MASTER EN SCIENCES INFORMATIQUES

Représentation formelle de structures de données. Application au système BS 2000

Bolle, Bernard

Award date:
1985

Awarding institution:
Universite de Namur

ANNEE ACADEMIQUE 1984-85

REPRESENTATION FORMELLE DE STRUCTURES DE DONNEES. APPLICATION AU SYTEME BS 2000

MEMOIRE PRESENTE PAR
BOLLE BERNARD EN VUE
DE L'OBTENTION DU GRADE
DE LICENCIE ET MAITRE
EN INFORMATIQUE

PROMOTEURS :
J.RAMAEKERS
PH.DUMONT

Nous tenons à remercier Mr. Ramaekers pour ses nombreux conseils méthodologiques et pour l'aide suivie qu'il a bien voulu nous apporter tout au long de cette année.

Nous tenons également à remercier tout spécialement Mr. Dumont pour son aide efficace ainsi que pour ses idées et ses conseils prodigués tout au long de l'élaboration de ce travail qui n'existerait pas sans lui.

Mr. de Cocqueau ne peut être oublié dans ces remerciements pour l'attention qu'il a bien voulu nous apporter et pour sa précieuse collaboration

Enfin, nous tenons également à remercier tout le personnel de Siemens Software de Rhisnes pour son accueil et ses conseils lors de l'implémentation de ce travail, et tout spécialement Mr. Mehaignoul et Mr. Sevrin.

TABLE DES MATIERES

1 INTRODUCTION

Ce travail constitue la première étape de la création d'un ensemble d'outils d'aide à la maintenance d'un software (en particulier du système BS 2000 de Siemens).

Nous allons tenter tout d'abord de situer le problème avant d'exposer les différentes méthodes possibles. Un survol rapide des étapes ultérieures à ce travail est fait. Enfin, nous définirons l'objet de ce travail.

1.1 Origine du problème

Lors d'un crash du système ou d'une partie du système, on vide généralement le contenu de la mémoire sur papier (dump), de facon à permettre aux specialistes d'analyser la situation qui a amené l'erreur et de découvrir les causes de cette erreur. Les crashes se produisent fréquemment lors du développement de nouvelles versions, mais aussi parfois dans les versions existantes.

Il y a une dizaine d'années, un dump représentait quelques centaines de pages et était donc facilement manipulable. D'autre part, vu le nombre modeste de développeurs et d'utilisateurs, on pouvait rapidement diffuser la correction, de sorte qu'une erreur avait relativement peu de chances de se reproduire.

A l'heure actuelle, la taille du systéme, le nombre de projets en développement et le nombre d'utilisateurs se sont multipliés. Il en résulte deux conséquences. La première est qu'il faut employer des procédures systématiques pour diffuser les corrections de manière groupée car il est impossible d'expédier à tout le monde toutes les corrections au fur et à mesure de leur production. De plus, il est encore plus difficile de contrôler si chacune d'elles est bien appliquée. En conséquence, il peut s'écouler un certain temps entre le moment où une correction est trouvée et le moment où elle est appliquée chez le client. Une même erreur a donc beaucoup de chances de se reproduire plusieurs fois, au point que, chez le support clientèle, la majorité des erreurs traitées sont des erreurs connues.

La seconde conséquence est l'augmentation gigantesque de la taille des dumps. Ceux-ci se présentent souvent sous la forme d'une pile de papier de 50 cm de haut, de sorte que le temps nécessaire pour trouver dans un dump une information utile devient de plus en plus élevé. En outre, le nombre de caractères lus par un spécialiste est extrêmement réduit. Signalons qu'actuellement, les dumps sont pris d'abord sur un support intermédiaire, bande ou fichier, avant d'être édités.

Pour situer correctement le problème, nous allons essayer de comprendre la manière dont le spécialiste utilise un dump en vue de

découvrir les causes d'une erreur. En fait, le travail consiste à vérifier le contenu de certaines tables et leur consistance. Il faut donc pouvoir retrouver ces tables dans la masse d'informations que constitue un dump. On utilise pour cela les nombreux pointeurs contenus dans les tables, ces pointeurs liant les tables les unes aux autres.

Pour aider le spécialiste dans ce travail, on a d'abord songé à adapter un produit de diagnostic interactif à l'analyse des dumps. On dispose en effet de certains produits (AID,HELGA) qui permettent d'afficher au terminal n'importe quel champ de la mémoire. Au lieu de tirer cette information de la mémoire, ils peuvent la tirer d'un fichier contenant une copie de celle-ci. Il semble donc que ces produits puissent être utilisés de manière bénéfique dans l'analyse des dumps.

En fait, on constate que ces outils ne sont que très rarement utilisés dans ce domaine. Pour expliquer ce fait,il faut se rendre compte que lorsque le spécialiste "circule" dans un dump, il éprouve fréquemment le besoin de conserver certains endroits importants à sa disposition (par exemple, s'il désire comparer le contenu de deux tables). Pour cela, il va utiliser un certain nombre de signets qu'il intercale entre les pages du dump. Par contre, lorqu'il utilise un terminal et un dump dans un fichier, il ne peut y mettre de signet. Quelle que soit l'information cherchée, il doit repartir chaque fois de la racine du graphe (constituée par une table du système) au lieu de repartir du sommet le plus proche. Le temps de recherche est alors multiplié par dix ou vingt. De plus, s'il doit comparer plusieurs champs dans deux tables différentes, il est obligé soit de trouver chaque table autant de fois qu'il y a de champs soit de recopier l'une des tables sur un papier.

1.2 Deux méthodes envisageables

Pour résoudre ce problème, deux voies semblent possibles. La première consiste à prendre acte du fait que le spécialiste veut avoir plusieurs endroits à sa disposition simultanément et à s'attacher à les lui fournir. La deuxième voie consiste à s'arranger pour qu'il n'ait plus besoin de conserver tous ces points de repère et à lui donner directement l'information finale.

La firme Siemens développe actuellement la première méthode au support clientèle. L'utilisateur pourra diviser son écran en plusieurs compartiments dans lesquels il peut faire apparaître les champs qui lui conviennent. Des commandes et même des touches programmées lui permettront de changer le contenu de l'un des compartiments sans changer les autres. D'autre part, il est prévu de pouvoir faire faire automatiquement un certain nombre de vérifications, dans l'idée de reconnaître immédiatement des erreurs connues.

L'efficacité de ce système dépendra de la qualité de l'interface offert (des commandes) et de la taille de l'écran. Mais, on peut se demander si les écrans actuels sont suffisants.

La deuxième méthode remonte beaucoup plus loin dans la logique du système. Nous avons dit qu'une grande partie du travail du spécialiste consistait à circuler dans le graphe pour atteindre certains endroits du dump. En fait, c'est la partie ingrate du travail. Elle s'oppose en cela au traitement de l'information ainsi obtenue. L'idéal serait donc d'automatiser au maximum la première partie du travail. On pourrait imaginer un outil qui puisse non seulement localiser une table, mais également donner des appréciations sur la valeur des divers champs et pointeurs de la table. Ceci est particulièrement important car ces appréciations permettent de décrire les symptômes d'une erreur. En effet, une erreur produit en général une situation du système non prévue et non supportée. Cette situation peut donc être décrite sous la forme d'une expression logique d'appréciations sur certains pointeurs ou autres champs d'une ou plusieurs tables.

La première idée venant à l'esprit est donc de créer un outil qui parcoure les données du système et permette d'accéder à la zone que l'on désire observer. Il faut pour cela que cet outil puisse "connaître" le graphe représentant les données du système. Ceci implique une déclaration préalable de ces données et bien évidemment un langage de déclaration des données.

1.3 Les étapes du travail

Ainsi qu'on peut le voir, il s'agit d'une recherche à relativement long terme qui dépasse largement le cadre d'un mémoire. Les étapes principales de ce travail pourraient être les suivantes:

1. Définition d'un langage décrivant la structure du système.

2. Description du système au moyen de ce langage.

3. Ecriture d'un outil qui circule dans une masse de données (système ou dump) en utilisant cette description.

4. Ecriture d'un produit utilisant le précédent et permettant d'analyser un dump à partir d'un terminal.

5. Ecriture d'un produit utilisé par le précédent qui retient la démarche effectuée par le spécialiste pour permettre

   (1) d'interrompre un travail d'analyse et de le reprendre plus tard,

   (2) de conserver les symptômes de l'erreur.

6. Ecriture d'un produit utilisant ceux décrits en 3 et en 5 et vérifiant dans un dump les symptômes d'une erreur fournis par le produit précédent (ceci pour vérifier automatiquement si l'erreur à corriger est connue).

7. Ecriture d'un produit de gestion des symptômes. Au lieu de vérifier indépendemment et systématiquement les symptômes de toutes les erreurs connues, on pourrait envisager de les grouper en tenant compte de parties communes, ceci pour pouvoir faire une recherche plus intelligente.

Cet ensemble de produits permettrait donc de vérifier automatiquement si on est en face d'une erreur connue. Si ce n'était pas le cas, le produit décrit en 3 permettrait au spécialiste de découvrir plus rapidement les symptômes de cette nouvelle erreur.

1.4 Objet du présent travail

En ce qui concerne ce mémoire, il s'attache à la première étape et à une partie de la deuxième comme exemple.

La première chose à faire est l'étude des données manipulées par le système en vue d'en extraire la structure logique.

Il faut ensuite décrire un langage permettant à l'utilisateur de "déclarer" les structures de données sur lesquelles il veut travailler.

Enfin, une description de l'implémentation de la représentation des données du système est faite.

En ce qui concerne le langage de déclarations, seule une version affaiblie a pu être implémentée. Une suggestion pour un langage plus puissant est présentée dans le chapitre 5.

2 DESCRIPTION DES STRUCTURES DE DONNEES

2.1 INTRODUCTION

La première étape du travail consiste à examiner les données manipulées par le système pour en extraire une représentation théorique. Une étude de ces données a été faite et l'analyse qui suit en découle.

La représentation théorique des données sera utilisée par un outil de parcours dans un dump du système ou dans le système lui-même. Cet outil a comme spécification de pouvoir accéder automatiquement à une partie des données du système en travaillant à la fois sur la représentation des données et sur un dump du système.

Bien que l'implémentation de cet outil ne fasse pas partie de ce travail, il semble utile d'analyser grossièrement son fonctionnement pour justifier la représentation des données.

2.2 Structures de données manipulées par le système

La description des données manipulées par le système va être illustrée par un exemple. Cet exemple constitue un noyau des données du système. En outre, la plupart des structures étudiées dans ce travail sont présentes dans ce noyau.

Une première approche permet de mettre en évidence que les données sont groupées en tables et que ces tables sont reliées entre elles. Les paragraphes qui suivent analysent plus précisément ces différentes structures.

2.2.1 Les tables

Le système manipule un certain nombre de tables. Ces tables contiennent des données qui définissent l'état du système (c'est à dire l'état des tâches et des ressources). Ainsi, pour chaque tâche existant dans le système, il existe une table appelée Task Control Block (TCB). Cette table contient des informations par exemple sur l'état de la tâche ou sur le type de tâche dont il s'agit. Il en va de même pour chaque fichier utilisé par une tâche.

On remarque immédiatement que certaines tables sont uniques (comme la table XVT qui contient des informations nécessaires pour le contrôle de toutes les tâches existantes), alors que d'autres existent en plusieurs exemplaires (comme les TCB déjà cités).

Dans le cas où il existe plusieurs occurrences du même type de table, ces occurrences sont liées entre elles par une ou plusieurs structures de données. Par exemple les Task File Table (cette table est utilisée entre autre pour relier un fichier existant dans le catalogue à un programme en utilisant un linkname) d'une même tâche forment une queue doublement chaînée.

Nous avons donc jusqu'ici mis en évidence deux types de structures de données :

- les tables

- les listes (que l'on peut subdiviser en queue simple, queue double, anneau simple et anneau double).

2.2.2 Les liens entre les tables

Toutes ces tables forment un réseau, mais les liens qui existent entre les différents éléments peuvent être de nature différente.

Nous avons déjà mis en évidence la structure de liste qui peut relier certaines tables entre elles, mais il existe bien sûr d'autres types de liens.

Le lien le plus courant entre deux tables est le pointeur. Ainsi, la XVT déjà citée contient entre autres un pointeur vers la TLT. Ces pointeurs eux-mêmes peuvent être de types différents. Il peut s'agir d'une adresse simple (la longueur est alors de quatre bytes ou parfois de trois bytes), d'un déplacement (sur deux bytes) ou encore de la longueur d'une zone. Ainsi, si on considère une suite d'articles dans un fichier séquentiel, on remarque que chaque article contient un champ qui donne la longueur de l'article. Il s'agit donc dans ce cas d'un pointeur vers l'article suivant basé sur l'article courant.

Un cas plus spécial de lien est ce que nous appelerons dans la suite le pointeur multiple. Essayons d'expliquer sa signification sur un exemple (sans entrer dans trop de détails). Lorsqu'une tache ouvre un fichier, le système initialise un certain nombre de tables qui relient la tache au buffer de ce fichier (Task File Table, P1 File Control Block, P2 File Control Block).

Une de ces tables (la P1FCB) permet d'accéder au buffer du fichier. Un buffer peut être considéré comme un type de tables. Cependant, ce buffer contient différents types de structures de données suivant le type du fichier (data-block ISAM ou block SAM, par exemple). Nous pouvons donc dire qu'il existe différents types de tables buffer (appelons-les SAMBUF et ISAMBUF).

Ainsi, le pointeur de la P1FCB qui permet d'accéder au buffer peut pointer vers des types de tables différents. Pour vérifier de quel type il s'agit, il faut, dans le cas présent, tester une variable contenue dans la P1FCB elle-même.

En résumé, un pointeur est appelé multiple s'il peut donner accès à des tables de types différents suivant la situation dans laquelle on se trouve.

Il existe un dernier type de lien entre deux types de tables. Il s'agit du vecteur. Prenons à nouveau un exemple typique. Nous avons déjà parlé des TCB. Il existe une table unique (la TLT) qui contient les adresses de tous les TCB actifs. Le lien entre TLT et TCB peut donc être formalisé par un vecteur dont chaque composante est un pointeur vers une TCB.

Remarquons enfin que, dans certains cas, une table a du type A pointe vers une table b du type B qui pointe elle-même vers la même occurrence a. C'est le cas par exemple entre la TFT et la P2FCB. Nous dirons dans ce cas que ces pointeurs possèdent un inverse.

2.2.3 Page partielle

Il existe dans le système une structure de données qui n'entre malheureusement pas dans les types définis plus haut. Il s'agit des données permettant la gestion des pages partielles que le système peut allouer à une tache.

Sans entrer dans les détails de ces données, disons simplement que chaque page partielle occupée est précédée d'un header permettant de connaître entre autre chose la longueur et le propriétaire de la zone, et que les pages partielles libres sont chaînées.

Considérant malgré tout que ces structures doivent figurer dans la représentation des données du système, un type supplémentaire doit être ajouté. Ce type sera appelé Partial Page. Contrairement au type table où il existe un nombre indéterminé de sous-types (les TCB forment par exemple un sous-type), le type Page Partielle ne contient que trois sous-types. Chacun de ces sous-types correspond à une classe de mémoire (classe 3, classe 4 et classe 5 - utilisateur).

2.3 Définition du graphe formel

De l'analyse faite ci-dessus, nous pouvons déduire que les données manipulées par le système forment un graphe. Ce graphe physique (c'est à dire existant dans le système) peut être représenté par un graphe formel. La définition de ce graphe constitue la première étape de la représentation des données du système.

2.3.1 Les sommets du graphe formel

Les sommets ou noeuds du graphe formel sont les tables, les listes (queues simples ou doubles et anneaux simples ou doubles), et les pages partielles.

Les sommets du graphe formel seront representés par un rectangle. Pour un type de tables, ce rectangle contiendra le nom de la table. Pour un type de queues simples, il contiendra le nom précédé du symbole % (et précédé de %% pour un type de queues doubles). Le nom sera précédé du symbole # pour un type d'anneaux simples (et de ## pour un type d'anneaux doubles). Pour le type Page Partielle, on utilisera les noms PP3, PP4 ou PP5 selon le sous-type considéré

2.3.2 Les arcs du graphe formel

2.3.2.1 Référence simple

Lorsqu'un type de tables contient un pointeur vers un autre type de noeud, les deux noeuds correspondants dans le graphe formel sont liés par une référence simple (symbole > dans le schéma). Notons qu'une référence (simple ou pas) appartient toujours a la table origine.

2.3.2.2 Référence vecteur

Lorsqu'un type de tables contient un vecteur de pointeurs vers un autre type de noeuds, on dira que les deux noeuds sont liés par une référence vecteur (symbole >>).

2.3.2.3 Référence multiple

Si un type de tables contient un pointeur multipe vers un autre type de noeuds, ces deux noeuds seront liés par une référence multiple. Ce lien est représenté par un arc ayant plusieurs extremités.

2.3.2.4 Relation de composition

Une liste est toujours composée d'un certain type de noeuds. Ces deux sommets sont donc liés par une relation de composition schématisée par le symbole ><.

2.3.3 Racine du graphe formel

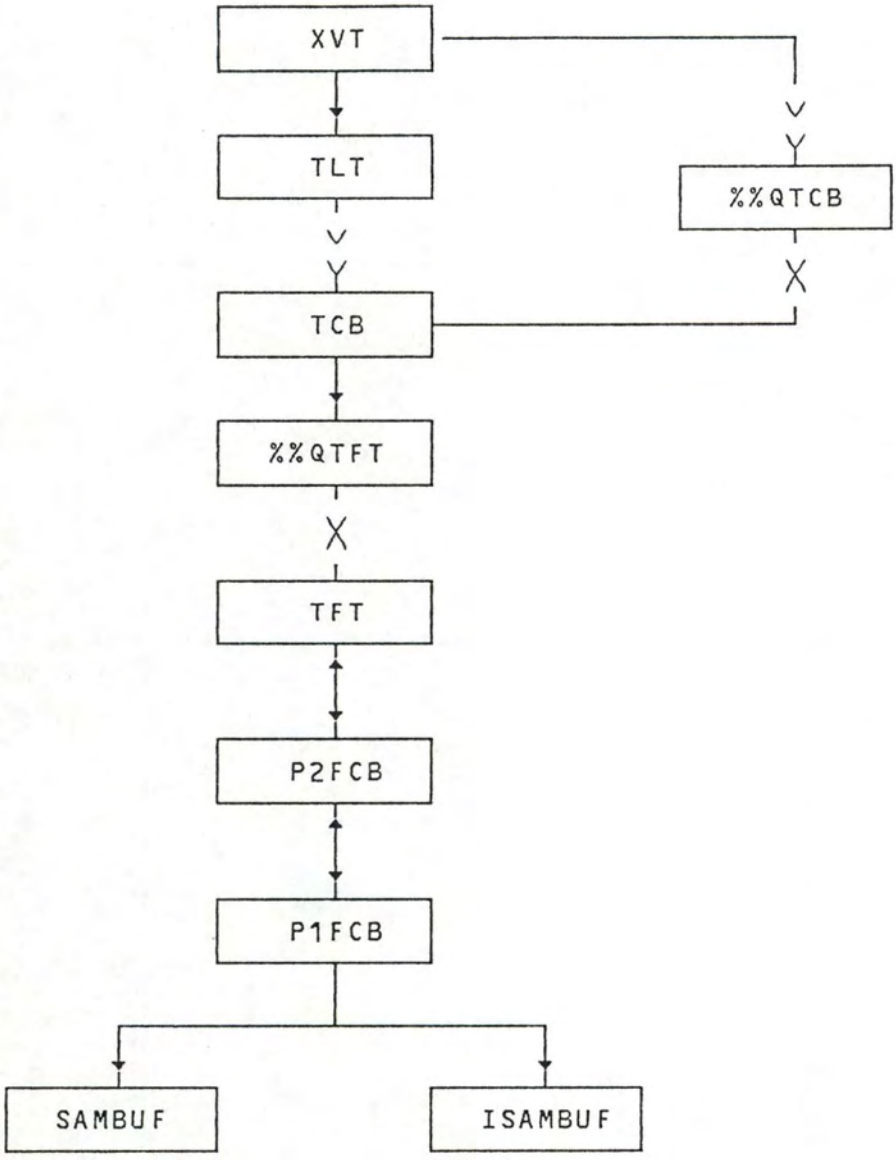
Le graphe formel est toujours supposé avoir une racine qui est un point d'entrée dans le graphe. Remarquons qu'en général, cette racine est la table XVT déjà citée.

2.3.4 Element NIL

Le graphe formel peut contenir un noeud fictif appelé NIL.Ce noeud permet de décrire un pointeur n'ayant pas d'extrémité.

2.3.5 Exemple de graphe formel

Reprenant les tables et les liens spécifiés en 2.2, nous obtenons le schéma suivant:


XVT
TLT
%%QTCB
TCB
%%QTFT
TFT
P2FCB
P1FCB
SAMBUF
ISAMBUF

Avec les significations suivantes :

- la XVT contient une référence simple vers la TLT (symbole >)

- la TLT contient (symbole >>) une référence vecteur vers la TCB

- la XVT contient (symbole >>) une référence vecteur vers un type de queues doubles appelées QTCB

- chaque queue double est composée (symbole ><) de TCB

- la TCB possède une référence vecteur vers un type de queues doubles appelées QTFT

- ce type de queues doubles est composé de TFT

- il y a une référence simple (possèdant un inverse) entre TFT et P2FCB, ainsi qu'entre P2FCB et P1FCB

- la P1FCB contient une référence multiple vers les types de tables SAMBUF et ISAMBUF.

Dans la description construite plus loin, chaque élément de ce graphe (tables, pointeurs...) sera représenté par un descripteur qui contiendra les attributs de l'élément.

2.4 Attributs des éléments du graphe formel

2.4.1 Attributs d'une table

- Nom : dans les cas (les plus nombreux) où la table est définie par une dsect, le nom (identifiant) de la table sera celui de sa dsect.

- Longueur : il peut s'agir d'une constante (par exemple pour une table en longueur fixe), d'un symbole ou encore de l'adresse d'une zone de la table qui contient sa longueur.

- Adresse de début : il s'agit d'un symbole servant d'adresse de base pour accéder aux éléments de la table.

- Unicité : détermine si la table existe en un seul exemplaire ou pas.

2.4.2 Attributs d'une queue simple

- Nom : identifiant le type de noeud (ou sommet). Ce nom doit être différent de tous les noms des autres types de noeud).

- Type des éléments formant la liste.

- Unicité : voir attribut d'une table.

- Définition du pointeur liant les tables de la liste. Cette définition contient :

    - le déplacement (offset) par rapport au début de la table

    - le type du pointeur (voir 2.4.6).

- Code représentant la fin de la liste.

2.4.3 Attributs d'une queue double

Ce type de noeud contient les mêmes attributs qu'une queue simple à la seule différence qu'il y a deux définitions de pointeur (un pointeur vers le précédent et un vers le suivant).

2.4.4 Attributs d'un anneau simple

Les attributs sont les mêmes que pour une queue simple.

2.4.5 Attributs d'un anneau double

Les attributs sont les mêmes que pour une queue double.

2.4.6 Attributs d'un pointeur

- Nom du noeud origine.
- Nom du noeud extrémité.
- Type : deux types généraux de pointeurs existent :
  - soit le pointeur est une adresse (sur deux, trois ou quatre bytes)
  - soit le pointeur est la longueur d'une table (voir 2.2.2).
- Déplacement de la zone contenant le pointeur par rapport au début de la table.
- Existence ou pas d'un inverse.

2.4.7 Attributs d'un pointeur multiple

- Nom du noeud origine.
- Déplacement.
- Type du pointeur (voir 2.4.6)
- Pour chaque choix :
  - nom du noeud extrémité
  - critère du choix (il s'agit d'une zone et d'une valeur).

2.4.8 Attributs d'un vecteur

- Nom du noeud origine.
- Nom du noeud extrémité.
- Nombre de composantes.
- Déplacement de la première composante par rapport au début de la table.
- Type des composantes (voir 2.4.6).

2.4.9 Attributs d'une partial page

- Nom du sous-type (voir 2.3.1)

2.5 Fonctionnement général de l'outil de parcours

2.5.1 Remarque

Ce paragraphe ne constitue en aucun cas une étude poussée du fonctionnement de l'outil de parcours. Il s'agit simplement de se convaincre que la représentation écrite dans ce travail permet un parcours dans un dump.

2.5.2 Spécifications

Etant donné un type de table et une liste de qualifications (numéro de composante d'un vecteur ,indice dans une liste...), rechercher dans un dump la (ou les) tables répondant aux qualifications et éventuellement donner le chemin parcouru pour y aboutir.

2.5.3 Notion de qualification

Les différentes qualifications que l'on peut donner pour accéder à une table peuvent concerner les vecteurs (numéro de composante) et les listes (premier, dernier,...). Ces éléments seront ainsi appelés des distributeurs.

Une qualification peut également être du type (symbole,valeur) ou (déplacement,valeur). Il s'agit alors d'accéder à telle table sachant que tel symbole (ou telle zone dont on donne le déplacement) doit prendre telle valeur pour l'élément recherché.

2.5.4 Description générale de l'algorithme de parcours

L'esprit de cette description est de justifier à priori la représentation des données du système définie dans le chapitre 4.

L'outil de parcours va devoir tout d'abord vérifier l'existence du type de table demandé (par exemple dans un dictionnaire des noeuds du graphe).

Il va ensuite rechercher tous les chemins permettant d'accéder à la table, par exemple en remontant le graphe à partir de ce type de table jusqu'à la racine du graphe (en général il s'agira de la XVT).

Ayant trouvé un certain nombre de chemins de la racine à la table cherchée, le programme peut alors chercher (s'il existe) un chemin répondant aux qualifications (par exemple en parcourant le graphe formel par les chemins trouvés plus haut).

Après cette étape, si un chemin a été trouvé, le programme devra alors parcourir ce chemin dans le dump en partant de la racine jusqu'à la ou les tables cherchées.

2.5.5 Deux conséquences importantes

2.5.5.1 Découpe du graphe en niveaux

De ce qui précède, on peut déduire qu'il semble utile de découper le graphe en niveaux. Bien que celui-ci possède des cycles, cela semble possible puisqu'il existe toujours une racine unique. L' intérêt de cette découpe est de faciliter la recherche d'un chemin d'une table vers la racine (empêcher le programme de boucler dans un cycle du graphe).

Les niveaux peuvent être définis de la manière suivante :

niv(racine) = 1

niv(A) = la longueur du chemin le plus court de la racine à A.

Le niveau est donc un attribut supplémentaire des noeuds du graphe.

2.5.5.2 Référence retour

Il semble également utile d'ajouter à la description d'un noeud de type A une liste des noeuds possèdant un pointeur vers A. Cette liste permettra une économie considérable lors de la recherche d'un chemin puisqu'à ce moment, on parcourt le graphe à l'envers. De telles références qui n'existent pas dans le dump lui-même, seront appelées références retour.

3 MANUEL DE L'UTILISATEUR : DESCRIPTION DES DECLARATIONS

3.1 Introduction

Avant de passer à la description de la réalisation physique du graphe, nous allons présenter la manière de déclarer les éléments de ce graphe.

En fait, pour pouvoir parcourir le dump, il est nécessaire de déclarer les éléments que l'on désire trouver dans le graphe formel. Il s'agit en fait de déclarer les données du système sur lesquelles on veut voir évoluer le programme de parcours. Cette déclaration est donc relativement statique.

L'idée de départ était de trouver un langage déjà implémenté dans lequel ces déclarations auraient été faites. L'essentiel du travail aurait pu alors se porter sur le programme de parcours. Malheureusement, il semble qu'un tel langage n'existe pas. Signalons en outre qu'un langage classique ne convient pas. Nous allons essayer d'expliquer pourquoi (la liste suivante n'étant pas exhaustive):

- les langages classiques (Pascal-like), permettent bien de déclarer des pointeurs, mais ceux-ci sont insuffisants pour dépeindre la structure elle-même. En effet, dans un langage classique, on dirait, par exemple, qu'une table du type A contient deux pointeurs vers des tables du même type plutôt que de parler de queues doubles. De plus, on ne pourrait pas remarquer la liaison entre la queue et son point d'ancrage. Par exemple, on dirait que le vecteur de la XVT pointe vers 13 TCB alors que ce qui est important ici c'est de dire qu'il pointe vers 13 queues de TCB.

- Ce genre de langage ne permet en général pas d'avoir des types différents de pointeurs (la notion de type de pointeurs n'étant pas liée ici l'objet pointt comme c'est le cas en Pascal, mais à la nature même du pointeur). Or nous avons vu que c'est le cas dans le système.

Il a donc été nécessaire d'implémenter une série de commandes de déclarations. Ces commandes ne forment pas réellement un langage, la syntaxe étant trop rigide. Elles permettent cependant de déclarer les différents éléments mis en évidence dans le graphe formel. On peut considérer une liste de telles commandes comme résultant du premier passage de la compilation d'un langage à définir.

Le programme décrit en annexe 1 reçoit une liste de telles commandes de déclarations et produit en sortie une représentation physique du graphe formel (cette représentation est décrite dans le chapitre 4).

Avant de donner un exemple de déclarations d'une partie des données du système (il s'agit du noyau spécifié en 2.2), nous allons passer en revue les différentes commandes de déclaration, leur syntaxe et leurs paramètres.

3.2 Déclaration d'un type de table

La commande est TABLE. Ses paramètres sont :

- NAME=<nom de la dsect> / <nom de la table>

  Si la table est décrite par une dsect, c'est par le nom de celle-ci que la table est identifiée. Sinon il s'agit d'un nom usuel de la table.

- DSECT=0 / 1

  La valeur 0 signifie que la table n'est pas décrite par une dsect, la valeur 1 signifiant le contraire.

- START=<start adress>

  Ce paramètre contient le symbole de début de la table. Il sera utilisé pour retrouver les éléments de la table (pour ces éléments , on donnera le déplacement par rapport au début de la table) Plusieurs cas sont possibles : si la table possède une dsect, ce paramètre contient le symbole de début de la dsect, sinon il contient 0.

- LENGTH=<valeur décimale> / <symbole> / '<déplacement>

  Ce paramètre peut être une constante (la valeur de la longueur elle-mêmê) ou le symbole contenant cette longueur. Il peut encore s'agir du déplacement (par rapport au début de la table) qui donne l'adresse de la zone de la table contenant la longueur (comme ce serait le cas pour les records SAM).

- LTYPE=L1 / L2 / L3 / L4

  Ce paramètre détermine la longueur de la zone contenant la longueur de la table. Il s'agit donc du type de la variable LENGTH. Notons que si la longueur est donnée par une constante (premier cas ci-dessus) ou par un symbole (second cas ci-dessus), ce type est indifférent.

- SINGLE=1 / 0

  Si la table existe en un seul exemplaire, alors SINGLE=1, sinon SINGLE=0.

Exemple de déclaration

```
TABLE NAME=DXVT,DSECT=1,START=EXVT,LENGTH=EXVTLEN,LTYPE=L2,SINGLE=1
```

Remarquons que l'ordre des paramètres est fixe.

3.3 Déclaration d'un type de pointeur simple

La commande est POINTER. Ses paramètres sont :

- FROM=<nom du noeud origine>

- TO=<nom du noeud extremit >/NIL

  La valeur spéciale NIL est permise pour indiquer que le pointeur ne contient l'adresse d'aucun noeud (ce cas est surtout intéressant pour les pointeurs multiples, car il permet d'indiquer dans quelle situation le pointeur n'existe pas).

- OFFSET=<valeur décimale> / <symbole>

  Cette adresse est soit un symbole soit un déplacement par rapport au début de la table.

- TYPE=A1 / A2 / A3 / A4 / L1 / L2 / L3 / L4

  Si le pointeur est une adresse, son type est A suivi de la longueur du pointeur ; si le pointeur est la longueur d'une zone, son type est L suivi de la longueur du pointeur.

- REVERSE=1 / 0

  Si le pointeur admet un inverse, alors REVERSE=1, sinon REVERSE=0.

Exemple de déclaration

```
POINTER FROM=DXVT,TO=TLT,OFFSET=EXVTLT,TYPE=A4,REVERSE=0
```

3.4 Déclaration d'un type de vecteur

La comnande est VECTOR. Ses paramètres sont :

- FROM=<nom du noeud origine>

  C'est le nom du noeud qui contient le vecteur.

- TO=<nom du noeud extremité>

- OFFSET=<valeur décimale> / <symbole>

  Il s'agit de l'adresse de la première composante (voir 3.3)

- TYPE=A1 / A2 / A3 / A4 / L1 / L2 / L3 / L4

  (Voir 3.3)

- RANGE=<valeur décimale>

  Nombre de composantes du vecteur.

Exemple de déclaration

```
VECTOR FROM=TLT,TO=DTCB,OFFSET=0,TYPE=A4,RANGE=256
```

3.5 Déclaration d'un type de pointeur multiple

Rappelons qu'un pointeur multiple représente une même zone qui peut référencer des noeuds différents selon le cas où on se trouve.

Pour chaque cas possible, il faut une commande de déclaration. Cette commande est MPOINTER. Certains de ces paramètres ont les mêmes valeurs pour tous les cas du type , d'autres prennent des valeurs différentes selon le cas déclaré. Les paramètres de cette commande sont :

- FROM=<nom du noeud origine>

  Ce champ est le même pour toutes les commandes relatives au même type.

- TO=<nom du noeud extrémité>

  Ce nom est différent selon le cas que l'on déclare.

- OFFSET= (voir 3.3)

  Remarquons qu'un type de pointeur multiple est identifié par les champs FROM et OFFSET.

- TYPE= (voir 3.3)

- REVERSE= (voir 3.3)

Suit alors la déclaration de la zone et de la valeur permettant de vérifier si on se trouve dans le cas déclaré ou pas. La syntaxe est :

- TEST=(NAME=<nom de table>,START=<start adress>,
OFFSET=<(voir 3.3)>,VALUE=<valeur>,TYPE=0 /1 /2 /3 /4,
MASK=<valeur du masque>)

  Le champ NAME détermine la table contenant la zone à tester. Les champs START et OFFSET déterminent l'adresse de la zone à tester et le champ VALUE la valeur que doit contenir cette zone pour être dans le cas déclaré plus haut. Si le champ TYPE est différent de 0, il détermine la longueur de la zone en bytes et VALUE doit être un entier. Si le champ TYPE vaut 0, cela signifie qu'il faut effectuer un test sur un certain nombre de bits. La valeur donnée par VALUE doit alors être constituée de 0 et de 1. Dans ce cas, le champ MASK donne la valeur du masque permettant d'effectuer ce test. Ces deux derniers champs doivent alors être constitués de 8 symboles '0' ou '1'.

Exemple de déclaration

```
MPOINTER FROM=IDFCB,TO=SAMBUF,OFFSET=ID1IOAR1,TYPE=A4,TEST=(NAME=
```

```
IDFCB,START=ID1CONAR,OFFSET=ID1IND2,VALUE=10000000,TYPE=0,
MASK=11000000)

MPOINTER FROM=IDFCB,TO=ISAMBUF,OFFSET=ID1IOAR2,TYPE=A4,TEST=(NAME=
IDFCB,START=ID1CONAR,OFFSET=ID1IND2,VALUE=110000000,TYPE=0,
MASK=11000000)
```

3.6 Déclaration d'une partial page

La commande est PARTIAL PAGE et ne possède qu'un seul paramètre :

- NAME= 'PP3' / 'PP4' / 'PP5'
  Suivant le sous-type considéré.

3.7 Déclaration d'un type de queue simple

La commande est SQUEUE. Ses paramètres sont :

- NAME=<nom de la queue>

  Ce nom doit être un identifiant dans l'ensemble des noeuds.

- OF=<nom des éléments>

  Il s'agit du nom des noeuds formant la queue.

- SINGLE=0 / 1

  (Voir 3.2)

Suivent alors les paramètres concernant le lien :

-ENDCODE=<valeur entiere>

  Ce paramètre contient la valeur indiquant la fin de la chaîne.

- START= (voir 3.2)

- OFFSET= (voir 3.3)

- TYPE= (voir 3.3)

Exemple de déclaration

L'exemple qui suit est une chaîne de CCMT (Comunications Controller Table). Ces tables contiennent des informations qui identifient un controller et sont en longueur fixe (28 bytes). S'il existe plus d'une telle table elles se suivent en mémoire, la dernière ayant la valeur X'FF' comme premier byte.

```
SQUEUE
NAME=QCCMT,OF=CCMT,SINGLE=1,ENDCODE=-1,START=0,OFFSET=0,TYPE=L1
```

3.8 Déclaration d'un type de queue double

La commande est DQUEUE. Ses paramètres sont :

- NAME=<nom de la queue>
- OF=<nom des éléments> (voir 3.7)
- SINGLE=0 / 1 (voir 3.2)

Suivent alors les paramètres concernant le lien en avant (Flink) et en arrière (Blink) :

- ENDCODE= (voir 3.7)
- START= (voir 3.2)
- OFFSETF= (voir 3.3)
- OFFSETB= (voir 3.3)
- TYPEF= (voir 3.3)
- TYPEB= (voir 3.3)

Exemple de déclaration

```
DQUEUE NAME=QTCB,OF=DTCB,SINGLE=0,ENDCODE=-1,START=ETCB,
OFFSETF=ETLLSPLIN,OFFSETB=ETLLPLIN,TYPEF=A4,TYPEB=A4
```

3.9 Déclaration d'un type d'anneau simple

La commande est SRING. Ses paramètres sont les mêmes que SQUEUE, sauf pour le ENDCODE qui n'existe pas dans ce cas.

3.10 Déclaration d'un type d'anneau double

La commande est DRING. Ses paramètres sont les mêmes que pour DQUEUE, sauf à nouveau pour le endcode qui n'existe pas.

3.11 Déclaration de la racine du graphe

Cette commande sert à déclarer quelle table est la racine du graphe. La commande est ROOT et ne possède qu'un seul paramètre :

- NAME=<nom de table>

Notons que la table qui est la racine doit être déclarée quelque part dans le graphe. Enfin, si cette commande n'existe pas dans la liste des déclarations, la table XVT est prise comme racine par défaut (elle doit donc , dans ce cas, être déclarée dans la liste des déclarations).

3.12 Exemple de déclaration d'une partie des données du système

Effectuons la déclaration des données spécifiées en 2.3. Remarquons d'abord que l'ordre des déclarations n'a aucune importance.

```
TABLE NAME=DXVT,DSECT=1,START=EXVT,LENGTH=EXVTLEN,LTYPE=L2,SINGLE=1
*
TABLE NAME=TLT,DSECT=0,START=0,LENGTH=1024,LTYPE=L1,SINGLE=1
*
TABLE NAME=DTCB,DSECT=1,START=ETCB,LENGTH=ETCBLEN,LTYPE=L2, SINGLE=0
*
TABLE NAME=IDTFT,DSECT=1,START=IDMFRLNK,LENGTH=IDMLFLEN,
LTYPE=L2,SINGLE=0
*
TABLE NAME=IDFCB,DSECT=1,START=ID1CONAR,LENGTH=ID1FCBSZ,
LTYPE=L2,SINGLE=0
*
TABLE NAME=IDFC2,DSECT=1,START=ID2CFLID,LENGTH=ID2FCBSZ,
LTYPE=L2,SINGLE=0
*
TABLE NAME=ISAMBUF,DSECT=0,START=0,LENGTH=2048,LTYPE=L1, SINGLE=0
*
TABLE NAME=SAMBUF,DSECT=0,START=0,LENGTH=2048,LTYPE=L1, SINGLE=0
* fin des déclarations des tables
DQUEUE NAME=QTCB,OF=DTCB,SINGLE=0,ENDCODE=-1,START=ETCB,
OFFSETF=ETLLSPLIN,OFFSETB=ETLLPLIN,TYPEF=A4,TYPEB=A4
*
DQUEUE NAME=QTFT,OF=IDTFT,SINGLE=0,ENDCODE=X'FF',START=IDMFRLNK,
OFSETF=IDMFRLNK,OFFSETB=IDMVLNK,TYPEF=A4,TYPEB=A4
*
POINTER FROM=DXVT,TO=TLT,OFFSET=EXVTLT,TYPE=A4,REVERSE=0
*
POINTER FROM=DTCB,TO=QTFT,OFFSET=ETCBFIL,TYPE=A4,REVERSE=0
*
POINTER FROM=IDTFT,TO=IDFC2,OFFSET=IDMP2FL,TYPE=A4,REVERSE=1
*
POINTER FROM=IDFC2,TO=IDTFT,OFFSET=ID2TFTLK,TYPE=A4,REVERSE=1
*
POINTER FROM=IDFC2,TO=IDFCB,OFFSET=ID2P1LNK,TYPE=A4,REVERSE=1
*
POINTER FROM=IDFCB,TO=IDFC2,OFFSET=ID1P2LNK,TYPE=A4,REVERSE=1
*
VECTOR FROM=DXVT,TO=QTCB,OFFSET=ESQ1,TYPE=A4,RANGE=13
*
VECTOR FROM=TLT,TO=DTCB,OFFSET=0,TYPE=A4,RANGE=256
*
MPOINTER FROM=IDFCB,TO=SAMBUF,OFFSET=ID1IOAR1,TYPE=A4,TEST=(NAME=
IDFCB,START=ID1CONAR,OFFSET=ID1IND2,VALUE=11000000,TYPE=0,
MASK=11000000)
*
MPOINTER FROM=IDFCB,TO=ISAMBUF,OFFSET=ID1IOAR2,TYPE=A4,TEST=
(NAME=IDFCB,START=ID1CONAR,OFFSET=ID1IND2,VALUE=10000000,
TYPE=0,MASK=110000000)
```

3.13 Commentaires

3.13.1 Syntaxe des commandes

La syntaxe des déclarations étant très stricte, on ne peut pas parler réellement d'un langage de déclaration.

Pour créer un vrai langage, il aurait fallu implémenter un compilateur. Avec des commandes de ce type, la traduction des déclarations est d'un ordre de difficulté moindre. Les spécifications du programme de l'annexe 1 sont d'ailleurs de traiter une liste de déclarations telle que décrite en 3.11. Si cette liste ne contient pas d'erreur, le programme produit alors une représentation du graphe, sinon il édite la liste des erreurs rencontrées.

Notons que le programme de traduction admet des blancs avant ou après les séparateurs ',', '=', '(' et ')', de même qu'en début de ligne. En outre, il faut au moins un symbole blanc entre le nom de la commande et la liste des paramètres.

Des lignes de commentaires peuvent être insérées à n'importe quel endroit, la ligne correspondante devant commencer par le symbole '*'.

Enfin, chaque commande doit se trouver sur une seule ligne (au maximum 256 caractéres).

3.13.2 Liste des erreurs de déclarations

Analysons les erreurs que l'on peut rencontrer en traitant une liste de déclarations.

3.13.2.1 Erreurs de syntaxe

Il s'agit de toutes les commandes qui ne respectent pas la syntaxe décrite en 3.2,...,3.10.

3.13.2.2 Noeuds déclarés deux fois

Nous l'avons déjà signalé, les noms des noeuds doivent être identifiants. Deux noeuds différents ne peuvent donc avoir le même nom.

3.13.2.3 Référence sans extrémité

Le champ TO d'une référence (pointeur, vecteur ou pointeur multiple) doit être un noeud déclaré dans la liste.

3.13.2.4 Référence sans origine

Le champ FROM d'une référence doit être déclaré.

3.13.2.5 Liste sans éléments

Les éléments constituant la liste (champ OF d'une queue simple, d'une queue double, d'un anneau simple ou d'un anneau double) doivent être déclarés.

3.13.2.6 Noeuds inaccessibles

Un noeud inaccessible est inutile dans cette description. En général, ce problème résulte d'un pointeur que l'on a oublié de déclarer.

3.13.2.7 Pointeurs inversibles sans inverse

Si un pointeur a été déclaré inversible (REVERSE=1), son inverse doit être déclaré.

3.13.2.8 Erreur de masque ou de valeur de test

Rappelons que, pour les pointeurs multiples, si le champ TYPE (du test) est 0, les champs MASK et VALUE doivent être constitués de 8 symboles '0' ou '1'.

3.13.2.9 Table contenant une zone à tester non déclarée

Nous avons déjà signalé que lorqu'un test doit être effectué pour une référence multiple, la table contenant la zone sur laquelle le test doit être effectué doit exister dans le graphe.

3.13.2.10 Racine non déclarée

Si une racine a été déclarée explicitement, la table considérée doit être déclarée quelque part. Si aucune racine n'est déclarée, la table XVT (prise par défaut comme racine), doit être déclarée obligatoirement.

4 REPRESENTATION PHYSIQUE DU GRAPHE

4.1 Introduction

L'objet de ce chapitre est de décrire aussi précisément que possible la représentation physique du graphe formel.

Si certains détails peuvent paraître superflus, il faut garder en mémoire que le concepteur de l'outil de parcours devra utiliser cette représentation. Il est donc nécessaire que la description qui suit soit aussi précise que possible.

Avant de passer à une description exhaustive de la représentation des différents éléments du graphe, nous allons décrire l'architecture globale de la représentation.

4.2 Architecture globale de la représentation

Chaque élément du graphe formel, noeud ou arc, est représenté par un descripteur. Un descripteur est une table très restreinte qui contient les informations nécessaires au programme de parcours.

En outre, un dictionnaire des noeuds est implémenté de manière à vérifier facilement l'existence d'un noeud de nom donné et à retrouver rapidement le descripteur relatif à un noeud quelconque.

Les différents descripteurs sont liés les uns aux autres. Ainsi, par exemple, si, dans le graphe formel, il existe une référence simple de la table A vers la table B, cela implique une déclaration de deux types de tables (A et B) et d'un type de pointeurs ayant A comme valeur de champ FROM et B comme valeur de champ TO. Dans la représentation du graphe formel, il existe alors quatre descripteurs : un pour le type de tables A, un pour le type de tables B, un pour la référence simple de A vers B et un pour la référence retour de B vers A. En outre, le desrcipteur de la table A contiendra un pointeur vers la liste des descripteurs de ses références simples et le descripteur de la référence simple déclarée plus haut, contiendra un pointeur vers le descripteur de la table B.

L'architecture générale de la représentation se présente donc comme suit :

- Chaque élément du dictionnaire contient le nom du noeud et l'adresse du descripteur de ce noeud.

- Chaque descripteur de noeud contient l'adresse de l'élément correspondant dans le dictionnaire et les adresses de ses différentes listes de références. Ainsi, pour une table, il existe quatre listes de références : une liste de références simples, une liste de références multiples, une liste de références vecteur et une liste de références retour.

- Chaque descripteur d'un arc (référence) contient l'adresse du descripteur du noeud pointé et l'adresse du descripteur d'arc suivant dans la liste.

- Un point d'entrée (entry point) est prévu pour accéder au descripteur de la racine du graphe.

Après cet aperçu de l'architecture générale, nous allons détailler chaque type de descripteur.

Rappelons avant toute chose que nous appelons référence retour la représentation d'un pointeur retour (ces pointeurs n'existent pas dans le système, mais permettent un parcours "à l'envers" du graphe).

4.3 Descripteurs des éléments du graphe formel

4.3.1 Descripteur d'une table

La longueur du descripteur est de 28 bytes. Celui-ci contient:

- un code qui identifie le type d'élément représenté.

  Ce code est implémenté sur 2 bytes. Trois critères déterminent sa valeur, l'existence d'une dsect qui décrit la table, le fait que la table existe en un seul exemplaire et le fait que sa longueur soit donnée par une valeur (ou un symbole) ou un déplacement (comme c'est le cas pour les records SAM). La table suivante donne les différentes valeurs possibles :

Dsect	Y	Y	Y	Y	N	N	N	N
Unicite	Y	Y	N	N	Y	Y	N	N
Longueur	C	D	C	D	C	D	C	D
Code	1	2	3	4	5	6	7	8

Remarquons que ce champ est un identifiant du type de descripteur dans l'ensemble de tous les types de descripteurs.

- niveau de la table implémenté sur les 2 bytes suivants (voir 2.5.5.1)

- adresse de l'élément du dictionnaire correspondant à cette table (4 bytes)

- longueur de la table (sur 8 bytes).

  Si la longueur est déclarée par un symbole ou une constante (C), ce champ contient la longueur de la table elle-mêmé (sur 4 bytes) et est suivi de 4 bytes a 0. Sinon, les 4 premiers bytes contiennent l'adresse de la zone contenant la longueur suivis du type sur 2 bytes (L1, L2, L3 ou L4) et de 2 bytes a 0.

Suivent alors les pointeurs vers les descripteurs des éléments du graphe liés à cette table :

- adresse de la liste des descripteurs de références simples appartenant à la table (4 bytes) ; il s'agit de l'adresse du premier élément de cette liste

- adresse de la liste des descripteurs de références vecteur appartenant à la table (4 bytes) ; il s'agit de l'adresse du premier élément de cette liste

- adresse de la liste des descripteurs de références multiples appartenant à la table (4 bytes) ; il s'agit de l'adresse du premier élément de cette liste

- adresse de la liste des descripteurs de références retour appartenant à la table (4 bytes) ; il s'agit de l'adresse du premier élément de cette liste

Notons une fois pour toutes que pour chaque liste de descripteurs existant dans cette architecture, le pointeur vers le suivant contient la valeur -1 s'il s'agit du dernier élément de la liste.

Pour une meilleure compréhension de la suite, nous poursuivons par le descripteur d'une référence simple.

4.3.2 Descripteur d'une référence simple

Le descripteur a une longueur de 20 bytes. Notons que les descripteurs de références simples d'une même table sont chaînés.

Le descripteur contient les champs suivants :

- un code; celui-ci est déterminé par un seul critère: existence ou pas d'un inverse. Les valeurs de ce code sont les suivantes: 30 si un inverse existe, 31 sinon.
- type du pointeur code sur 2 bytes (voir 3.3)
- offset du pointeur code sur 4 bytes.

Suivent alors les pointeurs vers d'autres descripteurs :

- adresse du descripteur de la table contenant la référence sur 4 bytes (champ FROM de la déclaration)
- adresse du descripteur du noeud extremité de l'arc sur 4 bytes (champ TO de la déclaration)
- adresse du descripteur de la référence simple suivante concernant cette table sur 4 bytes.

4.3.3 Descripteur d'une queue simple

La longueur du descripteur est de 28 bytes. Celui-ci contient:

- un code sur 2 bytes identifiant le type de descripteur.

Les valeurs de ce code sont :

10 si la queue existe en un seul exemplaire

11 sinon

- niveau de ce noeud dans le graphe (2 bytes)
- adresse de l'élément du dictionnaire correspondant à ce noeud (4 bytes). Cet élément contient le nom du noeud.
- adresse du descripteur du noeud constituant la queue (4 bytes)
- 2 bytes perdus
- le type du lien (voir type d'un pointeur), sur 2 bytes
- l'offset du lien (voir offset d'un pointeur), sur 4 bytes.
- la valeur correspondant à la fin de la queue, sur 4 bytes

Suit alors l'adresse de la liste des descripteurs liés à celui-ci :

- adresse de la liste de descripteurs des références retour de cette queue (4 bytes).

Il n'y a pas d'autre liste de descripteurs associée à ce type de noeud puisqu'il ne contient ni référence simple, ni référence vecteur, ni référence multiple.

4.3.4 Descripteur d'une queue double

Sa longueur est de 36 bytes. Il n'y a pas d'élément conceptuellement nouveau dans ce descripteur par rapport au précédent. Présentons rapidement son contenu :

- code : 12 si unicité, 13 sinon (sur 2 bytes)
- niveau (2 bytes)
- adresse de l'élément du dictionnaire de la queue (4 bytes)
- adresse du descripteur du noeud constituant la queue (4 bytes)
- type (2 bytes) du lien en avant
- type (2 bytes) du lien en arrière
- offset du lien en avant (4 bytes)
- offset du lien en arrière (4 bytes)
- code de fin de queue
- adresse de la liste des références retour de la queue (4 bytes)

4.3.5 Descripteur d'un anneau simple

Ce descripteur d'une longueur de 24 bytes contient les informations suivantes :

- code sur 2 bytes (14 si unicité, 15 sinon)
- niveau de ce noeud (2 bytes)
- adresse de l'élément du dictionnaire de l'anneau (4 bytes)
- adresse du descripteur de l'élément constituant l'anneau (4 bytes)
- 2 bytes perdus
- type du lien (2 bytes)
- offset du lien (4 bytes)
- adresse de la liste des références retour de l'anneau (4 bytes).

4.3.6 Descripteur d'un anneau double

ce type de descripteur, d'une longueur de 32 bytes contient les informations suivantes :

- code sur 2 bytes (16 si unicité, 17 sinon)
- niveau de ce noeud sur 2 bytes
- adresse de l'élément du dictionnaire de ce noeud (4 bytes)
- adresse du descripteur de l'élément constituant l'anneau (4 bytes)
- type du lien en avant (4 bytes en tout)
- type du lien en arrière (4 bytes en tout)
- offset du lien en avant (4 bytes)
- offset du lien en arrière (4 bytes)
- adresse de la liste des descripteurs des références retour (4 bytes)

4.3.7 Descripteur d'une partial page

Ce descripteur de 12 bytes contient :

- code sur 2 bytes (valeur 20)
- niveau (2 bytes)
- adresse de l'élément du dictionnaire de ce noeud
- adresse de la liste des descripteurs des références retour

4.3.8 Descripteur d'une référence vecteur

Ce descripteur de 24 bytes contient :

- un code (2 bytes). La valeur de ce code est 40.
- le type des composantes, sur 2 bytes (voir 3. 4)
- l'offset de la première composante du vecteur (4 bytes).
- le nombre de composantes du vecteur (4 bytes)
- l'adresse du descripteur du noeud contenant la référence, sur 4 bytes (champ FROM de la déclaration)
- l'adresse du descripteur du noeud extrémité de l'arc, sur 4 bytes (champ TO de la déclaration)
- l'adresse du descripteur de référence vecteur suivant concernant cette table (4 bytes).

4.3.9 Descripteur d'une référence multiple

Comme pour les autres références, il existe pour chaque table une liste des descripteurs des références multiples. Chaque descripteur d'une référence multiple est constitué d'un en-tête reprenant les valeurs constantes pour la référence et d'une liste dont chaque élément décrit un choix pour cette référence multiple. La longueur du descripteur est donc variable dans ce cas.

L'en-tête contient les champs suivants :

- un code sur 2 bytes (valeur 50).
- 2 bytes perdus
- offset (voir 4.3.2) sur 4 bytes
- adresse du descripteur du noeud contenant la référence (4 bytes). C'est le champ FROM de la déclaration.
- adresse de la liste des descripteurs des différents choix déclarés (4 bytes)
- adresse du descripteur de la référence multiple suivante dans la liste des descripteurs de références multiples de la table

Pour chaque choix déclaré, on a le descripteur suivant :

- code de la référence sur 2 bytes (voir code d'une référence simple)
- type de la référence sur 2 bytes (voir type d'une référence simple)
- adresse du descripteur du noeud extrémité de l'arc (champ TO de la déclaration) sur 4 bytes
- code identifiant le type de test à effectuer sur 2 bytes.

  Les valeurs possibles sont :
  0 si test avec masque
  1 si test sur 1 byte
  2 si test sur 2 bytes
  3 si test sur 3 bytes
  4 si test sur 4 bytes

- valeur du masque (1 byte). Ce masque n'est utile que si le code du test est 0.
- 1 byte perdu
- adresse du descripteur de la table contenant la zone à tester (4 bytes).
- offset relatif à la zone à tester (4 bytes)

- valeur de cette zone (4 bytes).
  Si le code du test est 0 (test avec masque) seul le premier byte est significatif.

- adresse du descripteur du choix suivant pour cette référence multiple (4 bytes).

4.3.10 Descripteur d'une référence retour

Le descripteur d'une référence retour, long de 16 bytes, contient les informations suivantes :

- un code sur 2 bytes.
  Ce code identifie le type de référence dont ce descripteur est un inverse logique. Il peut s'agir d'une référence simple (code 60), d'une référence vecteur (code 61) ou d'une référence multiple (code 62). En outre, lorsqu'on dit qu'une queue est composée d'un certain type de tables, on déclare un arc entre la queue et la table. Une référence retour sera donc également créée. Les valeurs du code sont dans ce cas :

  63 si référence retour d'une queue simple
  64 si référence retour d'une queue double
  65 si référence retour d'un anneau simple
  66 si référence retour d'un anneau double

Remarquons que dans la situation décrite plus haut, le descripteur de la référence retour appartient à la table et pas à la queue.

- 2 bytes perdus.

-l'adresse du descripteur du noeud contenant la référence (4 bytes). Si la référence avait une existence physique, ce serait le descripteur de la table qui la contiendrait.

- l'adresse du descripteur du noeud pointé (4 bytes)

- l'adresse du descripteur suivant dans la liste des descripteurs des références retour du noeud.

4.3.11 Elément de dictionnaire

Pour chaque noeud déclaré, le dictionnaire contient le nom du noeud (8 bytes) et l'adresse du descripteur du noeud (4 bytes).

Bien entendu, le dictionnaire est trié par ordre alphabétique sur le nom des noeuds.

La zone contenant le dictionnaire commence au symbole BEGDIC et se termine au symbole ENDIC (déclarés comme ENTRY).

4.3.12 Racine du graphe

L'adresse du descripteur de la racine du graphe est située à l'adresse déterminée par le symbole ROOT (déclaré comme ENTRY).

4.3.13 Elément NIL

Cet élément fictif est composé de 4 bytes contenant la valeur 0.

5 EXEMPLE DE LANGAGE DE DECLARATIONS

5.1 Introduction

Ce chapitre propose un exemple de grammaire d'un niveau plus élevé pour un langage permettant de déclarer les structures de données du système. Malheureusement, l'implémentation d'un compilateur pour ce langage ne semble pas un objectif réaliste dans le cadre d'un mémoire. La syntaxe est donc simplement décrite à titre d'exemple.

5.2 Description du langage

Le "programme" de déclaration se compose d'une série de déclarations:

```
<program> ::= BEGIN <declation> {; <declaration> } END.
```

Une déclaration est soit la déclaration d'une table, d'un pointeur, d'une liste, d'un élément itératif ou d'un élément alternatif:

```
<declaration> ::= <table-decl> / <ptr-decl> /
                  <queue-decl> / <iterative-decl> /
                  <alternative-decl>
```

La syntaxe de la déclaration d'une table est la suivante:

```
<table-decl> ::= TABLE <table-name>,LENGTH <length-decl>,
                 START <symbol> [ IS <table-description> ]
```

Le nom de la table est en général celui de dsect (ou un nom usuel si la table n'est pas décrite par une dsect). Pour la déclaration de la longueur, il s'agit bien sur d'une valeur, d'un symbole ou d'un déplacement par rapport au début de la table (voir 3.2). Le paramètre START est comme auparavant, le symbole de début de la table (voir 3.2).

La description des éléments de la table peut être faite au sein de la déclaration de la table ou en dehors de celle-ci.

```
<table-description> ::= BEGIN <table-elt> {<table-elt>} END

<table-elt> ::= <ptr-decl> / <iterative-decl> /
                <alternative-decl> / <queue-decl>
```

La déclaration d'un pointeur est faite de la manière suivante:

```
<ptr-decl> ::= POINTER ([<origin>,]<extremity>)
               <type-decl> AT <offset>
```

<type-decl> ::= (<ptr-type>,<ptr-length>)

<ptr-type> ::= A / L (voir 3.3)

<ptr-length> ::= 1 / 2 / 3 / 4

Remarquons que la déclaration de l'origine du pointeur est facultative, dans le cas où la déclaration du pointeur est faite au sein de la déclaration de la table laquelle il appartient.

La syntaxe d'une déclaration itérative est la suivante:

```
<iterative-decl> ::= ITERATE <declaration>
                     <integer> TIMES / UNTIL <condition>
                     END
```

Ce type de déclaration permet de définir des vecteurs de pointeurs, mais également des vecteurs de pointeurs multiples ou encore des vecteurs de tables.

La déclaration des pointeurs multiples est également enrichie par rapport à la commande décrite dans le chapitre 3. Sa syntaxe est:

```
<alternative-decl> ::= EITHER <declaration> ON <condition>
                       OR <declaration> ON <condition>
                       { OR <declaration> ON <condition> }
                       END
```

La déclaration d'une condition est:

<condition> ::= (<area>,<value>) / (<area>,<value>,<mask>)

<area> ::= (<table-name>,<offset>)

<value> ::= <integer> / X'<hexadecimal>' / B'<binary>'

<mask> ::= B'<binary>'

Pour plus de détails, voir 3.5.

Enfin, il reste à décrire la syntaxe de la déclaration d'une queue:

```
<queue-decl> ::= QUEUE <queue-name> <qtype> OF <table-name>
                 USING <link-decl>
```

<qtype> ::= (SQUEUE) / (DQUEUE) / (SRING) / (DRING)

<link-decl> ::= <link-elt> / (<link-elt>,<link-elt>)

<link-elt> ::= (<table-name>,<offset>)

Bien entendu, la déclaration du lien ne contient qu'un seul élément dans les déclarations d'une queue ou d'un anneau simple et contient deux éléments dans les déclarations d'une queue ou d'un

anneau double.

5.2.1 Exemple de déclarations

Nous allons illustrer la syntaxe par la définition du noyau des données du système. Ce noyau a déjà été spécifié dans le chapitre 2.

```
BEGIN
 TABLE dxvt,LENGTH exvtlen,START exvt
  IS
   BEGIN
    POINTER (tlt) (a,4) AT exvtlt
    ITERATE POINTER (qtcb) (a,4) AT esq1
     13 TIMES
    END (* iterate *)
   END (* dxvt *)

 TABLE tlt,LENGTH 1024,START 0
  IS
   BEGIN
    ITERATE POINTER (dtcb) (a,4) AT 0
     256 TIMES
    END (* iterate *)
   END (* tlt *)

 TABLE dtcb,LENGTH etcblen,START ETCB
  IS
   BEGIN
    POINTER (qtft) (a,4) AT etcbfil
   END (* dtcb *)

 QUEUE qtcb (dqueue) OF dtcb USING
  ( (dtcb,etllsplin),(dtcb,etllplin) )

 QUEUE qtft (dqueue) OF idtft USING
  ( (idtft,idmfrlnk),(idtft,idmvlnk) )

 TABLE idtft,LENGTH idmflen,START IDMFRLNK
  IS
   BEGIN
    POINTER (idfc2) (a,4) AT IDMP2FL
   END (* idtft * )

 TABLE idfc2,LENGTH id2fcbsz,START id2cflid
  IS
   POINTER (idtft) (a,4) AT idmp2fl
   POINTER (idfcb) (a,4) AT id2p1lnk
  end (* idfc2 *)

 TABLE idfcb,LENGTH fcbsz,START id1conar
  IS
   POINTER (idfc2) (a,4) AT id1p2lnk
   EITHER
     POINTER (sambuf) (a,4) AT id1ioar1
        ON ((idfcb,id1ind2),b'10000000',b'11000000')
   OR
```

```
    POINTER (isambuf) (a,4) AT id1ioar1
       ON ((idfcb,id1ind2),b'11000000',b'11000000')
   END (* or *)
  END (* idfcb *)

 TABLE sambuf,LENGTH 2048,START 0

 TABLE isambuf,LENGTH 2048,START 0
END (* program *)
```

6 CONCLUSION ET EXTENSIONS

Ce travail, comme nous l'avons déjà signalé, ne constitue pas une fin en soi, mais une première étape. Un langage de déclarations des données du système ayant été implémenté, il est maintenant possible de passer à l'étape suivante, c'est à dire l'écriture d'un outil de parcours dans un dump.

Signalons tout d'abord que certains tests ont été effectués hors mémoire. Ces tests portaient sur la possibilité de parcourir un chemin physique (dans un dump) connaissant le chemin logique (dans la représentation du graphe formel). Ces programmes peuvent constituer une aide pour la conception de la seconde étape.

Cet outil de parcours devra être capable de représenter des qualifications (concernant la table cherchée ou des tables intermédiaires) entrées par l'utilisateur, de chercher un chemin logique à l'aide de ces qualifications et, enfin, de parcourir le chemin physique en s'aidant du chemin logique.

Un autre domaine dans lequel peuvent s'orienter les recherches futures concerne une série d'améliorations qualitatives liées au présent travail. Ce travail ayant necessité de longs mois de défrichements, il n'est pas impossible que, profitant de ce fait, on puisse maintenant implémenter un langage de plus haut niveau pour déclarer les structures de données.

Signalons enfin que ces deux pistes peuvent être suivies en parallèle, tant que la représentation physique du graphe formel reste inchangée.

ANNEXE

TABLE DES MATIERES DE L'ANNEXE

IMPLEMENTATION DE L'OUTIL DE TRADUCTION DES DECLARATIONS

1 Architecture logique : découpe en niveaux

L'architecture logique se compose de trois niveaux liés entre eux par une relation "utilise".

1.1 Niveau 1 : analyseur syntaxique

Spécifications : contrôler la syntaxe des déclarations.

Au dessus de ce niveau, la notion de syntaxe n'apparaît plus. Au sein de ce niveau, ni la structure du graphe formel, ni la notion d'erreurs autres que syntaxique ne sont connues.

1.2 Niveau 2 : analyseur sémantique

Spécifications : produire une liste de macro-instructions assemblables ou produire la liste des erreurs de déclarations rencontrées.

Au dessus de ce niveau, la notion d'erreur de déclaration n'existe plus. Au sein de ce niveau, la structure de la représentation du graphe formel est connue.

1.3 Niveau 3 : construction de la représentation du graphe formel

spécifications : construire la représentation du graphe formel par assemblage de macro-instructions et calculer le niveau des noeuds du graphe formel.

Au sein de ce niveau, la notion d'erreur de déclaration n'existe plus et la structure de la représentation est connue.

1.4 Remarque

Une relation d'importation/exportation existe du niveau 1 vers le niveau 2 et du niveau 2 vers le niveau 3.

2 Structuration modulaire

2.1 Niveau 1

Ce niveau est décomposé en 2 modules liés entre eux par une relation d'importation/exportation.

2.1.1 Module de traitement syntaxique des déclarations

2.1.1.1 Spécifications

Arguments:
la liste des déclarations correspondant au graphe formel.

résultats:
ce module produit les listes suivantes:

- liste des tables déclarées (et leurs paramètres)
- liste des queues simples déclarées (et leurs paramétres)
- liste des " doubles déclarées (")
- liste des anneaux simples déclarés (")
- liste des " doubles déclarés (")
- liste des pages partielles déclarées (")
- liste des références simples déclarées (")
- liste des références vecteur déclarées (")
- liste des références multiples déclarées (")
- liste des références retour (et leurs paramétres)
- liste de tous les noeuds déclarés
- liste des erreurs de syntaxe

2.1.1.2 Design

Ce module est constitué d'une série de commandes de l'éditeur EDOR travaillant sur une copie des déclarations.

Chaque commande permet de retrouver les occurrences d'un type de déclaration (syntaxiquement correcte) pour un élément du graphe formel, place les paramètres de ces déclarations dans la liste correspondante (implémentée en fichier séquentiel) et efface les éléments considérés.

Un prétraitement des déclarations permet avec des commandes du même type, d'effacer les caractères blancs inutiles et les lignes de commentaires.

Les éléments restant dans la copie des déclarations sont syntaxiquement incorrects.

Exemple

Rechercher les occurrences de la commande de déclaration des tables :

```
D$,F.TAB!  (* ouverture du ficier résultat *)
SF <1> *(' '),'TABLE',*(' '),'NAME=',*(-',').@A,',',
       'DSECT=','0'/'1'.@B,',','START=',*(-',').@C,',',
       'LENGTH=',*(-',').@D,',','LTYPE=','L1'/'L2'/'L3'/'L4'.@F,
       ',',','SINGLE=','0'/'1'.@E:FE;
       $=@A8,@B1,@C8,@D9,@F2,@E1;
   <1> * = ''!  (* recherche et éffacement des tables *)
H$!--!  (* fermeture du fichier et repositionnement *)
```

2.1.2 Module de tri des listes produites par le module précédent

Ces tris sont effectués par un utilitaire du système. Les clés de tri sont:

- champ NAME pour toutes les listes de noeuds
- champ NAME pour le dictionnaire
- champ FROM pour les références simples et vecteur
- champ TO pour les références retour
- champs FROM et OFFSET pour les références multiples.

2.2 Niveau 2

2.2.1 Architecture du niveau 2

Le niveau 2 est structuré de la manière suivante :

2.2.1.1 Module de traitement des noeuds

Ce module utilise les modules suivants:

- traitement des références
- traitement des erreurs détectées
- accès au dictionnaire
- lecture
- production des macro-instructions
- séquenceur

2.2.1.2 Module de traitement des références

Ce module utilise les modules suivants :

- traitement des erreurs
- accès au dictionnaire
- lecture
- production des macro-instructions
- séquenceur

2.2.1.3 Module de traitement des erreurs détectées

Ce module utilise le module séquenceur

2.2.1.4 Module d'accès au dictionnaire

2.2.1.5 Module de lecture

2.2.1.6 Module de production des macro-instructions

2.2.1.7 Module d'initialisation/clôture

Ce module utilise le module séquenceur.

2.2.1.8 Module séquenceur

2.2.2 Conception des différents modules

2.2.2.1 Traitement des noeuds

Spécifications

Arguments : un noeud et ses attributs.

Résultats : - liste des erreurs éventuelles liées à ce noeud
- production des macros instructions relatives à ce noeud et à ses références.

2.2.2.2 Traitement des références

2.2.2.2.1 Spécifications

Arguments : - un noeud et ses attributs
- la liste des références déclarées

Résultats : - liste des erreurs éventuelles
- liste des macro-instructions relatives aux références du noeud.

2.2.2.2.2 Design

Pour chaque type de références la structure des données se présente comme suit :

lors du traitement d'un noeud N on a (en prenant l'exemple des références simples) :

- soit une liste de références simples dont l'origine est N
- soit une liste non vide de références simples dont l'origine n'est pas déclarée suivie d'une liste de références simples dont l'origine est N.

En outre, il y a équivalence avec la structure des résultats (une macro instruction par référence simple rencontrée ou une liste des erreurs suivie d' une macro instruction par référence rencontrée).

L'algorithme qui traite les références simples (ainsi que les autres types de références) est calqué sur ces structures de données.

2.2.3 Traitement des erreurs rencontrées

Spécifications :

Arguments : un type d'erreur et les éléments liés à cette erreur

Résultats : mémorisation de l'erreur.

2.2.4 Accès au dictionnaire

2.2.4.1 Spécifications

- Soit arguments : nom d'un noeud

  résultats : vrai si le nom appartient au dictionnaire, faux sinon

- Soit permet un parcours séquentiel du dictionnaire.

2.2.4.2 Design

L'algorithme de recherche dichotomique utilisé a été démontré dans le cadre du cours de Méthodologie de développement de logiciels de M. Van Lamsweerde. Il a été implémenté en Assembler pour des raisons de performances.

2.2.5 Lecture

2.2.5.1 Spécifications

Arguments : type d'élément du graphe formel

Résultats : l'élément suivant de la liste concernre.

2.2.5.2 Design

Chaque liste d'élément du graphe formel constitue un fichier séquentiel.

2.2.6 Production des macro-instructions

2.2.6.1 Spécifications

Arguments : un type d'élément du graphe formel et ses attributs

Résultats : production et mise en page d'une macro instruction relative à cet élément.

2.2.6.2 Design

La liste des macro-instructions produites constitue un fichier séqentiel.

L'ordre de ces instructions est fondamental. Pour le dictionnaire, on produit une liste de macro-instructions triée par ordre alphabétique. Pour chaque noeud, on a une macro instruction pour le descripteur du noeud suivie des instructions concernant ses références simples, ses références vecteur, ses références multiples et ses références retour.

2.2.7 Initialisation/clôture

2.2.7.1 Spécifications des initialisations

- Ouverture des fichiers
- Copie du dictionnaire en mémoire
- Contrôle de la non existence de doubles dans le dictionnaire
- Détermination de la racine du graphe formel
- Vérification de l'existence d'erreurs syntaxiques détectées au niveau précédent.

2.2.7.2 Spécifications de clôture

- Fermeture des fichiers
- Mise en page de la liste des erreurs éventuelles.

2.2.8 Séquenceur

Ce module assure le séquencement des différents modules.

2.3 Niveau 3

2.3.1 Architecture du niveau 3

Le niveau 3 est structuré de la manière suivante:
- module de définition des macros
- module de découpe en niveaux du graphe formel.

2.3.2 Définitions des macros

Pour chaque élément du graphe formel, une macro définit son descripteur. L'assemblage d'une série de macro-instructions (supposée sans erreur) définies dans ce module permet d'obtenir la représentation du graphe formel définie dans le chapitre 4. L'ordre des macro-instructions est fondamental (voir le paragraphe précédent). Cette liste de macro-instructions peut être obtenue comme résultat des 2 niveaux précédents.

Exemple

L'exemple suivant définit le descripteur d'une table

```
        MACRO
&N      TAB   &NAME,&DSECT,&LENGTH,&LTYPE,&SINGLE
        GBLC  &PTR,&VECT,&MPRO,&RPTR,&NODEP,&NODEV,&NODER,&NODEM
        GBLA  &CNODE,&CPTR,&CRPTR,&CVECT,&CMPTRO,&CMPTR
        LCLC  &NDX
        LCLA  &CODE
&CPTR   SETA  0  compteur du nbre de ref simples de la table
&CRPTR  SETA  0               "           retour      "
&CVECT  SETA  0               "           vecteur     "
&CMPTRO SETA  0               "           multiples   "
&CODE   SETA  1  init du code la table
        AIF   (&DSECT NE 0).T1       calcul du code
&CODE   SETA  &CODE+4      (voir criteres ds chap 4)
.T1     ANOP
        AIF   (&SINGLE NE 0).T2
&CODE   SETA  &CODE+2
.T2     ANOP
        AIF   ('&LENGTH(1,1)' NE '''').T3
&CODE   SETA  &CODE+1
.T3     ANOP
&N      DC    AL2(&CODE)   production du champ code
        DC    AL2(-1)      production du champ niveau
        DC    A(D&NAME)
        AIF   ('&LENGTH(1,1)' EQ '''').T4
        DC    A(&LENGTH)   production du champ longueur ds le
        DC    A(0)         ou celle-ci est donnee par une cte
        AGO   .T5          ou un symbole
.T4     ANOP                     production du champ longueur
        DC    A(&LENGTH-&START)  ds le cas ou celle-ci est donnee
        DC    CL2'&LTYPE'        par une adresse
        DC    AL2(0)
.T5     ANOP
&NDX    SETC  '&SYSNDX'
&PTR    SETC  'PTR&NDX'      creations de symboles pour
&NODEP  SETC  'XX1&NDX'      les differentes listes de ref
&VECT   SETC  'VECT&NDX'     ainsi que pour les pointeurs
&NODEV  SETC  'XX2&NDX'      vers ces listes
&MPRO   SETC  'MPRO&NDX'
&NODEM  SETC  'XX3&NDX'
&RPTR   SETC  'RPTR&NDX'
&NODER  SETC  'XX4&NDX'
&NODEP  DC    A(&PTR)   production du pointeur vers les ref simples
&NODEV  DC    A(&VECT)  production du pointeur vers les ref vect
&NODEM  DC    A(&MPRO)  production du pointeur vers les ref mult
&NODER  DC    A(&RPTR)  production du pointeur vers les ref retour
        MEND
```

2.3.3 Découpe du graphe en niveau

2.3.3.1 Spécifications

Arguments : une représentation d'un graphe formel comme décrite dans le chapitre 4.

Résultats : une procédure de commande dont l'exécution fournit une représentation du graphe formel découpe en niveaux y comprise.

2.3.3.2 Design

L'algorithme choisi est un parcours en largeur d'abord de la représentation du graphe formel. En voici les grandes lignes :

Initialisation

```
list-father := empty-list;
list-son := empty-list;
level(root) := 1;
crt-level := 2;
findson(root);
permute(list-father,list-son);
```

Programme principal

```
while list-father <> empty-list do
 begin
  crt-level := crt-level + 1;
  crt-node := first(list-father);
  while not end-of-list(list-father) do
   begin
    findson(crt-node);
    crt-node := next(list-father)
   end;
  permute(list-father,list-son);
  list-son := empy-list
 end.
```

```
PROCEDURE FINDSON(n: node);

 begin
  for x such as (there is a link n->x) and (x has level -1) do
   begin
    level(x) := crt-level;
    enqueue(x,list-son);
   end;
```

Il a été implémenté en Assembler pour des raisons de performances.

BIBLIOGRAPHIE

Visibility aspects of programmed dynamic data structures
(N. Madhavji)
(com. of the ACM 8/1984, p. 764-776)

File structures, program structures, and attributed grammars
(L. Logrippo & D. Skuce)
(IEEE Transactioon on software engineering
VOL. SE-9, No. 3, 5/1983, p. 260-286)

Programs to process trees, representing program structures and data structures
(A. D. Wilson)
(Software practice and experience
VOL. 14/9, 9/1984, p.807-816)

Cours de méthodologie de développements de logiciel
(A. Van Lamsweerde)