| | |
|---|---|
| **Titre:** Title: | Proposition d'un schéma relationnel pour le modèle DATGRAPH |
| **Auteurs:** Authors: | Jean Mayrand, Pierre N. Robillard, & Mario Simoneau |
| **Date:** | 1990 |
| **Type:** | Rapport / Report |
| **Référence:** Citation: | Mayrand, J., Robillard, P. N., & Simoneau, M. (1990). Proposition d'un schéma relationnel pour le modèle DATGRAPH. (Rapport technique n° EPM-RT-90-20). https://publications.polymtl.ca/10069/ |



| | |
|---|---|
| **Institution:** | École Polytechnique de Montréal |
| **Numéro de rapport:** Report number: | EPM-RT-90-20 |
| **URL officiel:** Official URL: | |
| **Mention légale:** Legal notice: | |

# Proposition d'un schéma relationnel pour le modèle DATGRAPH

Jean Mayrand, Ing. jr,
*Étudiant M. Sc. A.*

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

Pierre N. Robillard, Ph. D., Ing.,
*Directeur du laboratoire de recherche en génie logiciel*

gratuit

ÉCOLE POLYTECHNIQUE DE MONTRÉAL
Département de génie électrique et informatique
Laboratoire de génie logiciel

DÉCEMBRE 1990

Dépôt légal, 4e trimestre 1990
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) 340-4473

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'organismes canadiens.

# Table des matières

# Liste des figures

# Liste des tableaux

# Introduction

Ce rapport poursuit les travaux entrepris dans le rapport EMP/RT-89/21 sur la définition et la normalisation des entités[1] à conserver lors de l'analyse d'un fichier source. La première étude a produit le modèle DATGRAPH et ce document se veut une prolongation des bases posées en 1989.

Premièrement, l'étude présentée permet de relier le modèle DATGRAPH à une identification des différents niveaux d'abstraction disponibles lors de l'analyse de fichiers de code source. Une des composantes de l'évaluation de la qualité d'un produit informatique est le code source. L'évaluation du code source par niveaux d'abstraction vise à classifier les différentes activités de mesure possible sur le produit informatique. Le terme niveau est utilisé afin de mettre l'emphase sur la dépendance hiérarchique de ces activités de mesure. Ces niveaux partent des caractéristiques simples des fichiers et vont jusqu'à l'information nécessaire pour mener à terme une étude complète des dépendances entre les différentes composantes formant le logiciel.

Deuxièmement, l'étude présente le schéma conceptuel de la structure de données permettant de capturer les entités des fichiers sources et de l'environnement de développement d'un projet informatique. Ce schéma conceptuel est sous la forme d'un modèle d'entité-association. Le modèle d'entité-association est une méthode de design de schéma conceptuel qui est très utilisée dans le domaine des bases de données. Le modèle proposé se veut général et adapté aux langages procéduraux et aux langages objets. Le modèle supporte les grands concepts de ces langages et évite d'être particulier.

---

1 Une entité est un élément provenant d'un fichier source auquel il est possible d'attribuer un nom et de mesurer ses propriétés. Les entités provenant du code source sont les variables, les types construits et les fonctions, pour en nommer quelques-uns.

Troisièmement, un modèle relationnel sous la forme d'une série de tables[2] est présenté. Ce schéma relationnel est dérivé du schéma conceptuel. Ces tables sont groupées selon les niveaux présentés dans la première partie. Certains groupes de tables ne font pas directement partie des niveaux provenant des fichiers, mais contiennent plutôt des informations nécessaires à la gestion de l'outil DATRIX™. Outil pour lequel ce schéma relationnel est destiné.

Finalement, une étude sur les systèmes de gestion de base de données et de l'espace physique nécessaire à la conservation des tables est présenté. Cette section permettra d'envisager la construction de cette base de données.

---

2 Tout au long de ce document, le mot table fera référence à une table du schéma relationnel.

# Niveaux d'abstraction

L'évaluation du logiciel, par niveau d'abstraction, vise à classifier les différentes catégories d'attributs provenant des fichiers sources d'un produit informatique.

Pour réaliser l'analyse du code source, il faut effectuer les mêmes opérations préliminaires que pour la compilation. Parmi ces opérations, il y a l'analyse lexicale, l'analyse syntaxique et l'analyse sémantique. La distinction entre les deux processus s'effectue au niveau de l'utilisation des attributs retrouvés. Le but de la compilation est la traduction d'un programme source à un programme cible [AHO86]. Le cas généralement utilisé est la traduction d'un programme dans un langage de troisième génération à un programme exécutable[3]. Les compilateurs, au cours de cette traduction, négligent de transmettre au programme exécutable les attributs du programme source qui ne sont plus utiles. Par contre, la mesure du logiciel vise à évaluer le contenu du code source, c'est-à-dire tous les attributs mesurables. De plus, les compilateurs utilisent des fonctions d'optimisation qui déplacent certains énoncés sans modifier le résultat de l'exécution. L'analyse du code source doit permettre de conserver les identificateurs utilisés, l'ordre de l'utilisation ainsi que d'autres caractéristiques négligées par la compilation. Ceci est nécessaire, car le but de l'analyse est de représenter et d'évaluer le code tel qu'il est écrit et non comme il aurait dû l'être.

Par exemple, une fois l'espace mémoire allouée à une variable, la compilation n'a pas à tenir compte du nom des identificateurs utilisés dans les fichiers sources alors que l'analyseur de code doit pouvoir représenter l'information à partir du nom de ces

---

[3] Généralement le terme module objet est utilisé pour désigner le résultat de la compilation. Par la suite ces modules objets sont liés pour former un programme exécutable. Le terme exécutable est utilisé au lieu de module objet afin d'éviter la confusion avec les langages et la programmation par objet.

identificateurs en relation avec les fichiers du programme. La principale différence entre la compilation et l'analyse du code est la conservation des liens entre les entités provenant du code et leurs positions physiques dans les fichiers.

Les niveaux d'abstractions doivent être suffisamment divisés pour permettre de ne pas perdre de transformations tout en restant assez général pour être indépendants des langages de programmation analysés. Les entités d'un niveau ont une dépendance avec les entités du niveau précédent.

Le diagramme de la figure 1 permet de voir les différents niveaux définis. Les niveaux sont inscrits dans les boîtes, et les méthodes pour extraire ces entités sont inscrites dans les ellipses.

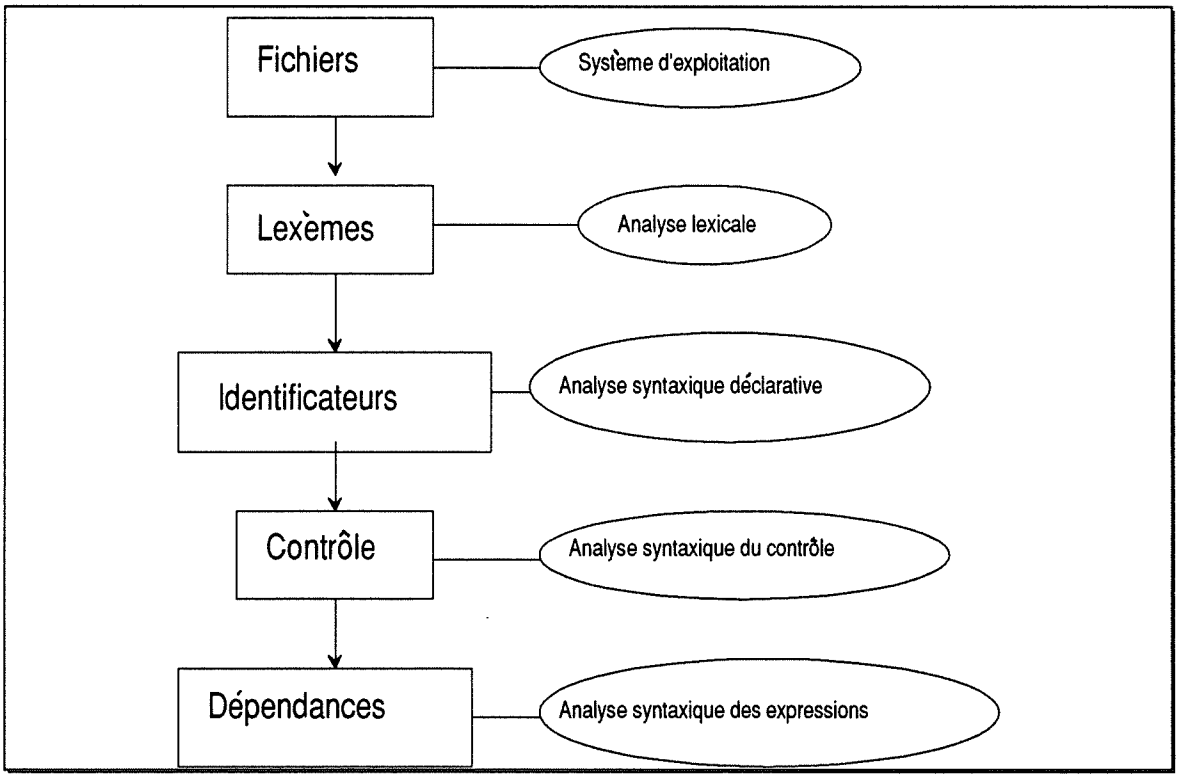


Figure 1 - *Diagramme des niveaux d'abstractions d'un logiciel*

### Niveau fichiers

Le premier niveau d'entité est composé des fichiers formant le projet à analyser. À ce niveau, seulement les attributs des fichiers et leurs relations sont concernés. Un attribut possible sont la dimensions en nombre de lignes ou en nombre de caractères, par exemple. Une liste exhaustive des attributs mesurables à chaque niveau d'abstraction est présentée avec le schéma relationnel à la section schéma relationnelle.

### Niveau lexèmes

Le support physique de l'analyse étant bien défini par le niveau fichier, la prochaine étape est l'analyse lexicale. Ce niveau conserve les attributs des lexèmes[4]. Une mesure bien connue de ce niveau est le décompte du nombre d'opérateurs et d'opérandes distincts et totaux de chaque fichier [HALS77]. Il est à noter que ces entités peuvent être classifiées par fichiers car celles-ci sont identifiées par les niveaux précédents. Grâce au fait que l'analyse des fichiers ne s'effectue pas par niveau mais bien concurremment, il sera possible de conserver l'information par routine. Par contre, la classification d'entités du niveau lexèmes en fonction d'un niveau supérieur pose plusieurs problèmes. Par exemple, le choix de classifier les lexèmes par routines pose un problème avec les lexèmes qui n'appartiennent à aucune routine. La seule caractéristique commune à tous les lexèmes est d'appartenir à un fichier.

### Niveau identificateurs

Les entités reconnues à partir des règles grammaticales sont de trois types. Il y a les entités relatives aux déclarations, celles relatives au contrôle et celles relatives aux

---

4 Un lexème est une suite de caractères normalement entourée de caractères espace. Il s'agit des mots reconnaissables par les compilateurs.

expressions. La raison pour laquelle les trois types d'entités sont dissociés est qu'il est plus avantageux de regrouper les entités de même nature pour en étudier les relations (intrarelations). Les relations entre les entités de différents types (interrelations) peuvent par la suite être étudiées deux à deux.

Les entités concernant les déclarations doivent conserver l'information relative à chaque identificateur, qu'il soit déclaré explicitement[5] ou non. Ce niveau est composé des déclarations des variables globales ou locales, des routines, des fonctions, ainsi que des types simples et construits. Le terme identificateur regroupe toutes les entités précédemment énoncées. Les attributs conservés sur les entités représentent leurs propriétés et dépendances hiérarchiques[6].

**Niveau contrôle**

Certains identificateurs du niveau précédent ont la propriété d'être des routines[7]. Pour ces routines, le flux d'exécutions peut être représenté par un graphe de contrôle. Isolées, les entités relatives au contrôle donnent un type d'information. Par exemple, le nombre de conditionnelles et la complexité cyclomatique de McCabe [MCCA76]se situent à ce niveau. Aucune connaissance de l'organisation des entités contrôle n'est nécessaire.

L'agencement des entités de contrôle mène au graphe de contrôle. Ce dernier permet d'obtenir des caractéristiques relatives à l'organisation de l'exécution. Par exemple, la

---

5 Certains langages, tel le FORTRAN, permettent l'utilisation d'une variable sans la déclarer explicitement. Il y a aussi l'utilisation des routines sans qu'elles soient au préalable déclarées à l'aide d'un prototype.

6 Par exemple, il y a hiérarchie en PASCAL pour les procédures déclarées dans des procédures, pour les types construits et pour les variables locales dans une procédure.

7 Le terme routine est utilisé pour définir les routines, les fonctions et les procédures dans les langages spécifiques.

métrique du nombre de chemins se situe à ce niveau. Au niveau contrôle, l'entité principale est le segment. Au niveau précédent, l'entité principale est l'identificateur. Il y a un ensemble de segments appartenant à chaque identificateur possédant l'attribut routine. D'où la notion de niveau et de dépendance de cette classification d'entités.

Dans les segments du graphe de contrôle d'une routine, il y a des déclarations de variables locales[8]. Normalement ces variables locales devraient faire l'objet d'un niveau supérieur au niveau segment car elles dépendent des segments pour la classification de leur appartenance. Par contre, ce niveau de variables locales serait la copie exacte du niveau identificateur. Pour cette considération, les variables locales déclarées dans les segments du graphe de contrôle d'une routine sont plutôt inscrites dans le niveau identificateur en gardant le segment de leur déclaration.[9]

**Niveau dépendances**

Les entités relatives aux expressions sont générées par les énoncés utilisant des identificateurs ou des constantes. Sur les véhicules d'information[10] trois opérations peuvent être appliquées. Premièrement, ces entités sont déclarées; les attributs relatifs à leurs déclarations sont conservés au niveau identificateur. La seconde opération est l'affectation; au cours d'une affectation, une valeur est inscrite dans le véhicule. La troisième opération est l'utilisation de la valeur inscrite dans un véhicule. Le niveau

---

8 Présentement, les seuls langages étudiés comportant ces caractéristiques sont les langages C et C++. Pour le langage C, il est possible de déclarer des variables à tous les débuts de segment. Pour le langage C++, les déclarations peuvent avoir lieu dans les segments.

9 Une variable locale a une portée qui est conservée dans l'association qui effectue le lien entre une variable locale et le segment qui la déclare.

10 Un véhicule d'informations est une entité du niveau identificateur ayant l'attribut fonction ou variable. Il s'agit des entités pouvant contenir une information pour une période de temps au cours de l'exécution du programme.

dépendance conserve ces affectations et utilisations entre les identificateurs du niveau identificateur. Tous ces transferts d'informations surviennent dans les segments du niveau précédent.

Plusieurs cas spéciaux se posent pour les transferts d'informations entre les variables et les fonctions. Ces cas seront expliqués à l'aide d'exemples lors de la présentation du schéma relationnel à la section traitant du niveau dépendance.

Ces entités concernent l'aspect général des expressions. Elles peuvent être caractérisées, entre autre, par une évaluation de la complexité de chacune des expressions, qu'elle soit conditionnelle ou non. La façon de procéder sera de morceler les relations plus complexes en relations élémentaires, basées sur le principe de compilation connu sous le nom de 'three-address code'.[AHO86] Les dépendances étudiées sont de trois types. Le premier est communément appelé le graphe d'appel et il décrit la hiérarchie des appels dans un système. Le seconde type est la dépendance fonctionelle qui décrit les interactions entre les fonctions et les variables. Dans une dépendance fonctionnelle, il y a toujours une fonction et une variable d'impliquée. Le troisième type de dépendance comporte deux variables.[WILD88]

# Schéma conceptuel

Afin de supporter les regroupements d'informations présentés à la section 2, ainsi que certains aspects du modèle DATGRAPH, un schéma conceptuel et un schéma physique sont définis. Dans un premier temps, un schéma d'entité-association a servi de schéma conceptuel, pour être traduit par la suite en schéma relationnel, en vue d'une implantation dans un système de gestion de base de données relationnelles. Le schéma d'entité-association est utile pour résumer l'information du modèle au niveau du design [ULLM88]. Cette méthode est très répandue pour la création des modèles de données. Dans un diagramme d'entité-association, il y a trois éléments graphiques: le carré qui est une entité, le losange qui est une association entre plusieurs entités et l'ellipse qui est un attribut d'une entité. Les ellipses pointillées indiquent que l'attribut est élément d'une clé. Les noms des divers éléments graphiques sont écrits au centre de ceux-ci. Les attributs attachés à une association sont aussi représentés par une ellipse attachée à un losange. Cet abus de notation est utilisé afin d'alléger le schéma. Ces attributs représentent des entités ayant un seul attribut et étant attaché à une seule association.

La figure 2 présente le résultat final du modèle entité-association. Les entités et les associations sont groupées en fonction des niveaux d'abstractions présentés à la section 2. Deux groupes s'ajoutent aux groupes définis à la section 2. Le groupe SYMBOLE permet de réduire l'ampleur de la base de données. Le groupe MÉTRIQUE permet de conserver les résultats de certaines mesures dont les éléments de base ne sont plus disponibles une fois l'analyse terminée.

Généralement chaque entité et chaque association engendre une table du modèle relationnel. Les détails concernant les entités, les associations et les attributs sont expliqués à la section 4 avec la définition des tables du schéma relationnel.

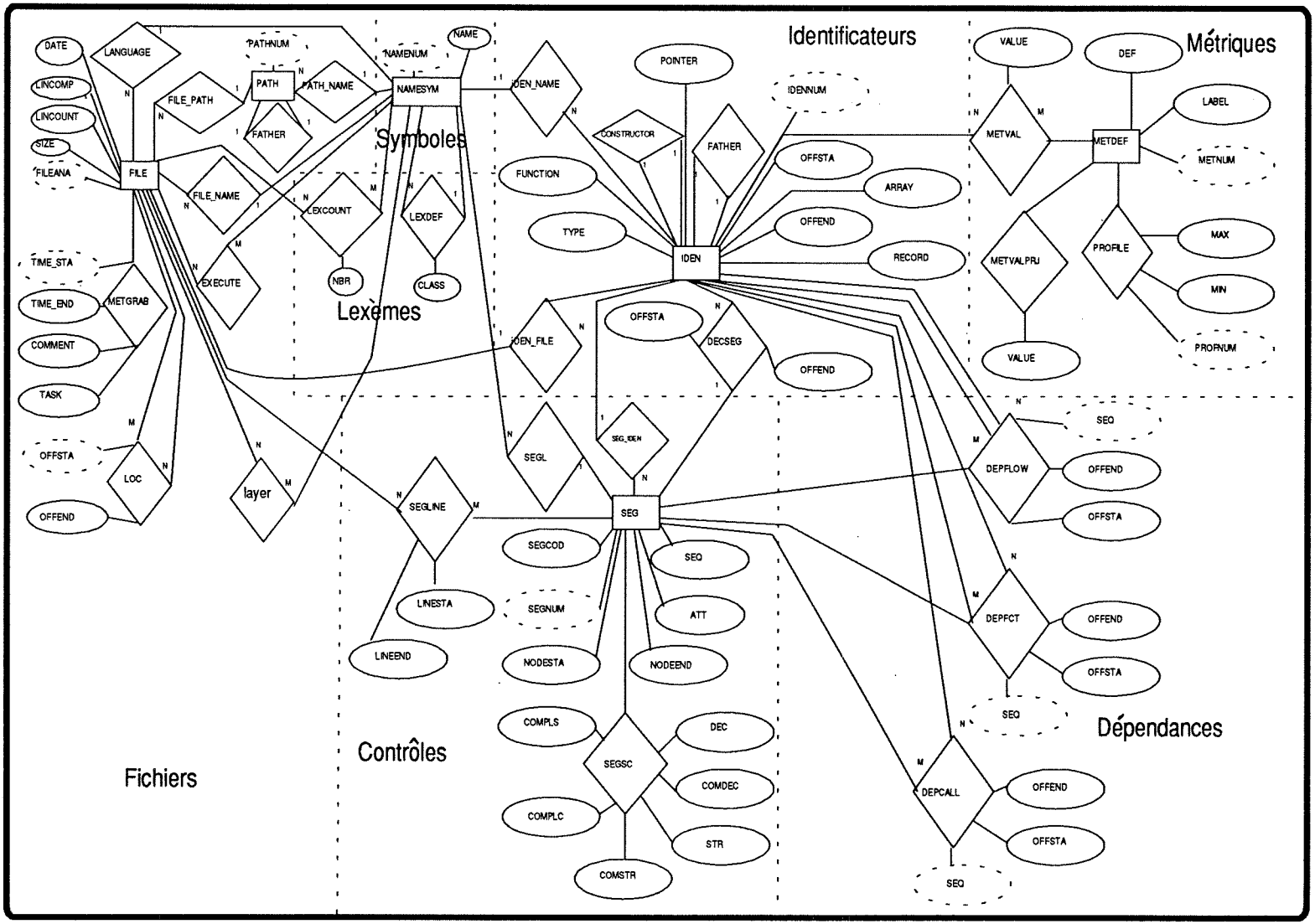


Figure 2 - *Diagramme du schéma entité-association*

# Schéma relationnel

Le choix d'utiliser une représentation relationnelle comme support à DATGRAPH est justifié par les éléments suivants: l'utilisation d'une base de données relationnelles est très flexible et facilite les requêtes. Tout ajout de métrique, de langage ou même d'une nouvelle couche d'informations est relativement simple. Cette méthode permet d'orienter la structure d'informations en fonction des associations entre les entités présentes dans le logiciel. Les bases de données relationnelles sont bien connues et un grand nombre d'outils est maintenant disponible sur le marché.

La description du schéma relationnel soutenant les différents niveaux d'abstractions sera présentée sous forme de tables relationnelles à l'aide d'un exemple. Dans un premier temps, l'exemple est présenté, ensuite les tables sont décrites dans l'ordre, suivant les niveaux d'abstractions. Le schéma relationnel est composé de vingt-deux tables classifiées selon sept groupes. Les groupes sont les suivants: symbole, fichier, lexème, identificateur, contrôle, dépendance, métrique. Ces classes se veulent le plus près possible de la définition des niveaux d'abstractions, mais le premier et le dernier servent de support à l'outil DATRIX™.

### Description du projet utilisé comme exemple

Afin de présenter les tables relationnelles qui sont le support du modèle DATGRAPH, un petit projet sera utilisé. Ce projet est composé de quatre fichiers sources. Deux de ces fichiers sont des fichiers inclus et ne contiennent que des déclarations. Le premier fichier contenant des énoncés exécutables, présenté à la figure 3, est inspiré de *Compilers Principles, Techniques, and Tools* page 436.[AHO86]

```
struct tree {          int left;
                       int right;      }

struct tree head;

int hashjw(s)
char *s;
{       char *p
        unsigned h =0, g;
        for( p = s; *p != '\0', p = p + 1 ) {
                h = ( h<<4) + (*p);
                if( g=h&0xf0000000) {
                        h = h ^ ( g>>24);
                        h = h ^ g;      }}
        return h % 211;
}
```

Figure 3 - *Code source hashjw.c*

Le fichier, présenté à la figure 4, est un exemple de l'utilisation d'une structure avec un pointeur.

```
#include <stdio.h>
#include <math.h>

void routine()
{

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

struct str1 * var1;
struct str1 * var2;

...[11]

var1.j = var2.suiv.j;
var2.j = var2.suiv.j;

}
```

Figure 4 - *Code source routine.c*

---

11 Certains énoncés de ce programme sont omis car ils ne servent pas pour l'exemple.

Le schéma relationnel proposé supporte l'environnement de développement du projet et les fichiers sources n'ont pas à être déplacés. La disposition des fichiers sources de l'exemple sur le disque est présentée à la figure 5.

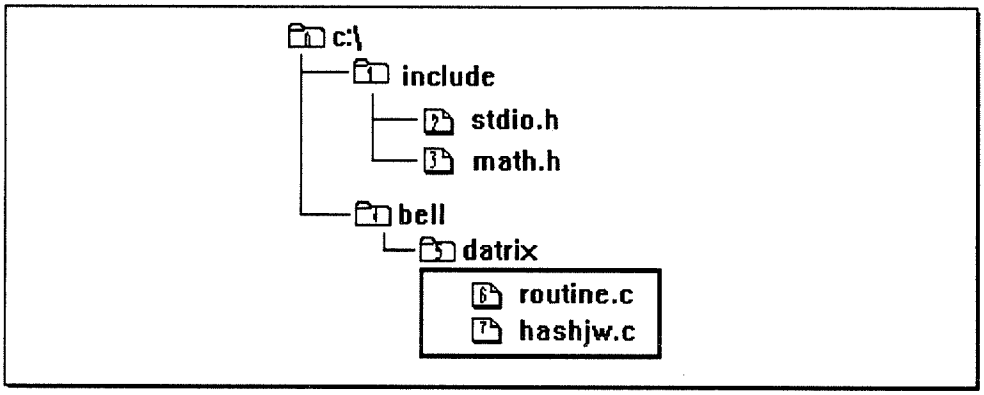


Figure 5 - *Disposition des sources sur le disque*

### Nomenclature de la description des tables relationnelles

Chaque table sera présentée selon la nomenclature suivante:

TABLE(ATTRIBUT:TYPE, ATTRIBUT:TYPE...).

Le mot TABLE est remplacé par le nom de la table. Le mot ATTRIBUT est remplacé par un nom de champ. Le mot TYPE est remplacé par le type prédéfini[12] de l'attribut.

Le nom des champs est choisi en fonction de l'information qu'il représente; il est unique pour l'ensemble des tables. Si un nom de champs se retrouve dans plus d'une table, cela signifie que ce champ est un élément de liaison entre ces tables relationnelles décrites.Le type des champs est limité aux choix décrit au tableau 1.

---

12 Dans les systèmes de gestion de base de données relationnelles, seuls les types prédéfinis dans le système sont disponibles. Les types complexes pouvant être définis par un usagé ne sont pas supportés.

| NOM | DIMENSION | DESCRIPTION |
|---|---|---|
| NUMBER | 4 octets | Ce type décrit des nombres entiers positifs. |
| STRING | | La longueur des champs STRING est défini pour chaque table. Cette longueur est définie à l'initialisation de la base de donnée et sera dans la plupart des cas dépendante du système d'exploitation. |
| DATE | 4 octets | Le type DATE comporte l'heure et la date. L'encodage utilisé est normalement un index du nombre de secondes écoulées depuis une date fixe. Par exemple, le 1er janvier 1980. |
| BOOLEAN | 1 octet | Ce type indique une valeur de vérité, vraie ou fausse. Le type d'encodage utilisé n'a pas d'importance pour la description des tables. |
| VALUE | 4 octets | Ce type décrit des nombres réels. |

Tableau 1 - *Type des domaines permis pour les attributs*

De plus, pour chaque table du schéma relationnel, les éléments du schéma d'entité-association qui ont servi à la définir sont présentés. Après une brève description de la fonctionnalité de la table, un tableau présente les liens entre cette table et les autres tables ainsi qu'une description de ses champs. Cette description comporte quelquefois des contraintes sur les valeurs du champ décrit.

**Table NAMSYM**

NAMSYM(NAMNUM:NUMBER, NAME:STRING32[13])

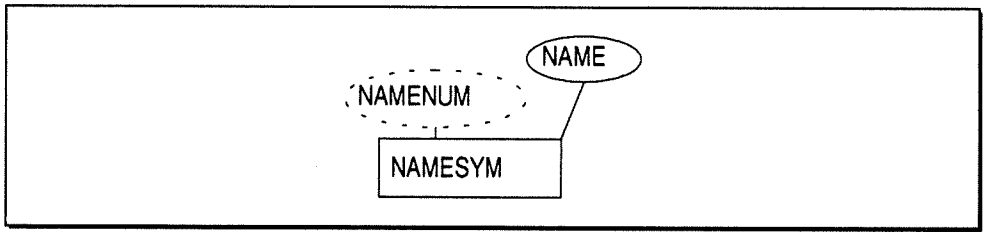


Figure 6 - *Entité NAMESYM*

13 La largeur maximum d'un nom est fixé à 32 à titre d'exemple. Ce paramètre dépend du langage et du compilateur propre à chaque projet. La valeur 32 devrait couvrir la plupart des situations.

Cette table convertit les noms en numéros uniques. Cette transformation est utile pour les autres tables qui fonctionnent uniquement avec les représentations numériques des identificateurs. Ceci évite la duplication des chaînes de caractères dans la base de données. Cette table possède un certain nombre de fiches fixes par défauts et les autres proviennent de l'analyse des fichiers sources. Une fiche fixe est un enregistrement qui est permanent dans la table et qui ne dépend pas du projet analysé. Les fiches fixes, sont en général, les mots réservés des différents langages[14] supportés. Pour l'implantation de ce schéma, des tables permettant l'initialisation d'un projet devront être prévues. Les détails de cette initialisation ne seront pas discutés ici, ceci relevant plus d'une spécification de l'implantation que de la fonctionnalité de la table de symboles. Cette table est la seule du groupe SYMBOLE.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| NAMNUM | NAMESYM | Numéro unique représentant la chaîne de caractères du champ NAME. |
| NAME | NAMESYM | Nom devant être converti en caractères. |

Tableau 2 - *Description des attributs de la table NAMSYM*

Pour cet exemple, tous les identificateurs, ainsi que certains opérateurs du langage C, des deux fichiers sources des figure 3 et 4, sont utilisés. Normalement, les identificateurs des fichiers inclus seraient aussi inscrits dans cette table, mais pour limiter l'exemple, ils sont exclus. Il est à noter que l'analyseur C fonctionne à partir d'un fichier préprocessé et que toutes les commandes du préprocesseur sont transparentes. Les commandes de préprocesseur sont de plus en plus abandonnées, car ces commandes cachent de

---

14 Le mot langage est toujours utilisé pour décrire les langages de programmation (C, FORTRAN, PASCAL...) à moins d'avis contraire explicite.

l'information au compilateur ce qui empêche la détection de certaines anomalies lors de l'analyse syntaxique [STRO87]. Il y a une exception à cette règle en ce qui concerne le nom des fichiers inclus, qui, eux, sont inscrits dans la table. Les figures 3 et 4 sont reprises pour former les figures 7 et 8 en soulignant les éléments intégrés à la table NAMSYM. Les éléments sont sélectionnés en simulant l'analyse successive des fichiers des figures 2 et 3.

```
struct tree {
                        int left;
                        int right;
                };

struct tree head;

int hashjw(s)
char *s;
{
        char *p
        unsigned h =0, g;
        for( p = s; *p != '\0', p = p + 1 ) {
                h = ( h<<4) + (*p);
                if( g=h&0xf0000000) {
                        h = h ^ ( g>>24);
                        h = h ^ g;
                }
        }
        return h % 211;
}
```

Figure 7 - *Code source hashjw.c avec les éléments sélectionnés*

```
#include <stdio.h>
#include <math.h>

void routine()
{

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

struct str1 * var1;
struct str1 * var2;

var1.j = var2.suiv.j;
var2.j = var2.suiv.j;

}
```

Figure 8 - *Code source routine.c avec les éléments sélectionnés*

En plus des mots provenant du code source, les noms des répertoires et des fichiers sont inscrits dans cette table, ainsi que le nom du langage de l'analyseur. Pour l'exemple l'analyseur ANSI_C est inscrit.

| NAMNUM | NAME |
|---|---|
| 0 | C: |
| 1 | #include |
| 2 | stdio.h |
| 3 | math.h |
| 4 | bell |
| 5 | datrix |
| 6 | routine.c |
| 7 | haswjw.c |
| 8 | struct |
| 9 | tree |
| 10 | { |
| 11 | int |
| 12 | left |
| 13 | , |
| 14 | right |
| 15 | } |
| 16 | head |

| NAMNUM | NAME |
|---|---|
| 17 | hashjw |
| 18 | ( |
| 19 | s |
| 20 | ) |
| 21 | char |
| 22 | * |
| 23 | p |
| 24 | unsigned |
| 25 | h |
| 26 | = |
| 27 | g |
| 28 | for |
| 29 | != |
| 30 | + |
| 31 | << |
| 32 | if |
| 33 | & |

| NAMNUM | NAME |
|---|---|
| 34 | ^ |
| 35 | >> |
| 36 | return |
| 37 | % |
| 38 | void |
| 39 | routine |
| 40 | str1 |
| 41 | i |
| 42 | float |
| 43 | j |
| 44 | suiv |
| 45 | var1 |
| 46 | var2 |
| 47 | . |
| 48 | ansi_c |
| 49 | ; |
| | |

Tableau 3 - *Exemple de la table NAMSYM*

**Table PATH**

PATH( PATHNUM:NUMBER, NAMENUM:NUMBER, FATHER:NUMBER)

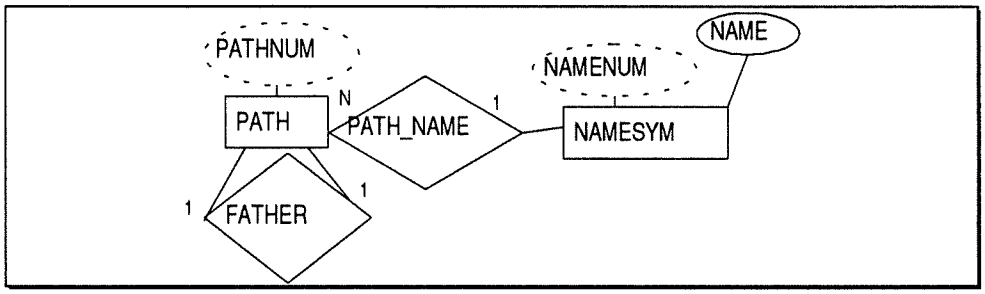


Figure 9 - *Association PATH*

Un logiciel est composé d'une multitude de fichiers sources. Ces fichiers sont disposés dans une arborescence de répertoires. La table PATH conserve l'arborescence des répertoires du système où se trouvent les fichiers. Cette table évitera tout déplacement des fichiers et l'évaluation d'un projet se fera dans son environnement de développement. De plus, des métriques pourront être créées sur l'environnement de développement du projet. Par exemple, la profondeur de l'arborescence des répertoires, le nombre de

répertoires et la quantité de fichiers contenus. Cette table numérote tous les répertoires du projet de façon unique. Cette table appartient au groupe FICHIER.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| PATHNUM | NUMBER | Numéro unique représentant un répertoire. |
| NAMENUM | NAMESYM:NAMENUM via PATH_NAME | Numéro représentant le nom du répertoire. |
| FATHER | PATH:PATHNUM | Répertoire père du répertoire PATHNUM. Si le père d'un répertoire est la racine du système de fichier, le champs FATHER aura la valeur 0. |

Tableau 4 - *Description des attributs de la table PATH*

Ce qui suit est un exemple d'utilisation de la table PATH avec les quatre fichiers de la figure 10. Dans le but de faciliter la compréhension des ces exemples, lorsqu'un champ de type NUMBER est utilisé pour représenter un nom, le nom en chaîne de caractères sera utilisé en remplacement du chiffre. Par contre, dans l'implantation, seule la table symbole contiendra des chaînes de caractères. Les nombres dans les boîtes sont les numéros des répertoires.

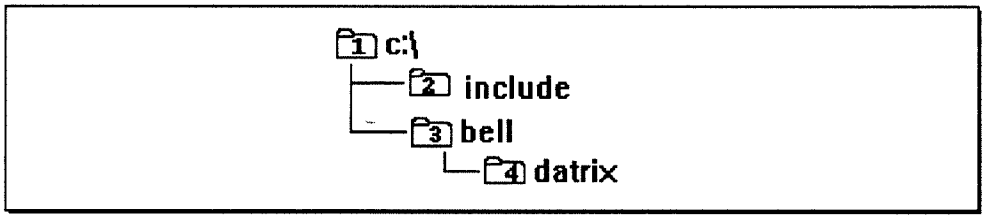


Figure 10 - *Disposition des sources sur le disque avec la table PATH*

| PATHNUM | FATHER | NAMENUM |
|---|---|---|
| 1 | 0 | C:\ |
| 2 | 1 | include |
| 3 | 1 | bell |
| 4 | 3 | datrix |

Tableau 5 - *Exemple de la table PATH*

**Table FILE**

FILE(FILANA: NUMBER, PATHNUM:NUMBER, NAMENUM:NUMBER, LANGUAGE:NUMBER, DATE:DATE, SIZE:NUMBER, LINCOUNT:NUMBER, LINCOMP:NUMBER)

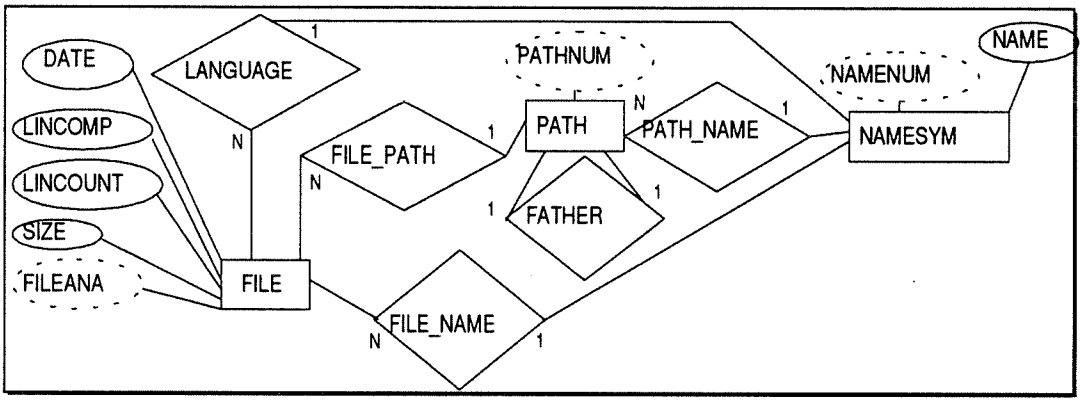


Figure 11 - *Entité FILE*

Cette table conserve les attributs et propriétés des fichiers du projet. Tous les champs de la fiche[15] ne sont pas évalués au même moment au cours de l'analyse d'un fichier. Cette table contient tous les fichiers même ceux qui ne sont vus que par l'entremise d'inclusions dans un autre fichier.

15 Une fiche est une ligne d'une table relationnelle.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| FILANA | FILE | Numéro unique représentant le fichier. |
| PATHNUM | PATH:PATHNUM | Numéro du répertoire contenant le fichier. |
| NAMENUM | NAMSYM:NAMENUM via FILE_NAME | Numéro représentant le nom du fichier. |
| LANGUAGE | NAMSYM:NAMENUM | Numéro de la table NAMESYM indiquant le langage utilisé. Il est à noter qu'un projet peut avoir plus d'un langage utilisé. |
| DATE | FILE | Ce champ conserve l'heure et la date de l'analyse du fichier. |
| SIZE | FILE | Nombre de caractères dans le fichier. |
| LINCOUNT | FILE | Nombre de lignes dans le fichier |
| LINCOMP | FILE | Nombre de lignes vu par le compilateur. Ce nombre est la somme des lignes du fichier ainsi que de tous les fichiers inclus. Dans le cas des fichiers inclus, ce nombre est 0 car les fichiers inclus ne sont pas directement compilés. |

Tableau 6 - *Description des attributs de la table FILE*

Pour l'exemple, le champ date n'est pas encodé, mais dans l'implantation, le type DATE sera utilisé. Cette table appartient au groupe FICHIER.

| FILANA | PATHNUM | NAMENUM | LANGUAGE | DATE | SIZE | LINCOUNT | LINCOMP |
|---|---|---|---|---|---|---|---|
| 1 | 2 | stdio.h | ANSI_C | 90-07-20 16:30 | 1837 | 73 | 0 |
| 2 | 2 | math.h | ANSI_C | 90-07-23 22:30 | 7443 | 210 | 0 |
| 3 | 4 | routine.c | ANSI_C | 90-07-30 9:20 | 232 | 17 | 73+210 +17-2 =298 |
| 4 | 4 | hashjw.c | ANSI_C | 90-07-30 9:20 | 349 | 24 | 24 |

Tableau 7 - *Exemple de la table FILE*

Le compte du nombre de lignes compilé LINCOMP du fichier routine.c s'effectue par l'addition du nombre de lignes dans les fichiers inclus, plus le nombre de lignes dans le fichier, moins deux qui est le nombre d'énoncés **#include.**

**Table EXECUTE**

EXECUTE( FILANA:NUMBER, EXCUTABLE:NUMBER)

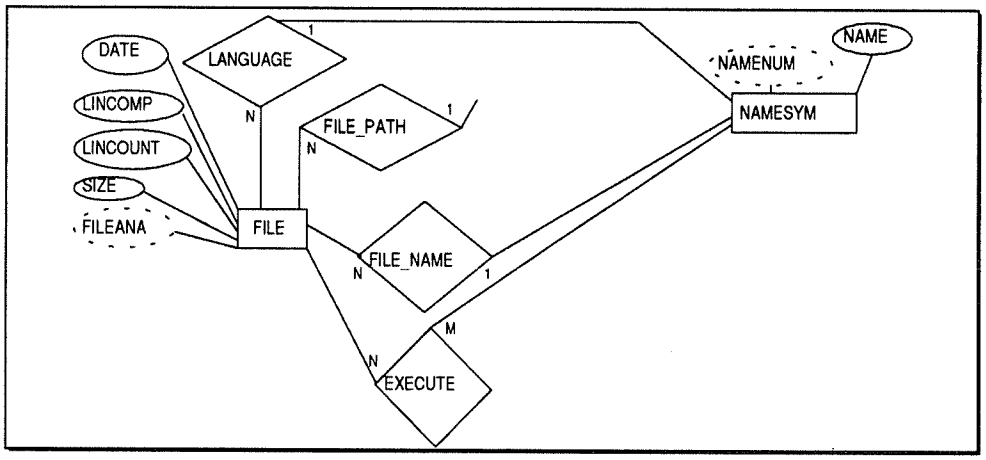


Figure 12 - *Association EXECUTE*

Cette table conserve les regroupements des fichiers en exécutables. Elle est équivalente aux listes fournies à un éditeur de liens pour la production des fichiers exécutables. L'information requise pour cette table ne peut être extraite des fichiers sources ou de l'arborescence des répertoires. Cette liste devra être fournie par l'entremise du directeur de projet ou de la documentation du design des modules exécutables. Cette table fait partie du groupe FICHIER.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| FILANA | FILE:FILANA | Numéro du fichier membre d'un exécutable. |
| EXECUTABLE | NAMESYM:NAMENUM | Numéro indiquant le nom d'un fichier exécutable. |

Tableau 8 - *Description des attributs de la table EXECUTE*

**Table METGRAB**

METGRAB( FILANA:NUMBER, TIME_STA:DATE, TIME_END:DATE, TASK:NUMBER, COMMENT:STRING80)

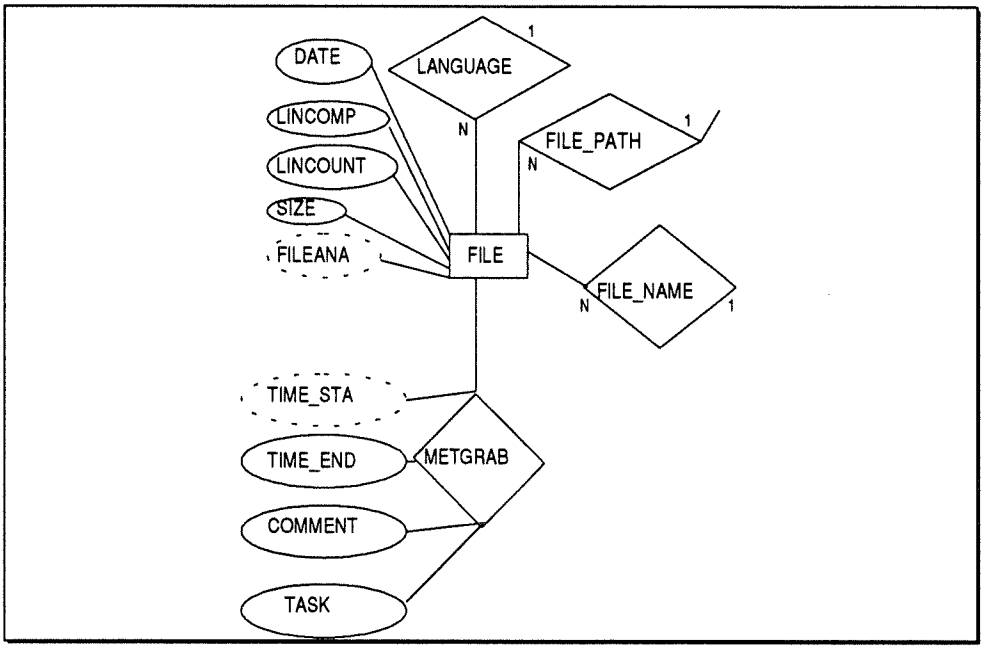


Figure 13 - *Association METGRAB*

Cette table permet de conserver l'effort de programmation investi dans un fichier et elle est la seule du schéma qui conserve une trace du processus de création du projet informatique. Cette table conserve le nombre de modifications à un fichier, la durée de ces modifications ainsi que la raison fournie par le programmeur. Le champ TASK permet à un groupe de développement comportant plusieurs programmeurs de définir des tâches projets par individu et ainsi vérifier l'impact sur le code source. Cette table fait partie du groupe FICHIER.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| FILANA | FILE:FILANA | Numéro du fichier modifié. |
| TIME_STA | METGRAB | Ce champ conserve l'heure et la date du début de la modification du fichier. L'encodage utilisé est normalement un index du nombre de secondes écoulées depuis une date fixe. Par exemple, le 1er janvier 1980. |
| TIME_END | METGRAB | Ce champ conserve l'heure et la date de la fin de la modification du fichier. |
| TASK | METGRAB | Nom de la tâche durant laquelle le fichier FILANA a été modifié. |
| COMMENT | METGRAB | Le commentaire fournit par le programmeur quant à la modification effectuée. |

Tableau 9 - *Description des attributs de la table METGRAB*

**Table LOC**

LOC( FILANA:NUMBER, OFFSTA:NUMBER, OFFEND:NUMBER, FILEINC:NUMBER )

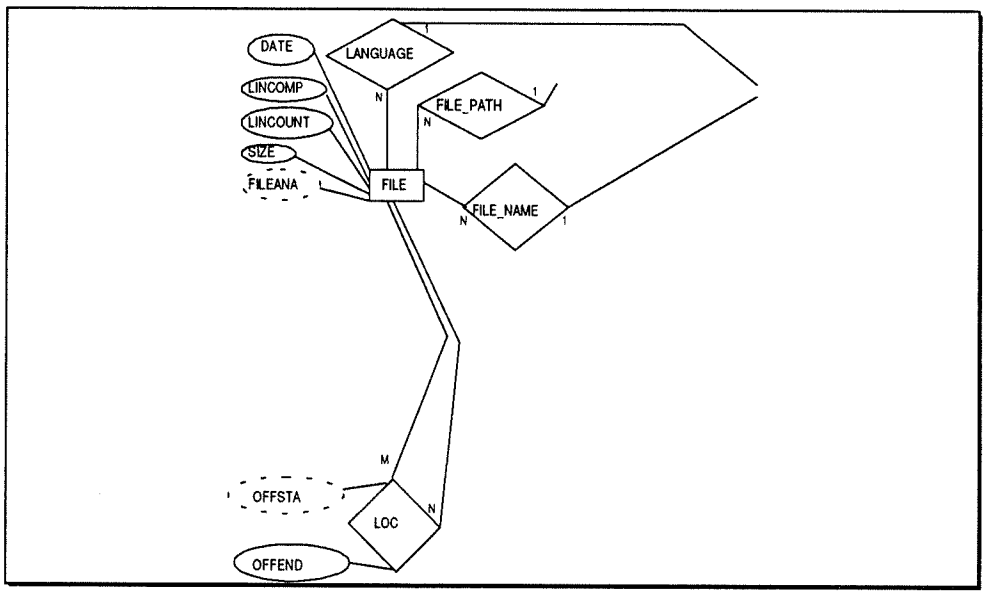


Figure 14 - *Association LOC*

Cette table est la dernière du groupe des tables qui traitent les fichiers. Cette table conserve la hiérarchie des inclusions de fichiers vue au cours de l'analyse des fichiers. Tous les éléments extraits d'un fichier sont identifiés par une clé unique qui permettra de

retracer leur provenance. L'usager pourra ainsi être mis en contact avec le fichier d'où provient l'élément qu'il analyse.

Le premier champ FILANA est le numéro du fichier qui a été analysé et provient de la table FILE. Par exemple, si le fichier A inclus le fichier B, le numéro du fichier analysé sera le numéro correspondant au fichier A. Les deuxième et troisième champs indiquent un bloc par des index du fichier de compilation. L'index utilisé est le nombre de caractères rencontrés depuis le début de la compilation d'un fichier en incluant les caractères compris dans les fichiers inclus. L'index est remis à zéro à tous les fichiers analysés. Le dernier champ indique le fichier où se trouve physiquement le bloc décrit par les index de compilations. Un bloc de caractères se retrouve toujours dans un fichier physique seulement. Ainsi l'analyse d'un fichier comportant des fichiers inclus est vue comme une série de blocs de caractères appartenant à différents fichiers.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| FILANA | FILE:FILANA | Numéro indiquant le fichier compilé |
| OFFSTA | LOC | Index de début du bloc |
| OFFEND | LOC | Index de fin du bloc |
| FILEINC | FILE:FILANA | Numéro indiquant le fichier contenant l'information. |

Tableau 10 - *Description des attributs de la table LOC*

Le fichier routine.c de la figure 4 est utilisé comme exemple car il possède des fichiers inclus..

| FILANA | OFFSETSTA | OFFSETEND | FILEINC |
|---|---|---|---|
| routine.c | 1 | 1837 | stdio.h |
| routine.c | 1838 | 1837+7443=9280 | math.h |
| routine.c | 9281 | 9280+232= 9512 | routine.c |

Tableau 11 - *Exemple de la table LOC*

**Table LAYER**

LAYER( FILANA:NUMBER, GROUP:NUMBER)

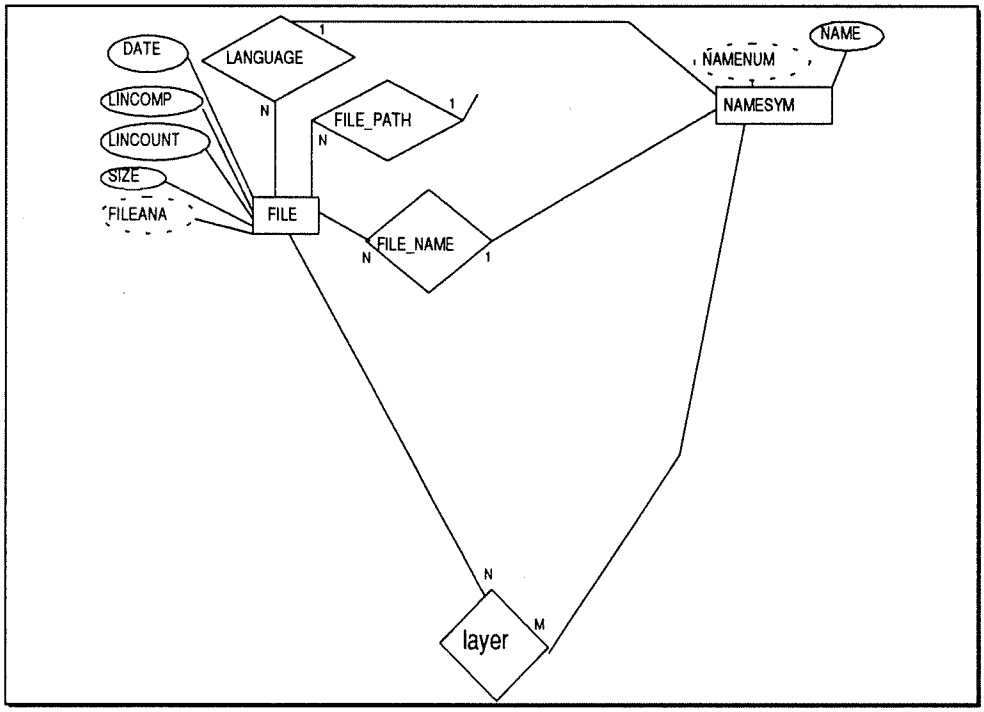


Figure 15 - *Association LAYER*

Cette table représente les regroupements des fichiers compilables en couches. La notion de couche est utilisée pour représenter les différents niveaux d'abstractions dans un système. Cette table permet le regroupement de fichiers selon une définition différente des fichiers sources et de l'arborescence des fichiers. La définition de couche devra provenir de la documentation du design ou du gestionnaire de projet. Ce regroupement permettra lors de l'analyse des résultats d'évaluer les différentes couches définies comme une seule et même entité.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| FILANA | FILE:FILANA | Numéro indiquant le fichier compilé. |
| GROUP | NAMESYM:NAMENUM | Nom du groupe de fichier défini comme étant une couche. |

Tableau 12 - *Description des attributs de la table LAYER*

**Table LEXCOUNT**

LEXCOUNT( LEXNAME:NUMBER, FILANA:NUMBER, NBR:NUMBER)

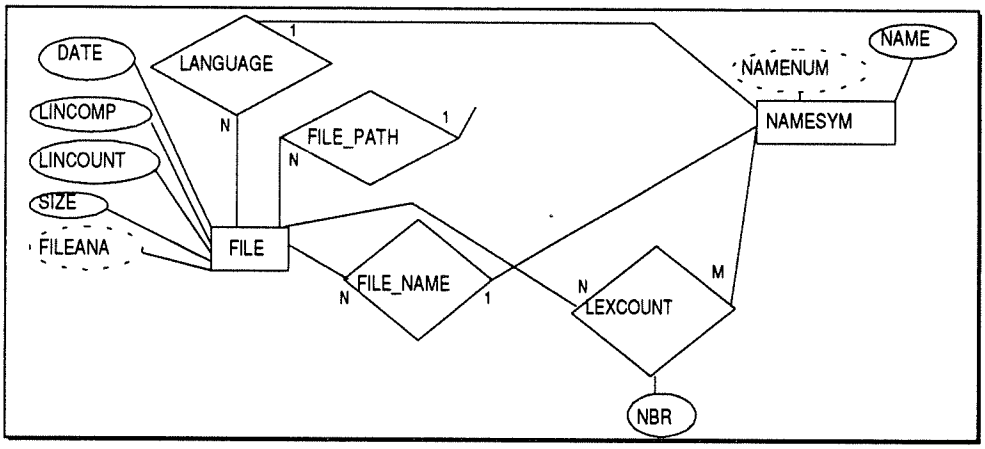


Figure 16 - *Association LEXCOUNT*

Cette table conserve l'occurrence des lexèmes par fichier. Tous les éléments lexicaux sont enregistrés dans la table NAMSYM et seulement leur représentation numérique est utilisée. La clé de cette table est la combinaison du lexème avec un fichier et l'information produite est le nombre de fois que le lexème est rencontré pour un fichier.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| LEXNAME | NAMESYM:NAMENUM | Numéro unique représentant le lexème rencontré. |
| FILEANA | FILE:FILANA | Numéro unique représentant le fichier où le lexème a été rencontré. |
| NBR | LEXCOUNT | Nombre d'occurrence du lexème. |

Tableau 13 - *Description des attributs de la table LEXCOUNT*

L'exemple de l'utilisation de cette table est construit à partir du fichier hashjw.c de la figure 3.

| LEXNAME | FILANA | NBR |
|---|---|---|
| struct | hashjw.c | 2 |
| tree | hashjw.c | 2 |
| { | hashjw.c | 4 |
| int | hashjw.c | 3 |
| left | hashjw.c | 1 |
| , | hashjw.c | 1 |
| right | hashjw.c | 1 |
| } | hashjw.c | 4 |
| head | hashjw.c | 1 |
| hashjw | hashjw.c | 1 |
| ( | hashjw.c | 6 |
| s | hashjw.c | 3 |
| ) | hashjw.c | 6 |
| char | hashjw.c | 2 |
| * | hashjw.c | 4 |
| p | hashjw.c | 6 |

| LEXNAME | FILANA | NBR |
|---|---|---|
| unsigned | hashjw.c | 1 |
| h | hashjw.c | 9 |
| = | hashjw.c | 7 |
| g | hashjw.c | 4 |
| for | hashjw.c | 1 |
| != | hashjw.c | 1 |
| + | hashjw.c | 2 |
| << | hashjw.c | 1 |
| if | hashjw.c | 1 |
| & | hashjw.c | 1 |
| ^ | hashjw.c | 2 |
| >> | hashjw.c | 1 |
| return | hashjw.c | 1 |
| % | hashjw.c | 1 |
| ; | hashjw.c | 10 |
| | | |

Tableau 14 - *Exemple de la table LEXCOUNT*

**Table LEXDEF**

LEXDEF( LANGUAGE: NUMBER, LEXNAME:NUMBER, CLASS:NUMBER)

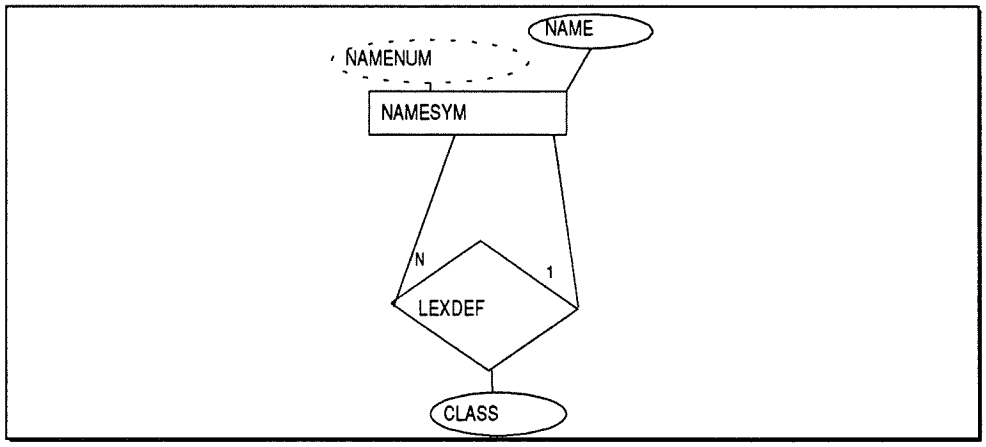


Figure 17 - *Association LEXDEF*

Cette table relationnelle permet la définition des classes de lexèmes afin d'évaluer les résultats de la table LEXCOUNT. Il est à noter que lorsque la table LEXCOUNT est utilisée

en combinaison avec la table LEXDEF, il peut arriver qu'un LEXNAME soit manquant dans la table LEXDEF. Pour tous les LEXNAME manquants, une classe par défaut est attribué, soit la classe 0. Ce cas particulier arrive avec les identificateurs dont la définition ne peut être connue d'avance.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| LANGUAGE | NAMESYM:NAMENUM | Numéro unique indiquant le langage utilisé. |
| LEXNAME | NAMESYM:NAMENUM | Numéro unique indiquant le lexème. |
| CLASS | LEXDEF | Numéro unique indiquant la classe d'appartenance. La science du logiciel de HALSTEAD a deux classes: les opérateurs et les opérandes. |

Tableau 15 - *Description des attributs de la table LEXDEF*

L'exemple de l'utilisation de cette table est construit à partir du fichier hashjw.c suivant la définition de [HALS77]. La classe 0 est la classe des opérandes et 1, la classe des opérateurs. Seuls les mots réservés du langage sont conservés dans cette table. Les variables et les noms de fonctions de l'exemple font partie de la classe exception, la classe 0.

| LANGUAGE | LEXNAME | CLASS |
|---|---|---|
| ANSI_C | struct | 1 |
| ANSI_C | { | 1 |
| ANSI_C | int | 1 |
| ANSI_C | , | 1 |
| ANSI_C | } | 1 |
| ANSI_C | ( | 1 |
| ANSI_C | ) | 1 |
| ANSI_C | char | 1 |
| ANSI_C | * | 1 |
| ANSI_C | unsigned | 1 |
| ANSI_C | = | 1 |
| ANSI_C | for | 1 |
| ANSI_C | != | 1 |
| ANSI_C | + | 1 |
| ANSI_C | << | 1 |
| ANSI_C | if | 1 |
| ANSI_C | & | 1 |
| ANSI_C | ^ | 1 |
| ANSI_C | >> | 1 |
| ANSI_C | return | 1 |
| ANSI_C | % | 1 |
| ANSI_C | ; | 1 |

Tableau 16 - *Exemple de la table LEXDEF*

**Table IDEN**

IDEN(IDENNUM: NUMBER, NAMENUM:NUMBER, FILANA:NUMBER, OFFSTA:NUMBER, OFFEND:NUMBER, ARRAY:BOOLEAN, RECORD:BOOLEAN, POINTER:BOOLEAN, FUNCTION:BOOLEAN, TYPE:BOOLEAN)

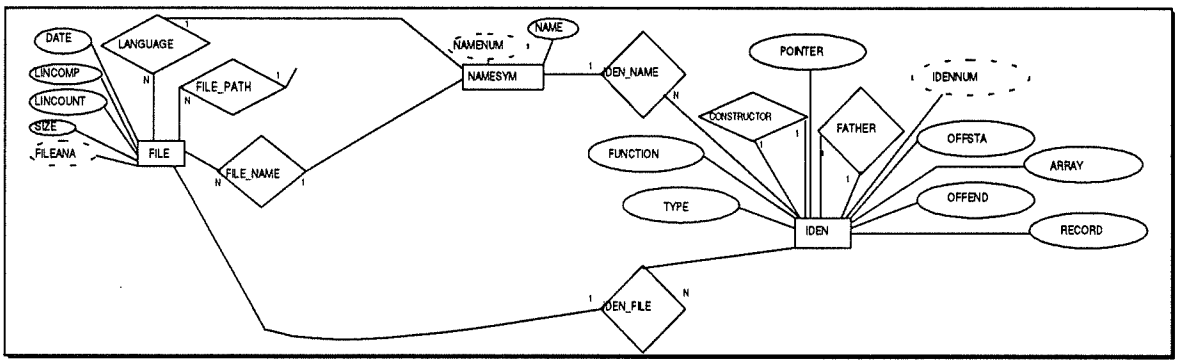


Figure 18 - *Entité IDEN*

Cette table conserve toutes les déclarations rencontrées dans le projet et les numérote de façon unique. Pour les fonctions, le bloc défini par les champs OFFSTA et OFFEND couvre

seulement la déclaration et non le corps de la fonction. On remarque que les relations IDEN_NAME, IDEN_FILE, FATHER et CONSTRUCTOR sont incluses dans la table IDEN. Ces quatre relations n'ayant aucun attribut propre, leurs champs sont ajoutés dans la table IDEN.

Cette table est le coeur des nouvelles métriques de flux de données. Tous les véhicules d'informations[16] ainsi que les types construits se retrouvent dans cette table.

Tous les identificateurs ont un père. Ce père est l'identificateur auquel un identificateur est accroché. Il s'agit ici des procédures internes en PASCAL, des membres d'un type construit, des variables locales dans une routine. Dans le cas d'un identificateur global leur père est la valeur 0, ceci est un cas spécial.

Tous les identificateurs ont un constructeur. Ce constructeur est l'identificateur qui a servi à les déclarer. Le cas où un identificateur n'a pas de constructeur la valeur du champ CONSTRUCTOR est 0. Ceci est un cas spécial. Par exemple, les types prédéfinis, comme **int** ou **float** en C, n'ont aucun constructeur.

---

16 Un véhicule d'information est une variable globale, une variable locale ou une fonction. Il s'agit des éléments pouvant prendre une valeur au cours de l'exécution d'un programme.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| IDENNUM | IDEN | Numéro unique représentant la déclaration. Il peut s'agir d'un type construit ou non, d'une variable ou d'une routine. |
| NAMENUM | NAMESYM:NAMENUM via IDEN_NAME | Numéro représentant le nom de l'identificateur. |
| FILANA | FILE:FILANA VIA IDEN_FILE | Fichier compilable d'origine. |
| OFFSTA | IDEN | Index de début. Ce numéro en combinaison avec la table loc permettra de retrouver la position exacte du début de la déclaration. |
| OFFEND | IDEN | Index de fin. Ce numéro en combinaison avec la table LOC permettra de retrouver la position exacte de la fin de la déclaration. |
| ARRAY | IDEN | La valeur vraie indique que l'identificateur IDENNUM a la propriété d'être un vecteur d'éléments. |
| RECORD | IDEN | La valeur vraie indique que l'identificateur IDENNUM a la propriété d'être composé. |
| POINTER | IDEN | La valeur vraie indique que l'identificateur IDENNUM a la propriété d'être un pointeur. |
| FUNCTION | IDEN | La valeur vraie indique que l'identificateur IDENNUM a la propriété d'être une routine. |
| TYPE | IDEN | La valeur vraie indique que l'identificateur IDENNUM a la propriété d'être un type au même sens que les type de base comme int ou float en C. |
| FATHER | IDEN:IDENNUM | Le IDENNUM de l'identificateur père de cet identificateur. |
| CONSTRUCTOR | IDEN:IDENNUM | Le IDENNUM de l'identificateur qui a servi à construire cet identificateur |

Tableau 17 - *Description des attributs de la table IDEN*

Cet exemple est tiré des identificateurs des fichiers sources des figures 3 et 4. La figure 19 est une représentation graphique des dépendances de divers identificateurs de ces fichiers. Chaque identificateur est représenté par un ellipse contenant un nom d'identificateur et un numéro. Ce numéro est le champ IDENNUM de l'identificateur. Les flèches pleines indiquent le père d'un identificateur. Les flèches pointillées indiquent le constructeur d'un identificateur.

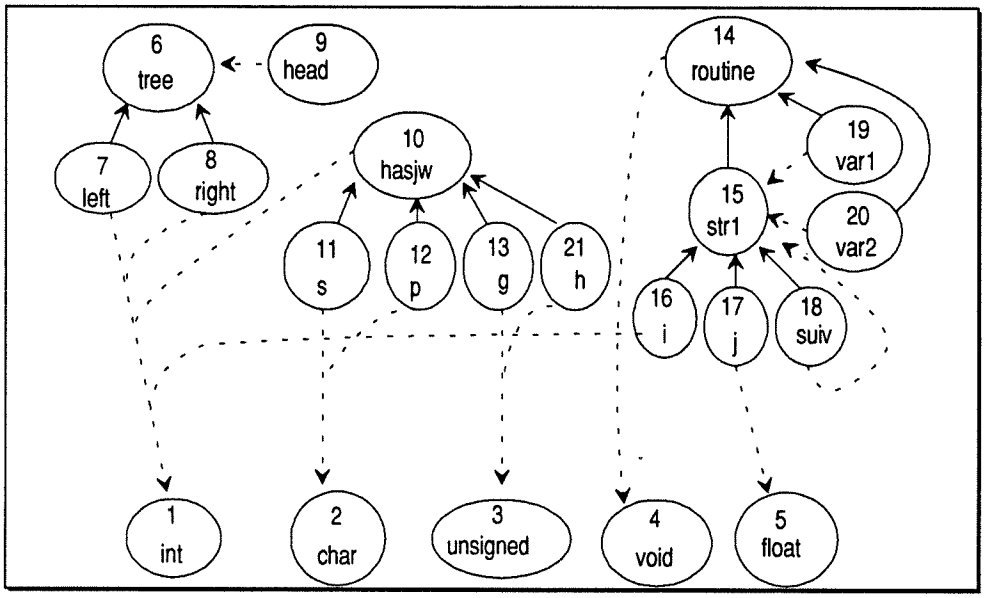


Figure 19 - *Représentation graphique des dépendances des identificateurs*

Les champs OFFSTA et OFFEND ne sont pas indiqués dans le tableau 18 puisque cette information est extrêmement difficile a calculer manuellement. De plus, elle n'importe pas beaucoup plus d'information quant au fonctionnement de cette table.

| IDENNUM | NAMENUM | FILANA | ARRAY | RECORDS | POINTER | FUNCTION | TYPE | CONST. | FATHER |
|---|---|---|---|---|---|---|---|---|---|
| 1 | int | 3 | faux | faux | faux | faux | faux | 0 | 0 |
| 2 | char | 2 | faux | faux | faux | faux | faux | 0 | 0 |
| 3 | unsigned | 3 | faux | faux | faux | faux | faux | 0 | 0 |
| 4 | void | 2 | faux | faux | faux | faux | faux | 0 | 0 |
| 5 | float | 3 | faux | faux | faux | faux | faux | 0 | 0 |
| 6 | tree | 3 | faux | vrai | faux | faux | faux | 0 | 0 |
| 7 | left | 3 | faux | faux | faux | faux | faux | 1 | 6 |
| 8 | right | 3 | faux | faux | faux | faux | faux | 1 | 6 |
| 9 | head | 3 | faux | faux | faux | faux | faux | 6 | 0 |
| 10 | haswjw.c | 3 | faux | faux | faux | vrai | faux | 1 | 0 |
| 11 | s | 3 | faux | faux | vrai | faux | faux | 2 | 10 |
| 12 | p | 3 | faux | faux | vrai | faux | faux | 2 | 10 |
| 13 | g | 3 | faux | faux | faux | faux | faux | 3 | 10 |
| 14 | routine | 2 | faux | faux | faux | vrai | faux | 0 | 0 |
| 15 | str1 | 2 | faux | vrai | faux | faux | faux | 0 | 14 |
| 16 | i | 2 | faux | faux | faux | faux | faux | 1 | 15 |
| 17 | j | 2 | faux | faux | faux | faux | faux | 5 | 15 |
| 18 | suiv | 2 | faux | faux | vrai | faux | faux | 15 | 15 |
| 19 | var1 | 2 | faux | faux | vrai | faux | faux | 15 | 14 |
| 20 | var2 | 2 | faux | faux | vrai | faux | faux | 15 | 14 |
| 21 | h | 3 | faux | faux | faux | faux | faux | 3 | 10 |

Tableau 18 - *Exemple de la table IDEN*

**Table SEG**

SEG( SEGNUM:NUMBER, IDENNUM:NUMBER, SEGCOD:NUMBER, ATT:NUMBER, SEQ:NUMBER )

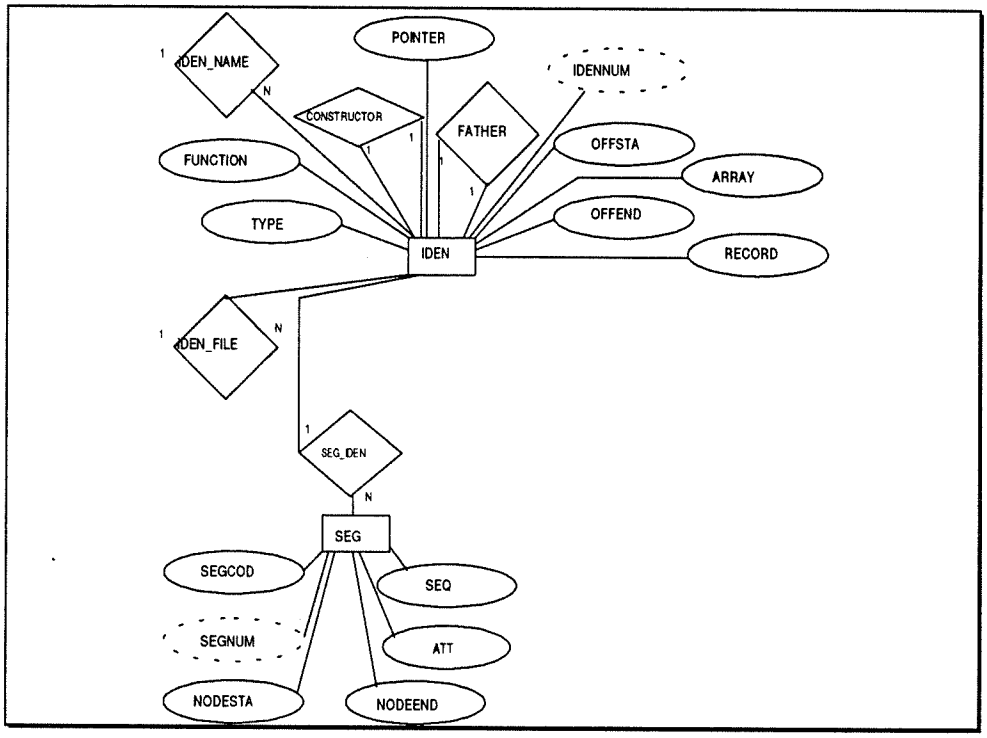


Figure 20 - *Entité SEG*

Cette table contient les segments du graphe de contrôle.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| SEGNUM | SEG | Numéro du segment. Ce numéro est unique. |
| IDENNUM | IDEN:IDENNUM via SEG_IDEN | Numéro unique provenant de la table IDEN et représentant la routine à laquelle appartient ce segment. |
| SEQCOD | SEG | Nom du segment. Voir DATGRAPH page 21 pour les différents noms de segments possible. |
| ATT | SEG | attribut d'un segment |
| SEQ | SEG | Séquence d'un segment qui suit le segment C. Ce numéro de séquence est défini seulement pour les arcs de type F et B. |
| NODESTA | SEG | Noeuds de départ. |
| NODEEND | SEG | Noeuds de fin. |

Tableau 19 - *Description des attributs de la table SEG*

## Table SEGLINE

SEGLINE(SEGNUM:NUMBER, FILANA:NUMBER, LINESTA:NUMBER, LINEEND:NUMBER )

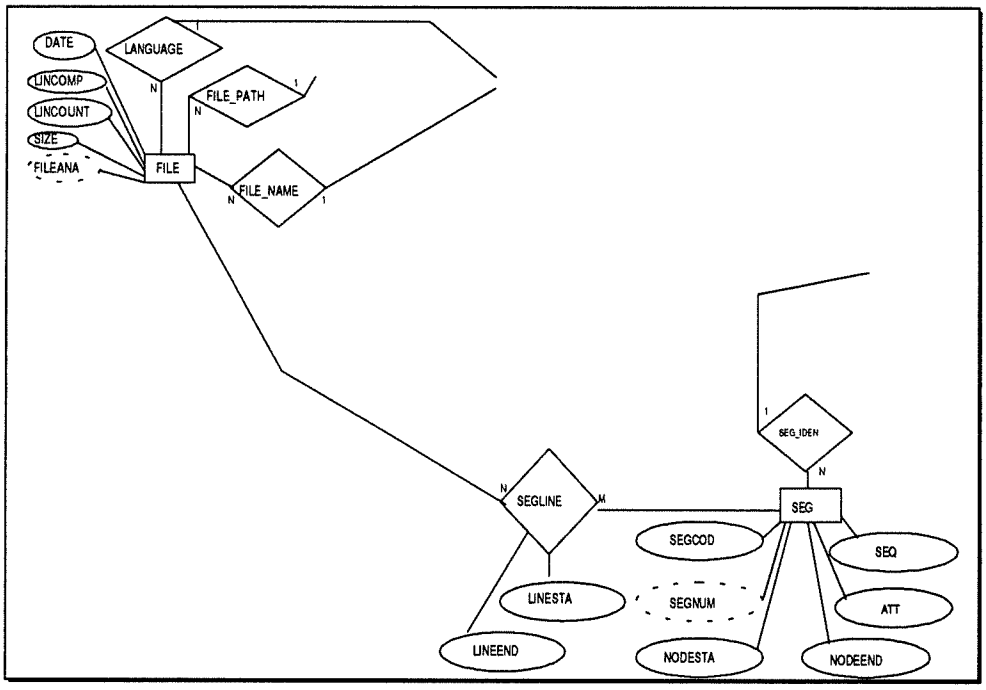


Figure 21 - *Association SEGLINE*

Cette table contient la description des points d'attache des segments du graphe de contrôle. Le graphe de contrôle est toujours vu par rapport à un fichier. Lorsqu'un noeud ne provient pas de ce fichier, il est affiché sans numéro de ligne.

Si un noeud est visible à partir de plusieurs fichiers celui-ci générera autant de fiches dans la table qu'il y a de fichiers. Ce cas est possible lorsqu'un noeud est généré dans un fichier inclus.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| SEGNUM | SEG | Numéro du segment. Ce numéro est unique. |
| FILANA | FILE:FILANA | Numéro du fichier pour lequel le champ NUMLIG est valide. |
| LINESTA | SEGLINE | Numéro de la ligne où commence le segment SEGNUM dans le fichier FILANA. |
| LINEEND | SEGLINE | Numéro de la ligne où se termine le segment SEGNUM dans le fichier FILANA. |

Tableau 20 - *Description des attributs de la table SEGLINE*

**Table SEGSC**

SEGSC(SEGNUM:NUMBER, COMPLC:NUMBER, COMPLS:NUMBER, DEC:NUMBER, COMDEC:NUMBER, STR:NUMBER, COMSTR:NUMBER )

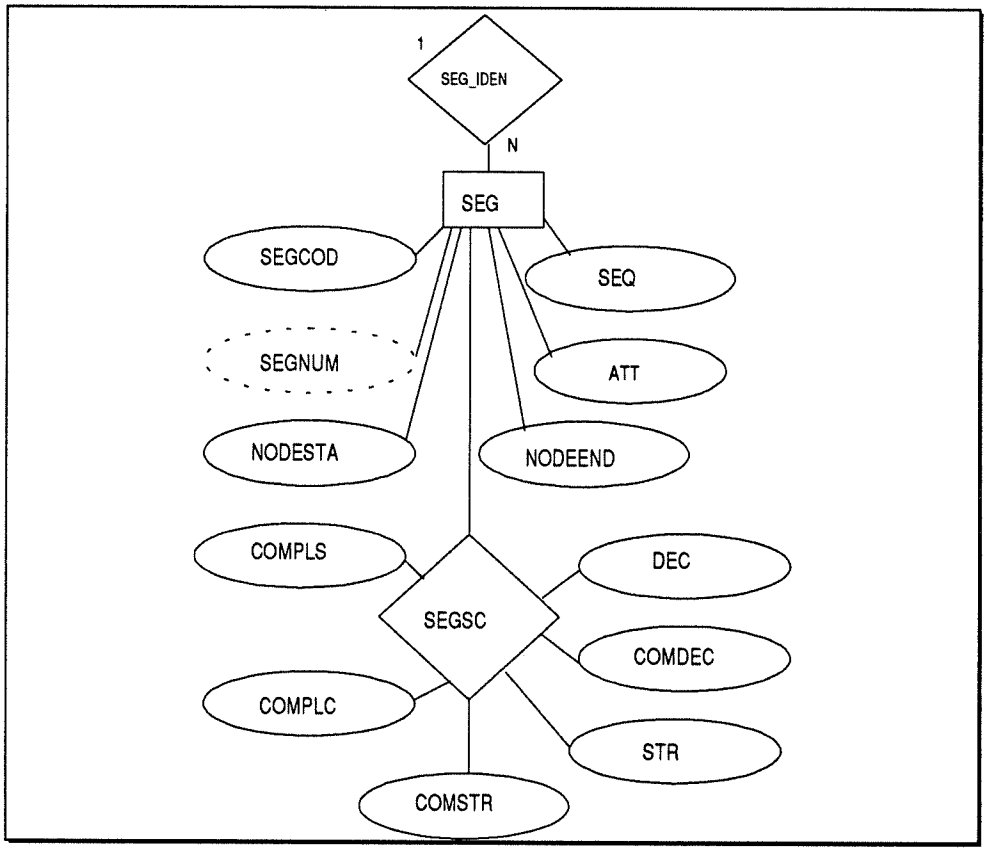


Figure 22 - *Association SEGSC*

Cette table contiendra l'information qui est propre au segment comportant une partie S ou une partie C. Les cas possibles sont : S, C, SF, LS, SC, LSF, LSC.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| SEGNUM | SEG:SEGNUM | Numéro du segment. Ce numéro est unique. |
| COMPLC | SEGSC | Complexité de la partie conditionnelle du segment. S'il n'y pas de partie conditionnelle dans le segment ce champ aura la valeur 0. |
| COMPLS | SEGSC | Complexité de la partie séquentielle du segment. S'il n'y pas de partie séquentielle dans le segment, ce champ aura la valeur 0. |
| DEC | SEGSC | Nombre d'énoncés contenus dans la section déclaration du segment. |
| COMDEC | SEGSC | Nombre d'énoncés commentaires contenus dans la section déclaration du segment. |
| STR | SEGSC | Nombre d'énoncés contenus dans la section structure du segment. |
| COMSTR | SEGSC | Nombre d'énoncés commentaires contenus dans la section structure du segment. |

Tableau 21 - *Description des attributs de la table SEGSC*

## Table SEGL

SEGL( SEGNUM:NUMBER, NAMENUM:NUMBER )

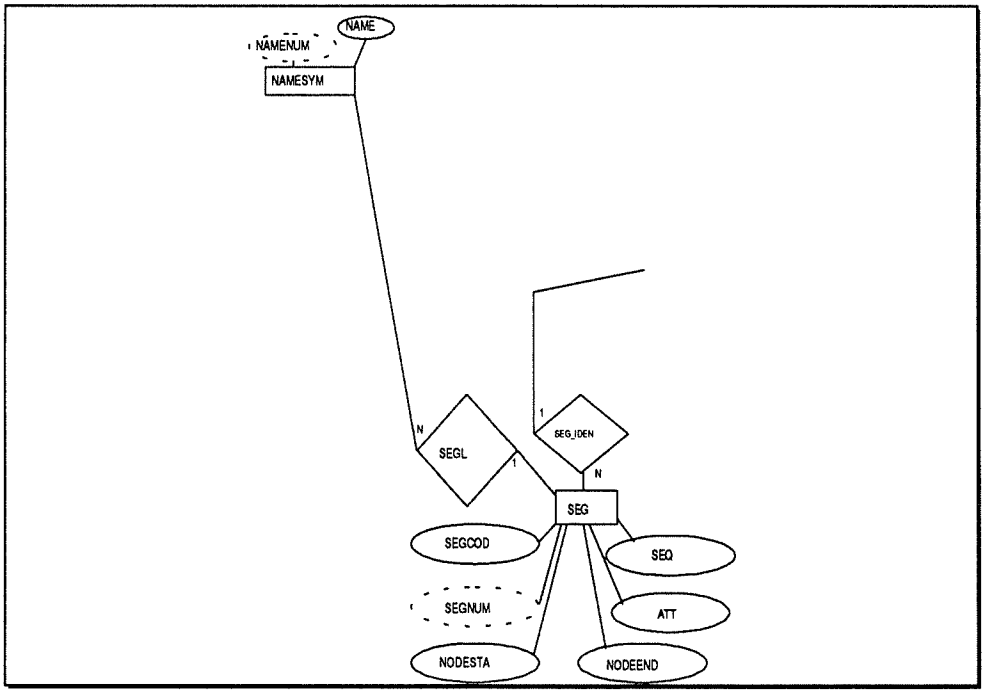


Figure 23 - *Association SEGL*

Cette table contient le nom des segments de type L.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| SEGNUM | SEG:SEGNUM | Numéro du segment. Ce numéro est unique. |
| NAMENUM | NAMESYM:NAMENUM | Identificateur qui sert d'étiquette. |

Tableau 22 - *Description des attributs de la table SEGL*

**Table DECSEG**

DECSEG( SEGNUM:NUMBER, IDENNUM:NUMBER, OFFSTA:NUMBER, OFFEND:NUMBER )

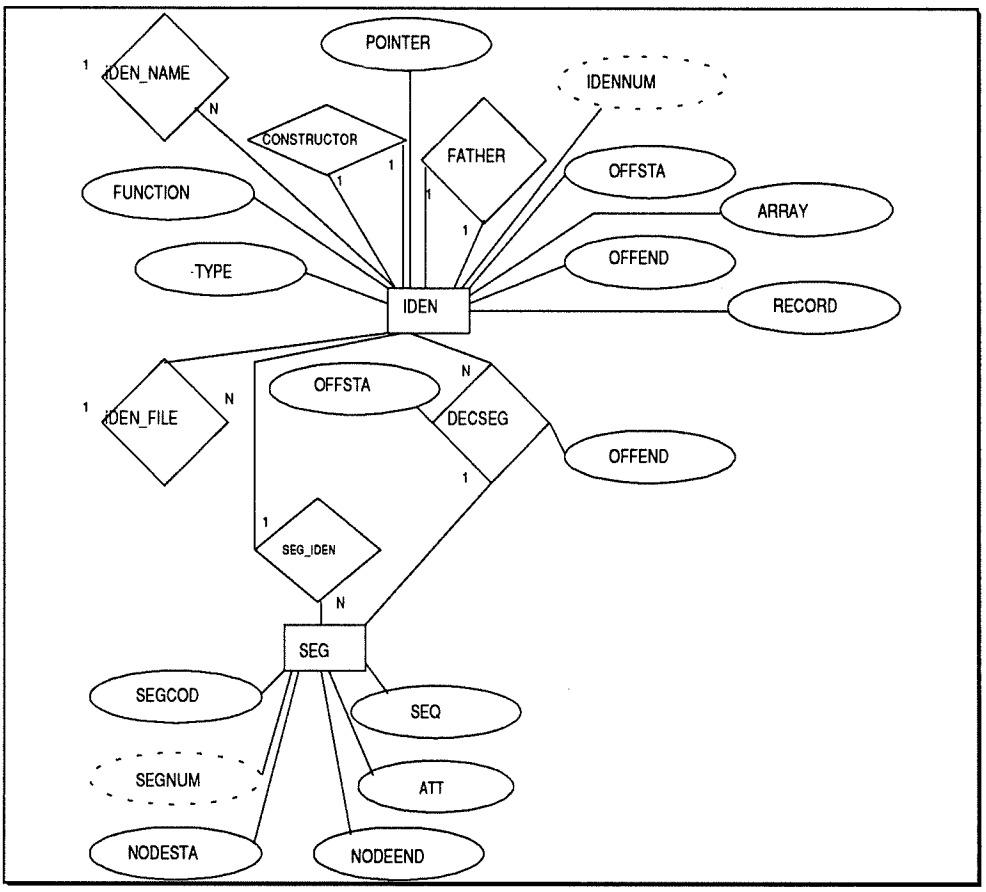


Figure 24 - *Association DECSEG*

Cette table conserve le lien entre les variables locales et la position de leur déclaration dans le graphe de contrôle, ainsi que leur portée.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| SEGNUM | SEG:SEGNUM | Numéro du segment. Ce numéro est unique. |
| IDENNUM | IDEN:IDENNUM | Numéro représentant la variable locale déclarée dans le graphe de contrôle. |
| OFFSTA | DECSEG | Début de la portée de la variable IDENNUM. |
| OFFEND | DECSEG | Fin de la portée de la variable IDENNUM. |

Tableau 23 - *Description des attributs de la table DECSEG*

## Table DEPCALL

DEPCALL(SEGNUM:NUMBER, IDENNUM:NUMBER, SEQ:NUMBER,OFFEND:NUMBER, OFFSTA:NUMBER)

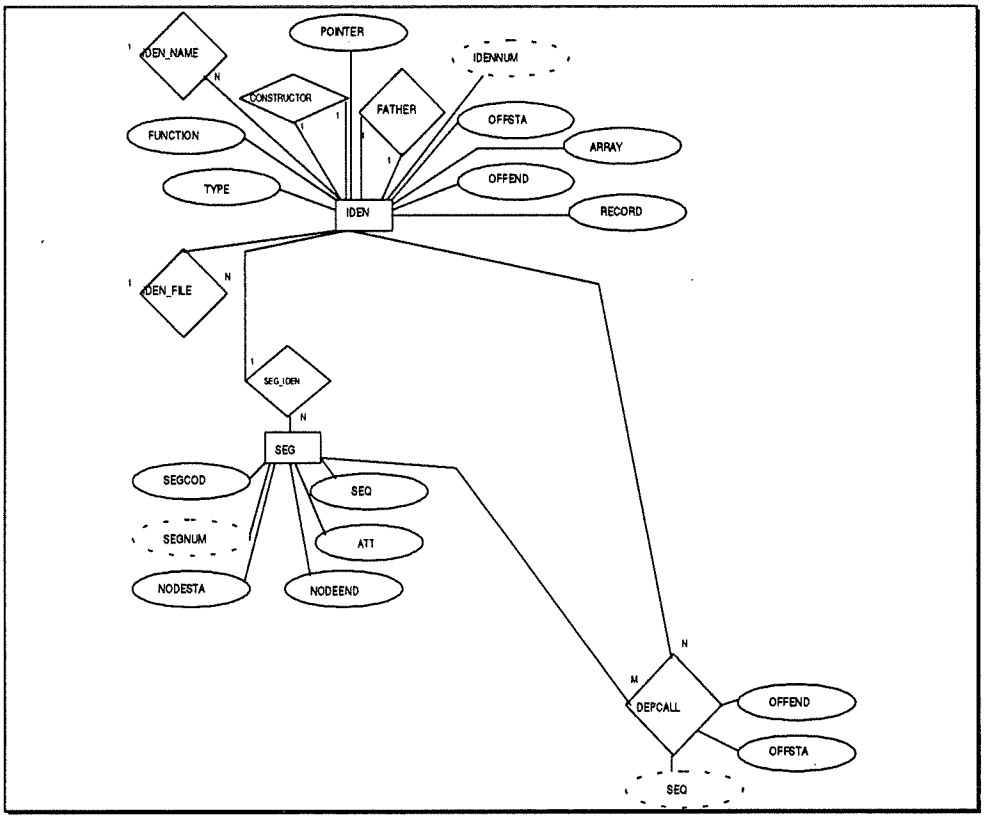


Figure 25 - *Association DEPCALL*

Cette table conserve les dépendances entre les fonctions. Tous les sites d'appels sont conservés avec leur position et leur séquence. La séquence d'une dépendance est le

l'ordre dans le segment où elle est située. La première dépendance d'un segment porte le numéro 1. La séquence des dépendances est utilisée dans les tables DEPCALL, DEPFCT et DEPFLOW.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| SEGNUM | SEG:SEGNUM | Numéro du segment auquel appartient cette appels de fonctions. |
| IDENNUM | IDEN:IDENNUM | Fonctions appelé. |
| SEQ | DEPCALL | Numéro de séquence de la dépendance dans le segment SEGNUM. |
| OFFSTA | DEPCALL | Index de début. Ce numéro en combinaison avec la table LOC permettra de retrouver la position exacte du début de la déclaration. |
| OFFEND | DEPCALL | Index de fin. Ce numéro en combinaison avec la table LOC permettra de retrouver la position exacte de la fin de la déclaration. |

Tableau 24 - *Description des attributs de la table DEPCALL*

**Table DEPFCT**

DEPFCT(SEGNUM:NUMBER, IDENDEF:NUMBER, IDENUSE:NUMBER SEQ:NUMBER, OFFEND:NUMBER, OFFSTA:NUMBER)

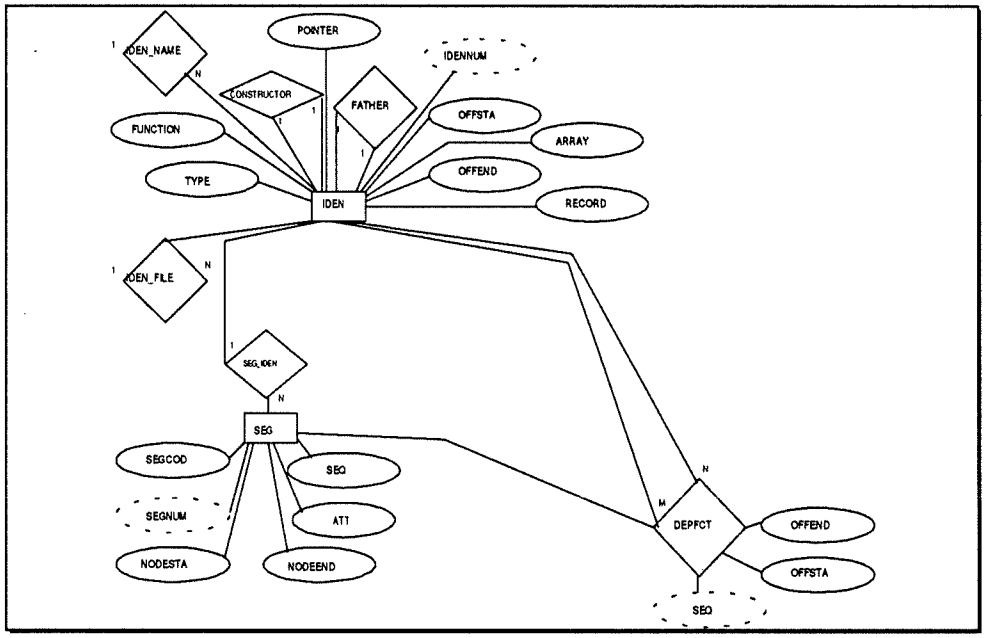


Figure 26 - *Association DEPFCT*

Cette table conserve les dépendances entre les routines et les variables. Toutes les affectations et les utilisations entre les fonctions et les variables sont conservées avec leur position et leur séquence.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| SEGNUM | SEG:SEGNUM | Numéro du segment auquel appartient cet appel de fonctions. |
| IDENDEF | IDEN:IDENNUM | Identificateur défini par une fonction ou une variable. |
| IDENUSE | IDEN:IDENNUM | Identificateur utilisé pour définir IDENDEF. |
| SEQ | DEPCALL | Numéro de séquence de la dépendance dans le segment SEGNUM. |
| OFFSTA | DEPCALL | Index de début. Ce numéro en combinaison avec la table LOC permettra de retrouver la position exacte du début de la déclaration. |
| OFFEND | DEPCALL | Index de fin. Ce numéro en combinaison avec la table LOC permettra de retrouver la position exacte de la fin de la déclaration. |

Tableau 25 - *Description des attributs de la table DEPFCT*

## Table DEPFLOW

DEPFLOW(SEGNUM:NUMBER, IDENDEF:NUMBER, IDENUSE:NUMBER, SEQ:NUMBER,OFFEND:NUMBER, OFFSTA:NUMBER)

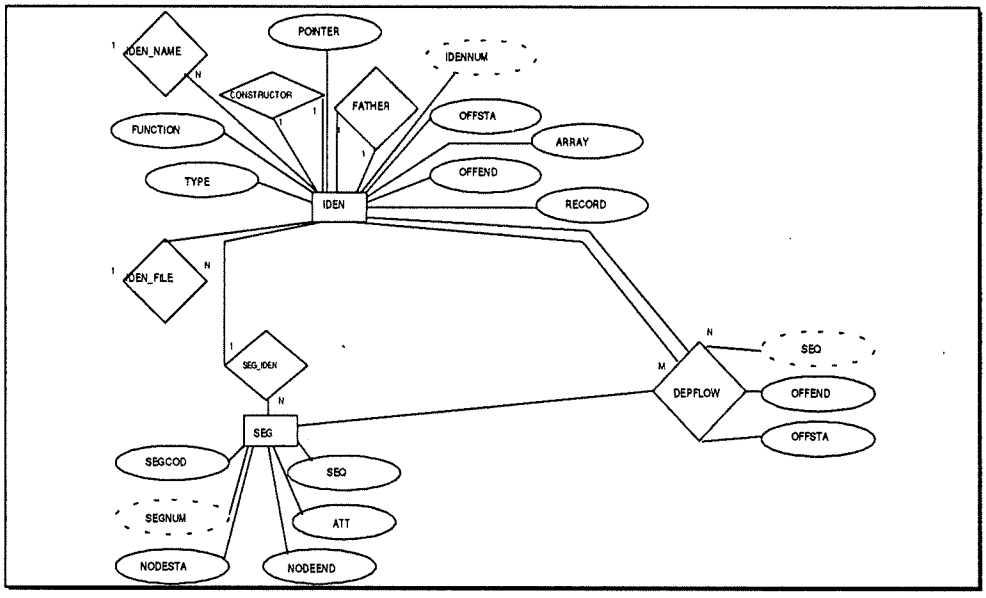


Figure 27 - *Association DEPFLOW*

Cette table conserve les dépendances entre les variables. Toutes les affectations sont conservées avec leur position et leur séquence.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| SEGNUM | SEG:SEGNUM | Numéro du segment auquel appartient cet appel de fonctions. |
| IDENDEF | IDEN:IDENNUM | Identificateurs de la variable modifiée. |
| IDENUSE | IDEN:IDENNUM | Identificateur de la variable utilisée pour la modification de IDENDEF. |
| SEQ | DEPCALL | Numéro de séquence du code intermédiaire dans le segment SEGNUM. |
| OFFSTA | DEPCALL | Index de début. Ce numéro en combinaison avec la table LOC permettra de retrouver la position exacte du début de la déclaration. |
| OFFEND | DEPCALL | Index de fin. Ce numéro en combinaison avec la table LOC permettra de retrouver la position exacte de la fin de la déclaration. |

Tableau 26 - *Description des attributs de la table DEPFLOW*

**Table METDEF**

METDEF(METNUM:NUMBER, LABEL:STRING4, DEF:LABEL:STRING64)

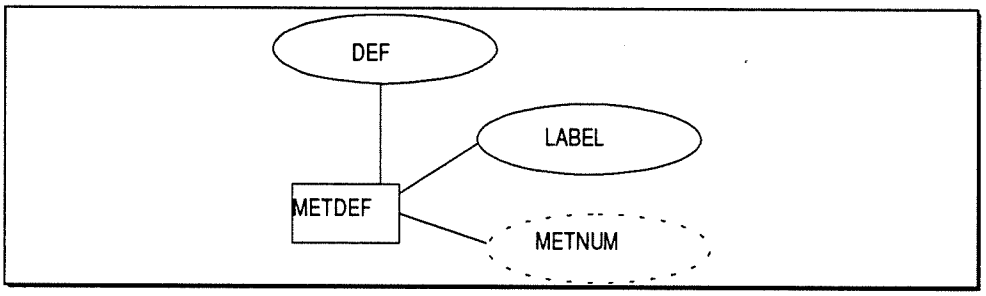


Figure 28 - *Entité METDEF*

La table METDEF contient la définition des métriques de routines, de projets et d'entités globaux. Le but premier de cette table ressemble beaucoup à la table de symboles et sert à numéroter les métriques supportées par DATRIX™. Les deux champs de caractérisation (LABEL et DEF) représentent la même information sous deux formats différents. Le champ LABEL est composé d'un maximum de quatre caractères. Par contre, le champ DEF est une brève description de la métrique.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| METNUM | METDEF | Identificateur qui sert d'étiquette. |
| LABEL | METDEF | Initiale de la métrique. |
| DEF | METDEF | Chaîne de caractères décrivant la métrique. |

Tableau 27 - *Description des attributs de la table METDEF*

**Table PROFILE**

PROFILE(PROFNUM:NUMBER, METNUM:NUMBER, MIN:NUMBER, MAX:NUMBER )

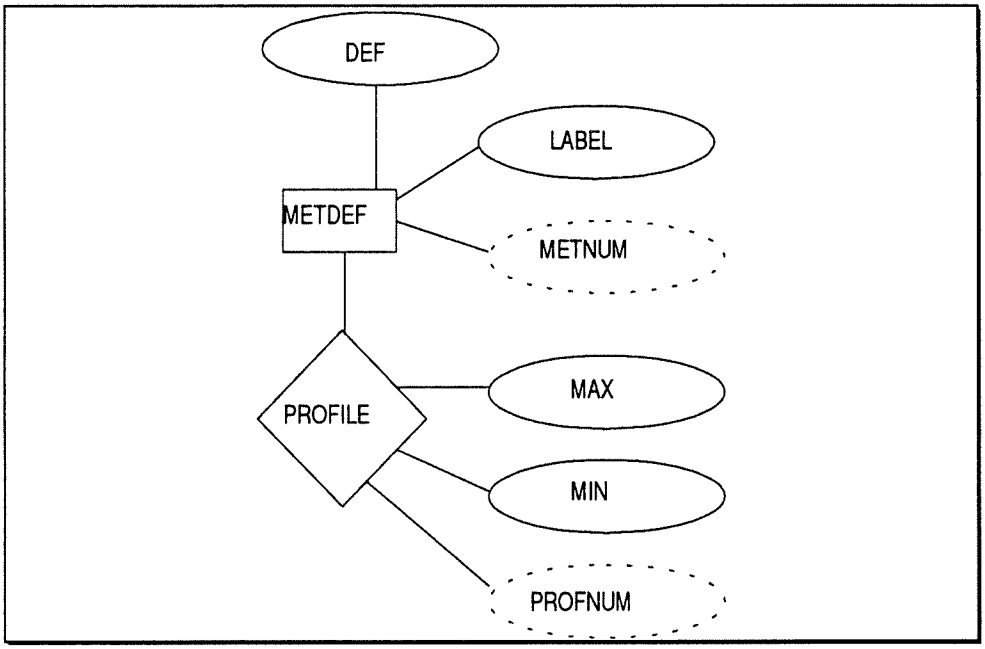


Figure 29 - *Association PROFILE*

La table PROFILE contient les bornes des métriques d'identificateurs et de projet.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| PROFNUM | PROFILE | Nom descriptif du profile |
| METNUM | METDEF:METNUM | Identificateur qui sert d'étiquette. |
| MIN | PROFILE | Valeur minimum de la métrique METNUM. |
| MAX | PROFILE | Valeur maximum de la métrique METNUM. |

Tableau 28 - *Description des attributs de la table PROFILE*

## Table METVAL

METVAL( IDENNUM:NUMBER, METNUM:NUMBER, VALUE:NUMBER )

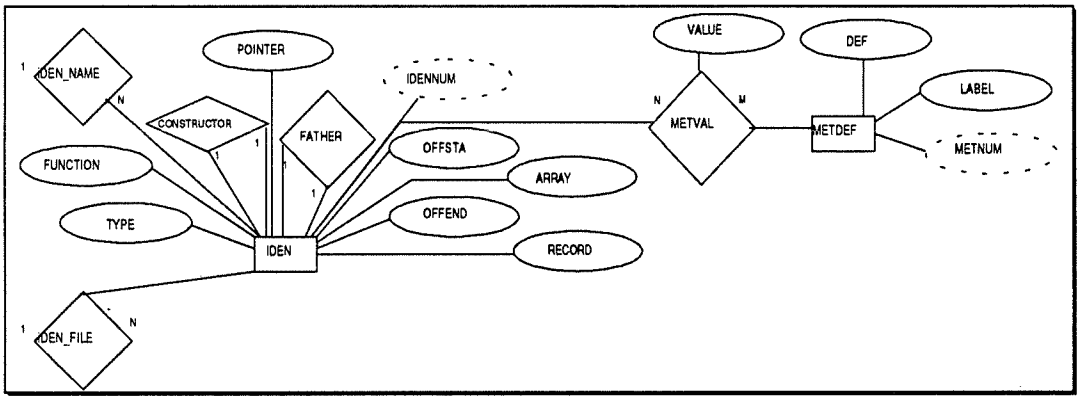


Figure 30 - *Association METVAL*

Cette table conserve la valeur des métriques d'identificateurs.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| IDENNUM | IDEN:IDENNUM | Identificateur qui sert d'étiquette. |
| METNUM | METDEF:METNUM | Identificateur qui sert d'étiquette. |
| VALUE | METVAL | Valeur de la métrique. |

Tableau 29 - *Description des attributs de la table METVAL*

**Table METVALPRJ**

METVALPRJ( METNUM:NUMBER, VALUE:NUMBER )

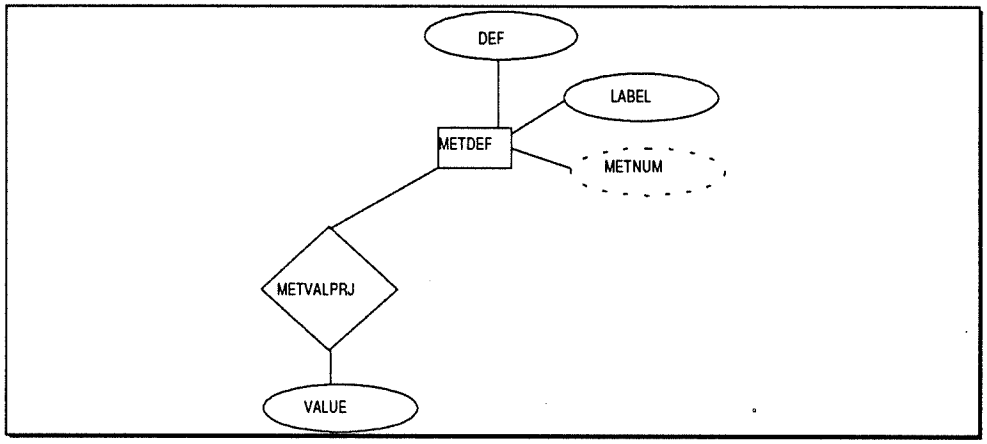


Figure 31 - *Association METVALPRJ*

Table conservant les valeurs des métriques de projet.

| NOM | ORIGINE | DESCRIPTION |
|---|---|---|
| METNUM | METDEF:METNUM | Identificateur qui sert d'étiquette. |
| VALUE | METVALPRJ | Valeur de la métrique. |

Tableau 30 - *Description des attributs de la table METVALPRJ*

# Système de gestion de base de donnée

Cette section permet d'évaluer l'utilisation d'un gestionnaire de base de données. Deux contraintes principales sont à observer, soient la grandeur de la base de données et la vitesse d'accès à l'information.

L'aspect grandeur de la base de données est très important vu l'information détaillée récupérée du code source. Par exemple, les tables DEPCALL, DEPFCT et DEPFLOW sont tout simplement une écriture normalisée des expressions présentes dans le code source. Rapidement, ces tables deviendront très grandes. La question à évaluer ici, est la capacité d'un système de gestion de base de données (SGBD) de gérer ce type d'information versus une approche conventionnelle de système de fichiers.

L'approche fichier procède souvent par accès séquentiel et transfère le contenu dans la mémoire vive de la machine. Cette approche n'est plus envisageable à cause de l'ampleur des projets analysés. Il n'est pas rare d'avoir des fichiers de données de plusieurs millions d'octets. Une approche d'accès direct avec un système de fichiers requiert souvent un fichier d'index et un fichier principal. Le problème avec cette technique est qu'elle tend à imiter les SGBD. Alors pourquoi refaire ce qui existe déjà.

Le vrai problème en ce qui concerne l'évaluation de l'ampleur de la base de données est plutôt d'évaluer le niveau de raffinement de l'information récupérée. Le modèle présenté ici peut être implanté en partie en fonction de la dimension prise par la base de données. Afin d'évaluer cette ampleur, il est possible d'estimer le volume occupé par chaque table en fonction d'un projet typique. Ce problème est le même pour les deux approches et il ne saurait discriminer celle qui est la meilleure.

Le problème de performance est très délicat. Il est connu que les systèmes flexibles et généraux sont toujours moins performants que les systèmes dédiés à une application

précise. Seuls des bancs d'essais en fonction de cette application précise permettront d'évaluer ce point. Le tableau 1 est utilisé pour l'évaluation de la dimension de la base de données résultant de l'analyse d'un projet. Le tableau 31 résume les éléments d'un projet quantifiant la dimension des tables du schéma relationnel.

| TABLE | CLÉ (OCTET) | ATT. (OCTET) | TOTAL (OCTET) | QUANTIFICATEUR |
|---|---|---|---|---|
| NAMESYM | 4 | 32 | 40 | Nombre de mots différents. |
| PATH | 4 | 8 | 16 | Nombre de répertoires. |
| FILE | 4 | 28 | 36 | Nombre de fichiers. |
| EXECUTE | 8 | 0 | 16 | Nombre d'exécutables. |
| METGRAB | 8 | 88 | 104 | Nombre de modifications aux fichiers. |
| LOC | 8 | 8 | 24 | Deux fois le nombre d'inclusions de fichier. |
| LAYER | 8 | 0 | 16 | Nombre de couches. |
| LEXCOUNT | 8 | 4 | 20 | La sommation des mots différents par fichier. |
| LEXDEF | 8 | 4 | 20 | Nombre de mots réservés. |
| IDEN | 4 | 21 | 29 | Nombre de déclarations. |
| SEG | 4 | 16 | 24 | Nombre de segments. |
| SEGLINE | 8 | 8 | 24 | Nombre de segments. |
| SEGSC | 4 | 24 | 32 | Nombre de segments. |
| SEGL | 8 | 0 | 16 | Nombre d'étiquettes. |
| DECSEG | 8 | 0 | 16 | Nombre de variables locales. |
| DEPCALL | 12 | 8 | 32 | Nombre d'appels de fonction. |
| DEPFCT | 16 | 8 | 40 | Nombre d'expressions. |
| DEPFLOW | 16 | 8 | 40 | Nombre d'expressions. |
| METDEF | 4 | 68 | 76 | Nombre de métriques. |
| PROFILE | 8 | 8 | 24 | Nombre de métriques. |
| METVAL | 8 | 4 | 20 | Nombre d'identificateurs fois le nombre de métriques. |
| METVALPRJ | 4 | 4 | 12 | Nombre de métriques de projet. |

Tableau 31 - *Dimension des tables et quantificateurs du nombre de fiches*

# Conclusion

Ce schéma est la base d'une implantation du modèle DATGRAPH. Les points suivants devront être étudiés afin de poursuivre la réalisation de ce schéma.

- ❒ Étudier les dépendances fontionnelles et des dépendances multivaluées afin de s'assurer que les tables sont bien normalisées.

- ❒ Étudier les requêtes probables afin d'évaluer si une dénormalisation augmenterait les performances du système.

- ❒ Évaluer de la dimension d'un projet qui serait conservé à l'aide de ce modèle. Effectuer cette évaluation pour plusieurs projets afin d'estimer le comportement du modèle selon les projets.

- ❒ Définir les entité sur lesquelles les métriques seront évaluées. La liste suivante est un exemple de ces entités mesurables: un projet, un module, un fichier exécutable, une couche de module, une routine, une variable, un type, un segment.

- ❒ Définir, parmi les quarante-sept métriques déjà extraites par DATRIX™, lesquelles seront conservées par valeur et lesquelles feront l'objet de requêtes sur les niveaux inférieurs.

- ❒ Définir de nouvelles métriques à partir du modèle et étudier l'apport d'information nouvelle par celles-ci.

# Bibliographie


[ADLE88] Adler Mike,"An algebra for Data Flow Diagram Process Decomposition", *IEEE Transaction on Software Engineering*, Vol. 14, No. 2, February 1988.

[AHO86] Aho Alfred V., Ravi Sethi, Jeffrey D.Ullman, *Compilers Principles, Techniques, and Tools*, Addison Wesley 1986

[BELA81b] Belady L.A. , Evangelisti C.J.,"System Partitioning and Its Measure",*The Journal of Systems and Software* 2, 23-29, 1981.

[BURK90] M. C. Burke and B. G. Ryder,"A Critical Analysis of incremental iterative data flow analysis Algorithms",*IEEE Trans. on soft. Eng.* Vol. 16, No. 7, July 1990.

[CHAM90] Craig Chambers, David Ungar,"Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oreinted programs",*Proc of ACM SIGPLAN 1990,* p.150-164.

[CHAS90] David R. Chase, Mark Wegman, F.Kenneth Zadeck,"Analysis of pointers and Structures",*Proc. of the ACM SIGPLAN 90,* White Plains, New York, june 22-22, 1990.

[GILL90] Mark L. Gillenson, "Physical Design Equivalence in Database Conversion", *Communication of the ACM*, vol. 33, No. 8, August 1990.

[HALS77] Halstead M.H.,*Element of Software Science*,Elsevier North-Holland, Amsterdam (1977)

[HENR81] Henry, S.; Kafura, D.,"Software Structure Metrics Based On Information Flow",*IEEE Trans. Software Eng.* (Usa) Vol.Se-7, No.5 510-18. Sept. 1981.

[LASK83] J. W. Laski and B. Korel,"A data flow oriented program testing strategy", *IEEE Transaction on Software Engineering*, Vol. SE-9, no. 3, May 1983, pp.347-354.

[MCCA76] McCabe, T.J.,"A Complexity Measure",*Ieee Trans. Software Eng.* (Usa) Vol.Se-2, No.4 308-20, Dec. 1976

[MOSE90] Moser Louise E., "Data Dependency Graphs for Ada Programs",*IEEE Transactions on Software engineering* Vol. 16 No. 5 May 1990.

[PODG90] Andy Podgurski, Lori A. Clarke,"A Formal Model of Program Dependences and Its Implications for software Testing, Debugging, and Maintenance", *IEEE Transaction on software Engineering*. Vol. 16, No 9. September 1990.

[RAPP85] Rapps R. and Weyuker E.,"Selecting Software Test Data Using Data Flow Information", *IEEE Transaction on Software Engineering*, Vol. SE-11, NO.4, April 1985

[RYDE86] Ryder, Barbara G., and Paul Marvin C.,"Elimination Algorithms for Data Flow Analysis",*ACM Computing Surveys,* VOl. 18, No. 3, Septembre 1986

[SIMO89] M. Simoneau, A. Beaucage, P.N. Robillard,"Definition d'un modèle pour conserver l'information des programmes sources (Modèle DATGRAPH) pour le logiciel DATRIX. EPM/RT-89/21, 1989.

[STRO87] B. Stroustrup, *The C++ programming language,* Addison Wesley, july 1987.

[TAI84] Tai, K.-C., A Program Complexity Metric Based On Data Flow Information In Control Graphs, *Proceedings Of The 7th International Conference On Software Engineering* (Cat. No. 84ch2011-5) 239-48. 1984.

[ULLM88] J. D. Ullman, *Principles of database and knowledge-base systems*, volume 1, Computer Science Press, 1988.

[WILD88] N. Wilde, R. Huitt, R. Ogando,"A Data-Base Program Representation for Software Maintenance Tools", SERC-TR-25-F, Software engineering research center, University of Florida, December 1988.

[YAU81] Yau, S. S. and Grabow, Paul C.,"A Model for representing Programs Using Hierarchical Graphs",*IEEE Trans. on Soft. Eng.*, vol. SE-7, No.6, pp. 556-574.