MASTER EN SCIENCES INFORMATIQUES

Le traitement des contraintes sémantiques en édition syntaxique

Bawin, Claire

Award date:
1988

Awarding institution:
Universite de Namur

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX

NAMUR

INSTITUT D'INFORMATIQUE

LE TRAITEMENT DES CONTRAINTES SEMANTIQUES EN EDITION SYNTAXIQUE

par Claire BAWIN

Promoteur :

Professeur A. van Lamsweerde

Mémoire présenté en vue
de l'obtention du titre
de Licencié et Maître
en Informatique

Année académique 1987-1988

Nous adressons nos plus vifs remerciements à Monsieur le professeur A. van Lamsweerde, pour sa lecture attentive de ce travail ainsi que pour les nombreux documents mis à notre disposition.

Nous remercions également Monsieur D. Delcourt pour ses éclaircissements sur l'atelier ALMA.

Enfin, notre gratitude s'adresse à tout ceux qui de près ou de loin nous ont aidée dans la réalisation de ce mémoire.

TABLE DES MATIERES

INTRODUCTION

Dans les deux dernières décennies, de nombreux travaux ont été entrepris en vue de diminuer le coût de la conception et de la maintenance de logiciels.

Ces travaux se divisent en deux axes importants de recherche: le premier a conduit à l'élaboration d'environnements de programmation destinés à réduire les difficultés présentées par l'écriture de programmes; des outils tels que les compilateurs incrémentaux, les interpréteurs, les optimiseurs ont ainsi vu le jour. Le second s'est orienté vers la conception d'environnements d'aide au développement de gros logiciels. La taille de ces logiciels et leur évolution dans le temps nécessitent des outils spécialisés: aide à la documentation, contrôle de version, gestion de projet, etc.

Que ce soit dans l'un ou l'autre de ces domaines de rercherche, l'information manipulée est structurée: il s'agira par exemple d'un programme (composé d'un en-tête, des déclarations, du corps - lui-même subdivisé en instructions) , du schéma logique d'une base de données projet (composé de noms de modules, des noms fonctions qui en dérivent, de la description des données en entrée et en sortie , etc) ou de produits logiciels spécifiques (spécifications formelles, pseudo-code, etc).

La structure de ces informations se prête à une représentation sous forme d'arbre. Celle-ci permet la description d'un objet à plusieurs niveaux d'agrégation: l'arbre principal d'un projet peut décrire les interconnections entre différents modules; à un niveau plus spécifique, le texte de chacun de ces modules peut être lui-même représenté sous la forme d'un arbre.

Les éditeurs syntaxiques font partie des outils conçus pour la manipulation d'objets structurés. Ils offrent un ensemble d'opérations de manipulation d'arbre. Deux types d'éditeurs peuvent être distingués: les premiers sont dédicacés à un formalisme particulier; les seconds sont génériques. ils ont alors une "connaissance" de la structure des arbres édités, mais ignorent le type des objets pouvant figurer dans les noeuds de ces arbres. Ces informations sont générées à partir d'une description de type BNF.

Ce mode de description d'un formalisme, s'il permet d'exprimer les contraintes structurelles liées à tel ou tel formalisme, est inapte à l'expression de contraintes liées au "contexte". D'où la nécessité de concevoir un ensemble de contrôles destinés à pallier les insuffisances de tels formalismes de définition de formalisme.

Ce travail est consacré à une étude comparative des trois approches les plus connues en cette matière:

. l'approche par routines sémantiques, conçue dans le cadre du projet GANDALF à l'université de Carnegie-Mellon,

. l'approche par grammaire attribuée développée par Knuth, puis approfondie à l'université de Cornell,

. l'approche par sémantique naturelle conçue au sein du projet GIPE à l'INRIA.

Cet exposé comporte deux parties.

La première s'attache tout d'abord à rappeler les concepts préliminaires à la compréhension de ce travail; elle aborde ensuite une étude systématique des trois approches précitées qui sont ensuite comparées.

Dans la deuxième partie, l'éditeur syntaxique développé dans le cadre de l'atelier ALMA conçu aux Facultés Universitaires de Namur est situé par rapport aux trois cadres formels abordés dans la première partie. Une étude d'une extension de cet éditeur est alors réalisée; elle porte sur la possibilité de prise en charge de certains contrôles sémantiques de type Entité-Relation-Association par des services génériques de cet éditeur, éditeur qui serait alors muni de capacité de contrôles sémantiques.

L'annexe 1 reprend une étude détaillée de l'évaluateur incrémental utilisé dans le cadre des grammaires attribuées. Les annexes 2 à 6 reprennent les différents documents ayant permis l'étude réalisée dans la deuxième partie.

PARTIE I : PRESENTATION ET COMPARAISON DE TROIS APPROCHES

CHAPITRE 1 : CONCEPTS PRELIMINAIRES

1.1. Introduction

Dans ce chapitre, nous nous proposons de reprendre les concepts élémentaires qui nous serviront dans la suite de ce travail.

Ainsi, nous passerons en revue les notions de syntaxes concrète et abstraite, et les grammaires y correspondant. Nous introduirons à cette occasion le langage PICO ([Klint,83]) qui nous servira d'exemple de base tout au long de cet exposé. Précisons d'emblée qu'il s'agit d'un langage élémentaire ne reprenant que quelques concepts des langages de programmation habituels. Il permettra cependant de présenter de façon uniforme les différentes conceptions de contrôles sémantiques dans les trois systèmes comparés par la suite; il permettra également une visualisation concrète du mode de spécification de ces contrôles et des possibilités qui y sont liées, tout en n'occultant pas les points essentiels à mettre en évidence dans cette comparaison, grâce à sa simplicité.

Ensuite, nous passerons en revue les différents types de propriétés sémantiques associées à un formalisme. Nous préciserons dans ce paragraphe le concept de contrôle sémantique statique qui est le principal sujet de cette première partie.

Finalement, nous verrons comment ces notions ont été utilisées dans le cadre d'environnements de programmmation : nous présenterons d'abord les principes sous-jacents à un éditeur syntaxique et nous les concrétiserons dans le cadre du système GANDALF. Nous terminerons par le principe de l'édition syntaxique générique.

1.2. Syntaxes abstraite, concrète et grammaires associées.

La syntaxe d'un langage décrit l'ensemble des expressions bien formées dans ce langage.

```
<pico-program>   ::= 'program' <decls> <series> 'end.'
<decls>          ::= 'declare' <id-list> ';'
<id-list>        ::= <id-type> ' ,' <id-list>  | <id-type>
<id-type>        ::= <id> ':' <type>
<type>           ::= 'bool' | 'int'
<id>             ::= <l-liste>
<l-liste>        ::= <lettres> <l-liste> | <lettres>
<lettres>        ::= 'a' | 'b' | ... | 'z'
<series>         ::= <instruction> ' ;' <series> | <instruction>
<instruction>    ::= <asgn-stat> | <if_stat>
<if-stat>        ::= 'if' <exp> 'then' <series>
                     'else' <series> 'fi'
<asgn-stat>      ::= <id> ':=' <exp>
<exp>            ::= <simple-exp> '+' <simple-exp> |
                     <simple-exp> '=' <simple-exp> |
                     <simple-exp>
<simple-exp>     ::= <id> | <number> | <val-bool> |
                     '(' <exp> ')'
<val-bool>       ::= <true> | <false>
<number>         ::= <d-list>
<d-list>         ::= <digit> <d-list>  | <digit>
<digit>          ::= '0' | '1' | ... | '9'
```

Fig. 1

- Syntaxe concrète du langage PICO -

Deux points de vue peuvent être adoptés dans cette description : on peut, d'une part, considérer une expression comme une chaîne de caractères sans structure; la syntaxe du langage décrit alors l'ensemble des chaînes de caractères correctes pour ce langage. C'est ce que nous appellerons par la suite syntaxe concrète du langage, appelée traditionnellement syntaxe. D'autre part, on peut considérer une expression comme une composition structurée de "constructions" du langage, chaque construction correspondant à un concept significatif du langage; la syntaxe décrit alors l'ensemble des compositions bien structurées du point de vue de telles constructions . Une telle syntaxe est alors appelée syntaxe abstraite du langage.

Plus précisément, la syntaxe concrète d'un langage décrit l'ensemble des phrases grammaticalement correctes de ce langage indépendamment de leur structure. Elle sera en général spécifiée en utilisant un méta-formalisme de type BNF (Backus Norm Form) : la syntaxe d'un langage est alors décrite par un ensemble de productions. Chaque production décrit une entité du langage en termes d'autres entités de ce langage. On distingue deux types d'entités : les entités non terminales qui doivent apparaître dans la partie gauche d'au moins une production (plusieurs apparitions correspondant à plusieurs choix possibles pour cette entité) et les entités terminales qui dénotent des unités de base du langage telles que des mots-clés et qui ne demandent pas de définitions supplémentaires.

Ainsi, la Fig. 1 décrit la syntaxe concrète de notre langage PICO . Les entités non terminales sont spécifiées entre < > ; les entités terminales sont spécifiées entre ' ' . Les différentes alternatives pour une entité non terminale sont séparées par un '|' .

Cependant, la syntaxe concrète d'un langage ne constitue pas un bon moyen d'étudier un formalisme en profondeur, et ce pour plusieurs raisons.

D'une part, la spécification de la syntaxe concrète d'un langage contient un ensemble de détails ne concernant que des pures conventions de représentation du langage (appelé communément 'sucre syntaxique') tels que des mots-clés (begin, end,...) et autres conventions syntaxiques ('.', ',', ';', ...). Ainsi, le même concept dans deux langages différents sera décrit de manière différente : par exemple, la construction if dans le langage ADA sera décrite par une règle du type :

<conditionnelle> ::= if <exp> then <series>
else <series> endif

et pour le langage PL1, par une règle du type :

<conditionnelle> ::= IF <exp> THEN BEGIN ; <series> :
ELSE ; <series> ;

D'autre part, la description de la syntaxe concrète d'un langage ne reflète en général pas les concepts fondamentaux sous-jacents à ce langage par le fait qu'elle n'est pas concernée par la structure de ce dernier.

A l'opposé, la syntaxe abstraite d'un langage décrit l'ensemble des expressions correctement structurées selon les constructions fondamentales du langage, indépendamment des détails de représentation. Ainsi, par exemple, on décrira la syntaxe abstraite de la construction if par une règle du type:

<conditionnelle> : <exp> <series> <series>

La description de la syntaxe abstraite est donnée par une grammaire abstraite. Celle-ci comprend, d'une part, un ensemble fini de types syntaxiques qui représentent chacun une classe de constructions du langage (correspondant à des concepts significatifs du langage tels que programme, instruction, variable...) et,d'autre part, un ensemble fini de productions. Chacune de ces productions définit un type syntaxique non terminal en termes d'autres types syntaxiques.

Trois sortes de productions peuvent être distinguées. Elles correspondent aux trois modes d'expression d'un type syntaxique en termes d'autres types syntaxiques du langage.

On a d'abord les productions de sorte 'liste' qui définissent un type syntaxique à l'aide du constructeur de type "séquence" appliqué à un autre type syntaxique. Par exemple, un élément de type "declaration" est une séquence de 0, 1 ou plusieurs éléments de type "déclarations d'identificateurs". Ce que nous noterons :

<decls> : <id-list>*

Ensuite, on a les productions de sorte 'union disjointe' qui définissent un type syntaxique par un ensemble d'alternatives à l'aide d'un constructeur de type "union disjointe". Ainsi par exemple, un élément de type "instruction" peut être soit un élément de type "assignation", soit un élément de type "conditionnelle". Ce que nous noterons :

<instruction> : <assign> | <if>

Enfin, on a les productions de sorte 'agregat' qui définissent un type syntaxique comme agrégat d'autres types syntaxiques à l'aide du constructeur de type "produit cartésien". Par exemple, un élément de type "conditionnelle" est constitué d'un élément de type "expression" et de deux éléments de type "instruction" . Ce que nous noterons :

```
<pico-program>   : <decls> <series>
<decls>          : <id-list> *
<id-list>        : <id> <type>
<type>           : <Bool> | <int>
<series>         : <instruction> *
<instruction>    : <assign> | <if>
<assign>         : <id> <exp>
<if>             : <exp> <series> <series>
<exp>            : <eq> | <plus> | <id> | <number> | <val-bool>
<eq>             : <exp> <exp>
<plus>           : <exp> <exp>
<id>             : <string>
<number>         : <integer>
<val-bool>       : <true> | <false>
```

Fig. 2

- Syntaxe abstraite du langage PICO -

<if> : <exp> <series> <series>

La grammaire abstraite correspondant à notre langage PICO est reprise Fig. 2.
Dans cette grammaire, les types élémentaires int, string et bool correspondent respectivement aux types prédéfinis "entier", "suite de caractères" et "booléen".

Remarquons la forme plus abstraite de la description de la Fig. 2 par rapport à celle de la Fig. 1. Elle est due à la fois à la suppression des détails purement syntaxiques ainsi qu'à l'introduction des constructeurs de type tels que séquence, union disjointe, etc.

Précisons que la présentation est ici simplifiée dans la mesure où les entités constituantes qui se correspondent dans les syntaxes abstraite et concrète, apparaissent en même position respective, ce qui en général n'est pas le cas (la position n'est pas nécessairement conservée).

A condition de se donner un langage de représentation abstrait, un programme peut alors être décrit de manière abstraite. Prenons les conventions suivantes:

- Un élément appartenant à un type syntaxique de sorte liste sera spécifié par le nom du type syntaxique le définissant, suivi, entre crochets, de l'ensemble des éléments le composant, précédé du nom de leur type commun.

Par exemple, si i1, i2 sont deux éléments de type idlist, l'expression suivante:

decls [idlist (i1, i2)]

définit une liste de déclarations constituée des déclarations d'identificateurs i1, i2.

- Un élément de sorte agrégat sera spécifié par le nom de son type suivi entre parenthèses des éléments le composant, chacun précédé du nom de leur type.

Par exemple, si dl est un élément de type decls et s est un élément de type series, l'expression suivante :

pico-program (decls(dl),series(s))

définit un programme composé d'une partie déclaration dl et d'une partie series s.

- finalement, un élément de sorte 'union disjointe' sera spécifié par le nom de son type suivi entre parenthèses de l'élément le composant effectivement, précédé de son type.

```
program

declare x : int,
        t : bool;

x:=1;
t:=true;
if t = true then x:=x+1;
          else x:=x+2;
end
```

Fig. 3

- Exemple de programme PICO -

```
pico-program (decls[idlist(id(x),type(int)),idlist(id(t),type(bool))],
  series [ instruction (assign(id(x),exp(number(1))),
         instruction (assign(id(t),exp(val-bool(true))),
         instruction (if (exp(eq(id(t),valbool(true))),
                  serie [instruction (assign(id(x),exp(plus(id(x),number(1))))])
                  serie [instruction (assign(id(x),exp(plus(id(x),number(2))))])
    ])
```

Fig. 4

- Exemple de programme en langage abstrait -

Par exemple, si a est une assignation, l'expression suivante:

instruction (assign(a))

définit une instruction d'assignation a.

Etant donné ces conventions, la Fig. 3 reprend un exemple de programme PICO . L'expression abstraite correspondante est reprise Fig. 4.

De façon équivalente, un tel programme peut être représenté de manière abstraite par un arbre. Chaque noeud correspond à une construction particulière du langage. On distingue trois types de noeuds : les noeuds non terminaux d'arité fixe (correspondant aux éléments de sorte "agrégat") ayant un nombre fixe de noeuds fils, les noeuds non terminaux d'arité variable (correspondant aux éléments de sorte "séquence") ayant un nombre variable de noeuds fils et les noeuds terminaux qui sont les feuilles de l'arbre. Un arc du noeud A vers le noeud B signifie "l'élément de type A est construit à partir de l'élément de type B". La syntaxe abstraite définit les lois de construction d'arbres bien formés. Dans la suite de ce travail, nous appellerons ce mode de représentation l'arbre syntaxique abstrait d'un programme.

Ainsi , le programme PICO décrit précédemment peut également être représenté par l'arbre présenté Fig.5.

Soulignons également qu'à une syntaxe abstraite peut correspondre plusieurs syntaxes concrètes différentes (y compris graphiques). A un arbre abstrait peut dès lors correspondre différents textes de programmes (différents par exemple par des détails syntaxiques).

1.3. Contraintes sémantiques statiques.

Les contraintes sémantiques statiques sont des contraintes structurelles du langage qui n'apparaissent pas dans la grammaire abstraite de ce dernier pour des raisons que l'on discutera par la suite.

Reprenons notre exemple PICO. Différentes contraintes peuvent être exprimées à propos de ce langage: on ne peut déclarer une variable deux fois, toute variable utilisée doit avoir été déclarée, on ne peut assigner qu'une expression de type booléen à une variable déclarée de ce type...

Rien ne spécifie ces contraintes dans notre grammaire. Remarquons cependant qu'en affinant la description, certaines pourraient y être incluses . Par exemple, dans la grammaire présentée Fig. 6,

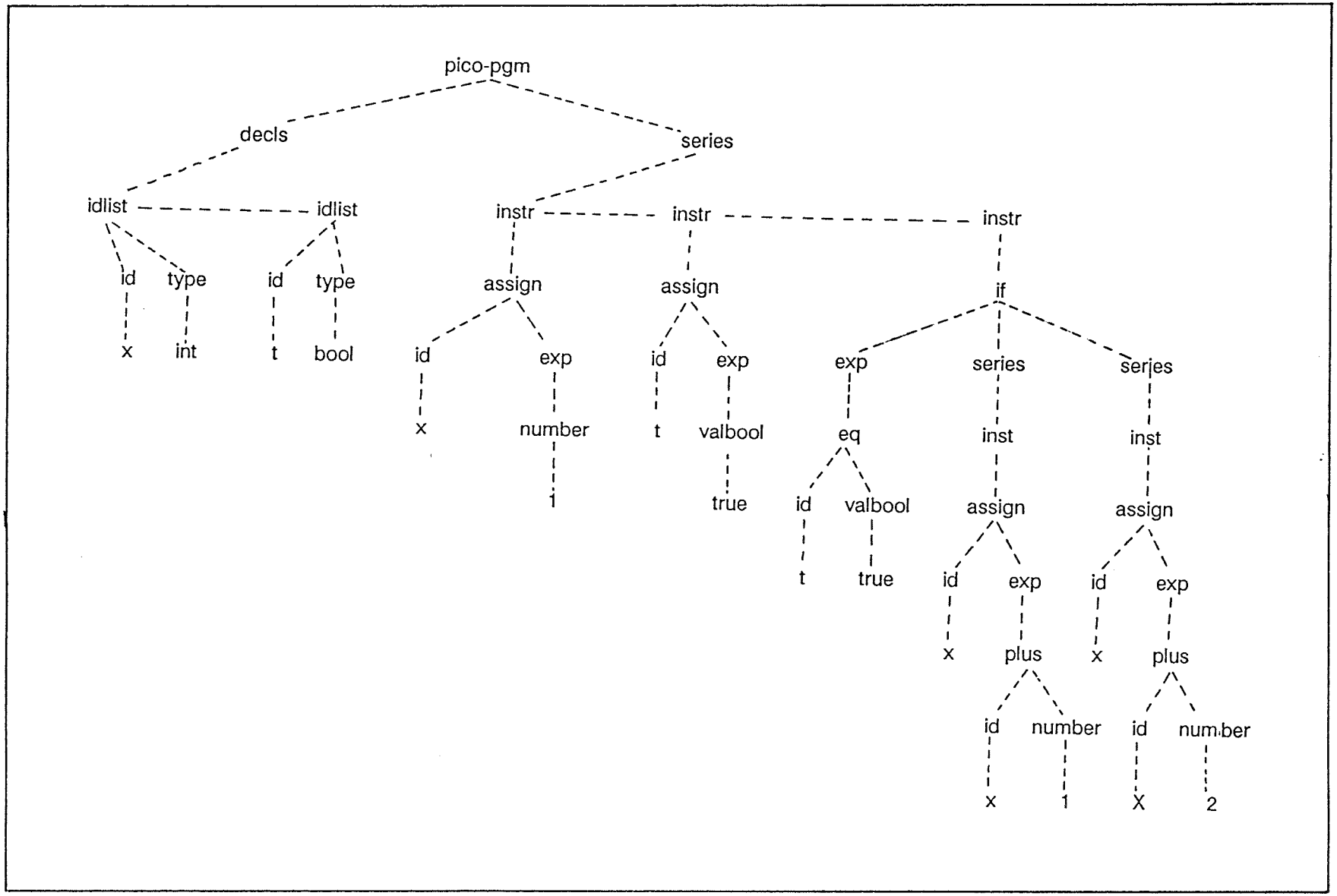


Fig. 5

- Exemple d'arbre abstrait -

nous avons rajouté la condition que la partie expression d'une conditionnelle doit avoir la structure d'une expression booléenne.

Cependant, d'autres types de contraintes ne peuvent être exprimées par simple modification de la grammaire abstraite. Comment pourrions-nous en effet exprimer que, par exemple, une variable ne peut être déclarée deux fois?

En fait, le langage de spécification de la syntaxe abstraite ne permet pas d'exprimer une contrainte qui mette en jeu des aspects du langage dépendant du contexte dans lequel ils apparaissent. Ainsi, par exemple, les contraintes de type ne peuvent être exprimées par le simple biais de la grammaire abstraite, car elles dépendent de renseignements liés à la déclaration des variables dans le programme.

C'est pour ces raisons que la spécification abstraite d'un formalisme sera toujours constituée de deux composantes: on aura d'une part la grammaire abstraite, décrivant l'ensemble des expressions bien structurées de ce formalisme; d'autre part, on aura les contraintes sémantiques qui restreindront ce dernier ensemble en tenant compte de tous les aspects du langage dépendants du contexte.

Par la suite, nous parlerons de contraintes sémantiques statiques. Le terme "statique" provient du fait que nous ne serons concernés que par des contraintes sémantiques pouvant être vérifiées par simple analyse d'un texte écrit en langage abstrait .

De manière plus générale, on peut retrouver quatre types de contraintes dans un formalisme. Ces contraintes portent sur:

- la définition multiple d'objets;
- l'utilisation d'objets non définis;
- la mauvaise utilisation d'objets définis (utilisation en contradiction avec les propriétés définies de ces objets);
- la mauvaise utilisation d'objets non définis explicitement (utilisation en contradiction avec les propriétés inférées de ces objets).

Particularisons ces différents types de contraintes au cas des langages de programmation.

Le premier groupe concerne la propriété générale de non-déclaration multiple d'identificateurs tels que nom de variable, nom de fonction, nom de paramètre dans une fonction, etc. Cette propriété peut être requise à différents niveaux suivant que le langage contienne le concept de "bloc" ou qu'il ne le contienne pas.

```
<pico-program>    : <decls> <series>
<decls>           : <idlist>*
<idlist>          : <id> <type>
<type>            : <int> | <bool>
<series>          : <instruction>*
<instruction>     : <assign> | <if>
<assign>          : <id> <exp>
<exp>             : <exp-bool> | <exp-int>
<if>              : <exp-bool> <series> <series>
<exp-bool>        : <eq> | <val-bool> | <id>
<exp-int>         : <plus> | <id> | <number>
<eq>              : <bool-binary> | <int-binary>
<plus>            : <int-binary>
<bool-binary>     : <exp-bool> <exp-bool>
<int-binary>      : <int-exp> <int-exp>
<id>              : <S>
<number>          : <I>
<val-bool>        : <B>
```

Fig. 6

- Autre syntaxe abstraite du langage PICO -

Le deuxième groupe concerne l'utilisation d'identificateurs non déclarés. Remarquons que, tout comme pour le groupe précédent, cette propriété n'a de sens que si le langage contient explicitement le concept de déclaration. De même, cette propriété ne peut être vérifiée qu'en tenant compte des règles concernant le concept de "bloc".

Le troisième groupe reprend les propriétés générales liées à la déclaration des identificateurs: appel d'une procédure avec le bon nombre de paramètres, non dépassement des bornes d'un tableau, concordance de type entre paramètres formels et effectifs, non utilisation d'une variable déclarée unifiable dans une assignation (contrainte existant par exemple pour le langage ML), non assignation d'une valeur booléenne à une variable déclarée de type entier... Remarquons que ces différentes propriétés de type dépendent à la fois des propriétés des objets déclarés dans le texte du programme et des propriétés des concepts prédéfinis du langage, telles que les règles de type liées à l'assignation, l'addition...

Le quatrième groupe concerne des propriétés générales liées à l'utilisation d'identificateurs dans le cadre de langages ne contenant pas explicitement le concept de déclaration. Les informations liées à l'identificateur sont alors déduites de l'emploi de ces derniers. Deux cas peuvent être distingués : le nom de l'identificateur contient en lui-même toutes les informations nécessaires (nous pensons par exemple au langage FORTRAN, où toute variable non déclarée dont le nom commence par une lettre comprise entre i et n est de type entier), ou il ne la contient pas. Dans cette dernière hypothèse, l'information doit être inférée à partir du contexte dans lequel l'identificateur est utilisé. Il existe alors un risque que cette information ne soit pas connue complètement avant l'exécution du programme. Ainsi, dans l'exemple suivant:

```
program
read(x,y)
w:=x+y
write(w)
```

rien ne permet de connaître le type de x et y (dans l'hypothèse où + peut désigner par exemple l'addition entière ou réelle). Ce type de problème se pose par exemple pour des langages, tel que SETL, qui sont très souples au niveau des règles de type et de portée ("scoping") [Donzeau-Gouge et al. , 87].

D'autre part, dans tout langage de programmation, un ensemble d'anomalies peuvent être détectées. La liste ci-dessous en reprend quelques exemples:

- utilisation d'une variable non initialisée;
- calcul d'une valeur non utilisée;

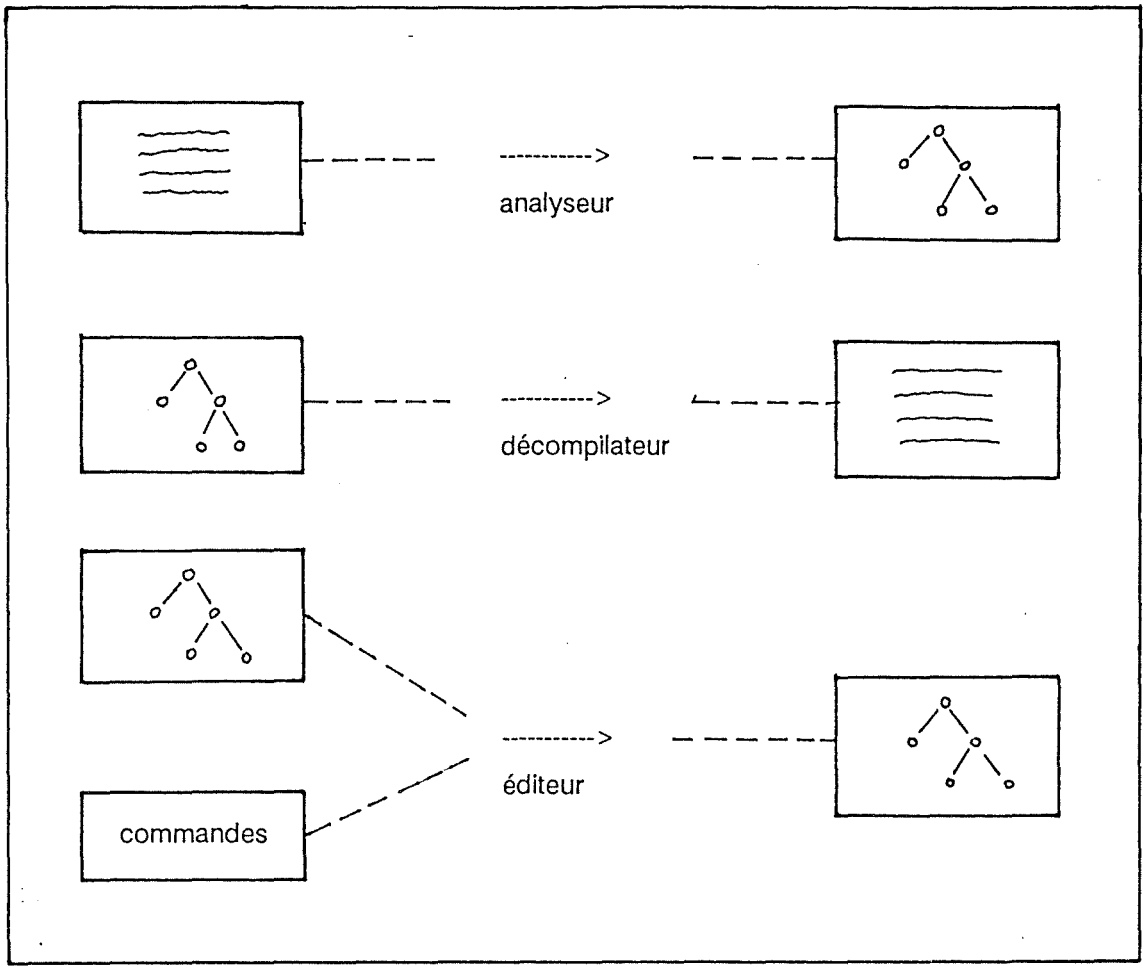


Fig. 7

- Principe de l'édition syntaxique -

- modification d'un paramètre d'entrée;
- division par la constante zéro.

Nous incluerons ces anomalies dans la liste des contraintes sémantiques statiques puisqu'elles peuvent être vérifiées par simple analyse du programme.

1.4. Editeur syntaxique

Une des applications importantes du concept de syntaxe abstraite est l'édition syntaxique.

Un éditeur syntaxique est un éditeur conçu pour construire et manipuler des documents structurés tels que des programmes, spécifications formelles, etc. En opposition aux éditeurs de type texte tels que Emacs qui traîtent un texte comme une séquence uniforme de caractères, un éditeur syntaxique a une "connaissance" de la structure profonde des objets manipulés et édite ces objets en termes de cette structure. Par exemple, un éditeur syntaxique pour le langage PASCAL aura une connaissance de la structure des programmes PASCAL; il sera ainsi capable de maintenir la validité syntaxique des programmes édités et pourra soulager l'utilisateur des détails purement relatifs à la syntaxe concrète tels que le ';' , les mots-clés 'begin', 'end', etc.

Pour réaliser ces tâches, un éditeur syntaxique se base à la fois sur une grammaire abstraite définissant l'ensemble des constructions possibles du langage, et sur des schémas de décompilation permettant de faire la conversion entre la représentation interne du document (sous forme d'arbre syntaxique abstrait) et sa représentation externe telle qu'elle est vue par l'utilisateur. De manière générale, un tel éditeur offre à l'utilisateur deux types de commandes : des commandes de parcours de son texte exprimées en termes de la structure interne de celui-ci et des commandes de modification spécifiques au formalisme (que l'on appellera commandes constructives), qui permettent d'y créer de nouvelles constructions, d'en remplacer par d'autres ou d'en suprimer. A tout moment, les commandes dépendent de la position du curseur (vue externe) et du noeud courant où l'on se trouve dans l'arbre abstrait (vue interne) : seules les commandes correspondant à des constructions syntaxiques valides sont permises par l'éditeur.

La Fig. 7 reprend un schéma du principe de l'édition syntaxique.

Les trois éditeurs syntaxiques étudiés au chapitre 2 se basent sur le même mode de spécification de la grammaire abstraite (inspiré des concepts du système MENTOR [Donzeau-Gouge et al., 83]): une grammaire est constituée d'une liste de règles définissant successivement l'opérateur racine, la liste des opérateurs non terminaux, la liste des opérateurs terminaux et une liste de classes. Un opérateur correspond à un noeud dans l'arbre abstrait. Une classe est une liste d'opérateurs. Elle correspond à la notion de sorte "union disjointe". Une classe est utilisée pour guider l'édition : elle permet d'offrir à l'utilisateur un choix entre plusieurs opérateurs en une position de l'arbre.

Plus précisément, l'opérateur racine est le noeud non terminal qui sera la racine de tout arbre vérifiant la syntaxe ainsi définie. Les classes définissent les noeuds légaux des non terminaux. Elles constituent la partie droite des règles définissant ces non terminaux : pour les noeuds non terminaux d'arité fixe, il y a un nom de classe pour chaque fils (contenant l'ensemble des opérateurs-fils permis à cet endroit); pour les noeuds non-terminaux d'arité variable, il y a un seul nom de classe (tous les noeuds-fils devant appartenir à la même classe).

A titre d'exemple, nous avons repris ci-dessous la grammaire abstraite correspondant à notre langage PICO telle qu'elle est spécifiée dans le système GANDALF [Staudt, 86]. Les noeuds non terminaux d'arité variable y sont spécifiés en entourant le nom de la classe correspondante par < >.

Liste des opérateurs non terminaux

PICO-PROGRAM = decls series
DECLS = <idenlist>
IDEN-LIST = id type
SERIES = <stat>
ASGT = id exp
IF = exp series series
PLUS = exp exp
EQ = exp exp

Liste des opérateurs terminaux

ID = representation
NUMBER = representation
VAL-BOOL = representation
BOOL = static
INT = static

Liste des classes

decls = DECLS
series = SERIES
stat = ASGN IF
exp = PLUS EQ ID NUMBER VAL-BOOL
type = BOOL INT

La production définissant l'opérateur if signifie, par exemple, que cet opérateur a trois fils, le premier devant appartenir à la classe exp, les deux autres à la classe series. Autrement dit, le premier noeud-fils d'un noeud if ne peut être qu'un des noeuds : PLUS, EQ , ID, NUMBER ou VAL-BOOL; les deux autres noeuds-fils ne peuvent, quant à eux, être qu'un noeud SERIES.

Les schémas de décompilation décrivent comment chaque opérateur doit être imprimé dans une représentation textuelle. Dans l'environnement GANDALF, un schéma de décompilation est une mixture de textes et de commandes de mise en format. Ces schémas sont interprétés de gauche à droite. Les parties textes sont imprimées telles quelles, tandis que les commandes de mise en format servent de commandes spéciales au décompilateur.

Ainsi, par exemple, l'opérateur ASSIGN de notre grammaire abstraite PICO pourrait être décrit par le schéma :

@1 := @2 ; @n

où @1 est une commande de mise en format signifiant "décompiler mon premier sous-arbre fils". Elle est suivie de l'impression des symboles := imprimés tels quels. Ensuite, la commande @2 prescrit la décompilation du deuxième sous-arbre fils. Après cette opération, le littéral ';' est imprimé. Enfin la commande @n oblige un passage à la ligne suivante.

Sur base de ces informations, l'éditeur procède de la manière suivante :

A tout moment, la partie de l'arbre déjà éditée est imprimée à l'écran à partir des schémas de décompilation appliqués à cette partie. Les noeuds non terminaux non encore développés (que l'on appellera méta-noeuds) sont imprimés à l'écran précédés du caractère $.

Considérons ,par exemple, l'arbre abstrait présenté Fig.8 a. Il est imprimé à l'écran selon la représentation externe donnée Fig.8 b.

Fig. 8

- Exemple d'une session d'édition -

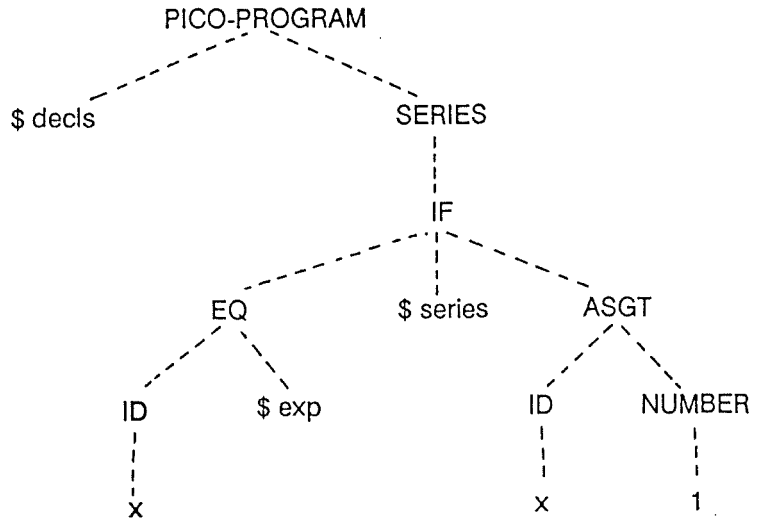


Fig. 8 a

```
program

declare <$ decls> ;

if x  = <$ exp>
      then <$ series>;
      else x : = 1 ;

end.
```

Fig. 8 b

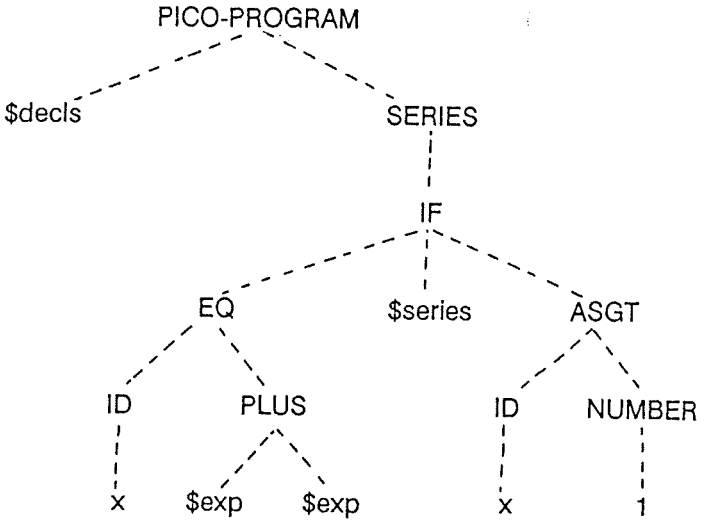


Fig. 8 c

```
program
declare <$decls>;
if x = <$exp> + <$exp>
      then <$series>;
      else x:=1;
end.
```

Fig. 8 d

Pour continuer l'édition de son programme, l'utilisateur déplace son curseur à l'endroit d'un méta-noeud ; il peut alors appliquer une commande constructive valide. Par exemple, à partir du méta-noeud exp, il a le choix entre les commandes constructives plus , eq, id, number et val-bool. Supposons qu'il choisisse la commande plus. Nous avons alors le nouvel arbre abstrait représenté Fig. 8 c. Grâce aux schémas de décompilation, cette forme interne éditée apparait à l'utilisateur sous la forme externe présentée Fig. 8 d.

Le programme est ainsi peu à peu édité. A côté de ces commandes constructives, l'utilisateur a à sa disposition un ensemble de commandes de parcours de l'arbre (aller au noeud parent, au premier noeud fils, au noeud frère droit,...), ainsi que des commandes plus globales de déplacement de sous-arbres, de destruction de sous-arbre, etc.

En plus d'une édition en mode "entrée structurée", l'utilisateur peut également éditer son texte en mode "entrée libre" : les parties de document à éditer en mode texte sont alors mémorisées temporairement dans un tampon de type texte. Lorsque l'utilisateur demande de repasser à une édition à entrée structurée, cette partie est analysée et convertie en l'arbre syntaxique abstrait correspondant. Si des erreurs syntaxiques sont détectées, l'utilisateur doit alors les corriger.

Un des avantages de l'édition syntaxique à entrée structurée est que l'utilisateur ne doit se préoccuper d'aucun détail relatif à la syntaxe concrète. Il édite son programme uniquement en fonction de la structure abstraite de celui-ci, ce qui l'assure que la structure interne de son programme correspond bien à ce qu'il veut. Ceci n'est pas nécessairement le cas pour un éditeur orienté-ligne où le parser "décide" quel type de structure est signifié par l'utilisateur sans lui fournir aucune information à ce sujet.

Cependant, une telle édition peut dans certains cas se révéler assez lourde. Par exemple, ce type d'édition se prête rarement à une introduction commode d'expressions arithmétiques. De même, il peut paraître fastidieux à l'utilisateur de devoir choisir la commande ID avant l'introduction de chaque identificateur. Pour ces raisons, la plupart des éditeurs syntaxiques combinent les deux types d'édition et permettent le passage d'un mode à l'autre.

1.5. Editeur syntaxique générique.

L'objectif visé par les éditeurs syntaxiques génériques est de créer des environnements conçus indépendamment d'un langage particulier. Ces environnements pourront alors être automatiquement instanciés à tel ou tel langage.

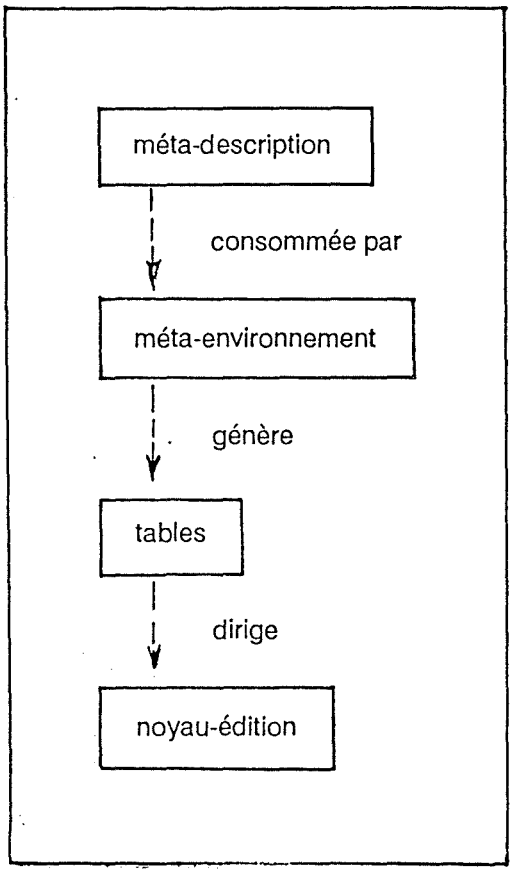


Fig. 9

- Principe d'un éditeur générique -

Un tel environnement est basé sur un noyau éditeur indépendant de tout langage. Les renseignements nécessaires à son fonctionnement se trouvent dans des tables contenant les informations pour chaque langage particulier ; la paramétrisation sur les langages est alors réalisée par un mécanisme d'indirection, les outils du noyau d'édition étant "dirigés" par de telles tables.

Ces tables sont générées par un méta-environnement à partir d'une description du langage fournie par l'implémenteur du système, selon le schéma repris Fig. 9.

Cette méta-description du langage est en général écrite dans un langage de haut niveau et porte sur les aspects suivants:

* Pour les éditeurs syntaxiques stricts (ne permettant qu'une édition en mode "entrée structurée"):

- schémas de décompilation
- syntaxe abstraite

* Pour les éditeurs mixtes (permettant les modes d'édition "entrée structurée" et "entrée libre"):

- schémas de décompilation
- syntaxe abstraite
- syntaxe lexicale (qui définit l'ensemble des mots du langage tels que mots-clés, ponctuation,...)
- syntaxe concrète (qui définit la correspondance entre représentation textuelle et représentation sous forme d'arbre syntaxique abstrait).

En généralisant la Fig. 7, on peut schématiser l'organisation d'un environnement de programmation indépendant d'un langage particulier par la Fig. 10 [Klint, 83].

La description de la syntaxe lexicale et de la syntaxe concrète permet de créer un analyseur pour le langage ainsi défini. L'arbre syntaxique construit est alors transformé en arbre syntaxique abstrait. Les schémas de décompilation sont utilisés pour diriger le décompilateur générique permettant la transformation d'un arbre syntaxique abstrait en la représentation textuelle correspondante.

L'avantage d'un tel environnement est qu'il permet une modularisation complète du système ainsi construit: ce dernier peut être organisé en une collection de processeurs complètement indépendants, où chaque processeur accède simplement aux tables adéquates. La référence commune à chacun d'eux est la syntaxe abstraite du langage. Ainsi, si l'on veut permettre deux syntaxes concrètes différentes pour un même langage, on aura deux ensembles distincts de tables qui conduiront l'analyseur lexical, l'analyseur grammatical et le décompilateur. Toutes seront interprétées par le même méta-analyseur lexical, méta-analyseur grammatical et méta-décompilateur. De même,

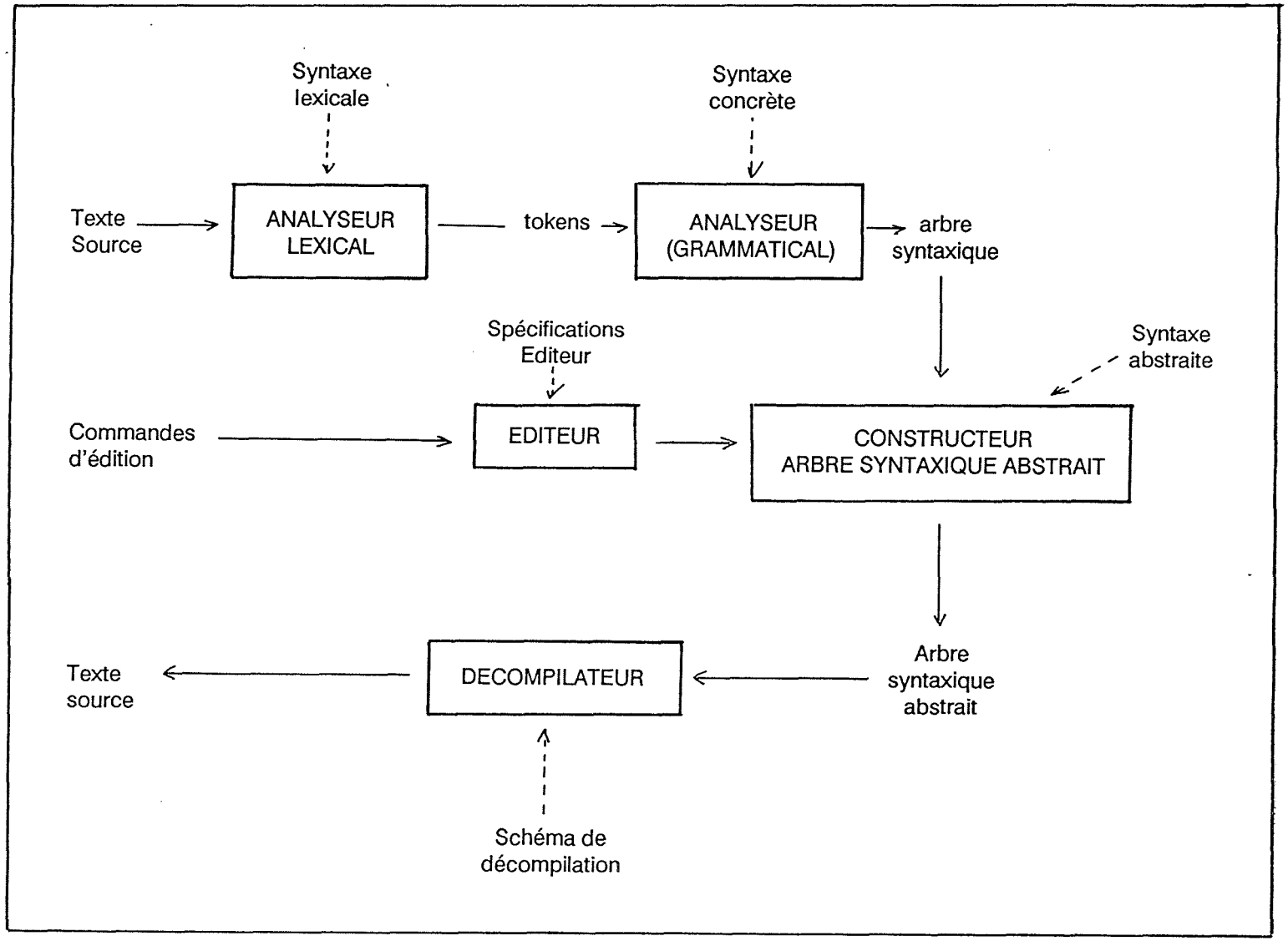


Fig. 10
- Organisation générale d'un éditeur syntaxique -

tous les outils internes au système ne manipulant que l'arbre syntaxique peuvent ignorer l'existence de ces différentes représentations concrètes externes.

De tels environnements permettent ainsi de faciliter la manipulation automatique de programmes et leur transformation systématique. Un autre avantage est de fournir une interface uniforme à l'utilisateur quel que soit le langage utilisé: les programmes écrits dans des langages différents peuvent être manipulés à l'aide d' environnements de programmation quasi similaires.

CHAPITRE 2 : PRESENTATION GENERALE DES APPROCHES DU TRAITEMENT DES CONTRAINTES SEMANTIQUES.

2.1. Introduction.

Dans ce chapitre, nous allons reprendre en détail les trois approches les plus connues en matière de contrôles sémantiques dans un éditeur syntaxique. Nous les comparerons ensuite dans le chapitre suivant.

Nous commencerons par présenter l'approche par routines sémantiques qui a été conçue dans le cadre du projet GANDALF à l'université de Carnegie-Mellon. Il s'agit d'une approche essentiellement procédurale : les contraintes sémantiques y sont décrites par un ensemble de routines qui sont automatiquement activées par l'éditeur en fonction des commandes d'édition de l'utilisateur.

La deuxième approche que nous étudierons est l'approche par grammaire attribuée qui a été principalement conçue par Reps à l'Université de Cornell [Reps, 83]. A l'origine de ces recherches se trouvent les travaux de Knuth [Knuth, 68] montrant comment l'utilisation de grammaires avec attributs permet de décrire la sémantique de langages de programmation. D'où l'idée de concevoir des environnements de programmation interactifs qui combinent une édition dirigée par la syntaxe et la détection d'erreurs sémantiques, en se basant sur ce concept de grammaire attribuée. Cette recherche a conduit au "Synthesizer Generator" que nous étudierons en deuxième lieu.

Enfin, la troisième approche que nous considérerons est l'approche par sémantique naturelle, développée dans le cadre du projet ESPRIT GIPE à l'Institut National de Recherche en Informatique et en Automatique (INRIA) en extension du système MENTOR [Donzeau-Gouge et al., 83]. La sémantique naturelle est un formalisme de spécification de la sémantique de langage de programmation. Elle trouve son origine dans la sémantique opérationnelle structurée proposée par Plotkin [Plotkin, 81], mais elle n'en garde que l'aspect logique. Nous verrons dans le cadre de ce chapitre, les caractéristiques principales de cette sémantique et son emploi dans le cadre d'une édition syntaxique.

Comme nous l'avons déjà souligné précédemment, le traitement spécifique des contrôles sémantiques provient de l'impossibilité d'exprimer les différentes propriétés qui dépendent d'un

contexte précis (telles que, par exemple, les contraintes de type) par le seul biais d'une grammaire abstraite. D'où la nécessité de concevoir des fonctions capables d'analyser l'arbre syntaxique abstrait et de filtrer l'information nécessaire à la détection d'erreurs éventuelles.

Notons que différents axes de liberté peuvent être laissés à l'utilisateur quant à l'édition d'un texte. Les quatre axes que nous avons retenus sont les suivants :

- Processus d'édition à entrée structurée ou édition à entrée libre ;
- simple avertissement d'anomalies sémantiques ou maintien permanent d'un arbre sémantiquement correct ;
- Contrôles effectués à son insu ou à sa requête explicite;
- Contrôle d'un programme en fin ou en cours d'édition.

Remarquons que si les contrôles sont effectués avant la fin de l'édition du programme, il faut que les fonctions d'analyse tiennent compte de l'existence de noeuds non complètement développés. De même, si les contrôles se font en parallèle avec l'édition, il faut essayer que ceux-ci n'augmentent pas excessivement le temps de réponse vis-à-vis de l'utilisateur. D'où la nécessité de pouvoir faire dépendre les fonctions d'analyse de l'arbre d'une modification locale de celui-ci (en exprimant les contrôles en fonction de la syntaxe abstraite) et de travailler au maximum de manière incrémentale (par utilisation de structures de données auxiliaires regroupant les informations nécessaires aux différents diagnostics d'erreurs). Remarquons également que la combinaison des deux axes "maintien permanent de la correction sémantique" et "contrôle en parallèle avec l'édition" peut se révéler assez lourde, et peu appropriée dans le cadre d'une édition à entrée structurée : il est en effet concevable que l'utilisateur désire, par exemple, éditer le corps de son programme avant de déclarer toutes les variables utilisées dans ce dernier. Une telle façon de procéder lui serait interdite dans les hypothèses énoncées ci-dessus.
Cependant, dans le cadre d'un enseignement d'une méthodologie de programmation, une telle démarche peut être intéressante. De même, certains types de contrôles (tel que la vérification de la non-déclaration multiple d'un identificateur) peuvent être adaptés à des vérifications immédiates.

Les trois approches que nous allons présenter dans ce chapitre combinent ces axes de liberté de manière différente. Nous partirons de cette constatation pour les étudier. Pour chacune d'entre elles, nous nous proposons d'exposer successivement leurs objectifs, les idées de base sous-jacentes à la réalisation concrète de ces objectifs, ainsi que les problèmes qui y sont liés. Nous étudierons ensuite le langage de spécification des contrôles et la réalisation concrète de ces idées. Le langage PICO (introduit dans le premier chapitre) nous servira à concrétiser cet exposé. Nous terminerons, pour chacune de ces méthodes, par un ensemble de remarques pouvant être formulées à leur propos.

2.2. Approche par routines sémantiques.

2.2.1. Objectifs.

L'approche par routines sémantiques a été développée dans le cadre du projet GANDALF [Staudt, 86]. De manière générale, ce projet concerne la génération automatique d'environnements de développement de logiciels. En particulier, un éditeur syntaxique générique a été développé dans ce cadre. Celui-ci est un éditeur syntaxique ne permettant qu'une édition à entrée structurée. Les contrôles sémantiques ont été conçus pour être effectués en parallèle avec l'édition. L'implémenteur du système peut soit forcer l'édition de programmes sémantiquement corrects (en empêchant les opérations non valides), soit avertir l'utilisateur d'anomalies sémantiques (par impression de messages adéquats).

L'approche par routines sémantiques combine donc les quatre axes de liberté suivants :

- édition à entrée structurée;
- maintien de la validité sémantique statique de l'arbre édité ou simple avertissement d'anomalies;
- contrôle en parallèle avec l'édition;
- contrôle sur des programmes non terminés.

2.2.2. Présentation générale des concepts de base.

L'approche par routines sémantiques a été essentiellement motivée par le caractère interactif de l'édition. L'idée de base est que, pour maintenir un temps de réponse raisonnable, l'environnement doit réagir de manière incrémentale aux modifications de l'utilisateur. D'où l'idée d'associer à chaque type de noeud de la grammaire abstraite zéro ou une routine. Cette routine est activée chaque fois qu'un noeud de ce type est affecté par une opération d'édition. La routine peut alors parcourir l'arbre, effectuer les contrôles nécessaires et, le cas échéant, interdire l'opération.

Pour minimiser le travail à effectuer, un ensemble de paramètres est communiqué à la routine lors de son activation. Ceux-ci concernent en particulier le type d'opération d'édition effectuée par l'utilisateur et l'adresse du noeud affecté par la modification. Les paramètres permettent d'adapter le comportement d'une routine en fonction du type d'opération d'édition effectuée par l'utilisateur et du

type de noeud. Il est évident, en effet que les vérifications lors de la déclaration d'un identificateur et lors de la suppression de cette déclaration ne se mènent pas de la même manière...

D'autre part, pour éviter aux routines de réanalyser complètement l'arbre à chaque activation, des structures de données auxiliaires peuvent être utilisées pour mémoriser certaines informations sur le programme en cours d'édition . Ces structures sont alors mises à jour et consultées par les routines. Par exemple, une table de symboles peut être utilisée pour mémoriser l'ensemble des identificateurs figurant dans un programme. A chacun d'eux, on peut associer son type, son nombre de déclarations, son nombre d'utilisations en vue de détecter des erreurs éventuelles dans le programme. Cette table est mise à jour lors de chaque création ou suppression d'un identificateur ; elle est consultée pour les vérifications de type.

2.2.3. Problèmes rencontrés.

Deux problèmes ont été essentiellement rencontrés dans la réalisation de ces idées. Ils concernent, d'une part, le modèle d'édition intégrant de tels concepts (décrivant les effets d'une commande d'édition en termes des opérations primitives d'édition et des activations de routines); et, d'autre part, la validation des opérations effectuées par les routines.

Dans la première version de ce système, les routines sémantiques étaient écrites en langage C . Les structures de données auxiliaires étaient implémentées par des variables globales de l'environnement. Elles étaient manipulées par certaines routines lors de leur activation par le noyau d'édition. Un des problèmes qui s'est alors posé est le manque de restrictions imposables sur le comportement de ces routines . D'une part, rien n'interdisait aux routines de manipuler l'arbre du programme de manière arbitraire. Ainsi, par exemple, une routine pourrait supprimer le sous-arbre sur lequel le curseur de l'utilisateur est positionné, laissant cette dernière position indéterminée. D'autre part, rien n'obligeait les routines à utiliser elles-mêmes les opérations primitives du noyau d'édition pour effectuer des modifications ou pour se mouvoir à travers l'arbre. Dès lors, les routines qui auraient dû être activées théoriquement si les mêmes opérations avaient été effectuées par l'utilisateur, ne l'étaient pas. Aucune garantie n'existait donc quant à la validité de l'état sémantique de l'arbre édité, état mémorisé dans les structures de données auxiliaires.

Le deuxième problème qui s'est posé concerne l'ordre dans lequel les routines sémantiques étaient activées en fonction des opérations primitives effectuées par le noyau d'édition. Cet ordre n'était pas toujours cohérent pour les utilisateurs externes au système. Ainsi, par exemple, une commande de création d'un noeud activait la routine correspondante après que le noeud soit réellement créé par le

noyau, tandis que pour une commande de suppression la routine était activée avant. Différents exemples de telles incohérences peuvent être trouvés dans [Kaiser, 85].

Cet ensemble de problèmes fut la cause de beaucoup de critiques à l'égard de cette approche. En pratique, seuls les concepteurs du système furent réellement capables d'implémenter avec succès un ensemble non trivial de contôles sémantiques.

Pour remédier à cet état de fait, un nouveau modèle d'édition fut conçu [Ambriola et al., 84]. Nous en reprendrons les caractéristiques principales au paragraphe 2.2.5 de ce chapitre.

De même, un nouveau "langage à orientation arbre" (ARL) fut conçu pour implémenter les routines sémantiques. Il sera présenté au paragraphe 2.2.4. Le langage ARL améliore le concept de routines sémantiques en permettant de contrôler leur travail : ainsi, les routines n'ont plus directement accès aux structures de données auxiliaires et ne peuvent plus court-circuiter l'utilisation des opérations primitives du noyau.

D'autre part, les structures de données auxiliaires ont été intégrées dans l'arbre du programme : elles sont désormais attachées à des noeuds de l'arbre et correspondent à des attributs décorant cet arbre. Ces attributs sont eux-mêmes des arbres syntaxiques attribués, instances de langages particuliers. L'arbre du programme n'est donc plus constitué d'un seul arbre syntaxique, mais d'un ensemble d'arbres, l'arbre du programme étant l'arbre racine (le seul dont l'utilisateur ait conscience).

2.2.4. Le langage de spécification des contrôles.

La spécification des contrôles sémantiques dans l'approche par routines sémantiques est essentiellement composée de deux parties : elle comprend la description des attributs et la description des routines sémantiques.

Les attributs sont uniquement utilisés par des routines sémantiques. Ils permettent d'attacher de l'information auxiliaire à certains noeuds de l'arbre édité par l'utilisateur. Chaque attribut peut être vu comme une cellule ,avec un compartiment pour la valeur de l'attribut et un compartiment pour son statut (défini ou indéfini). La valeur d'un attribut est indéfinie jusqu'à ce qu'une routine sémantique y mette une valeur. Elle devient indéfinie lorsqu'une routine sémantique retire cette valeur.

De manière générale, il existe deux types d'attributs : les attributs scalaires et les attributs de type arbre. Ces derniers sont décrits par une grammaire abstraite et sont donc des instances particulières d'un langage.

Un attribut est déclaré, dans la méta-définition du formalisme considéré, comme attaché à un opérateur de la grammaire abstraite du langage (la méthode de spécification de cette dernière a été vue au paragraphe 1.4). Il pourra ainsi être attaché par l'éditeur à chaque instance de cet opérateur dans l'arbre syntaxique abstrait.

A titre d'exemple, nous avons repris ci-dessous la déclaration d'un opérateur non terminal PROCEDURE :

```
PROCEDURE =
name blocksdecls command | daemon: aPROCEDURE
attributes :
name : errors | type : BOOLEAN
name : table  | type : TREE | grammar:symtab|
        daemon : <none>
```

Tout opérateur de ce type a trois fils, appartenant respectivement aux classes 'name' (auquel correspondra le nom de la procédure), 'blocksdecls' (auquel correspondra le sous-arbre des déclarations), 'command' (auquel correspondra le corps de la procédure). La déclaration ci-dessus lie la routine sémantique aPROCEDURE à l'opérateur PROCEDURE.

De même, tout opérateur de ce type a deux attributs. Le premier 'errors' est scalaire de type booléen. Il détecte une erreur dans la procédure. Le second 'table' est de type arbre. Il est donc décrit par une grammaire abstraite, à savoir 'symtab'. Il est utilisé pour mémoriser l'ensemble des déclarations locales à la procédure.

Une routine sémantique peut également être attachée à un attribut de type arbre. Elle sera activée chaque fois que cet attribut sera accédé par une (autre) routine sémantique (cfr paragraphe 2.2.5). Dans l'exemple ci-dessus, aucune routine n'a été attachée à l'attribut table.

La spécification d'une routine sémantique est écrite dans le langage ARL. Il s'agit d'un langage impératif spécialement conçu à cet effet.

Ce langage présente un ensemble de caractéristiques communes à tous les langages de programmation impératifs : on y retrouve, par exemple, les concepts habituels de blocs, assignation,

conditionnelle, boucle... De même, il reprend les types de données bool, char, int et string. Il reprend également les opérateurs traditionnels tels que les opérateurs relationnels (<, =, >, ...), les opérateurs booléens (not, and, or), les opérateurs arithmétiques (+, *, -, ...).

A côté de cela, il se différencie totalement des autres langages impératifs par un ensemble de caractéristiques directement liées au concept de routines sémantiques. Ainsi, deux nouveaux types de données ont été introduits pour permettre la manipulation de concepts spécifiques, à savoir le type noeud et le type curseur . Une variable de type noeud a une valeur qui représente un noeud spécifique de l'arbre; un curseur est simplement un pointeur. Ces variables peuvent être utilisées dans différents types d'expressions. Le langage fournit par exemple un ensemble d'expressions permettant de calculer un noeud à partir de la valeur d'un autre noeud ou d'un curseur (calcul du troisième noeud fils, calcul du noeud parent).
Un des aspects intéressants du langage ARL est qu'il permet non seulement de se mouvoir dans l'arbre en fonction de la structure spécifique des arbres produits par l'éditeur mais aussi par le biais des notions de grammaire, opérateurs et classes. On peut désigner par exemple le noeud fils d'un noeud en spécifiant son nom de classe. Notons que ce genre de caractérisation est nécessaire si l'on veut que ce langage soit utile : c'est en effet par l'utilisation de telles expressions que l'on peut par exemple créer un noeud associé à un opérateur spécifique. Cet aspect permet également de séparer la définition de la grammaire des calculs sémantiques : il est plus stable d'identifier un noeud fils par sa classe, que par sa position en cas de modification de la grammaire par exemple. De même, les mêmes spécifications des contrôles peuvent alors être reprises dans le cadre de grammaires abstraites ne différant que par la position des noeuds dans l'arbre.

Pour permettre la modification de l'arbre du programme ainsi que l'accès et la modification des attributs, le langage fournit également un ensemble de commandes. Ces commandes sont quasi similaires aux commandes d'édition mises à la disposition de l'utilisateur. Par exemple, nous retrouvons les commandes habituelles 'change', 'delete', 'insert', 'move' d'un curseur... Nous avons également les commandes 'acces', 'put' et 'get' qui concernent la manipulation des attributs. Ces commandes sont interprétées par l'éditeur comme des commandes d'édition spécifiques; elles provoquent donc les contrôles syntaxiques et sémantiques correspondants (cfr 2.2.5).

Le langage ARL fournit de plus la possibilité à une routine de renvoyer une valeur. Cette valeur peut être soit 'success' soit 'failure'. Par défaut, la valeur renvoyée est 'success'. Le renvoi d'une condition de 'failure' signifie que l'opération en cours doit être abandonnée. Nous reviendrons en détail sur ce point au paragraphe 2.2.5 de ce chapitre.

Notons encore l'existence d'un ensemble de fonctions booléennes permettant de déterminer le type d'un noeud (par exemple, 'ismeta', 'islist',...), d'expressions permettant l'interprétation du signal

passé à la routine lors de son activation (cfr 2.2.5) ainsi que des commandes d'entrée/sortie (entre autres pour imprimer des messages d'erreurs à l'utilisateur).

Nous avons ainsi repris les principales caractéristiques du langage ARL. Une description complète de ce langage peut être trouvée dans [B.Staudt et al., 86]. Pour concrétiser ce rapide exposé, nous avons repris au paragraphe 2.2.6. la description d'une des routines sémantiques faisant partie de notre langage PICO (cet exemple est traité complètement dans ce paragraphe). Il nous permettra de mettre en évidence que le langage ARL, s'il permet d'améliorer sensiblement le concept de routine sémantique, reste cependant difficile au niveau de l'écriture et de la lecture.

2.2.5. Intégration dans un modèle d'édition.

Nous avons vu que la spécification abstraite d'un langage de programmation dans un environnement GANDALF était composée de la grammaire abstraite de ce langage, des attributs attachés aux opérateurs de cette grammaire, ainsi que d'un ensemble de routines sémantiques.

Nous allons étudier dans ce paragraphe le fonctionnement interne de l'éditeur : nous verrons d'abord comment , d'une manière générale, une commande utilisateur est interprétée par le noyau éditeur et par les routines sémantiques. Puis, nous traiterons le mécanisme de communication entre le noyau et une routine. Nous verrons ensuite les moyens mis en oeuvre pour garantir la validité de l'arbre tout au long de l'édition. Nous terminerons par l'étude de la commande-utilisateur de construction. Cela nous permettra à la fois de resynthétiser les différentes notions vues précédemment et de visualiser de manière concrète les interactions entre le noyau éditeur et les routines sémantiques.

2.2.5.1. Modèle d'édition de base.

Toute commande d'édition de l'utilisateur est réalisée par le noyau éditeur en effectuant une séquence d'opérations primitives. Ces opérations primitives sont les seules qui modifient l'arbre interne du programme. C'est à ce niveau que les traitements sémantiques se passent.

Les principales opérations primitives sont les suivantes :

- création d'un noeud à la place d'un méta noeud ;
- suppression d'un noeud ;

1. le noyau vérifie la validité syntaxique.
   Si l'opération est syntaxiquement incorrecte,
   l'opération est abandonnée.
2. le noyau effectue l'initialisation syntaxique
   de l'opération.
3. une routine est appelée pour vérifier la validité
   sémantique de l'opération. Si elle détermine que
   l'opération n'est pas légale, cette opération
   est alors abandonnée (phase permission).
4. une routine est appelée pour effectuer l'initialisation
   sémantique de l'opération (phase pré).
5. le noyau modifie l'arbre ou le contexte de
   l'utilisateur.
6. une routine est appelée pour effectuer des clôtures
   sémantiques (phase post).
7. le noyau effectue les opérations de nettoyage.

Fig. 11

- Interactions entre l'éditeur et les routines sémantiques -

- déplacement du curseur-utilisateur d'un noeud à un de ses fils ;
- déplacement du curseur-utilisateur d'un noeud vers son noeud parent .

Ces opérations peuvent être classifiées en deux groupes suivant qu'il s'agisse d'une opération modifiant l'arbre du programme ou d'une opération modifiant le contexte de l'utilisateur (par déplacement du curseur).

Certaines des commandes utilisateurs sont traduites directement en une opération primitive. Par exemple, à une commande constructive correspondra une opération primitive de création. Par contre, d'autres commandes nécessitent plusieurs opérations primitives. Par exemple, une commande de recherche peut provoquer le déplacement du curseur-utilisateur à un noeud arbitraire de l'arbre. Le déplacement du curseur est alors subdivisé en une série de déplacements élémentaires soit vers un noeud parent soit vers un noeud fils.

L'exécution de chaque opération primitive est elle-même subdivisée en plusieurs étapes : la première consiste à vérifier la validité de l'opération. Lors de la deuxième étape, l'opération est initialisée (par exemple, le nouveau noeud qui sera ajouté à l'arbre par une opération primitive de création est créé en mémoire auxiliaire). Dans une troisième, l'arbre ou le contexte utilisateur est effectivement modifié. La dernière étape permet d'effectuer certaines opérations de clôture (l'éditeur libère la place mémoire associée à l'arbre juste détruit ou au méta-noeud juste remplacé).

Pour chacune de ces étapes, à l'exception de l'étape de modification proprement dite, certains traitements sémantiques peuvent être requis. Ce sont les phases d'une routine sémantique : la phase permission est supposée vérifier la légalité sémantique de l'opération primitive. La phase pré permet d'effectuer les initialisations sémantiques. La phase post permet d'effectuer des opérations de clôture.

Plus précisément, l'interaction entre le noyau et une routine pendant une opération primitive suit le schéma repris Fig. 11.

L'ensemble des opérations décrites dans ce schéma est appelé une transaction. Le noyau garantit qu'une fois la transaction est commencée, elle sera exécutée complètement ou pas du tout. Remarquons qu'une transaction peut ne pas modifier l'arbre : par exemple, si l'opération n'est pas légale du point de vue syntaxique, la transaction sera terminée avant que l'arbre ne soit modifié. Remarquons également qu'une routine peut n'avoir de code que pour certaines phases.

1. validité syntaxique : est-ce que l'utilisateur est à un noeud méta ? est-ce que la déclaration d'une variable est syntaxiquement légale à cette position de l'arbre ?
2. le noyau demande à l'utilisateur le nom de la variable. Le noeud correspondant à cette déclaration est construit en mémoire auxiliaire.
3. phase permission : une routine vérifie si le nom de la variable est unique dans le bloc considéré.
4. phase pre : aucune routine n'est nécessaire.
5. le noyau remplace le noeud méta par le nouveau noeud construit.
6. phase post : une routine ajoute la variable dans la table de symboles.
7. le noyau libère la mémoire associée à l'ancien noeud méta.

Fig. 12

- Exemple de transaction -

Considérons à titre d'exemple le cas de l'opération de création. Supposons que l'utilisateur effectue une commande constructive qui déclare une nouvelle variable dans son programme. La Fig. 12 reprend alors la forme de la transaction.

Dans cet exemple, il n'y a pas de traitement sémantique dans la phase pré. Par contre, si nous supposons que l'utilisateur supprime la déclaration d'une variable, lors de la phase post de la transaction aucune routine ne sera activée. La phase permission servira alors à vérifier que la variable n'est pas utilisée dans le programme. Lors de la phase pré, une routine supprimera l'identificateur de la table de symboles correspondante.

2.2.5.2 **Signaux.**

Lorsqu'une routine sémantique est activée, le noyau fournit à celle-ci un ensemble d'informations lui indiquant les conditions exactes dans lesquelles elle est activée. Ces conditions correspondent à une structure de données appelée signal, qui est passée à la routine comme paramètre. Nous allons reprendre dans ce paragraphe les principaux composants de ce signal et montrer leur utilité.

Le premier champ d'un signal est le champ Event. Il précise l' opération d'édition effectuée. Ce champ permet d'adapter le comportement d'une routine en fonction du type d'opération d'édition effectuée par l'utilisateur. Il est évident par exemple que les vérifications lors de la déclaration d'un identificateur et lors de la suppression de cette déclaration ne se mènent pas de la même manière... Nous avons repris ci-après quelques uns des événements les plus fréquents :

- Create : un nouveau noeud est ajouté à l'arbre où un noeud méta existe. Ce signal est envoyé lorsque l'utilisateur effectue une commande constructive.

- Delete : un sous-arbre est enlevé de l'arbre et remplacé par un méta-noeud. Ce signal est envoyé lorsque l'utilisateur effectue une commande delete ou replace.

- Insert : une copie d'un arbre existant est ajoutée à l'arbre à la place d'un méta-noeud. Ce signal est envoyé lorsque l'utilisateur effectue une commande yank.

- Entry : le curseur utilisateur est déplacé vers un de ses fils.

- Exit : le curseur utilisateur est déplacé vers son parent.

Le second champ d'un signal est le champ phase. Il précise quand la routine est activée par rapport à l'exécution de l'opération d'édition. Les trois valeurs possibles sont les valeurs permission, pré et post. Ce champ permet d'empêcher la réalisation d'opérations non-valides et d'adapter le travail effectué par la routine en fonction des opérations effectuées par le noyau. Par exemple, lors de la phase perm, on vérifiera si un identificateur a déjà été déclaré; lors de la phase pré, un identificateur sera enlevé de la table des symboles avant sa destruction par le noyau ; lors de la phase post, un identificateur sera ajouté à cette même table après sa création par le noyau.

Le troisième champ est le champ relation. Il précise "où suis-je par rapport au noeud concerné par l'opération ?" Il y a entre autres deux réponses possibles : composant, context.

La valeur "composant" signifie que l'opération d'édition est effectuée sur le même noeud que celui auquel la routine est attachée. C'est le cas le plus fréquent.

La valeur "context" signifie que la routine est attachée à un noeud qui représente le contexte de l'opération. Il s'agit en général du noeud parent du noeud concerné par l'opération d'édition. Cette valeur est surtout utile lorsqu'un contrôle ne doit être effectué que pour certaines occurrences de noeuds d'une certaine classe. Par exemple, reprenons une partie de la grammaire décrivant notre langage PICO :

IF : exp series series
PLUS : exp exp
EQ : exp exp
ID : variable
NUMBER : integer
VAL-BOOL : bool

exp = PLUS EQ ID NUMBER VAL-BOOL

Une des contraintes sémantiques de ce langage est que l'expression apparaissant dans une construction IF soit de type booléen. Une routine pourrait dès lors être attachée à chaque opérateur membre de la classe exp (i.e. PLUS, EQ, ID, NUMBER, VAL-BOOL). Pour les opérateurs PLUS et NUMBER, elle aurait pour effet d'interdire la création d'un noeud de ce type si le noeud père est un noeud IF. Par contre, la même opération serait permise par la routine attachée aux opérateurs EQ et VAL-BOOL. En ce qui concerne la routine attachée à l'opérateur ID, elle devrait décider sur base d'une table de symboles. Remarquons qu'une telle solution demande de distribuer une connaissance purement liée à l'opérateur IF à travers plusieurs routines sémantiques. La solution qui a dès lors été

adoptée est d'attacher cette connaissance directement à l'opérateur IF. En utilisant la valeur context du champ relation, la routine sémantique attachée au noeud IF vérifie elle-même dans ce cas que son premier fils est bien de type booléen. Bien que le travail effectué soit le même, cette solution est plus logique. D'autre part, cette solution est plus efficace puisqu'il y aura probablement plus de noeuds de classe exp que de noeuds de type IF.

Le quatrième champ d'un signal est le champ caller. Il spécifie qui est l'initialisateur de l'événement. Parmi les valeurs possibles, nous avons la valeur utilisateur (l'événement provient d'une commande utilisateur), la valeur routine (l'événement a été provoqué par l'activité d'une routine) et la valeur undo (nous en verrons la signification au paragraphe 2.2.5.3). Ce champ permet d'adapter le comportement d'une routine en cas de détection d'une opération non valide. Il est évident que suivant l'initiateur de l'événement, la réaction de la routine peut différer.

Les trois derniers champs font référence à l'arbre. Il s'agit du curseur courant (pointant vers le noeud auquel la routine sémantique courante est attachée), du curseur initial (pointant vers le noeud auquel a trait l'opération primitive) et le noeud auxiliaire qui fournit une information supplémentaire à l'implémenteur (par exemple, pendant les phases permission et pré d'une opération de création, le noeud auxiliaire est le noeud qui remplacera le méta-noeud dans l'arbre. Quand l'opération aura lieu, le noeud auxiliaire sera alors le méta noeud juste enlevé de l'arbre).

2.2.5.3 Mécanismes de protection.

Lorsque l'utilisateur ou une routine sémantique essaient de modifier l'arbre du programme, il y a toujours une possibilité que cette modification ne soit pas effectuée par l'éditeur. Différentes raisons peuvent en être la cause :

1. l'opération est syntaxiquement non légale. Par exemple, une commande constructive n'est pas effectuée si l'opérateur syntaxique n'est pas légal (selon la syntaxe abstraite) à la position du curseur utilisateur.
2. l'opération est sémantiquement non légale : la routine sémantique activée lors de la phase permission renvoie dans ce cas la valeur "failure" et l'opération est abandonnée.
3. une routine sémantique demande de défaire l'opération lors de la postphase.
4. l'opération viole une des règles de protection implémentées par le noyau.

Les deux premières raisons ont déjà été explicitées dans les paragraphes précédents.

La troisième raison évoquée ci-dessus correspond à la possibilité donnée à l'implémenteur de défaire une opération lors de la phase post d'une transaction. La routine sémantique correspondante renvoie alors la valeur failure. Elle indique ainsi au noyau éditeur qu'il doit défaire l'opération. Le noyau effectue alors l'opération inverse. Les signaux correspondants sont activés pour permettre les modifications sémantiques effectuées lors de la phase pré. La justification de cette possibilité offerte à l'implémenteur du système ne nous est pas apparue très clairement : en pratique, la phase permission nous semble suffisante pour vérifier la validité sémantique d'une opération primitive. Nous pensons que ce choix a été guidé par la recherche de la plus grande généralité possible. (Ou par la difficulté d'anticiper les activations en chaîne des routines? D'où la nécessité de pouvoir récupérer des erreurs de conception?). Remarquons que cette facilité est encore à l'étude : dans l'état actuel des choses, seules certaines opérations primitives peuvent être défaites. Les opérations correspondant à des commandes impliquant des modifications complexes de l'arbre ne peuvent pas encore être défaites pendant la phase post d'une opération primitive.

Le quatrième point présenté ci-dessus correspond au mécanisme de protection implémenté par le noyau éditeur. Ce mécanisme a pour but d'empêcher les routines sémantiques d'effectuer des modifications désastreuses sur l'arbre interne du programme. Si une routine sémantique viole une de ces lois, une exception est déclenchée par le noyau.

Les principales règles de protection sont les suivantes :

1. On ne peut modifier l'arbre ou mouvoir le curseur utilisateur pendant la phase permission d'une routine sémantique. Cette phase est en effet réservée uniquement à la vérification de la validité sémantique d'une opération.
2. Le noeud identifié comme contexte d'une opération ne peut être modifié. Modifier un tel noeud rendrait en effet souvent l'acomplissement d'une opération d'édition impossible.
3. De même, aucun noeud sur le chemin d'un curseur à la racine de l'arbre ne peut être détruit car cela rendrait l'opération d'édition impossible : cette protection permet d'éviter qu'une routine supprime la branche de l'arbre contenant le noeud désigné par le curseur.

Nous avons ainsi repris les différents cas où une commande de modification de l'arbre du programme n'est pas effectuée. La question qui se pose est alors la suivante : que se passe-t-il si un de ces cas se présente ?

Lorsqu'il s'agit d'une commande utilisateur, la réponse est immédiate: c'est à la routine sémantique ou à l'éditeur qui reconnaît l'opération illégale d'envoyer le message d'erreur adéquat.

Par contre, la réponse est moins claire lorsqu'une commande effectuée par une routine sémantique échoue. La solution qui a été adoptée est de déclencher une exception.

En fait, toute commande ARL de modification de l'arbre est composée de deux parties : une partie "commande" et une partie "handler".

Cette deuxième partie permet de prescrire ce qu'il faut faire au cas où la commande correspondante échoue. Il s'agit essentiellement d'un "case statement" opérant sur une valeur d'exception. Cette dernière peut avoir cinq valeurs : 'NoKernelPermission', 'NoImplémenteurPermission', 'Undone', 'ProtectionViolation' et 'Succesfull'. Les quatre premières correspondent à la liste des conditions d'erreurs décrites ci-dessus. La valeur successfull signifie que la commande a pu être effectuée sans problème.

Si la partie "handler" d'une commande est omise et qu'une commande ARL échoue, un message d'erreur est alors envoyé à l'utilisateur. Par contre, l'implémenteur peut utiliser cette partie pour effectuer différentes choses : d'abord, il peut décider de continuer et inhiber ainsi le message envoyé à l'utilisateur. Cela peut être utile si les traitements sémantiques ne sont pas affectés par le fait que cette commande a échoué.

L'implémenteur peut également prescrire certains traitements dans le corps du "handler" pour compenser l'erreur ou pour fournir un message plus précis à l'utilisateur.

Une autre alternative est de terminer l'exécution de la routine en envoyant une condition de failure. La dernière possibilité est de tuer l'éditeur et d'arrêter son exécution.

2.2.5.4. La commande-utilisateur de construction.

Pour visualiser les différentes notions vues précédemment, nous allons détailler dans ce paragraphe la commande de construction offerte à tout utilisateur par le noyau éditeur. Rappelons que cette commande permet à l'utilisateur d'insérer un nouveau noeud dans son programme. Elle ne peut être appelée que lorsque le curseur est positionné sur un noeud de type méta. Elle est valide syntaxiquement si l'opérateur proposé par l'utilisateur appartient à la classe correspondant à cette position et définie dans la syntaxe abstraite du formalisme.

Cette commande est en général réalisée par plusieurs opérations primitives : d'abord, le noyau effectue l'opération primitive create. Si elle s'effectue complètement, le noyau effectue ensuite une suite d'opérations enter et exit pour mouvoir le curseur utilisateur au noeud méta suivant. Dans certains cas, il est possible que suite à une commande constructive, le noyau effectue plusieurs opérations primitives

create : cela se passe lorsque le noyau peut automatiquement effectuer certaines constructions (lorsqu'un noeud méta créé suite à cette commande ne peut être remplacé que par un seul opérateur).

Nous allons détailler ci-dessous l'opération primitive create : nous reprendrons d'abord le schéma précis de son exécution. Ensuite, nous ferons le détail des différents signaux générés par le noyau.

Le schéma exact de l'exécution de l'opération create est le suivant :

1. Permission syntaxique noyau. Le curseur-utilisateur doit pointer sur un noeud méta; l'opérateur devant être appliqué pour effectuer la construction doit être légal eu égard à ce méta-noeud. Si une de ces conditions n'est pas valide, l'opération est abandonnée.

2. Préparation noyau. Le noeud qui sera placé dans l'arbre par l'opération atomique est construit. Il est constitué de l'opérateur correspondant avec les méta-noeuds adéquats comme fils. Ce noeud constitue l'information auxiliaire utilisée pour les phases permission et pré des routines sémantiques.

3. Permission sémantique. Elle est programmée par la routine sémantique create-permission (qui vérifie la validité sémantique de l'opération). L'information auxiliaire est le noeud qui a été créé par le noyau lors de la phase préparation de ce dernier.

4. Préparation sémantique. Elle est programmée par la routine sémantique create-pré (qui initialise l'opération). Les paramètres pour ces routines sont les mêmes que ceux passés aux routines sémantiques permission.

5. Opération atomique. Le noeud créé par le noyau lors de la phase préparation est greffé à l'arbre et le méta-noeud est retiré de l'arbre.

6. Implémenteur-post. Elle est programmée par la routine sémantique create-post. Les curseurs courants et start pointent vers le noeud qui vient d'être ajouté à l'arbre. L'information auxiliaire est le noeud méta juste remplacé dans l'arbre. Le noeud contexte est le même que pour les phases permission et pré.

7. Noyau post : le travail effectué par l'éditeur dépend de la réussite de l'opération.

a. succès : si l'opération s'effectue complètement, les opérations suivantes sont effectuées : le noyau libère la mémoire associée au noeud méta remplacé, construit automatiquement les enfants du noeud et détermine la nouvelle position du curseur utilisateur.

b. permission refusée lors de l'étape "permission sémantique" : la mémoire associée au nouveau noeud créé est libérée.

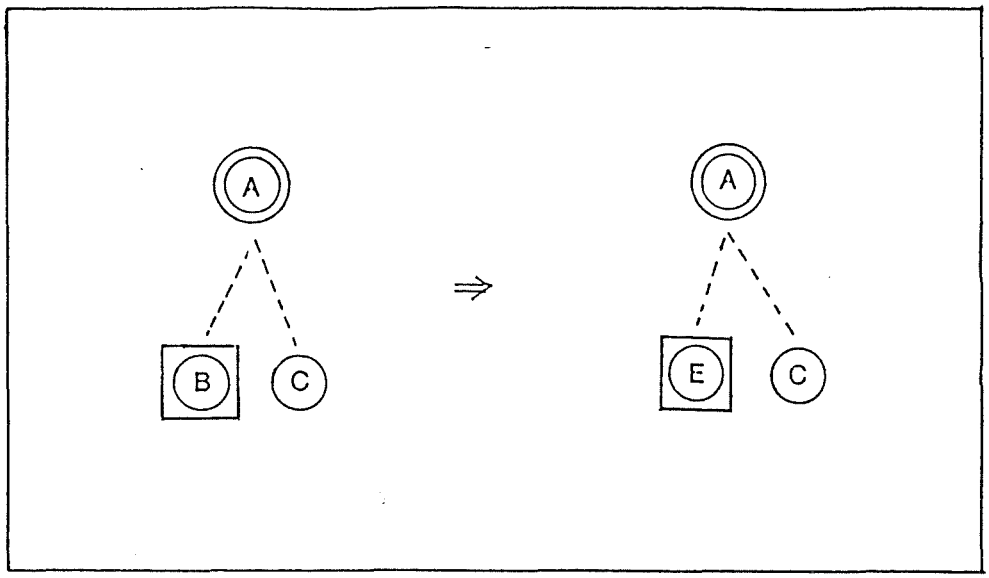


Fig. 13

- Substitution du noeud B par le noeud E -

Event	Phase	Relation	Noeud courant	Noeud start	Info aux.	R.S appelée
Create	Perm.	Composant	B	B	E	E
Create	Perm.	Context	A	B	E	A
Create	Pre	Composant	B	B	E	E
Create	Pre	Context	A	B	E	A
placement de E à la position occupée par B et mémorisation de B						
Create	Post	Composant	E	E	B	E
Create	Post	Context	A	E	B	A

Fig. 14

- La commande-utilisateur de construction -

c. opération à défaire (étape "implémenteur-post") : la mémoire associée à l'ancien noeud méta est détruite. Ensuite, l'opération "delete" est appelée pour remplacer le nouveau noeud construit par un noeud méta.

Le tableau de la Fig. 14 reprend l'ensemble des signaux générés lors d'une opération de création. Les numéros font référence aux différentes étapes évoquées ci-dessus. Il illustre la fig. 13 où un noeud méta B est remplacé par un opérateur E. Un carré y représente la position du curseur tandis qu'un double cercle représente le noeud contexte de l'opération.

2.2.6. Exemple PICO.

Terminons la présentation de l'approche par routines sémantiques par les spécifications complètes des contrôles sémantiques de notre langage PICO.

Rappelons que les contraintes de ce langage sont les suivantes :

- pas de déclaration multiple d'un identificateur;
- non utilisation d'une variable non déclarée;
- propriétés générales de type :
  - on ne peut additionner que des expressions de type entier.
  - on ne peut comparer que des expressions de même type.
  - on ne peut assigner à une variable déclarée de type T, qu'une expression de type T.
  - l'expression apparaissant dans une conditionnelle doit être de type booléen.

Les routines sémantiques que nous présenterons ci-dessous réagissent uniquement aux événements create, delete et exit. Il serait en effet trop long de reprendre toutes les réactions possibles pour tous les événements possibles. De plus, la description complète de ces routines n'ajouterait rien à la compréhension de ce concept.

De même, pour augmenter la lisibilité de cet exemple, nous avons choisi d'écrire les routines sémantiques en un langage pseudo-naturel. Cependant, en vue de concrétiser le langage ARL présenté au paragraphe 2.2.4., nous présenterons la première routine dans les deux langages.

Rappelons encore la syntaxe abstraite de ce langage PICO:

Liste des opérateurs non terminaux

PICO-PROGRAM = decls series
DECLS = <idenlist>
IDEN-LIST = id type
SERIES = <stat>
ASGT = id exp
IF = exp series series
PLUS = exp exp
EQ = exp exp

Liste des opérateurs terminaux

ID = representation
NUMBER = representation
VAL-BOOL = representation
BOOL = static
INT = static

Liste des classes

decls = DECLS
series = SERIES
stat = ASGN IF
exp = PLUS EQ ID NUMBER VAL-BOOL
type = BOOL INT

Pour permettre les traitements sémantiques, nous allons ajouter à l'arbre d'un programme PICO les attributs suivants:

* env qui est déclaré attaché à l'opérateur PICO-PROGRAM. Cet attribut jouera le rôle de table de symboles. Il contiendra l'ensemble des noms de variables apparaissant dans le programme. De plus, pour chaque variable, cet attribut mémorisera si elle a été déclarée ou non, le nombre d'apparitions de cette variable dans le corps du programme ainsi que le type de cette variable (pour

autant qu'elle ait été déclarée). Cet attribut est un attribut de type arbre. Il est défini par la syntaxe abstraite suivante :

-partie non terminale

ENTRIES = <entry>

ENTRY = name declared usage type

-partie terminale

NAME = variable

DECLARED = boolean

USAGE = integer

TYPE = variable

-partie classe

entry = ENTRY

name = NAME

declared = DECLARED

type = TYPE

* type qui est déclaré attaché à tous les opérateurs de la classe exp. Cet attribut contiendra le type de l'expression correspondant à l'opérateur racine concerné : il prendra la valeur integer ou boolean si l'expression est correcte et complètement dérivée; il prendra la valeur undefined sinon.

* status qui est déclaré attaché à tous les opérateurs SERIES. Cet attribut permettra de détecter les phrases non correctes du langage. Il prendra

- la valeur false si une erreur est détectée (i.e. - si l'expression apparaissant dans une instruction conditionnelle est de type entier - ou - si, dans une assignation, le type de l'identificateur apparaissant dans le membre gauche et le type de l'expression apparaissant dans le membre droit sont différents et si ces mêmes types ont la valeur integer ou boolean) .
- la valeur true sinon (i.e. si les expressions issues de l'opérateur concerné ne sont pas complètement développées ou sont erronées ou sont correctes et complètement développées).

Muni de ces informations, nous pouvons écrire les routines sémantiques implémentant les contrôles sémantiques de notre langage PICO.

Par convention, le nom de chaque routine prendra la forme "R.S.op" dans laquelle op désigne l'opérateur syntaxique associé.

Avant d'expliciter chaque routine,précisons que la routine liée à l'opératuer ID a pour but d'empêcher la déclaration multiple de variables. Elle est également appelée pour les mises à jour de la table de symboles lors de la création ou suppression de variables. Nous avons choisi de mettre à jour les informations concernant les types des expressions et le statut des instructions en fin d'édition des

déclarations. Ces opérations sont réalisées par la routine R.S.idenlist. Les routines associes aux opérateurs de la classe "type" mettent à jour, dans env, le type associé à une déclaration. Les routines associéées aux opérateurs de la classe "exp" et "stat" sont activées en réaction à l'événement exit. Elles mettent respectivement à jour les attributs type et statut après analyse du sous-arbre issu de l'opérateur correspondant (d'après les spécifications précisées ci-dessus).

R.S. id

/* Cette routine est activée dans trois conditions :

. phase permission - opération de création : la création de l'opérateur ID est refusée si l'opération correspond à une déclaration multiple d'un identificateur.

. phase post - opération de création : l'environnement est mis à jour.

cas 1: déclaration d'un identificateur non-utilisé.
Ajout de l'entrée (id, true, o, undefined) à env.

cas 2: déclaration d'un identificateur déjà utilisé. Le champ declared de l'entrée correspondante dans env reçoit la valeur true.

cas 3: première utilisation d'un identificateur non déclaré. Ajout de l'entrée (id, false, 1, undefined). L'attribut type de l'opérateur id reçoit la valeur undefined.

cas 4: nième utilisation d'un identificateur non déclaré. Incrémentation d'une unité du champ used de l'entrée correspondante. L'attribut type associé à l'identificateur reçoit la valeur undefined.

cas 5: utilisation d'un identificateur déclaré.
Incrémentation d'une unité du champ used de l'entrée correspondante. L'attribut type associé à l'identificateur reçoit le type déclaré pour cet identificateur.

. phase post - opération de suppression : l'environnement est mis à jour.

cas 1: suppression d'une déclaration. Le champ declared de l'entrée correspondante reçoit la valeur false.
Suppression de l'entrée si le champ used = 0.

cas 2: suppression d'une utilisation. Le champ used de l'entrée correspondante est décrémenté d'une unité.

```
        Suppression de l'entrée si le champ declared = false et
        le champ used = 0.

/*
Quand PERM create et REL = composant
    = >   si le père du noeud à créer n'est pas l'opérateur IDENLIST
              alors /* il s'agit de l'utilisation d'une variable. Aucune vérification ne doit être
                       effectuée */
                    renvoyer (vrai)
              sinon /* il s'agit d'une déclaration de variable */
                       accéder à l'attribut env de l'opérateur  PICO-PROGRAM
                       s'il n'existe pas d'entrée dans env pour cet identifiant
                            alors renvoyer (vrai)
                            sinon si ENV. declared = 'true'
                                 alors /* déclaration multiple */
                                      renvoyer (faux)
                                 sinon /* l'identificateur a déjà
                                      été utilisé mais non déclaré */
                                      renvoyer (vrai)
Quand POST create et REL = composant
    = > accéder à l'attribut env de l'opérateur PICO-PROGRAM
      si le père du noeud courant (i.e. identificateur venant d'être créé) est l'opérateur IDENLIST
              alors /* ajouter la déclaration de la variable */
                   si  l'identificateur n'apparaît pas encore dans l'environnement
                        alors créer une entrée
                                 mise à jour des champs correspondants :
                                 ENV.name := 'id'
                                 ENV.declared := 'true'
                                 ENV.usage := 0
                                 ENV.type := 'undefined'
              sinon /* L'identificateur a déjà été utilisé mais non déclaré.
                        mise à jour du champ declared : */
                                 ENV.declared: = 'true'
```

```
sinon /* Utilisation d'une variable: mise à jour du champs usage dans env */
    si l'identificateur n'apparaît pas  encore dans l'environnement
        alors  créer une entrée et mise à jour des champs correspondants
            ENV.name := 'id'
                ENV.declared := 'false'
                ENV.usage := 1
                ENV.type := 'undefined'
                ID. type := 'undefined'
        sinon accès à l'entrée correspondante
                ENV.usage := ENV.usage + 1
                si  ENV.declared = 'true'
                        alors  ID.type := ENV.type
                        sinon  ID.type := 'undefined'
```

Quand POST delete et REL = composant

=>accéder à l'attribut env de l'opérateur PICO-PROGRAM et recherche de l'entrée correspondant à l'identificateur supprimé (il constitue l'information auxiliaire du signal passé à la routine lors de son activation)

si le père du noeud courant (i.e. le méta-noeud juste créé) est l'opérateur IDENLIST

alors /* supprimer la déclaration de la variable */

ENV. declared := 'false'

suppression de l'entrée si ENV. used = 0

sinon ENV. usage := ENV. usage - 1

suppression de l'entrée si ENV. used = 0 et si ENV. declared = 'false'.

Cette même routine est écrite ci-dessous dans le langage ARL. Nous avons supposé la définition de quatre fonctions :

. Declare : qui met à jour l'environnement lors de la déclaration d'un identificateur.

. Undeclare : qui met à jour l'environnement lors de la suppression d'une déclaration.

. Use : qui met à jour l'environnement lors de l'utilisation d'un identificateur.

. Unuse : qui met à jour l'environnement lors de la suppression d'un identificateur.

Daemon ald

```
current pico cursor cc ;
start   pico cursor sc ;
aux     pico node   an ;
signal  s ;

When : create
   = > cond
        When s "isperm and s "iscomponent
          = > begin
   ENV cursor envcursor ;
   envcursor : = getEnv (^cc) ;
   searchentry (Envcursor, an. variable ) ;
   if op (^envcursor) = ENV! entries
       then return (true) ;
   if ^envcursor. ENV $ declared.{boolean}
       then return (true) ;
       else return (false) ;
   end
 end cond

= > cond
 When s "is post and s" is component
  = > begin
    ENV cursor envcursor ;
    envcursor : = GetEnv (^cc) ;
    if op (^cc.father) < > PICO!Id
      then Use (envcursor, ^cc. variable  )
      else Declare (envcursor, ^cc. variable )
    end
 end cond
```

```
When : delete
  = > cond
   When s"post and s" iscomponent
     = > begin
       ENV cursor envcursor ;
       envcursor : = GetEnv (^cc)
       if op (^cc.father) < > PICO! id
         then unuse (envcursor, ^cc. variable )
         else Undeclare (envcursor, ^cc. variable )
       end
     end cond
```

Remarquons que le langage ARL, s'il permet d'améliorer le concept de routines sémantiques, reste cependant difficile au niveau de la lecture et de l'écriture.

R.S. bool

/* cette routine met à jour dans env le type associé à la déclaration d'une variable */

Quand POST create et REL = composant

= >accéder à l'attribut env de l'opérateur PICO-PROGRAM

accéder à l'opérateur frère gauche du noeud courant

recherche de l'entrée correspond à cet identificateur dans env

ENV. type : = 'boolean'

Quand POST delete et REL = composant

= >accéder à l'attribut env de l'opérateur PICO-PROGRAM

accéder à l'opérateur frère gauche du noeud courant

recherche de l'entrée correspond à cet identificateur dans env

ENV. type : = 'undefined'

R.S. int

/* cette routine met à jour, dans ENV, le type associé à la déclaration d'une variable */

Quand POST create et REL = composant

= >accéder à l'attribut env de l'opérateur PICO-PROGRAM

accéder à l'opérateur frère gauche du noeud courant

recherche de l'entrée correspond à cet identificateur dans env

ENV. type := 'integer'

Quand PRE delete et REL = composant

= >accéder à l'attribut env de l'opérateur PICO-PROGRAM

accéder à l'opérateur frère gauche du noeud courant

recherche de l'entrée correspond à cet identificateur dans env

ENV. type := 'undefined'

R.S. idenlist

/* Cette routine met à jour les informations concernant les types dans tout le programme en fin d'édition des déclaration */

Quand PRE exit et REL = composant

= > . Parcours de toutes les utilisations des identificateurs mémorisés dans env et mise à jour de leur attribut type

. Réévaluation de tous les attributs type et status des expressions et instructions conçernées

R.S. val-bool

/* Cette routine met l'attribut type d'un opérateur VAL_BOOL à la valeur booléenne lors de la création d'un tel opérateur */

Quand POST create et REL = composant

= > VALBOOL.type := 'boolean'

R.S. number

/* Cette routine met l'attribut type d'un opérateur NUMBER à la valeur entière lors de la création d'un tel opérateur */

Quand POST create et REL = composant

= > NUMBER.type := 'integer'

R.S. plus

/* Cette routine met à jour l'attribut type d'un opérateur PLUS. Cet attribut prend la valeur integer si les deux epressions fils sont de de type entier et sont complètement dérivées; undefined sinon (i.e. une des expression est erronée ou non complètement développée) */

Quand EXIT

= > Parcours des sous-arbres fils du noeud courant (i.e. l'opérateur PLUS)

Si on rencontre un noeud méta

alors PLUS.type := 'undefined'

sinon si la valeur des attributs type des 2 expressions fils sont égales et ont la valeur integer

alors PLUS.type := 'integer'

sinon PLUS.type := 'undefined'

R.S. eq

/* Cette routine met à jour l'attribut type d'un opérateur EQ. Cet attribut prend la valeur boolean si le type des deux expressions filles est integer ou boolean et que ces deux types sont égaux; la valeur undefined sinon (i.e. une des expressions est erronnée ou non complètement dérivée).
*/

Quand EXIT

= > Parcours des sous-arbres fils du noeud courant (i.e. l'opérateur EQ)

Si on rencontre un noeud méta

alors EQ.type := 'undefined'

sinon si la valeur des attributs type des 2 expressions fils sont égales et différentes de la valeur undefined

alors EQ.type := 'boolean'

sinon EQ.type := 'undefined'

R.S. if

/* Cette routine met à jour l'attribut status d'un opérateur IF . Il prend la valeur true si l'expression fille gauche est de type boolean ou undefined; false sinon */

Quand EXIT

= > si l'attribut type de l'opérateur fils gauche (de classe exp) est différente de integer

alors IF.status := 'true'

sinon IF.status := 'false'

R.S. asgt

/* Cette routine met à jour l'attribut status des opérateur ASGT. Il prend la valeur true si les attributs type des deux sous-arbres fils sont égaux et différents de undefined; false sinon */

Quand exit

= > si les attributs type des deux sous-arbres fils sont

égaux et différents de undefined

alors ASGT.status := 'true'

sinon ASGT.status := 'false'

2.2.7. Remarques

Comme nous l'avons déjà souligné, l'approche par routines sémantiques a été conçue pour s'exécuter en paralèlle avec l'édition et dans le cadre d'une édition à input structuré. Remarquons que cette approche s'adapterait aisément à une édition de type texte: en effet, il suffirait, dans cette hypothèse, de mémoriser temporairement le texte dans un buffer. Lorsque l'utilisateur veut repasser en mode structuré, la conversion du texte en l'arbre syntaxique correspondant prendrait alors lieu . Les contrôles seraient effectués en parallèle en simulant la création de chaque noeud ainsi obtenu.

Notons, de plus, que dans ce cas, le "maintien" de la correction sémantique des programmes édités s'exclut.

2.3 Approche par grammaires attribuées.

2.3.1. Objectifs.

Rappelons que la paternité du concept de grammaire attribuée revient à [Knuth, 68] et que [Reps, 84] fut le premier à avoir eu l'idée d'utiliser ce concept pour implémenter des contrôles sémantiques dans un éditeur syntaxique. Ses recherches ont conduit au développement du "Synthesizer Generator".

Il s'agit d'un éditeur syntaxique générique permettant à la fois une édition à entrée structurée et une édition à entrée libre. Les contrôles sémantiques y sont conçus pour s'effectuer en parallèle avec l'édition. Lorsqu'une erreur sémantique est détectée, un message d'erreur est automatiquement généré par l'éditeur. L'éditeur ne force pas l'édition de programmes sémantiquement corrects.

L'approche par grammaire attribuée combine donc les quatre axes de liberté suivants :

- édition à entrée structurée et à entrée libre;
- simple contrôle de la validité sémantique statique des objets édités;
- contrôles effectués en parallèle avec l'édition;
- contrôles sur des programmes non terminés.

2.3.2. Présentation générale des concepts de base.

L'idée de base est de décorer l'arbre interne représentant le programme édité d'attributs. Ces attributs ont comme rôle de dériver de manière automatique les informations relatives aux aspects dépendant du contexte dans le programme.

2.3.2.1. Les grammaires attribuées.

On peut distinguer deux types d'attributs : les attributs hérités et les attributs synthétiques. De manière intuitive, les premiers sont chargés de passer de l'information vers le bas de l'arbre tandis que les seconds remontent de l'information vers le haut de l'arbre.

La valeur de tout attribut est déterminée par une fonction: la valeur d'un attribut synthétique attaché à un noeud de l'arbre est définie comme fonction de la valeur des attributs attachés à des noeuds fils de ce noeud; la valeur d'un attribut hérité est définie comme fonction de la valeur d'attributs de noeuds père et frères.

De manière plus précise, une grammaire attribuée est une grammaire abstraite (ou plus généralement une grammaire context-free) étendue de la manière suivante : à chaque type syntaxique, on associe un ensemble fini d'attributs. D'autre part, à chaque production de la grammaire abstraite, on associe un ensemble d'équations sémantiques définissant la valeur des attributs attachés à chaque type syntaxique intervenant dans cette production. Chaque équation définit la valeur d'un attribut synthétique associé au type syntaxique du membre gauche de la production ou d'un attribut hérité associé au type syntaxique du membre droit de la production; cette valeur est définie comme fonction des autres attributs des types de la production.

(i) **Définition formelle**

De manière formelle, la définition d'une grammaire attribuée a été établie par Knuth [Knuth, 68] de la manière suivante :
soit la grammaire G = (V,N,S,P) où V est l'ensemble des types terminaux et non terminaux ; N est l'ensemble des types non terminaux ; S est le type racine (qui n'apparaît dans la partie droite d'aucune règle de production) ; et P est l'ensemble des règles de production .

Les règles sémantiques sont ajoutées à G de la manière suivante :
à chaque type X∈V, on associe un ensemble fini A(X) d'attributs ; A(X) est partitionné en deux ensembles disjoints, les attributs synthétiques A0(X) et les attributs hérités A1(X) . L'ensemble A1(S) doit être vide (i.e. le type racine n'a pas d'attributs hérités) ; de même l'ensemble A0(X) doit être vide si X est un type terminal . Chaque attribut dans A(X) peut avoir un ensemble de valeurs possibles V_{α} dans lequel une valeur sera sélectionnée (par le biais de règles sémantiques) pour chaque apparition de X dans l'arbre de dérivation .

Supposons que P soit un ensemble de m productions et que la pième production soit :
Xp0 : Xp1 Xp2 ... Xpnp
où np>0, Xp0∈N et Xpj∈V pour 1<=j<=np

Les règles sémantiques sont des fonctions fpj_{α} définies pour tout 1<=p<=m, 0<=j<=np et $\alpha \in A0(Xpj)$ si j=0, $\alpha \in A1(Xpj)$ si j>0. Chacune est une fonction d'un ensemble $V_{\alpha 1} \times V_{\alpha 2} \times \ldots \times V_{\alpha t}$ dans V_{α} ,pour un t=t(p,j,α) >= 0 où chaque i = i (p,j,α) est un attribut d'un Xpkj pour 0<=ki =ki(p,j,α) <=np , 1<=i<=t. (En d'autres mots, chaque règle sémantique évalue la valeur d'un attribut d'un Xpj à partir de la valeur de certains attributs associés à Xp0, Xp1, ... Xpnp.)

(ii) **Exemple**

Concrétisons ces notions sur un exemple [Knuth, 68] et voyons comment une grammaire attribuée permet de définir la sémantique d'un nombre en représentation binaire (i.e. la valeur entière correspondant à ce nombre).

La Fig. 15 reprend la grammaire abstraite définissant un nombre binaire. Pour définir la sémantique d'un nombre en représentation binaire, nous allons nous définir cinq attributs , à savoir :

- value-N : attribut synthétique du type syntaxique Number
- value-I : attribut synthétique du type syntaxique Integer
- value-B : attribut synthétique du type syntaxique Bit

<Number> : <integer>

<integer> : <integer> <bit> | <bit>

<bit> : <0> | <1>

Fig. 15

- Grammaire abstraite d'un nombre binaire -

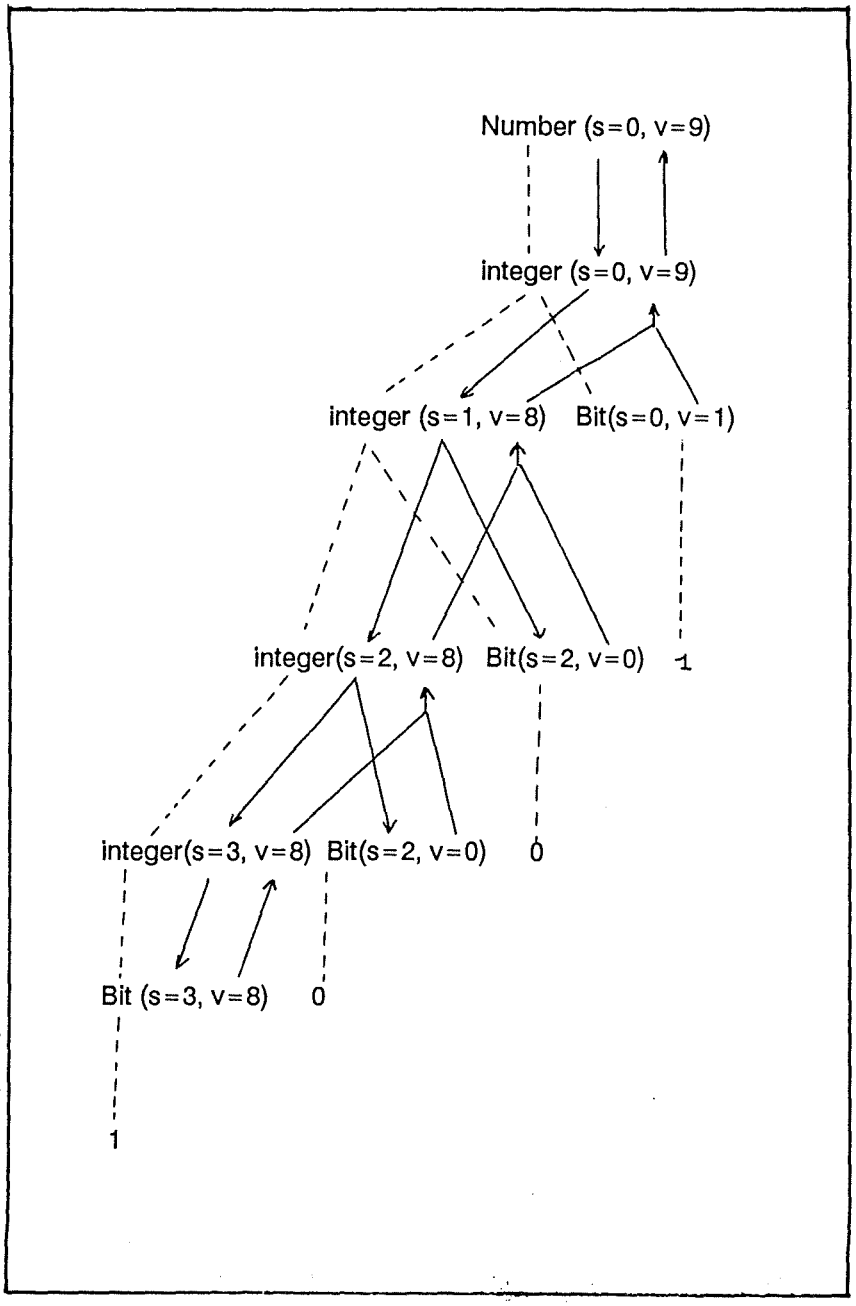


Fig. 16

- Arbre attribué correspondant au nombre binaire 1001 -

- scale-I : attribut hérité du type syntaxique Integer
- scale-B : attribut hérité du type syntaxique Bit

La valeur de l'attribut "value" associé à chaque noeud de type Number, Integer ou Bit représente la valeur entière correspondant au sous-arbre ayant ce noeud comme racine.
La valeur de l'attribut scale-B associé à un noeud de type Bit représente la position de ce bit dans le nombre binaire correspondant. La valeur de l'attribut scale-I associé à tout noeud de type integer représente quant à elle la position du bit le plus à droite dans le sous-arbre correspondant.

Ils sont définis par les équations sémantiques suivantes :
(Remarque : lorsqu'un même attribut apparaît plusieurs fois dans une équation sémantique, le numéro entre parenthèses qui le suit désigne la position du type syntaxique auquel il est attaché).

```
<number> : <integer>
        value-N = value-I
        scale-I = 0

<integer> : <integer> <bit>
        value-I(1) = value-I(2) + value-B
        scale-I(2) = scale-I(1) + 1
        scale-B    = scale-I(1)

<integer> : <bit>
        value-I = value-B
        scale-B = scale-I

<bit> : <0>
        value-B = 0

<bit> : <1>
        value-B = 2 * * (scale-B)
```

Remarquons que la grammaire attribuée présentée ci-dessus ne correspond qu'à une des façons de définir la sémantique des nombres binaires. Ce n'est pas la plus simple. Une autre définition n'utilisant que des attributs synthétiques peut être trouvée dans [Knuth, 68]. La grammaire présentée ci-dessus a cependant l'avantage d'illustrer à la fois les concepts d' attributs hérités et synthétiques.

Nous avons repris Fig. 16 , l'arbre représentant le nombre binaire 1001. Pour chaque noeud, "s" représente la valeur de l'attribut "scale" calculée en évaluant les équations reprises ci-dessus; "v" représente la valeur de l'attribut "value". Une flèche A -> B signifie que la valeur de l'attribut B est calculée à partir de la valeur de l'attribut A d'après l'équation sémantique correspondante.

(iii) **Quelques définitions**

Dans la suite de cet exposé, nous dirons qu'une grammaire attribuée est bien formée si les trois conditions ci-dessous sont remplies :

(i) il faut que les types syntaxiques terminaux n'aient pas d'attribut synthétique associé ;
(ii) il faut que le type syntaxique racine n'ait pas d'attribut hérité associé ;
(iii) il faut que pour chaque production de la grammaire, on ait une équation sémantique par attribut synthétique déclaré associé au type syntaxique du membre de gauche de la production ou attribut hérité associé à un type syntaxique du membre de droite, et ce pour chaque type y apparaissant.

Remarquons que les équations sémantiques données dans une grammaire attribuée doivent être telles que tous les attributs doivent toujours pouvoir être calculés quel que soit l'arbre de dérivation. Il faut donc s'assurer que les équations ne conduisent pas à une définition circulaire (étant donné que les attributs ne sont pas évalués dans une seule direction). Cette restriction peut être formalisée en utilisant le concept de graphe de dépendance. Il s'agit d'un graphe orienté représentant les dépendances fonctionnelles entre les attributs d'une production P ou d'un arbre T. Ce graphe peut être défini de la manière suivante :

(i) pour chaque attribut b, le graphe contient un sommet b'
(ii) pour chaque attribut b qui est un argument de la fonction définissant un attribut c, le graphe de dépendance contient un arc orienté (b', c'). (Nous dirons désormais que c est un argument de b.)

Une grammaire est dite non circulaire si le graphe de dépendance de n'importe quel arbre de dérivation ne contient pas de cycles . Un algorithme pour tester la circularité est donné dans [Knuth, 68].

Dans la suite de ce chapitre, nous supposerons que nous travaillerons avec une grammaire bien formée et non circulaire. De même nous supposerons que nous travaillerons avec une grammaire sous forme normale. Une grammaire est sous forme normale si chaque équation sémantique définit la valeur d'un attribut synthétique du type syntaxique de gauche de la production ou celle d'un attribut hérité d'un type syntaxique de droite, comme fonction de zéro, un ou plusieurs attributs hérités du type syntaxique de gauche ou d'attributs synthétiques de la partie droite de la production. La restriction aux formes normales nous facilitera la présentation des algorithmes conduisant l'évaluateur incrémental.

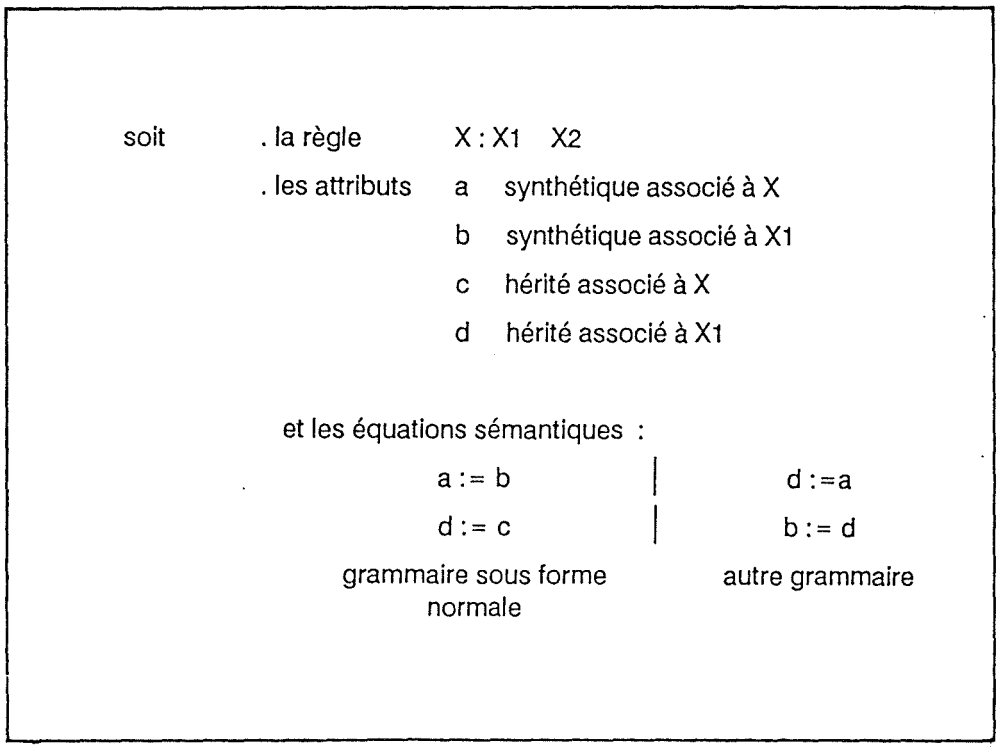


Fig. 17

- illustration du concept grammaire sous forme normale -

La figure ci-contre reprend un extrait d'une grammaire sous forme normale et d'une grammaire ne vérifiant pas cette propriété .

2.3.2.2 **Application à l'édition syntaxique.**

L'idée de Reps est d'utiliser les attributs pour faire passer l'information relative aux aspects du langage dépendant d'un contexte précis jusqu'aux noeuds où les contrôles peuvent être faits. Ainsi par exemple, en faisant passer une table de symboles à travers l'arbre, on peut vérifier à tous les noeuds utilisant une variable, que celle-ci a bien été déclarée avant d'être utilisée ; de même en construisant cette table, on peut vérifier qu'elle ne contient pas de déclaration multiple... Ces vérifications sont effectuées par des expressions booléennes dont la valeur sert à détecter une erreur et permet ainsi d'annoter le programme.

Illustrons ces remarques dans le cadre de notre langage PICO et voyons comment vérifier la contrainte :" toute variable utilisée a été déclarée". (nous n'avons repris ici que quelques unes des règles de production du langage PICO. Cet exemple est traité complètement au paragraphe 2.3.6 de ce chapitre).

Pour ce faire, nous allons utiliser trois types d'attributs:
- l'attribut env qui contiendra la liste des identificateurs déclarés dans le programme;
- l'attribut used qui contiendra l'ensemble des identificateurs utilisés dans une expression;
- l'attribut check qui prendra la valeur faux lorsqu'une variable est utilisée bien que non déclarée ;vrai sinon .

Nous pouvons décrire la valeur de ces attributs par les équations sémantiques reprises Fig. 18.

Nous avons supposé implicitement les déclarations suivantes :

- déclaration des attributs synthétiques env-P, env-D, env-ID pour les types syntaxiques pico-program, decls et idlist
- déclaration des attributs hérités env-S, env-IS, env-AS, env-IF pour les types syntaxiques series, instruction, assign et if
- déclaration des attributs synthétiques used-EXP, used-PL pour les types syntaxiques exp et plus.

La huitième équation détecte une erreur sémantique . Elle vérifie si toutes les variables utilisées dans l'Instruction d'assignation ont bien été déclarées. De tels attributs sont utilisés dans le "Synthesizer Generator" pour générer de manière conditionnelle des messages d'erreurs.

<pico-program> : <decls> <series>

(1) env-P = env-D
(2) env-S = env-P

/* construction de la table des symboles */

<decls> : <idlist>*

(3) env-D = U {(env-ID)}

<idlist> : <id> <type>

(4) env-ID = {id}

/* descente de la table des symboles */

<series> : <instruction>*

(5) env-IS = env-S

<instruction> : <assign>

(6) env-AS = env-IS

<instruction> : <if>

(7) env-IF = env-IS

<assign> : <id> <exp>

(8) check = (used-EXP U {id}) env-AS

<exp> : <plus>

(9) used-EXP = used-PL

<plus> : <exp> <exp>

(10) used-PL = used-EXP(1) U used-EXP(2)

<exp> : <id>

(11) used-EXP = {id}

Fig. 18
- Application du concept de grammaire attribuée à l'édition syntaxique -

Remarquons que si les contrôles sémantiques sont effectués en parallèle avec l'édition, le concept de grammaire attribuée ne peut s'appliquer tel quel : en effet, dans ce cadre,les attributs sont réévalués après chaque modification de l'arbre. Un problème se pose alors aux noeuds non complètement développés: quelle est, en effet, la valeur de leurs attributs synthétiques ? Pour qu'à tout moment, les attributs aient une valeur définie, Reps a choisi d'ajouter à tout type syntaxique non terminal, une production complémentaire:

< X > : < meta >

Ce type < meta > signifie non développé et les équations sémantiques associées à cette production complémentaire définissent la valeur des attributs synthétiques ce X .
Ainsi par exemple, au type syntaxique < decls >, nous devons ajouter la production

< decls > : < meta >

et l'équation sémantique définissant l'attribut synthétique env-D comme étant l'ensemble ne contenant aucune déclaration:

env-D = { }

2.3.3. Problèmes rencontrés.

Le principal problème rencontré dans la réalisation de ces idées est un problème lié à l'efficacité. Ce problème concerne à la fois le temps de réponse à l'utilisateur et la place nécessaire à la mémorisation des attributs.

Dans la mesure où l'on veut que les contrôles sémantiques s'effectuent en parallèle avec l'édition, il est important de minimiser le temps de réponse à l'utilisateur. Il a donc fallu concevoir un modèle d'édition qui permette de diminuer au maximum le travail à effectuer après chaque modification de l'arbre édité (i.e. minimiser le nombre d'attributs à réévaluer). Nous verrons au paragraphe 2.3.5.1 de ce chapitre le modèle adopté dans le cadre du "Synthesizer Generator". Ce modèle permet de localiser les attributs directement affectés par une modification de l'arbre interne de façon à ne pas devoir réévaluer tous les autres attributs . Cet ensemble d'attributs est fourni en entrée d'un évaluateur incrémental. Le principe est de se baser sur les relations de dépendance existant entre les valeurs des attributs de l'arbre pour réévaluer un nombre minimal d'attributs. Cet évaluateur est optimal en ce sens que le nombre d'attributs réévalués après chaque modification est proportionnel au nombre d'attributs affectés par la modification. Nous étudierons en détail cet algorithme d'évaluation au paragraphe 2.3.5.

Un autre problème posé par le concept de grammaire attribuée est la place nécessaire à la mémorisation des attributs de l'arbre. Cette dernière peut en effet rapidement devenir très importante et

dépasser à la limite la place requise par l'arbre syntaxique pur. Différentes techniques ont été étudiées pour résoudre ce problème. Un bref aperçu peut être trouvé dans [Reps, 84].

Reps lui-même a étudié deux algorithmes d'évaluation permettant de minimiser la consommation mémoire nécessaire aux attributs. Ces algorithmes sont dynamiques en ce sens que les décisions d'allocation de place dépendent non seulement de la grammaire abstraite mais également de l'arbre particulier en train d'être édité. Ces algorithmes n'ont pas un comportement optimal : contrairement à l'évaluateur incrémental évoqué ci-dessus, ils peuvent réévaluer un même attribut plus d'une fois. Cependant, ils présentent l'avantage de mémoriser dans tous les cas au plus $O(\sqrt{n})$ valeurs d'attributs pour le premier algorithme, et O(log n) pour le second (où n est le nombre d'attributs de l'arbre). L'inconvénient de ces algorithmes est qu'ils ont pour but l'évaluation d'un attribut particulier de l'arbre. Ils ne peuvent dès lors s'intégrer dans le schéma d'édition présenté ci-dessus.

Reps a également étudié une méthode générale permettant de partager la place occupée par de gros attributs tels qu' une table de symboles. Cette technique se base sur la structure de données partageables"2-3-tree". Nous n'étudierons pas cette technique dans le cadre de ce mémoire. Pour de plus amples informations, nous prions le lecteur de se référer à [Reps,84].

2.3.4. Langage de spécification des contrôles.

Dans l'approche par grammaire attribuée, la spécification des contrôles sémantiques est composée de deux parties : elle comprend la spécification des attributs et la spécification des équations sémantiques définissant la valeur de ces attributs.

Dans le cadre du "Synthesizer Generator", ces deux composants sont intégrés dans la description générale d'un langage à deux niveaux : d'une part, les attributs sont déclarés attachés aux classes de la grammaire abstraite du langage. Les équations sémantiques sont définies pour chaque opérateur de ces classes. Ceci permet à l'éditeur de créer les attributs adéquats lorsque des occurrences particulières de ces opérateurs sont générées. D'autre part, certains attributs peuvent être intégrés dans la description des schémas de décompilation. La valeur de ces attributs est alors insérée dans la représentation externe de l'arbre du programme. Ces attributs correspondent à des messages d'erreurs. Leur valeur dépend de l'état de l'arbre édité.

De manière plus générale, les attributs peuvent contenir de l'information auxiliaire (autre que des diagnostics d'erreurs). Ces attributs constituent une base de données de faits dérivés qui peut être présentée à l'écran et utilisée pour contrôler le processus d'édition .

Le méta-langage de spécification permettant de décrire un nouveau langage est le langage SSL (Synthesizer Specification Langage). Dans ce paragraphe, nous allons essentiellement nous concentrer sur les aspects de ce langage qui concernent la spécification des contrôles sémantiques. A cet effet, nous allons étudier les points suivants : d'abord, nous verrons comment spécifier la grammaire abstraite d'un langage. Puis, nous présenterons le mode de déclaration des attributs. Nous étudierons ensuite la spécification des équations sémantiques. Pour éclairer cet exposé, nous reprendrons des extraits de la spécification de notre langage PICO. Rappelons encore que cet exemple est complètement traîté au paragraphe 2.3.6. de ce chapitre.

2.3.4.1. Spécification de la grammaire abstraite.

En SSL, le mode de spécification d'une grammaire abstraite est quasi similaire à celui conçu dans le cadre du projet GANDALF : chaque opérateur est défini comme appartenant à une classe (appelée phylum) et ayant un certain nombre de fils de phylum spécifié.(Rappelons que ce mode de spécification est basé sur les concepts du système MENTOR [Donzeau -Goudge,83])

Ainsi, par exemple, la déclaration suivante :

```
PICO-PGM :    pgmnull ()
              pgmpair (DECLS, SERIES);
```

déclare le phylum PICO-PGM comme composé des deux opérateurs pgmnull et pgmpair. Le premier est un opérateur terminal. Le second est un opérateur non terminal ayant deux fils : le premier de ces fils doit être un opérateur appartenant au phylum DECLS. Le deuxième doit être quant à lui un opérateur du phylum SERIES.

Certains phyla peuvent être déclarés comme de type liste. Leur spécification est alors précédée de l'en-tête "list" et est constituée de deux productions : l'une est une production d'arité nulle et l'autre est une production binaire dont la partie droite est récursive.

A titre d'exemple, la déclaration suivante déclare le phylum DECLS comme étant une liste d'opérateurs appartenant au phylum DECL :

```
list DECLS
DECLS : declsnull ()
      | declpair (DECL, DECLS);
```

Comme nous l'avons vu au paragraphe 2.3.2 de ce chapitre, la déclaration de chaque phylum doit contenir la spécification d'un opérateur complémentaire. Cet opérateur est utilisé pour définir les

attributs synthétiques de noeuds non complètement développés. Il correspond au premier opérateur déclaré de chaque phylum. Cet opérateur est toujours d'arité nulle. Dans les exemples présentés ci-dessus, ces opérateurs sont les opérateurs declsnull pour le phylum DECLS et pgmnull pour le phylum PICO-PGM.

2.3.4.2. **Déclaration des attributs.**

Les attributs sont quant à eux déclarés comme associés à un ou plusieurs phyla. Ceci permet à l'éditeur de les attacher à chaque instance d'un opérateur de ces phyla. Chaque déclaration spécifie également si l'attribut est synthétique ou hérité.

Un des choix qui a été effectué est de représenter les attributs par des arbres syntaxiques abstraits. Leur domaine de valeur est spécifié par un nom de phylum. Tout comme dans GANDALF, on a donc une unification des domaines syntaxiques et sémantiques.

Par exemple, la déclaration suivante déclare l'attribut 'env' comme étant synthétique, attaché aux opérateurs de phylum PICO-PGM (i.e. pgmnull, pgmpair) et de domaine de valeur ENV

PICO-PGM { synthesized ENV env }

Chaque instance de cet attribut sera donc un arbre d'opérateur racine appartenant au phylum ENV. il vérifiera la syntaxe définie pour ce phylum.

Il est également possible de définir des attributs locaux à une production. Cette possibilité est offerte par le langage SSL pour éviter de devoir associer un même attribut à tous les opérateurs d'un phylum.De même, certains attributs peuvent être déclarés comme "facultatif".Ces attributs ont alors la particularité de n'être évalués que sur une demande explicite (à l'initiative de l'utilisateur ou de l'éditeur si cet attribut est nécessaire à l'évaluation d'un attribut voisin).

2.3.4.3.**Spécification des équations sémantiques.**

La valeur de chaque attribut est définie par une équation sémantique. Il y a une équation pour chaque opérateur appartenant au phylum auquel l'attribut est associé.L'ordre dans lequel ces équations sont spécifiées n'a pas d'importance.

Par exemple , la déclaration ci-dessous spécifie la valeur de l'attribut synthétique 'env' de l'opérateur pgmpair du phylum PICO-PGM. Cette valeur est définie comme étant égale à la valeur de l'attribut env associé à l'opérateur fils gauche de cet opérateur pgmpair (pour rappel, ce premier fils gauche est une instance du phylum DECLS)

PICO-PGM : pgmpair { PICO-PGM.env = DECLS.env }

Différentes expressions peuvent être utilisées pour spécifier la valeur d'un attribut. Une description complète de ces expressions SSL peut être trouvée dans [Reps, 87]. Dans le reste de ce paragraphe, nous allons en reprendre les grandes caractéristiques.

Une expression peut tout d'abord être une variable. Une variable est un nom dénotant un arbre abstrait . Ce dernier est appelé la valeur de la variable . Il existe différentes sortes de variables . Les principales sont les suivantes :

(i) l'occurence d'un phylum dans une règle de production de la grammaire est une variable . Sa valeur est l'arbre dérivé de cette occurence dans l'instance particulière de cette production .
Par exemple, définissons les phyla DECL et TYPE par les règles de production :

DECL : declnull () | declpair (id,TYPE)
TYPE : null () | bool () | int ()

Nous pouvons associer au phylum DECL l'attribut synthétique type par l'équation :

DECL : declpair { DECL.type = TYPE }

Cet attribut mémorise pour chaque identificateur déclaré son type.

Cette possibilité de faire des références syntaxiques dans la description de la sémantique souligne l'avantage d'utiliser une représentation des attributs sous forme d'arbre abstrait. En effet, si un arbre syntaxique était un type de valeur différent d'une valeur d'attribut, on devrait convertir l'arbre syntaxique référencé dans le domaine sémantique .

(ii) Dans les équations sémantiques associées à une production, chaque attribut hérité et synthétique d'un phylum apparaissant dans la production est une variable dont la valeur est déterminée par son équation sémantique dans la production. Si X désigne une occurence d'un phylum dans une production donnée et attr est un attribut déclaré de ce phylum, alors X.attr désigne cet attribut.
Dans l'ensemble suivant :

PICO-PGM : pgmpair { PICO-PGM.env = DECLS-env }

l'attribut "env" attaché au phylum PICO-PGM est calculé sur base de la valeur de l'attribut "env" associé au premier fils gauche.

(iii) Dans les équations sémantiques d'une production p, il est possible de faire référence de manière non-locale aux attributs d'une production différente. Celle-ci doit apparaître nécessairement au-dessus de n'importe quelle production p dans un arbre (par au-dessus, nous entendons "entre n'importe quelle instance de la production p et la valeur de l'arbre").
Ceci permet d'alléger l'écriture des équations en évitant de devoir descendre une information le long des branches d'un arbre jusqu'à l'endroit où le contrôle peut être fait. Ainsi, dans l'exemple présenté Fig. 18, les règles (5) à (7) deviennent inutiles.

Une expression peut également être un opérateur appliqué à k arguments appartenant aux phyla appropriés. Sa valeur est alors un arbre ayant l'opérateur donné comme racine et les k arbres arguments comme constituants. Le nom d'opérateur peut donc être utilisé comme un constructeur d'arbre.
Ainsi, par exemple, l'expression

declpair (id (X), bool ())

désigne un arbre instance du phylum DECL. Il représente la déclaration d'un identificateur X de type booléen.

Le nom d'un opérateur peut également être utilisé comme un discriminateur : le langage SSL fournit des expressions permettant d'analyser un arbre instance d'un phylum spécifique par une sélection multiple où les différents cas sont étiquetés par les noms d'opérateurs.
Un exemple d'une telle expression est l'expression "with".La syntaxe de cette expression est la suivante :

```
with(expo)
 ( pattern 1 : exp 1
 .
 .
 pattern k : exp k
 )
```

La valeur d'une expression "with" est la valeur de l'expression i correspondant au premier pattern qui concorde avec la valeur de l'expression o . Un pattern peut contenir des variables. Leur portée est locale à l'expression correspondante. Si le pattern concorde avec l'expo, alors ses variables seront liées aux constituants adéquats dans l'expo. La valeur de l'expression i peut être calculée en terme de ces variables.

```
TYPE look up (ID id, ENV env)
return  with (env)
                    (nullenv() : null () ,
                    envconcat(b,e) :with (b) (paire(s,t) : (id = = s ? t : lookup (id,e)))
```

Fig. 19

- Illusration de l'expression "with" -

Par exemple, considérons un environnement comme une liste de paires (identificateur, type). Nous pouvons le décrire par la syntaxe abstraite :

```
ENV : nullenv()
    | envconcat (BINDING ENV) ;
BINDING : paire (ID TYPE) ;
TYPE  : null() | int() | bool () ;
```

Pour illustrer l'utilisation d'une expression "with", définissons une fonction récursive lookup qui renvoit pour un identificateur donné son type si l'identificateur est présent dans l'environnement, null() sinon (voir Fig. 19).

Cette fonction reçoit deux arguments id et env appartenant respectivement aux phyla ID et ENV et renvoie un résultat appartenant au phyla TYPE. Son corps correspond à une expression "with". La dernière expression y apparaissant est une expression conditionnelle traditionnelle de la forme :

exp1 ? exp2 : exp3

qui est une abréviation de :

with (exp1) (true : exp2, false : exp3)

Comme dans tout autre langage, le langage SSL fournit également les opérateurs traditionnels tels les opérateurs relationnels, les opérateurs booléens et les opérateurs arithmétiques opérant sur des valeurs primitives (entières, booléennes...). Il offre également des opérations permettant la concaténation de liste, l'ajout d'un élément en tête de liste,etc.

2.3.5. Intégration dans un modèle d'édition.

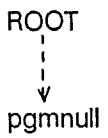
Dans ce paragraphe, nous allons étudier la façon dont l'éditeur effectue les mises à jour après chaque commande utilisateur. Pour ce faire, nous allons d'abord présenter un modèle d'édition simplifié. Toute commande d'édition sera soit une opération déplaçant le curseur soit une opération de remplacement d'un sous-arbre. Ce cadre posé, nous verrons comment l'évaluateur incrémental procède pour réévaluer les attributs après chaque modification de l'arbre. Nous généraliserons ensuite le modèle d'édition et nous discuterons des impacts de cette généralisation sur l'évaluateur présenté auparavant.

2.3.5.1. Modèle d'édition simplifié.

Dans un premier temps, nous allons considérer le modèle d'édition suivant : toute commande d'édition consiste soit à mouvoir le curseur utilisateur à travers l'arbre, soit à remplacer un sous-arbre

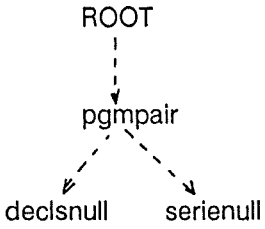
du programme par un autre. Chaque noeud non complètement développé est complété à l'aide des opérateurs complémentaires définis dans la grammaire abstraite.

Par exemple, l'édition d'un programme en langage PICO est vue comme une succession d'opérations de remplacement et de mouvements du curseur opérant sur l'arbre initial:



où le curseur est positionné sur pgmnull.
Chaque insertion à un noeud non développé consiste à remplacer l'opérateur complémentaire par un sous-arbre dont le noeud racine appartient au même phylum que l'opérateur remplacé.

Par exemple, si nous utilisons la deuxième production définissant le phylum PICO-PGM (i.e. pgmpair (DECLS, SERIES)), nous obtenons l'arbre suivant :



Chaque suppression d'un sous-arbre consiste à remplacer ce sous-arbre par l'opérateur complémentaire de même phylum.

Ces modifications ont un impact sur la valeur des attributs de l'arbre. L'objectif visé est qu'après chaque modification de l'arbre, la valeur des attributs soit consistante (i.e. en accord avec les équations sémantiques les définissant).

De manière plus précise, nous dirons qu'un attribut b est consistant si :

(i) les arguments de b ont été évalués ;
(ii)la valeur de b est égale à la valeur de son équation sémantique pour autant que b ne soit pas un attribut hérité d'un noeud racine d'un sous-arbre libre.

Dans le modèle présenté ici, nous supposerons que toute opération de remplacement est effectuée en détachant d'abord le sous-arbre concerné suivi de l'accrochage d'un sous-arbre consistant.

Plus précisément, si T est l'arbre édité et U un sous-arbre de T de racine r appartenant au phylum X, U sera détaché de T en enlevant le sous-arbre de racine r. Soit U' un sous-arbre libre de racine r' appartenant également au phylum X. U' est attaché à T à la feuille r en assignant aux attributs synthétiques de r' la valeur des attributs synthétiques de r et en remplaçant ensuite r par U' dans T (la valeur des attributs hérités de r' est quant à elle inchangée). Le remplacement du sous-arbre U par U' est effectué en détachant d'abord U et en accrochant ensuite U' à sa place.

Cette façon de procéder ainsi que notre hypothèse selon laquelle la grammaire avec laquelle nous travaillons est sous forme normale assurent que les attributs origines d'inconsistances sont concentrés au noeud r où a été effectué le remplacement : en effet, de part l'hypothèse de normalité, les attributs apparaissant dans la partie supérieure de l'arbre (arbre obtenu en supprimant de T le sous-arbre U') ne peuvent dépendre que des attributs synthétiques de r' (dont la valeur est restée identique en raison de la définition d'accrochage d'un sous-arbre). De même, la valeur des attributs apparaissant dans U' ne peut dépendre que de la valeur des attributs hérités de r'.

Les algorithmes présentés ci-dessous se basent sur le graphe de dépendance que nous avons introduit au paragraphe 2.3.2 de ce chapitre. Ce graphe permet en effet de mettre en évidence les dépendances existant entre les différents attributs définis dans la grammaire. De ce fait, nous pouvons évaluer la propagation des inconsistances à travers l'arbre du programme.

2.3.5.2. Algorithmes de réévaluation.

Le problème posé est de réévaluer un nombre minimal d'attributs après chaque modification de l'arbre après une opération d'édition. Appelons l'ensemble des attributs dont la valeur doit être recalculée NOUVEAU. Il est à remarquer que cet ensemble n'est pas connu à priori après une modification. Il faut dès lors identifier les attributs à réévaluer et recalculer leurs valeurs. Différents algorithmes ont été étudiés par Reps pour résoudre ce problème [Reps, 84]. L'un d'entre eux a un coût proportionnel à la taille de NOUVEAU. Il est donc asymptotiquemnt optimal en temps puisque le nombre d'attributs devant être réévalués ne peut être inférieur au nombre d'attributs dans NOUVEAU.

Dans ce paragraphe, nous allons présenter les idées sous-jacentes à cet algorithme. Une analyse plus détaillée peut être trouvée en annexe 1. Dans un premier temps, nous nous restreindrons au cas d'une grammaire attribuée non circulaire et sous forme normale. Dans un deuxième temps, nous montrerons comment abandonner cette deuxième hypothèse.

Remarquons encore que d'autres algorithmes ont également été étudiés dans [Reps,84] pour des classes de grammaires plus restrictives présentées dans la littérature : les grammaires L-attribuées,les grammaires ordonnées et les grammaires absolument non circulaires. Par rapport à celui

```
S : X S
 c = a(2)
 a(1) = b + a(1)

S : t
 a = 1

S : s
 a = 2

X : x
 b = c
```

Fig. 20

- Grammaire illustrant l'algorithme présenté en 2.3.5.2. -

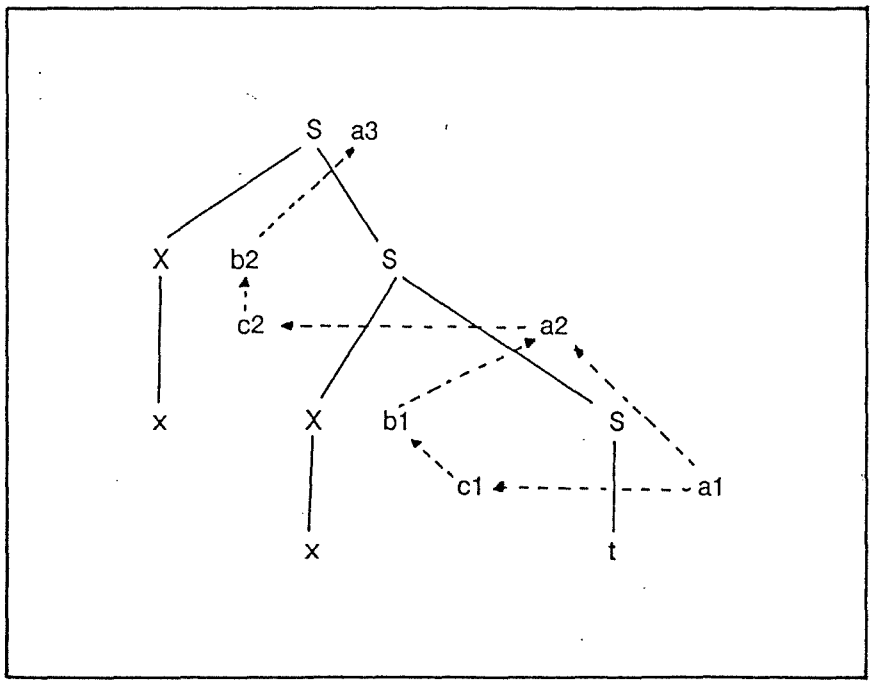


Fig. 21

- Exemple d'un arbre attribué pour la grammaire présentée Fig. 20 -

présenté ci-dessous, ces algorithmes présentent l'avantage de devoir mémoriser moins d'informations à chaque noeud de l'arbre édité, l'ordre dans lequel les attributs devant être réévalués étant induit par la forme de la grammaire.

(i) **Présentation générale :**

L'algorithme que nous allons présenter a donc comme objectif de réévaluer un minimum d'attributs suite à la modification de l'arbre interne. Comme nous l'avons remarqué en 2.3.5.1, dans notre modèle d'édition , les attributs à l'origine d'inconsistance sont localisés à l'endroit de la modification.

La façon la plus simple de procéder est alors de se baser sur les relations de dépendance existant entre les attributs de l'arbre, relations représentées par le graphe de dépendance associé à l'arbre (cfr 2.3.2) : on établit au départ la liste des attributs pouvant être inconsistants. On réévalue chaque attribut de cette liste. Si sa valeur est modifiée, on introduit dans la liste l'ensemble de ses successeurs (dans le graphe de dépendance). On itère jusqu'à ce que la liste devienne vide.

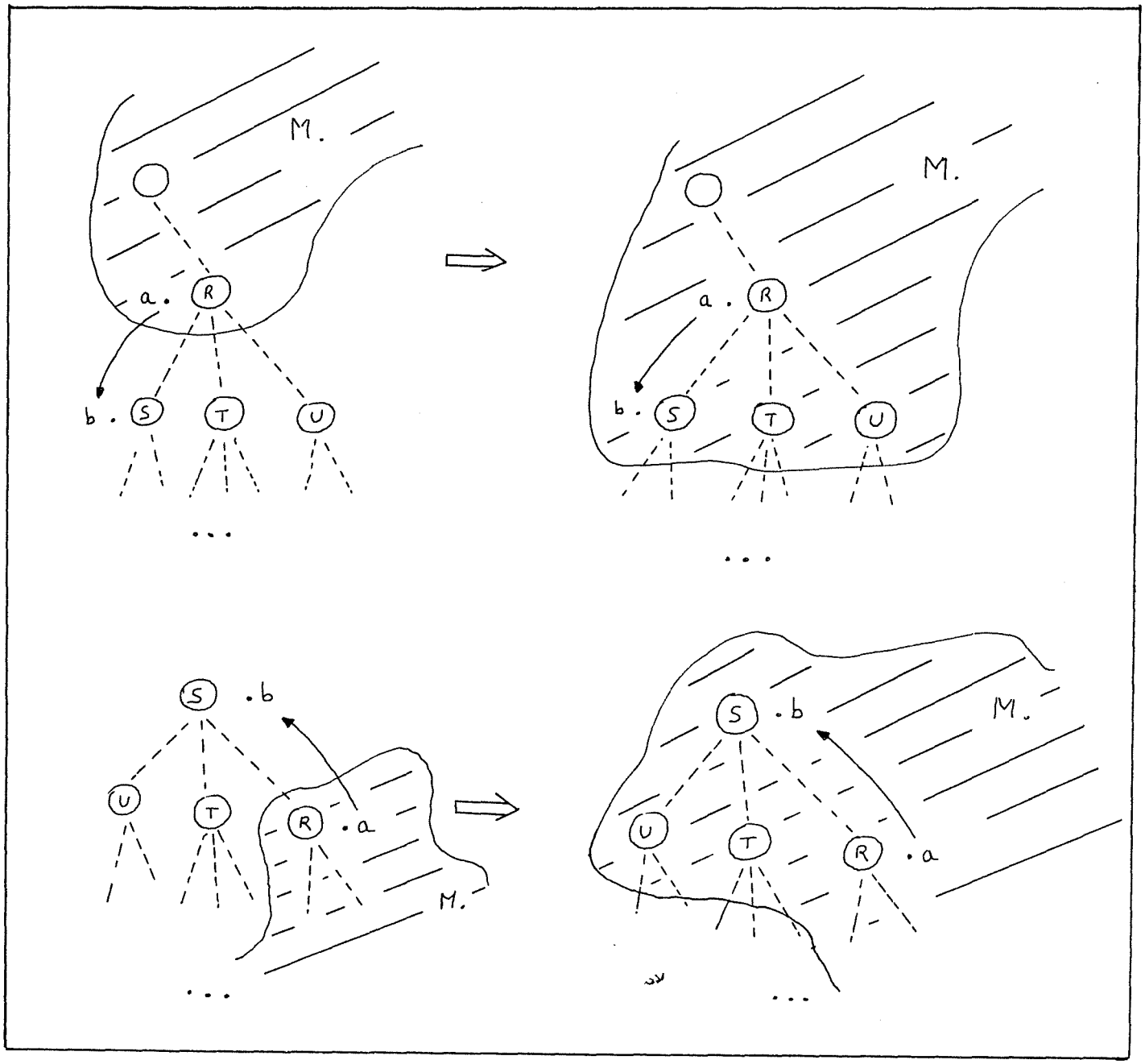
Le problème avec cette façon de travailler est qu'elle dépend de l'ordre dans lequel les attributs sont choisis pour être réévalués. Si les dépendances entre attributs sont suivies de manière aveugle, cette façon de procéder peut être non linéaire en le nombre d'attributs réévalués. En particulier, ce comportement non linéaire apparaît lorsque les attributs sont connectés par plus d'un chemin .

Ce comportement peut être visualisé dans l'exemple donné Fig. 20. Trois attributs synthétiques y sont définis :

l'attribut - a associé au type S
- b associé au type X
- c associé au type X

La Fig. 21 reprend un exemple d'arbre dérivé et des relations de dépendances existant entre les attributs correspondants.

Supposons que nous travaillons comme présenté ci-dessus et que la liste des attributs à réévaluer est mise à jour suivant un ordre LIFO. Modifions l'arbre présenté Fig. 21, en remplaçant l'instance de la production S : t par la production S : s. Faisons encore l'hypothèse qu'après la réévaluation d'un attribut a, les deux attributs successeurs sont insérés dans la liste de travail suivant l'ordre c puis a. Nous avons repris ci-dessous les différents états de cette liste au fur et à mesure que la réévaluation progresse.



Les parties hachurées représentent la portion de l'arbre couverte par M. R et S sont deux noeuds de l'arbre édité et a et b sont respectivement un de leurs attributs. L'attribut b dépend fonctionnellement de a.

Dans le haut de la figure, l'attribut b est associé à un noeud fils du noeud R. Suite à l'évaluation de a, le graphe M est étendu de la production

R -> T U S

Dans le bas de la figure, l'attribut b est associé au noeud père de R. Le graphe M est étendu de la production

S -> T U S

Fig. 22

- Principe de réévaluation des attributs -

{a1}, {c1, a2}, {c1, c2, a3}, {c1, c2}, {c1, b2},
{c1, a3}, {c1}, {b1}, {a2}, {c2, a3}, {c2}, {b2}, {a3}

En analysant l'ordre d'évaluation des attributs, on peut remarquer que le temps requis pour la réévaluation de l'arbre vérifie la relation :

$$T(h) = 2\,T(h-1) + \text{const.}$$

où h est la hauteur de l'arbre de dérivation.
Il dépend donc de manière exponentielle du nombre d'attributs à réévaluer.

(ii) **Réalisation**

L'idée à la base de l'algorithme de Reps est alors la suivante : si nous voulons réévaluer un nombre minimal d'attributs, il est primordial qu'un attribut ne soit réévalué que si nous obtenons ainsi sa valeur finale. Cette remarque suggère une énumération dans un ordre topologique des attributs de NOUVEAU d'après le graphe de dépendance correspondant et une réévaluation des attributs d'après cet ordre.

Ce principe est appliqué de la manière suivante : l'algorithme travaille avec une liste (L) ainsi qu'un graphe (M) de travail. L contient l'ensemble des attributs de degré intérieur zéro dans M (i.e. le nombre d'arcs du graphe aboutissant à ce sommet est nul). Il correspond à l'ensemble des attributs pouvant être réévalués de façon définitive. Le graphe M est construit dynamiquement au fur et à mesure que la réévaluation progresse. Au départ, ce graphe est constitué de l'ensemble des attributs localisés au noeud de la modification; un arc y représente une dépendance entre attributs correspondants (dépendance directe ou transitive via les autres attributs de l'arbre). A chaque réévaluation d'un attribut a de L, ce graphe M est étendu si la valeur de a est modifiée et s'il existe un attribut successeur de a n'apparaissant pas encore dans M. La Fig. 22 visualise ce procédé.Les arcs aboutissant en a sont également supprimés de M. La liste L est quant à elle mise à jour de la façon suivante : on supprime a de L et on y introduit les nouveaux sommets de degré intérieur zéro. L'algorithme s'arrête lorsque L devient vide.

Une discussion plus précise de cet algorithme et des différentes informations maintenues par l'éditeur en cours d'édition peut être trouvée en annexe 1.

2.3.5.3 Généralisation du modèle d'édition.

Jusqu'à présent, nous n'avons considéré que les opérations d'édition remplaçant un sous-arbre et déplaçant le curseur. L'insertion et la suppression d'un sous-arbre peuvent être considérées comme un cas particulier de cette dernière opération.

Cependant toutes les opérations d'edition dans un éditeur syntaxique n'incluent pas une seule opération de remplacement. Or l'algorithme présenté ci-dessus suppose que toutes les inconsistances sont localisées à un seul noeud de l'arbre. Cette hypothèse peut être abandonnée si le graphe M est initialisé dans l'algorithme à la plus petite région connectée de l'arbre incluant tous les noeuds affectés par une restructuration. De même, si nous abandonnons l'hypothèse de normalité, les attributs inconsistants ne sont plus localisés au noeud r de la modification mais peuvent apparaître dans les noeuds parents, frères et fils. Il suffit alors d'initialiser M à cette région.

2.3.6 Exemple PICO.

Terminons la présentation de l'approche par grammaire attribuée par les spécifications complètes de notre langage PICO.

Reprenons tout d'abord la spécification SSL de la grammaire abstraite de ce langage:

```
root PICO_PGM ;
PICO_PGM: pgmnull ()
      |pgmpair (DECLS,SERIES);

list DECLS
DECLS: declsnull ()
    |declspair (DECL,DECLS);

DECL: declnull ()
   |declpair (ID,TYPE);

TYPE: null ()
   |bool ()
   |int();

ID: id < [a-zA-Z] [ a-zA-Z0-9] * | [?] >

list SERIES
SERIES: serienull ()
     |seriepair (STAT,SERIES);
```

```
STAT: statnull ()
    |asgn (ID,EXP)
    |if (EXP,SERIES,SERIES);

EXP: nullexp()
    |plus (EXP,EXP)
    |eq (EXP,EXP)
    |idused (ID)
    |valbool (BOOL)
    |number (INT);
```

Pour permettre les contrôles sémantiques, nous allons utiliser deux types d'attributs différents: d'une part nous utiliserons l'attribut synthétique env qui contiendra l'ensemble des identificateurs déclarés dans le programme ainsi que leur type respectif. Cet attribut sera attaché à tous les opérateurs de phyla DECLS et PICO_PGM. D'autre part, nous utiliserons l'attribut synthétique type qui mémorisera le type de chaque expression. Il sera dès lors attaché à tous les opérateurs appartenant au phylum EXP. Cet attribut pourra prendre quatre valeurs différentes :

- null si l'expression n'est pas totalement dérivée,
- bool si l'expression est de type booléen,
- int si l'expression est de type entier,
- undefined si l'expression viole une de contraintes de type de notre langage PICO

Nous utiliserons également un attribut paire associé au phylum DECL qui synthétisera chaque déclaration d'identificateur.

Pour définir ces attributs, complétons tout d'abord notre grammaire abstraite avec les opérateurs et phyla nécessaires à la spécification de ces attributs:

```
list ENV
ENV: nullenv ()
    |envconcat (BINDING,ENV);

BINDING: binding (ID,TYPE);

TYPE: undefined ();
```

Muni de ces déclarations, nous pouvons définir les attributs:

```
PICO_PGM { synthesized ENV env ; } ;
DECLS    { synthesized ENV env ; } ;
EXP      { synthesized TYPE type ; } ;
DECL          ( synthesized BINDING paire ; ) ;
```

Remarquons que les attributs de type erreur qui serviront à imprimer des messages d'erreurs à l'écran seront déclarés comme locaux pour chaque opérateur concerné.

Avant de donner la liste des équations sémantiques définissant la valeur des attributs déclarés ci-dessus, spécifions encore la fonction lookup qui nous servira par la suite. Cette fonction reçoit en argument un identificateur et un environnement. Si l'identificateur est présent dans ce dernier, elle renvoie cet identificateur avec son type. Sinon elle renvoie la paire ("?" , undefined). Cette fonction nous servira à vérifier la non déclaration multiple d'un identificateur,l'utilisation d'identificateurs déclarés ainsi qu'à vérifier les propriétés de type.

```
BINDING lookup ( ID id , ENV env )
    /* Cette fonction détermine  s'il existe le premier binding(s,t) de env tel que id = = s . Dans le cas
contraire, elle renvoie binding("?",undefined). */

      {with(env) (nullenv: binding("?",undefined);
                  envconcat(b,e): with(b) (binding(s,t):(id = = s ? b : lookup(id,e)))
                  )
                  }
```

Terminons enfin par la liste des équations sémantiques. Les trois premières productions présentées concernent la construction de l'environnement et la non déclaration multiple d'un identificateur. L'environnement est construit par concaténation des différentes déclarations synthétisées par l'attribut "paire" (cfr règle (2)). Pour chaque déclaration d'un identificateur, une erreur est détectée si cet identificateur est déjà présent dans l'environnement (cfr règle (3)).

La vérification de la déclaration d'un identificateur utilisé est effectuée en (4) et (5) par un attribut "erreur" associé à tout opérateur "asgn" et "idused". Il prend la valeur " undefined" quand l'identificateur correspondant n'est pas présent dans l'environnement. Dans le cas contraire, cet attribut a la valeur d'un string vide.

Les autres équations concernent les vérifications de type. L'attribut "error2" associé à l'opérateur "asgn" (cfr règle (4)) prend la valeur " type mismatch" si le type de l'identificateur apparaissant dans la

partie gauche de l'arbre asgn (ID, EXP) est bool ou int et que le type de l'expression apparaissant dans le membre droit est bool ou int et que ces deux types sont différents (les autres cas sont corrects car ils correspondent soit à un identificateur dont le type est "null" (type non déclaré), ou "undefined" (identificateur non déclaré), soit à une expression non complètement dérivée (de type "null") ou erronée (de type "undefined, soit au cas normal).

L'attribut "error" associé à l'opérateur "if" prend la valeur "type error" si le type de l'expression y apparaissant a une valeur entière.

L'attribut "error" associé à l'opérateur "plus" prend la valeur "type mismatch" si une des expressions intervenant dans la somme est erronée ou qu'elle est de type booléen. Dans ce cas, l'attribut "type" associé prend la valeur "undefined". Sinon, si une de ces expressions n'est pas complètement dérivée, il prend la valeur nulle. Dans les autres cas, il prend la valeur int.

Enfin, l'attribut "error" associé à l'opérateur "eq" prend la valeur "type mismatch" si une des deux expressions apparaissant dans eq (EXP1, EXP2) est erronée ou que ces deux expressions sont complètement dérivées et de type différent. Dans ce cas, l'attribut "type" associé prend la valeur undefined. Sinon, si une des expressions n'est pas complètement dérivée, il prend la valeur nulle. Dans les autres cas, il prend la valeur bool.

```
(1) PICO_PGM: pgmpair { PICO_PGM.env = DECLS.env; };

(2) DECLS   : declsnull { DECLS.env = nullenv;};
     |declspair {DECLS.env(1) = DECL.paire# DECLS.env(2);};

(3) DECL:  declnull {DECL.paire = binding ("?", undefined);};
      | declpair {local STR error;
              local BINDING b;
              b = lookup(ID,PICO-PCM.env);
              error = with(b) (binding(s,t): s = = "?" ? "" : "<-Multiple Defined") ;
             DECL.paire = binding (ID, TYPE)};};
```

```
(4) STAT: asgn  {local STR error1,error2;
                 local BINDING b;
                 b = lookup(ID,PICO_PGM.env);
                 error1 = with(b) (binding(s,t): s = = "?" ? "<-Undefined") : "" ;};
                 error2 = with(b) (binding(s,t): ((t= =bool and EXP.type= =int)
                      or (t= =int and EXP.type= =bool ? :" type mismatch" : " " ;};
   |if { local STR error;
      error = EXP.type != int ? "" : "<-type error";};

(5) EXP: nullexp { EXP.type = null;};
              |plus {local STR error;
               local BOOL a;
               a = (EXP.type(2) = = undefined) or (EXP.type(3) = = undefined)
                    or (EXP.type(2) = = bool) or (EXP.type3 = = bool) ;
        EXP.type(1) = a ?  undefined:((Exp.type(2)= =null)       or (Exp.type(3)= =null)) ? null:int
        error = a ?  "<-Type mismatch" :  "";
       };
   |eq  {local STR error;
      local BOOL a;
      a = (EXP.type(2) = = undefined)
        or (EXP.type(3)= =undefined)
        or((EXP.type(2) = = bool) and (Exp.type(3) = = int))
        or ((EXP.type(3) = = bool) and (Exp.type(2) = = int))
       EXP.type(1) = a ? undefined : ((Exp.type(2) = = null or (Exp.type(3) = = null)) ? null : bool
      error = a ?  "<-Type mismatch" : "" ;
       };
   |idused {local STR error;
      local BINDING b;
      b = lookup(ID,PICO_PGM.env);
      error = with(b) (binding(s,t): s = = "?" ? "<-Undefined") : "" ;
      idused.type = with (b) (binding(s,t):s= ="?"
            ? undefined : t };
   |valbool { valbool.type = bool;};
   |number { number.type = int ;};
```

2.3.7 Remarques.

Lors de la description du "Synthesizer Generator", nous avons remarqué que cet éditeur permettait à la fois une édition à entrée structurée et à entrée libre. Jusqu'à présent, nous n'avons discuté que de l'édition à entrée structurée.

D'après nos sources [Reps et al.,87], il semblerait que le passage d'un mode d'édition à l'autre s'effectue de la manière suivante : l'utilisateur peut à tout moment demander l'édition d'un sous-arbre de son programme en mode texte. Le texte de ce dernier est alors mémorisé temporairement dans un buffer texte. Lorsque l'utilisateur redemande l'autre mode d'édition, cette partie de son programme est restructurée et une analyse syntaxique s'effectue. L'utilisateur est prié de corriger toutes les erreurs détectées. Lorsque cette tâche est terminée, les attributs sont seulement réévalués et les messages d'erreurs correspondants sont imprimés à l'écran.

Remarquons également que le "Synthesizer Generator" ne fait que détecter des erreurs sémantiques. Bien que le maintien de programmes sémantiquement corrects ne soit en général pas adapté à une édition à entrée structurée, notons cependant que cette propriété pourrait être facilement intégrée au "Synthesizer Generator" : il suffirait en effet de simuler après chaque commande utilisateur la réévaluation des attributs. Si un attribut error prend une valeur non vide, la commande est alors annulée.

2.4. Approche par Sémantique Naturelle.

2.4.1 Objectifs.

L'approche sémantique naturelle fut développée dans le cadre des recherches à l'INRIA [Donzeau-Gouge et al., 80]. De manière générale , la sémantique naturelle est un formalisme de spécification de la sémantique des langages de programmation. Il s'agit d'un formalisme exécutable. TYPOL est le langage qui l'implémente.

Les programmes TYPOL sont conçus pour s'effectuer dans le cadre de l'environnement MENTOR [Donzeau-Gouge et al., 83]. Ils s'exécutent à la requête de l'utilisateur et analysent des programmes dont l'édition est terminée. Pour cette approche le type d'édition n'a donc pas d'importance: en effet, un programme TYPOL analyse un arbre syntaxique abstrait (conforme à une grammaire abstraite) indépendamment du type d'édition.

L'approche par sémantique naturelle combine donc les quatres axes de liberté suivants:

- édition à entrée structurée ou à entrée libre;
- simple contrôle de la validité sémantique des objets édités;
- contrôle effectué à la requête de l'utilisateur;
- contrôle sur des programmes terminés.

2.4.2. Présentation générale des concepts de base.

L'idée générale de la sémantique naturelle est de caractériser les différentes propriétés pouvant être exprimées à propos d'un langage par un ensemble de règles d'inférence et d'axiomes.

Par exemple, dans le cadre d'une vérification de type, on pourrait axiomatiser la propriété que l'expression E est de type T dans l'environnement env (où env contient un ensemble d'affirmations sur les types des variables de E) par la formule :

env |- E : T

De même, on pourrait exprimer qu'une expression E1 dans le langage source L1 est traduite en l'expression L2 du langage objet L2 par :

env |- E1 -> E2

où env mémorise un ensemble d'hypothèses sur les identificateurs.

Un autre exemple , en sémantique dynamique, est le suivant:

s1 |- E = > s2

Il exprime que l'évaluation d'une expression E pour un état de la mémoire s1 conduit à un nouvel état s2.

Une définition sémantique est constituée d'un ensemble de règles d'inférence et d'axiomes définissant un des prédicats présentés ci-dessus. Chaque règle exprime une propriété d'une construction du langage, en terme de propriétés de ses composants.

Ces règles et axiomes permettent alors de résoudre des équations telles que: étant donné un environnement env, est-il possible d'assigner à l'expression E le type T de sorte que env |- E : T soit vrai?

Remarquons que la question pourrait être renversée en: étant donné une expression E et un type T, existe-t-il un environnement env telque env |- E : T soit vrai? On se trouve alors devant un problème d'inférence de type.

Comme souligné dans [Kahn, 87], il existe un parallélisme entre la sémantique naturelle et le mode de spécification de la syntaxe d'un formalisme : en effet, une grammaire est constituée d'un ensemble de règles. Ces règles définissent les arbres syntaxiques légaux et de ce fait les phases légales du langage. De manière analogue, les axiomes et les règles d'inférence définissent les arbres de preuve légaux et par conséquent les faits qui peuvent être dérivés de ces règles et axiomes. Une grammaire peut être utilisée de deux manières : pour générer l'ensemble des phases légales ou pour reconnaître les phases valides. De manière similaire, une description sémantique peut être vue comme générant un ensemble de faits ou comme décrivant des calculs possibles. Aux algorithmes généraux d'analyse syntaxique correspondent les interpréteurs de définition sémantique.

2.4.3. Langage de spécification des contrôles.

Dans ce paragraphe, nous allons présenter les principales caractéristiques du langage TYPOL. En vue de faciliter la présentation, nous avons choisi d'introduire de manière progressive les différents éléments du formalisme. L'exemple PICO détaillé en 2.4.3 concrétise cet exposé.

2.4.3.1. Syntaxe abstraite.

Tous les objets manipulés par TYPOL sont des arbres syntaxiques abstraits. Ils sont décrits par une syntaxe abstraite. Le mode de spécification de cette syntaxe est celui de MENTOR [Kahn et al, 83]: chaque opérateur appartient à un phylum. Il peut être d'arité fixe ou variable. Dans le premier cas, cet opérateur est spécifié par la liste des phyla des opérateurs fils ; dans le deuxième cas, il est spécifié par l'unique phylum auquel ses opérateurs fils doivent appartenir.

A titre d'exemple, les spécifications suivantes

```
pico-pgm -> DECLS SERIES;

decls -> DECL +...;


DECLS ::= decls

DECL ::= decl

SERIES ::= series
```

déclarent que l'opérateur 'pico-pgm' a deux fils appartenant respectivement aux classes DECLS et SERIES. Celles-ci sont constituées des opérateurs 'decls' et 'series'. La deuxième production déclare l'opérateur 'decls' comme étant une liste d'opérateurs 'decl'.

2.4.3.2. **Les termes TYPOL.**

La sémantique naturelle utilise les notions de "pattern matching" et d'unification : deux formules sont unifiables si elles sont identiques symbole par symbole à des substitutions de variables près et si les variables y apparaîssant sont de même type.(Nous reviendrons plus en détail sur les variables TYPOL en 2.4.3.6.)

Dans les règles TYPOL, les arbres sont décrits sous forme linéaire : les opérateurs d'arité fixe sont écrits en notation préfixée. Ainsi, on désigne un arbre de racine 'pico-pgm' en écrivant :

pico-pgm (DECLS, SERIES)

Dans cette formule, les variables DECLS et SERIES désignent respectivement le premier et le second fils du noeud pico-pgm (par convention, nous utiliserons pour une variable le même nom que celui du phylum correspondant). Dans une même règle, on pourra faire référence à ces composants en écrivant uniquement DECLS et SERIES (cfr 2.4.3.5).

Pour les opérateurs d'arité variable, il existe une notation spéciale. Il est possible d'écrire des termes désignant un nombre fixe d'éléments.

L'exemple suivant désigne une liste de déclarations vide :

decls [ ].

La notation

decls [A, B, C]

désigne quant à elle une liste de trois déclarations (désignées par les variables A, B, C).

Il est aussi possible d'isoler certains éléments au début de la liste du reste de la liste, par l'utilisation d'un point :

decls [A.Q].

Dans cet exemple, A correspond à un élément de liste, tandis que Q correspond à une sous-liste (de déclarations).

Ces notations présentent l'avantage de permettre l'analyse d'un arbre ou sa construction de façon unifiée : les variables apparaissant dans un arbre désignent des sous-arbres. Nous pouvons faire référence à ces sous-arbres en nommant uniquement la variable correspondante ; nous pouvons aussi reconstruire un arbre en écrivant un nouveau terme.

2.4.3.3. Les propositions.

En sémantique naturelle, la vérification d'une propriété associée à un langage revient à prouver une proposition particulière. Le langage TYPOL offre différentes manières d'écrire des propositions :

- le style le plus traditionnel est la forme

P(x)

où P est un nom de proposition et x une liste de termes TYPOL.
Par exemple, "EST-CORRECTE(pico-pgm (DECLS, SERIES))" est une proposition TYPOL.

- TYPOL offre aussi un ensemble d'opérateurs relationnels infixés prédéfinis comme propositions (->, =>, :, ?, ...). Aucune signification particulière n'est associée à priori à ces symboles. Par exemple, on peut décider que

x : t

signifie "x est de type t".

- les propositions anonymes sont des propositions non nommées. Elles n'ont pas de symboles de prédicat associé. Comme pour les relations, la signification de ces propositions n'est pas fixée à priori. La forme générale de ces propositions est

x

où x est une liste de termes TYPOL.

2.4.3.4. Les séquents.

Les séquents espriment que certaines hypothèses sont nécessaires à la preuve d'une proposition particulière. Un séquent a deux parties : la première contient les hypothèses ; il s'agit d'une liste de termes TYPOL. La deuxième appelée conséquent est une proposition. Ces deux parties sont séparées par le symbole '|-'.

Voici quelques exemples de séquents :

|- pico-pgm (DECLS, SERIES)
env [ ] |- DECLS -> e
e |- EXP:t

Leur signification intuitive est respectivement :
. le programme pico-pgm (DECLS, SERIES) est correct
. étant donné un environnement vide, la liste des déclarations DECLS produit un environnement e
. l'expression EXP a le type t dans l'environnement e

Les différentes formes de séquents apparaissant dans une définition sémantique sont appelées des jugements. Les différents jugements sont utilisés sans nom explicite. Le symbole |- est donc "surchargé".

2.4.3.5. **Les axiomes et règles d'inférence.**

Une règle d'inférence explique quand un séquent peut être déduit d'autres séquents. Elle est composée de deux parties : un numérateur et un dénominateur. Le numérateur consiste en un ensemble de séquents tandis que le dénominateur consiste en un séquent. Une règle avec un numérateur vide est appelée un axiome. Intuitivement, une règle d'inférence dit que si tous les séquents du numérateur sont valides, alors le dénominateur est valide.

Voici quelques exemples de règles d'inférence :

env [ ] |- DECLS:e & e |- SERIES
|- pico-pgm (DECLS, SERIES)

Cette règle dit qu'un programme PICO est valide si, étant donné un environnement vide, la partie déclaration produit un environnement e et si, en utilisant cet environnement, le corps du programme est valide (il n'y a pas d'erreurs de type,...).

. e |- decls [ ] : e

Cet axiome dit qu'une liste vide de déclarations ne modifie pas l'environnement.

e |- DECL : e1 & e1 |- DECLS : e2
e |- decls [DECL.DECLS] : e2

Cette dernière règle explique qu'en PICO, l'élaboration des déclarations est réalisée de manière linéaire.

2.4.3.6. Les variables.

Les variables TYPOL sont typées par un nom de phylum. Elles ne peuvent être instanciées qu'à un sous-arbre d'opérateur racine appartenant au phylum correspondant. Elles ne doivent être déclarées que si leur type ne peut être inféré d'après leur utilisation (i.e. d'après leur position dans un terme). Il y a une différence entre un nom de variable et une variable. Un nom de variable est local à une règle d'inférence ou à un axiome, tandis qu'une variable survit en dehors d'une règle. Différentes occurences d'une même variable dans une règle signifient que ces variables doivent être unifiées.

2.4.3.7. Spécifications sémantiques.

Une spécification sémantique consiste en un ensemble non ordonné de règles d'inférence et d'axiomes. Ces règles définissent un système formel dans lequel il est possible de prouver qu'une proposition particulière est valide.

Il est possible de modulariser une spécification en regroupant des règles dans des ensembles. Chaque ensemble est un système formel complet et porte un nom. Les séquents dans le numérateur d'une règle font référence à l'ensemble local considéré (un programme est un cas particulier d'ensemble). Il est possible de nommer explicitement un autre ensemble à utiliser. Ceci permet de passer d'un système formel à un autre. Différents exemples d'ensembles peuvent être trouvés en 2.4.4.

2.4.3.8. Les actions.

Il est également possible d'attacher des "actions" aux règles TYPOL. Ces "actions" sont activées après qu'une règle d'inférence ait été considérée comme applicable. Une "action" peut accéder à la valeur d'instanciation des variables , mais elle ne peut en aucune façon interférer avec le processus de déduction. Typiquemnt, ces "actions" sont utilisées pour faire la trace des règles d'inférence, pour envoyer des messages... Remarquons que le processus d'inférence peut effectuer des "backtrakking" et que bien qu'une règle ait été choisie à un certain moment, elle peut être délaissée par la suite et donc ne pas faire partie de la preuve globale.

2.4.4 Intégration dans un modèle d'édition.

Comme nous l'avons déjà spécifié, aucun modèle d'édition précis n'est nécessaire à l'utilisation d'un programme TYPOL: la vérification sémantique s'effectue en fin d'édition à la requête de l'utilisateur.

Concrètement, les définitions sémantiques sont transformées en code exécutable en les compilant en PROLOG. A l'origine, le compilateur était écrit en PASCAL. Actuellement, il est écrit en TYPOL. Chaque règle sémantique est compilée en une ou plusieurs clauses PROLOG. De manière générale, la conclusion d'une règle correspond à la tête d'une clause PROLOG tandis que les prémices correspondent à la queue. Chaque jugement distinct est traduit en un prédicat distinct.

Comme l'ordre dens lequel les séquents sont spécifiés au numérateur d'une règle n'a pas d'importance, l'utilisation de PROLOG standard s'est révélée peu adéquate: en effet, il fallait avoir la possibilité de postposer la résolution d'un objectif jusqu'à ce que les variables soient instanciées. Dès lors, le choix s'est porté sur MU-PROLOG qui répondait à ces exigences.

2.4.5 Exemple PICO.

Reprenons notre exemple PICO et voyons comment les contrôles sémantiques sont spécifiés en sémantique naturelle.

Tout d'abord, respécifions la grammaire abstraite de ce langage telle qu'elle est définie dans le système MENTOR:

```
pico-pgm -> DECLS SERIES ;

decls   -> DECL +... ;

decl    -> ID TYPE ;

series  -> STAT +... ;

asgn    -> ID EXP ;

if      -> EXP SERIES SERIES ;

plus    -> EXP EXP ;

eq      -> EXP EXP ;

bool    -> implemented as STRING ;

int     -> implemented as STRING ;

id      -> implemented as identifier ;
```

number -> implemented as INTEGER ;

valbool -> implemented as STRING ;

PICO-PGM ::= pico-pgm ;
DECLS ::= decls ;
DECL ::= decl ;
ID ::= id ;
TYPE ::= bool int ;
SERIES ::= series ;
STAT ::= asgn if ;
EXP ::= plus eq id number val-bool ;

Spécifions également la syntaxe abstraite de l'environnement qui nous servira aux contrôles :

env -> BINDING + ... ;
binding -> ID TYPE ;

BINDING ::= binding ;

Les règles spécifiant les contrôles sémantiques sont reprises ci-dessous.

Les variables en majuscule sont des variables dont le type peut être inféré d'après le contexte. Nous les avons désignées par le nom de leur classe. Les variables en minuscule supposent les déclarations suivantes:

e1,e2,e : ENV ;
u1,u2,u : TYPE ;

Nous avons défini cinq ensembles: le premier DECLARE ajoute une déclaration dans l'environnement. Le deuxième TYPE_OF établit le type d'un identificateur s'il a été déclaré. Les trois derniers ensembles COMP, BOOL et INT établissent respectivement si deux types sont égaux, si un type est booléen et si un type est entier.

R1: cette règle exprime qu'un programme en langage PICO est valide si, étant donné un environnement vide, la partie déclaration produit un environnement e et si d'après cet environnement, le corps du programme est valide.

env[] |- DECLS : e & e |- SERIES
|- pico_pgm (DECLS, SERIES)

R2: cet axiome dit qu'une liste vide de déclarations ne modifie pas l'environnement.

e |- decls[] : e

R3: en PICO, l'environnement est construit de manière linéaire.

e |- DECL : e1 & e1 |- DECLS : e2
e |- decls[DECL.DECLS] : e2

R4: l'insertion d'une déclaration dans l'environnement se fait par l'ensemble des règles DECLARE (spécifié ci-dessous).

DECLARE (e |- decl (ID,TYPE) : e1)
e |- decl (ID,TYPE) : e1

R5: une liste vide d'instructions est un corps de programme valide.

e |- series[]

R6: la vérification des instructions du programme est effectuée de manière linéaire

e |- STAT & e |- SERIES
e |- series[STAT.SERIES]

R7: l'assignation d'une expression EXP à un identificateur ID est valide si le type de cet identificateur (i.e. u1) est le même que celui de EXP (i.e. u2). La sémantique du segment :

e |- ID: u

est que ID a le type u dans l'environnement e.

e |- ID : u1 & e |- EXP : u2 & COMP(u1,u2)
e |- assign (ID,EXP)

R8: la somme de deux expressions entières est de type entier .

e |- EXP1 : u1 & e |- EXP2 : u2 & IS_INT(u1) & IS_INT(u2)
e |- plus (EXP1, EXP2) : int

R9: l'égalité de deux expressions de même type est une expression de type booléen.

e |- EXP1 : u1 & e |- EXP2 : u2 & COMP(u1,u2)
e |- eq (EXP1, EXP2) : bool

R10: une construction conditionnelle est valide si l'expression y apparaîssant est de type booléen et si les deux constructions SERIES sont valides.

e |- EXP : u1 & e |-SERIES1 & e |- SERIES2 & IS_BOOL(u1)
e |- if (EXP , SERIES1 , SERIES2)

R11: le type d'un identificateur est établi par l'ensemble des règles TYPE-OF.

TYPE-OF (e |- id(X) : u)
e |- id(X) : u

R12: le type d'un nombre est entier.

e |- number(X) : int

R13: le type de la constante "true" ou "false" est booléen.

e |- val_bool(X) : bool

set DECLARE is:

cet ensemble de règles ajoute une déclaration en fin de l'environnement, si l'identificateur déclaré n'y est pas déjà présent.

D1: env[] |- decl(id(X),u) : env [binding (id(X), u)]

D2: Le prédicat DIFF(X,Y) utilisé ci-dessous correspond à une "condition à effet de bord" . De telles conditions sont utilisées en TYPOL quand une contrainte ne peut être exprimée en utilisant un simple pattern.

$$\frac{E \mid\!- \text{decl}(\text{id}(X),u) : L1}{\text{env}[\text{binding}(\text{id}(Y), u1).E] \mid\!- \text{decl}\ (\text{id}(X),u) : \text{env}[\text{binding}(\text{id}(Y),u1).L1]}\ \text{DIFF}(X,Y)$$

end DECLARE

set TYPE_OF is :

cet ensemble de règles associe à un identificateur mémorisé dans l'environnement son type.

t1: env [binding (id(X), u).E] |- id(X) : u

t2:

$$\frac{E \mid\!- \text{id}(X) : u}{\text{env}[\text{BINDING.E}] \mid\!- \text{id}(X) : u}$$

end TYPE_OF

set COMP is

|- u,u

end COMP

set IS_BOOL is

|- bool

end IS_BOOL

set IS_INT is

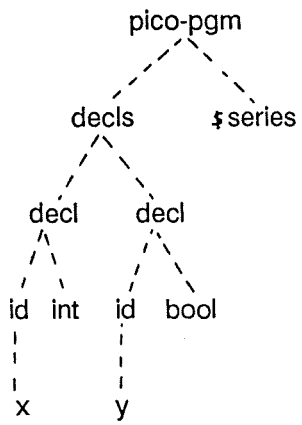
|- int

end IS_INT

A titre d'illustration, considérons un programme composé de deux déclarations :

```
program

declare     x : int,
            y : bool;
end.
```

L'arbre abstrait y correspondant est :



Nous pouvons le réécrire sous la forme linéaire suivante:

pico-pgm(decls[decl(id(x),int),decl(id(y),bool)],series[])

Vérifier la validité de ce programme revient alors à établir l'objectif :

G1: |- pico-pgm(decls[decl(id(x),int), decl(id(y),bool)],series[])

La règle R1 nous dit que pour cela il faut établir les sous-objectifs :

G2: env[] |- decls[decl(id(x),int),decl(id(x),bool)] : e
G3: e |- series [ ]

Remarquons l'instanciation de la variable DECLS au sous-arbre gauche de l'opérateur pico-pgm, ainsi que celle de SERIES au sous-arbre droit.

L'objectif G2 ne peut être établi par la règle R2 puisque la liste des déclarations n'est pas vide. Nous utiliserons dès lors la règle R3 avec les instanciations:

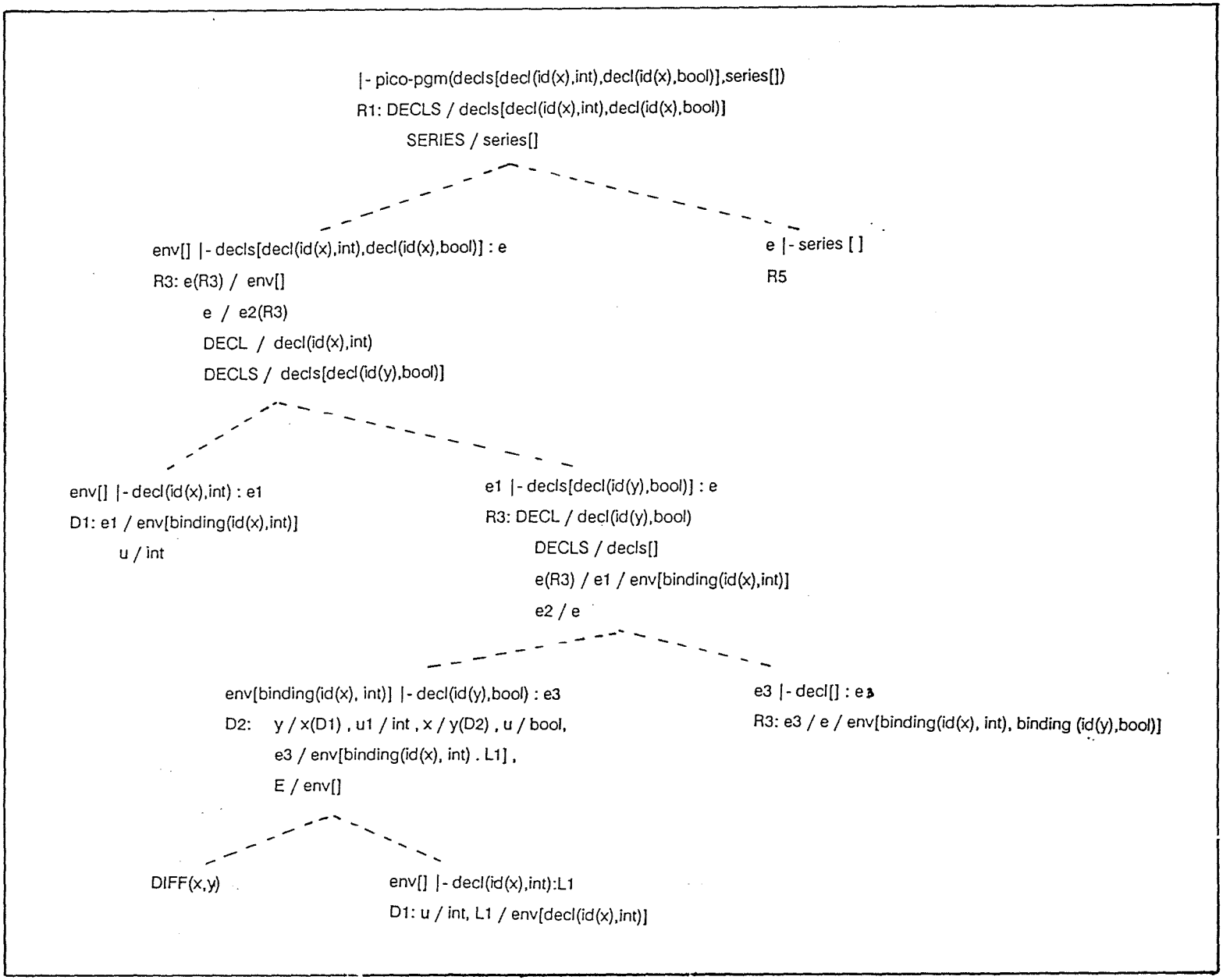


Fig. 23

- Exemple d'arbre de preuve pour un programme PICO -

e(R3) / env[]

e(G2) / e2(R3)

DECL / decl(id(x),int)

DECLS / decls[decl(id(y),bool)]

Nous avons alors à établir les deux sous-objectifs :

G4: env[] |- decl(id(x),int) : e1

G5: e1 |- decls[decl(id(y),bool)] : e

La règle R4 nous dit que l'objectif G4 est établi par le système formel constitué par l'ensemble DECLARE. La première règle de ce dernier établit G4. Nous obtenons par la même occasion l'instanciation :

e1 / env [binding(id(x), int)]

L'objectif G5 revient donc à:

G5: env[binding(id(x), int)] |- decls[decl(id(y),bool)] : e2
Il est établi par l'emploi des règles R2, R3 et par les règles de DECLARE.

Le sous-objectif G3 :
env[binding(id(x), int), binding(id(y, bool))] |- series [ ]
est établi quant à lui par la règle R5.

Ceci termine la preuve de G1.

La Fig.23 reprend l'arbre de preuve correspondant à cet exemple. Chaque noeud fils correspond à un sous-objectif à établir en vue d'établir l'objectif correspondant au noeud père. Nous avons repris en dessous de chaque noeud le nom de la règle d'inférence ou de l'axiome utilisé pour construire l'arbre ainsi que les instanciations effectuées.

2.4.6. **Remarques.**

Les contrôles en sémantique naturelle s'effectuent sur des programmes dont l'édition est terminée. Remarquons que cette restriction pourrait être facilement supprimée : il suffirait en effet, tout comme dans le cadre d'une grammaire attribuée, d'ajouter à chaque phylum un opérateur complémentaire. Chaque groupe de règles définissant la condition de validité d'un séquent (pour

chaque instance d'un opérateur d'un phylum précis) serait alors augmenté d'une règle. Celle-ci préciserait la condition de validité du séquent pour cet opérateur particulier.

A titre d'illustration, reprenons les règles définissant la validité du séquent e |- STAT (règles R7 - R10).

Ajoutons - au phylum STAT, l'opérateur complémentaire (d'arité nulle) statnull.
- au phylum TYPE, l'opérateur complémentaire null.

Les règles R7 et R10 deviennent alors les suivantes :

R7 :

$$\frac{e \mid\!- ID : u1 \;\&\; e \mid\!- EXP : u2 \;\&\; COMP(u1,u2)}{e \mid\!- assign\ (ID,EXP)}$$

R7': une assignation est valide si le type de l'identificateur n'a pas encore été édité.

$$\frac{e \mid\!- ID{:}u1 \;\&\; e \mid\!- EXP{:}u2 \;\&\; COMP(u1,\ null)}{e \mid\!- assign(ID,\ EXP)}$$

R7'' : une assignation est valide si l'expression y intervenant n'est pas complètement dérivée :

$$\frac{e \mid\!- ID{:}u1 \;\&\; e \mid\!- EXP{:}u2 \;\&\; COMP(u2,\ null)}{e \mid\!- assign(ID,\ EXP)}$$

R10:

$$\frac{e \mid\!- EXP : u1 \;\&\; e \mid\!- SERIES1 \;\&\; e \mid\!- SERIES2 \;\&\; IS\_BOOL(u1)}{e \mid\!- if\ (EXP\ ,\ SERIES1\ ,\ SERIES2)}$$

R10' : une construction conditionnelle est valide si l'expression y apparaisant n'est pas complètement dérivée et si les deux constructions SERIES sont valides.

$$\frac{e \mid\!- EXP : u1 \;\&\; e \mid\!- SERIES1 \;\&\; e \mid\!- SERIES2 \;\&\; COMP(u1,null)}{e \mid\!- if\ (EXP\ ,\ SERIES1\ ,\ SERIES2)}$$

CHAPITRE 3 : COMPARAISON DES APPROCHES

Dans ce chapitre, nous allons comparer les trois approches présentées précédemment.

A cet effet, nous allons repartir de l'exemple PICO qui nous a déjà servi à illustrer chacune de ces méthodes. Rappelons que ce langage a été défini au paragraphe 1.2 et a servi d'illustration dans les paragraphes 2.2.6, 2.3.6 et 2.4.5.

Notons tout d'abord que les contraintes sémantiques associées à ce langage reprennent les trois premiers types de contrôles mis en évidence au paragraphe 1.3., à savoir
- l'unicité des identificateurs déclarés
- l'utilisation d'identificateurs déclarés
- l'utilisation de ces identificateurs en accord avec leurs propriétés déclarées (contrôle de types).

En terme d'arbre abstrait, ces contraintes se matérialisent par des contrôles portant sur différents sous-arbres : le sous-arbre des déclarations de racine <decls>, les sous-arbres d'expressions de racine <exp> ainsi que les sous-arbres correspondant à des instructions de racine <stat>.

Pour chacune des méthodes, l'idée à la base des contrôles est la même : construire un environnement synthétisant l'ensemble des identificateurs déclarés ainsi que les propriétés de type définies pour ces identificateurs. Les contrôles concernant l'unicité des identificateurs déclarés sont effectués lors de la construction de cet environnement. Les deux autres types de contrôles sont réalisés par consultation de cet environnement. Remarquons déjà que nous retrouverons la même démarche d'analyse lors de notre étude de contrôles sémantiques dans l'atelier ALMA présentée dans la deuxième partie de ce mémoire.

Dans les trois approches, cet environnement (ou toute autre structure de données auxiliaire mémorisant de l'information sur l'arbre) diffère essentiellement par deux aspects : son intégration dans l'arbre abstrait et sa méthode de consultation. Dans l'approche par routine sémantique et par grammaire attribuée, l'environnement est intégré dans l'arbre du programme sous forme d'attribut. Par contre dans l'approche par sémantique naturelle, les structures de données auxiliaires sont mémorisées de manière indépendante de l'arbre. Dans cette approche, les vérifications syntaxiques et sémantiques sont en effet dissociées.

D'autre part, dans l'approche par routines sémantiques les attributs sont consultés et mis à jour par n'importe quelle routine. En ce sens, on peut dire que cette approche utilise des structures de données globales. Par contre, dans l'approche par grammaire attribuée, la valeur d'un attribut ne peut être dérivée que localement à partir de la valeur d'autres attributs de la production. Toute la

connaissance sémantique du programme est concentrée localement au travers de ces attributs. Remarquons cependant que, dans la dernière version du "synthesizer generator", cette contrainte a été diminuée en permettant de faire référence à la première instance d'un attribut d'un noeud ancêtre du noeud considéré (cfr 2.3.4.3)

Dans les trois approches, les structures de données auxiliaires sont représentées sous forme d'arbres abstraits. On a donc unification des domaines syntaxiques et sémantiques.

Ce mode de représentation présente l'avantage d'être très souple et très adaptable aux besoins. Il permet également de soumettre les structures de données aux mêmes contrôles de validité structurelle que l'arbre édité. On retrouve cet aspect dans l'approche par routine sémantique où la mise à jour des attributs de l'arbre est directement contrôlée par l'éditeur au même titre qu'une commande d'édition d'un utilisateur.

Dans l'approche par grammaire attribuée, ce choix a permis de calculer la valeur d'un attribut par référence syntaxique à un sous-arbre de l'arbre abstrait édité. Comme nous l'avons déjà souligné en 2.3.4., si le mode de représentation des attributs était différent de celui adopté pour la représentation des programmes édités, une transformation du domaine syntaxique dans le domaine sémantique serait nécessaire.

De manière générale, ce mode de représentation permet l'exploitation des structures de données de façon uniforme par n'importe quel outil guidé par la syntaxe des différents formalismes présents dans l'arbre attribué. Par exemple, un interpréteur pourrait directement exploiter les informations présentes dans l'environnement pour exécuter le programme édité. D'autre part, dans le cadre d'un langage où les types des variables doivent être inférés (comme par exemple dans le langage SETL [Donzeau-Gouge, 87]), les attributs pourraient être utilisés pour mémoriser les informations nécessaires à un optimiseur.

Rappelons que ce même principe se retrouve dans le contexte général d'une information structurée (où peuvent se rencontrer des formalismes différents au sein d'un même arbre). Ainsi, un manuel de référence d'un langage peut être vu comme composé des formalismes suivants : les titres et sous-titres de l'ouvrage, les paragraphes décrivant le langage, les exemples écrits dans ce langage ainsi que la description syntaxique de celui-ci. De même, l'arbre principal peut être un programme écrit dans un formalisme pour la réalisation de gros projets (i.e. spécialisé dans la description de l'interconnection de modules) tandis que certaines feuilles sont des modules écrits dans différents langages de programmation.

Si nous étudions plus en profondeur le travail de conception des contrôles dans chaque approche, nous pouvons mettre en évidence les points suivants :

(i) dans l'approche par **routines sémantiques**, les contrôles s'effectuent par parcours de l'arbre, consultation et mise à jour des attributs.

Différentes difficultés nous sont apparues dans l'application de cette méthode d'analyse :

- d'abord, la description du parcours de l'arbre nécessite une connaissance de sa structure. Celle-ci peut varier d'après l'état d'édition de l'arbre (existence de noeuds meta) et d'après les opérateurs choisis dans chaque classe (induisant différentes formes d'arbres). Ceci contraint le concepteur des contrôles d'anticiper toutes les situations pouvant se présenter. Une autre conséquence est le fait de retrouver une grande partie de la description de la syntaxe abstraite dans la description des contrôles.

- Cette dernière est également alourdie par la nécessité d'y inclure une analyse du contexte permettant d'identifier le cadre d'activation d'une routine : ceci concerne non seulement l'analyse du signal (cfr 2.2.5.2.) mais est également lié au fait qu'une routine est associée à un opérateur indépendamment des règles grammaticales. Comme un opérateur peut apparaître et jouer différents rôles dans l'arbre abstrait, il faut dès lors inclure dans la description de la routine une analyse du contexte. Ainsi, par exemple, un opérateur <id> dans notre langage PICO peut correspondre à une déclaration ou une utilisation d'un identificateur. Selon l'un ou l'autre cas, le travail effectué par la routine diffère. Dès lors, le premier travail réalisé par la routine associée est d'analyser le noeud père de l'opérateur courant.

- Une autre difficulté dans l'application de cette méthode d'analyse est de tenir compte du dynamisme de l'édition : puisque l'ordre dans lequel l'arbre est édité n'est pas fixé à priori, le concepteur des contrôles doit concevoir une stratégie de mise à jour des structures de données de telle sorte qu'elles soient à tout moment en cohérence avec l'état de l'arbre abstrait. Ceci implique la modification des structures à la fois lors de l'insertion et de la suppression de noeuds. Pour des contrôles complexes, la conception des contrôles qui en résulte peut devenir rapidement compliquée. Le travail effectué par les routines peut aussi se révéler important. Ainsi , lors de la modification de la déclaration d'un identificateur dans notre exemple PICO, une réévaluation de tous les attributs "type" et "status" des expressions et instructions doit être réalisée.

- En outre, dans cette approche une erreur dans la spécification des contrôles pourrait conduire à une détection d'erreurs sémantiques dépendant de l'ordre dans lequel l'arbre est édité. Ainsi par exemple, si un programme est édité dans un ordre ascendant, il pourrait être considéré comme correct, alors qu'une édition descendante amènerait la détection d'erreurs.

- Enfin, rappelons qu'une routine dispose d'un statut similaire à l'utilisateur quant à l'édition de l'arbre. Ceci implique que le travail de mise à jour d'une routine peut entraîner l'activation d'une ou

plusieurs routines et ceci récursivement. Deux problèmes peuvent alors surgir : d'une part, l'enchaînement des routines pourrait provoquer une modification incohérente de l'arbre (par exemple, destruction d'un sous-arbre contenant le noeud courant) ; d'autre part, on peut se demander dans quelle mesure le résultat final de ces activations produira un arbre sémantiquement correct.

(ii) Dans l'approche par **grammaire attribuée**, l'analyse de l'arbre s'effectue au travers des attributs chargés de dériver l'information nécessaire aux contrôles. Rappelons que les attributs synthétiques sont chargés de remonter de l'information vers le haut de l'arbre tandis que les attributs hérités descendent de l'information vers le bas. Le processus de construction sous-jacent consiste à analyser l'information nécessaire aux contrôles et à identifier les chemins par lesquels dériver cette dernière.

Cette approche est plus facile à appliquer que celle par routine sémantique et ce pour plusieurs raisons :

- D'abord , le concepteur des contrôles doit uniquement décrire les relations qui existent entre les attributs. La dérivation de l'information au travers de l'arbre est implicite au formalisme. Comme nous l'avons vu, lorsque l'arbre est modifié, des relations consistantes entre les attributs peuvent être rétablies automatiquement. Par conséquent on ne retrouve pas, dans cette approche, de notions explicites de mise à jour des structures de données auxiliaires (en fonction de la création et de la suppression de noeuds ou du déplacement du curseur par l'utilisateur). Le concepteur ne doit donc se préoccuper que de la description des états sémantiquement valides de l'arbre et non du "comment" dériver ces états. Il en résulte une plus grande facilité d'écriture des contrôles, bien qu'il ne soit pas nécessairement évident de construire un système complet et cohérent d'équations correctes lorsque le nombre d'attributs devient élevé.

- En outre, le concepteur peut aussi tout ignorer du modèle d'édition sous-jacent à cette approche, modèle très complexe dans le cadre de l'approche par routine sémantique.

(iii) Dans l'approche par **sémantique naturelle**, l'analyse de l'arbre s'effectue par établissement de séquents. Les règles d'inférence permettent de décrire la validité d'un séquent à partir d'autres séquents. Chaque variable qui y apparait désigne une classe d'objets et peut être instanciée (lors du processus de démonstration) à un sous-arbre de racine appartenant à cette classe. Ainsi, une même règle d'inférence permet de décrire la condition de validité d'un sous-arbre de manière générique en terme des sous-arbres fils. Par exemple, la règle R1, dans notre langage PICO, exprime la validité d'un arbre pico-pgm (DECLS, SERIES) en fonction de la validité des sous-arbres de type DECLS et SERIES (cfr 2.4.5.). Pour chaque opérateur de ces classes, on spécifie alors la condition de validité du séquent correspondant (règles R2, R3, R5 et R6).

L'utilisation de variables TYPOL identiques au numérateur et au dénominateur des règles rend le parcours de l'arbre implicite. Le résultat de l'analyse d'un sous-arbre peut être synthétisé par l'utilisation de variables. Ainsi, l'environnement e qui apparait dans la règle R1 de notre exemple est instancié au cours de l'analyse du sous-arbre des déclarations. Il est établi de façon linéaire par analyse de la séquence des déclarations (rappelons que ceci est réalisé par utilisation de mêmes variables comme hypothèses et résultats des prémices de la règle R3). L'environnement e résultant de cette analyse est alors transmis comme hypothèse au séquent qui analyse le corps du programme.

Dans cette méthode, le travail de conception peut donc être réalisé de manière très systématique; les contrôles sont définis de manière récursive sur la structure du formalisme.

Comme dans l'approche par grammaire attribuée, tout caractère dynamique a disparu dans la description des contrôles sémantiques : le concepteur ne doit spécifier que les relations à vérifier par les différents composants de l'arbre ; le moteur d'inférence de PROLOG utilisé dans cette approche automatise la résolution de ces relations.

Notons également que le concepteur des contrôles ne doit spécifier que les conditions de validité à vérifier pour tout arbre édité. Contrairement à l'approche par grammaire attribuée, aucune référence n'est faite aux conditions d'erreurs. La spécification des contrôles qui en résulte est dès lors allégée : par exemple, aucun type complémentaire (tel que le type "undefined" en 2.3.6.) n'a dû être introduit dans notre langage PICO. Remarquons cependant que nous aurions pu adopter la même stratégie de contrôle que pour une grammaire attribuée : l'idée aurait alors été d'annoter (lors du processus de démonstration) l'arbre édité d'informations sur le type des expressions et le statut des constructions de langage. Cette information aurait pu être utilisée pour générer des mesages d'erreurs. Cette solution présente l'avantage de ne pas interrompre le processus de validation quand une erreur est détectée.

Par rapport aux deux autres approches, la description des contrôles en sémantique naturelle présente un plus grand découplage par rapport à la description de la syntaxe abstraite : dans une grammaire attribuée, les équations sémantiques sont associées à chaque opérateur de la grammaire abstraite. La valeur de chaque attribut y est spécifiée en terme d'autres attributs de la production. De même, les routines sémantiques sont spécifiées pour chaque opérateur .Comme nous l'avons déjà souligné, leur description reprend une large part de la spécification de la syntaxe abstraite. Par contre, en sémantique naturelle, les règles ne font référence à la syntaxe abstraite du formalisme que de manière implicite au travers des arbres référencés. Cet aspect présente l'avantage d'augmenter la lisibilité des spécifications.

Malgré les avantages de l'approche par sémantique naturelle, elle ne s'adapte pas à des contrôles effectués en parallèle avec l'édition : en effet, une opération d'édition implique en général une modification locale de l'arbre. Or aucun moyen n'est fourni pour faire dépendre les spécifications d'une modification précise. De plus, nous pensons que la méthode d'analyse sous-jacente à cette approche ne s'adapte pas à une analyse incrémentale d'un arbre : celle-ci se réalise en effet de manière descendante, la validité d'un composant étant exprimée en terme de ses constituants. Dès lors, une analyse progressive du haut de l'arbre à partir de l'endroit de la modification ne se conçoit que difficilement.

En sémantique naturelle, les contrôles sont réalisés en fin d'édition de l'arbre. Comme nous l'avons remarqué en 2.4.7. cette restriction pourrait être facilement supprimée : il suffirait en effet, tout comme dans le cadre d'une grammaire attribuée, d'ajouter à chaque phylum un opérateur complémentaire. Chaque qroupe de régles définissant la condition de validité d'un séquent (pour chaque instance d'un opérateur d'un phylum précis) serait alors augmenté d'une règle. Celle-ci préciserait la condition de validité du séquent pour cet opérateur particulier.

Du point de vue de l'analyse de l'arbre ou d'un attribut, c'est le langage ARL qui offre les notations de plus bas niveau : en effet, le parcours est effectué de manière explicite par déplacement d'un curseur et par le test des types de noeuds rencontrés. De même, la création ou la suppression d'un noeud se fait par un ensemble de commandes similaires aux commandes d'édition d'un utilisateur.

En SSL, la construction et l'analyse d'un arbre se basent sur le nom des opérateurs : un nouvel arbre est construit par application du nom d'un opérateur à ses k constituants. La structure d'un arbre peut être analysée par des expressions multi-branches dont la sélection se base sur le nom des opérateurs (par exemple, par une expression "with", cfr 2.3.4.).

En ce domaine, le langage TYPOL offre les notations de plus haut niveau : on ne retrouve pas ici de notations différentes pour construire et analyser un arbre. Ces opérations sont effectuées par unification. Ainsi dans la règle R1 de notre exemple PICO, l'analyse de l'arbre est guidée par l'utilisation de variables (DECLS, SERIES) identiques au numérateur et au dénominateur. Dans la première règle de l'ensemble DECLARE (de ce même exemple), la déclaration d'une variable est ajoutée à l'environnement par écriture directe de l'arbre correspondant.

Notons aussi que le langage ARL n'offre aucun mécanisme permettant une spécification modulaire des contrôles . Par contre, le langage SSL permet la modularisation des spécifications à trois niveaux :

(i) il permet d'ajouter un nouvel attribut à un phylum défini ;

(ii) il permet d'ajouter une nouvelle équation
sémantique à un opérateur défini ;
(iii) il permet d'ajouter de nouveaux opérateurs et
leurs équations sémantiques à un phylum défini.

Dans le langage TYPOL, c'est la notion d'"ensemble" qui permet de modulariser les spécifications . (Rappelons qu'un ensemble est un groupe de règles formant un système formel complet auquel on peut faire référence d'un autre ensemble TYPOL).

L'avantage d'un langage de description modulaire des contrôles est qu'il permet d'isoler la spécification d'aspects différents d'un formalisme dans des entités différentes. Il en résulte une plus grande lisibilité, ainsi qu'une plus grande possibilité de réutilisation (dans le cadre, par exemple, de contrôles liés à différents dialectes d'un langage).

Nous avons mentionné les différents avantages de l'approche par grammaire attribuée et par sémantique naturelle par rapport à l'approche par routine sémantique. Ces avantages se paient au prix d'une certaine inefficacité : dans l'approche par grammaire attribuée, les attributs sont définis par des relations locales au sein des équations sémantiques ; dès lors, lorsque l'arbre abstrait est modifié, le calcul des nouvelles valeurs d'attributs peut s'effectuer en chaîne et couvrir une partie importante de l'arbre. A l'opposé, dans l'approche par routine sémantique, les structures de données auxiliaires permettent de regrouper de l'information et d'éviter ainsi de réanalyser de grosses parties de l'arbre abstrait. Rappelons cependant que dans ce cadre le concepteur des contrôles se trouve face à la tâche souvent difficile de maintenir l'ensemble de ces structures en cohérence avec l'arbre édité. Dans l'approche par sémantique naturelle, l'inefficacité est liée à l'algorithme d'unification et au mécanisme de "backtraking" utilisé par le moteur PROLOG.

Enfin, remarquons que les trois systèmes GANDALF, MENTOR et le "synthesizer generator" sont génériques : ils sont conçus autour d'un noyau éditeur qui est instancié à l'un ou l'autre langage par un méta-système.

Remarquons que cette généricité peut être conçue de deux manières différentes : d'une part, on peut considérer un noyau éditeur identique, quel que soit le formalisme utilisé. Dans cette hypothèse, le noyau trouve l'information nécessaire à son fonctionnement dans des tables générées à partir de la description du langage fournie au système. D'autre part, on peut considérer qu'un nouvel éditeur est généré pour chaque formalisme particulier par un programme générateur. Dans cette hypothèse, on a un éditeur par langage considéré.

D'après les sources [Reps et al., 87], [Donzeau-Gouge, 80], [Kaiser et al., 86] , il semblerait que la généricité dans MENTOR et dans le "Synthesizer Generator" soit réalisée d'après le premier principe, tandis que GANDALF corresponde au second.

PARTIE II : APPLICATION A L'ATELIER ALMA

INTRODUCTION

Cette deuxième partie est consacrée à une étude de l'atelier ALMA [van Lamsweerde et al., 87], atelier d'aide au développement et à la maintenance de produits logiciels multi-personnes, multi-versions.

Cet environnement offre un support intégré pour la manipulation d'une documentation consistante, complète et traçable d'un projet. Un éditeur syntaxique générique a été conçu pour permettre la manipulation de textes formels trouvés dans cette documentation. Dans l'état actuel, le méta-langage de description d'un nouveau formalisme n'offre aucune possibilité de méta-définir des contrôles sémantiques interprétables par l'éditeur. Bien qu'une telle extension soit fort intéressante, elle sort du champ de ce travail.

Cependant, un certains nombres de contrôles sémantiques doivent être réalisés pour deux langages de type Entité-Relation conçus dans ce cadre: le premier est un langage de définition d'un modèle de cycle de vie. Il permet de méta-définir le schéma d'une base de données projet. Le second est un langage de documentation de cette dernière. Dans l'état actuel, ces contrôles sont réalisés par des outils indépendants du noyau d'édition.

L'objectif de cette deuxième partie est d'étudier la possibilité de prise en charge d'un certain nombre de ces contrôles par un éditeur offrant des services génériques qui permettent la validation sémantique des arbres abstraits édités.

Cet exposé comporte deux parties: dans la première, après une description succinte du projet ALMA, nous positionnerons l'éditeur syntaxique conçu dans ce cadre par rapport aux trois éditeurs déjà étudiés. Nous examinerons ensuite plus en détails les différents contrôles mentionnés ci-dessus. Dans la deuxième partie, nous étudierons la possibilité d'exprimer ces contrôles dans les trois cadres formels étudiés dans la première partie. Nous discuterons ensuite de leur réponse au problème précédemment formulé.

CHAPITRE 1 : POSITIONNEMENT DU PROBLEME.

1.1. Présentation du projet ALMA.

La présentation succinte qui suit est reprise de [van Lamsweerde et al., 87].

Le noyau ALMA est un environnement conçu pour fournir un ensemble d'outils intégrés et cohérents permettant l'automatisation de différentes tâches effectuées lors du cycle de vie (CV) d'un projet logiciel multi-personnes, multi-versions.

Sa conception se base sur les 2 grandes tendances actuelles en matière de recherche sur les architectures d'environnements logiciels :
- d'une part, la nécessité d'articuler les outils autour d'une base de données qui centralise les informations sur le projet logiciel. Toutes les informations concernant le CV d'un projet logiciel particulier sont modélisées en terme des objets et des relations entre ceux-ci; ces informations sont conservées de manière consistante et traçable dans une base de données d'atelier.
- d'autre part, il est également de plus en plus reconnu que certains produits logiciels spécifiques sont des objets structurés dans différents formalismes. Ces objets sont donc représentés de manière interne sous forme d'arbre syntaxique abstrait et les outils qui les manipulent sont dirigés par leur syntaxe.

Ces deux tendances ont été intégrées dans le cadre du projet ALMA qui fournit principalement les deux fonctions suivantes :
- support intégré pour la manipulation d'une documentation consistante, complète et traçable d'un projet; documentation mémorisée dans une Base de Données Projet (PDB).
- support dirigé par la syntaxe pour la manipulation de textes trouvés dans cette documentation et écrit dans différents formalismes.

Les aspects caractéristiques d'ALMA sont les suivants :

- stratégie d'intégration : une PDB est le moyen unique d'intégrer des outils autonomes (qui autrement ne connaîtraient rien les uns des autres). Elle a une structure Entité-Relation avec les composants suivants :

(i) des objets de différents types prédéfinis. Il serait par exemple possible de retenir dans une phase d'analyse fonctionnelle des objets de type FONCTION ou DONNEE; une architecture logicielle pourrait être composée d'objets de type MODULE. Un modèle CV plus complet pourrait comprendre des objets de type PROGRAMME SOURCE , FICHIER, PROGRAMMEUR,...

(ii) des relations logicielles n-aire entre types d'objets. Par exemple, une relation binaire DERIVE DE pourrait être définie sur les types d'objets FONCTION et MODULE. Des modèles de CV plus riches pourraient comprendre d'autres relations telles qu' une relation IMPLEMENTE PAR entre les types d'objets MODULE et PROGRAMME SOURCE, une relation réflexive VERSION DE définie sur des types tels que FONCTION, MODULE ou PROGRAMME SOURCE, une relation ternaire AFFECTE A définie entre les objets MODULE, PROGRAMMEUR et CONTRAT...

(iii) des propriétés logicielles caractérisant des objets et/ou relations logicielles. Par exemple, des objets de type MODULE ou PROGRAMME SOURCE pourraient avoir des propriétés telles que NOM, DATE de création/mise à jour, STATUT; une propriété délai pourrait être attachée à la relation ternaire AFFECTE A définie ci-dessus...

Une propriété importante d'ALMA est la possibilité de décrire des propriétés ayant des textes formels comme valeur. Ceci permet par exemple d'attacher des propriétés telles que des spécifications formelles, du pseudo-code, des explications formelles à un objet de type MODULE... N'importe quel texte formel peut ainsi être attaché à n'importe quelle instance d'objet ou de relation logicielle et peut être manipulé avec une connaissance de sa syntaxe.

- couverture de tout le cycle de vie : comme le suggèrent les exemples ci-dessus, les différents objets logiciels, relations et textes formels manipulés par le noyau peuvent faire référence à toutes les étapes d'un CV.

- paramétrisation du modèle de CV et des langages : le noyau ALMA s'adapte automatiquement aux types d'objets, relations, propriétés et formalismes spécifiques à un modèle de CV particulier. Ceci est réalisé par une architecture à deux niveaux :

(i) au niveau instancié, un environnement est associé à un modèle de CV particulier, méta-défini au niveau instanciant. Un environnement instancié repose sur un schéma spécifique de la PDB

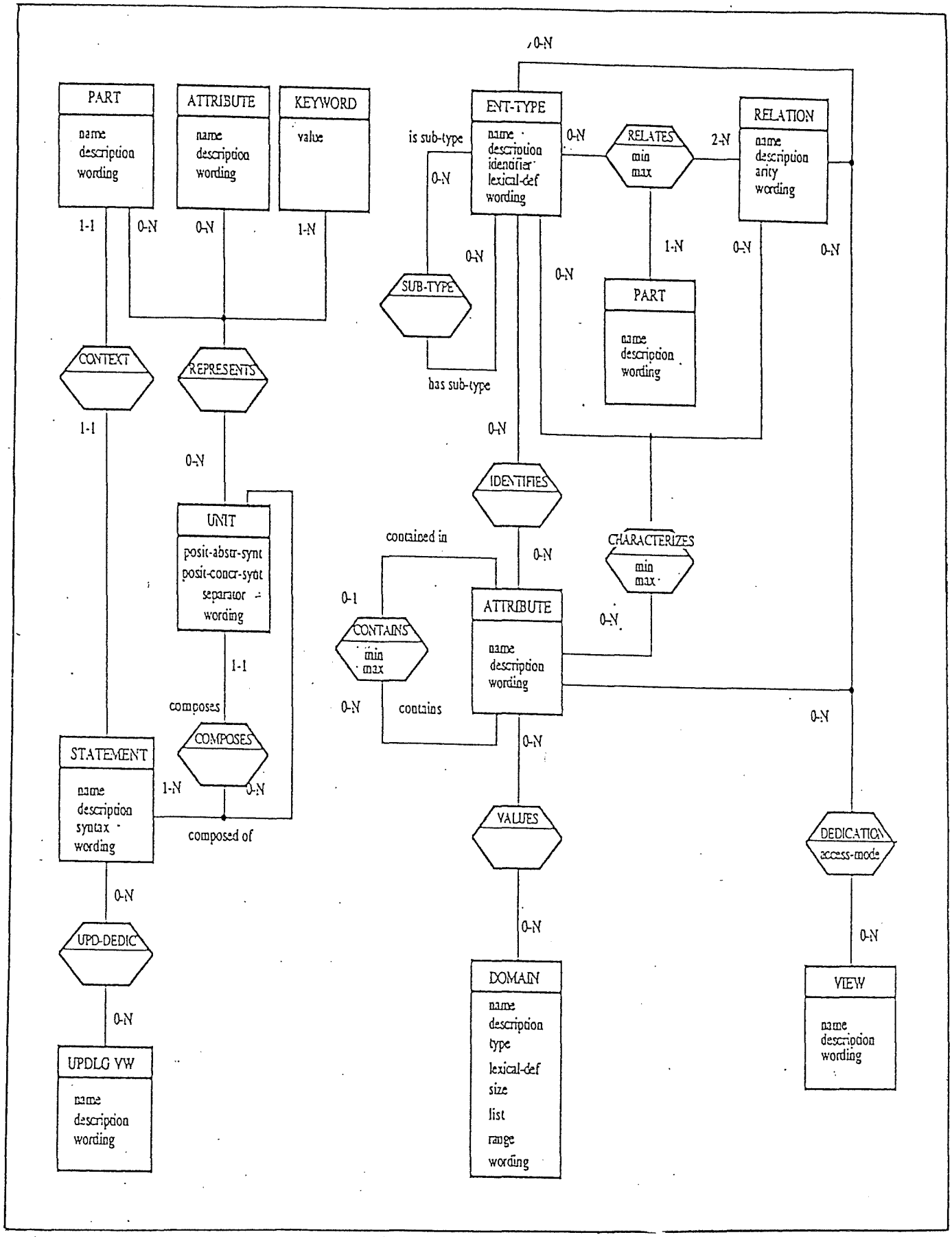


Fig. 24.

- Le méta-modèle ALMA -

(extrait de [van Lamsweerde et al., 87]) -

correspondant au modèle de CV particulier; il est composé d'outils génériques tels qu' un outil de mise à jour de la PDB, un outil d'interrogation de la PDB, des éditeurs structurés, analyseurs et décompilateurs. Ces outils génériques sont automatiquement instanciés au modèle de CV approprié : ils sont dirigés par une base de données du modèle (MDB) qui contient la méta-description des différents types d'objets, relations, propriétés et formalismes trouvés dans ce modèle.

(ii) au niveau instanciant, l'environnemnt méta génère automatiquement toutes les informations nécessaires à la production d'environnements instanciés. Il accepte 2 sortes de méta-définitions : (i) une méta-définition en termes d'un méta-modèle Entité-Association étendu, des différents types d'objets, relations et propriétés trouvées dans un modèle de CV particulier,et (ii) une méta-définition des formalismes trouvés dans ce modèle.

A partir de ces définitions, le méta-environnement génère le contenu de la MDB qui conduit les outils génériques au niveau instancié. D'autres méta-informations sont également générées comme par exemple les méta-définitions des langages utilisés pour mettre à jour et interroger la PDB; ces outils sont nécessaires pour générer à leur tour des éditeurs structurés instanciés à ces langages.

Une description plus détaillée des caractéristiques d'ALMA peut être trouvée dans [van Lamsweerde et al.,86] et [van Lamsweerde et al.,87].

Une représentation Entité-Relation-Attribut (ERA) du méta-modèle est présenté Fig. 24. Comme on peut le voir, il est composé de deux parties connectées mais disjointes : une partie sémantique (partie droite de la Fig. 24.) et une partie syntaxique (partie gauche de la Fig. 24.). La partie sémantique contient les différents concepts et relations en termes desquels le modèle de CV doit être défini; nous pouvons remarquer la possibilité de définir des relations n-aires, de caractériser des relations par des propriétés, de définir des relations sur l'union de types d'objets. Une relation de spécialisation est également disponible avec la possibilité d'héritage multiple de propriétés et relations; des vues peuvent être définies. La partie syntaxique contient quant à elle les différents concepts et relations en termes desquels la syntaxe des langages de documentation et d'interrogation de ce modèle sont définis. Le lecteur intéressé peut trouver une discussion plus approfondie de ce méta-modèle dans [van Lansweerde et al.,87].

Un diagramme de flux du méta-système ALMA peut également être trouvé Fig. 25. Comme on peut le voir, le méta-système reçoit en entrée une méta-définition d'une extension d'un modèle de CV. La création d'un nouveau modèle est donc traitée comme un cas particulier où le modèle original est vide. La méta-description de l'extension du modèle est convertie en l'arbre syntaxique abstrait correspondant grâce à l'analyseur instancié au méta-langage de description d'un modèle. Un certain nombre de contrôles sémantiques sont ensuite effectués (nous reviendrons plus en détails sur ce point au paragraphe 1.3.1. de ce chapitre). La MDB est ensuite mise à jour à partir d'un arbre syntaxique sémantiquement consistant. Les méta-informations dirigeant les outils génériques sont ensuite mises à jour ainsi que le schéma logique de la PDB.

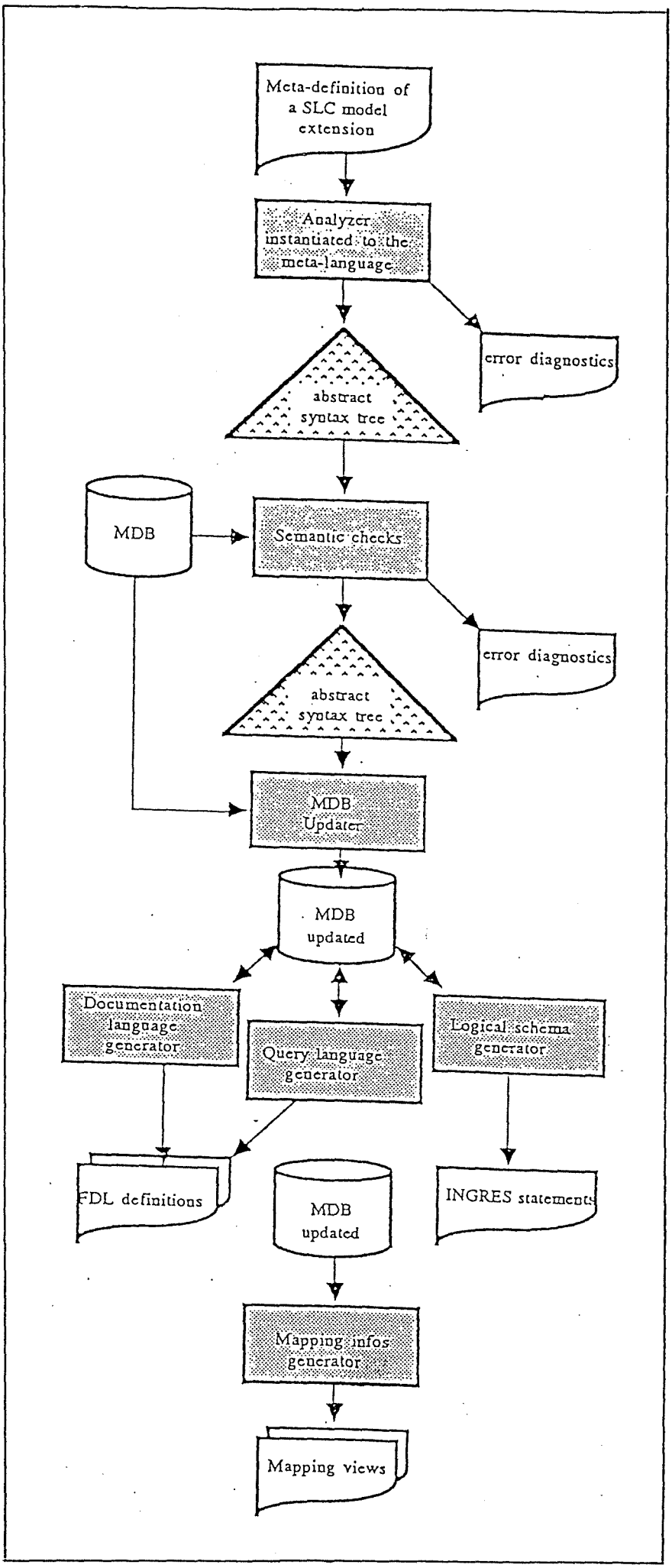


Fig. 25.

- Diagramme de flux du méta-système -

(extrait de [van Lamsweerde et al., 87])

1.2. Positionnement de l'éditeur syntaxique ALMA.

L'éditeur syntaxique ALMA permet une édition en mode à entrée stucturée ou à entrée libre. Il est générique et puise donc les informations nécessaires à son fonctionnement dans des tables. Ces informations sont générées à partir de la description d'un nouveau formalisme ; cette description est rédigée dans un méta-langage appelé FDL (Formalisme Description Language).

La conception de ce méta-langage s'inspire des idées et concepts du système MENTOR. Cependant, il en diffère par un certain nombre de caractéritiques.

Une méta-définition FDL consiste en un ensemble de règles de production. Les règles synthétisent de façon non-redondante les différentes informations relatives à:

- la syntaxe abstraite;
- la syntaxe concrète;
- la syntaxe lexicale;
- le schéma de décompilation;

ainsi que des informations sur :

- la possibilité d'insérer des textes dans d'autres formalismes connus à certains endroits (annotations) ;
- la possibilité de passer d'une édition en mode à entrée structurée à un mode à entrée libre à un certain niveau de raffinement;

Les aspects de syntaxe abstraite, concrète et lexicale ont été regroupés pour éviter des définitions longues et redondantes ainsi que pour assurer la consistance entre ces différents points.

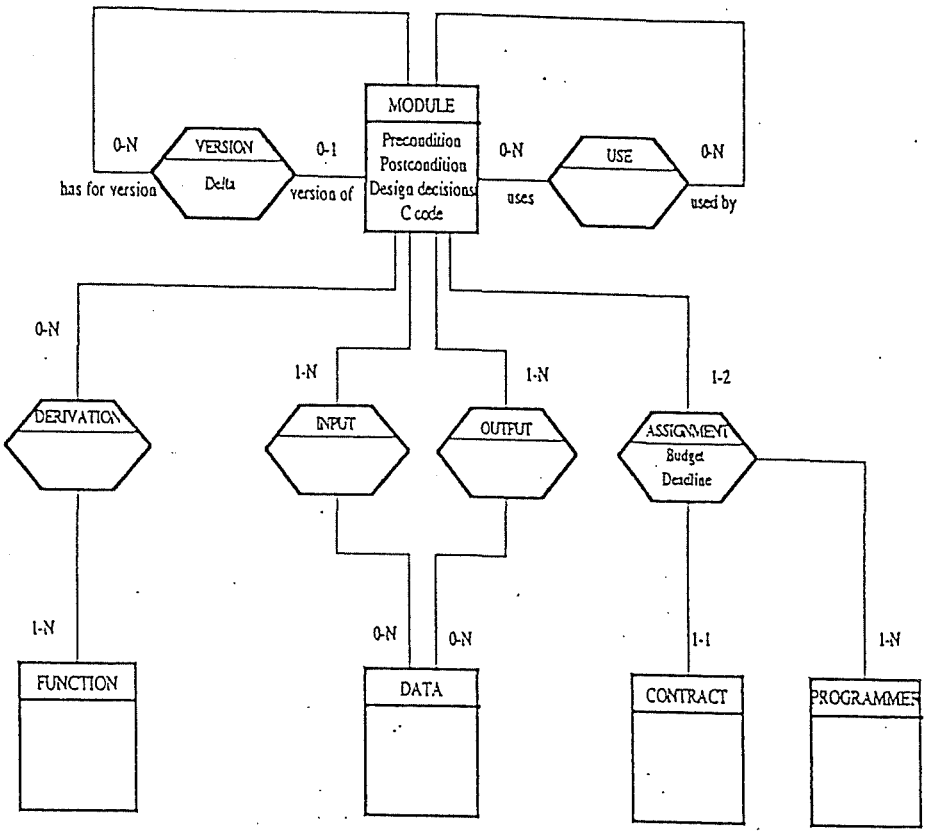
Dans une définition FDL , tout opérateur est

- soit un opérateur de type liste,par exemple,

L <statms> ::= <statm>* ";"

- soit un opérateur d'arité comprise entre 0 et 4.

La partie droite d'une règle définissant un opérateur unaire contient soit un opérateur non terminal, soit une liste d'alternatives de non terminaux. Ces opérateurs unaires ont un statut spécial en ce sens que le concepteur de la définition FDL peut décider qu'aucun noeud dans l'arbre syntaxique abstrait ne sera associé à l'opérateur unaire défini dans la règle (on parle alors d'opérateur sans constructeur).




```
META-DEF MODEL example ;
  ENTITY-TYPE module ;
    IDENTIFIER = module-name ;
    IDENTIFIER'S LEXICAL DEFINITION = "[a-zA-Z][a-zA-Z0-9-]*" ;
  RELATION assignment ;
    THE PART mod-assign IS PLAYED BY 1-N module ;
    THE PART pgr-assign IS PLAYED BY 0-N programmer ;
  PART mod-assign ;
  STATEMENT mod-assigned-to ;
    FOR mod-assign ;
    SYNTAX 'assigned to' pgr-assign 'on budget' budget 'with deadline' deadline ;
  PART pgr-assign ;
  STATEMENT pgr-assigned-to ;
    FOR pgr-assign ;
    SYNTAX 'is assigned' (mod--assign 'on budget' budget 'with deadline' deadline : ',') ;
  ATTRIBUTE deadline ;
    CHARACTERIZES assignment WITH CONNECTIVITY 1-1 ;
    ACCEPTS DOMAIN(S) dom-date ;
  ATTRIBUTE c-code ;
    CHARACTERIZES module WITH CONNECTIVITY 0-1;
    ACCEPTS DOMAIN(S) formal-text ;
  DOMAIN dom-date ;
    TYPE = char ;   SIZE = 8 ;
    LEXICAL DEFINITION = "[0-9][0-9] | [0-9][0-9] | [0-9][0-9]" ;
```

Fig. 26

- Meta-description d'un modèle de CV -

(extrait de [van Lamsweerde et al., 87])

L'opérateur "stamt" défini ci-dessus illustre cette possibilité :

SC <stamt> ::= <while> | <ifthenelse> | <assign>

Cette notion d'opérateur sans constructeur joue le rôle du concept de classe des trois systèmes présentés dans la première partie (i.e. permettre le choix entre plusieurs opérateurs en une position de l'arbre abstrait et éviter d'introduire des niveaux intermédiaires dans l'arbre). Elle présente l'avantage de condenser les définitions de la syntaxe abstraite.

1.3. Etude des contrôles sémantiques.

1.3.1. Le méta-langage de spécification d'un modèle de CV.

Le noyau ALMA contient au niveau instanciant 2 méta-langages de spécification prédéfinis : le langage FDL de description d'un nouveau formalisme et le langage de spécification d'un nouveau modèle de CV. C'est sur ce dernier que nous allons nous concentrer dans ce paragraphe.

Ce langage est non procédural: c'est un langage ERA dont la structure reflète celle du méta-modèle présenté en Fig. 24. A titre d'illustration, nous avons repris en Fig. 26. un fragment d'un modèle de CV et en vis-à-vis un extrait de la méta-description de ce modèle.

La description FDL de ce langage est reprise en annexe 2.

1.3.1.1. Contraintes à vérifier

Un certain nombre de contraintes n'ont pu être inclues dans cette description. Celles-ci sont de deux ordres.

D'une part, nous avons les contraintes d'intégrité liées au méta-modèle. Elles concernent les différentes méta-connectivités apparaissant en Fig. 24.. Nous avons aussi des contraintes plus sophistiquées telles que " il ne peut y avoir de circuit dans les déclarations de SOUS-TYPE ", "une occurence du méta-type d'entité PART est associée à une et une seule occurence du méta-type d'entité RELATION ", " une occurence du méta-type d'entité ATTRIBUT spécifié comme composant ne peut caractériser une occurence du méta-type d'entité RELATION" ... Une liste complète de ces contraintes peut être trouvée en annexe 4.

D'autre part, le deuxième type de contraintes concerne la légalité d'une extension. Comme nous l'avons déjà mentionné, la création d'un nouveau modèle de CV est traitée par le système comme un cas particulier où le modèle original est vide. Ceci permet d'étendre facilement un modèle déjà existant. Il faut dès lors vérifier que l'extension proposée n'introduise pas d'inconsistance par rapport à la PDB en cours d'exploitation. Par exemple, de nouvelles occurences des méta-types d'entité "ENTITY", "RELATION", "ATTRIBUTE" et "VIEW" peuvent être créées; une nouvelle occurence de la méta-relation "CHARACTERIZES" peut être introduite entre une nouvelle occurence d'ATTRIBUT et une ancienne occurence de RELATION, mais pas entre d'anciennes occurences d'ATTRIBUT et de RELATION.

Une liste complète de ces contraintes peut être trouvée en annexe 5.Nous avons également repris en annexe 3 le profil des différents arbres syntaxiques abstraits pouvant être générés lors de la description d'un nouveau modèle de CV.

De l'analyse de ce profil et des différentes contraintes évoquées précédemment, nous avons pu classifier les contrôles à effectuer en 3 groupes:

* contraintes liées à un opérateur syntaxique précis

- contraintes de format : ces contraintes concernent certains opérateurs terminaux dont la valeur est limitée à un certain nombre de caractères. Par exemple, la valeur des opérateurs "name" et "wording" des différents méta-types d'entité apparaissant dans le méta-modèle doivent être des chaînes de caractères limitées respectivement à 16 et 64 caractères. Le choix qui a été effectué est de vérifier ces contraintes à postériori à l'aide d'un analyseur lexical généré par l'outil LEX de Unix.

- contraintes de non modifiabilité : ces contraintes concernent certains opérateurs terminaux dont la valeur ne peut être modifiée lors d'une extension du modèle de CV considéré. Par exemple, la valeur de l'opérateur "wording" des différents méta-types d'entité ainsi que la valeur de l'opérateur "lexical-def" associé au méta-type d'entité "ENT-TYPE" ne peut être modifiée.

* contraintes de non multiplicité : ces contraintes portent sur la non occurrence mutiple d'un opérateur comme fils d'un autre. Ces contraintes sont liées à l'aspect facultatif de certains opérateurs : ainsi, par exemple, un méta-type d'entité du méta-modèle peut avoir au plus un attribut "wording" ou "description". Ceci est formalisé dans la description du langage de la façon suivante :

<entity-section> ::= <entity-text> *
<entity-text> ::= <description | wording | name-id>

Cette description n'empêche cependant pas l'occurrence multiple des opérateurs "wording" et "description", contrainte qu'il faut donc surimposer.

```
DOC MODULE FTsubstit ;
    LAST DATE OF UPDATE = 12/04/86
    STATUS = ok
    ASSIGNED TO (programmer) Buyse UNDER (contract) kbar/soft/03
            WITH BUDGET 9040
            WITH DEADLINE 01/06/86 ;
    DERIVED FROM (function) tree-substitution ;
    ACCEPTS AS INPUTS (data) given-tree, (data) subtree1 , (data) subtree2
    OUTPUTS (data) newtree
    PRECONDITION = ...(formal specification)... ;
    POSTCONDITION = ...(formal specification)...;
    DESIGN-DECISIONS = ...(informal or formal explanation)...;
    USES (module) FTcopy , (module) FCnson , (module) FTabstr-synt-verif ;
    C-CODE = ...(formal C text)... ;
    IS A VERSION OF (module) ... WITH DELTA ...(formal text)... ;
```

Fig. 27

- Une section d'entite pour le modèle de la Fig. 26 -

(extrait de [van Lamsweerde et al., 87])

* contraintes liées au contexte : nous avons évidemment un ensemble de contraintes liées aux aspects dépendants du contexte. Nous retrouvons dans ce cadre les 3 premiers types de contraintes évoqués dans le premier chapitre de la première partie , à savoir l'unicité des identificateurs définis (pour les occurences de méta-types d'entité), l'utilisation d'identificateurs définis (pour les occurences de méta-relations) ainsi que la vérification de différentes propriétés lors de l'utilisation des identificateurs (contraintes de type, contraintes d'intégrité du modèle et d'extensibilité).

Nous étudierons dans le chapitre 2 la possibilité d'exprimer ces contrôles dans les trois cadres formels étudiés dans la première partie.

1.3.2. Le langage de documentation d'un CV.

Au cours du CV d'un projet logiciel, l'utilisateur a la possibilité de mettre à jour le contenu de la PDB grâce à un outil de mise à jour : il fournit à ce dernier un ensemble de phrases écrites dans un langage de documentation de très haut niveau; il y décrit les nouveaux objets logiciels à créer, à supprimer, ainsi que les valeurs de propriétés ou occurences de relations à ajouter, supprimer et/ou modifier.

Ce langage de documentation est un langage formel dédicacé à un modèle de CV particulier (au niveau instancié). Il a une structure Entité-Association-Relation (EAR); les constructions et mots-clés y apparaissant font référence aux types d'objets, relations et propriétés trouvées dans ce modèle (ou, plus précisément, dans la vue de l'utilisateur).

Une documentation est organisée en sections d'entité. Une section d'entité fait référence à un objet logiciel spécifique dont le type doit appartenir à la vue considérée. Elle contient une déclaration implicite du nom de l'objet, une phrase de documentation par propriété ainsi que par occurence de relation associée à l'objet dans cette vue.

La Fig. 27. montre un exemple d'une section d'entité pour le modèle de CV suggéré Fig. 26. Les mots-clés y sont décrits en caractères gras. Les phrases sur les propriétés (par exemple STATUS) d'un objet sont de la forme

<nom-propriété> '=' <valeur>*

tandis que les phrases sur les relations (par exemple ASSIGNMENT) et leurs propriétés (par exemple DEADLINE) ont une syntaxe dépendant du modèle (cette syntaxe ayant été définie dans la méta-définition donnée au méta-système).

Le langage de documentation de CV est un langage non procédural; l'ordre entre les sections entités et entre les phrases n'a pas d'importance. Il s'agit également d'un langage typé. Chaque objet apparaissant dans une phrase décrivant une relation doit appartenir au bon type (ce type peut apparaître de manière optionnelle entre parenthèses).

Ce langage étant dédicacé à un modèle CV particulier, la définition de sa syntaxe dans le langage FDL est générée automatiquement par le méta-système. Cette définition permet de conduire l'éditeur syntaxique ALMA lors de la description d'une documentation par l'utilisateur.

1.3.2.1. Mise à jour de la PDB

De manière précise, la mise à jour de la PDB s'effectue selon les phases suivantes :

1. L'utilisateur fournit l'identificateur et le type de l'objet logiciel à mettre à jour.
2. Un squelette de section d'entité correspondante est imprimé à l'écran. Ce squelette contient toutes les informations sur les valeurs de propriétés et occurences de relations déjà mémorisées dans la PDB.

Cette phase est effectuée en trois étapes :

fragment de PDB ----------> formulaire ERA
formulaire ERA ----------> arbre syntaxique abstrait
arbre syntaxique abstrait -> représentation visuelle externe

Un formulaire ERA est une structure de données intermédiaire qui rassemble toutes les informations concernant le contexte local de l'objet mis à jour. Ces informations sont nécessaires à la création de l'arbre syntaxique abstrait, à la mise à jour de la PDB ainsi qu'à la validation de cette mise à jour.

3. L'utilisateur complète et/ou modifie les informations concernant l'objet logiciel désigné. Un menu dirigé par la syntaxe abstraite du langage de documentation lui présente le ou les propriétés et relations qui peuvent être mises à jour.

On se trouve ici devant une situation où toute la puissance d'une édition à entrée structurée est particulièrement mise en évidence : comme le langage de documentation méta-défini par l'utilisateur se veut très proche du langage naturel, sa syntaxe concrète contient des suites de mots-clés pouvant être relativement longues. Grâce à l'éditeur instancié, l'utilisateur peut introduire la documentation sans taper un seul mot-clé ou information de type. De plus, la possibilité du système de manipuler plusieurs syntaxes concrètes pour une même syntaxe abstraite permet d'adapter le langage de documentation à la langue de l'utilisateur (par utilisation de mots-clés français, anglais,...).

4. L'arbre ainsi obtenu subit les transformations inverses de l'étape 2. Avant la mise à jour de la PDB, les différentes contraintes sémantiques méta-définies sont vérifiées. Finalement, la PDB est mise à jour sur base des différences entre les deux arbres syntaxiques abstraits correspondant au squelette initial et celui édité.

1.3.2.2. Contraintes à vérifier

Etudions plus en détails les contraintes sémantiques à vérifier lors de la mise à jour de la PDB. Ces contraintes sont de trois types :

. Elles concernent les valeurs de propriétés définies par l'utilisateur pour l'objet logiciel édité. Ces valeurs doivent appartenir au domaine défini dans le modèle de CV considéré (méta-défini en terme de la méta-relation 'VALUES' de la Fig. 24.).
En théorie ces contraintes pourraient être vérifiées en parallèle avec l'édition: il suffirait en effet que la grammaire FDL décrivant le langage de documentation reprenne ces différentes restrictions. Cette solution n'a cependant pas été retenue: elle présente l'inconvénient de générer des grammaires beaucoup trop importantes.

. Le second type de contrainte concerne différentes connectivités méta-définies dans le modèle de CV considéré, à savoir :
- le caractère obligatoire de certaines propriétés (méta-défini par l'attribut 'min' de la méta-relation 'characterizes' de la Fig. 24.).
- le nombre d'occurences d'une propriété (il ne peut y avoir qu'une phrase par section d'entité qui définit la valeur d'une propriété).
- le nombre d'occurrences d'une relation auxquelles une entité peut participer (méta-défini par l'attribut 'min' et 'max' de la méta-relation 'relates' de la Fig. 24.).

. Le troisième type de contrainte concerne les propriétés de type. Rappelons en effet que le langage de documentation est un langage typé.

Comme nous l'avons vu ci-dessus, ces différentes contraintes sont vérifiées a postériori, en fin d'édition d'une documentation. Nous étudierons au paragraphe 2.2 la possibilité d'exprimer ces contrôles dans les trois cadres formels étudiés précédement.

CHAPITRE 2 : INTRODUCTION DE CONTROLES SEMANTIQUES DANS L'EDITEUR D'ALMA.

2.1. Contrôles relatifs à l'extension d'un modèle de CV.

2.1.1. Remarques préliminaires.

Rappelons que la définition d'un nouveau modèle de CV est traîtée par le méta-système ALMA comme un cas particulier d'une extension d'un modèle préexistant : l'utilisateur définit dans le méta-langage de description l'extension désirée; en fin d'édition, l'arbre interne ainsi produit est analysé et les différentes composantes du système sont mises à jour quand il n'y a plus d'erreurs sémantiques (cfr Fig. 24).

Le but de ce paragraphe est d'étudier la possibilité d'effectuer les contrôles sémantiques en parallèle avec l'édition d'une telle définition de modèle. Nous étudierons en particulier la capacité d'exprimer ces contrôles dans les trois formalismes présentés précédemment. Nous discuterons ensuite du caractère plus ou moins approprié de tel ou tel cadre formel.

Avant d'entrer plus en profondeur dans cette étude, quelques remarques s'imposent : notons tout d'abord, que si nous voulons vérifier la validité d'une extension d'un modèle de CV au cours de sa définition, nous devons travailler sur base de la définition de l'ancien modèle de CV (mémorisé dans la MDB) pour accéder à l'information nécessaire aux contrôles.

Deux stratégies peuvent alors être envisagées :

(i) la première consiste à travailler directement avec l'ancien arbre syntaxique abstrait mémorisé. Une extension est dès lors vue comme une modification de l'arbre préalablement édité. Une telle démarche nécessite cependant certaines restrictions : dans ce cadre, l'utilisateur a directement accès à sa première définition; il peut dès lors non seulement l'étendre mais également la modifier (en supprimant par exemple telle méta-entité définie, telle méta-relation...). Cette opportunité est dangereuse et actuellement interdite par le système. Elle pourrait en effet laisser la PDB dans un état incohérent.

(i) Une deuxième stratégie consiste à travailler avec deux arbres différents : l'ancien arbre édité et l'extension définie. Nous pourions nous ramener au cas particulier d'un seul arbre à analyser en généralisant la définition d'un modèle à un arbre composé de deux parties : l'une d'entre elles serait la définition du modèle actuel (auquel l'utilisateur n'aurait pas accès), la deuxième correspondrait à la définition de l'extension. Seule cette partie pourrait être présentée à l'utilisateur.

Après analyse, il nous est appparu que la première stratégie devait être retenue. Les solutions envisagées se différencient par les trois aspects suivants :

(i) Comme nous l'avons déjà souligné, laisser la possibilité à l'utilisateur de modifier directement l'ancienne définition du modèle CV est dangereuse. Un contrôle des opérations d'édition doit donc être envisagé. Il porterait sur la possibilité de supprimer ou de modifier un noeud de l'arbre édité. Une solution à ce problème serait d'inclure une protection dans le noyau éditeur: chaque noeud pourrait être marqué comme modifiable ou non modifiable. Seul un noeud marqué modifiable de l'arbre édité pourra être modifié par l'utilisateur. Cette solution nous parait à bannir car elle entrainerait des extensions "ad hoc" dans le noyau d'édition.

(ii) Les contrôles liés à l'extension d'un modèle de CV supposent la capacité de différencier la définition d'un nouvel identificateur d'un ancien. Cette information disparaît dans la première solution suggérée ci-dessus. Elle pourrait être réintroduite de la façon suivante : à l'opérateur père d'une section, un troisième opérateur fils pourrait être ajouté par une production de la forme

<« - section> ::= <id> < -text> <nouveau>

où « ∈ {entity, relation, ...}

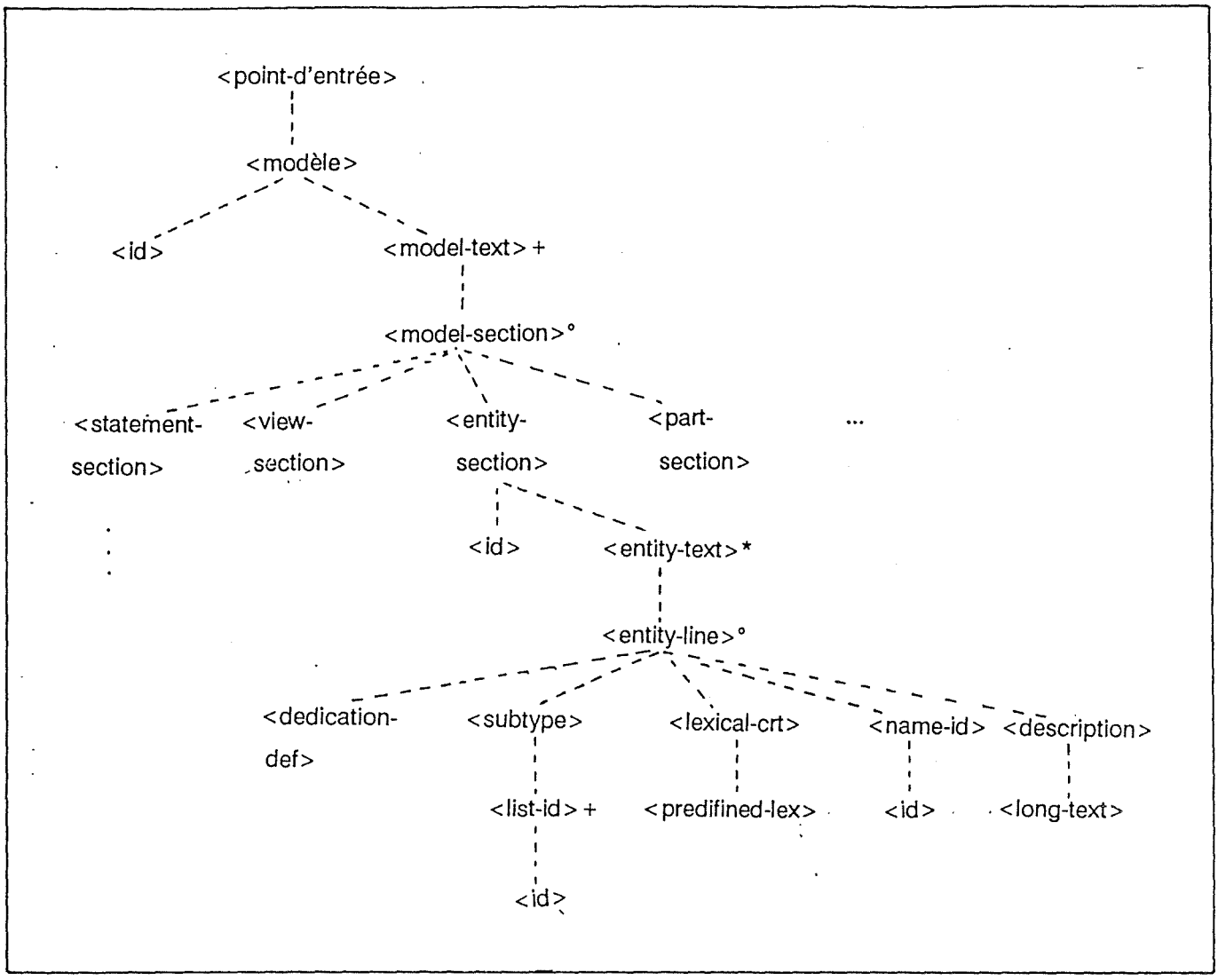
Il serait modifié par programme après la mise à jour de la MDB et remplacé par l'opérateur <ancien> (Pour éviter l'impression de ces opérateur à l'écran, nous proposons qu'ils correspondent à des opérateurs terminaux de valeur vide).

(iii) Un inconvénient de travailler avec deux arbres distincts (l'ancienne définition du modèle de CV et son extension) est que l'information relative à la définition d'une section d'entité est répartie à plusieurs endroits. Cependant, d'après notre classification des contrôles, il nous semble que seuls les contrôles relatifs "à un opérateur précis" et ceux "de non-multiplicité" s'en trouvent affectés . (En effet, les contrôles liés à un contexte global font appel à de l'information répartie dans plusieurs sous-arbres). Notons d'autre part que ces contrôles ne constituent qu'une faible part de l'ensemble des contrôles à réaliser .

Au vu de ces remarques, il nous parait que la solution à retenir est celle qui consiste à travailler en parallèle sur les deux arbres.

Dans la liste des contrôles présentés au paragraphe 1.3.1., nous avons mentionné : contrôle de format. Celui-ci pourrait disparaitre si nous l'introduisons dans l'éditeur. Ce contrôle est en effet différé en fin d'édition pour diminuer l'ampleur de la grammaire abstraite correspondante.

Dans la suite de ce chapitre, nous allons successivement étudier la possibilité d'exprimer les autres types de contrôles dans les trois approches routines sémantiques, grammaire attribuée et sémantique naturelle.



(< >* = opérateur de type liste avec au moins un élément

< > + = opérateur de type liste

< >° = opérateur sans constructeur).

Fig.28.

- Arbre de définition d'une section entité -

Pour concrétiser cet exposé, nous avons choisi un exemple représentatif pour chaque type de contrôle dégagé. A cet effet, nous nous concentrerons sur la partie de la grammaire permettant de définir une nouvelle entité. La Fig. 28. reprend la forme de l'arbre syntaxique abstrait correspondant. Elle montre que tout modèle consiste en un identificateur et une liste de définitions de sections (permettant de définir une occurrence d'un méta-type d'entité relation, entity-type, vue, ...). Une section d'entité est caractérisée quant à elle par un identificateur (le méta-attribut name de la Fig. 24.) ainsi que par une liste de méta-attributs (wording, decription, name-id, lexical-def) et de méta-relations (subtype, dedication).

Pour chaque type de contrôle, nous avons retenu l'exemple suivant :

- contraintes de non multiplicité : au plus un opérateur <wording> comme fils de l'opérateur <entity-text>.
- contraintes liées au contexte :

-unicité des identificateurs définis : unicité des opérateurs <id> fils des opérateurs <modele>, <entity- section>, <view-section),...

-utilisation d'identificateurs définis : contrôle de la définition des identificateurs <id> fils de <list-id> dans une autre section entité.

-utilisation d'identificateurs en accord avec leurs propriétés: non existence de circuit dans la relation de sous-typage définie (contrainte faisant appel à de l'information dispersée dans tout l'arbre du modèle); non définition d'une relation 'is sub-type' nom d'entité, cette dernière est une ancienne entité (contrôle lié à l'extensibilité du modèle).

2.1.2. Approche par routines sémantiques.

2.1.2.1. Remarques générales.

Rappelons que le principe de cette approche est d'associer aux opérateurs de la grammaire abstraite des routines sémantiques. Ces routines sont activées lorsqu'un noeud de ce type est concerné par une opération d'édition. Suivant le contexte de leur activation, elles effectuent certains contrôles et mettent à jour si nécessaire des attributs de l'arbre (synthétisant de l'information sur l'état sémantique de ce dernier). Si une erreur est détectée, elles peuvent soit interdire l'opération en cours, soit générer un message d'erreurs.

L'application de cette approche dans le cadre défini ci-avant ne devrait, en théorie, pas poser de problème : en effet, toute l'information nécessaire aux contrôles se trouve mémorisée dans l'arbre; les routines peuvent y accéder lors de leur activation.

L'idée de base serait de concevoir un certain nombre de structures de données auxiliaires qui synthétiseraient l'information nécessaire à la vérification de contraintes complexes. La vérification des contraintes locales se ferait par simple parcours de l'arbre.

Chaque structure de données serait dédicacée à un certain type de contrôle (par exemple, contrôle sur la définition des identificateurs, contrôle sur les relations de sous-typage, contrôle sur les relations de dédication...). Ces structures pourraient être mémorisées en dehors de l'arbre étant donné l'absence de la notion de bloc dans notre langage (toutes les structures devraient être en effet rattachées à l'opérateur <model-text>). Pour chacune de ces structures, un certain nombre de primitives de base pourraient être définies (tel que ajouter un nouvel identificateur, ajouter une nouvelle relation de sous-typage entre 2 entités, vérifier l'absence de circuit...) qui pourraient être activées par les routines. Ceci permettrait de simplifier leur écriture.

Remarquons également que ces contrôles sont loin d'être triviaux : la plupart d'entre eux font en effet appel à de l'information répartie dans tout l'arbre décrivant le modèle. Ceci se concrétisera par la consultation d'une ou plusieurs structures de données lors de la validation des opérations d'édition.

Cette analyse pourrait être effectuée à des moments différents : soit après chaque mise à jour d'une structure ; soit à des moments précis (à la fin de l'édition d'une section, à la requête de l'utilisateur ...).

Soulignons l'aspect complexe du maintien des structures données en cohérence avec l'état d'édition de l'arbre. Une étude approfondie des contrôles devrait être réalisée. Elle porterait sur les différentes structures requises ainsi que sur la conception d'une stratégie de réanalyse intelligente de celles-ci. Nous ne pousserons pas plus loin cette étude dans le cadre de ce mémoire.

Dans les exemples repris ci-dessous, nous avons brièvement repris les principes de construction pour chaque type de contrôle.

2.1.2.2. **Principes et exemples.**

(i) contrôles de non multiplicité.

Le principe est d'associer à l'opérateur concerné (par exemple "Wording") une routine qui vérifie , lors de la création de cet opérateur, s'il n'existe pas déjà d'opérateur frère de ce type. Dans l'affirmative, l'opération de création est interdite.

(ii) contrôles liés au contexte.

a. unicité des identificateurs.

Principe : définir une structure de données 'list-id' jouant le rôle d'une table de symboles qui contiendra l'ensemble des identificateurs apparaissant dans le modèle. A chacun de ces identificateurs pourra être associée l'information suivante : son type (entity, view, relation, ...), son statut nouveau ou ancien (pour les contrôles d'extension d'un modèle), le nombre de définitions de cet identificateur (pour les contrôles d'unicité des identificateurs), ainsi que son nombre d'utilisations (pour les contrôles d'utilisation d'identificateurs définis).

Les primitives associées à cette structure seraient du type:

-insérer (id, type, statut) : insertion de la définition d'un identificateur dans la structure;

-supprimer (id) : suppression de la définition d'un identificateur;

-est-défini (id) : fonction renvoyant vrai si l'identificateur est présent dans la table.

....

Ces primitives seraient activées lors de la création ou la suppression d'un identificateur (par la routine associée). Si l'utilisateur définit de façon multiple un identificateur, l'opération peut soit être interdite (si les deux identificateurs sont de type différent, par exemple), soit une erreur est détectée.

b. utilisation d'identificateurs non définis.

Principe : pour chaque utilisation d'un identificateur, mettre à jour le champ adéquat de la structure 'list-id'. Si l'identificateur n'est pas défini bien qu'utilisé, détecter une erreur.

c. propriétés non vérifiées par les identificateurs.

c.1. contrôles d'extensibilité .

Principe : consulter la structure list-id pour vérifier que tous les identificateurs concernés ont le statut adéquat. Ces contrôles sont associés aux différentes structures de données.

Exemple : lors de l'analyse de la structure mémorisant les relations de sous-typage, si une ancienne entité est détectée comme sous-type d'une nouvelle, une erreur est générée.

c.2. contrôles globaux.

Principe : pour chaque type de contrôle, concevoir une structure de données mémorisant l'information adéquate. Les contrôles sont effectués par analyse globale de ces différentes structures. Celles-ci sont mises à jour en parallèle avec le processus d'édition par activation des routines.

Exemple : considérons le contrôle portant sur la non présence de circuit dans les définitions de relation de sous-typage. Nous pouvons envisager une structure de données à laquelle serait associées les primitives :

-insérer (id, listid) : mémorisation d'une occurrence de relation de sous-typage;

-supprimer (id, id) : suppression d'une occurrence de relation de sous-typage;
-est-sous-type (id) : fonction renvoyant vrai si l'identificateur est un sous-type d'un autre.
Ces primitives sont activées soit par les routines sémantiques, soit lors des contrôles effectués sur la validité sémantique de l'arbre édité.

2.1.3. Approche par grammaire attribuée.

2.1.3.1. Remarques générales.

Rappelons que le principe de cette approche est d'associer aux opérateurs de l'arbre des attributs dont le rôle est de dériver l'information sémantique nécessaire à la vérification des contraintes.

Les contraintes identifiées dans le cadre du langage de définition d'un modèle de CV sont de deux types : nous avons d'une part l'ensemble des contraintes d'intégrité du modèle et d'autre part les contraintes d'extensibilité. Le premier type ne pose aucun problème puisque ces contraintes ne font appel qu'à de l'information présente dans l'arbre édité.

Comme nous l'avons déjà mentionné, le deuxième type de contrôle peut également être ramené à des contraintes statiques (la statut nouveau ou ancien d'un identificateur pouvant être déduit de sa position dans l'arbre).

Dans les exemples présentés ci-dessous, nous avons choisi de ne pas formaliser les équations sémantiques dans le langage SSL. Ceci nous permettra d'alléger notre présentation.

Nous avons pris les conventions de représentation suivantes : nous désignerons l'attribut a d'un opérateur op par la notation <op>.a .

L'équation définissant un attribut sera de la forme

<opi>.a = f (<op1>.b ,..., <opn>.k) ,

où f est une fonction dont nous ne préciserons que la spécification et opk est un opérateur apparaissant dans une règle de production.

Nous noterons l'opérateur complémentaire désignant un noeud méta par null.

Nous désignerons également par <α-section> l'ensemble des opérateurs tels que $\alpha \in$ {view,attribut, statment, entity, part, relation }.

2.1.3.2. **Principes et exemples.**

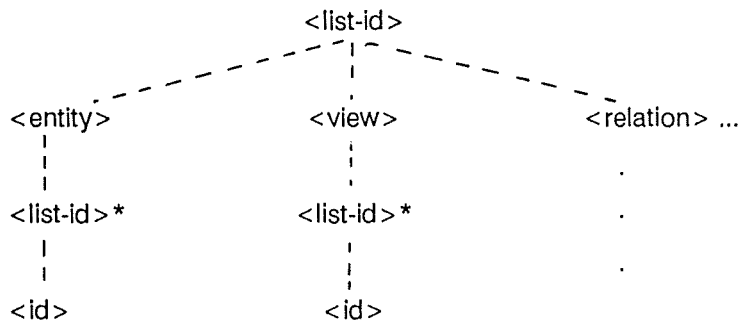
(i) contrôles de non multiplicité.

Le principe est d'associer aux opérateurs ‹- section un attribut qui synthétise le nombre d'apparitions de l'opérateur fils wording. Si la valeur de cet attribut est supérieure à 2, un message d'erreur est généré.

(ii) contrôles liés au contexte.

a. unicité des identificateurs :

Principe : définir un attribut 'list-id' pour l'opérateur <model-text>. Cet attribut aura comme rôle de synthétiser l'ensemble des définitions d'identificateurs dans le modèle. Ces définitions pourraient être partitionnées par type d'identificateur (view, entity-type, relation,...) suivant le schéma suivant :



Cet attribut sera consulté au niveau de chaque opérateur définissant un nouvel identificateur (i.e. <id> fils d'un opérateur ‹-section>). Une erreur sera détectée si l'identificateur défini est présent en plusieurs exemplaires dans 'list-id'.

b. utilisation d'identificateurs non définis.

Principe : pour tout opérateur représentant l'utilisation d'un identificateur, consulter l'attribut 'list-id'. Si cet opérateur n'apparaît pas dans 'list-id', générer un message d'erreur.

c. propriétés non vérifiées par les identificateurs :

c.1. contrôle d'extensibilité :

Principe : - ajouter à chaque identificateur apparaissant dans l'attribut 'list-id' un indicatif de son statut nouveau ou ancien .

- consulter cet indicatif pour diagnostiquer la validité d'une extension.

Exemple : ajoutons à l'opérateur <entity-section> un attribut erreur identifiant la définition d'une relation de sous-typage sur un type-entité ancien. Cet attribut pourrait être défini de la manière suivante :

<entity-section>.erreur =
. "extension non valide" si ext-subtype (<id>,<entity-section>.subtype,<model-text>.list-id)=faux

. " " sinon

où la fonction ext-subtype renvoie

. vrai si (∀ id défini dans subtype,statut(id) = nouveau dans <model-text>.listid)

. faux sinon.

c.2. contrôles globaux

Principe : faire remonter l'information dispersée dans chaque <x-section> jusqu'à un opérateur où les contrôles peuvent être faits.

Exemple : considérons la relation de sous-typage. Pour pouvoir diagnostiquer la présence d'un circuit, il faut connaître toutes les relations de sous-typage définies dans le modèle. Nous allons dès lors définir un attribut synthétique 'subtype' associé à l'opérateur <model-text> qui synthétisera cette information. La valeur de cet attribut est construite à partir d'autres attributs définis par les équations :

<subtype>.subtype = <list-id>
<entity-text>.subtype = union (<subtype>.subtype)

où union est une fonction construisant la liste de
toutes les sous-listes d'identificateurs déclarés sous-type dans la section.

<entity-section>.subtype =

. null si ((<id> = null)∨(<entity-text>.subtype = null)

.<couple> (<id>, <entity-text>.subtype) sinon

Nous pouvons ainsi définir l'attribut <model-text>.subtype par l'équation :
<model-text>.subtype = concat (<entity-section>.subtype)

où concat crée une structure mémorisant l'ensemble des relations de sous-typage.

Pour ce même opérateur, nous définissons un attribut erreur qui détecte la présence de circuit :
<model-text>.erreur =
."circuit dans la relation de sous-typage" si circuit (<model-text>.subtype) = vrai
." " sinon

où circuit est une fonction booléenne renvoyant vrai si un circuit est détecté dans l'attribut subtype, faux sinon.

2.1.4. Approche sémantique naturelle

2.1.4.1. Remarques générales

Rappelons que le principe de cette approche est de se donner une théorie constituée d'un ensemble d'axiomes et de règles d'inférences: les états valides d'un arbre sont alors déterminés en démontrant des théorèmes particuliers dans cette théorie.

Tout comme dans l'approche par grammaire attribuée, le seul type de contrôles qui pourrait ici poser problème concerne les contrôles liés à la validité d'une extension: en effet, si la position d'un identificateur ne suffit pas à déterminer son statut nouveau ou ancien (cfr. paragraphe 2.1.1), il faut insérer cette information dans l'arbre.

Dans les exemples présentés ci-dessous, nous avons repris le concept de classes qui permet de typer les variables TYPOL. A partir d'une définition FDL, ces classes peuvent être retrouvées d'après les règles suivantes:

. A tout opérateur sans constructeur correspond une classe dont les éléments sont les opérateurs fils de cet opérateur.

. A tout opérateur avec constructeur correspond une classe contenant cet opérateur.

2.1.4.2. Principes et exemples.

(i) contrôles de non multiplicité

Principe : concevoir un ensemble de règles établissant le nombre d'occurences d'un opérateur "wording" comme fils d'une section entité. Si ce nombre est supérieur à deux, échec de l'objectif.

(ii) contrôles liés au contexte.

a. unicité des identificateurs.

Principe : construire un environnement synthétisant l'ensemble des identificateurs définis dans le modèle. Comme dans les deux approches vues précédemment, cet environnement servira de base à l'analyse des contrôles liés au contexte et à la non déclaration multiple d'identificateurs. Il contiendra donc pour chaque identificateur défini, son nom, son type et son statut nouveau ou ancien. Le

processus de construction s'effectuera par analyse de chaque section définie. Il échouera si un identificateur doit être inséré deux fois dans l'environnement.

b. utilisation d'identificateurs non définis

Principe : vérifier que chaque identificateur utilisé est bien présent dans l'environnement construit ci-dessus , sinon échec du processus de démonstration.

c. propriétés non vérifiées par les identificateurs.

c.1. contrôle d'extensibilité

Principe : pour chaque contrôle, filtrer l'information adéquate de l'arbre et établir la validité de l'extension par analyse de l'environnement.

Exemple : reprenons la contrainte d'extensibilité liée à la définition d'une relation de sous-typage. Si nous considérons un environnement e défini par la grammaire suivante:

opérateurs	classes
env -> DEF+	ENV := env
def -> ID TYPE STATUS	DEF := def
	TYPE := entite view stat rel attr
	STATUS := nouveau ancien

nous pouvons vérifier la non définition d'une nouvelle relation de sous-typage sur un type entité ancien par les règles données ci-dessous. La variable e désigne l'environnement obtenu lors de la vérification de l'unicité des opérateurs. Les trois premières règles expriment que l'arbre d'un modèle est valide si les sous-arbres fils sont valides. La règle R4 dit que les sous-arbres correspondant à la définition d'une section autre q'une section entité sont valides. La règle 5 et 6 expriment la validité d'une section d'entité en terme de ses sous-arbres fils. La règle R7 établit la validité d'une liste d'entités sous-types de manière linéaire: chaque identificateur doit être un nouvel identificateur. Les règles R8 et R9 associent à chaque identificaateur son type en fonction de l'environnement.

[R1] $\dfrac{e \mid\!- \text{EXT-VAL (MODEL-TEXT)}}{e \mid\!- \text{EXT-VALIDE(point-d'entree(modele(ID,MODEL-TEXT)))}}$

[R2] $\dfrac{e \mid\!- \text{EXT-VAL(MODELE-SECTION)} \;\&\; e \mid\!- \text{EXT-VAL(MODEL-TEXT)}}{e \mid\!- \text{EXT-VAL(modele-text [MODEL-SECTION.MODELE-TEXT]}}$

[R3] $\dfrac{e \mid\!- \text{EXT-VAL(ENTITY\_TEXT)}}{e \mid\!- \text{EXT-VAL(entity-section(ID,ENTITY-TEXT))}}$

[R4] $e \mid\!- \text{EXT-VAL}(\alpha\text{-section(X,Y))}$ où $\alpha \in$ {view, statement,...} \ {entity}

[R5] $\dfrac{e \mid\!- \text{EXT-VAL(ENTITY-LINE)} \;\&\; e \mid\!- \text{EXT-VAL(ENTITY-TEXT)}}{e \mid\!- \text{EXT-VAL(entity-text[ENTITY-LINE.ENTITY-TEXT]}}$

C.2. contrôles globaux

Principe : pour chaque type de contrôle, analyser l'arbre en construisant l'information nécessaire à la détection d'une erreur. Les contrôles seront en général effectués en fin de construction puisque les informations sont réparties dans les différentes sections.

Exemple : reprenons le contrôle portant sur l'absence de circuit dans les relations de sous-typage. Nous allons construire par analyse de l'arbre une structure de données synthétisant les différentes relations définies dans l'arbre.Cette structure sera constituée d'une liste d'éléments où chaque élément est un couple. Le premier élément de ce couple sera une entité sur-type, tandis que le deuxième élément sera la liste des entités sous-types correspondante.Cette structure peut être décrite par la grammaire:

opérateurs	classes
soustype -> ELEMENT+	ELEMENT := element
element -> ID LISTID	LISTID := listid
listid -> ID+	SOUSTYPE := soustype

Elle peut être construite par la liste des règles données ci-dessous. St,st1,st2 y désignent une variable de type SOUSTYPE. La première règle spécifie qu'il n'y a pas de circuit dans les relations de sous-typage si la structure st ne contient pas de circuit; cette structure est construite en établissant le séquent:

soustype[] |- MODEL-TEXT : st

grâce aux règles R2 à R7 ci-après.

La règle R2 exprime que cette structure est construite de manière linéaire. La règle R3 dit qu'une section différente d'une section entité ne modifie pas cette structure. La règle R4 construit la structure st1 par analyse du sous-arbre de type ENTITY-TEXT. La règle R5 dit que cette analyse se fait de manière linéaire. La règle R6 exprime que chaque liste d'entités sous-type est ajoutée à la structure.

[R1] sous-type[] |- MODEL-TEXT : st |- PAS-CIRCUIT(st)
|- PAS-CIRCUIT (point-d'entree(modele(ID,MODEL-TEXT))

[R2] st |- MODEL-SECTION : st1 st1 |- MODEL-TEXT : st2
st |- modele-text [MODEL-SECTION. MODEL-TEXT] : st2

[R3] st |- -section (ID,X) : st $\alpha\epsilon$ {view, statement...}
\ {entity-section}

[R4] st |- CONS (ENTITY-TEXT,ID,st1)
st |- entity-section (ID,ENTITY-TEXT) : st1

[R5] st |- CONS (ENTITY-LINE,ID,st1) st1 |- CONS(ENTITY-TEXT,ID,st2)
st |- CONS(entity-text [ENTITY-LINE.ENTITY-TEXT,ID,st2])

[R6] st |- CONS (subtype(LISTID, ID, soustype [element(ID,LISTID).st])

[R7] st |- CONS (dedication-crt(X)) : st

Les règles suivantes détectent la présence d'un circuit dans la structure st construite ci-dessus. Ces règles ont été construites suivant le principe élémentaire suivant: on analyse la structure couple par couple. Pour chaque couple, on vérifie que l'identificateur n'est pas sur un circuit en étudiant les relations de sous-typages définies dans les couples suivants. A titre d'illustration Nous avons repris ci-dessous les règles principales:

[R8] La strusture st ne contient pas de circuit si chaque élément sur-type n'est pas sur un circuit.

|- PAS-CIRCUIT(ID,IDLIST,SUBTYPE) |- PAS-CIRCUIT(SUBTYPE)
|- PAS-CIRCUIT (subtype[element(ID,IDLIST).SUBTYPE])

[R9] Un identificateur ID n'est pas sur un circuit si ses successeurs directs et indirects sont différents de lui.

|- DIRECT(ID,IDLIST) idlist[ID.IDLIST] |- INDIRECTS(ID,SUBTYPE)
|- PAS-CIRCUIT (ID,IDLIST,SUBTYPE)

[R10] Chaque successeur direct est valide s'il est différent de id(x).

DIFF(X,Y) |- DIRECT(id(X),IDLIST)

|- DIRECT (id(X), idlist [id(Y).IDLIST])

|- DIRECT (id(x), idlist[])

[R11] Une liste L des successeurs de ID est construite par analyse linéaire de la structure. Chacun d'eux doit être différent de ID.

DIFF(ID,ID2) L|- INDIRECT(ID2,IDLIST):L1 L1|-INDIRECT(ID,SUBTYPE)

L |- INDIRECT (ID,subtype[element(ID2,IDLIST).SUBTYPE)]

2.2. Contrôles relatifs au langage de documentation

Nous avons présenté au paragraphe 1.3.2.2. les contraintes à vérifier lors de la mise à jour de la PDB.

Rappelons que le premier type de contrôle mis en évidence concerne la valeur des propriétés apparaissant dans la documentation. Ils n'ont pas été inclus dans la description de la syntaxe du langage pour ne pas générer des grammaires trop importantes.

Les autres types de contrôle concernent différentes connectivités méta-définies dans le modèle de CV considéré, les contraintes de type et les contraintes de définition des identificateurs utilisés dans la documentation.

Deux questions peuvent se poser :
- est-il possible d'exprimer ces contrôles dans les trois formalismes étudiés précédemment et de générer ces descriptions de manière automatique ?
- l'information nécessaire aux contrôles est-elle présente dans l'arbre édité?

A la première question la réponse est positive: la forme de l'arbre de documentation généré est en effet prédéfini (sur base du modèle de CV considéré) et ces contrôles correspondent à des contraintes déjà étudiées dans le paragraphe 2.1. où les méta-règles sous-jacentes à leur conception ont été explicitées . Leur description pourrait également être générée de manière automatique à partir de la description d'un nouveau modèle de cycle de vie.

Par contre, certaines vérifications nécessitent actuellement d'effectuer des requêtes à la PDB : il s'agit des contraintes de type, des contraintes de connectivité et des contraintes de définition des indentificateurs déclarés.

Pour permettre ces vérifications, nous proposons d'enrichir l'arbre d'une documentation de l'information suivante: pour chaque type d'entité en relation avec le type de l'objet édité
- le nom de toutes les occurrences de ce type.
- le nombre de connexions auquelles ces occurences participent dans le cadre de la relation-type correspondante.

Soulignons que cette analyse ne correspond qu'à une première réflexion. Un travail approfondi sur cette question devrait être mené par la suite.

2.2. Conclusion

Ce chapitre était consacré à l'étude des différents types de contraintes mis en évidence au chapitre 1 de cette deuxième partie. Cette analyse a mis en évidence les types de contraintes suivants:

. contraintes liées à un opérateur syntaxique précis, contraintes de non multiplicité, contraintes liées au contexte (non définition multiple d'identificateurs, définition des identificateurs utilisés, contraintes de type, contraintes d'intégrité et d'extensibilité du modèle) pour le langage de définition d'un modèle de CV.

. contraintes de valeurs pour les propriétés définies de l'objet logiciel mis à jour, contraintes de définition des identificateurs utilisés, contraintes de type et de connectivité pour le langage de documentation.

Cette étude a montré que ces contrôles pouvaient être intégrés dans les trois cadres formels étudiés dans la première partie.

Deux points de vue peuvent être adoptés pour le choix de l'un de ces trois formalismes:

. point de vue de l'utilisateur:

Pour l'utilisateur, il est important de réaliser les contrôles de la manière la plus efficace et de la manière la plus souple.

Remarquons que certains contrôles se prêtent à des vérifications effectuées en parallèle avec l'édition : il s'agit, par exemple, des contraintes liées à un opérateur précis, contraintes de non multiplicité, des contraintes de non multiplicité, contraintes de non définition multiple d'identificateurs, contraintes de valeur pour les propriétés définies et les contraintes générales de type. Ces contrôles peuvent en effet être rapidement réalisés et sont indépendant de l'ordre d'édition du texte formel. Pour les raisons opposées, il semble plus adéquat de réaliser les autres contrôles à la requête explicite de l'utilisateur.

Un autre critère pertinent pour un utilisateur est l'efficacité. Remarquons qu'il ne doit être pris en considération que si les contrôles sont réalisés en parallèle avec l'édition ou si le résultat de l'analyse doit être obtenu rapidemment (ce qui n'est pas nécessairement le cas lors de la définition d'une extension d'un modèle de CV).

Rappelons que l'approche par sémantique naturelle ne s'adapte pas à des contrôles réalisés de manière incrémentale en parallèle avec l'édition. D'autre part, c'est l'approche par routines sémantiques qui est la plus souple à cet égard: elle permet à la fois de signaler des erreurs sémantiques mais également d'interdire des opérations sémantiquement invalides. Cette facilité pourrait être intéressante pour interdire à l'utilisateur des erreurs fondamentales tels que la déclaration multiple d'un identificateur, la modification d'un attribut "wording",des erreurs de type. C'est également dans cette approche que les contrôles sont réalisés de la manière la plus efficace.

. point de vue du concepteur :

Si nous adoptons le point de vue du concepteur, nous pouvons classifier les trois cadres formels suivant l'ordre de préférence: sémantique naturelle, grammaire attribuée, routines sémantiques.

Rappelons en effet qu'à cet ordre correspond la plus grande facilité de conception, de validation, de modifiabilité, de réutilisabilité et de lisibilité des méta-descriptions des contrôles.

CONCLUSION

Le présent travail se proposait d'étudier trois approches du traitement des contraintes sémantiques en édition syntaxique et leur intégration dans l'atelier ALMA.

Nous avons effectué une étude aussi complète que possible des approches en question. Pour chacune d'entre elles, nous avons exposé successivement leurs objectifs, les idées sous-jacentes à la réalisation concrète de ces objectifs ainsi que les problèmes qui y sont liés. Ensuite, une étude des langages de spécification des contrôles et de leur interprétation par un noyau d'édition a été effectuée. Il nous a paru pertinent de ponctuer notre exposé d'exemples concrets: ainsi, le langage PICO introduit au début de ce travail a servi de base à une visualisation concrète de ces approches et à leur comparaison.

La présentation systématique de ces diverses approches nous a permis, d'une part, de réunir en un même ouvrage, des informations qui, quoique pertinentes, étaient relativement dispersées dans la documentation mise à notre disposition. D'autre part, notre démarche a rendu plus évidents les aspects communs et divergents de ces différentes approches.

Divers points ont ainsi pu être dégagés: si nous respectons l'ordre suivant - routines sémantiques, grammaire attribuée, sémantique naturelle - nous constatons un niveau d'abstraction de plus en plus élevé des langages de spécification des contrôles. Le travail du concepteur des méta-descriptions des contrôles est de plus en plus automatisé. De plus, pour ce même ordre, la lisibilité des spécifications est de plus en plus grande. Il est dès lors de plus en plus aisé de se convaincre que ces spécifications sont correctes et entrainent les vérifications attendues. De même, il en résulte une plus grande facilité de modifiabilité et de réutilisabilité des méta-descriptions. Ces avantages se soldent par un coût à payer en performances.

Nous avons ensuite analysé un ensemble de contraintes liées à deux langages de type Entité-Relation conçu dans le cadre de l'atelier ALMA: le premier est un langage de définition d'un modèle de cycle de vie d'un projet logiciel; le second est un langage de mise à jour d'une base de données de projet.

L'intérêt de cette analyse était double:
- d'une part, étudier la possibilité d'enrichir le noyau ALMA d'un éditeur offrant des services génériques qui permettent la validation sémantique des arbres édités. Rappelons qu'actuellement, les contrôles précités sont réalisés de manière "ad hoc", par des outils indépendants du noyau d'édition.
- d'autre part, étudier plus en profondeur chacun des trois cadres formels analysés dans la première partie et ce pour des langages plus complexes que le langage PICO, langages qui présentent des caractéristiques propres aux langages de type Entité-Relation.

L'analyse et la classification de ces contrôles ont montré que l'ensemble des contrôles pouvait être intégré dans les trois cadres formels présentés. Bien que le langage PICO et le langage de définition d'un modèle de cycle de vie d'un projet soient très différents, une études approfondie des contraintes qui y sont liées a mis en évidence une grande similitude des contrôles à réaliser. La même démarche d'analyse a donc pu être appliquée. Elle nous a permis de décrire un ensemble de méta-règles à appliquer lors de la conception des différentes méta-description des contraintes et ce pour chaque formalisme étudié dans la première partie. Une première ébauche d'une étude plus approfondie a pu ainsi être esquissée.

Quant au choix de l'une ou l'autre de ces approches, nous avons retenu deux positions: le point de vue de l'utilisateur du système et celui du rédacteur des contrôles à réaliser.Ces points de vue s'opposent: l'utilisateur privilégiera l'approche par routines sémantiques pour son efficacité et sa plus grande souplesse (contrôles réalisés de manière incrémentale en parallèle avec l'édition ou à sa requête explicite; interdiction d'opérations d'édition sémantiquement non valides ou simple avertissement d'anomalies sémantiques).
A l'autre extrême, le concepteur des contrôles choisira l'approche par sémantique naturelle: c'est dans cette approche que la rédaction des contrôles est la plus aisée, la plus sûre et la plus facile à modifier et à adapter.

Nous conseillons le choix de l'approche par grammaire attribuée qui nous semble présenter un bon compromis entre ces deux points de vue: elle remédie à l'aspect "ad hoc" de l'approche par routines sémantiques par une plus grande systématisation dans la rédaction des contrôles. Elle en garde la souplesse d'utilisation mentionnée ci-dessus tout en offrant un niveau acceptable d'efficacité.

BIBLIOGRAPHIE

AMBRIOLA V., KAISER G.E. et ELLISON R.J., 1984 : An Action Routine Model for ALOE. Dept of Computer Science, Carnegie- Mellon University. August 1984.

ARTHUR J. et RAMANATHAN J., 1981 : Design of Analyzers for Selective Program Analysis. IEEE Transactions on Software Engineering., vol. SE-7, N°1, pp.39-51. January 1981.

CLEMENT D., DESPEYROUX T., DESPEYROUX J. et KAHN G., 1985: Naturel Semantics on the Computer, INRIA rapport de recherche 416, June 1985.

CLEMENT D., 1987: The Natural Dynamic Semantics of Mini-Standard ML. A paraitre dans Proceedings CFLP, Pisa, March 1987.

DESPEYROUX J., 1986 : Proof of Translation in Natural Semantics.Logic in Computer Science, Cambridge, Massachussets, June 1986.

DESPEYROUX T., 1987 : TYPOL, A Formalism to Implement Natural Semantics. Draft Version INRIA. August 1987.

DONZEAU-GOUGE V., LANG B. et MELESE B., 1974 : Practical Applications of a Syntax-directed Program Manipulation Environment. Proc 7th Int. Conf. on Software. Engineering, Orlando, Florida. March 1974.

DONZEAU-GOUGE V., HUET G., KAHN G. et LANG B., 1980 : Programming Environments Based on Structured Editors: The Mentor Experience. Rapport de recherche n°26, INRIA. July 1980.

DONZEAU-GOUGE V., KAHN G., LANG B., MELESE B., MARCOS E.,1983: Outline of a Tool for document Manipulation. IFIP, PARIS. September, 1983.

DONZEAU-GOUGE V., DUBOIS C., FACON P. et Jean F., 1987:Development of a Programming Environment for SETL. 1° conf. internationnale de génie logiciel. ESEC pp 23-34, 1987

GERHART S.L., 1975 : Correctness-Preserving Program Transformations. Conf. Record of the 2° ACM Symp. on Principles of Programming Languages. Paolo Alto , Californie, pp 54-66. Janvier 20-22 1975.

HABERMANN N., ELLISON R., MEDINA-MORA R., FEILER P., NOTKIN D.B. et POGOVITCH S., 1982 : The Second Compendium of Gandalf Documentation. Dept of Computer Sciences, Carnegie-Mellon University. May 1982.

HABERMANN A. et NOTKIN D., 1986 : GANDALF: Software Development Environments. IEEE Transactions on Software Engineering : vol.12, n°12. December 1986.

HABERMANN A., ELLISON R.J., NOTKIN D.S., KAISER G.E., STAUDT B.J. et AMBRIOLA V., 1985: Special Issue on the Gandalf Project. The Journal of the Systems and software 5(2), May 1985.

HORWITZ S.B., 1985 : Generating Language-Based Editors : A Relationally-Attributed Approach. Ph. D. Thesis. Dept of Computer Science,Cornell University. August 1985.

KAHN G., 1987 : Natural Semantics. Proc. of Symp. on Theoritical Aspects of Computer Science, Passau, Germany. February 1987.

KAHN G., LANG B., MELEX B. et MARCOS E., 1983 : METAL : a Formalism to Specify Formalisms. Science of Computer Programming 3, North Holland, 151-188, 1983.

KAISER G.E., 1985 : Semantics for Structure Editing Environments. Ph. D. Thesis. Dept of Computer Science. Carnegie-Mellon University. May 1985.

KAISER G.E. et KRUEGER C.W., 1986 : Using the New Gandalf System. Dept of Computer Science, Carnegie-Mellon University. 1986.

KAISER G.E., KAPLAN S.M. et MICALLEF J., 1987 : Multiuser, Distributed Language-Based Environments. IEEE Software, pp.58-67. 1987.

KLINT P., 1983 : A Survey of Three Language-Independent Programming Environments. Rep IW 240 / 83, Mathematisch Centrum. November 1983.

KNUTH D.E., 1968 : Semantics of Context-Free Languages. Mathematical Systems Theory vol n° 2,2, pp.127-145. June 1968.

MEDINA-MORA R. et NOTKIN D.S., 1981 : ALOE User's and Implementors ' Guide. Dept of Computer Science, Carnegie-Mellon University. November 1981.

MEYER, 1986 : Introduction to the Theory of Programming Languages. Interactive Software Engineering, Santa Barbara, California. September 1986.

PLOTKIN G.D., 1981 : A structural Approach to Operational Semantics, DAIMI FN 19, Dept of Computer Science, Aarhus University. September 1981.

REPS T., TEITELBAUM T. et DEMERS A., 1983 : Incremental Context-Dependant Analysis for Language-Based Editors ACM Transaction on Programming Language and System. vol 5 n°3. July 1983.

REPS T., 1984 : Generating Language-Based Environments. The M.I.T. Press, Cambridge, Masso, Massachussetts, 1984.

REPS T. et TEITELBAUM T., 1984 : The Synthesizer Generator, ACM, Transaction on Programming Language and System. 1984.

REPS T. et TEITELBAUM T., 1987 : The Synthesizer Generator Reference Manual. Dept of Computer Science, Cornell University. 1987.

STAUDT B.J., KRUEGER C.W., HABERMANN A.N. et AMBRIOLA V. 1986: The Gandalf System Reference Manuals. Dept of Computer Science, Carnegie-Mellon University. May 1986.

STAUDT B.J., 1986 : The Implementor's Guide to Writing Daemons for ALOE. Dept of Computer Science, Carnegie-Mellon University. May 1986.

STAUDT B.J. et AMBRIOLA V., 1986: The Aloe Action Routine Language Manuel. Dept of Computer Science, Carnegie-Mellon University. May 1986.

TENNENT R.D., 1976 : The Denotational Semantics of Programming Languages. Communications ACM, vol.19, n°8, pp.437-453. August 1976.

VAN LAMSWEERDE A., BUYSE M., DELCOURT B., DELOR E., SCHAYES

M.C., BOUQUELLE J.P., CHAMPAGNE R., NISOLE P. et SELDESLACHTS J. 1987 : The Kernel of a Generic Software Development Environment. Proc. 2nd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environment, Palo Alto, December 1986, SIGPLAN Notices 22(1), 208-217, January 1987.

VAN LAMSWEERDE A., DELCOURT B., DELOR E., SCHAYES M.C. et CHAMPAGNE R. 1987 : Generic Lifecycle Support in the ALMA Environment. Report RP 87/7, May 20, 1987.

ANNEXES

ANNEXE 1

Cette annexe concerne une étude détaillée de l'algorithme présenté en 2.3.5.2. de la première partie. Rappelons que cet algorithme est utilisé dans le Synthesizer Generator pour réévaluer un nombre minimal d'attributs après une modification locale de l'arbre.

Dans cette annexe, nous allons reprendre plus en détail l'algorithme présenté en 2.3.5.2.

A cet effet, nous allons d'abord présenter un exemple qui nous servira d'illustration. Ensuite, nous préciserons quelques définitions utiles pour formaliser les concepts sous-jacents à l'algorithme. Nous présenterons ensuite une première version de ce dernier. Celle-ci sera généralisée dans la suite pour assurer un travail de réévaluation minimum. Nous verrons également les différentes informations maintenues par l'éditeur en cours d'édition. Enfin, nous présenterons une version tenant compte de différentes particularités du langage SSL (cfr. 1° partie 2.3.5.2)

1. Exemple

Pour expliciter les algorithmes, nous allons reprendre un exemple introduit dans [Reps et al., 83]. Il s'agit d'un problème de formatage. Supposons la grammaire suivante :

```
ROOT --> S
S    --> S S
S    --> word 1
  .
  .
  .
S    --> word n
```

Elle génère des phrases d'un ou plusieurs mots séparés par des blancs. Ces mots sont choisis dans le vocabulaire word1,..., wordn. Supposons de plus que ces phrases doivent être imprimées sur un écran de largeur maximale W de façon telle que chaque ligne contienne un maximum de mots et qu'aucun d'eux ne soit coupé en deux. Faisons encore l'hypothèse que chaque mot est de longueur inférieure à W et que les colonnes de l'écran sont numérotées.

Le problème posé est alors le suivant : nous désirons associer à chaque mot S un attribut synthétique 'S.last' qui désigne la colonne de l'écran où est imprimé le dernier caractère de ce mot. Pour définir cet attribut, nous devons connaître la colonne du dernier caractère du mot précédent immédiatement le mot concerné. Nous pouvons représenter cette donnée par l'attribut hérité 'S.previous'.

Ce qui nous donne les équations sémantiques suivantes :

```
ROOT --> S
```

S.previous = -1

S1 --> S2 S3

S2.previous = S1.previous

S3.previous = S2.last

S1.last = S3.last

S --> word1

S.last = if (S.previous+1+length(word1) W)

then S.previous+1+length(word1)

else length(word1)

etc.

Pour fixer les idées, prenons W=13 et considérons la séquence suivante :

11234567890123

Candy is

dandy but

liquor is

quicker

Un des arbres de dérivations possibles pour cette phrase est présenté fig.1.

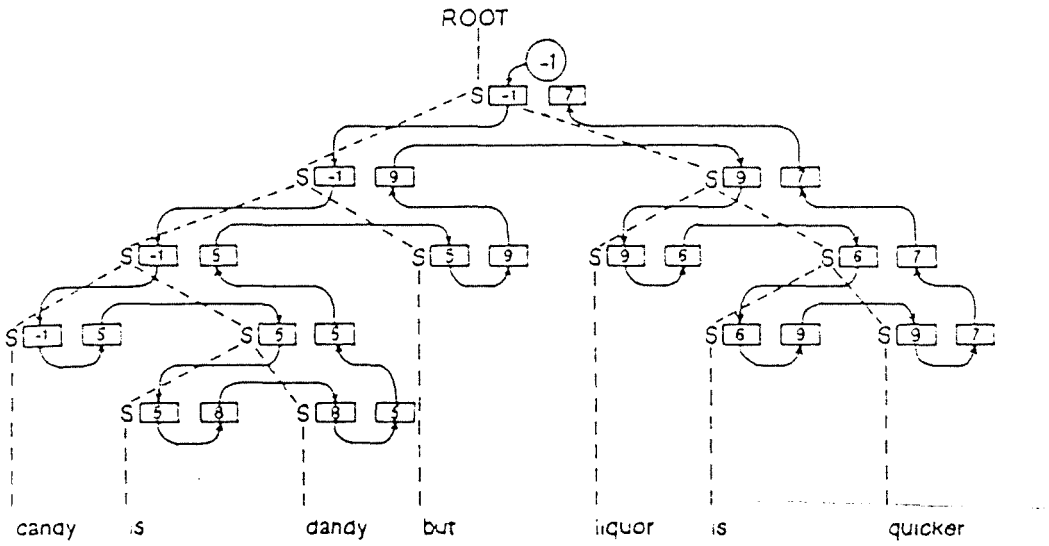


FIG. 1.

2. Quelques définitions

Avant d'étudier plus en profondeur les algorithmes, introduisons les définitions suivantes:

(i) étant donné les graphes orientés A=(Va,Ea) et B=(Vb, Eb)

l'union de A et B est définie par :

$$A \cup B = (Va \cup Vb, Ea \cup Eb)$$

(ii)sous les mêmes hypothèses, la suppression de B de A définie par :

$$A - B = (Va, Ea - Eb)$$

Remarquons que la suppression ne détruit que les arcs de B

(iii) étant donné le graphe orienté A=(V, E) et l'ensemble de sommets V'c V , la projection de A sur V' est définie par :

$$A/V' = (V', E')$$

avec $E' = \{(v,w) \mid v,w \in V'$ et il existe un chemin de v à w dans A qui ne contient aucun élément de $V'\}$

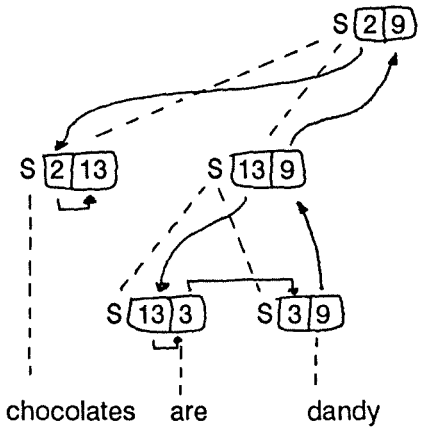
(iv) Les dépendances transitives entre les attributs d'un noeud de l'arbre peuvent être représentées localement par les graphes caractéristiques subordonnés et supérieurs. Dans la suite de cette annexe, nous noterons le graphe caractéristique subordonné d'un noeud s par s.C , tandis que $s.\overline{C}$ désignera le graphe caractéristique supérieur de ce même noeud. Le graphe caractéristique subordonné d'un noeud s correspond à la projection des dépendances du sous-arbre de racine s sur les attributs de s. Pour former le graphe caractéristique supérieur de s, nous pouvons imaginer que le sous-arbre en s a été détaché de l'arbre attribué. Il consiste alors en la projection des dépendances sur les attributs de s. Remarquons que les sommets des graphes caractéristiques correspondent aux attributs de s.

De manière plus formelle, supposons que s est un noeud de l'arbre attribué T, que Ts désigne le sous-arbre de racine s et que Vs désigne les arcs du graphe de dépendance D(T) qui correspondent aux attributs de s. Les graphes caractéristiques subordonnés et supérieurs au noeud s peuvent être définis par

$$s.C = D(Ts) \,/\, Vs$$
$$s.\overline{C} = (D(T) - D(Ts)) \,/\, Vs$$

Connaissant les graphes caractéristiques subordonnés et supérieurs au point de remplacement r, nous pouvons construire le graphe r.C U r.$\overline{C}$. Un arc de ce graphe correspond à une dépendance transitive entre deux attributs de r.

Si nous reprenons notre exemple, nous pouvons concrétiser ces notions de la manière suivante : supposons que le sous-arbre "Candy is dandy" de la fig.1. soit remplacé par le sous-arbre libre présenté fig.2. :



- fig.2. -

Après avoir accroché ce sous-arbre, mais avant la propagation des modifications, les valeurs des attributs au point de remplacement r sont

r.previous = 2 (valeur de l'attribut hérité du sous-arbre remplacé)

r.last = 5 (valeur de l'attribut synthétique de la racine du sous-arbre remplacé)

Au point r, le graphe caractéristique r.C est

Il reflète la dépendance transitive existant de r.previous vers r.last. Le graphe caractéristique supérieur r.$\overline{C}$ est

L'absence d'arcs reflète le fait que r.previous ne dépend en aucune manière de r.last.

3. Premier algorithme de réévaluation.

Pour rééordonner les réévaluations, l'algorithme de proposé par Reps emploie un graphe M appelé modèle et un ensemble S utilisé comme liste de travail (cfr. 2.3.5.2). Initialement, le graphe M reçoit comme valeur r.C U r.$\overline{C}$, si r est le noeud de la modification. Ce graphe indique un ordre partiel suivant lequel réévaluer les attributs. Comme nous l'avons expliqué, ce graphe est évalué dynamiquement au fur et à mesure que l'évaluation des attributs progresse (cfr. fig. 22): si un attribut b est réévalué à une nouvelle valeur, et si b est un argument d'un attribut en dehors du modèle courant M, alors M est étendu d'une instance d'une production de telle sorte qu'il inclue les successeurs de b.

Pour décrire une extension de manière précise, nous définissons les fonctions Expanded Subordinate et Expanded Superior qui produisent des graphes qui sont des raffinements des graphes caractéristiques d'un noeud.

Si le noeud s0 est le noeud parent d'une instance d'une production p : <s0,....,sk> , nous définissons

Expanded Subordinate (s0) = D(p) U s1.C U ... U sk.C

Pour chaque autre noeud sj instance de cette production, nous définissons :

Expanded Superior (sj) = D(s) U s0.$\overline{C}$ U s1.C U ... U sj-1.C U Sj+1.C U ... U sk.C

Alors, un modèle est étendu par la procédure EXPAND donnée fig.3. En plus de détruire un graphe caractéristique du modèle et d'augmenter le modèle avec le graphe caractéristique étendu correspondant, une expension implique des insertions dans la liste de travail S. Chaque fois qu'un attribut est inséré dans le modèle, il est inséré dans la liste de travail si son degré intérieur vaut zéro. Il est en effet prêt dans ce cas à être réévalué.

EXPAND (M,b,S)

let M = un graphe orienté
b,c = des instances d'attributs
S = un ensemble d'instances d'attributs

in if ((il existe C, un successeur de b dans D(T) qui n'est pas dans M)
and (TreeNode(C) est un fils de TreeNode(b))) then

M:= (M-TreeNode(b).C) U Expanded Subordinate(TreeNode(b))

Insérer dans S tous les sommets de Expanded Subordinate
(TreeNode(b) de degré intérieur zéro

if((il existe c, un successseur de b dans D(T) qui n'est
pas dans M)
and (TreeNode(c) est parent de TreeNode(b)) then

M:=(M-TreeNode(b).C) U Expanded Subordinate(TreeNode(b))
Insérer dans S tous les sommets de Expanded Superior
(TreeNode(b)) de degré intérieur zéro.

- fig.3. -

Ces définitions posées, l'algorithme proposé par Reps est alors le suivant:

PROPAGATE (T,r)

let T = un arbre attribué
r = un noeud non terminal contenant tous les attributs in-
consistants de T
S = un ensemble d'instances d'attributs
M = un graphe orienté
b,c = des instances d'attributs
Oldvalue, Newvalue = des valeurs d'attributs

in M := r.C U r.$\overline{C}$
S:=l'ensemble des sommets de M de dgré intérieur zéro
dans M
while S <> ϕ do
choisir et supprimer un sommet b de S
Oldvalue:=valeur de b
evaluer b
Newvalue:=valeur de b
if((Oldvalue Newvalue) and (M ne contient pas tous
les successeurs de b dans D(T)))
then EXPAND(M,b,S)
for (chaque c qui est successeur de b dans M) do
enlever l'arc(b,c) de M
if degré-int(c)=0 then Insérer c dans S
od

od

- fig.4. -

Illustrons cet algorithme sur notre exemple. La fig.5. montre les modifications attendues si nous remplaçons "Candy is dandy" par "chocolates are dandy"

1234567890123	1234567890123
Candy is	chocolates
dandy but	are dandy but
liquor is	liquor is
quicker	quicker

fig.5.

Les différents états du modèle M sont montrés fig.6. Les attributs dont la valeur est réellement modifiée y sont entourés de doubles lignes, avec leur valeur initiale au dessus de leur valeur finale. Chaque attribut qui est inclus dans le modèle est réévalué. Remarquons que le modèle n'inclut ni les attributs de "is" ni ceux de "quicker" : en effet, bien que l'attribut S.previous de la phrase "is quicker" est recalculé, il ne change pas de valeur.

La fig.7. montre en détails les quatre premiers états du modèle M .

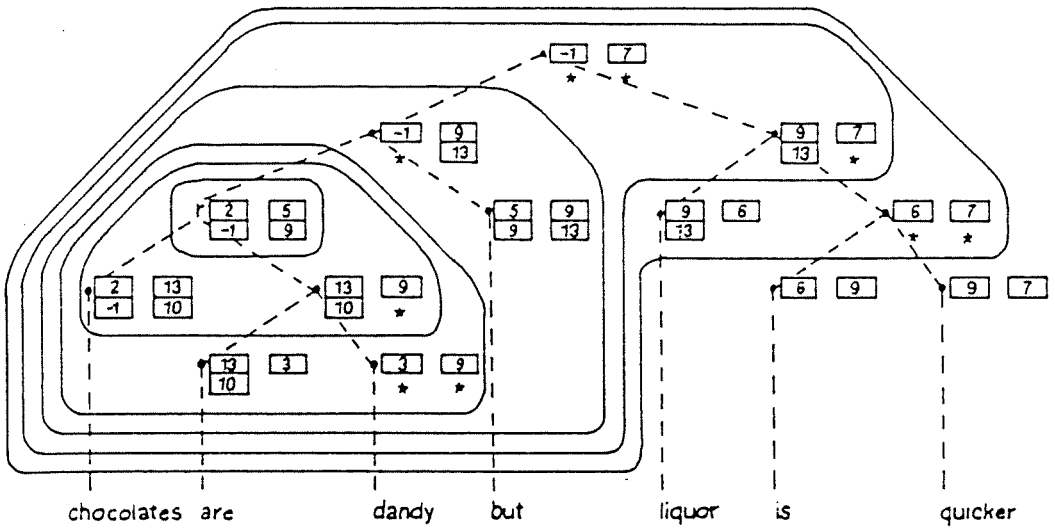

chocolates
are
dandy
but
liquor
is
quicker

Fig. 6

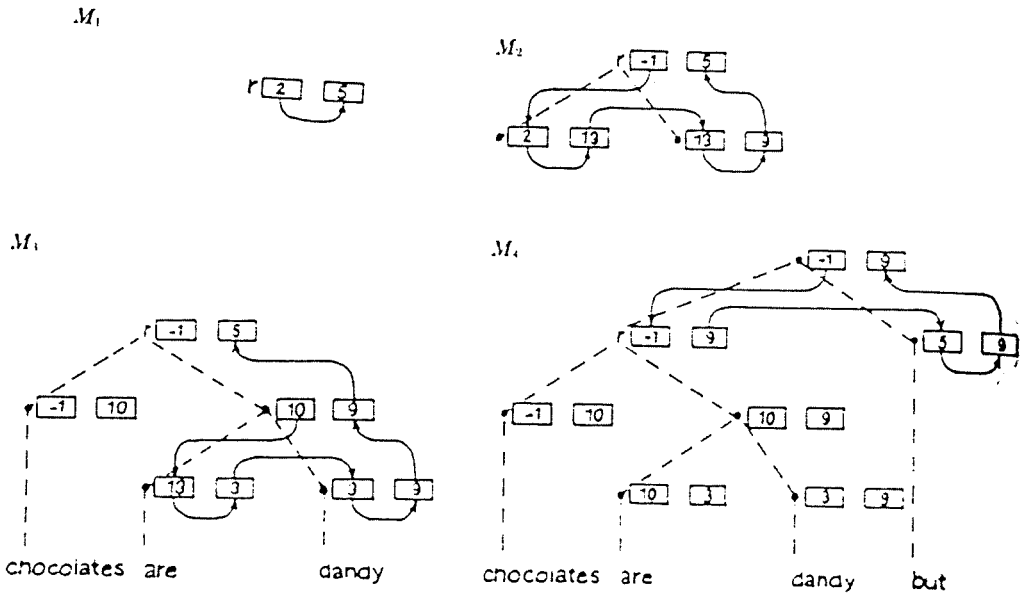

M_1
M_2
M_3
M_4
chocolates
are
dandy
chocolates
are
dandy
but

Fig 7.

4. Deuxième algorithme de réévaluation

L'algorithme présenté fig.4. présente encore un défaut : une instance d'un attribut qui fait partie de M est évalué même si aucun de ses arguments ne reçoit une nouvelle valeur. Par exemple, la figure 6 contient neuf instances d'attributs (marqué d'une *) qui n'ont jamais besoin d'être réévalués. De plus, ce phénomène peut se présenter plusieurs fois pour un même attribut. Par exemple, dans la figure 7, la présence d'un arc vers l'attribut r.previous implique que cet attribut sera réévalué bien que de toute évidence sa valeur ne sera plus modifiée. Comme en général, la réévaluation d'un attribut peut être coûteuse, on évite ces situations.

Ces évaluations inutiles peuvent être évitées en utilisant un nouvel ensemble que nous baptiserons NeedToBeEvaluated. Cet ensemble est défini de la manière suivante :

(i) NeedToBeEvaluated est initialisé à tous les sommets du modèle initial;

(ii) Quand la valeur d'une instance d'un attribut b est modifiée, chaque successeur de b est inséré dans NeedToBeEvaluated;

(iii) Quand b est enlevé de S, il n'est réévalué que si b appartient à NeedToBeEvaluated.

Ces idées sont insérées dans l'algorithme PROPAGATE présenté fig.8.

```
PROPAGATE (T,r)

let T= un arbre attribué
     r= un noeud non terminal contenant tous les attributs
   inconsistants de T
     S, NeedToBeEvaluated= un ensemble d'instances d'attributs
     M= graphe orienté
     b,c= instances d'attributs
     changed= booleen
     OldValue, Newvalue= valeurs d'attributs

     in M:= r.C U r.C̄
       S:= l'ensemble des sommets de M de dgré intérieur null
             dans M
       NeedToBeEvaluated:= l'ensemble des sommets de M
       while S <> ϕ do
                  sélectionner et supprimer un sommet b de S
```

```
                changed:=false
    if (b  NeedToBeEvaluated) then
          enlever b de NeedToBeEvaluated
        Oldvalue:= valeur de b
        évaluer b
        NewValue:= valeur de b
        if (OldValue <> NewValue) then
            changed:= true
                if (M ne contient pas tous les
              successeurs de b dans D(T) then
              EXPAND (M,b,S)

    for (chaque c qui est successeur de b dans M) do
        enlever l'arc (b,c) de M
        if (degré-int(c)=0) then insérer c dans S
            if (changed=true) then insérer c dans
                    NeedToBeEvaluated
    od
od
```

- fig.8. -

5. Invariant maintenu par l'éditeur.

Dans l'exposé ci-dessus, nous avons supposé implicitement que les graphes caractéristiques subordonnés étaient connus pour chaque noeud de l'arbre. Cependant, le remplacement d'un sous-arbre peut modifier complètement les dépendances transitives entre attributs. Dès lors, une telle hypothèse rend les opérations de remplacement très coûteuses.

Remarquons que l'algorithme PROPAGATE n'a pas besoin de chaque graphe caractéristique : en effet, après chaque remplacement à un noeud r, PROPAGATE n'a plus jamais besoin des graphes caractéristiques subordonnés d'aucun noeud du chemin de r vers la racine de l'arbre. D'autre part, il n'a jamais besoin des graphes caractéristiques supérieurs nulle part ailleurs. En fait, l'algorithme PROPAGATE n'a besoin des deux graphes caractéristiques qu'au noeud r.

Nous dirons dès lors que T est préparé pour la réévaluation au noeud r si

(i) r est étiquetté avec à la fois son graphe caractéristique subordonné r.C et son graphe caractéristique supérieur r.$\overline{C}$;

(ii) chaque noeud s sur le chemin de r à la racine de T est étiquetté avec son graphe caractéristique supérieur $s.\overline{C}$;

(iii) chaque noeud T ne se trouvant pas sur le chemin de r à la racine de T, est étiquetté avec son graphe caractéristique subordonné t.C.

Dans le Synthetisor Generator, l'éditeur maintient l'invariant que l'arbre attribué est préparé pour la réévaluation à la position du curseur utilisateur. Cet invariant est réétabli après chaque mouvement de ce curseur à un nouvel emplacement.

Plus précisément, chaque mouvement du curseur est défini par une séquence d'opérations AscendToParent et DescendToChild (j). Si le curseur utilisateur est positionné sur le noeud r de T et que T est préparé pour la réévaluation en r, AscendToParent a l'effet suivant :

parent(r).C := ExpandedSubordinate(parent(r))/{attributs
de parent(r)}

Tandis que DescentToChild(j) est défini par :

rj.C := ExpandedSuperior(rj)/{attributs de rj}
où rj désigne le je fils de r.

L'invariant que l'arbre doit être préparé pour la réévaluation à la position du curseur utilisateur doit également être réétabli après un remplacement d'un sous-arbre, avant l'appel à PROPAGATE. Par conservation, lorsqu'un sous-arbre est détaché de l'arbre principal, l'arbre libre ainsi obtenu est préparé pour la réévaluation à sa racine. Après qu'un sous-arbre U de noeud r ait été remplacé par un sous-arbre U' de noeud s, l'invariant est dès lors réétabli en assignant au graphe caractéristique supérieur du curseur la valeur $r.\overline{C}$ et au graphe caractéristique subordonné la valeur de s.C.

6. Le langage SSL et l'évaluateur incrémental.

Comme nous l'avons vu au paragraphe 2.3.4., le langage SSL permet de spécifier la valeur d'un attribut comme fonction de plusieurs types d'arguments. Rappelons que ceux-ci peuvent être non seulement des attributs de la production, mais également une référence syntaxique à une partie de l'arbre en train d'être édité. Une telle possibilité complique le problème de l'évaluation incrémentale des attributs : en effet, la valeur d'un tel attribut peut devenir inconsistante dès qu'on modifie le sous-arbre référencé dans l'équation. Dès lors, une modification d'un noeud M peut rendre inconsistants les attributs des noeuds situés le long du chemin de M à la racine (et non plus uniquement dans le voisinage de M). Cependant, nous pouvons utiliser l'algorithme d'évaluation présenté ci-dessus puisqu'il suffit de connaître la région de l'arbre contenant tous les attributs inconsistants.

Nous avons également introduit au paragraphe 2.3.5.3 la notion d'attributs spécifiés comme "facultatif". Ces attributs ne sont évalués que sur une demande explicite de l'utilisateur ou de l'évaluateur incrémental. Cette possibilité du Synthesizer Generator permet d'éviter de réévaluer des attributs dont la valeur n'a aucun effet observable après une opération d'édition.

Si de tels attributs sont introduits dans la spécification des contrôles sémantiques, l'algorithme PROPAGATE présenté fig.8. doit être modifié. Nous avons présenté une nouvelle version fig.9. : les attributs "facultatif" y sont traités comme des attributs normaux excepté qu'aucun de leurs arguments ne change de valeur : plus précisément, un attribut "facultatif" garde son ancienne valeur lorsqu'il est supprimé de la liste de travail s'il ne fait pas partie de NeedToBeEvaluated. Si un de ses arguments a changé de valeur, il reçoit alors la valeur NULL et est réévaluée si sa valeur est nécessaire à la réévaluation d'un attribut normal.

```
PROPAGATE (T,r)

let T = un arbre attribué
      r = un noeud non terminal contenant tous les attributs
    inconsistants de T
      S, NeedToBeEvaluated = un ensemble d'instances d'attributs
      M = graphe orienté
      b,c = instances d'attributs
      changed = booleen
      OldValue, Newvalue = valeurs d'attributs

      in M:= r.C U r.C̄
        S:= l'ensemble des sommets de M de dgré intérieur null
              dans M
        NeedToBeEvaluated:= l'ensemble des sommets de M
        while S <> ∅ do
                    sélectionner et supprimer un sommet b de S
                    changed:=false
       if (b  NeedToBeEvaluated) then
            enlever b de NeedToBeEvaluated
          Oldvalue:= valeur de b
          if (b est un attribut facultatif)
             then  b = Null
             else  for (chaque argument c de b
               qui est un attribut facultatif) do
                DEMANDVALUE(c)
```

```
            od
        évaluer b
        NewValue:= valeur de b
        if (OldValue <> NewValue) then
            changed:= true
                if (M ne contient pas tous les
            successeurs de b dans D(T) then
            EXPAND (M,b,S)
    for (chaque c qui est successeur de b dans M) do
        enlever l'arc (b,c) de M
        if (degré-int(c)=0) then insérer c dans S
            if (changed=true) then insérer c dans
                    NeedToBeEvaluated
    od
  od
```

```
DEMANDVALUE(b)

let  b,c = instances d'attributs

 in while (il existe c ,argument de b de valeur Null) do
     DEMANDVALUE(c)
     od
   evaluer b
```

- fig.9. -

ANNEXE 2

Définition LDF du langage de définition d'un modèle de cycle de vie (cfr 2° partie 1.3.1.)

```
(* laf *)
definition de meta ;



<point_d_entree> ::=    <model> ;
<model> ::=             "MODEL" <id> ";" @
                           <model_text> "." $ ;
L<model_text> ::=       <model_section>+ @ ;
SC<model_section> ::=   <view_section>          |
                        <statement_section>     |
                        <domain_section>        |
                        <attribute_section>     |
                        <part_section>          |
                        <relation_section>      |
                        <entity_section>        ;



<entity_section> ::=    "ENTITY-TYPE" <id> ";" #
                           <entity_text> ";" ;
L<entity_text> ::=      <entity_line>* ";" % ;
SC<entity_line> ::=     <dedication_etrt>   |
                        <sub_type>          |
                        <lexical_def_ident> |
                        <name_ident>        |
                        <description>       |
                        <wording>           ;
<wording> ::=           "WORDING =" <id> ;
<description> ::=       "DESCRIPTION =" # <long_text> ;
<name_ident> ::=        "IDENTIFIER =" <id> ;
<lexical_def_ident> ::= "IDENTIFIER'S LEXICAL DEFINITION =" <predefined_lex> ;
<sub_type> ::=          "IS SUB-TYPE OF" <list_id> ;
<dedication_etrt> ::=   "IS DEDICATED TO" <list_dedic_etrt> ;
L<list_dedic_etrt> ::=  <elt_dedic_etrt>+ "," % ;
<elt_dedic_etrt> ::=    <id> "WITH ACCESS MODE" <access_etrt> ;
SC<access_etrt> ::=     <nn_create_delete>  |
                        <ny_create_delete>  |
                        <yn_create_delete>  |
                        <yy_create_delete>  ;
<yy_create_delete> ::=  "create_delete" ;
<yn_create_delete> ::=  "create_notdelete" ;
<ny_create_delete> ::=  "notcreate_delete" ;
<nn_create_delete> ::=  "notcreate_notdelete" ;
```

```
<relation_section> ::=  "RELATION-TYPE" <id> ";" #
                        <relation_text> ";" ;
L<relation_text> ::=    <relation_line>* ";" % ;
SC<relation_line> ::=   <dedication_etrt> |
                        <association>     |
                        <description>     |
                        <wording>         ;
<association> ::=       "THE PART" <id> "IS PLAYED BY" <actors> ;
L<actors> ::=           <actor>+ "," % ;
<actor> ::=             <range> <id> ;
SC<range> ::=           <i_n> |
                        <i_i> ;
<i_i> ::=               <uns_integer> ^"-"^ <uns_integer> ;
<i_n> ::=               <uns_integer> ^"-N" ;
```

```
<part_section> ::=      "PART" <id> ";" #
                          <part_text> ";" ;
L<part_text> ::=        <part_line>* ";" % ;
SC<part_line> ::=       <description> |
                        <wording>     ;
```

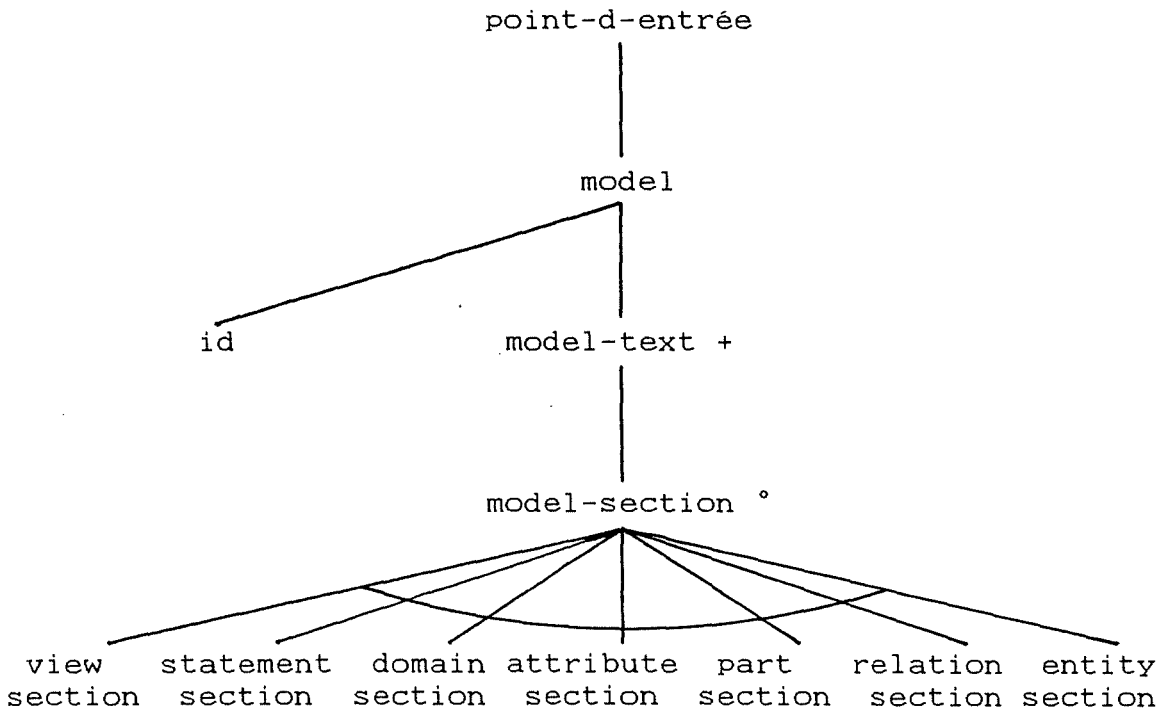
```
<attribute_section> ::= "ATTRIBUTE" <id>  ";" #
                           <attribute_text>  ";" ;
L<attribute_text> ::=   <attribute_line>* ";" % ;
SC<attribute_line> ::=  <dedication_att>   |
                        <values>           |
                        <decomposition>    |
                        <identification>   |
                        <characterization> |
                        <description>      |
                        <wording>          ;
<characterization> ::=  "CHARACTERIZES" <list_char> ;
L<list_char> ::=        <elt_char>+ "," % ;
<elt_char> ::=          <id> "WITH CONNECTIVITY" <range>;
<identification> ::=    "IDENTIFIES" <list_id> ;
<decomposition> ::=     "CONTAINS" <list_att> ;
L<list_att> ::=         <triplet>+ "," % ;
<triplet> ::=           <range> <id> <position> ;
<position> ::=          "("^ <uns_integer> ^")" ;
<values> ::=            "ACCEPTS DOMAIN(S)" <list_id> ;
<dedication_att> ::=    "IS DEDICATED TO" <list_dedic_att> ;
L<list_dedic_att> ::=   <elt_dedic_att>+ "," % ;
<elt_dedic_att> ::=     <id> "WITH ACCESS MODE" <access_att> ;
SC<access_att> ::=      <n_modify> |
                        <y_modify> ;
<y_modify> ::=          "modify" ;
<n_modify> ::=          "notmodify" ;
```

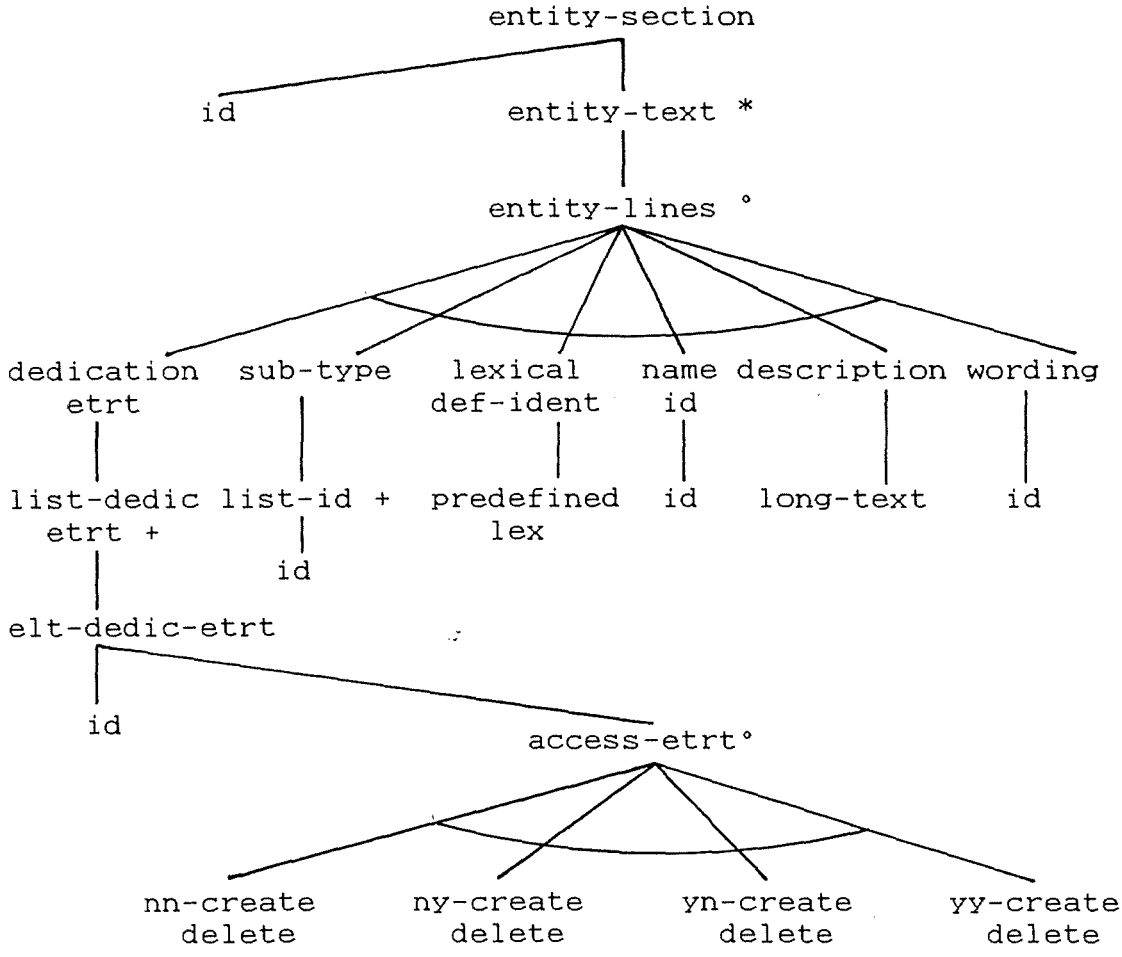
```
<domain_section> ::=      "DOMAIN" <id> ";" *
                            <domain_text> ";" ;
L<domain_text> ::=        <domain_line>* ";" % ;
SC<domain_line> ::=       <range_values>  |
                          <list_values>   |
                          <lexical_def>   |
                          <size>          |
                          <type>          |
                          <description>   |
                          <wording>       ;
<type> ::=                "TYPE =" <predefined_type> ;
SC<predefined_type> ::=   <informal_text> |
                          <formal_text>   |
                          <real_type>     |
                          <integer_type>  |
                          <char_type>     ;
<real_type> ::=           "real" ;
<integer_type> ::=        "integer" ;
<char_type> ::=           "char" ;
<formal_text> ::=         "formal text" ;
<informal_text> ::=       "informal text" ;
<size> ::=                "SIZE =" <uns_integer> ;
<list_values> ::=         "LIST =" <list> ;
L<list> ::=               <element>+ "," ;
SC<element> ::=           <integer>       |
                          <real> ;
<range_values> ::=        "RANGE =" <element> ".." <element> ;
<lexical_def> ::=         "LEXICAL DEFINITION =" <predefined_lex> ;
SC<predefined_lex> ::=    <identifier>  |
                          <string> |
                          <def_lex>  ;
<identifier> ::=          "identifier" ;
<string> ::=              "string" ;
```

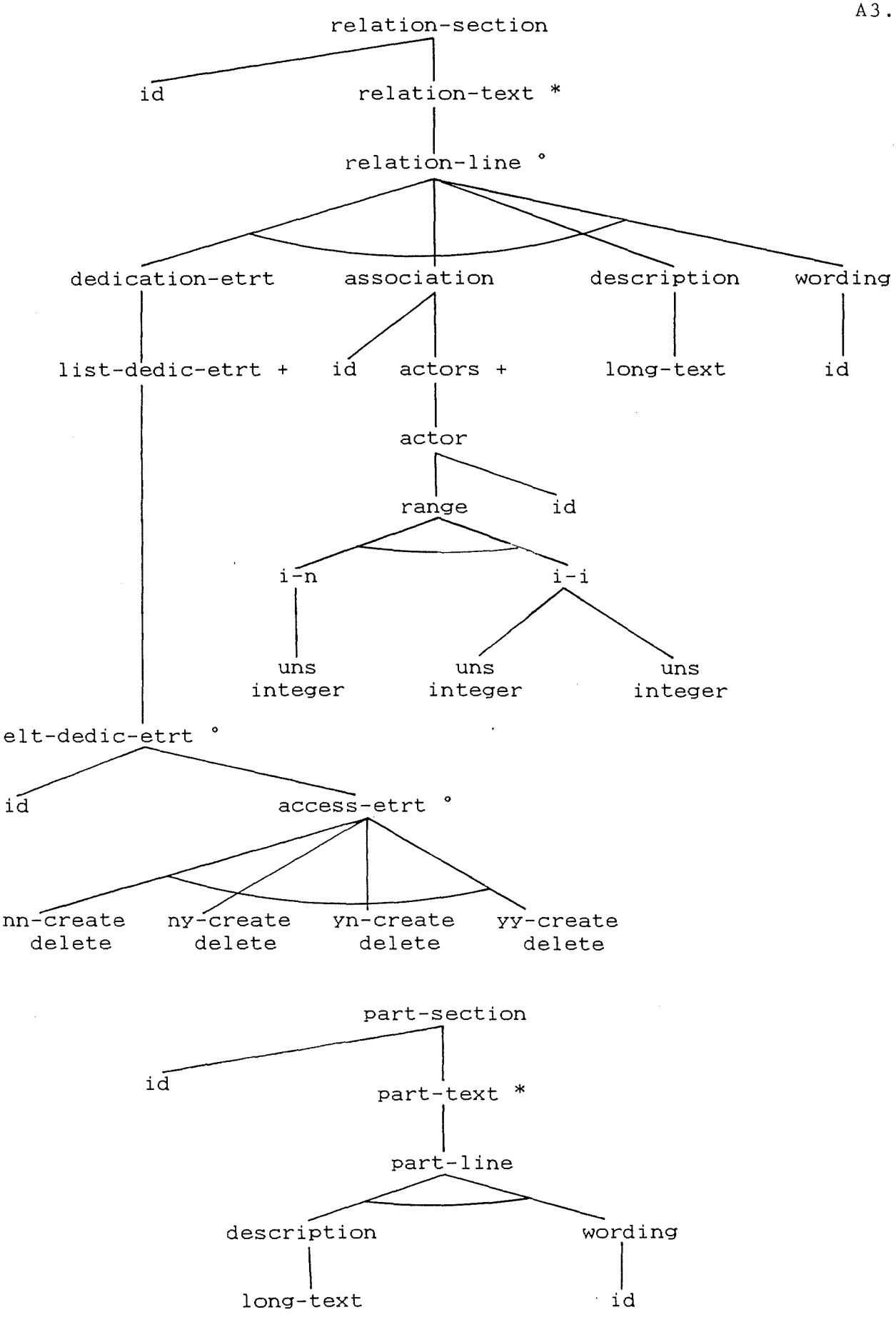
```
<statement_section> ::= "STATEMENT" <id> ";" #
                          <statement_text> ";" ;
L<statement_text> ::=   <statement_line>* ";" % ;
SC<statement_line> ::=  <dedication_updlg> |
                        <syntax>           |
                        <context>          |
                        <description>      |
                        <wording>          ;
<context> ::=           "FOR" <id> ;
<syntax> ::=            "SYNTAX =" <units> ;
L<units> ::=            <unit>+ ;
SC<unit> ::=            <composed_unit> |
                        <simple_unit>   ;
SC<simple_unit> ::=     <keyword> |
                        <id>      ;
<composed_unit> ::=     "(" <simple_units> ":" <keyword> ")" ;
L<simple_units> ::=     <simple_unit>+ ;
<dedication_updlg> ::=  "IS DEDICATED TO" <list_id> ;



<view_section> ::=      "VIEW" <id> ";" #
                          <view_text> ";" ;
L<view_text> ::=        <view_line>* ";" % ;
SC<view_line> ::=       <description> |
                        <wording>     ;



L<list_id> ::=          <id>+ "," % ;


ALT<integer> ::=        <plus_integer> |
                        <minus_integer> |
                        <uns_integer> ;
<minus_integer> ::=     "-"^ <uns_integer> ;
<plus_integer> ::=      "+"^ <uns_integer> ;
GEN<uns_integer> ::=    "[0-9]+" ;


GEN<real> ::=           "([+-]?)([0-9]+)"".""([0-9]+)(""E""([+-]?)([0-9]+))?"


GEN<id> ::=             "[a-z][a-z0-9_\/]*" ;


GEN<keyword> ::=        "'([^\n']|'')*'" ;


GEN<comment> ::=        "#!" "!#" ;


GEN<long_text> ::=      "(/" "COMMENT-LIKE" "/)" ;


GEN<def_lex> ::=        "\""([^\n\""]|\""\"")*\""" .
```

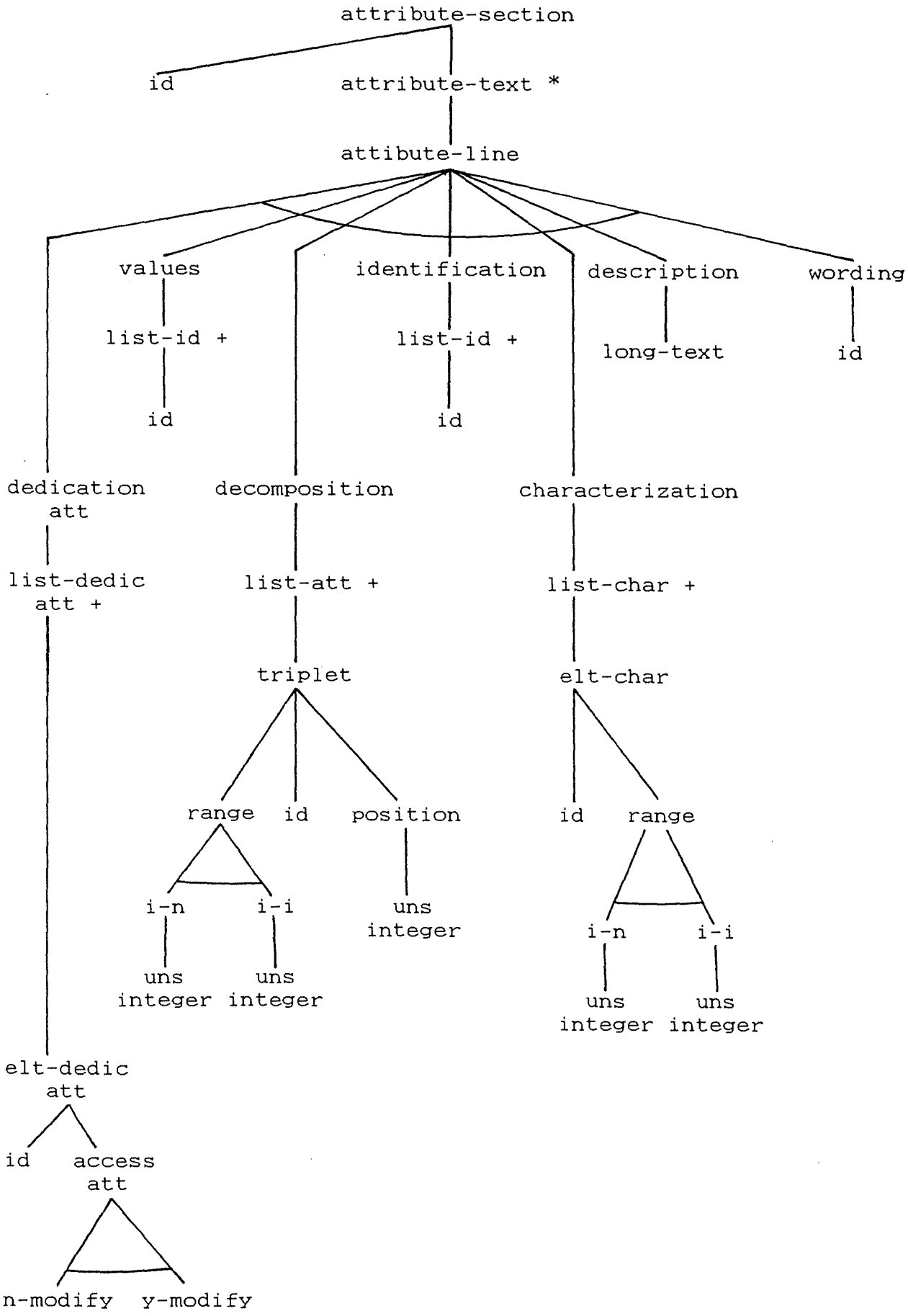
ANNEXE 3

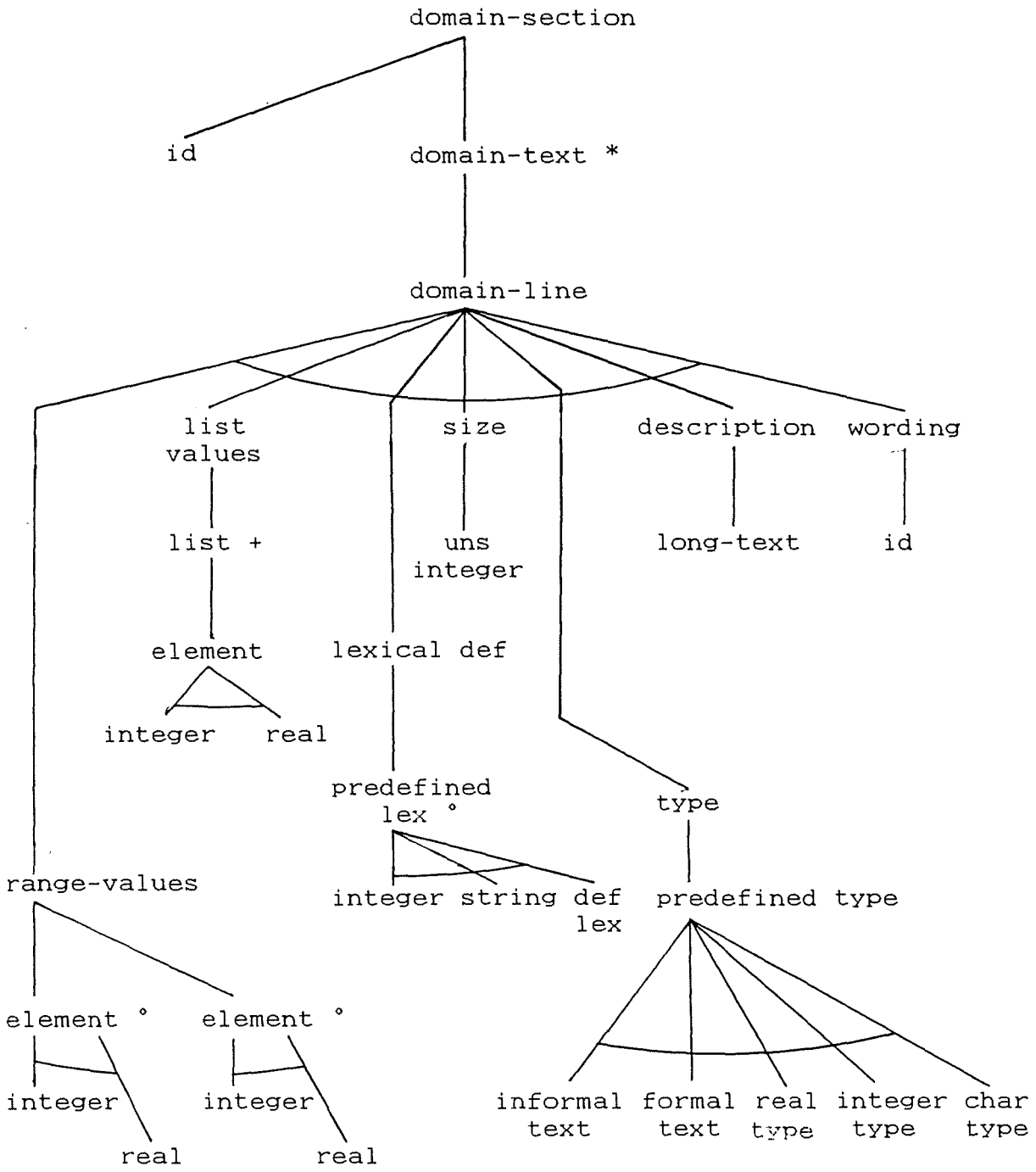
Cette annexe reprend la structure générale d'un arbre correspondant à la définition d'un modèle de cycle de vie. Nous avons choisi les conventions de représentation suivante: chaque opérateur de type liste est suivi du symbole + (si elle doit contenir au moins un opérateur fils) ou du symbole *. Les opérateurs sans constructeur sont suivi du symbole °. Le choix entre plusieurs opérateurs fils est marqué par un arc de cercle.

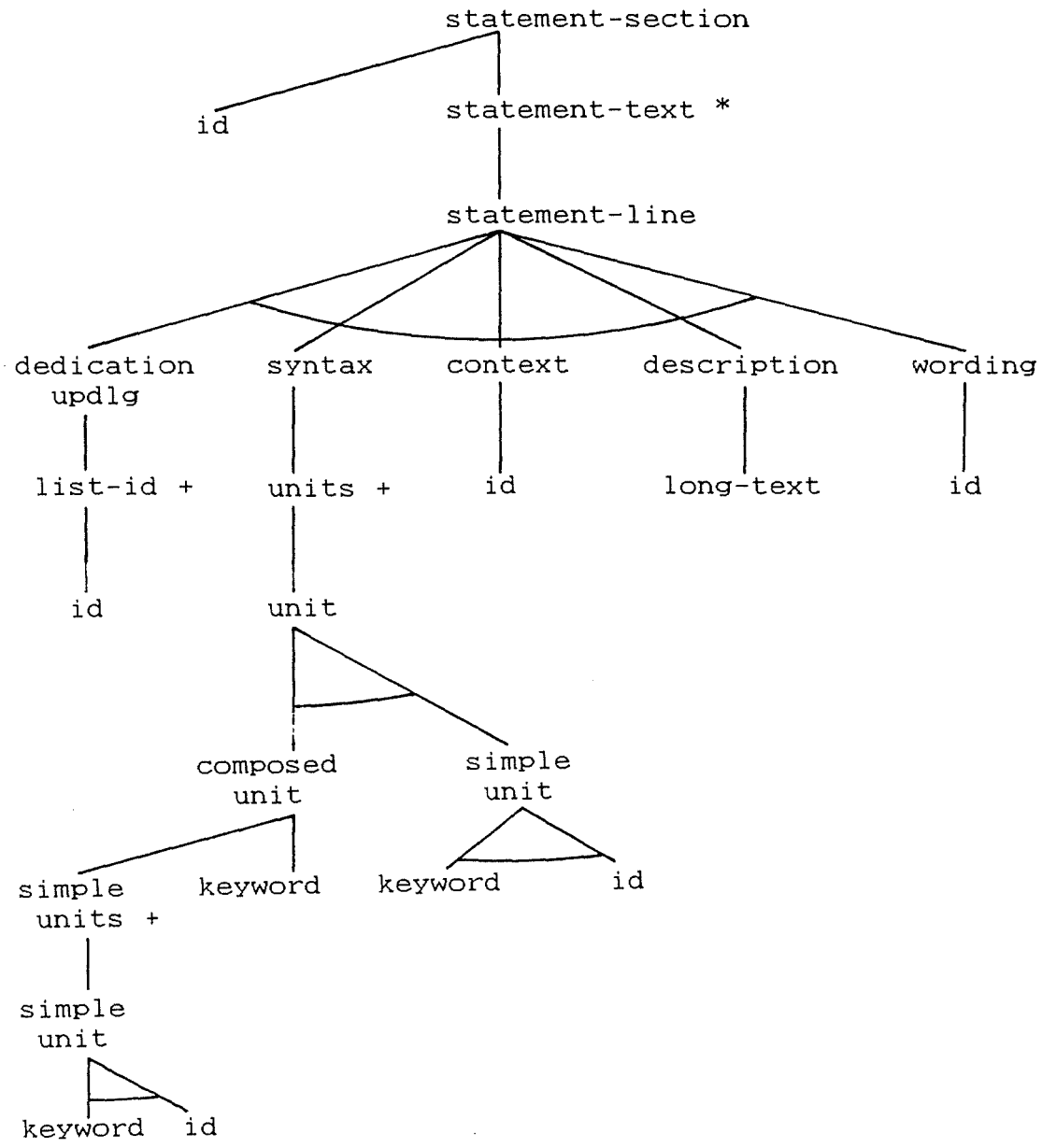

point-d-entrée
model
id
model-text +
model-section °
view section
statement section
domain section
attribute section
part section
relation section
entity section

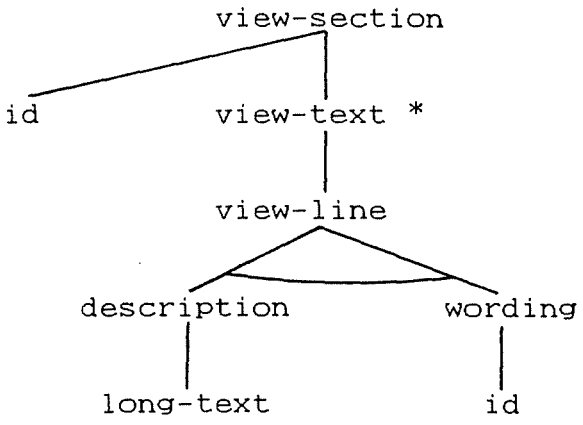

entity-section
id
entity-text *
entity-lines °
dedication etrt
sub-type
lexical def-ident
name id
description
wording
list-dedic etrt +
list-id +
predefined lex
id
long-text
id
id
elt-dedic-etrt
id
access-etrt°
nn-create delete
ny-create delete
yn-create delete
yy-create delete


relation-section
id
relation-text *
relation-line °
dedication-etrt
association
description
wording
list-dedic-etrt +
id
actors +
long-text
id
actor
range
id
i-n
i-i
uns
integer
uns
integer
uns
integer
elt-dedic-etrt °
id
access-etrt °
nn-create
delete
ny-create
delete
yn-create
delete
yy-create
delete
part-section
id
part-text *
part-line
description
wording
long-text
id


attribute-section
id
attribute-text *
attibute-line
values
identification
description
wording
list-id +
list-id +
long-text
id
id
id
dedication att
decomposition
characterization
list-dedic att +
list-att +
list-char +
triplet
elt-char
range
id
position
id
range
i-n
i-i
uns integer
i-n
i-i
uns integer
uns integer
uns integer
uns integer
elt-dedic att
id
access att
n-modify
y-modify


domain-section
id
domain-text *
domain-line
list values
size
description
wording
list +
uns integer
long-text
id
element
lexical def
integer
real
predefined lex °
type
range-values
integer string def lex
predefined type
element °
element °
integer
integer
real
real
informal text
formal text
real type
integer type
char type


statement-section
id
statement-text *
statement-line
dedication updlg
syntax
context
description
wording
list-id +
units +
id
long-text
id
id
unit
composed unit
simple unit
simple units +
keyword
keyword
id
simple unit
keyword
id

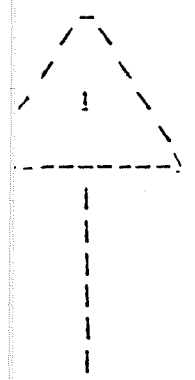

view-section
id
view-text *
view-line
description
wording
long-text
id

ANNEXE 4

Cette annexe reprend la liste des contraintes d'intégrité associées au schéma de définition du méta-modèle (cfr 2° partie 1.3.1.2.)

jouts de meta-entites.

- Entity-type : ajouter un nouveau ENTITY-TYPE.
- Relation-type : ajouter un nouveau RELATION-TYPE.
- Attribute : ajouter un nouvel ATTRIBUTE.
- Part : ajouter un nouveau PART.
- Domain : ajouter un nouveau DOMAIN.
- Statement : ajouter un nouveau STATEMENT.
- View : ajouter une nouvelle VIEW.



Une chaine de caracteres definie comme nouveau KEYWORD, peut avoir ete utilisee comme 'IDENTIFICATEUR' dans la BD_projet pour representer le nom d'une occurrence de meta-entite.

Pour eviter pareil cas,
nous avons adopte la convention suivante :
les KEYWORDS seront automatiquement convertis en MAJUSCULES,
nous conseillons donc d'ecrire les IDENTIFICATEURS en MINUSCULES.

;uts de meta-relations.

- Characterizes : pour un nouvel ATT, un E.T nouveau ou ancien.
  pour un nouvel ATT, un R.T nouveau.
  pour un ancien ATT, un E.T ou un R.T nouveau.

  Rqe : Pour pouvoir ajouter un nouvel ATT
  a un E.T ANCIEN, il doit etre OPTIONNEL.

- Identifies : pour un nouvel ATT, un E.T nouveau.
  pour un ancien ATT, un E.T nouveau.

- Relates : pour un nouveau R.T, un nouveau PART, un E.T nouveau ou ancien.

  Rqe : Pour pouvoir ajouter un nouveau PART dans le cadre d'un nouveau R.T sur un ANCIEN E.T, la connectivite doit etre 0-x.

- Sub_Type : pour un nouveau E.T, un SUR_TYPE nouveau ou ancien.

- Contains : pour un nouvel ATT, un COMPOSANT nouveau.

- Values : pour un nouvel ATT, un DOMAIN nouveau ou ancien.
  pour un ancien ATT, un DOMAIN nouveau.

  Rqe : Pour un ancien ATT, un DOMAIN nouveau, s'il est de type 'char' doit avoir un format <= au plus grand format des domaines existant deja pour cet attribut.

- Context : pour un nouveau STATEMENT, un PART nouveau ou ancien.

- Syntax : pour un nouveau STATEMENT, des PARTS nouveaux ou anciens, des ATTRIBUTES nouveaux ou anciens, des KEYWORDS nouveaux ou anciens.

- Dedication : pour une nouvelle VIEW et un E.T, R.T, ATT, STA nouveau ou ancien.
  pour une ancienne VIEW et un E.T, R.T, ATT, STA nouveau.

outs de meta-proprietes.

- Description : ajouter/modifier pour nouvelle ou ancienne META-ENTITE.
- Name : pour nouvelle META-ENTITE.
- Wording : pour nouvelle ou ancienne META-ENTITE.
- Identifier : pour nouveau ENTITY-TYPE.
- Identifier's lexical definition : pour nouveau ENTITY-TYPE;
- Degree : pour nouveau RELATION-TYPE.
- Type : pour nouveau DOMAIN.
- Size : pour nouveau DOMAIN.
- Def_lex : pour nouveau DOMAIN.
- Value : pour DOMAIN nouveau ou ancien.
- Range : pour DOMAIN nouveau ou ancien.
- Min_o,Max_o : pour nouvelle META-RELATION.

ANNEXE 5

Cette annexe reprend la liste des ajouts autorisés à un modèle de cycle de vie. Ces restrictions ont pour but de garder une base de données projet cohérente et de ne pas y engendrer de changement sémantique (cfr 2° partie 1.3.1.1.).

VERIFICATIONS SEMANTIQUES :

- Pour toute META-ENTITE :

  Verification de l'unicite des NOMS des meta-entites;

  Contrainte de format sur le NOM : maximum 16 caracteres;

- Pour SECTION_VIEW :

  Controle du aractere simple et de la non modification
  de la meta_propriete WORDING;

  Controle de format sur le WORDING : maximum 64 caracteres;

  Controle du caractere simple de la meta_propriete DESCRIPTION;

- Pour SECTION_OBJECT :

  Controle du caractere simple et de la non modification
  de la meta_propriete WORDING;

  Controle de format sur le WORDING : maximum 64 caracteres;

  Controle du caractere simple de la meta_propriete DESCRIPTION;

  Controle de la meta-propriete IDENTIFIER
  la meta-propriete IDENTIFIER est simple, optionnelle et
  non modifiable; sa valeur par defaut est 'name'; elle
  ne peut etre declaree que pour un nouvel entity-type;
  sa valeur doit differer du nom de toutes les meta-
  entites du modele;

  Controle de la meta-propriete IDENTIFIER'S LEXICAL DEFINITION
  la meta-propriete DEF LEX est simple, optionnelle et
  non modifiable; sa valeur par defaut est 'strid';
  elle ne peut etre declaree que si un IDENTIFIER a ete
  declare explicitement et par consequent ne peut etre
  egalement declaree que pour un nouvel entity-type;

  Controle de la meta_relation SUB_TYPE
  IS SUB_TYPE OF nom de entity-type;
  declaree seulement pour un nouvel entity-type;

  Verification de l'absence de circuit dans les declarations de
  meta_relations SUB_TYPE;

  Controle de la meta_relation DEDICATION
  IS DEDICATED TO nom de view;
  declaree seulement pour une nouvelle view et/ou un
  nouvel entity-type;
  ne peut pas etre declare pour un entity-type sous_type;

- Pour SECTION_RELATION :

  Controle du caractere simple et de la non modification de la meta_propriete WORDING;

  Controle de format sur le WORDING : maximum 64 caracteres;

  Controle du caractere simple de la meta_propriete DESCRIPTION;

  Controle de la meta_relation RELATES
  THE PART nom de part IS PLAYED BY nom de entity_type;
  declaree seulement pour un relation-type et un part nouveaux et un entity-type nouveau ou ancien (dans ce cas, la meta_propriete min = 0);
  - un part est associe a 1! relation-type;
  - au moins 2 meta_relations RELATES par relation-type;
  - controle du caractere simple et de la non modification des meta_proprietes min et max;
  - les valeurs des meta_proprietes min et max sont telles que min <= max et que max > 0;

  Controle de la meta_relation DEDICATION
  IS DEDICATED TO nom de view;
  declaree seulement pour une nouvelle view et/ou un nouveau relation-type;
  si un relation-type est dedicace a une view, les entities-types qu'il relie doivent etre dedicaces a cette meme view;

- Pour SECTION_PART :

  Controle du caractere simple et de la non modification de la meta_propriete WORDING;

  Controle de format sur le WORDING : maximum 64 caracteres;

  Controle du caractere simple de la meta_propriete DESCRIPTION;

- Pour SECTION_ATTRIBUTE :

Nom de l'attribut differe de "name" reserve pour l'identifiant;

Controle du caractere simple et de la non modification de la meta_propriete WORDING;

Controle de format sur le WORDING : maximum 64 caracteres;

Controle du caractere simple de la meta_propriete DESCRIPTION;

Controle de la meta_relation DECOMPOSITION
CONTAINS nom d'attribute;
declaree seulement pour des nouveaux attributes;
controle du caractere simple et de la non modification des meta_proprietes min et max; min <= max et max > 0;
verification d'une hierarchie de decomposition avec au maximum 3 niveaux de decomposition;
verification de la nonr-repetitivite des attributs au niveau intermediaire et de l'exclusion de repetitivite entre l'attribut decomposable au niveau superieur et ses composants au niveau inferieur;

Controle de la meta_relation VALUES
declaree seulement pour un nouvel attribute et/ou un domaine nouveau;
tous les domaines d'un attribute sont de meme type;
lors d'une extension au MCV, le format d'un nouveau domaine char defini pour un attribut ne peut etre superieur au plus grand format des domaines deja definis;

Controle de la meta_relation CHARACTERIZES
CHARACTERIZES nom de entity_type ou relation_type;
un nouvel attribute peut caracteriser un entity-type nouveau ou ancien, et un relation-type nouveau;
un ancien attribute peut caracteriser un nouvel entity-type ou un nouveau relation-type;
si l'entity-type caracterise est ancien, min = 0;
controle du caractere simple et de la non modification des meta_proprietes min et max; min <= max et max < 0;

Controle de la meta_relation IDENTIFIES
IDENTIFIES nom de entity_type;
declaree seulement avec un entity_type nouveau;

- un attribute composant ne peut pas caracteriser;
- un attribute non composant doit caracteriser;

- un attribute qui se decompose n'a pas de domain;
- un attribute qui ne se decompose pas doit avoir un domain;

- un attribute qui se decompose ne peut pas caracteriser un un relation-type;

- un attribut composant ne peut avoir un domaine texte;

- un attribute qui identifie un entity_type doit le caracteriser de maniere obligatoire;

Controle de la meta_relation DEDICATION
IS DEDICATED TO nom de view;
declaree seulement pour une nouvelle view et/ou un nouvel attribute;
ne peut pas etre declaree pour un attribute composant;
si un attribute est dedicace a une view, les entities-types et les relations-types qu'il caracterise doivent etre dedicaces a cette meme view;

- Pour SECTION_DOMAIN :

Controle du caractere simple et de la non modification de la meta_propriete WORDING;

Controle de format sur le WORDING : maximum 64 caracteres;

Controle du caractere simple de la meta_propriete DESCRIPTION;

Controle du caractere simple, obligatoire (lors de la le declaration du domaine) et de la non modification de la meta_propriete TYPE;

Les meta_proprietes SIZE et DEF_LEX sont simples, obligatoires (lors de la le declaration) et non modifiables si le type du domaine est 'caractere'; la valeur de SIZE <= 200;

Pas de meta-proprietes SIZE et DEF-LEX si le type est 'entier', 'reel', 'texte libre' ou 'texte formel';

Pas de meta_proprietes LIST et RANGE (min,max) si le type est 'caractere', 'texte libre' ou 'texte formel';

Compatibilite des elements de la liste et des ranges avec le type du domaine;

Pour la meta_propriete range (min,max), min <= max;

Contraintes de format sur les definitions lexicales, les entiers, les reels et les strings;

- Pour SECTION_STATEMENT :

Controle du caractere simple et de la non modification de la meta_propriete WORDING;

Controle de format sur le WORDING : maximum 64 caracteres;

Controle du caractere simple de la meta_propriete DESCRIPTION;

Controle de la meta_relation CONTEXT
FOR nom de part;
declaree seulement pour un nouveau statement;
1! meta_relation CONTEXT par statement;

Un statement definit completement un relation-type;

Minimum un keyword par statement;

Un statement ne peut contenir 2 keywords se succedant;

Maximum une unite composee par statement;

1! part dans une unite composee;

Un seul niveau de decomposition dans unite composee;

Si il y a une unite composee dans un statement, elle doit contenir toutes les proprietes du relation-type defini;

Pour un meme entity-type contexte, non synonymie des statements;

ANNEXE 6

Cette annexe reprend l'ensemble des contraintes sémantiques davant être vérifiées par un arbre d'extension à un modèle de cycle de vie. Ces contraintes portent sur les contraintes d'intégrité du méta-modèle ainsi que sur le concept d'extension (cfr 2) partie 1.3.1.).

POUR TOUTE META-ENTITE c'est-a-dire POUR TOUT ENTITY-TYPE,
RELATION-TYPE,
ATTRIBUTE,
PART,
DOMAIN,
STATEMENT,
VIEW
DU MODELE DE CYCLE DE VIE DECRIT :

- 1-1 NAME : maximum 16 caracteres
  UNIVOQUE dans le modele de cycle de vie decrit
- 0-1 WORDING : maximum 64 caracteres
  non modifiable
- 0-1 DESCRIPTION : texte informel (/ ..... /) modifiable

META-ENTITE ENTITY-TYPE :

- 0-1 IDENTIFIER (= intitule de l'identifiant des occurrences de l'entity-type au niveau de la dd-projet)
  : non modifiable
  different des noms des autres concepts du MCV
  valeur par defaut "name"
- 0-1 IDENTIFIER'S LEXICAL DEFINITION : non modifiable
  valeur par defaut "strio"
- pour pouvoir declarer une DEFINITION LEXICALE de l'identifiant, il faut avoir declare un INTITULE particulier pour cet identifiant

META-RELATION SUB-TYPE :

- pas de CIRCUIT dans les declarations de sous-types

META-RELATION RELATES :

- un PART est associe a un et un seul RELATION-TYPE
- min <= max et max > 0

META-ENTITE ATTRIBUTE :

- le nom de l'attribut ne peut etre "name", reserve comme identifiant

META-RELATION CHARACTERIZES :

- min <= max et max > 0

META-RELATION IDENTIFIES :

- un attribut qui identifie un entity-type doit le CARACTERISER de maniere OBLIGATOIRE (c'est-a-dire min >= 1)

META-RELATION CONTAINS :

- min <= max et max > 0
- les POSITIONS dans la decomposition d'un attribut doivent former une sequence commencant a 1
- HIERARCHIE de decomposition avec maximum n NIVEAUX
- un attribut de niveau intermediaire ne peut etre repetitif
- si l'attribut decomposable au niveau superieur est repetitif, les attributs qui le composent aux niveaux inferieurs ne peuvent etre repetitifs
- un attribut qui compose ne peut caracteriser
- un attribut qui ne compose pas doit caracteriser
- un attribut decomposable ne peut caracteriser un relation-type

ETA-ENTITE DOMAIN :

- 1-1 TYPE : (integer, real, char, formal text, informal text)
  non modifiable

- 0-1 SIZE : ne peut exister si type n'est pas char
  obligatoire si type est char
  valeur <= 200
  non modifiable

- 0-1 DEF-LEX : ne peut exister si type n'est pas char
  optionnelle si type char (valeur par defaut "strid")
  non modifiable

- 0-N LIST : meta-propriete a valeurs repetitives
  ne peut exister que si type est integer ou real
  les valeurs doivent etre du type du domaine

- 0-N RANGE (MIN,MAX) : ne peut exister que si type est integer ou real
  les valeurs doivent etre du type du domaine
  min <= max

ETA-RELATION VALUES :

- tous les DOMAINES d'un attribut sont de MEME TYPE

- un attribut decomposable ne peut avoir de domaine

- un attribut non decomposable doit avoir un domaine

- un attribut qui en compose un autre ne peut avoir un domaine de type texte (formal text ou informal text)

ETA-ENTITE STATEMENT :

- une phrase definit COMPLETEMENT un relation-type

- minimum un keyword par phrase

- pas 2 keywords consecutifs

- maximum une UNITE COMPOSEE REPETITIVE par phrase

- un et un seul part par unite composee repetitive

- s'il y a dans la phrase une unite composee repetitive, tous les attributs definis dans la phrase doivent en faire partie

- pas de phrases SYNONYMES pour un meme entity-type

ETA-RELATION DEDICATION :

- si un relation-type est dedicace a une vue, les entity-types qui participent a ce relation-type doivent etre dedicaces a cette meme vue

- si un attribut est dedicace a une vue, les entity-types et/ou les relation-types qu'il caracterise doivent etre dedicaces a cette meme vue

- si un statement est dedicace a une vue, le relation-type qu'il exprime doit etre dedicace a cette meme vue

- un statement ne peut etre dedicace qu'au type particulier de vue 'upd_lg?' (vue-utilisateur de langage de mise a jour)

- un attribut composant ne peut etre dedicace, etant donne qu'il est l'objet de dedicaces heritees de son pere

- un entity-type sous-type ne peut etre dedicace, etant donne qu'il est l'objet de dedicaces heritees de ses peres

- un access_mode doit etre defini pour toute dedicace, sauf pour les dedicaces de phrases.
  valeurs pour un type d'objet ou de relation (creable_deletable, creable_nondeletable, noncreable_deletable, noncreable_nondeletable)