par exemple) permettent de définir des identificateurs identiques qui ne se distinguent que par la portée de leur validité[23]. Il apparaît donc nécessaire de conserver les différentes relations de déclaration (validité) à l'intérieur même d'un graphe qui servira d'ossature pour la ceuillette des informations des autres représentations (graphe du flux de données, graphe d'appel et graphe des déclarations).

C'est directement sur ce graphe que sera disposée l'information complémentaire nécessaire à la construction des autres représentations de données. Après l'analyse de chacune des routines, les informations relatives aux autres représentations seront placées dans leurs graphes respectifs. Puis, pour la suite de l'analyse du fichier, l'information relative aux déclarations sera conservée pour tenir compte des informations nécessaire à l'analyse de la routine suivante mais les informations inutiles seront retirées. L'information du graphe de construction est donc dynamique dans le sens où elle n'est présente que pendant son temps d'utilité.

L'information sera transférée aux autres graphes de façon précise après l'analyse complète d'une routine. Les différents cheminements possibles par lesquels les données locales sont véhiculées seront représentés par le graphe du flux de données. Les appels entre les unités fonctionnelles (routines) seront représentés par le graphe d'appel. Alors que les liens de déclarations, de disponibilités, et d'utilisation relatives aux types définis et aux régions de données communes à plusieurs unités fonctionnelles seront représentés par le graphe des déclarations.

Le graphe de construction est donc associé à une unité de compilation (habituellement un fichier) et ne sera nécessaire que pour l'analyse. Il n'est pas conservé après que l'information nécessaire ait été conservée dans les autres représentations. Il existera donc un graphe temporaire par fichier analysé.

23: Habituellement, la dernière déclaration associée à un identificateur est celle qui devient valide pour toute la durée de sa déclaration.

4.1 Construction du graphe

Tel que mentionné, le graphe de construction contient deux niveaux d'information. Le premier niveau concerne les déclarations des identificateurs alors que le second concerne l'utilisation de ces identificateurs.

La représentation de l'information relative aux déclarations se projette très bien sous la forme de graphe. Plus précisément, la structure principale s'apparente à celle d'un arbre. Le sommet de tête représente le fichier analysé alors que tous les niveaux inférieurs représentent les éléments déclarés. La construction de l'arbre contenant l'information relative aux déclarations peut se faire en associant les sommets aux différents identificateurs pouvant être définis. Dans ce cas, chacun des sommets représente un identificateur qui est soit un nouveau type défini[24], soit une fonction, soit une variable, soit une constante définie.

Méthodologie

Au sommet de tête (représentant le fichier) sont reliés les sommets des éléments déclarés de façon globale à l'extérieur de toutes les unités fonctionnelles du fichier. Par conséquent, la déclaration du début d'une unité fonctionnelle provoque un lien entre le sommet principal et le sommet de cette unité fonctionnelle. Le sommet de l'unité fonctionnelle devient le sommet courant tant que la fin de l'unité fonctionnelle n'a pas été rencontrée ou que le début d'une autre unité interne n'est pas détectée. À ce dernier sommet seront reliés les sommets de tous les identificateurs déclarés directement à l'intérieur de sa portée. Ce processus peut se répéter à plusieurs niveaux. Chaque niveau représentant une nouvelle portée pour la validité des identificateurs.[25]

D'une façon générale,

- À chacun des sommets représentant les unités fonctionnelles sont reliés les sommets des identificateurs (variables, fonctions, etc.) déclarés par cette unité. Le sommet de l'unité fonctionnelle sera du genre **routine**. Les arcs reliant les sommets pères aux sommets fils contiennent l'information indiquant une relation de **déclaration de paramètre** (**P**) pour ceux qui relient les identificateurs déclarés comme paramètres, et indiquent une relation de **déclaration locale** (**L**) pour les autres. En plus de contenir le nom de l'identificateur et le nom de son type, les sommets des éléments déclarés contiennent des indicateurs donnant le genre, la classe du type et un attribut décrivant le mode de déclaration (pointeur, vecteur, etc.) du type de l'identificateur.

- À chacun des sommets représentant une nouvelle déclaration de type correspond un arbre local (possiblement vide si le nouveau type n'est pas un type composé) représentant les différents membres de ce type. Le sommet représentant le type est du genre **type**. Les arcs reliant les sommets pères aux sommets fils contiennent l'information indiquant une relation de **déclaration locale** (**L**). Les sommets des éléments déclarés contiennent les mêmes indicateurs que ceux décrit précédemment. Le genre des descendants sera toujours nommé **variable**.

- Chacun des sommets représentant une variable n'implique aucune sous-arborescence, sauf dans le cas d'une variable représentant un type composé, alors l'arborescence du type

---

24: De nouveaux types peuvent être défini de plusieurs façon. Il peut s'agir d'un nom correspondant à un type déjà existant (TYPEDEF en langage C). Il peut s'agir d'un regroupement d'objets de différents types [KERN85] (une structure en langage C, un Record en Pascal). Les types ayant un regroupement seront ultérieurement appelés *types composés*.

25: Les niveaux peuvent apparaître de plusieurs façons selon le langage analysé. Par exemple, le Pascal reconnaît ses niveaux par la déclaration de procédures internes. Pour sa part, le langage C les reconnaît par des déclarations effectuées à l'intérieur d'une paire d'accolade ({}).

composé est copiée sous ce sommet. Les raisons et les implications de cet acte seront vues plus loin.

Par exemple, le code source présenté à la figure 15 permet d'obtenir le graphe de construction présenté à la figure 16.

```
struct str1 {
              int   mem1;
              float mem2;
              int   mem3;
            };
int  glo1;

void routine1( i )

   int  i;

   {
      int  j;
      .
      .
      .
   }
```

Figure 15 : Code source pour le graphe de construction

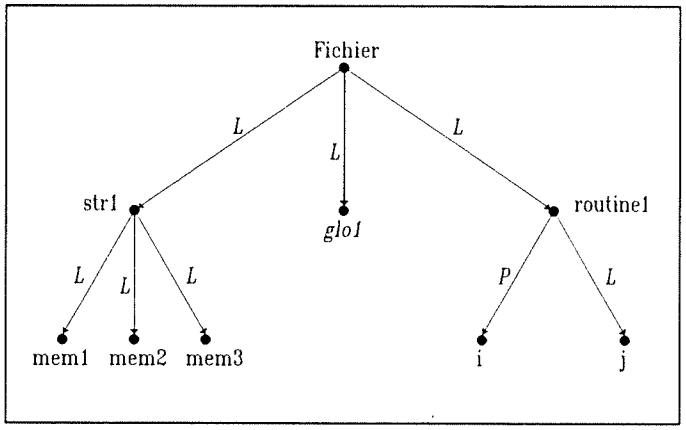


Figure 16 : Exemple de graphe de construction

Cette organisation des déclarations permet de conserver autant le type des identificateurs rencontrés que leur portée. Ce sont ces arcs qui servent d'ossature principale au graphe de construction. Le parcours du graphe le long de ces arcs permet de retrouver la disponibilité des identificateurs. À certaines particularités près, le sens et les caractéristiques du parcours du graphe sont relativement indépendant du langage de programmation.[26]

26: Une différence a rapport à l'allure générale du graphe par rapport aux niveaux. Par exemple, le Pascal permet plusieurs niveaux de profondeur par rapport aux unités fonctionnelles dans le graphe par la déclaration de procédures internes. Pour sa part, le langage C ne permet pas de profondeur pour les unités fonctionnelles, mais permet une profondeur à l'intérieur même de l'unité par l'utilisation d'une paire d'accolade ({}) pour déclarer localement des identificateurs.

À ces liens s'ajoutent les liens de transfert d'information par rapport au flux de données.

- Un arc du type **A** est généré pour démontrer l'**affectation** d'un identificateur par un autre identificateur. Par exemple, lorsqu'il unit deux variables, ce lien indique que la variable du sommet d'origine fait partie d'une expression qui sert à affecter la variable du sommet de destination. Lorsqu'il unit une constante à une variable, il indique qu'une constante a servi à affecter cette variable. Lorsqu'il unit une fonction à une variable il indique que le retour de la fonction sert à affecter la variable. Lorsqu'il unit une variable à une fonction il indique que la variable est utilisée pour donner la valeur de retour à la fonction[27]. Lorsqu'un arc de ce type unit une variable au booléen il indique que la variable a servi à déterminer la valeur résultante du booléen.

- Un arc du type **M** indique que l'identificateur du sommet d'origine sert à **modifier** l'évaluation de l'identificateur du sommet de destination. Par exemple, lorsqu'il unit une variable à une fonction il indique que la variable est utilisée comme paramètre à la fonction. Lorsqu'il unit une fonction à une variable il indique que l'appel de la fonction peut modifier la valeur de la variable (cas pour une variable passée par adresse, par exemple). En général ce type indique les relations entre les fonctions et leurs paramètres.

À ces liens s'ajoutent également les liens de transfert de contrôle qui sont habituellement reliés aux appels entre les unités fonctionnelles.

- Un arc du type U est ajouté entre le sommet de la routine courante et le sommet représentant une routine appelée par celle-ci. Et ce, peu importe si la routine appelée a ou n'a pas de paramètre, et peu importe si elle retrourne une valeur ou non. Cette information servira à construire le graphe d'appel qui est relativement indépendant du transfert d'information lors des appels.

Par exemple, le code partiel présenté à la figure 17 donne le graphe de construction présenté à la figure 18.

27: En Pascal, par exemple, la valeur de retour est donnée en utilisant un énoncé du genre: "Nom_Fctn := Nom_Var", alors qu'en langage C elle est donnée par l'énoncé "return (Nom_Var);". Ces deux cas auront la même forme dans le graphe puisqu'ils sont équivalents.

```
.
.
.
void routine1( i )

   int  i;

   {
      int  j;

      j = 10;
      i = i + j;

      {
         int  j;

         routine2(&j);
         if ( j > i ) i = j;
      }
      .
      .
      .
   }
```

Figure 17 : Code source pour le graphe de construction

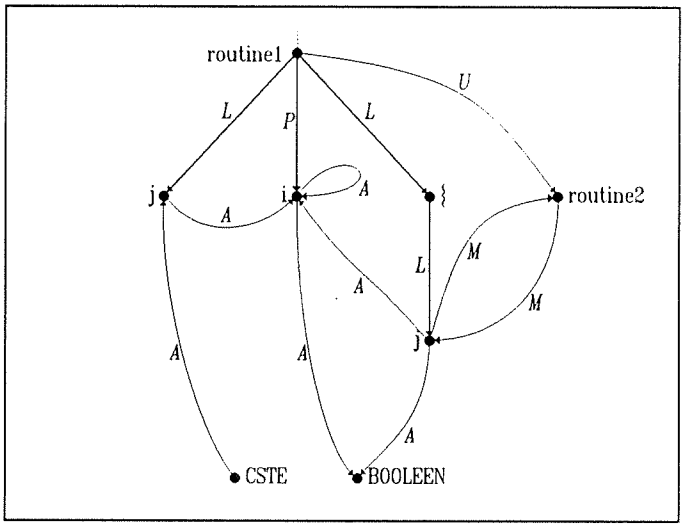


Figure 18 : Exemple de graphe de construction

Suite à l'analyse d'une routine, les arcs du type A, M et U sont retirés du graphe après avoir servi à la construction des autres graphes. Ces constructions seront expliquées dans les sections qui traitent spécifiquement de ces graphes. De plus, toute l'information relative aux déclarations internes à la routine et qui ne sont plus disponibles à l'extérieur de celle-ci peut être éliminée.

Il est possible de voir que le graphe de construction contient toute l'information qui sera répartie dans les autres graphes. Il pourrait sembler avantageux de conserver ce graphe au lieu

des trois autres. Cependant, plusieurs raisons justifient le choix de cette répartition des informations.

Tout d'abord, sous cette forme, le graphe est légèrement dépendant du langage de programmation. La standardisation de la forme du graphe et de son parcours ne serait cependant pas très difficile. Également, ne conserver que ce graphe impliquerait une énorme quantité d'information à traiter et à conserver en même temps et ce, autant pour l'analyse que pour les requêtes. Cette quantité d'information pose un problème d'espace (pour l'emmagasiner) et de temps (pour les recherches).

De plus, il est avantageux de passer par le graphe de construction au lieu de construire directement les autres graphes car toutes les recherches concernant les identificateurs disponibles y sont simplifiées. Le graphe de construction est donc très utile (indispensable) pour la construction dynamique[28] lors de l'analyse, mais devient inutile par la suite.

Information requise

Bien que les grandes lignes de la construction aient été vues dans la section précédente, certains détails de construction du graphe doivent être explicités en rapport à l'information qui est conservée.

Les principaux détails concernent la disponibilité et l'utilisation de la même variable par plusieurs unités fonctionnelles. Ainsi que l'utilisation par plusieurs variables d'un même type défini par un énoncé du langage de programmation.

Tel que vu précédemment, la disponibilité est caractérisée par la recherche des identificateurs dans un arbre contenant les déclarations. Cette recherche s'effectue, règle générale, dans l'arborescence directe puis dans l'arborescence montante par rapport au sommet courant[29]. Lorsque l'identificateur est retrouvé dans l'arborescence directe du sommet courant, il s'agit d'un identificateur déclaré par celui-ci, donc localement (L) ou par paramètre (P). Cependant, si l'identificateur est retrouvé dans l'arborescence montante, il s'agit alors d'un identificateur qui est vu globalement[30] puisque la déclaration est externe à la routine. Dans ce dernier cas, les liens qui seraient directement branchés sur cet identificateur pourraient faire en sorte que de l'information relativement à son utilisation serait perdue. Pour éviter ce problème, une copie du sommet représentant l'identificateur sera placée dans l'arborescence directe du sommet courant. Le nouveau lien entre le sommet de l'unité fonctionnelle analysée et l'identificateur copié sera étiquetté par le statut **E** pour démontrer qu'il vient d'une déclaration **externe**. Ce statut indique qu'il s'agit d'une copie d'un élément déjà déclaré à l'extérieur de cette unité.

Ce processus de copie sera également utilisé dans le cas d'identificateurs déclarés comme étant d'un type composé. L'arborescence du type sera copiée sous la définition de l'identificateur concerné en transformant les pointeurs sur un même type en arc remontant. Ceci a pour but de permettre d'identifier tous les cheminements possibles à travers les membres du type et aussi pour conserver l'information relative à l'utilisation des différents membres. Le code source de la figure 19 et le graphe de construction correspondant présenté à la figure 20 montrent cette utilisation. Les deux variables étant d'un même type composé, il devient nécessaire de copier l'arborescence de ce type afin de ne pas mélanger l'information de chacune des variables. Il est à remarquer que

---

28: Dynamique dans le sens où l'information apparaît et disparaît selon sa disponibilité en cours de traitement.

29: En correspondance avec les définitions de la théorie des graphes, l'arborescence *directe* est constituée de tous les fils d'un sommet, c'est-à-dire les descendants immédiats. Et l'arborescence *montante* est constituée des ancêtres d'un sommet et des fils de ces ancêtres (les sommets irrelatés de ces ancêtres).

30: Le terme *global* est utilisé pour désigner un identificateur qui est disponible par au moins une unité fonctionnelle différente de celle qui l'a déclaré.

l'arborescence du type ne forme pas de cycle s'il y a un membre qui est un pointeur à lui-même, mais qu'il y a cycle orienté dans les copies effectuées pour les variables.

```
void routine ()

{

struct str1 {
              int          i;
              float        j;
              struct str1 *suiv;
            };

struct str1 *var1;
struct str1 *var2;


var1.i = var2.suivant.j;
var2.j = var2.suivant.i;
       .
       .
       .
}
```

Figure 19 : Code source pour le graphe de construction

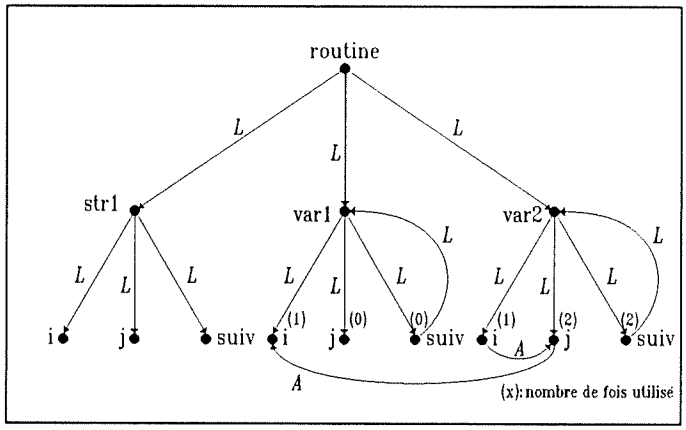


Figure 20 : Exemple de graphe de construction

L'exemple précédent permet également d'introduire le traitement des pointeurs. Afin de conserver les différentes utilisations des variables déclarées comme étant des pointeurs à un type quelconque, il faut séparer les cas où le pointeur est modifié et les cas où c'est le contenu de l'élément pointé qui l'est. Ceci est effectué en associant un sommet à l'identificateur représentant le pointeur et un autre sommet (son fils) pour représenté le contenu pointé. Il est possible de voir que le cas d'un type composé avec un pointeur sur lui même n'est qu'un cas spécial de cette règle.

En général, le processus de copie permet également de simplifier le traitement dynamique du graphe. En effet, le fait de copier les éléments appartenant aux ancêtres sous le sommet courant (unité fonctionnelle analysée) permet de conserver tous les liens sous ce sommet.

Autrement dit, aucun lien ne remonte à un niveau supérieur à ce sommet. Le parcours pour transporter l'information dans les autres graphes est donc limité à ce secteur et, surtout, l'élimination de l'information de la routine peut se faire beaucoup plus facilement et plus prudemment. La destruction consistera à éliminer tout ce qu'il y a sous le sommet de l'unité fonctionnelle qui vient d'être analysée.

4.2 Éléments du graphe de construction

Cette section résume la liste des éléments énumérés précédemment. Les composantes du graphe de construction sont dénommées par les termes typiques de la théorie des graphes. C'est-à-dire que le graphe est composé de sommets et d'arcs. Les sommets doivent permettre de représenter tous les éléments pouvant être déclarés et ceux permettant de regrouper ces éléments afin de conserver les régions de validité de ceux-ci. De plus, ils doivent représenter tous les véhicules de transfert de données disponibles afin de pouvoir représenter le flux de ces données. Les arcs représentent, d'une part, les relations de déclarations et, d'autre part, les relations qui existent entre les véhicules d'information.

Sommets:

Tel que mentionné précédemment, les sommets du graphe de construction doivent représenter tous les véhicules disponibles pour conserver, modifier ou utiliser l'information dans une routine. Les différents véhicules disponibles peuvent être regroupés en quatre *genres* typiques. Les quatre genres représentent les **constantes**, les **variables**, les **routines** et les **booléens**.

- Les *constantes* représentent tous les éléments d'information qui ne peuvent varier (qui reste constant). Par exemple, il peut s'agir de valeurs numériques ou de chaînes de caractères utilisées directement, de constantes définies par un énoncé du langage de programmation, ou de tout autre élément dont la valeur ne peut être modifiée. Les deux sortes de constantes pouvant être distinguées à l'intérieur de ce groupe sont les *constantes directes* et les *constantes par identificateurs*. Le premier groupe représente une utilisation directe d'une constante sous une forme numérique ou de chaîne de caractères, alors que le second groupe représente l'utilisation d'un identificateur pour utiliser la constante.[31]

- Les *variables* doivent représenter les éléments d'information qui peuvent prendre une certaine valeur parmi une région définie. Cependant, cette valeur doit être connue et fixe avant l'appellation de l'élément. En d'autres mots, l'appellation de l'élément ne doit pas faire varier (évaluer) la valeur de son contenu. Toutes les variables de la plupart des langages véhiculent l'information sous cette forme. Par exemple, les variables du type entier, réel, pointeur à un entier et plusieurs autres font partie de cette forme. Le cas d'un tableau indicé fait aussi partie de cette forme car, bien que l'indice puisse être lui-même une variable, le contenu du tableau ne varie pas lors de son appellation. Dans ce dernier cas, il ne s'agit que de la sélection d'un élément.

- Les *routines* représentent les éléments compléments du genre variable dans le sens où elles représentent les éléments qui sont variables et qui contiennent une information dont la valeur est évaluée lors de l'appel. Il s'agit principalement des unités fonctionnelles (fonctions, procédures, etc.) et de tout autre appel similaire qui permet de déterminer une valeur.

31: Les constantes ayant un identificateur associé peuvent être conservées séparément selon chacun des identificateurs. Cependant, les constantes utilisées directement (valeurs numériques, chaînes de caractères, etc.) peuvent être représentées par un seul sommet commun.

- Les *booléens* représentent une entité qui peut prendre une valeur dans le seul but de prendre une décision à partir du résultat obtenu. Le résultat final ne peut en aucun temps servir directement pour modifier l'information d'une autre forme de données. Il s'agit en général des expressions booléennes des conditionnelles.[32]

De plus, certains sommets du graphe de construction permettront de représenter les différents **types** pouvant être définis par un énoncé du langage de programmation.

À chacun de ces genres peut être associé différentes caractéristiques qui identifient plus précisément les composantes principales de l'élément d'un genre donné.

Arcs:

Les arcs relient les sommets du graphe. Le premier type d'arc indique les relations de *déclarations* entre les identificateurs. Ils sont étiquetés par les lettres **L**, **P**, ou **E** selon le mode de déclarations qu'ils représentent. Les identificateurs qui sont déclarés comme étant des **paramètres** sont reliés par des arcs nommés **P**. Les identificateurs qui sont déclarés à l'**extérieur** de la routine présentement analysée sont reliés par des arcs nommés **E**. Les identificateurs qui sont déclarés **localement** par la routine présentement analysée sont reliés par des arcs nommés **L**.

Les autres arcs servent à conserver l'information pertinente au graphe du flux de données, au graphe d'appel et au graphe des déclarations.

Les arcs qui servent à identifier qu'un véhicule d'information a servi à **affecter** un autre véhicule d'information sont identifiés par la lettre **A**. Les arcs qui servent à identifier qu'un véhicule d'information a servi pour **modifier** un autre véhicule d'information sont identifiés par la lettre **M**. Ces deux types d'arc permettront de conserver les transferts d'information et serviront, par le fait même, à construire le graphe du flux de données.

Les arcs notés **U** servent à identifier les liens entres les unités fonctionnelles. Elles indiquent que la routine représentée par le sommet d'origine a **utilisé** la routine représentée par le sommet de destination. Cette information sert par la suite à la construction du graphe d'appel.

Pour sa part, le graphe des déclarations est construit à partir des informations relatives à l'ossature du graphe de construction et des autres arcs relatifs aux données. Les explications sur la façon de tirer l'information nécessaire pour ce graphe seront présentées ultérieurement.

4.3 Définition du contenu d'un sommet

Les différents sommets qui peuvent être retrouvés dans le graphe représentent les éléments suivants:

- Le sommet principal qui est celui du nom du fichier analysé.
- Le sommet de l'unité fonctionnelle analysée.
- Les sommets représentant chacune des fonctions, des variables et des constantes (par identificateur) qui sont déclarées.
- Les sommets représentant chacunes des variables, des fonctions et des constantes (par identificateur) qui sont utilisées dans une expression de la fonction analysée.

32: Les booléens peuvent être représentés par un seul sommet.

- Les sommets représentant chacun des types déclarés et chacun de ses membres.
- Un sommet représentant toutes les constantes du type directe.
- Un sommet représentant tous les booléens.

Selon les besoins mentionnés précédemment et selon les différents genres de sommets pouvant être présents, les éléments qui sont contenus dans les sommets du graphe de construction sont les suivants:

A) Le **nom de l'identificateur** pour permettre de retrouver les symboles recherchés lors de l'analyse et de vulgariser les informations.

B) **Genre du sommet** (ou de l'identificateur). Ce genre doit être un et un seul des genres suivants:
   - *Constante* : pour les constantes directes et par identificateur.
   - *Variable* : pour tous les identificateurs représentant une variable.
   - *Routine* : pour tous les identificateurs représentant une routine.
   - *Booléen* : pour représenter le registre utilisé pour évaluer les booléens.
   - *Type construit* : pour tous les types définis par un énoncé du langage.

C) **Classe du type** de l'identificateur. Elle doit être vide lorsqu'elle ne s'applique pas, ou une et une seule des classes suivantes:
   - *Unitaire* : pour représenter les types "primaires".
   - *Composé* : pour représenter les types associés à un regroupement de types.

D) L'**attribut du type**. Cet attribut a pour but de montrer la façon dont la référence à l'élément est effectuée selon la déclaration. Il peut prendre une ou plusieurs des classes qui suivent. Lorsqu'il ne s'applique pas, l'attribut n'aura aucune de ces classes.
   - *Élément* : pour indiquer qu'il y a un élément (i.e. l'attribut s'applique).
   - *Pointeur* : pour indiquer qu'il s'agit d'un pointeur à un type donné.
   - *Vecteur* : pour indiquer qu'il s'agit d'un vecteur d'un type donné.

E) Le **nom du type** de l'identificateur pour permettre de vulgariser ce type. Bien que la représentation associée au nom du type soit dépendante du langage de programmation analysé, le fait d'utiliser ce nom sous forme de texte ne pose pas de dépendence vis-à-vis du langage de programmation.

F) Le **nombre de fois utilisé**. Ce champ est particulièrement utile pour connaître l'utilisation des membres des variables de type composé.

Le tableau 14 permet de montrer l'utilisation de ces champs pour identifier certains cas possiblement rencontrés.

exemple	nom identificateur	genre sommet	classe type	attribut type	nom type
int var1;	var1	variable	unitaire	élément	int
struct str1 *var2;	var2	variable	composé	pointeur élément	str1
int var3[10];	var3	variable	unitaire	vecteur élément	int
a = 10;	NULL(pour 10)	constante	unitaire	élément	-
const b 5;	b	constante	unitaire	élément	-
struct str1 {...};	str1	type con.	composé	-	-
	NULL	booléen	unitaire	-	-
int fctn1(x)	fctn1	routine	unitaire	élément	int
void fctn2()	fctn2	routine	-	-	-

Tableau 14 : Exemples d'utilisation des champs des sommets du graphe de construction

4.4 Définition du contenu d'un arc

Cette section montre le contenu des arcs du graphe de construction. L'information de ces arcs doit permettre d'indiquer la position de la relation dans le code source avec le même index que celui utilisé par le graphe du flux de contrôle. Ceci permettra entre autre de distinguer les identificateurs identiques par un autre moyen que de conserver l'arborescence[33] et d'étudier les relations selon leur position dans le code source aussi bien que dans le graphe de contrôle. De plus, les arcs reliant les sommets indiquent le statut de la relation entre ces sommets. Comme il fut mentionné précédemment, les relations sont reliées à la déclaration (**P**, **L**, **E**), au flux de données (**A**, **M**) ainsi qu'aux appels entre les unités fonctionnelles (**U**).

Plus spécifiquement, le contenu des arcs du graphe de construction est le suivant:

A) Position dans le fichier source de l'énoncé qui a créé la relation. Cette position est appelée l'index et est définie comme l'est l'index du graphe de contrôle.

B) Statut de la relation. Il peut être un et un seul des cas suivants:

- *P* : pour les identificateurs déclarés comme paramètres.
- *L* : pour les identificateurs déclarés localement.
- *E* : pour les identificateurs déclarés à l'extérieur de la routine analysée.
- *A* : pour les relations où un identificateur affecte un autre identificateur.

33: La présence de l'index de déclaration n'est pas réellement utile pour les arcs de déclarations de ce graphe mais permettra de distinguer les identificateurs qui ont un nom identique lors du transfert des information dans les autres graphes. Ceci est nécessaire afin de ne pas avoir à conserver l'arbre des déclarations.

- *M* : pour les relations où un identificateur modifie l'évaluation d'un autre identificateur.
- *U* : pour représenter l' appel d'une routine par une autre.

5. Graphe du flux de données

Le graphe du flux de données doit conserver l'information nécessaire pour retrouver les différents transferts d'information (de données) à l'intérieur de la routine. Cette représentation pourra permettre entre autre de visualiser et comprendre le flux de données qui s'effectue à l'intérieur d'une routine. De plus, certaines métriques relatives au flux de données et au transfert d'information pourront plus facilement être définies et calculées avec ce modèle.

L'avantage principal de cette méthode de représentation est qu'elle rend l'étude du flux de données indépendante du langage de programmation. Les éléments du graphe se doivent donc de permettre de contenir et de représenter les différents véhicules d'information et les différentes méthodes de transfert d'information possible dans les langages procéduraux.

Des études de la structure du graphe formé par les relations concernant le flux de données pourront permettre d'en révéler les caractéristiques, la complexité, ainsi que plusieurs autres informations. Ces études peuvent faire partie d'un travail ultérieur.

La construction de ce graphe prend son information à partir de ce qui a été construit à l'intérieur du graphe de construction précédemment décrit.

5.1 Caractérisation du transfert de données

Les véhicules disponibles pour conserver, modifier ou utiliser l'information dans une routine ont précédemment été regroupés en quatre groupes appelés *genres*. Plusieurs actions de transfert peuvent survenir entre ces genres. Ceux qui peuvent être affectés, ou dont le résultat (la valeur) peut être influencé par les autres genres sont les variables, les routines (par leurs paramètres), et les booléens. Les formes qui peuvent servir à modifier l'information véhiculée sont les constantes, les variables et les routines.[34]

La représentation de ces relations sous la forme de graphe peut se faire en associant les sommets aux genres définis précédemment. Comme pour le graphe de construction, chacun des sommets est un identificateur représentant soit une variable, soit une fonction, soit un booléen, soit une constante. Pour leur part, les arcs du graphe servent à représenter l'orientation du flux d'information. L'orientation des arcs du graphe permet d'indiquer quel sommet (variable, fonction, etc.) a servi à affecter ou à modifier quel autre sommet. Les différents branchements possibles entre les sommets correspondent aux différents types de transfert d'information identifiés à la section précédente. Par exemple, si l'appel à une fonction apparaît dans l'expression qui affecte une variable, alors un lien orienté existe à partir du sommet représentant cette fonction vers le sommet représentant la variable concernée.

Le graphe ainsi construit n'a pas réellement de tête. Il s'agit beaucoup plus d'un ensemble de sommets reliés entre eux par des liens dispersés. Différents algorithmes peuvent par la suite être appliqués à ce graphe pour en identifier des caractéristiques conceptuelles. Il sera aussi important de conserver la façon dont l'identificateur a été déclaré (P, L ou E). De cette façon, les données d'entrée et de sortie seront plus facilement identifiables peu importe si ces données étaient contenues dans les identificateurs déclarés comme étant des paramètres ou comme étant

34: La seule forme non présente dans ce dernier groupe est le booléen car il ne permet que de modifier la direction du flux de contrôle et non le flux de données.

des identificateurs externes. La figure 21 permet de schématiser les informations qui servent à l'entrée de la routine et celles qui sortent de la même routine après un traitement quelconque.

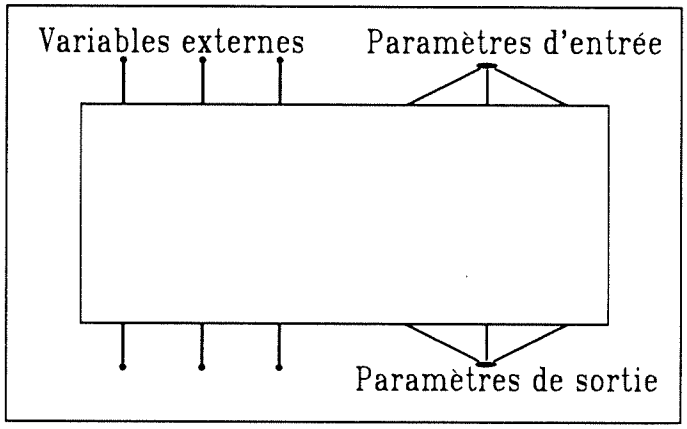


Figure 21 : Schématisation des entrées-sorties d'une routine

5.2 Construction et représentation interne du graphe

Il est possible de voir que le graphe de flux de données est simplement dérivé du graphe de construction présenté précédemment. Tous les sommets fils de l'unité fonctionnelle analysée sont copiés directement. Un cas particulier apparaît dans le cas des variables qui sont déclarées comme étant d'un type composé (ayant des membres). Dans ce cas, le sous-graphe composé des membres n'est pas copié. Chacune des possibilités d'expansion de cette variable génère un sommet qui est en réalité une autre variable.

Par la suite, il s'agit simplement de transférer les arcs du type A et du type M du graphe de construction sur leurs sommets associés dans le graphe du flux de données. Il sera donc possible de facilement identifier les sommets qui ne sont pas référencés par aucune action.

Le contenu de ces arcs va permettre de localiser l'endroit où est apparu ce flux de données par rapport au graphe du flux de contrôle. La précision de l'endroit où est apparu ce transfert d'information n'est nécessaire qu'à l'énoncé près. L'ordre d'apparition et d'évaluation des opérandes nécessaires pour former l'énoncé n'a pas d'importance.

Distinction des liens

Le niveau d'information contenu dans ce graphe concerne plus spécifiquement les liens représentant les tranferts d'information entre les éléments constituant un programme. Ce sont ces liens qui vont réellement permettre de représenter le flux de données. Chaque lien peut prendre une de deux identifications pour représenter la façon dont le flux de données est effectué. Les identifications disponibles sont:

- A pour indiquer que le sommet d'origine a *affecté* une valeur au sommet de destination, et;
- M pour indiquer que le sommet d'origine a *modifié* l'évaluation du sommet de destination.

La présence de ces deux identifications permet de distinguer certaines ambiguïtés possibles. Par exemple, l'appel à une fonction avec un paramètre qui est de la forme variable et du

type passé par adresse crée deux liens. Un premier lien orienté de la variable vers la fonction pour identifier que la variable modifie le résultat de l'évaluation de la fonction. Alors qu'un second lien unit la fonction à la variable (sens inverse du précédent) pour indiquer que la fonction a (possiblement) modifié la variable. Ce dernier lien pose l'ambiguïté car on ne peut distinguer si la relation est causée par le passage d'un paramètre par adresse ou si elle est causée par l'affectation de la variable par le résultat de l'évaluation de la fonction. L'identification des liens permet de distinguer ces deux cas. Comme le montre la figure 22, le cas de la variable passée par adresse produit une identification M, tandis que l'autre cas produit une identification A. Seule cette identification permet de distinguer les deux cas. Un autre exemple similaire permettrait de voir que pour le cas où la valeur de retour de la fonction active est spécifiée en affectant le nom de la fonction par la valeur ou l'expression désirée (en Pascal par exemple), l'ambiguïté est également levée par cette distinction.

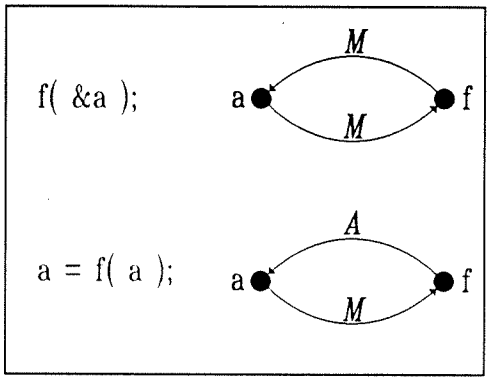


Figure 22 : Exemple de graphes de flux de données

Pour leur part, le genre, la classe de type et l'attribut associés à chacun des sommets permettront d'identifier plus précisément les éléments en cause dans ces transferts d'information.

5.3 Définition du contenu d'un sommet

Les différents sommets qui peuvent être retrouvés dans le graphe contiennent les mêmes informations que ceux du graphe de construction présenté précédemment. Cependant, il s'en ajoute quelques unes, qui proviennent de certaines informations des arcs du graphe de construction. Les éléments qui s'ajoutent sont:

- L'**index** de déclaration des identificateurs. Tel que mentionné précédemment, cet index permettra de distinguer les identificateurs identiques et est le même que celui du graphe de contrôle.

- Le **statut de la déclaration** de l'identificateur. Il permet de connaître la portée de son action. Les trois possibilités sont celles mentionnées lors de la définition du contenu des arcs du graphe de construction. C'est-à-dire:
  - *P* : pour les identificateurs déclarés comme paramètres.
  - *L* : pour les identificateurs déclarés localement.
  - *E* : pour les identificateurs déclarés à l'extérieur de la routine analysée.

5.4 Définition du contenu d'un arc

L'information des arcs du graphe du flux de données doit permettre d'indiquer la position de la relation dans le code source avec le même index que celui utilisé par le graphe du flux de contrôle. De plus, les arcs reliant les sommets indiquent le type de relation entre ces sommets. Le type peut être A, ou M.

Le contenu des arcs est tel que mentionné précédemment pour ceux du graphe de construction:

5.5 Exemple de construction du graphe du flux de données

À partir de l'exemple précédemment construit lors de la définition du graphe de construction (figure 23 répétée ci-après) il est possible de mieux illustrer la construction du graphe du flux de données. Le graphe du flux de données aura les liens montrés à la figure 24.

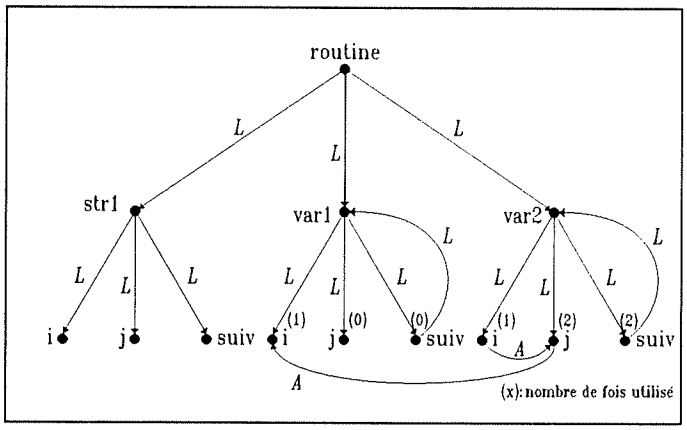


Figure 23 : Exemple de graphe de construction

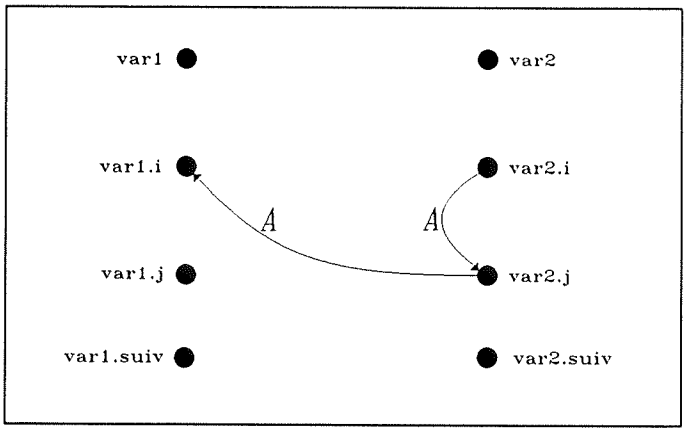


Figure 24 : Exemple de graphe de flux de données

6. Graphe d'appel

Le graphe d'appel conserve les liens des différents appels entre les routines. Ceci permet de pouvoir plus simplement retrouver non seulement les routines appelées par une routine particulière, mais aussi de retrouver qui appelle cette routine.

Il faut noter qu'une partie de ces liens est contenue de façon indirecte dans chacun des graphes du flux de données des routines. Cependant, ces liens ne concernent que les fonctions appelées qui ont soit des paramètres, soit une valeur de retour. Il est donc très avantageux de réunir dans un seul et même graphe tous les appels entre les routines de façon à pouvoir retracer certains effets globaux relatifs aux appels. De plus, la définition et le calcul de certaines métriques relatives à ce domaine seront plus facilement établis. Un certain niveau de représentation peut même être envisagé.

La forme de ce graphe peut être très diversifiée selon le projet analysé. Dans le cas d'un projet complet, le graphe devrait posséder un seul sommet d'entrée (tête). De plus, le phénomène de cycle dans ce graphe ne sera présent que s'il y a récursion dans les appels. Une boucle[35] démontrera un appel récursif de niveau zéro. De façon plus générale, un circuit composé de N arcs démontrera un appel récursif de niveau N-1. Certaines opérations sur la matrice associée de ce graphe pourront permettre de déterminer l'occurence de ce type d'appel.

6.1 Construction du graphe

Les sommets du graphe représentent les routines et sont identifiés par les identificateurs (noms) de ces routines. Chacun des sommets contient l'information pertinente relative à la routine qu'il représente. L'information doit permettre entre autre d'identifier la routine, de retrouver la position de son code source, et aussi de contenir des indicateurs de son statut et de ses caractéristiques. Ces caractéristiques sont quelques peu reliées au fait que la liste des sommets de ce graphe sert de coeur à l'outil d'évaluation de code source puisqu'ils correspondent à chacune des unités de base de l'analyse.

Un arc reliant deux sommets indique que la routine du sommet d'origine appelle la routine du sommet de destination. L'information à conserver pour chacun des appels est l'endroit de cet appel dans le code source de la routine appelant. L'utilité de cette information tient son sens par la relation potentielle avec l'étude du flux de données pour les données communes. Il y a donc autant de liens entre deux routines qu'il y a d'appel, ce qui conduit au fait que ce graphe correspond à un multigraphe.

La construction du graphe d'appel est incrémentale. À chaque routine analysée, l'information relative à ce graphe est ajoutée à celle déjà accumulée. Cette information provient directement du graphe de construction précédemment décrit. Chaque arc identifié U dans ce dernier graphe génère un arc (et les sommets correspondant s'ils n'existent pas déjà) dans le graphe d'appel.

Ceci mène donc à la création d'un graphe représentant typiquement la hiérarchie des appels entre les unités fonctionnelles.

35: Selon les définitions précédentes de la théorie des graphes, une boucle est un arc dont le sommet d'origine est le même que celui de destination.

6.2 Définition du contenu d'un sommet

Les éléments qui sont contenus dans les sommets du graphe d'appel doivent permettre d'indiquer les différentes caractéristiques de chaque routine. La liste de ces éléments est la suivante:

Identification de la routine

A) Nom de la routine.

B) Nom des parents.

Position du code source de la routine

C) Nom du fichier contenant la routine.

D) Numéro de la ligne où débute la routine.

E) Numéro de la ligne où se termine la routine.

Caractéristiques de la routine

F) État de la routine analysée (il peut y avoir plus d'un état à la fois):

- Routine de système
- Routine ne rencontrant pas les normes

G) Nombre de fois que la routine a été analysée.

H) Nombre de lignes de la routine

I) Nombre d'énoncés de la routine.

J) Type d'analyseur utilisé pour l'analyse de cette routine.

K) La somme des poids des métriques non respectées.

L) Nombre de métriques non respectées.

M) Numéro de la métrique non respectée.

N) Écart maximum de la métrique non respectée.

6.3 Définition du contenu d'un arc

Le contenu des arcs du graphe d'appel permet de retrouver où est apparu l'appel dans la routine appelant. Cette position est l'index et correspond à celui conservé dans le graphe du flux de contrôle.

7. Graphe des déclarations

Le graphe des déclarations permet de conserver les liens entre les routines, les variables locales et globales (variables disponibles à plus d'une routine) et les types définis. En plus de permettre de conserver quelle variable et quel type a été disponible pour quelle routine, l'information sur la déclaration, l'utilisation et la modification des variables est conservée par rapport aux routines. L'utilisation et la déclaration des types sont également conservés par rapport aux routines. Finalement, des liens existent aussi entre les variables et les types de façon à identifier le type de chacune des variables.

Ce graphe peut être vu comme étant composé par trois groupes de sommets possédant des liens inter-groupes. Un groupe représente les routines, un autre les variables alors que le troisième représente les types. Chaque élément du groupe des routines est relié à certains éléments du groupe des variables et du groupe de types. Si le lien n'existe pas c'est que la variable ou le type n'est pas disponible pour cette routine. Si le lien existe, alors la variable ou le type est disponible pour la routine. Dans ce dernier cas, l'arc contient l'information qui dit si la variable ou le type a été déclaré, utilisé ou modifié (pour la variable seulement) par cette routine. Si un lien existe entre une variable et un type, alors la variable est du type indiqué par le sommet de destination. La structure du graphe s'apparante donc à celle d'un graphe simple à trois composantes.

Ce graphe permettra de conserver plusieurs informations relatives au projet entier. Entre autre, les variables qui sont disponibles par plus d'une unité pourront être facilement identifiées. L'utilisation des types pourra être étudiée de façon à permettre de savoir si tous les membres d'un type particulier sont utilisés et de prévoir les implications d'une modification à un type. Ainsi, plusieurs autres informations peuvent être extraites de ce graphe et ce, autant au niveau de la complexité qu'au niveau de la documentation.

7.1 Construction du graphe

Le graphe des déclarations est unique pour un projet. L'analyse de chaque unité fonctionnelle permet d'ajouter l'information pertinente à ce graphe par le biais du graphe de construction. La construction du graphe des déclarations est donc incrémentale. À chaque routine analysée, l'information relative à ce graphe est ajoutée à celle déjà accumulée.

Les sommets du graphe représentent les routines, les variables et les types[36]. Ils sont principalement identifiés par leurs identificateurs (noms).

Un arc reliant deux sommets permet d'indiquer la relation entre les deux sommets concernés.

- Si l'arc réunit une routine à une variable, alors il indique que la variable du sommet de destination était disponible pour la routine du sommet d'origine.
- Si l'arc réunit une routine à un type, alors il indique que le type du sommet de destination était disponible pour la routine du sommet d'origine.

36: Les variables définies comme étant d'un type composé et les type composés eux-mêmes seront transformés comme dans le cas du graphe du flux de données. C'est-à-dire que chacune des possibilités des types seront étendues.

- Si l'arc réunit une variable à un type, alors il indique que la variable du sommet d'origine était du type du sommet de destination .

Un de ces arcs sera créé entre deux sommets lorsque le graphe de construction permettra de déterminer une des relations précédemment définies. En plus de la présence des arcs (qui indiquent la disponibilité), des indicateurs sont nécessaires pour montrer d'autres caractéristiques de la relation.

- Pour un arc reliant une routine à une variable:
  - Si une variable a été déclarée par une routine, alors un indicateur de *déclaration* (D) est contenu dans l'arc.
  - Si une variable a été modifiée à au moins un endroit dans une routine, alors un indicateur montrant une *modification* (M) est contenu dans l'arc.
  - Si une variable a été utilisée (excluant les cas où elle a été modifiée) à au moins un endroit dans une routine, alors un indicateur montrant une *utilisation* (U) est contenu dans l'arc.
- Pour un arc reliant une routine à un type:
  - Si un type a été déclaré par une routine, alors un indicateur de *déclaration* (D) est contenu dans l'arc.
  - Si un type a été utilisé à au moins un endroit dans une routine (pour définir une variable), alors un indicateur montrant une *utilisation* (U) est contenu dans l'arc.
- Pour un arc reliant une variable à un type aucune indication n'est nécessaire.

Il est important de mentionner que les indicateurs ne peuvent être présents que s'il y a disponibilité. Donc ils ne sont présents que s'il y a un arc, ce qui élimine les cas où il pourrait y avoir un indicateur sans arc.

7.2 Définition du contenu d'un sommet

Chacun des sommets doit contenir l'information pertinente relative à son identification. Les sommets du graphe des déclarations sont les mêmes que ceux du graphe de construction. Certains sommets représentent les routines, d'autres représentent les variables, alors que les autres représentent les types. Les éléments qui sont contenus dans les sommets du graphe des déclarations sont les mêmes que ceux contenus dans le graphe de construction auquels il faut ajouter l'index de la déclaration de l'identificateur afin de dissocier les identificateurs identiques.

7.3 Définition du contenu d'un arc

Les arcs du graphe des déclarations relient toujours une routine à une variable ou à un type, ou une variable à un type. Les différentes possibilités de présence d'arcs ont été vues précédemment. Le contenu des arcs du graphe des déclarations est le suivant:

A) **Statut de la relation** entre les deux sommets. Il s'agit de zéro, un, ou plusieurs des cas suivants dont la signification a été décrite précédemment.
   - *D* : le sommet d'origine a *déclaré* le sommet de destination.
   - *M* : le sommet d'origine a *modifié* le sommet de destination.

- U : le sommet d'origine a *utilisé* le sommet de destination.

8. Conclusion

Les travaux présentés dans ce document reposent sur des concepts nouvellement étudiés. Une évolution du modèle développé est donc à prévoir. De plus, l'orientation des futurs travaux reliés à ce rapport devrait concerner principalement:

- l'étude de l'implantation du modèle (compromis et modifications);
- l'étude des applications du modèle.

9. Références

[BERG81], G. D. Bergland, *A Guided Tour of Program Design Methodologies*, IEEE Computer Society, Los Alamitos, Calif., pp. 13-37, 1981.

[BOHL71], M. Bohl, *Flowcharting Techniques*, Science Research Associates, Chicago, Ill., 1971.

[COUL83], N. S. Coulter, "Software Science and Cognitive Psychology", *IEEE Transactions on Software Engineering*, SE-9(2), pp. 166-171, 1983.

[FERS78], O. Ferstk, "Flowcharting by Stepwise Refinement", *SIGPLAN Not.*, Vol. 13, No 1, pp. 34-42, 1978.

[HALS77], M. H. Halstead, *Elements of Software Engineering*, North-Holland, 1977.

[HAME82], P. G. HAMER, G. D. FREWIN, "M. H. Halstead Software Science: a critical examination", *Proceedings of the Sixth International Conference on Software Engineering*, Tokyo, Japan, 1982.

[KERN85], B. W. Kernighan, D. M. Ritchie, *Le langage C*, Masson, 1985.

[LASS81], J. L. Lassez, D. Van des Knijff, J. Shepher, C. Lassez, "A Critical Examination of Software Science", *Journal of Systems and Software*, Vol. 2, pp.105-112, 1981.

[LIND77], C. H. Lindsey, "Structure Charts, a Structured Alternative to Flowcharts", *SIGPLAN Not.*, Vol. 12, No 11, pp. 36-49, 1977.

[MART85], J. Martin, C. McClure, *Diagramming Techniques for Analysts and Programmers*, Prentice-Hall, Englewood Cliffs, N.J., 1985.

[MCCA76], T. J. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, SE-2(4), pp. 308-320, 1976.

[PELL66], R. Pellet, *Initiation à la théorie des graphes*, Entreprise moderne d'édition, Paris, 1966.

[RAMS83], H. R. Ramsey, M. E. Atwood, J. R. Van Doren, "Flowchart versus Program Design Languages: An Experimental Comparison", *Communications of the ACM*, Vol. 26, No 6, pp. 445-449, 1983.

[ROBI85], P. N. Robillard, "A software tool and a schematic notation that improve the use of programming langages", *Proceedings of Softfair Conference II*, San Fransisco, pp. 149-158, 1985.

[ROBI86], P. N. Robillard, "Schematic Pseudocode for program constructs and its computer Automation by SCHEMACODE", *Communications of the ACM*, Vol. 29, No 11, pp. 1072-1089, 1986.

[SCHN79], N. F. Schneidewind, H. Hoffman, "An Experiment in Software Error Data Collection and Analysis," *IEEE Transactions on Software Engineering*, SE-2(4), pp. 276-286, 1979.

[SCHN83], N. F. Schneidewind, "A Complexity Metric which Integrates Structural and Textual Metrics," *2nd Annual Phoenix Conference*, March 14-16 1983, pp. 95-99.

[SHAW82], C. Shaw, "Structure Charts for Jackson Structured Programming", *ACM SIGSOFT*, Vol. 7, No 1, pp. 78-81, 1982.

[SHEN83], V. Y. Shen, S. D. Conte, H. E. Dunsmore, "Software Science Revisited: a critical analysis of the theory and its empirical support", *IEEE Transactions on Software Engineering*, SE-9(2), pp. 155-165, 1983.

[SHEP88], M. Shepperd, "A Critique of Cyclomatic Complexity as a Software Metric", *Software Engineering Journal*, March 88, pp. 30-36, 1988.

[SHNE77], B. Shneiderman, R. Mayer, D. McKay, P. Heller, "Experimental Investigations of the Utility of Detailed Flowcharts in Programming", *Communications of the ACM*, Vol. 20, No 6, pp. 373-381, 1977.

[WHIT87], R. Whitty, "Comments on Control Flow as a Measure of Program Complexity", *UK Alvey Programme Software Reliability and Metrics Club Newsletter*, 5, pp.1-2, 1987.