MASTER EN SCIENCES INFORMATIQUES

Étude et évaluation d'un outil de prototypage

Langelez, Fabienne; Paris, Cécile

Award date:
1986

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix (Namur)

Institut d'informatique

ETUDE ET EVALUATION D'UN OUTIL DE PROTOTYPAGE

Mémoire présenté par

Fabienne LANGELEZ

Cécile PARIS

en vue de l'obtention

du titre de

Licencié et Maitre en Informatique

Année académique 1985-1986

REMERCIEMENTS

Nous tenons à remercier pour l'aide qu'ils nous ont apportée dans la réalisation de notre travail, Mr. A. van Lamsweerde, promoteur du mémoire, Mrs. Nisole, Poels et Lardinois pour le concours qu'ils ont apporté lors de notre stage au C.I.G. (Bruxelles), Mr. N. Habra pour sa collaboration constructive à notre travail, Melle M.-C. Schayes, Mrs. B. Delcourt et J. Selderslacht pour leur aide dans l'utilisation des outils d'ALMA et du système, Mrs Y. Deville et J.-M. Jacquet pour leurs critiques judicieuses.

TABLE DES MATIERES

INTRODUCTION

Il est généralement reconnu que les coûts de développement d'un gros système informatique sont très élevés, et que la spécification des besoins constitue une étape critique dans ce développement. Ecrire de bonnes spécifications (complètes, cohérentes, minimales, ...) est très difficile, et les erreurs commises à ce niveau peuvent être lourdes de conséquences : plus elles sont détectées tard, plus les corrections sont coûteuses. D'où la nécessité de disposer de méthodologies rigoureuses pour améliorer la fiabilité des systèmes informatiques, en particulier des méthodes d'aide à la spécification et à la validation des spécifications.

Dans ce contexte, la production d'un prototype, c'est-à-dire une version immature d'un système informatique, s'avère très utile. Cette version ne présente généralement pas toutes les caractéristiques du système final, mais elle possède l'avantage d'être rapidement productible. Présenté au client, un prototype permet d'évaluer le plus tôt possible, l'adéquation entre les spécifications et les besoins effectifs du client, afin de provoquer un éventuel feedback dans les délais les plus brefs. Il est donc essentiel qu'un prototype soit une approximation fidèle du système final, qu'il soit rapidement disponible et puisse être construit au moindre coût. Le développement d'outils de prototypage rapide ("rapid prototyping") fait actuellement l'objet de nombreuses recherches. Diverses méthodes sont proposées telles que l'utilisation de spécifications "exécutables" et la génération

automatique de programmes. Le contexte dans lequel s'inscrit le prototypage, ainsi que des méthodes possibles seront abordés dans le chapitre 1.

L'outil de prototypage qui fait l'objet de notre travail relève de la dernière approche (génération automatique de programmes). Il consiste en effet à produire semi-automatiquement un programme, dans un langage de haut niveau, à partir d'une spécification exprimée dans un langage de spécification formel. Cet outil repose donc sur les éléments suivants :

(1) un langage de spécification : le langage RSL (Requirement Structuring Language), actuellement en cours de développement [DUBOIS 85], [VLA 86]. La spécification d'un système, en RSL, est basée sur deux modèles complémentaires : l'un pour décrire les objets du système, l'autre pour décrire les opérations du système. RSL est en fait plus qu'un langage. C'est le support d'une méthode de spécification reposant sur une composition de différentes stratégies pour structurer les spécifications. Le langage, les modèles sous-jacents, et les stratégies seront présentés dans le point 1 du chapitre 2.

(2) un langage de programmation : le langage PROLOG, utilisé pour résoudre des problèmes impliquant des objets et des relations entre ces objets [CLOCKSIN 81]. Vu sous un angle idéalisé, un programme PROLOG se présente comme un ensemble de faits et de relations au sujet d'un problème, plutôt qu'une séquence d'étapes à accomplir pour résoudre ce problème. Ce langage sera exposé au point 2 du chapitre 2.

(3) des règles de traduction permettant de passer d'un texte RSL à un programme PROLOG correspondant. Ces règles, amorcées au cours de travaux parallèles [HABRA 86], ont été complétées en fonction de nos besoins. Elles seront présentées dans le point 3.1 du chapitre 2.

(4) un programme automatisant partiellement le processus de traduction et utilisant des outils développés dans le cadre du projet ALMA (Atelier Logiciel sur Machine Abstraite) [VLA 86b]. Ces outils sont génériques et permettent la manipulation d'arbres syntaxiques abstraits. Paramétré sur un langage déterminé, l'analyseur construit, à partir d'un texte exprimé dans ce langage, l'arbre correspondant , tandis que le décompilateur effectue l'opération inverse. L'éditeur permet des transformations d'arbres. Une "boite à outils" contenant un ensemble de primitives de manipulation d'arbres est également disponible.

La traduction semi-automatique est constituée des étapes suivantes :

- construction, par l'analyseur syntaxique d'ALMA, de l'arbre abstrait correspondant au texte RSL,
- création, par notre traducteur, de l'arbre abstrait PROLOG, à partir de l'arbre abstrait RSL, selon les règles de traduction mentionnées,
- transformation, par le décompilateur d'ALMA, de l'arbre abstrait PROLOG en un programme PROLOG adéquat :

Dans le cadre de ce mémoire, seules quelques règles de transformation ont été automatisées, l'objectif premier étant de tester la faisabilité de l'outil. L'automatisation partielle du passage de RSL à PROLOG sera décrite dans le point 3.2 du chapitre 2 (outils utilisés, principes et état actuel du programme de traduction)

Lors de notre stage au C.I.G. (Bruxelles), nous avons expérimenté le langage et la méthode de spécification RSL sur une application de gestion en vraie grandeur (l'application "Ventes" d'une cimenterie, constituée de la gestion du fichier des tiers, l'enregistrement de contrats, et de livraisons, la facturation, ainsi que diverses statistiques). Cette application, lourde et complexe, nous a permis de mettre en évidence les qualités et les défauts de RSL. Ils seront détaillés dans le point 1 du chapitre 3.

Une partie de la spécification (gestion du fichier des tiers, des contrats et enregistrement des livraisons) a été transformée en un programme PROLOG, par application manuelle des règles de traduction. Ce travail, assez systématique, a

fait ressortir la possibilité d'automatisation - partielle - de cette étape. L'automatisation complète est en effet impossible car certaines choses, exprimables en RSL, ne le sont pas directement en PROLOG. L'intervention de l'utilisateur de l'outil sera parfois nécessaire. L'évaluation de PROLOG dans le cadre du prototypage fera l'objet du point 2 du chapitre 3. Quant à l'outil de prototypage proprement dit, il sera évalué dans le point 3 du chapitre 3. Nous y aborderons l'adéquation des outils d'ALMA, les qualités et les défauts des règles de transformation, ainsi que les limites du programme de traduction RSL-PROLOG. Nous proposerons également quelques perspectives d'amélioration.

CHAPITRE 1 : LE PROTOTYPAGE

1 Une définition

Le prototypage a pour objectif la préparation rapide de versions "immatures" d'un système informatique, autrement dit de prototypes. Ces versions sont qualifiées d'immatures car elles ne présentent généralement pas toutes les caractéristiques du système final. Elles servent à introduire un élément de communication entre les futurs utilisateurs et les réalisateurs du système, permettant ainsi un feedback dans le processus de construction des spécifications.

2 Utilité du prototypage

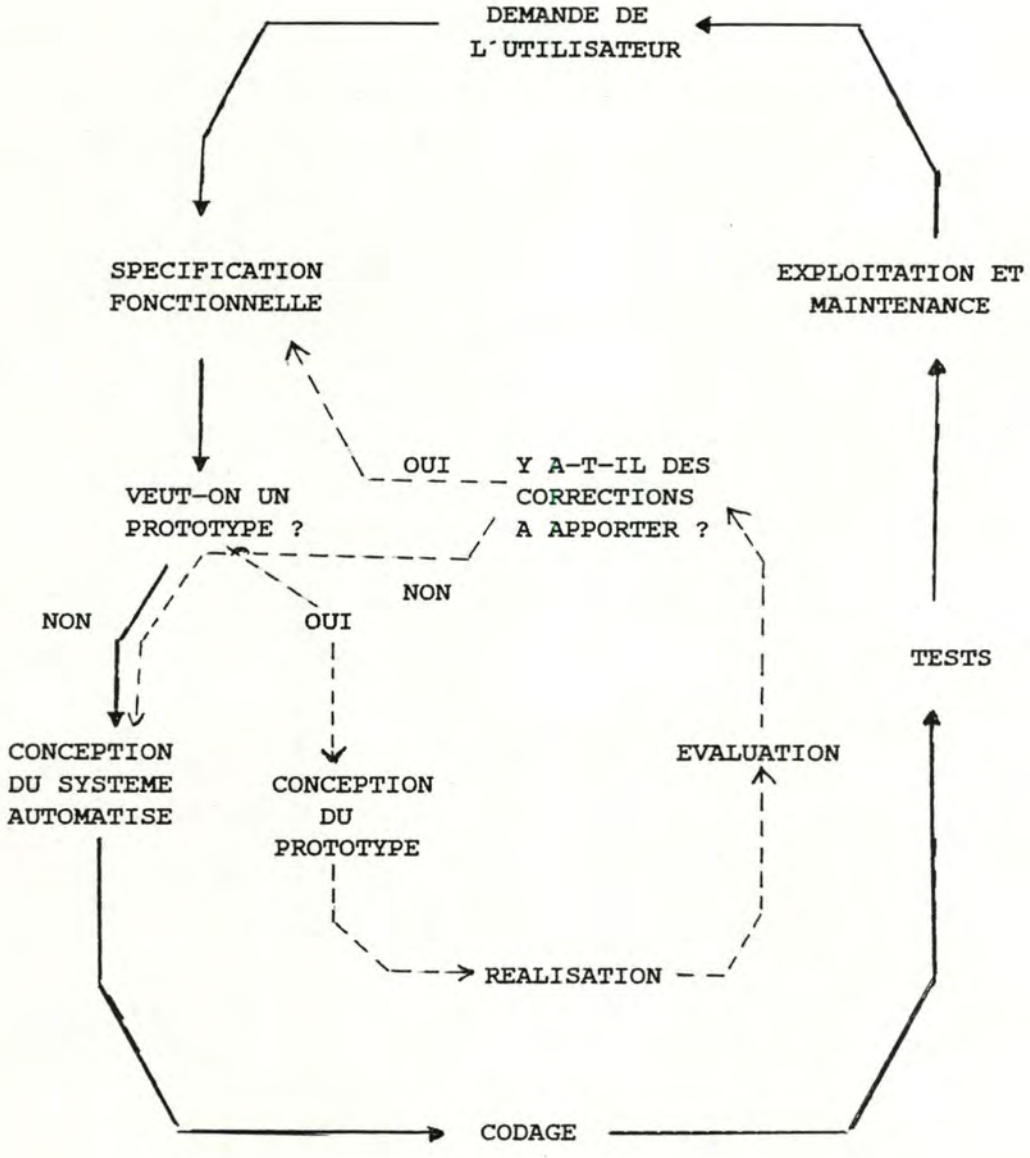
La spécification des besoins constitue une étape critique dans le processus de développement d'un système informatique, d'une part, parce qu'il est très difficile d'écrire une bonne spécification, c'est-à-dire une spécification adéquate par rapport aux besoins, complète, cohérente, minimale, ... et d'autre part, parce que le coût de correction d'une erreur de spécification détectée durant la maintenance, est, selon certaines estimations, de l'ordre de 100 fois le coût de correction de la même erreur lors de l'étape de spécification elle-même [BOEHM 82]. Il est donc important de disposer de méthodologies rigoureuses afin d'améliorer la qualité des spécifications des besoins.

De nombreux outils d'aide au développement de logiciels existent déjà [VLA 82] , portant sur les différentes étapes du cycle de vie d'un projet informatique. Le prototypage constitue une approche particulièrement pertinente dans le cadre de l'étape de spécification. On peut en effet envisager diverses utilisations d'un prototype à ce niveau :

- aide à la construction de la spécification : après avoir "grossièrement" déterminé les besoins, on développe un prototype que l'on présente à l'utilisateur afin d'affiner et/ou de compléter les spécifications,

- validation de la spécification : après avoir complètement et soigneusement défini les besoins, on produit un prototype que l'on soumet à l'utilisateur pour évaluation, de façon à s'assurer qu'il y a bien correspondance entre les besoins identifiés et les besoins réels de l'utilisateur.

Le prototype permet donc d'avoir un feedback plus tôt dans le processus de développement d'un système informatique. Celui-ci peut alors se schématiser comme suit :



En général, le prototypage s'avère utile chaque fois que les besoins de l'utilisateur sont incertains. Cette incertitude peut provenir d'un manque de connaissance de la part de l'utilisateur quant aux possibilités offertes par l'automatisation , ou de la part des informaticiens au sujet du domaine d'application.

L'incertitude peut aussi provenir d'une incapacité de l'utilisateur à exprimer ce qu'il veut réellement, ou encore d'une incapacité à prédire le futur, c'est-à-dire la façon dont l'automatisation influencera la manière de travailler des utilisateurs (de nouveaux besoins surgiront...). Un prototype favorise la collaboration entre informaticiens (experts en automatisation) et utilisateurs (experts dans le domaine d'application), ce qui devrait conduire à une plus grande qualité du système final.

On distingue généralement quatre étapes dans le prototypage : la sélection fonctionnelle, la construction proprement dite, l'évaluation et l'usage ultérieur.

- La sélection fonctionnelle est le choix des fonctions que le prototype réalisera. L'étendue fonctionnelle du prototype peut être différente de celle du produit final, seules certaines fonctions considérées comme cruciales étant implémentées. D'autre part, les fonctions sélectionnées peuvent être implémentées sous leur forme finale, ou sous une forme moins détaillée (avec une simulation de certaines parties, ...).

  On distingue parfois deux types de prototypage : le prototypage d'interface dont le but est de tester (ou affiner) les spécifications de l'interface homme/machine, et le prototypage fonctionnel qui, lui, a pour objectif de tester (ou affiner) les spécifications fonctionnelles. L'outil faisant l'objet de notre travail est du type

prototypage fonctionnel.

- La construction fait référence à l'élaboration du prototype. Dans cette construction, l'accent devrait être mis sur l'évaluation voulue, et pas sur l'utilisation régulière et à long terme du prototype. C'est pourquoi des aspects techniques d'implémentation peuvent être négligés, comme les bonnes performances, la portabilité, les recouvrements d'erreur, les droits d'accès, ... importants pour le système final, mais coûteux ou impossibles à implémenter rapidement.

- L' évaluation est l'étape décisive du prototypage puisqu'elle permet au client de s'exprimer, et ainsi d'obtenir plus tôt un feedback dans le processus de développement.

- Les possibilités d' usage ultérieur du prototype, après son évaluation, dépendent de la méthode de prototypage choisie, ainsi que des outils disponibles. Le prototype peut être "jeté" ou incorporé dans le produit final . Une autre possibilité est la transformation automatique du prototype dans le produit final.

3 Danger du prototypage

Le danger du prototypage dans l'étape de spécification est qu'il conduise à un manque de soins dans le processus d'élaboration des spécifications. Le prototypage risque en effet d'être utilisé au détriment d'une analyse rigoureuse et

structurée, et ainsi donner lieu à un cercle vicieux d'erreurs et d'essais. Les utilisateurs se lasseront s'ils doivent évaluer trop de prototypes différents. Le prototypage n'est qu'un outil. Il ne peut en rien se substituer à une méthode rigoureuse d'élaboration et de formulation des spécifications.

De plus, l'activité de prototypage ne peut remplacer l'activité de spécification (un prototype n'est pas une spécification). Les raisons principales en sont les suivantes :

- la spécification doit être une description complète et cohérente de ce que le système doit faire, alors que le prototype n'en est souvent qu'une approximation. Le prototype est donc dépendant d'une mise en oeuvre, contrairement à la spécification,
- pour ces raisons, la spécification peut être à la base du contrat liant l'utilisateur et le réalisateur du futur système, tandis que le prototype est évidemment insuffisant.

4 Qualités d'un prototype

Un bon prototype devrait présenter les caractéristiques suivantes :

- Aptitude à l'évaluation : le prototype doit être une approximation fidèle du futur système, au point de vue des fonctionnalités dans le cas du prototypage fonctionnel, et au point de vue des interfaces dans le cas du prototypage

d'interface. Ceci afin que l'évaluation soit pertinente.

- Faible coût de production : le coût supplémentaire dû au prototypage doit être réduit, comparé au coût total. Remarquons cependant que le prototypage n'augmente pas nécessairement le coût global du système, puisque l'investissement supplémentaire aide à réduire la probabilité d'erreurs fondamentales et coûteuses.

- Disponibilité rapide. Il faut en effet que les parties concernées (développeurs et utilisateurs) en tirent un maximum de profit. Le prototype doit être obtenu rapidement, mais d'une façon rationnelle et contrôlée. Pour cela, une méthode sérieuse et (partiellement) automatisée est nécessaire.

- Aptitude à être montré. Le prototype doit pouvoir être montré aux utilisateurs pour évaluation, et donc présenter un minimum de convivialité.

- Modifiabilité : un prototype doit être facilement modifiable pour permettre des cycles d'évaluation sur un même prototype.

- Aptitude à l'apprentissage : après une évaluation positive, un prototype peut servir d'environnement d'apprentissage pour préparer les futurs utilisateurs au système final.

- Réutilisable : selon certains, il faudrait avoir la possibilité de réutiliser tout ou partie du prototype dans la version finale du système, afin de profiter au maximum de

l'effort consenti pour sa réalisation. Mais pour d'autres, un prototype doit être jeté car, à se concentrer sur la possibilité de réutilisation du prototype, on risque de négliger la construction des spécifications, d'allonger le délai d'obtention du prototype, et d'augmenter son coût.

5 Approches de prototypage

Le prototypage devenant populaire, il existe désormais plusieurs méthodes et outils de prototypage. Les trois méthodes évoquées le plus souvent dans la littérature sont les suivantes :

- l'utilisation d'un langage de programmation de haut niveau pour écrire le prototype : le prototype est écrit dans un langage facilitant l'expression (langages fonctionnels, logiques ou orientés objets tels que LISP, PROLOG, APL, SMALLTALK ...). Ces méthodes "manuelles" peuvent nécessiter un long délai avant que le prototype ne soit disponible.

- la rédaction de spécifications exécutables : le système est décrit dans un langage de spécification formel, ayant une sémantique opérationnelle, de sorte qu'il peut être directement exécuté (ou interprété) d'une certaine façon (notamment par une exécution symbolique). L'avantage de ces méthodes est que le prototype ne doit pas nécessairement être construit. Cependant, elles présentent un risque de confusion entre spécification, c'est-à-dire le problème à résoudre, et code exécutable, c'est-à-dire une solution à ce

problème.

- la dérivation automatique du prototype. Cette méthode consiste, à partir de la spécification du système, à produire automatiquement ou semi-automatiquement, le prototype correspondant. L'outil que nous présentons relève de cette approche.

CHAPITRE 2 : UN ENVIRONNEMENT DE PROTOTYPAGE

La démarche de prototypage suivie est la suivante :

Partant des spécifications de l'application écrites dans un langage de spécification RSL (Requirements Structuring Language), le programme exécutable obtenu est écrit en Prolog en appliquant des règles de transformations de façon systématique.

Ce chapitre suit la démarche et se décompose en 3 points. Dans un premier temps, nous exposerons le langage RSL. Ensuite, nous présenterons le langage Prolog. Nous terminerons ce chapitre par une présentation des règles de transformation utilisées et de l'outil que nous avons développé afin d'automatiser ces règles.

1 RSL : un langage de spécification

Nous nous limiterons à une présentation générale du langage. Pour plus de détails, on peut consulter [DUBOIS 84] ainsi que les articles [DUBOIS 85] et [DUBOIS 86].
Une évaluation de RSL en tant que langage de spécification sera développée dans le chapitre 4.

Notation : |-> dénote l'opérateur d'explicitation.
Les mots en majuscules dénotent des types et les mots en minuscules des objets (i.e les occurences du type).

1.1 Introduction

RSL comporte un langage ainsi qu'une méthodologie de spécification.

L'objectif du processus de spécification est d'obtenir d'une part, la spécification de l'énoncé du problème (c'est-à-dire la spécification des différentes fonctions devant être remplies par le système) et d'autre part, la spécification de l'univers du problème (c'est-à-dire la spécification des différents types de données manipulés). Une forte interaction existe entre les deux spécifications.

Le processus de spécification est guidé par un méta-algorithme que peut appliquer le spécifieur. Il repose sur l'emploi de 2 modèles complémentaires : l'un pour spécifier les objets et l'autre pour spécifier les opérations. Ces modèles sont structurés en fonction des relations existant entre les composantes de la spécification. Les liens entre les opérations peuvent être définis au moyen des connecteurs AND, OR, FORALL et WITH. Les liens entre un type et des types plus élémentaires s'expriment à l'aide des constructeurs de type (produit cartésien, union disjointe, suite, ensemble, table). Ces relations seront illustrées dans le point suivant. On suppose aussi que l'on dispose déjà d'un certain nombre d'opérations et de types de base pré-spécifiés une fois pour toutes. Cet ensemble est appelé "bibliothèque de spécification", et est extensible en fonction des besoins.

Les résultats du processus de spécification sont consignés dans des documents appelés "blocs de spécification". On établira un bloc par type d'objet de l'univers du problème et un bloc par opération de l'énoncé du problème. L'assemblage de ces blocs constitue la spécification complète du système ; une version synthétique en est également fournie sous forme de graphiques représentant les 2 forêts, celle de l'énoncé et celle de l'univers. La structure des blocs et celle des forêts seront développées au point suivant (1.2.5).

Il est assez difficile de déterminer tous les arguments sans une explicitation du résultat et de l'opération associée. Aussi pour indiquer ce fait durant le processus de spécification, nous prendrons la convention suivante : res = OP(arg, ...).

Terminons cette introduction en soulignant le rapport qui existe avec les types abstraits. Les constructeurs de type sont des types abstraits paramétrés dont les opérations associées sont définies algébriquement dans la bibliothèque de spécification.
De même, l'univers du problème constitue une architecture de types abstraits dont les opérations associées sont définies dans l'énoncé du problème, leur structure l'étant dans l'univers.
Une présentation des types abstraits spécifiés algébriquement est développée dans l'article [GOGUEN 78].

1.2 Les modèles utilisés et la bibliothèque de spécification

Le processus d'élaboration des 2 modèles repose sur différents mécanismes d'abstraction :

- le mécanisme de spécialisation de types permet de caractériser un sous-type d'objet d'un type donné. Il est l'inverse du mécanisme de généralisation.

  Il est représenté par l'opérateur IS-A.

  Il permet de tenir compte des caractéristiques individuelles des sous-classes de données. Le sous-type défini hérite par défaut de toutes les propriétés du type donné (c'est-à-dire les opérations associées, les invariants et la structure) sauf si la décision a été prise d'inhiber ce mécanisme d'héritage. Par exemple, on peut définir un type CLIENT qui est un tiers de la société.

  CLIENT |-> IS-A [TIERS] without EDIT-F-FOURN

  où CLIENT est le sous-type défini,

  TIERS est le type de plus haut niveau,

  EDIT-F-FOURN est l'opération d'édition des fiches fournisseurs.

  En outre, on peut associer au sous-type de nouveaux invariants et opérations. Seule la structure ne peut être modifiée.

  Le mécanisme de spécialisation permet ainsi d'émettre des restrictions en ajoutant un ou plusieurs invariants au sous-type. Par exemple, on définira un client comme un tiers ayant passé une commande.

- le mécanisme de décomposition de types ou d'opérations permet de caractériser les composantes d'un type ou d'une opération donné(e). Il est l'inverse du mécanisme d'agrégation.

  Deux types de décomposition sont à distinguer. La décomposition structurelle met l'accent sur la décomposition des types tandis que la décomposition fonctionnelle le met sur la décomposition des opérations. Au niveau des opérations, la décomposition structurelle définit par exemple une conjonction d'équations exprimant la relation entre le résultat des opérations et leurs arguments (décomposition-AND et décomposition-FORALL) ou une disjonction d'équations (décomposition-OR). Au niveau des types, elle définit un n-uple de types (décomposition-AND), une union disjointe de types (décomposition-OR), une suite ou un ensemble de types (décomposition-FORALL). Notons que chacun des types d'une union disjointe est une spécialisation.

  La décomposition fonctionnelle définit une composition fonctionnelle au niveau des opérations (décomposition-WITH) ou une dépendance fonctionnelle entre 2 types un niveau des objets (décomposition-WITH).

- le mécanisme de classification : il s'agit de regrouper des objets considérés individuellement dans une classe. Les propriétés définies sur une classe sont partagées par tous les éléments de la classe. Chaque fois qu'un objet apparait, on lui donne un type.

  Exemple : secrétaire : EMPLOYE.

Il existe d'autres mécanismes tels que l'instanciation/paramétrisation ou l'extension/restriction.
Pour plus de détails concernant ces mécanismes, on consultera les articles [DUBOIS 85] et [DUBOIS 86].

1.2.1 Modèle des opérations : l'énoncé du problème

Les opérations sont définies comme des relations mathématiques. Ces opérations, à l'exception de celles associées aux types de données de base (opérations de la bibliothèque), sont spécifiées par leurs domaines, leur codomaine et 2 types d'assertions :

- une précondition, facultative, qui restreint le domaine de l'opération. Cette condition doit être satisfaite par l'opération pour que celle-ci soit définie. Elle se présente sous la forme d'un prédicat du 1er ordre.

- une assertion explicitant la relation existant entre le résultat de l'opération et ses arguments, sous forme d'équations où interviennent d'autres opérations devant à leur tour être spécifiées et/ou des opérations pré-spécifiées de la bibliothèque. L'ensemble de ces équations est structuré au moyen de connecteurs (and, or, forall, with) en fonction du mécanisme de décomposition utilisé.

Exemples :

* pour une décomposition-forall

```
res-majs-tiers  =  MAJS-F-TIERS  (fich-tiers,
                                  ddes-maj-tiers,
                                  date-du-jour)

  |->

    for all  dde-maj-tiers  :

       IN ( dde-maj-tiers, ddes-maj-tiers)  :

     res-maj-tiers = MAJ-F-TIERS (fich-tiers,
                                  dde-maj-tiers,
                                  date-du-jour)
```

* pour une décomposition-or

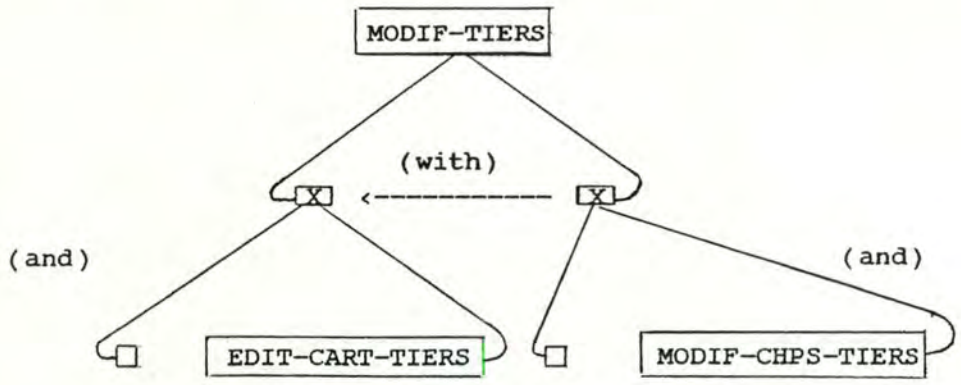
```
res-maj-tiers  =  MAJ-F-TIERS  (fich-tiers,
                                dde-maj-tiers,
                                date-du-jour )

  |->

    ( not  OK-MAJ-TIERS  ( fich-tiers, dde-maj-tiers,
                           date-du-jour )
      and

      mess-err-maj-tiers
        =  DEF-MESS-ERR-MAJ-TIERS ( fich-tiers,
                                    dde-maj-tiers,
                                    date-du-jour ))

   or

    ( OK-MAJ-TIERS  (fich-tiers, dde-maj-tiers,
                     date-du-jour)
      and

      res-maj-tiers-ok
        = MAJ-TIERS-OK ( fich-tiers, dde-maj-tiers))
```

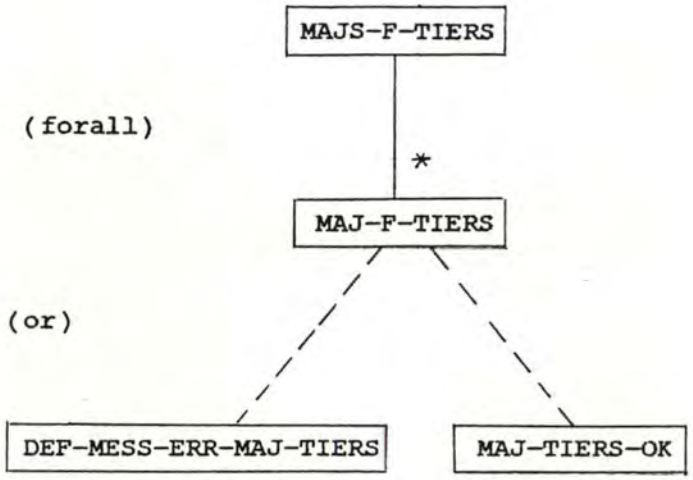
* pour une décomposition-with et -and

```
res-maj-tiers-ok  =  MODIF-TIERS (fich-tiers,
                                  dde-maj-tiers )
  |->

     fich-tiers' = SORT ( APPEND (fich-tiers-filtre,
                                  tiers-modifie),
                           INDIC-T)
   and

     carton-tiers = EDIT-CART-TIERS (tiers-modifie)

  with

     fich-tiers-filtre
       = FILTER (fich-tiers,TUPLESEL (INDIC-T, tiers)
                 <> TUPLESEL (INDIC-T, dde-maj-tiers))
   and

     tiers-modifie
       =  MODIF-CHPS-TIERS (fich-tiers, dde-maj-tiers)
```

Par application récursive des mécanismes d'abstraction ci-dessus, on obtient une forêt d'arbres. Chaque arbre est associé à une fonction devant être réalisée par le système. Chaque noeud dans un arbre concerne la spécification d'une opération, c'est-à-dire un bloc-opération. L'arc entre le noeud-père et le noeud-fils représente la relation résultant de l'explicitation du noeud-père (décomposition -and, -or, -forall, -with). Les feuilles sont les opérations pré-spécifiées dans la bibliothèque.

Voici les arbres correspondants aux exemples ci-dessus.





où ⊠ dénote un noeud intermédiaire pour expliciter un and et □ dénote une opération de la bibliothèque ou une combinaison de celles-ci.

1.2.2 Modèle des objets : l'univers du problème

Les types d'objets, à l'exception de ceux de la bibliothèque, sont spécifiés par les opérations qui leur sont associées et par 2 types de clauses :

- un invariant, facultatif, permettant d'exprimer les propriétés devant être conservées par l'application de n'importe quelle opération associée au type,

- une assertion d'explicitation de structure définissant le type comme une composition de constructeurs de type de la bibliothèque, soit directement, soit via des types intermédiaires qui ont été, à leur tour, spécifiés.

Exemples : décomposition structurelle :

```
RES-MAJ-TIERS-OK |-> CART-PROD [ FICH-TIERS,
                                 CARTE-TIERS ]   (1)

FICH-TIERS  |->  SEQ [ TIERS ]    (2)
```

décomposition fonctionnelle :

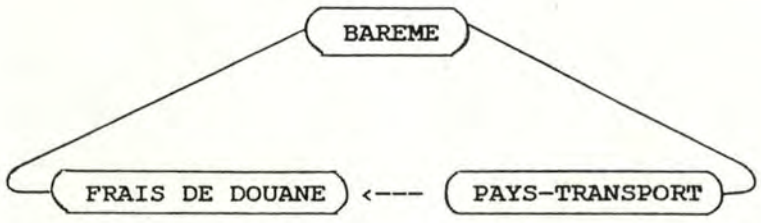
```
BAREME |-> TAB [PAYS-TRANSPORT -> FRAIS-DE-DOUANE]  (3)
```

spécialisation :

```
RES-MAJ-F-TIERS  |-> UNION [ RES-MAJ-TIERS-OK,
                             MESS-ERR-MAJ-TIERS ]  (4)
```

La structure résultant de l'application récursive des différents mécanismes est à nouveau une forêt d'arbres. Chaque arbre est associé à une classe d'objets manipulée par le système. Chaque noeud dans un arbre concerne la spécification d'un type d'objet. L'arc entre le noeud-père et le noeud-fils représente la relation résultant de l'explicitation du noeud-père (décomposition -CART-PROD, -SEQ, -TAB, -UNION, spécialisation IS-A). Les feuilles représentent les types de base de la bibliothèque.

Voici l'arbre correspondant à l'exemple (3)



1.2.3 Le lien entre les deux modèles

L'intégration des deux modèles s'effectue de la façon suivante.

1. Ils utilisent les composants d'une même bibliothèque de types abstraits prédéfinis.
2. Les types d'objets apparaissant au fur et à mesure dans l'énoncé du problème sont spécifiés formellement dans l'univers.De façon duale, les opérations associées aux types d'objets dans l'univers sont spécifiées formellement dans l'énoncé du problème.

Notons qu'il existe une certaine redondance entre l'univers et l'énoncé du problème. Celle-ci est utile pour les contrôles de cohérence et de complétude.

1.2.4 La bibliothèque de spécification

La bibliothèque comprend un certain nombre de types et d'opérations associées qui sont spécifiées algébriquement [DUBOIS 85]. Une de ses caractéristiques principales est de n'être pas figée. En effet, son contenu peut évoluer au gré des besoins. L'objectif est de définir une fois pour toutes les types d'objets et les opérations revenant couramment dans la spécification d'applications d'un domaine donné. On distingue les types de base (par exemple, entiers, naturels, caractères, string, date) des types paramétrés (produit cartésien, union disjointe, suite, ensemble et table).

A chaque type d'objet sont associées des opérations. Ainsi, par exemple, on pourra associer au type SUITE les opérations d'insertion ou de sélection d'un élément, de filtrage sur une propriété afin de définir des sous-suites dont tous les éléments satisfont cette propriété. La suite temporelle est une spécialisation intéressante du type "SUITE" car elle offre des opérations faisant intervenir le temps et les historiques du système à spécifier.
Pour le type "PRODUIT CARTESIEN", on aura l'opération de construction (TUPLEFORM) et celle de sélection d'un champ (TUPLESEL).

1.2.5 La structure des documents

La stucture des blocs est la suivante.

- Pour une opération,

  le bloc se décompose en trois parties :

  - la description formelle : elle reprend la liste des noms du résultat et des arguments de l'opération ainsi que l'assertion d'explicitation de l'opération et celle des préconditions éventuelles.

    Exemple : description formelle de l'opération

    EDIT-F-C-NTVA

    FORMAL SPEC :

    RESULT : fiches-cli-ntva

    ARGUMENTS : fich-tiers

```
EXPLICITATION OF INPUT-OUTPUT ASSERTION :

  fiches-cli-ntva  =  EDIT-F-C-NTVA  (fich-tiers)

  |->

  for all    tiers    :

             IN (tiers, fich-tiers-trie-ntva)
             and
             TUPLESEL (QUALI-T, tiers)  =  1000  :

       fiche-cli-ntva  = tiers

  with

  fich-tiers-trie-ntva = SORT (fich-tiers, NUM-TVA)
```

- le profil : il donne le type de tout objet apparaissant dans la description formelle ainsi que les domaines et le codomaine de toutes les opérations et préconditions apparaissant dans la description formelle. (Les préconditions sont en fait des opérations dont le codomaine est toujours le type prédéfini BOOLEEN).
  Notons que ces types d'objets sont spécifiés formellement dans l'univers du problème.

- la description en langage naturel de l'objectif et de la portée de l'opération, des différents objets, opérations et préconditions apparaissant dans la description formelle. L'intention est de rendre la spécification plus compréhensible et communicable à l'utilisateur.

- Pour un type d'objet,

  le bloc se décompose en trois parties :

  - la description formelle : elle reprend la liste des noms des opérations associées au type, l'explicitation de la structure de type ainsi que l'explicitation des invariants s'il y en a.

    Exemple : description formelle du type FICH-TIERS

```
FORMAL SPEC :
-------------

 ASSOCIATED OPERATIONS :

     GEST-F-TIERS,    MAJS-F-TIERS,
     EDIT-F-F,        EDIT-F-C,
     EDIT-F-C-NTVA,   MAJ-F-TIERS,
     MAJ-TIERS-OK,    DEF-MESS-ERR-MAJ-TIERS,
     CREA-TIERS,      MODIF-TIERS,
     ANNU-TIERS,      ...

 EXPLICITATION OF TYPE STRUCTURE :

    FICH-TIERS  |->  SEQ [ TIERS ]
```

  - le profil : il reprend les domaines et le codomaine des opérations associées ainsi que le type de chaque objet intervenant dans la description formelle.
  - la description en langage naturel des différents opérations, types et invariants.

Il existe une distinction entre les blocs terminaux et non-terminaux.

En ce qui concerne les blocs terminaux,

- dans l'énoncé du problème, l'explicitation de l'opération se fera en termes des opérations prédéfinies de la bibliothèque.
- dans l'univers du problème, l'explicitation du type se fera en termes des types de base.

Par contre, pour les blocs non-terminaux,

- dans l'énoncé du problème, on utilisera des opérations à spécifier ou une combinaison de ces dernières et d'opérations de la bibliothèque dans l'assertion d'explicitation.
- dans l'univers du problème, le type spécifié contiendra, dans son explicitation, des types à spécifier avec ou non des types de base.

La structure de la forêt correspondant à l'énoncé (ou l'univers) du problème se présente sous forme d'arbres dont chaque noeud est identifié par le nom de l'opération (ou du type d'objet) correspondant et les arcs sont représentés selon la convention graphique illustrée ci-dessus et exposée dans l'article [DUBOIS 85].

1.3 Présentation du méta-algorithme

La construction de l'énoncé et de l'univers du problème est guidée par un méta-algorithme. Celui-ci consiste en une composition d'applications des mécanismes d'abstraction mentionnés précédemment, selon des stratégies qui seront exposées par la suite.

Le fait de savoir quelle stratégie appliquer dans un état donné du processus de spécification fait actuellement l'objet de recherches [DUBOIS 86]. Les stratégies à suivre semblent dépendre à la fois du type de problème considéré et de la façon dont le spécifieur voit le problème. Des choix sont à faire, notamment entre un processus orienté-objet ou orienté-fonction.

Un processus orienté-fonction consiste en une application récursive des règles d'explicitation aux opérations. Cependant, dans certains cas, une approche orientée-objet semble plus naturelle. Les règles d'explicitation sont alors appliquées aux types d'objets et la spécification des opérations n'est qu'un effet de bord de ces applications. Il est parfois plus utile de suivre une approche mixte.

Un autre choix réside dans la démarche. Généralement, on procède de manière descendante en appliquant les règles d'explicitation récursivement de façon à produire des spécifications d'opérations et de types d'objets de plus en plus fines. Mais une approche ascendante par généralisation ou agrégation d'opérations et/ou de types d'objets peut s'avérer mieux adaptée, à moins que ce ne soit une approche mixte.

Au niveau local d'un pas d'explicitation lors du processus de spécification, on peut appliquer différentes stratégies : par exemple, une stratégie régressive, une stratégie progressive, une stratégie de réutilisation des connaissances ou une stratégie mixte. Chacune des stratégies locales va maintenant être présentée.

1. la stratégie de réutilisation des connaissances

Cette stratégie consiste à essayer de rapprocher le problème courant d'un problème déjà spécifié.

Ayant trouvé, dans la bibliothèque de spécification ou dans la partie déjà spécifiée de l'application, une opération (un type d'objet) équivalente à celle (celui) que l'on veut exprimer, il faut s'assurer de la compatibilité de l'opération (du type d'objet) et celle (celui) que l'on explicite.

Cette stratégie est terminale dans le processus de spécification : l'opération ou le type d'objet ainsi spécifié(e) ne sera plus explicité(e) au pas suivant.

Cette stratégie peut comporter des risques. On peut en effet avoir un circuit dans les spécifications : une opération spécifiée à l'aide d'une autre, cette dernière étant elle-même explicitée à l'aide de la première, soit directement, soit via des opérations intermédiaires. Une situation analogue peut se présenter au niveau des types d'objets.

Exemple : pour les opérations, nous aurons à des endroits différents de la spécification :

```
res = OP (arg)   |->  res = OP1 (arg)
...
res = OP1 (arg)  |->  res = OP' (arg)
...
res = OP' (arg)  |->  res = OP" (arg2)
                      with arg2 = OP (arg)
```

pour les types,

```
T   |-> CART-PROD [T1, T2]
...
T1  |-> SEQ [T]
```

2. les stratégies régressives

Il s'agit de stratégies guidées par l'analyse des résultats d'une opération, c'est-à-dire de la structure et des propriétés des résultats attendus. L'expérience révèle qu'il est souvent plus naturel et plus sûr de se concentrer sur la spécification des résultats d'une opération plutôt que sur la spécification des différents arguments nécessaires à la définition de ce résultat (le risque d'oublier des résultats est réduit ...).

L'application d'une stratégie régressive à une opération consiste à déduire, à partir de la structure du résultat de cette opération, une structure d'opérations plus explicites.
A chaque pas d'explicitation, de nouvelles opérations sont ainsi mises en évidence et doivent être à leur tour spécifiées. L'application de la stratégie peut faire aussi apparaitre des données intermédiaires. Les opérations définissant ces données intermédiaires devront être spécifiées elles aussi. Ces opérations apparaitront, dans le pas d'explicitation considéré, par le mécanisme d'une décomposition-with. La décomposition-with exprime une composition fonctionnelle au niveau des opérations.
Ces stratégies se différencient selon le type de constructeur introduit pour définir la structure du résultat.

Soit PRE et res = OP(arglist), les résultats d'un pas d'explicitation précédent, où

OP est l'opération considérée,

res est de type RES, le résultat attendu,

arglist, une liste d'arguments

et PRE, la précondition.

Voici quelques exemples de formulations de stratégies régressives que nous avons utilisés dans la spécification de l'application "Ventes" de CBR (présentée au point 1.1 du chapitre 3). Elles peuvent être complétées par d'autres stratégies associées à d'autres constructeurs de type.

A. De RES |-> CART-PROD [RES1, RES2, ..., RESn]
on peut déduire

```
res = OP(arglist) |->     res1 = OP1(arglist1, intres)
                    and   res2 = OP2(arglist2, intres)
                    and   ...
                    and   resn = OPn(arglistn, intres)
                     with intres = OPINTRES(arglist, ...)
```

où intres est un résultat intermédiaire défini par OPINTRES.

Ce scénario permet de passer de la définition d'un objet à celle de ses composants. De manière générale, cette stratégie sera choisie lorsque le résultat est perçu comme un agrégat dont les composants peuvent être étudiés séparément.

B. De RES |-> UNION [RES1, RES2, ..., RESn]
on peut déduire

```
PRE |-> PRE1 or PRE2 or ... or PREn
res = OP(arglist) |->     PRE1 and res1 = OP1(arglist1, intres)
                    or    PRE2 and res2 = OP2(arglist2, intres)
                    or    ...
                    or    PREn and resn = OPn(arglistn, intres)
                     with intres = OPINTRES(arglist, ...)
```

Remarquons qu'il s'agit ici de "ou" exclusifs.

L'application de cette stratégie permet de passer de la définition d'un objet à une définition plus précise tenant compte du fait que le résultat est soit de type RES1, soit de type RES2, ..., soit de type RESn.

C. De RES |-> SEQ [RES']

un des 2 schémas suivants peut être choisi :

C.1 Si un argument de arglist, est aussi une suite telle qu'il existe une correspondance élément par élément entre cette suite et la suite résultat, l'explicitation devient :

```
res = OP(arglist)
    |->  forall e : IN (e, sg) and PRE (e) :
                    res' = OP'(e, ...)
```

où res' et e dénotent les élément correspondants des suites res et sg, avec res' de type RES', et où sg est la suite "guide" que l'on a pu mettre en évidence dans arglist et IN est une opération de la bibliothèque de spécification.

C.2 Une autre stratégie peut être appliquée dans le cas d'une suite : c'est la stratégie d'éclatement d'une suite introduisant ainsi des suites auxiliaires.

La suite résultat peut être définie par application d'opérations de la bibliothèque à une ou plusieurs suites auxiliaires, ces dernières devant, à leur tour, être définies.

Cela donne :

```
res = OP(arglist)
    |->      res = BIBLIO-OP(res1, res2, ..., resn)
         with     res1 = OP1(arglist1, intres)
              and  res2 = OP2(arglist2, intres)
              and  ...
              and  resn = OPn(arglistn, intres)
               with intres = OPINTRES(arglist, ...)
```

où BIBLIO-OP est une opération de la bibliothèque définie sur les suites et chaque resi est une suite auxiliaire.

Des stratégies similaires à celles du point C peuvent être définies sur les ensembles. On notera aussi que C.1 est une stratégie mixte déductive/inductive tandis que C.2 est une stratégie mixte déductive/réutilisation des connaissances.

D. De RES |-> TABLE [RES1 -> RES2]
on peut déduire

```
res = OP(arglist)
    |->     res2 = OP2(arglist, res1, intres)
       with res1 = OP1(arglist, intres)
          with intres = OPINTRES(arglist, ...)
```

où res2 (res1) est de type RES2 (RES1), et arglist et intres ont la même signification que précédemment.

3. les stratégies progressives.

Elles correspondent à l'approche guidée par l'analyse des données c'est-à-dire des structures et propriétés des arguments. Ces stratégies sont les contreparties des stratégies régressives où les résultats sont remplacés par les arguments.

Une stratégie souvent utilisée est la stratégie d'analyse par cas.

De ARG |-> UNION [ARG1, ARG2, ..., ARGn]

on peut déduire

```
PRE |-> PRE1(arg1) or PRE2(arg2) or ... or PREn(argn)

res = OP(arg) |->     PRE1(arg1) and res = OP1(arg1, intarg)
                  or  PRE2(arg2) and res = OP2(arg2, intarg)
                  or  ...
                  or  PREn(argn) and res = OPn(argn, intarg)
                    with intarg = OPINTARG(arg, ...)
```

où ARG est le type des arguments arg avant l'explicitation, chaque ARGi est caractérisé par un cas possible PREi concernant les éléments de ARG, argi est de type ARGi, res est le résultat de type RES, intarg est un argument intermédiaire défini par une nouvelle opération OPINTARG.

Nous présentons maintenant le méta-algorithme correspondant à une démarche descendante guidée par les opérations. Il s'agit du méta-algorithme que nous avons appliqué dans la spécification de l'application CBR. Notons, cependant que RSL peut tout aussi bien être utilisé dans une approche ascendante et/ou guidée par les données.

La première étape consiste, à partir des besoins exprimés par l'utilisateur, à identifier des fonctions que devra réaliser le système ainsi que des données et les résultats du système.
A chaque fonction (ou type de donnée) ainsi identifié(e) sera associée une racine d'arbre dans l'énoncé du problème (ou l'univers du problème).

De sorte que pour chaque fonction, leur spécification aura la forme :

res = FONCTION (arg, ...)

arg : INPUT

res : OUTPUT.

Les spécifications ainsi obtenues seront explicitées par la suite. Il s'agit de les raffiner par choix et application d'une stratégie.
A chaque pas d'explicitation, il faut :

choisir une opération non-terminale, choisir une stratégie et appliquer cette dernière afin de spécifier l'opération au moyen d'une précondition, d'une assertion d'explicitation et d'un typage des objets.

On construit ainsi pas à pas la double arborescence.
L'explicitation est terminée lorque les opérations obtenues sont des opérations de la bibliothèque ou une combinaison de celles-ci.

Ayant terminé la phase d'explicitation, il faut encore éliminer les redondances dans l'énoncé et l'univers du problème. Cette phase ascendante peut porter sur l'ensemble des arbres et non plus sur chacun des arbres comme dans la phase d'explicitation.

Ceci donne la forme générale suivante de méta-algorithme :

```
* ABSTRACTION (besoins de l'utilisateur)
  { identification d'un certain nombre d'opérations FONCTION,
    avec types INPUT et OUTPUT associés}

* POUR CHAQUE (FONCTION, INPUT, OUTPUT) identifié
      SPECIFICATION (FONCTION, INPUT, OUTPUT)
  { description complète de l'énoncé du problème et de
    l'univers du problème }

* SI nécessaire : UNIFICATION ( énoncé du problème,
                                univers du problème)
  { élimination des redondances }
```

où la procédure SPECIFICATION est de la forme :

```
SPECIFICATION (OP, ARG, RES)
  DEBUT
    * SELECTIONNER une stratégie parmi les suivantes :
                  réutilisation, régressive, progressive ;
    * APPLIQUER la stratégie choisie ;
    * SI  une stratégie  régressive ou progressive  a été
      appliquée
      ALORS
       DEBUT
         * POUR CHAQUE objet (arg,res) introduit : le TYPER ;
```

* POUR CHAQUE nouvelle opération OP' introduite :

DEBUT

* DONNER son domaine INPUT' et son codomaine OUTPUT'
* SPECIFICATION (OP', INPUT', OUTPUT')

FIN

* COMPLETER la liste des arguments de OP par les nouvelles données identifiées et AJOUTER les types correspondants s'il y en a

FIN

FIN

Cet algorithme est repris de l'article [DUBOIS 85].
Remarquons qu'a priori cet algorithme travaille en profondeur d'abord en ce qui concerne le traitement des opérations non-terminales ; d'autres stratégies de contrôle peuvent évidemment être considérées (largeur d'abord, opération la plus critique ou la plus spécifique d'abord, etc.).

Relevons que l'application de ce méta-algorithme nécessite un certain nombre de décisions de la part du spécifieur telles que, par exemple, le choix d'une stratégie, le choix d'un constructeur de type, le nom d'une opération ou d'un type d'objet, ou encore le choix d'une explicitation terminale d'une opération, d'un prédicat (précondition ou invariant) ou d'un type.

2 PROLOG : un langage de programmation

2.1 Introduction

Le langage de programmation PROLOG (PROgramming in LOGic) constitue un premier essai dans la conception d'un langage qui permettrait au programmeur de décrire ses tâches au moyen de la logique plutôt qu'au moyen de constructions de la programmation impérative conventionnelle exprimant comment la machine doit accomplir ces tâches [CLOCKSIN 81].

L'une des idées principales de la programmation logique est qu'un algorithme consiste en deux composants : la logique et le contrôle. Le composant logique exprime ce qu'est le problème à résoudre, et le composant de contrôle exprime la façon dont il doit être résolu [KOWALSKI 79b]. Ces deux composants sont étroitement liés en programmation impérative. Le but de PROLOG, et de la programmation déclarative en général, est de les séparer autant que possible : idéalement, le programmeur devrait simplement décrire le composant logique d'un algorithme, et le contrôle serait exercé par le système.

Un programme logique consiste en un ensemble de "phrases" exprimant de la connaissance sur le problème à résoudre. La formulation de cette connaissance passe par deux concepts de base :

- des objets ou individus constituant le domaine du problème, et
- des relations entre ces objets.

Le langage formel utilisé dans ce contexte, et donc à la base de PROLOG, est la logique des prédicats du premier ordre. Nous allons rappeler les points importants concernant la logique des prédicats du premier ordre, la résolution de problèmes dans ce cadre, ainsi que les principales caractéristiques de PROLOG. Pour plus de détails, on peut consulter [CLOCKSIN 81], [HOGGER 84], et [KOWALSKI 79]. Les raisons du choix de PROLOG comme langage de programmation seront explicitées dans le chapitre 3, point 2.

2.2 La logique des prédicats du premier ordre

La logique permet d'étudier si des suppositions impliquent des conclusions. Elle nous permet de dire, par exemple, que les suppositions

Toute femme est mortelle, et

Madame Tatcher est une femme

impliquent la conclusion

Madame Tatcher est mortelle.

La logique ne s'intéresse pas à la vérité ou à l'acceptabilité de phrases individuelles, mais aux relations entre phrases [KOWALSKI 79a]. La logique des prédicats du premier ordre est un sous-ensemble de la logique possédant une grande puissance d'expression. Elle permet en effet l'emploi de variables (contrairement au calcul

des propositions), et offre la possibilité de les quantifier. Cependant, ces variables ne peuvent représenter que des objets alors que dans la logique des prédicats du second ordre, les variables peuvent également représenter des prédicats.

A. Syntaxe

Les quatre concepts de base sont les suivants :

1. Un terme peut être
   - une constante,
   - une variable,
   - un terme composé f(t1, ..., tk) où f est un symbole de fonction k-aire et les ti sont des termes,

2. Une formule atomique ou prédicat a la forme P(t1, ..., tn) où P est un symbole de prédicat n-aire et les ti sont des termes.

3. Une formule bien formée
   - est un prédicat, ou
   - a l'une des formes suivantes :

(F1)	F1 $\wedge$ F2 (F1 et F2)
~F1 (non F1)	F1 V F2 (F1 ou F2)
F1 <- F2 (F1 si F2)	F1 <-> F2 (F1 équivaut à F2)
($\exists$x) F1	($\forall$x) F1

où F1 et F2 sont des formules bien formées.

4. Une formule bien formée fermée (ou phrase) est une formule bien formée où toute variable est quantifiée.

5. Un littéral est une formule bien formée ou une négation de formule bien formée.

L'ensemble de toutes les phrases que l'on peut construire à partir des règles 1 à 5 constitue le langage de la logique du premier ordre. En pratique, tous les problèmes concernant l'implication logique dans la logique des prédicats du premier ordre peuvent être traitées sous forme clausale :

Une clause est une expression de la forme

B1, B2, ..., Bm <- A1, A2, ..., An

où

- les Ai et Bj sont des formules atomiques,
- n et m >= 0,
- les Ai sont les conditions jointes de la clause,
- les Bj sont les conclusions alternatives de la clause,
- toute variable apparaissant dans une formule atomique est implicitement quantifiée universellement.

Une clause B1, ..., Bm <- A1, ..., An contenant les variables x1, ..., xk est à interpréter comme suit :

pour tout x1, ..., xk,

B1 ou B2 ou ... ou Bm si A1 et A2 et ... et An

Si n = 0, la clause est à interpréter comme une exression sans condition :

pour tout x1, ..., xk,

B1 ou B2 ou ... ou Bm

Si m = 0, la clause est à interpréter comme une négation :

pour tout x1, ..., xk,

~ (A1 et A2 et ... et An)

Si n = m = 0, la clause est à interpréter comme une inconsistance, une contradiction.

Une clause de la forme

B1, ..., Bm <- A1, ..., An

peut être représentée comme une disjonction de formules atomiques et de négations de formules atomiques, de la façon suivante :

B1 V B2 V ... V Bm V ~A1 V ~A2 V ... V ~An

Un cas particulier de clause est la clause de Horn où m <= 1 (au plus un élément dans la conclusion).

Les phrases écrites sous forme clausale ont une syntaxe très simple, tout en ayant une puissance d'expression équivalente à celle de la formulation standard de la logique des prédicats. Il

existe des méthodes pour transformer des phrases de premier ordre sous forme clausale. Ces méthodes sont décrites, entre autres, dans [CLOCKSIN 81].

B. Sémantique

Alors que la syntaxe traite de la grammaire des phrases, la sémantique traite de leur signification. La sémantique associe à chaque expression, des objets dans un domaine d'interprétation.

Soit S un ensemble de phrases, et D le domaine du discours (c'est-à-dire un ensemble non vide d'individus). Une interprétation de S sur le domaine D consiste en trois associations A1, A2 et A3 où

A1 associe à chaque constante de S un individu de D,

A2 associe à chaque fonction n-aire de S une fonction de Dn dans D,

A3 associe à chaque symbole de prédicat n-aire de S une fonction de Dn dans l'ensemble des booléens [HOGGER 84].

Exemple :

S positif (UN)
(pour tout x) (positif(double(x)) <- positif(x))

D l'ensemble des nombres naturels 1, 2, 3, ...

A1 associe 1 avec UN

A2 associe { (1,2), (2,4), ... } avec double

A3 associe { (1,true), (2,true), ... } avec positif

Le rôle de l'interprétation est de permettre d'attribuer une valeur de vérité à chaque phrase de S. Cette attribution se fait comme suit : soit F une formule de la phrase considérée,

- si F est un prédicat sans variable, remplacer chaque constante de F par l'individu que A1 lui associe, et remplacer chaque fonction n-aire par l'individu que A2 associe à son n-uple. Ceci conduit à un prédicat de la forme p(d) où d est un n-uple d'individus ; la valeur de F est alors la valeur que A3 associe à d pour le symbole de prédicat p.

- si F a la forme $(\forall x)$ F' où F' est une formule dans laquelle x apparait , la valeur de F est true si true est la valeur de chaque instance de F' obtenue en remplaçant toutes les occurrences de x par un individu dans D. Elle est false sinon.

- si F a la forme $(\exists x)$ F' où F' est une formule dans laquelle x apparait. La valeur de F est true si la valeur d'au moins une instance de F' obtenue en remplaçant toutes les occurrences de x par un individu de D est true. Elle est false

sinon.

- si F a la forme ~F1, F1 ∧ F2, F1 <- F2, F1 V F2 ou F1 <-> F2 où F1 et F2 sont des formules bien formées, la valeur de F est obtenue par la table de vérité classique :

F1	F2	~F1	F1 ∧ F2	F1 V F2	F1 <- F2	F1 <-> F2
t	t	f	t	t	t	t
t	f	f	f	t	t	f
f	t	t	f	t	f	f
f	f	t	f	f	t	t

2.3 La résolution de problèmes en programmation logique

Dans le cadre de la programmation logique, la résolution de problèmes s'assimile à la preuve de théorèmes : à partir d'un ensemble (S) d'axiomes, c'est-à-dire de formules bien formées, on veut savoir si une formule bien formée particulière (C) est vraie dans toute interprétation satisfaisant S. Si oui, c'est un théorème, une conséquence logique des axiomes.

Prouver qu'une formule bien formée est un théorème peut se faire de deux façons :

- sémantiquement, en montrant que la formule est vraie dans toute interprétation satisfaisant l'ensemble des axiomes, ou
- syntaxiquement, en utilisant une procédure de démonstration [KOWALSKI 79a].

La preuve automatique de théorème se fait syntaxiquement, à l'aide d'un démonstrateur. Celui-ci, recevant un ensemble S d'axiomes et une formule bien formée C, répond oui ou non (ou ne fournit pas de réponse).

Les qualités d'un bon démonstrateur sont les suivantes :

- complétude : si C est une conséquence logique de S, tôt ou tard, le démonstrateur répondra oui.

- cohérence : si le démonstrateur répond oui, C est vraiment une conséquence logique de S. S'il répond non, C n'est pas une conséquence logique de S.

- semi-décidabilité : si C n'est pas une conséquence logique de S, il se peut que le démonstrateur ne réponde rien. Il y a en effet une impossibilité théorique d'avoir un démonstrateur entièrement décidable.

Principe de démonstration

Dans la forme clausale de la logique, le principe de démonstration le plus utilisé est la réfutation : démontrer que C est une conséquence logique de S équivaut à prouver que S U { ~C } est inconsistant, c'est-à-dire vrai dans aucune interprétation. La réfutation consiste donc à nier l'expression que l'on veut démontrer et montrer qu'en ajoutant cette négation à l'ensemble des axiomes, on arrive à une contradiction.

Pour cela, on utilise le plus souvent le principe de résolution. La résolution est une règle d'inférence selon laquelle si l'on a les clauses

E1 V ~E2
et E2 V ~E3,

la clause E1 V ~E3 en est une conséquence logique.

Deux clauses sont "résolubles" (au sens du principe de résolution) si elles contiennent des paires de littéraux identiques, comme dans l'exemple suivant :

père (adam, abel)
parent (adam, abel) V ~père (adam, abel)
---
parent (adam, abel)

ou si elles contiennent des paires de littéraux qui peuvent être rendus identiques par un processus de substitution, l' unification. Une substitution est un ensemble de remplacements de variables par des termes.

Soient θ une substitution { v1 = t1, ..., vn = tn }, et
E une expression.

Appliquer la substitution θ à E consiste à remplacer chaque occurrence de vi dans E par le terme ti (i = 1, ..., n). Le résultat est noté Eθ. Deux littéraux E1 et E2 sont unifiables s'il existe une substitution θ telle que E1θ = E2θ.

Exemple : père (adam, X)
parent (A, B) V ~père (A, B)

sont résolubles, avec la substitution { A = adam, X = B }.

La clause résultante est parent (adam, B)

Stratégies de démonstration

Dans le cas de clauses de Horn (c'est à ces clauses que l'on se limite souvent, et notamment en PROLOG), on distingue deux stratégies principales de démonstration. Toutes deux utilisent le principe de résolution. Il s'agit de la stratégie top-down (ou chainage arrière) et de la stratégie bottom-up (ou chainage avant).

Dans la stratégie TOP-DOWN , on raisonne "en arrière", à partir de l'expression à démontrer. Pour prouver qu'une clause C est une conséquence logique d'un ensemble S de clauses, on cherche une clause de S résoluble avec C ; on applique la résolution, et on continue avec la clause résultante ... On exploite ainsi les clauses dérivées jusqu'à dériver la clause vide.

Dans la stratégie BOTTOM-UP , par contre, on raisonne "en avant" : on applique la résolution aux clauses de départ jusqu'à ce que, éventuellement, la résolution d'une clause dérivée et de la clause à prouver donne la clause vide. Il est inévitable, dans

cette stratégie, de dériver des clauses inutiles, qui ne contribueront pas à prouver le théorème. C'est pourquoi cette stratégie est moins efficace que la précédente.

Certaines méthodes de démonstration incorporent les deux stratégies. Cependant, la plus utilisée est la stratégie TOP-DOWN. Elle est notamment utilisée en PROLOG. Nous l'illustrerons par un exemple dans le point suivant.

2.4 PROLOG

On retrouve en PROLOG les mêmes concepts syntaxiques que dans la logique des prédicats du premier ordre : termes, prédicats et phrases, réduites ici aux clauses de Horn.

Une constante est représentée par une chaine de caractères minuscules.

Une variable est représentée par une chaine de caractères commençant par une majuscule.

Un terme composé est représenté par un nom de foncteur en minuscules, suivi d'une liste de termes : "f(t1, ..., tn)"

Un prédicat est représenté par un nom de prédicat en minuscules suivi d'une liste de termes : "p(t1, ..., tn)"

Une clause de Horn (positive) est représentée par une règle "B :- A1, ..., An." où B et les Ai sont des prédicats.

Une clause de Horn sans partie droite est représentée comme suit : "B." où B est un prédicat, et est appelée un fait.

Une clause de Horn sans partie gauche est représentée de la façon suivante : "?- A1, ..., An." où les Ai sont des prédicats, et est appelée question. (Une question en PROLOG correspond donc à une négation)

Un programme PROLOG est destiné à être exécuté. C'est pourquoi, outre leur interprétation déclarative, les clauses PROLOG ont une interprétation procédurale. L'interprétation procédurale des clauses en PROLOG est la suivante :

B :- A1, ..., An. B :- C1, ..., Cn.	est interprété comme une déclaration de procédure dont le nom est B et le corps { A1, ..., An } ou { C1, ..., Cn } où les Ai et Ci sont des appels de procédures,
B.	est interprété comme un fait sans condition,
?- A1, ..., An.	est interprété comme une déclaration d'objectifs à établir (goal statement), un ensemble d'appels de procédures,
La clause vide	est interprétée comme une inconsistance, une contradiction.

Poser la question "?- p(X), q(Y)." équivaut à demander pour quelles instances (ou valeurs) de X et Y, p(X) et q(Y) sont démontrables.

Un programme PROLOG consiste en une question et un nombre quelconque de faits et de règles. Il est exécuté en le soumettant à un interpréteur PROLOG. L'interpréteur est un programme capable d'accomplir une démonstration en utilisant le principe de résolution.

Le moteur d'inférence de PROLOG travaille par chaînage arrière (TOP-DOWN) : il part de la question, et par réduction à des sous-objectifs, il tente d'arriver à des sous-objectifs très simples qui répondent aux faits, construisant ainsi un arbre de recherche et/ou (search tree).

Lors de la démonstration d'un théorème, l'interpréteur est confronté à des choix : quelle règle sélectionner pour satisfaire un (sous-)objectif quand plusieurs règles sont possibles, et par quel sous-objectif commencer quand il y en a plusieurs. En PROLOG, l'ordre de considération des règles est statique : on choisit la première dont la partie gauche s'unifie avec l'objectif à réaliser. L'ordre de considération des sous-objectifs est également statique : on procède de gauche à droite.

Il arrive que lors des réductions à des sous-objectifs, l'interpréteur fasse de mauvais choix (ne conduisant pas à la

clause vide). Dans ce cas, il faut, si possible, revenir en arrière, jusqu'au dernier choix effectué, pour essayer une autre possibilité. Cette procédure de retour est le "backtracking". Lors de l'échec d'un objectif courant, deux situations peuvent se présenter :

- Cet objectif se trouve dans un sous-arbre "ou". Dans ce cas, l'interpréteur essaie de satisfaire l'objectif suivant dans le sous-arbre (noeud frère droit s'il y en a un ou règle suivante), après avoir annulé les instanciations faites en vue d'établir l'objectif ayant échoué.

- L'objectif se trouve dans un sous-arbre "et". Dans ce cas, l'interpréteur retourne au dernier objectif établi, annule les instanciations faites pour établir cet objectif, et réessaie de satisfaire cet objectif avec un autre jeu d'instanciations pour les variables désinstanciées.

Si l'interpréteur arrive à dériver la clause vide signifiant qu'il y a une solution, il répond "yes" et fournit les valeurs des variables apparaissant dans la question et instanciées au cours de l'exécution. Dans le cas contraire,

- soit il répond "no".
- soit il ne répond rien. Il arrive en effet qu'un programme PROLOG "boucle".

Exemple : Soit le programme suivant :

```
p( X, Y) :- p ( Y , X).
p( a, b).
?- p( b, a).
```

Il est clair que la réponse à la question doit être oui, mais pour satisfaire p(b, a), l'interpréteur choisit la première règle, et réduit donc p(b, a) au sous-objectif p(a, b), lui-même réduit à p(b, a), lui-même ... L'interpréteur PROLOG n'est donc pas un démonstrateur complet.

La logique n'étant pas suffisante comme langage de programmation, outre ses caractéristiques logiques, PROLOG possède également des caractéristiques extra-logiques. Celles-ci sont constituées par 4 classes de prédicats prédéfinis

- pour effectuer des opérations d'entrée/sortie telles que lire et écrire dans un fichier, ... (see, tell, read, write, ...)
- pour contrôler partiellement le processus d'exécution. Ainsi, le "cut" (!) permet d'éviter des parcours d'arbre inutiles.
- pour évaluer des expressions arithmétiques (is).
- pour manipuler des clauses (ajouter des clauses à un programme, ...)

Ces prédicats ont une sémantique logique artificielle.

Exemple de résolution de problème en PROLOG :

soit une description de la famille Dupont et de certains liens de famille :

```
(1)  pere( anatole, barnabe).
                /* Anatole est le père de Barnabe */

(2)  pere( anatole, beatrice).

(3)  pere( barnabe, claude).

(4)  pere( barnabe, caroline).


(5)  mere( antoinette, barnabe).

(6)  mere( antoinette, beatrice).


(7)  parent( X, Y) :- mere(X,Y).
                /* X est un parent de Y si X est la
                   mère de Y */

(8)  parent( X,Y)  :- pere(X,Y).
                /* X est un parent de Y si X est le
                   père de Y */


(9)  grand-parent( X, Y) :- parent( X, Z), parent (Z,Y)
                /* X est grand-parent de Y s'il existe
                   Z qui est un enfant de X et en même
                   temps un parent de Y */
```

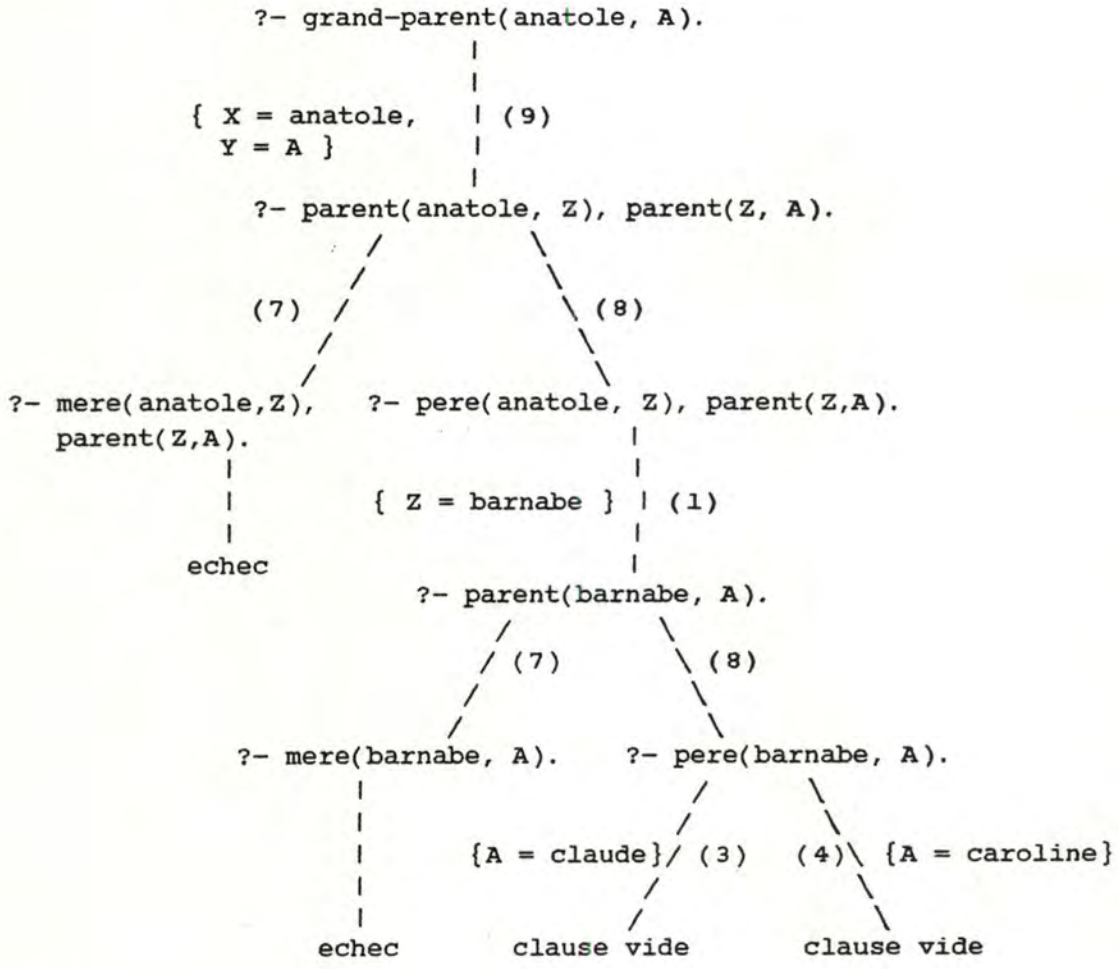
et la question

```
?- grand-parent( anatole, A).
```

Cette question revient à demander s'il existe un A tel que grand-parent(anatole, A) est démontrable, et quel est cet A.

Une question PROLOG correspondant à la négation du théorème à prouver, l'interpréteur l'ajoute à la suite des faits et règles de départ. L'objectif grand-parent(anatole, A) s'unifie avec la tête de la règle (9),avec la substitution { X = anatole, Y = A }. L'objectif de départ est donc réduit à deux sous-objectifs : parent(anatole, Z) et parent(Z, A). L'interpréteur essaie d'abord de satisfaire parent(anatole, Z). Pour cela, 2 règles sont possibles : la (7) et la (8). L'interpréteur commence par la (7), réduisant ainsi le sous-objectif parent(anatole, Z) au sous-objectif mere(anatole, Z). Aucun fait ou règle ne s'unifiant avec ce sous-objectif, il y a échec, et l'interpréteur retourne jusqu'au dernier choix effectué (la sélection de la règle (7) pour satisfaire parent(anatole, Z)). Il essaie de re-satisfaire parent(anatole, Z) au moyen de la règle (8). Il y parvient, en instanciant Z à barnabe. Ensuite, il essaie de satisfaire parent(barnabe, A). Il réussit, en instanciant A à claude. On a ainsi satisfait l'objectif de départ, la réponse à la question est donc "yes", avec A = claude.

L'arbre de recherche correspondant à cet exemple est le suivant :



Réponse : Yes
A = claude (Anatole est un grand-parent de claude)

Remarques :

- Il est possible, en PROLOG, de demander plusieurs réponses à une même question ; dans notre cas, cela donnerait A = caroline.

- Les règles (7) et (8) peuvent s'écrire de façon plus synthétique :

  parent (X, Y) :- mere(X, Y) ; pere(X, Y).

  où le ";" exprime un "ou" inclusif.

3 Passage de RSL à Prolog

Ayant présenté le langage de spécification et le langage cible, nous allons maintenant aborder le problème du passage de l'un à l'autre. Nous allons d'abord traiter des règles de traduction et de l'évaluateur. L'évaluateur est la procédure Prolog implémentant les opérations de la bibliothèque. Ensuite, nous traiterons de l'automatisation de ce passage en exposant les outils utilisés et en expliquant les principes du programme. Il s'agit essentiellement de manipulation d'arbres syntaxiques.
Nous n'exposerons ici qu'une solution possible car le sujet est vaste. La génération systématique d'un programme logique Prolog à partir de spécifications formelles fait actuellement l'objet de travaux de recherches à l'Institut.

3.1 Principes

L'objectif est de transformer les spécifications écrites en RSL en un programme Prolog. Pour ce faire, il faut, dans un premier temps, déterminer des règles de transformation d'opérations RSL (terminales et non terminales) en clauses Prolog. Ceci amène à considérer la représentation des objets de l'univers du problème.
La présentation de ce chapitre suit le méta-algorithme de spécification où l'on commence par les opérations non terminales pour aboutir aux opérations terminales.
La validité des transformations sera justifiée par une analyse de l'état des données (arguments et résultats) à chaque pas du programme.

3.1.1 La transformation des opérations non terminales

La première étape consiste à transformer la notation équationnelle d'une opération, de forme "res = OP (arg1, ...)", en une notation relationnelle "OP (res, arg1, ...)". Ceci permet de voir OP comme un prédicat qui s'écrira dans la syntaxe Prolog : op(Res, Arg1, ...).
Les préconditions RSL correspondent à un type particulier d'opérations et sont déjà sous une forme relationnelle PRE (arg1, ...). Elles seront dès lors directement écrites sous la forme de prédicats Prolog : pre(Arg1, ...). Néanmoins, si l'on désire faire apparaitre explicitement le résultat booléen et que la forme équationnelle de RSL est utilisée, on écrira alors en Prolog le prédicat pre(Res, Arg1, ...) où Res est de type booléen.

Considérons maintenant un pas d'explicitation. Celui-ci peut être perçu comme une équivalence entre une opération complexe et une combinaison d'opérations plus simples. Ces dernières sont reliées par des connecteurs logiques (and, with, or, forall). Par exemple, la règle d'explicitation-and

```
res = OP (arglist)    |->        res1 = OP1 (arglist1)
                      ssi   and
                                 res2 = OP2 (arglist2)
```

produira en Prolog :

```
op( Res, Arglist) :- op1( Res1, Arglist1),
                     op2( Res2, Arglist2).
```

Dans les transformations, seul le sens "si" de l'équivalence sera considéré. L'équivalence n'étant pas traduisible en Prolog, il a fallu choisir un sens. Le sens de l'explicitation a été retenu pour deux raisons principales :

- l'objectif poursuivi est de réduire un problème à des sous-problèmes plus simples. Ce qui correspond au mode de travail du moteur d'inférence de Prolog. L'objectif à établir est réduit à des sous-objectifs, eux-mêmes réduits à des sous-(sous-objectifs), eux-mêmes ... et cela récursivement jusqu'au faits.
- Prolog travaille sur des clauses de Horn. Dans une telle clause, on ne peut avoir qu'un seul conséquent (littéral positif) dans le membre de gauche. Or, dans le sens retenu, cette restriction est déjà respectée.

Le mode définissant la manière d'exploiter la procédure Prolog obtenue sera toujours le suivant. A l'appel, les arguments doivent tous être instanciés tandis qu'aucun résultat ne peut l'être ; après l'appel, tous les arguments et résultats doivent être instanciés. Si P désigne l'appel, on peut exprimer ceci par

{ in (var, ground) } P { out (ground, ground) }

où in est la situation à l'appel,
out est la situation après l'appel,
var dénote une variable,
ground (abréviation gr.) dénote une expression complètement instanciée (aucune variable n'apparait plus dans l'expression).

Il faudra donc que chaque procédure, à l'appel, soit exploitable selon ce mode afin d'obtenir à la fin de l'exécution du programme prolog généré tous les résultats instanciés. Cette forte restriction sur le mode d'utilisation de Prolog correspond à l'utilisation normale d'un prototype.

Remarquons que l' explicitation-and est traduite par une conjonction de buts qui devront tous être satisfaits. Il reste encore à exprimer la relation entre les résultats apparaissant dans les membres de gauche et de droite de la règle d'explicitation ainsi que la relation du même type entre les arguments.

```
op( Res, Arglist) :- comparg( Arglist, Arglist1, Arglist2),
                     op1( Res1, Arglist1),
                     op2( Res2, Arglist2),
                     compres( Res, Res1, Res2).
```

où comparg(Arglist, Arglist1, Arglist2) est une procédure établissant que Arglist est décomposé en Arglist1 et Arglist2 et compres(Res, Res1, Res2) est une procédure établissant que Res est décomposé en Res1 et Res2.

En fait, comparg et compres sont des relations entre un objet et ses composants. La structure du résultat a guidé l'explicitation. Cette structure sera représentée en Prolog par une liste. Ceci sera développé dans le point 3.1.2 où la traduction de l'univers du problème sera traitée.

La transformation de l'explicitation-and répond à la contrainte de mode énoncée ci-dessus. En effet, on a

```
in ( var, gr.)                  in (  gr.,      var,      var)
 op( Res, Arglist ) :- comparg( Arglist, Arglist1, Arglist2),
                        out(  gr.,      gr.,      gr.)

                      in ( var,    gr.)
                      op1( Res1, Arglist1),
                      out( gr.,    gr.)

                      in ( var,    gr.)
                      op2( Res2, Arglist2),
                      out( gr.,    gr.)

                         in ( var, gr.,   gr.)
                      compres( Res, Res1, Res2).
                         out( gr., gr.,   gr.)

out ( gr., gr.)
```

En imposant un mode sur op, on s'est imposé deux contraintes supplémentaires :

- on s'est imposé une contrainte de mode sur op1 et op2.
- les compositions doivent fonctionner dans les deux sens (arguments ground et résultats var/arguments var et résultats ground) pour respecter le mode d'utilisation. Il faut donc en faire une procédure qui accepte les deux modes.

L'ordre entre les buts à satisfaire garantit la validité du programme Prolog généré. Comparg est le premier but à satisfaire. On est assuré ainsi d'avoir Arglist1 et Arglist2 instanciées (ground) pour les appels de op1 et op2. L'ordre entre op1 et op2 importe peu. Compres doit venir en dernier lieu de façon à avoir le résultat Res instancié à la sortie de la procédure op.

Pour une explicitation-with d'opérations, la même démarche est utilisée.

L'explicitation

```
res = OP (arglist)    |->    res = OP2 (res1)
                                  with
                                  res1 = OP1 (arglist)
```

est traduite en

```
op( Res, Arglist) :- op1( Res1, Arglist),
                     op2( Res, Res1).
```

Le connecteur "with" est une variante du "et" logique exprimant une composition fonctionnelle au niveau des opérations ou une dépendance fonctionnelle au niveau des types d'objets. En Prolog, la traduction correspondant à l'opération du "with" précèdera celle du reste de l'explicitation. Dans notre exemple, il faut mettre op1 avant op2 afin que Res1 soit instancié lors de l'appel de op2. La notion d'ordre est prise en charge par le moteur d'inférence.

A nouveau, la traduction respecte la contrainte de mode.

```
in ( var,  gr.)          in ( var,    gr.)
 op( Res, Arglist)  :-   op1( Res1, Arglist),
                         out( gr.,    gr.)

                         in ( var,  gr.)
                         op2( Res, Res1).
                         out( gr.,  gr.)

out ( gr., gr.)
```

La traduction d'une explicitation-ou d'opérations peut donner deux formes équivalentes.

(i) Tenant compte du fait que les différentes règles d'une même procédure Prolog sont implicitement reliées par un "ou", une solution consiste à obtenir plusieurs règles pour une explicitation.

(ii) En utilisant la disjonction Prolog, on obtient une seule règle dont les buts sont séparés par l'opérateur ";". Rappelons brièvement la signification de la disjonction en Prolog. Si X et Y sont des objectifs, l'objectif X ; Y réussit si X réussit ou si Y réussit.
Dans cette alternative, il est parfois nécessaire d'introduire des parenthèses pour lever l'ambiguité de l'utilisation mixte du et/ou.

Illustrons cela sur un exemple. Soit la règle d'explicitation

```
res = OP (arglist)   |->   PRE1 (arglist)
                           and res = OP1 (arglist)
                      or
                           PRE2 (arglist)
                           and res = OP2 (arglist).
```

En utilisant la forme disjonctive (ii), on obtient :

```
(1)   op( Res, Arglist) :- ( pre1( Arglist) ,
                             op1( Res, Arglist) );
                           ( pre2( Arglist) ,
                             op2( Res, Arglist) ).
```

(On notera l'introduction de parenthèses supplémentaires).

alors qu'en utilisant (i), on obtient :

```
(2)   op( Res, Arglist) :-  prel( Arglist) ,
                            opl( Res, Arglist).

      op( Res, Arglist) :-  pre2( Arglist) ,
                            op2( Res, Arglist).
```

L'ordre d'évaluation des termes d'une disjonction n'a pas d'importance en RSL ; ce qui importe, c'est qu'une des opérations fournisse une réponse. Par contre, en Prolog, l'ordre doit être considéré car si opl conduit à une recherche infinie (boucle), op2 ne sera jamais envisagé.

Pour des raisons de rapidité de traduction et de facilité de celle-ci (notamment celle des explicitations longues et complexes), la forme (1) et l'introduction de parenthèses supplémentaires ont été choisies.

Une fois de plus, la contrainte d'utilisation est respecté car on a :

```
in( var,   gr.)          in ( gr.)
op( Res, Arglist) :- ( prel( Arglist) ,
                        out( gr.)

                       in ( var,   gr.)
                       opl( Res, Arglist) );
                       out( gr.,   gr.)

                        in ( gr.)
                      ( pre2( Arglist) ,
                        out( gr.)

                       in ( var,   gr.)
                       op2( Res, Arglist) ).
                       out( gr.,   gr.)


out( gr., gr.)
```

Dans la version de RSL que nous avons utilisée, les "ou" sont exclusifs alors qu'en Prolog, ils ne le sont pas. Nous sommes donc confrontés à un problème d'équivalence lors de la traduction. Puisque dans pareille situation, deux préconditions associées à des opérations distinctes dans la disjonction ne peuvent être satisfaites simultanément, on pourra utiliser la traduction

```
op( ...) :- ( pre1(...), !, op1(...) );
            ( pre2(...), !, op2(...) ).
```

L'emploi du cut (!) évite d'évaluer les autres préconditions si celle qui précède est satisfaite. Il permet d'améliorer l'efficacité du programme.

Le connecteur "forall" est une généralisation du connecteur "et". L'opération plus fine est appliquée sur chaque élément d'une suite guide pour définir un élément correspondant de la suite résultat. Ainsi, on aura l'explicitation suivante :

```
res = OP (arglist)  |->   forall e : IN (e, sg) :
                                    res' = OP' (e, ...)
```

où e est un élément de la suite guide apparaissant dans les arguments et res' est un élément de la suite résultat res.

La transformation consiste en :

1) mise en évidence de la suite guide comme argument de l'opération à expliciter.

```
suite-res = OP ( suite-guide, argl, ...)
  |->   forall e : IN (e, suite-guide) :
                   res' = OP' (e, argl, ...)
```

De manière alternative, cette mise en évidence peut se faire (si la suite guide n'est pas en argument, mais est un objet intermédiaire) au moyen d'une décomposition-with.

2) représentation des suites RSL par des listes Prolog.

3) transformation de la relation quantifiée universellement en une forme récursive.

Pour cela, deux procédures Prolog définies sur des listes sont nécessaires.

- vide(S) : procédure établissant si une suite S est vide.
- compsuite(S, Tete, Queue) : procédure exprimant la décomposition d'une suite S en son premier élément "Tête" et une sous-suite restante "Queue".

On obtient alors deux règles Prolog : l'une correspondant à la forme générale de la récurrence, l'autre au cas de base. Cette dernière vient en première position. Il s'agit d'un choix. L'ordre dans lequel on considère le fait et la règle n'a aucune importance car la procédure se terminera toujours du point de vue de la récursivité.

```
op( Suite_res, Suite_guide, Arg1, ...)
  :-  vide( Suite_res),
      vide( Suite_guide).

op( Suite_res, Suite_guide, Arg1, ...)
  :-  compsuite( Suite_guide, Tete_guide, Queue_guide),
      op'( Tete_res, Tete_guide, Arg1, ...),
      op( Queue_res, Queue_guide, Arg1, ...),
      compsuite( Suite_res, Tete_res, Queue_res).
```

Notons que les relations "vide" et "compsuite" expriment une relation entre un objet et ses éventuels composants.

L'ordre au sein de la deuxième règle est choisi afin d'obtenir un programme correct. Il s'agit de respecter le mode d'utilisation du programme. On a en effet :

```
in (   var,         gr.,         gr. )
 op( Suite_res, Suite_guide, Arg1, ...)

          in (    gr.,         var,          var )
 :-  compsuite( Suite_guide, Tete_guide, Queue_guide),
          out(    gr.,         gr.,          gr. )

     in (   var,         gr.,     gr. )
     op'( Tete_res, Tete_guide, Arg1, ...),
     out(   gr.,         gr.,     gr. )

     in(    var,         gr.,      gr. )
     op( Queue_res, Queue_guide, Arg1, ...),
     out(   gr.,         gr.,      gr. )

          in (     var,       gr.,       gr. )
     compsuite( Suite_res, Tete_res, Queue_res).
          out(     gr.,       gr.,       gr. )


out ( gr.,    gr.,     gr. )
```

Chaque appel de procédure respecte la contrainte de mode d'utilisation du programme. Tout comme pour comparg et compres, compsuite doit fonctionner dans les deux sens.

L'appel récursif mérite qu'on s'y arrête. Les valeurs en sortie seront fournies par les appels récursifs de la procédure "op". Au dernier appel récursif, la procédure aura la Suite_guide instanciée à la liste vide [ ]. La première règle sera alors choisie par le moteur d'inférence. On aura

```
in (   var,        gr.,      gr., gr.)
op( Suite_res, Suite_guide, Arg1, ...)

      in (    var )
  :-  vide( Suite_res),
      out(    gr. )

      in (    gr. )
     vide( Suite_guide).
      out(    gr. )


out (   gr.,     gr.,       gr., gr.)
```

L'application de la première règle va instancier la suite résultat courante. Le deuxième appel à compsuite dans la seconde règle assure l'instanciation du résultat courant à chaque sortie de la procédure op. La valeur de Queue_res à la sortie du premier appel recursif sera obtenue par restaurations successives des résultats intermédiaires instanciés.

Ces deux règles peuvent être mises sous la forme condensée suivante :

```
op( [], [], Arg1, ...).

op( [Tete_res|Queue_res], [Tete_guide|Queue_guide],
    Arg1,...)
  :-  op'( Tete_res, Tete_guide, Arg1, ...),
      op( Queue_res, Queue_guide, Arg1, ...).
```

La représentation des suites par des listes Prolog nous permet de remplacer compsuite (Suite, Tete, Queue) par :

Suite = [Tete | Queue]

Remplacer Suite_guide par [Tete_guide | Queue_guide] dans le membre de gauche revient à mettre en premier lieu dans le membre de droite Suite_guide = [Tete_guide | Queue_guide]. Peut-on le faire dans le cas de la décomposition de la suite résultat ? Oui, car Suite_res étant une variable , si on met

Suite_res = [Tete_res | Queue_res] au début, on obtient que Tete_res et Queue_res sont aussi des variables. On respecte bien la directionnalité in(var, ground) qui est en entrée. On a bien :

```
in (    var,      var,      gr.,       gr.,      gr.)
op( [Tete_res|Queue_res],[Tete_guide|Queue_guide],Arg1,...)

        in (    var,      gr.,      gr.)
    :-  op'( Tete_res, Tete_guide, Arg1, ...),
        out(    gr.,      gr.,      gr.)

        in (    var,      gr.,      gr.)
        op( Queue_res, Queue_guide, Arg1, ...).
        out(    gr.,      gr.,      gr.)


out(    gr.,      gr.,      gr.,       gr.,      gr.)
```

Une autre règle d'explicitation forall est aussi souvent utilisée.

res = OP (arglist) |-> forall e : IN (e, sg) and PRE(e) :
res' = OP' (e, ...)

où e est un élément de la suite guide apparaissant dans les arguments, res' est un élément de la suite résultat res et PRE est une précondition devant être vérifiée par e.

Il s'agit du cas d'une explicitation avec filtrage sur la suite guide. Le traitement du forall est soumis à une précondition qui n'est pas nécessairement satisfaite par tous les éléments de la suite guide. Il faut, dans ce dernier cas, rappeler l'opération op avec Queue_guide. Par application de la règle de transformation précédente, on aurait :

```
op( [], [], Arg1, ...).

op( [Tete_res|Queue_res], [Tete_guide|Queue_guide],
    Arg1,...)
  :-( pre( Tete_guide),
      op'( Tete_res, Tete_guide, Arg1, ...),
      op( Queue_res, Queue_guide, Arg1, ...) );
      op( [Tete_res|Queue_res],Queue_guide,Arg1,...).
```

Cependant, cette dernière ne fonctionne plus. Le problème se situe au niveau du dernier appel récursif dans le cas où la précondition n'est pas satisfaite par l'élément courant de la suite guide. On a l'appel : op([Tete_res|Queue_res], [], Arg1, ...). Prolog ne pouvant unifier [Tete|Queue] à la liste vide [], aucune des deux règles ne pourra être choisie et le moteur d'inférence Prolog va générer un échec. Il est donc nécessaire de modifier la forme condensée de façon à obtenir Suite_res à la place de [Tete_res|Queue_res] dans cet appel récursif. Pour cela, il s'agit de réinsérer la décomposition des suites dans le corps de la procédure. On obtient ainsi une nouvelle règle de transformation :

```
op( [], [], Arg1, ...).

op( Suite_res, Suite_guide, Arg1,...)
  :- Suite_guide = [Tete_guide|Queue_guide],
    ( pre( Tete_guide),
      op'( Tete_res, Tete_guide, Arg1, ...),
      op( Queue_res, Queue_guide, Arg1, ...),
      Suite_res = [Tete_res|Queue_res] );
      op( Suite_res, Queue_guide).
```

Notons que le choix de l'autre alternative pour la traduction du "ou" aurait fourni une solution plus élégante :

```
op( [], [], Argl, ...).

op( [Tete_res|Queue_res], [Tete_guide|Queue_guide],
    Argl,...)
  :- pre( Tete_guide),
     op'( Tete_res, Tete_guide, Argl, ...),
     op( Queue_res, Queue_guide, Argl, ...).

op( Suite_res, [Tete_guide|Queue_guide], Argl,...)
  :- op( Suite_res,Queue_guide,Argl,...).
```

Le cas d'une explicitation gardée se ramène à celui du connecteur "et". Ainsi, l'explicitation

```
res = OP (argl, arglist)    |->    PRE ( argl)
                                   and
                                   res = OP' (arglist)
```

sera traduite en

```
op( Res, Argl, Arglist) :- pre( Argl),
                           op'( Res, Arglist).
```

En RSL, l'ordre n'a pas d'importance. Par contre, en Prolog, il en a, notamment pour des raisons d'efficacité. On teste d'abord la précondition puis on effectue l'opération ; il est inutile de l'effectuer si la précondition n'est pas satisfaite.

A nouveau, la règle de traduction respecte la directionnalité, car on a :

```
in ( var, gr., gr.)          in ( gr.)
 op( Res, Argl, Arglist) :- pre( Argl),
                            out( gr.)

                            in ( var,   gr.)
                            op'( Res, Arglist).
                            out( gr.,   gr.)


out ( gr., gr., gr.)
```

3.1.2 La représentation des objets

Lors de la mise au point des règles de transformation des opérations non-terminales, nous avons été confrontées au problème de la traduction de l'univers en structures de données Prolog.
La représentation des données joue un rôle déterminant dans les contextes suivants :

- lorsqu'une opération de la bibliothèque est appliquée à des objets structurés. Par exemple, l'accès à un champ d'un produit cartésien.
- pour exprimer les relations entre un objet et ses composants.

Dans ces deux contextes, nous avons besoin de connaitre la structure des données pour la traduction des opérations en Prolog. La traduction de l'univers ne sera considérée que dans ce cadre.

Les objets en Prolog sont généralement représentés par des faits ou des termes. Les termes permettent de représenter tout objet structuré.
Un fait Prolog a la même structure qu'un terme mais il est traité différemment par le moteur d'inférence. Il est fixe dans le texte du programme.

Représenter les objets de l'univers par des faits est à rejeter. En effet, dans ce cas, les objets font partie intégrante du programme. Or le prototypage nécessite leur manipulation. Il faudrait alors des prédicats extra-logiques pour pouvoir les modifier (suppression de faits, ajout de faits, ...). Ce qui rend cette solution lourde et inadéquate.

Les objets seront donc désignés à l'aide de termes Prolog en utilisant le constructeur prédéfini de liste. La liste permet de représenter des termes composés ayant une arité variable.

Par exemple, [el1, el2, ..., eln] désignera une suite ou un n-uple de n champs

et [Tete | Queue] est une autre manière de désigner une suite.

Cette solution a pour inconvénient d'utiliser la même représentation dans des sens différents (pour un produit cartésien et une suite).

On aurait pu également définir un foncteur pour chaque constructeur de type.

Par exemple : pc(chp1, chp2).

suite(el1, el2, ..., eln).

Cette solution n'a pas été retenue car elle a perdu l'intérêt de l'arité variable.

3.1.3 La transformation d'une opération terminale

L'explicitation récursive des opérations en opérations plus simples se termine par une expression équationnelle définissant le résultat en termes d'une composition d'opérations de la bibliothèque de spécification de la forme :

res = BIBLIO_OP1 ... BIBLIO_OPn (arglist)

où n peut valoir 1.

Le cas particulier des préconditions et invariants doit être distingué, dans la mesure où ceux-ci s'explicitent au niveau terminal en prédicats du 1er ordre dans lesquels peuvent également intervenir des opérations de la bibliothèque. Leur traitement requiert éventuellement des choix spécifiques et peut ne pas être systématique.

Nous allons tout d'abord traiter les opérations terminales, ensuite les préconditions et les invariants.

Les procédures Prolog qui traduisent les opérations de la bibliothèque seront pré-programmées, une fois pour toutes, à partir de la spécification algébrique qui en est donnée dans la bibliothèque. Chaque opération de la bibliothèque donnera lieu à un prédicat de la forme

```
biblio_op( Res, Arglist).
```

Exemple : la procédure de l'opération IN (e, s) déterminant l'appartenance d'un élément à une suite sera donnée par

```
member( X, [X|_] ).
member( X, [_|L] ) :- member( X, L).
```

L'implémentation des opérations de la bibliothèque doit également respecter le mode d'utilisation du programme.

```
      in ( var,   gr.)
biblio_op( Res, Arglist).
      out( gr.,   gr.)
```

Néanmoins, cette traduction peut ne pas être aussi directe qu'il n'y parait. Il y a en effet deux types de problèmes :

- la formulation logique de certaines opérations,
- la combinaison d'opérations de la bibliothèque.

Certaines opérations de la bibliothèque donnent lieu à une formulation logique d'un ordre supérieur à 1 (certains composants de la structure sont des prédicats). Les procédures Prolog qui réalisent ces opérations doivent utiliser des primitives Prolog "extra-logiques".

C'est le cas de l'opération FILTER qui filtre une suite selon une propriété donnée, et dont la définition algébrique est :

```
FILTER (SEMPTY, p) = SEMPTY
FILTER (APPEND (s, e), p) =
                    if p(e) then APPEND (FILTER (s, p), e)
                            else FILTER (s, p).
```

où p est la propriété donnée.

Nous allons maintenant exposer le raisonnement suivi lors de l'implémentation de cette opération en Prolog.

La 1ère étape consiste à transformer la spécification algébrique sous forme relationnelle. La convention a été prise de mettre le résultat comme premier argument.

```
FILTER (SEMPTY, SEMPTY, p)
FILTER (APPEND (FILTER (s, p), e), APPEND (s, e), p) si p(e)
FILTER (FILTER (s, p),APPEND (s, e), p) si non p(e)
```

La définition algébrique étant récursive, la procédure Prolog générée doit l'être aussi. Le cas de base se traduit facilement en un fait avec la suite-résultat et la suite-argument instanciées à la liste vide, les autres paramètres éventuels étant représentés chacun par une variable anonyme puisqu'ils n'ont aucune influence sur le résultat.

Les suites RSL étant représentées sous forme de listes Prolog, l'ordre de considération des éléments de la suite a dû être modifié car il est plus facile de traiter le 1er élément d'une liste que le dernier. Aussi la décomposition de la suite argument APPEND(s, e) sera traduite par [élément | suite-restante]. De même pour la composition de la sous-suite résultat.

Se pose alors le problème de la propriété. Il fallait trouver un système afin qu'on puisse vérifier pour chaque élément de la suite qu'il satisfait bien cette dernière. Pour cela, nous avons utilisé la procédure "substitute" qui remplace chaque occurrence d'un terme par un autre dans un troisième. La procédure a 4 arguments : les deux premiers spécifient ce qui va être substitué et à quoi ; les deux derniers indiquent le terme dans lequel la substitution a lieu et ce terme après la substitution. L'analyse de la procédure "substitute" fait apparaître une procédure auxiliaire "subst_arg" qui réalise la substitution au niveau d'une occurrence. L'emploi de la procédure "substitute" permet l'appel du prédicat correspondant à la propriété avec la valeur de l'élément courant de la suite. Par le biais de la combinaison de "substitute" et de "goal", on obtient d'abord l'évaluation de la propriété et ensuite l'évaluation du prédicat filter. La présence du cut (!) est justifiée pour des raisons de performances : en effet, il n'est pas nécessaire d'essayer de resatisfaire "substitute" et "goal" en cas d'échec de l'appel récursif puisque l'on considère un seul élément de la liste à la fois. Notons que le "substitute" ne fonctionne que dans le sens considéré.

On obtient 3 règles correspondant aux 3 situations possibles :

- la liste vide
- l'élément vérifie la propriété
- l'élément ne vérifie pas la propriété.

Notons que nous obtenons un argument de plus que dans la définition algébrique. Le dernier argument représente en fait le terme qui va être substitué dans la propriété Pre.

La procédure Prolog correspondante se présente comme suit :

```
filter( [], [], _, _ ).

filter( [El | Sres], [El | Sarg], Pre, Var)
  :- substitute( El, Var, Pre, Goal), Goal, !,
     filter( Sres, Sarg, Pre, Var).

filter( Sres, [ El | Sarg], Pre, Var)
  :- filter( Sres, Sarg, Pre, Var).

/*-------------------------------------------------------------------*/
/* substitute( New, Old, Val, Newval) : remplacer Old par New */
/*                                      dans Val donne Newval */
/*-------------------------------------------------------------------*/

substitute( New, Old, Old, New) :- !.

substitute( New, Old, Val, Val) :- atomic( Val), !.

substitute( New, Old, Val, Newval)
  :- functor( Val, Fn, N),
     functor( Newval, Fn, N),
     subst_args( N, New, Old, Val, Newval).

/*-------------------------------------------------------------------*/

subst_args( 0,_,_,_,_) :- !.

subst_args( N, New, Old, Val, Newval)
  :- arg( N, Val, Oldarg),
     arg( N, Newval, Newarg),
     substitute( New, Old, Oldarg, Newarg),
     N1 is N-1,
     subst_args( N1, New, Old, Val,Newval).
```

Un problème se pose pour les opérations où les résultats d'une autre opération sont utilisés implicitement comme argument car les prédicats en Prolog sont du 1er ordre. Il faut donc expliciter les résultats intermédiaires.

Une solution possible serait de réécrire une expression sous forme de prédicat du 1er ordre en introduisant explicitement les résultats intermédiaires ; ainsi,

```
res = BIBLIO-OP1 (arg1, BIBLIO-OP2 (arg2))
```

serait traduit en

```
res = BIBLIO-OP1 ( arg1, resint)
with
resint = BIBLIO-OP2 ( arg2)
```

ou en Prolog

```
biblio_op2( Resint, Arg2),
biblio_op1( Res, Arg1, Resint).
```

Mais ceci représente une certaine lourdeur. C'est comme si des étapes d'explicitation en opérations plus simples et des résultats intermédiaires étaient rajoutés alors que le spécifieur avait estimé que ce n'était pas nécessaire pour la compréhension.

D'où l'idée d'écrire une procédure d'évaluation des expressions. Elle est appelée "l'évaluateur". Cette procédure aura la forme :

```
eval( Valeur, Expression).
```

Elle est utilisée chaque fois que l'on aura des opérations de la bibliothèque ou des combinaisons de celles-ci dans une explicitation d'opération du texte RSL. Recevant une expression constituée d'opérations de la bibliothèque ou d'une combinaison de celles-ci, la procédure "eval" fournit le résultat de l'application de ces opérations à leurs arguments.

Il s'agit d'une évaluation récursive de l'expression en profondeur. Le premier prédicat évalué est le plus profond. La condition d'arrêt est l'évaluation d'expressions finales, c'est-à-dire une variable ou une constante.

Le principe peut s'énoncer de la manière suivante :

- évaluation récursive de chacun des arguments de l'opération de la bibliothèque,
- appel au prédicat implémentant l'opération de la bibliothèque avec pour valeurs d'arguments les résultats de l'évaluation précédente. Ceci fournit la valeur finale pour l'appel courant.

L'algorithme récursif de l'évaluateur se présente comme suit :

```
règle d'arrêt : eval( X, X) :- functor( X, _, 0).
                               ( X = variable ou constante)

règle récursive : eval( Val, op( X, Y) )
                        :- eval( Vx, X),
                           eval( Vy, Y),
                           predicat_op( Val, Vx, Vy).
```

L'évaluateur est une généralisation d'un opérateur prédéfini Prolog : l'opérateur "is" qui force l'évaluation d'une expression arithmétique, ce qui fournit un résultat. La variable du terme de gauche, si elle est non instanciée, le sera à ce résultat et "is" réussit. Sinon, les valeurs obtenues sont comparées et "is" réussit ou échoue sur base du résultat du test.

Notre évaluateur généralise cela pour les opérateurs de la bibliothèque. Il possède aussi les mêmes limites que le "is" : il faut que tout soit instancié dans l'expression.

L'évaluateur répond bien à la contrainte de mode d'utilisation du programme, car on a :

```
   in ( var,   gr.)          in ( var, gr.)
eval( Val, op( X, Y) ) :-  eval( Vx,  X),
                           out( gr., gr.)

                           in ( var, gr.)
                          eval( Vy,  Y),
                           out( gr., gr.)

                                  in ( var, gr., gr.)
                          predicat_op( Val, Vx,  Vy).
                                  out( gr., gr., gr.)

   out ( gr.,   gr.)
```

Par le mode d'utilisation, la règle d'arrêt sera toujours l'évaluation d'une constante.

Toutes les opérations de la bibliothèque ont été introduites dans l'évaluateur afin de considérer toute combinaison de celle-ci comme une simple expression à évaluer.
Néanmoins, la primitive de sélection d'un élément d'un produit cartésien sur base du type de l'élément à selectionner et du n-uple (TUPLESEL) ne peut être implémentée une fois pour toutes dans l'évaluateur. Elle dépend en effet de la structure particulière de chaque produit cartésien. Pour chaque type d'objet de type produit cartésien, il faudra générer, dans l'évaluateur, autant de règles qu'il y a d'éléments dans le n-uple (nombre de règles = n).
Ayant un type d'objet produit cartésien de la forme

```
TYPE |-> CART-PROD [ TYPE1, TYPE2 ],
```

il faut compléter l'évaluateur de la façon suivante :

```
eval( X, tuplesel( type1, Y)) :- eval( Vy, Y),
                                 eval( [X, _], Vy).

eval( X, tuplesel( type2, Y)) :- eval( Vy, Y),
                                 eval( [_, X], Vy).
```

La première règle permet d'avoir accès au premier champ du produit cartésien et la seconde au second.

Une partie de l'évaluateur figure en Annexe 2 page 7.

La transformation des préconditions peut poser des problèmes car celles-ci peuvent être des prédicats du 1er ordre arbitrairement complexes comprenant par exemple, un quantificateur existentiel, ou une équivalence.
La mise sous forme de clause de Horn de telles formules nécessitera souvent une intervention du spécifieur. Ceci constitue une limite à l'automatisation.

Premièrement, il faut passer de formules bien formées à des clauses. Ce passage est automatisable mais se pose le problème de la skolemisation. Le traitement de la quantification existentielle est le suivant : il s'agit d'éliminer tous les quantificateurs existentiels en remplaçant chaque variable quantifiée existentiellement par un symbole de fonction ayant pour arguments toutes les variables, quantifiées universellement, dont la portée recouvre cette variable dans la formule de départ.

exemple : $\forall x \forall y \exists z : op1(x,y) \vee op2(x,z)$ devient

$\forall x \forall y \quad : op1(x,y) \vee op2(x, f(x,y))$.

Pour plus d'informations concernant la conversion sous forme de clause, on consultera, par exemple, [CLOCKSIN 81].

Une solution est de demander à l'utilisateur d'introduire une fonction de Skolem qui ait un sens dans le contexte du problème spécifié car la transformation automatique peut mener à une solution correcte mais peu lisible. Ceci correspond à effectuer une décomposition-with supplémentaire avec mise en évidence d'un problème intermédiaire.

Ensuite, nous devons nous restreindre aux clauses de Horn . Ce qui pose entre autres le problème de la traduction des équivalences.
Il faut, dans ce cas, réduire l'équivalence à une implication, et donc choisir un sens.

Enfin, le passage des clauses de Horn à Prolog peut poser des problèmes
Considérons la précondition suivante : $x \neq 0$. Sa traduction sous forme clausale donnera

precond(X) :- not(x = 0) & ....

c'est-à-dire qu'on utilisera la négation par échec de Prolog.

La négation de Prolog n'est pas une négation logique mais une négation par échec. Le but "not(p)" réussit si on n'est pas arrivé à établir "p". Selon l'emploi du "not", on peut obtenir des réponses fausses. En effet, il est dangereux d'affirmer que quelque chose est faux seulement parce qu'on est incapable de le prouver. L'utilisation du not implique qu'il faut tenir compte de l'ordre d'évaluation des règles. Illustrons ceci sur un exemple.

Ayant les faits et les règles suivantes

```
a.
r.
q :- a, !, b.      (1)
q :- true.         (2)
p :- not(q), r.
```

et la question ?- p.

Des deux règles candidates (1) et (2), le moteur d'inférence choisira la première. Aussi, on obtient que p réussit alors que le choix de la seconde règle aurait mené à l'échec.

En résumé, la transformation de ces préconditions reviendra, dans certains cas, à une forme de programmation nécessitant l'intervention de l'utilisateur de l'outil.

Les invariants ayant la même forme que les préconditions, le traitement est identique à celui que nous venons d'exposer.

3.1.4 Les règles de transformation (synthèse)

* Cas d'une décomposition structurelle -and :

énoncé du problème	univers du problème
res = OP(arg1, arg2, ...)	res : RES,
|->	
res1 = OP1(arg1, arg2, ...) and res2 = OP2(arg1, arg2, ...)	res1 : RES1, res2 : RES2 RES |-> CART-PROD[RES1, RES2]

```
                    |
                    |
                    V
```

traduction

```
op( [Res1, Res2], Arg1, Arg2, ...)
  :- op1( Res1, Arg1, Arg2, ...),
     op2( Res2, Arg1, Arg2, ...).
```

* Cas d'une décomposition fonctionnelle -with :

énoncé du problème

```
res = OP(arg1, arg2, ...)

|->

res = OP2(res1)
with
res1 = OP1(arg1, arg2, ...)
```

univers du problème

```
res : RES,

...
```

|
|
V

traduction

```
op( Res, Arg1, Arg2, ...)
  :- op1( Res1, Arg1, Arg2, ...),
     op2( Res, Res1).
```

* Cas d'une décomposition structurelle -forall :

énoncé du problème

```
suite-res
    = OP(suite-guide, arg1, ...)

|->

forall e :
   IN (e, suite-guide) :
   res' = OP'(e, arg1, ...)
```

univers du problème

```
suite-res : SRES, res' : RES,

suite-guide : SG,

e : I

SRES |-> SEQ [RES],
SG   |-> SEQ [I]
```

|
|
V

traduction

```
op( [], [], _, ...).

op( [Tete_res,Queue_res], [Tete_guide,Queue_guide],
    Arg1,...)
  :-  op'( Tete_res, Tete_guide, Arg1, ...),
      op( Queue_res, Queue_guide, Arg1, ...).
```

* Cas d'une décomposition fonctionnelle terminale :

énoncé du problème

```
res = OP( arg1, ...)

|->

res = BIBLIO-OP( res1, ..., resn)
with
   res1 = OP1( arg1, ...)
   and
   ...
   and
   resn = OPn( arg1, ...)
```

univers du problème

```
res : RES,

resi : I

...
```

traduction

```
op( Res, Arg1,...)
  :-  op1( Res1, Arg1, ...),
      ...
      opn( Resn, Arg1, ...),
      eval( Res, biblio_op( Res1, ..., Resn)).
```

* Cas d'une décomposition -ou :

énoncé du problème

```
res = OP(arg1, arg2, ...)

|->

PRE1( arg1, arg2, ...)
and
res = OP1(arg1, arg2, ...)
or
PRE2( arg1, arg2, ...)
and
res = OP2(arg1, arg2, ...)
```

univers du problème

```
res : RES,

res1 : RES1,

res2 : RES2,

RES -> UNION [RES1, RES2]
```

traduction

```
op( Res, Arg1, Arg2, ...)
  :- ( pre1( Arg1, Arg2, ...),
       op1( Res, Arg1, Arg2, ...) );
     ( pre2( Arg1, Arg2, ...),
       op2( Res, Arg1, Arg2, ...) ).
```

Remarques :

(1) L'interface entre l'utilisateur et le système n'étant pas spécifié lors de l'application du méta-algorithme, nous avons été obligées d'écrire nous-mêmes les procédures Prolog s'y rapportant. Aussi, il n'y a pas de règles de traduction concernant cette partie. Les procédures résultantes utilisent des prédicats extra-logiques (read, write, writeq, seeing, seen, tell, told, display, ...) ainsi que des appels systèmes.

(2) Nous avons justifié la validité des règles de transformation selon une directionnalité. Voyons ce que cela implique si l'on désire générer de jeux de tests, c'est-à-dire utiliser la réversibilité de Prolog et changer la directionnalité.
Le mode d'utilisation du programme devient

{ in (ground, var) } P { out (ground, ground) }.

Tout va bien jusqu'au moment où l'on arrive aux opérations de la bibliothèque (et l'évaluateur).
L'appel de biblio_op(Res, Arg) avec Res instancié et Arg variable ne fonctionne souvent pas, notamment lorqu'il s'agit d'opérations arithmétiques.
Les changements seront donc localisés si l'on désire la réversibilité. Ils se situent au niveau de l'implémentation des opérations de la bibliothèque.

(3) Les cas présentés ici sont les plus simples. En pratique, nous pouvons avoir des cas plus complexes vu la grande liberté laissée au spécifieur dans la méthode de spécification. Ainsi, les décompositions forall imbriquées dans d'autres structures,

en raison de la récursivité à introduire en PROLOG, nécessitent, lors de la traduction, l'introduction de résultats intermédiaires. Cela revient à introduire une étape d'explicitation supplémentaire dans les spécifications RSL. Ce problème sera illustré dans le chapitre 3 point 3.

(4) La traduction de la spécification d'une opération définie par une récurrence ne pose pas de problème mais le programme obtenu n'est pas nécessairement optimal.

Par exemple, le calcul de la factorielle.

```
res = FACT ( arg )
|->
   ( arg = 0
     and
     res = 1 )
or
     res = arg * FACT ( arg - 1 )
```

devient

```
fact( Res, Arg ) :- ( Arg = 0,
                      Res = 1 );
                    ( X is Arg - 1,
                      fact( Res_int, X ),
                      Res is Arg * Res_int ).
```

au lieu de

```
fact( 1, 0 ).

fact( Res, Arg ) :- X is Arg - 1,
                    fact( Res_int, X ),
                    Res is Arg * Res_int.
```

On voit que le fait sera inséré dans la règle récursive.

Notons cependant que l'utilisation de l'autre stratégie de traduction du "ou" (deux règles plutôt que le ";")

```
fact( Res, Arg ) :- Arg = 0,
                    Res = 1.

fact( Res, Arg ) :- X is Arg - 1,
                    fact( Res_int, X ),
                    Res is Arg * Res_int.
```

n'aurait nécessité qu'une réécriture de la première règle en fact(1, 0).

Ces règles de transformation ont été appliquées à la gestion des tiers, celle des contrats et celle des livraisons. Cette phase a permis de mettre en évidence la possibilité d'une automatisation partielle. Celle-ci va maintenant être présentée.

n'aurait nécessité qu'une réécriture de la première règle en fact(1, 0).

Ces règles de transformation ont été appliquées à la gestion des tiers, celle des contrats et celle des livraisons. Cette phase a permis de mettre en évidence la possibilité d'une automatisation partielle. Celle-ci va maintenant être présentée.

3.2 Automatisation du passage de RSL en PROLOG

Pour assurer la génération de prototypes PROLOG à partir de spécifications RSL, nous avons utilisé des outils développés dans le cadre du projet ALMA (Atelier Logiciel sur Machine Abstraite). Ce projet, mené aux F.N.D.P. en collaboration avec la société C.I.G. s.a. a pour but d'apporter une aide dans le développement et la maintenance de produits logiciels importants, impliquant de nombreuses personnes ("multi-personnes"), et faisant l'objet de nombreuses versions ("multi-versions"). On dispose actuellement d'un premier prototype du noyau de l'environnement ALMA [ALMA 86]. Ce noyau a été conçu pour rencontrer les objectifs suivants :

- l'intégration d'outils autonomes, grâce à un interface unique constitué par une structure de données commune à tous les outils.

- la couverture du cycle de vie complet d'un produit logiciel : spécification, conception, codage et maintenance.

- la neutralité vis-à-vis de méthodologies et de langages spécifiques. Les outils n'imposent ni méthodologie, ni langage particuliers ; ils sont adaptables à ceux de l'utilisateur. Cet objectif de neutralité est atteint grâce à deux niveaux d'environnement : un "méta-environnement" générique (paramétrable), et des environnements instanciés.

- la modularité de l'atelier, basée sur l'existence d'un certain nombre d'interfaces abstraits permettant de cacher de

l'information.

Le noyau de l'environnement ALMA offre deux fonctionalités de base :

- un support intégré pour la documentation du cycle de vie complet d'un projet logiciel, et
- un support pour la manipulation syntaxique de textes formels.

Dans le cadre de l'automatisation du passage d'un texte RSL à un programme PROLOG, c'est la seconde fonctionalité qui nous intéresse : la manipulation de textes formels. Nous allons présenter les outils disponibles à ce niveau et que nous avons employés, avant d'exposer les grands principes sous-jacents à notre programme de production semi-automatique d'un prototype PROLOG à partir d'une spécification RSL.

3.2.1 Outils utilisés

Les outils syntaxiques de manipulation de textes formels offerts par ALMA sont, comme nous l'avons mentionné, adaptables aux formalismes employés par les utilisateurs. Ces outils sont donc paramétrés sur la définition des syntaxes des formalismes utilisés. Pour manipuler des textes exprimés dans un formalisme donné, il faut donc instancier les outils pour ce formalisme. Pour cela, il suffit de définir la syntaxe du formalisme, dans un méta-langage de définition de formalisme (LDF), de style BNF, et ensuite, de fournir (une fois pour toutes) cette syntaxe à un "transformateur".

Celui-ci va générer les informations nécessaires à l'instanciation de l'analyseur syntaxique, du décompilateur et de l'éditeur syntaxique.

L'analyseur syntaxique est l'outil assurant le passage de la représentation externe (textuelle) d'un document à sa représentation interne sous forme d'arbre syntaxique ; le décompilateur est l'outil effectuant l'opération inverse, et l'éditeur syntaxique est l'outil réalisant des transformations d'arbres. Outre l'analyseur, le décompilateur et l'éditeur, une "boite à outils" est également disponible, contenant un ensemble de primitives de manipulation et de création d'arbres. Tous ces outils sont implémentés dans le langage C. Nous allons présenter plus en détails le Langage de Définition de Formalisme, la représentation interne des documents, l'analyseur syntaxique, le décompilateur, et l'éditeur syntaxique. Le transformateur, quant à lui ne sera pas détaillé. Il est décrit dans la documentation interne d'ALMA.

1. Le Langage de Définition de Formalisme : LDF

On distingue, pour un langage, sa syntaxe "concrète" et sa syntaxe "abstraite". La syntaxe concrète permet de déterminer les constructions syntaxiques admises dans le langage. C'est la syntaxe de la forme textuelle (externe) d'un texte. La syntaxe abstraite est la syntaxe concrète sans les mots et symboles réservés du langage. C'est la syntaxe de la forme interne (arborescente).

La définition LDF d'un formalisme contient une description enrichie, dans un style BNF, de la syntaxe concrète. Une description LDF est une suite de règles de production, le membre gauche d'une règle étant l'entité syntaxique définie, et le membre droit, la définition de cette entité.

La définition d'une entité syntaxique est soit une définition générique, soit une composition d'entités syntaxiques définissantes.

Une définition générique d'une entité syntaxique consiste en une définition des suites de caractères valables pour chaque occurrence de cette entité.

Exemple :

GEN ‹ident› ::= "[a-z] [a-z0-9]*" ;

signifiant que ‹ident› est une suite de caractères minuscules et de chiffres commençant par un caractère. "*" signifie 0, 1 ou plusieurs, tandis que "+" signifie 1 ou plusieurs.

La composition d'entités syntaxiques définissantes peut être répétitive, séquentielle ou sélective, et est complétée des mots et symboles réservés du formalisme. Ces définitions peuvent être récursives. Il existe, en LDF,

- des règles de liste pour exprimer une répétition

Exemple : L ‹list_obj_names› ::= ‹obj_name›+ "," ;

où "," est le séparateur des éléments de la liste.

- des règles d'arité 1 à 4, exprimant une composition séquentielle de (resp.) 1 à 4 entités définissantes.

Exemple de règle binaire :

```
‹and_explicitation›  ::=  ‹op_explicitation›
                          "and"
                          ‹op_explicitation› ;
```

Une règle unaire contient dans sa partie droite, soit une entité définissante, soit une liste d'entités définissantes alternatives. L'alternative peut s'exprimer de deux façons, selon qu'un noeud de l'arbre abstrait doit être associé avec l'entité définie ou pas. Dans le premier cas, la règle commence par le mot-clé ALT, tandis que dans le second, elle commence par SC (pour Sans Constructeur).

Exemple :

```
SC ‹op_explicitation› ::=  ‹terminal_explicitation›
                         | ‹and_explicitation›
                         | ‹or_explicitation› ;
```

- Des règles d'arité 0 permettant de définir les mots et symboles réservés du formalisme.

Exemple :

```
‹cut›  ::=  "!" ;
```

Le type d'information que l'on peut fournir, en LDF, ne se limite pas à ces indications purement grammaticales. On peut en

effet insérer des informations de formattage à effectuer dans la représentation externe, avant ou après un mot ou symbole réservé du formalisme, pour indiquer qu'il faut passer à la ligne, indenter, ... Ces informations seront reprises dans des tables qui guideront le processus de décompilation.

Exemple :

```
<probl_tree_op> ::= "RESULT :" <obj_name>
                    $ "ARGUMENTS :" <list_obj_names>
                    $ "EXPLICITATION OF I-O ASSERTION :" #
                    <explicitation_io_assertion> ;
```

où $ signifie passer à la ligne, et
signifie passer à la ligne et indenter.

La description LDF de la syntaxe RSL figure en Annexe 3 ; celle de PROLOG se trouve en Annexe 4.

2. La représentation interne des textes manipulés

La représentation interne d'un texte est une arborescence binaire, l'arbre abstrait, correspondant à la syntaxe abstraite du langage. Cette représentation fait donc abstraction des mots et symboles réservés et facilite la manipulation des textes. En effet, un texte vu sous forme d'une suite de caractères ne permet que des manipulations élémentaires, en termes de caractères. Par contre, il est aisé, dans un texte représenté par une arborescence, d'ajouter, modifier, ou supprimer un sous-arbre, accéder à telle partie du texte, ... Cette facilité sera illustrée dans le chapitre 3, point 3.2 (adéquation des outils d'ALMA).

Un noeud de l'arbre abstrait représente une entité syntaxique du langage.

Un arc orienté, d'un noeud père vers des noeuds fils représente une relation de définition (père = entité définie, fils = entités définissantes).

Une feuille représente une unité lexicale du langage (un générique) ou un noeud zéro-aire.

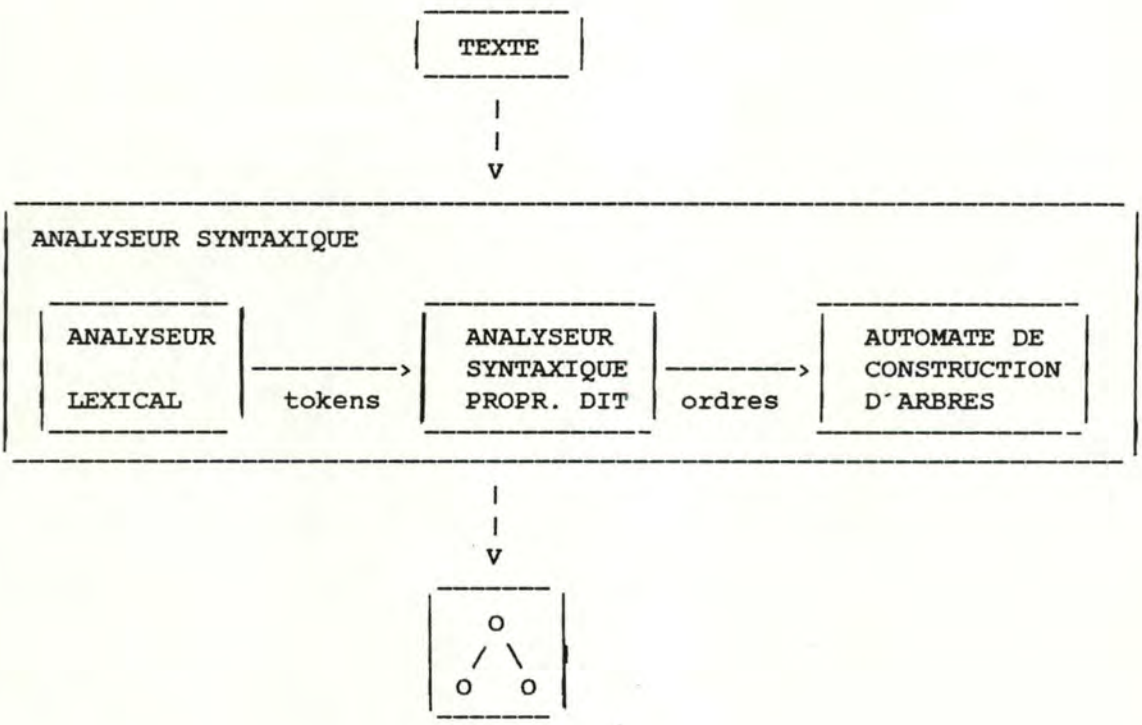
Chaque noeud de l'arbre abstrait se compose de cinq éléments : les références aux noeuds père, premier fils, frère droit, ainsi que le code du noeud (attribué par le transformateur), et une zone laissée libre.

3. L'analyseur syntaxique

L'analyseur syntaxique est l'outil assurant le passage de la forme externe d'un texte à sa forme interne. Il est composé d'un analyseur lexical, d'un analyseur syntaxique proprement dit, tous deux paramétrés sur le formalisme, ainsi que d'un automate de construction d'arbres.

L'analyseur lexical découpe le texte en unités lexicales appelées tokens ; l'analyseur syntaxique, quant à lui, vérifie que la suite des tokens identifiés est légale pour la syntaxe du formalisme. L'analyseur syntaxique effectue également des appels aux fonctions de construction d'arbres afin de créer la

représentation interne du texte. L'analyseur lexical et l'analyseur syntaxique proprement dit sont générés grâce aux outils LEX et YACC (respectivement) dont les inputs sont produits par le transformateur, à partir de la syntaxe LDF. L'outil d'analyse syntaxique peut donc se schématiser comme suit :



4. Le décompilateur

Le décompilateur est l'outil permettant le passage de la représentation interne d'un texte à sa représentation externe. La décompilation est un parcours d'arbre, avec affichage des mots et symboles réservés, des valeurs des génériques, et des éléments de formattage.

Le décompilateur se schématise comme suit :

5. L'éditeur syntaxique

L'éditeur syntaxique n'intervient pas directement dans la production automatique d'un prototype PROLOG, mais il facilite l'écriture des spécifications RSL. C'est pourquoi nous le présentons ici.

L'éditeur est un éditeur visuel syntaxique paramétré. Comme tout éditeur syntaxique, son unité de travail est l'unité syntaxique et non le caractère. L'éditeur travaille sur la représentation arborescente des textes. Un tel environnement permet de saisir, mettre à jour, afficher, ... les unités syntaxiques composant un document. De plus, il garantit la correction syntaxique des textes sous son contrôle, dégageant ainsi l'utilisateur de cet aspect.

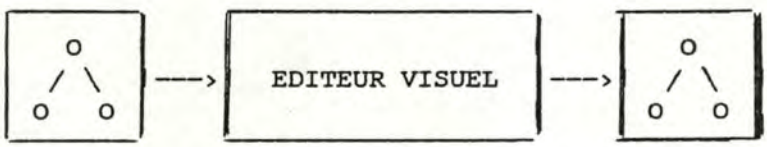
L'éditeur permet de transformer des arbres syntaxiques à partir de commandes externes. Pour cela, deux scénarios d'interaction avec l'utilisateur sont possibles.

Dans le premier scénario , un arbre abstrait est transformé à partir d'une commande de l'éditeur (insérer, modifier, supprimer), appliquée à une unité du texte figurant à l'écran, et à partir d'une nouvelle unité introduite par l'utilisateur, par exemple au moyen d'un éditeur conventionnel. L'analyse syntaxique est effectuée après l'introduction de la nouvelle entité.

Dans le second scénario , un arbre abstrait est transformé à partir d'une commande de l'éditeur (affiner, ...), appliquée à une unité syntaxique figurant à l'écran. Ceci a pour effet de faire apparaitre dans la fenêtre des menus les seuls types d'unités syntaxiques permises par la syntaxe du langage pour la transformation voulue. L'utilisateur choisit l'une d'elles, et l'éditeur réalise la transformation. Dans ce scénario, l'analyse syntaxique est "pré-mastiquée", puisque les options offertes sont toujours correctes.

Ces deux scénarios ne s'excluent pas mutuellement. On peut en effet passer de l'un à l'autre au cours d'une même session d'édition.

Le schéma de l'éditeur visuel est le suivant :



6. La boite à outils

La boite à outils contient un ensemble de primitives de manipulation et de création d'arbres. Ces primitives sont utilisables par tout outil effectuant des manipulations d'arbres abstraits.

On dispose, entre autres, de primitives permettant

- de chercher la première occurrence d'un (sous-)arbre dans un autre,
- de tester l'égalité de 2 (sous-)arbres
- d'obtenir le code d'un noeud à partir de son nom, et inversément,
- de créer un noeud zéroaire, unaire, ..., quaternaire, liste.

3.2.2 Principes du programme de traduction

Notre programme de prototypage reçoit en entrée un texte RSL spécifiant une application, et produit un programme PROLOG exécutable.

Les outils d'ALMA auxquels il est fait appel dans ce programme sont l'analyseur syntaxique, le décompilateur et la boite à outils (primitives de construction d'arbres). Les textes formels manipulés dans notre travail étant exprimés en RSL et en PROLOG, nous avons dû instancier les outils pour ces deux formalismes. Les définitions LDf de RSL (Annexe 3) et PROLOG (Annexe 4) ont été soumises au transformateur qui réalise les opérations nécessaires à l'instanciation des outils présentés précédemment.

La structure générale du programme de traduction est la suivante :

(1) Analyse syntaxique, par l'analyseur d'ALMA, du texte RSL à prototyper (considéré comme sémantiquement correct). Ceci fournit l'arbre abstrait correspondant au texte RSL.

(2) Traduction de l'arbre abstrait RSL en un arbre abstrait PROLOG, selon les règles de transformation énoncées au point 3.1 du chapitre 2.

(3) Décompilation, par le décompilateur d'ALMA, de l'arbre abstrait PROLOG, ce qui donne le texte PROLOG correspondant.

La deuxième étape consiste à parcourir l'arbre RSL, et créer un arbre PROLOG équivalent : rencontrant tel sous-arbre RSL, on génère tel sous-arbre PROLOG.

Nous allons présenter les parties de l'arbre abstrait RSL qu'il faut utiliser dans la traduction, ensuite, nous spécifierons à l'aide de RSL, quelques procédures, puis nous mentionnerons le procédé que nous avons utilisé pour construire le programme à partir de la spécification. Nous terminerons par l'état actuel du programme.

1. Parties utiles dans l'arbre abstrait RSL

L'arbre abstrait RSL contient deux branches principales : celle correspondant à l'énoncé du problème (la liste des blocs décrivant les opérations), et celle correspondant à l'univers du problème (la liste des blocs décrivant les types d'objet). Dans ces deux branches, seules certaines parties sont à considérer pour la traduction.

Les parties utilisées sont les suivantes :

A. Les assertions d'explicitation d'opérations. En effet, pour chaque sous-arbre correspondant à une assertion RSL de forme

```
res = OP (...)
|->
...
```

il faut générer un sous-arbre correspondant à une règle PROLOG de forme

```
op( Res, ...)   :-  ...
```

plus, dans le cas d'une explicitation forall, un sous-arbre correspondant à un fait PROLOG

B. Les assertions d'explicitation de préconditions : un sous-arbre correspondant à une explicitation de précondition donnera lieu à la création d'un sous-arbre représentant une règle PROLOG.

- Les assertions de typage des objets apparaissant dans les assertions d'explicitation d'opérations, ainsi que les assertions d'explicitation des types correspondants. Ces informations sont nécessaires pour traduire certaines explicitations d'opérations car elles permettent de déterminer la structure du résultat sur base du sous-arbre représentant l'explicitation du type correspondant.

Exemple :

Pour traduire

```
res-maj-tiers-ok = CREA-TIERS (..., argi, ...)
|->
fich-tiers-resultat = OP1 (..., argi, ...)
and
carton-tiers = OP2 (...)
```

en

```
crea_tiers( [Fich_tiers_resultat, Carton_tiers],
            ..., Argi, ...)
:-
op1( Fich_tiers_resultat, ...),
op2( Carton_tiers, ...).
```

on a besoin de l'assertion de typage

```
res_maj_tiers_ok     : RES-MAJ-TIERS-OK
fich-tiers-resultat  : FICH-TIERS
carton-tiers         : CARTON-TIERS
argi                 : TYPEi
...
```

et de l'assertion d'explicitation du type RES-MAJ-TIERS-OK

```
RES-MAJ-TIERS-OK
|->
CART-PROD [FICH-TIERS, CARTON-TIERS]
```

Remarque : un problème se posera dans le cas où plusieurs objets de même type apparaissent dans l'assertion de typage. Comment alors décomposer le résultat ?

Exemple :

Soient

- l'explicitation d'opération suivante :

```
res-maj-tiers-ok = CREA-TIERS (..., fich_tiers, ...)
|->
carton-tiers = OP1 (..., fich_tiers', ..., fich_tiers'', ...)
with
 fich_tiers'' = OP2 ( ..., fich_tiers', ...)
 with
 fich-tiers' = OP1 (..., argi, ...)
```

- l'assertion de typage

```
res_maj_tiers_ok              : RES-MAJ-TIERS-OK
fich-tiers, fich_tiers',
fich_tiers''                  : FICH-TIERS
carton-tiers                  : CARTON-TIERS
...
```

- et l'assertion d'explicitation du type RES-MAJ-TIERS-OK

```
RES-MAJ-TIERS-OK
|->
CART-PROD [FICH-TIERS, CARTON-TIERS]
```

Il est impossible dans cet exemple de trancher entre les décompositions de résultat [Fich_tiers, Carton_tiers], [Fich_tiers', Carton_tiers], ou [Fich_tiers'', Carton_tiers].

4. Les assertions d'explicitation de type sont également utilisées pour compléter la procédure PROLOG implémentant les opérations de la bibliothèque de spécification. En effet, comme nous l'avons déjà mentionné, l'une (au moins) de ces opérations ne peut être implémentée une fois pour toutes. Il s'agit de l'opération de sélection d'un élément dans un objet de type produit cartésien : élément = TUPLESEL (TYPE-ELEMENT, objet). Pour chaque explicitation d'un type produit cartésien, il faudra donc générer autant de règles PROLOG qu'il y a d'éléments dans le produit cartésien.

Exemple :

```
DDE-DE-MODIFICATION-TIERS
|->
CART-PROD [INDICATIF-TIERS, ELEMENTS-A-MODIFIER]
```

donnera lieu à

```
eval( Res, tuplesel( INDICATIF-TIERS, X))
    :- eval ([Res, _], X).

eval( Res, tuplesel( ELEMENTS-A-MODIFIER, X))
    :- eval ([_, Res], X).
```

2. Spécification RSL des principales procédures du programme

Dans cette section figurent les blocs de spécification des procédures suivantes :

- traitement de l'énoncé du problème : TRT-LIST-OPER-BLOCK
- traitement d'un bloc de l'énoncé : TRT-OPER-BLOCK
- traitement d'une assertion d'explicitation d'opération : TRT-EXPLICITATION-IO-ASSERTION
- traitement d'une op-expression (partie gauche d'une règle d'explicitation) : TRT-OP-EXPRESSION
- traitement d'une explicitation d'opération (partie droite d'une règle d'explicitation) : TRT-OP-EXPLICITATION
- traitement d'une explicitation and : TRT-AND-EXPL
- traitement d'une explicitation or : TRT-OR-EXPL
- traitement d'une explicitation with : TRT-WITH-EXPL
- traitement d'une explicitation gardée : TRT-GUARDED-EXPL
- traitement d'une explicitation terminale : TRT-TERMINAL-EXPL
- traitement d'une explicitation entre parenthèses : TRT-PARENTH-EXPL
- traitement d'une assertion d'explicitation forall : TRT-FORALL-IO-EXPLICITATION
- génération du résultat PROLOG : TRT-RES
- décomposition du résultat PROLOG (en [Res1, ..., Resn] ou [Tres | Qres]) : DECOMP-RES

Figurent aussi les représentations graphiques

- des blocs de spécification précités,
- de la partie de l'univers correspondante.

Les procédures de bas niveau sont quant à elles spécifiées de manière informelle dans le texte du programme en Annexe 5.

SPECIFICATION OF OPERATION TRT-LIST-OPER-BLOCK

LEXICON :

OBJECTIVE :

traduction d'un sous-arbre RSL correspondant à une liste de blocs spécifiant des opérations (list-oper-block) en un sous-arbre correspondant à un programme PROLOG

OBJECTS :

program-prlg : sous-arbre correspondant à un programme PROLOG

énoncé : sous-arbre correspondant à une liste de blocs spécifiant des opérations

univers : sous-arbre correspondant à une liste de blocs spécifiant les types

oper-block : sous-arbre RSL correspondant à un bloc de spécification d'opération

OPERATIONS :

TRT-OPER-BLOCK :

traduction d'un sous-arbre RSL correspondant à un bloc de spécification d'opération en un sous-arbre PROLOG correspondant à une procédure PROLOG

OUTLINE :

TYPES :

program-prlg : PROGRAM-PRLG

énoncé : LIST-OPER-BLOCK

univers : LIST-OBJ-TYPE-BLOCK

oper-block : OPER-BLOCK

procedure-prlg : PROGRAM-PRLG

OPERATIONS :

TRT-LIST-OPER-BLOCK : LIST-OPER-BLOCK
X LIST-OBJ-TYPE-BLOCK
-> PROGRAM-PRLG

TRT-OPER-BLOCK : OPER-BLOCK
X LIST-OBJ-TYPE-BLOCK
-> PROGRAM-PRLG

FORMAL SPEC :

RESULT : program-prlg

ARGUMENTS : énoncé, univers

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

program-prlg
= TRT-LIST-OPER-BLOCK (énoncé, univers)

|->

for all oper-block :

IN (énoncé, oper-block) :

procédure-prlg

= TRT-OPER-BLOCK (oper-block, univers)

SPECIFICATION OF OPERATION TRT-OPER-BLOCK

LEXICON :

OBJECTIVE :

traduction d'un sous-arbre RSL correspondant à un bloc spécifiant une opération en un sous-arbre correspondant à une procédure PROLOG

OBJECTS :

procédure-prlg : sous-arbre PROLOG correspondant à une procédure

oper-block : sous-arbre RSL correspondant à un bloc de spécification d'opération

univers : sous-arbre correspondant à une liste de blocs spécifiant les types

expl-io-assertion : sous-arbre RSL correspondant à une assertion d'explicitation

assertions-de-typage : sous-arbre RSL correspondant à la liste des assertions de typage

OPERATIONS :

TRT-EXPLICITATION-IO-ASSERTION : traduction d'un sous-arbre RSL correspondant à une assertion d'explicitation en un sous-arbre correspondant à une procédure PROLOG

TUPLESEL : opération de la bibliothèque sélectionnant un élément dans un objet de type produit cartésien

OUTLINE :

TYPES :

```
procédure-prlg     : PROGRAM-PRLG
oper-block         : OPER-BLOCK
univers            : LIST-OBJ-TYPE-BLOCK
expl-io-assertion  : EXPLICITATION-IO-ASSERTION
assertions-de-typage : LIST-OUTLI-OBJ
```

OPERATIONS :

```
TRT-OPER-BLOCK  :  OPER-BLOCK
                   X LIST-OBJ-TYPE-BLOCK
                   -> PROGRAM-PRLG

TRT-EXPLICITATION-IO-ASSERTION :
                   EXPLICITATION-IO-ASSERTION
                   X LIST-OUTLI-OBJ
                   X LIST-OBJ-TYPE-BLOCK
                   -> PROGRAM-PRLG

TUPLESEL : in specbase
```

FORMAL SPEC :

```
RESULT : procédure-prlg

ARGUMENTS : oper-block, univers

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

  procédure-prlg  = TRT-OPER-BLOCK ( oper-block,
                                     univers )

  |->

  procédure-prlg

       = TRT-EXPLICITATION-IO-ASSERTION
          ( expl-io-assertion,
            assertions-de-typage, univers )

  with

   expl-io-assertion
    = TUPLESEL (EXPL-IO-ASSERTION,
                TUPLESEL (FORMAL-SPEC-OP,
                          TUPLESEL (BODY-OPER-BLOCK,
                                    oper-block)))

   and

   assertions-de-typage

    = TUPLESEL (LIST-OUTLI-OBJ,
                TUPLESEL (OUTLINE-OP,
                          TUPLESEL (BODY-OPER-BLOCK,
                                    oper-block)))
```

SPECIFICATION OF OPERATION TRT-EXPLICITATION-IO-ASSERTION

LEXICON :

OBJECTIVE :

traduction d'un sous-arbre RSL correspondant à une assertion d'explicitation d'opération (explicitation_io_assertion) en un sous-arbre PROLOG correspondant à une procédure

OBJECTS :

procedure_prlg : sous-arbre correspondant à une procédure PROLOG

explicitation_io_assertion : sous-arbre RSL correspondant à une assertion d'explicitation d'opération

assertions-de-typage : sous-arbre RSL correspondant aux assertions de typage des objets de l'explicitation d'opération

univers : sous-arbre RSL correspondant à l'univers du problème

op-expression : sous-arbre RSL correspondant à une expression d'opération RSL (res = OP (...))

explicitation_operation : sous-arbre RSL correspondant à une explicitation d'opération (partie droite de |->)

tete : sous-arbre PROLOG correspondant à un prédicat PROLOG (tête de procédure)

queue : sous-arbre PROLOG correspondant à un body (corps de procédure)

OUTLINE :

TYPES :

```
procedure-prlg              : PROGRAM-PRLG

explicitation_io_assertion  : EXPL-IO-ASSERTION

assertions-de-typage        : LIST-OUTLI-OBJ

univers          :  LIST-OBJ-TYPE-BLOCK

tete             :  PREDICATE

queue            :  BODY

explicitation-operation : OP-EXPLICITATION

op-expression    :  OP-EXPRESSION
```

OPERATIONS :

```
TRT-FORALL-IO-ASSERION : EXPL-IO-ASSERTION
                         X LIST-OUTLI-OBJ
                         X LIST-OBJ-TYPE-BLOCK
                         ->  BODY

TRT-OP-EXPRESSION  : OP-EXPRESSION
                     X EXPL-IO-ASSERTION
                     -> PREDICATE-PRLG

TRT-OP-EXPLICITATION : EXPL-IO-ASSERTION
                       -> BODY
```

FORMAL SPEC :

RESULT : procedure_prlg

ARGUMENTS : explicitation_io_assertion,
assertions-de-typage,
univers

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
   procedure-prlg
    = TRT-EXPLICITATION-IO-ASSERTION
      ( explicitation_io_assertion,
        assertions-de-typage, univers )
   |->

   ( TYPE-OF ( EXPLICITATION-FORALL,
               explicitation-operation)
     and
     procedure-prlg
       = TRT-FORALL-IO-ASSERTION (...))

 or

   ( procedure-prlg = TUPLEFORM ( tete, queue)

     with

      tete = TRT-OP-EXPRESSION(op-expression,
                               assertions-de-typage,
                               univers, explicitation-
                               operation )

      and

      queue = TRT-OP-EXPL(explicitation-operation)

      with

        op-expression
         = TUPLESEL (OP-EXPRESSION,
                     explicitation_io_assertion)

        and

        explicitation-operation
         =  TUPLESEL (OP-EXPL,
                     explicitation_io_assertion))
```

OPERATIONS :

TRT-FORALL-IO-EXPL : traduction d'un sous-arbre RSL correspondant à une explicitation forall en un sous-arbre PROLOG correspondant à un "body"

TRT-OP-EXPRESSION : traduction d'un sous-arbre RSL correspondant à une expression d'opération (res = OP (...)) en un sous-arbre PROLOG correspondant à une tête de procédure

TRT-OP-EXPL : traduction d'un sous-arbre RSL correspondant à une explicitation d'opération (partie droite de |->) en un sous-arbre PROLOG correspondant à un corps de procédure

TUPLEFORM : opération de la bibliothèque de spécification, construisant un produit cartésien

APPEND : opération de la bibliothèque de spécification, ajoutant un élément à une suite

SEMPTY : opération de la bibliothèque de spécification, construisant une suite vide

TYPE-OF : prédicat de la bibliothèque indiquant si l'élément donné en second argument set du type donné en premier argument

SPECIFICATION OF OPERATION TRT-OP-EXPRESSION

LEXICON :

OBJECTIVE :

traduction d'un sous-arbre RSL correspondant à une expression d'opération (partie gauche de |->) en un sous-arbre PROLOG correspondant à un prédicat

OBJECTS :

predicat : sous-arbre correspondant à un prédicat PROLOG

op-expression :
sous-arbre RSL correspondant à la partie gauche d'une explicitation d'opération (res = OP (...))

op-explicitation :
sous-arbre RSL correspondant à la partie droite d'une explicitation d'opération

assertions-de-typage :
sous-arbre RSL correspondant à la liste des assertions de typage des objets apparaissant dans l'explicitation d'opération

univers : sous-arbre RSL correspondant à l'univers du problème

nom-de-predicat : sous-arbre PROLOG correspondant au nom du prédicat PROLOG à générer

liste-termes-prolog : sous-arbre PROLOG correspondant à la liste des arguments à générer

res-prlg : sous-arbre PROLOG correspondant au résultat PROLOG à générer (premier des arguments du prédicat à générer)

type-res-rsl : type du résultat de l'opération à traduire

OUTLINE :

TYPES :

```
predicat              : PREDICATE
op-expression         : OP-EXPRESSION
assertions-de-typage  : LIST-OUTLI-OBJ
univers               : LIST-OBJ-TYPE-BLOCK
nom-de-predicat       : VARIABLE
liste-termes-prolog,
LISTE-ARGS-PRLG       : LISTE-TERMS-PRLG
res-prlg              : TERM-PRLG
type-res-rsl          : CHARST
op-explicitation      : OP-EXPLICITATION
```

OPERATIONS :

```
TRT-OP-EXPRESSION : OP-EXPRESSION
                    X LIST-OUTLI-OBJ
                    X LIST-OBJ-TYPE-BLOCK
                    ->  PREDICATE

TRT-NOM-PREDICAT : OBJ-NAME -> VARIABLE

TRT-RES : OBJ-NAME X CHARST
          X LIST-OUTLI-OBJ
          X LIST-OBJ-TYPE-BLOCK
          -> TERME-PRLG

TRT-LIST-ARGS : LIST-ARG-NAMES
                -> LIST-TERMS-PRLG

CHERCHE-TYPE : OBJ-NAME X LIST-OUTLI-OBJ
               ->  CHARST
```

```
TUPLEFORM   : in specbase
TUPLESEL    : in specbase
APPEND-B    : in specbase
```

FORMAL SPEC :

RESULT : predicat

ARGUMENTS : op-expression,
assertions-de-typage,
univers, op-explicitation

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
predicat
 = TRT-OP-EXPRESSION (op-expression,
                      assertions-de-typage,
                      univers,op-explicitation)
|->

predicat = TUPLEFORM (nom-de-predicat,
                      liste-termes-prlg)

with

 nom-de-predicat
 = TRT-NOM-PREDICAT(TUPLESEL(OPER-NAME,
                             op-expression)))

and

( liste-termes-prlg = APPEND-B(res-prlg,
                               liste-args-prlg)
  with
   res-prlg
    = TRT-RES ( TUPLESEL (OBJ-NAME,
                          op-expression),
                type-res-rsl,
                assertions-de-typage,
                univers, op-explicitation)
   and
   liste-args-prlg
    = TRT-LIST-ARGS (TUPLESEL (LIST-ARG-NAMES,
                               op-expression))
   with
     type-res-rsl
      = CHERCHE-TYPE (TUPLESEL (OBJ-NAME,
                                op-expression),
                      assertions-de-typage ))
```

liste-args-prlg : sous-arbre PROLOG
correspondant à une liste
d'arguments PROLOG

OPERATIONS :

TRT-NOM-PREDICAT :
traduction d'un sous-arbre correspondant
à un nom d'objet RSL (ex : arg) en
un sous-arbre correspondant à une
variable PROLOG (ex : Arg)

TRT-RES :
traduction d'un sous-arbre correspondant
à un nom d'objet RSL (le résultat de
l'opération) en un sous-arbre corres-
pondant à une variable ou à une
liste PROLOG (si le résultat doit
être décomposé)

TRT-LIST-ARGS :
traduction d'un sous-arbre correspondant
à une liste de noms d'objets RSL en
un sous-arbre correspondant à une
liste de variables PROLOG

CHERCHE-TYPE : recherche du type du résultat
dans les assertions d'explicitation

APPEND-B : opération de la bibliothèque
ajoutant un élément en début
de liste

TUPLEFORM : opération de la bibliothèque
construisant un objet de type produit
cartésien

TUPLESEL : opération de la bibliothèque
sélectionnant un élément dans
un objet de type produit cartésien

SPECIFICATION OF OPERATION TRT-RES

LEXICON :

OBJECTIVE :

traduction d'un sous-arbre correspondant à un nom d'objet RSL (le nom du résultat à traduire tel quel ou à décomposer) en un sous-arbre PROLOG correspondant, soit à une variable, soit à une liste

OBJECTS :

res-prlg :
sous-arbre correspondant soit à une variable PROLOG, soit à une liste de variables PROLOG
res-rsl :
nom de résultat d'une opération
type-res-rsl :
type du résultat RSL à traduire
assertions-de-typage :
liste des assertions de typage des objets apparaissant dans l'explicitation d'opération
univers : liste des blocs de spécification des objets
op-explicitation :
sous-arbre correspondant à une explicitation d'opération

OPERATIONS :

DECOMP-RESULTAT : décomposition de résultat en une liste PROLOG

TRT-OBJ-NAME :
traduction d'un sous-arbre object-name RSL en un sous-arbre variable PROLOG

PRECONDITIONS :

RESULTAT-A-DECOMPOSER :
prédicat vrai si le résultat doit être décomposé

OUTLINE :

TYPES :

```
res-prlg      : TERM-PRLG
res-rsl       : CHARST
type-res-rsl  : CHARST
assertions-de-typage : LIST-OUTLI-OBJ
univers       : LIST-OBJ-TYPE-BLOCK
op-expl       : OP-EXPLICITATION
```

OPERATIONS :

```
TRT-RES : CHARST X CHARST X LIST-OUTLI-OBJ
          X LIST-OBJ-TYPE-BLOCK
          X OP-EXPLICITATION
               -> LISTE-PRLG

DECOMP-RESULTAT :
     CHARST  X  CHARST  X  LIST-OUTLI-OBJ
   X LIST-OBJ-TYPE-BLOCK
   X OP-EXPLICITATION
          ->. LISTE-PRLG

TRT-OBJ-NAME :  OBJ-NAME -> VARIABLE
```

PRECONDITION :

```
RESULTAT-A-DECOMPOSER : CHARST X CHARST
                        X LIST-OUTLI-OBJ
                        X LIST-OBJ-TYPE-BLOCK
                        X OP-EXPLICITATION
                        -> BOOLEAN
```

FORMAL SPEC :

RESULT : res-prlg

ARGUMENTS : res-rsl, type-res-rsl,
assertions-de-typage,
univers, op-expl

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
res-prlg = TRT-RES ( res-rsl, type-res-rsl,
                     assertions de typage,
                     univers, op-expl )

|->

( RESULTAT-A-DECOMPOSER(type-res-rsl,
                        assertions-de-typage,
                        univers, op-expl)
  and res-prlg
    = DECOMP-RESULTAT (res-rsl, type-res-rsl,
                       assertions-de-typage,
                       univers, op-expl ) )

or

  res-prlg = TRT-OBJ-NAME ( res-rsl )
```

SPECIFICATION OF OPERATION DECOMP-RESULTAT

LEXICON :

OBJECTIVE :

création d'un sous-arbre correspondant
à une liste PROLOG de variables

OBJECTS :

res-prlg :
sous-arbre correspondant à une
liste PROLOG de variables
([Elt1, ..., Eltn])
res-rsl : nom de résultat
assertions-de-typage :
assertions de typage des objets
d'une explicitation d'opération
univers :
liste des blocs de spécification
des types
type-res-rsl : type du résultat
explicitation-type-res :
explicitation de la structure du
type du résultat
op-expl : sous-arbre RSL correspondant à
une explicitation d'opération

OPERATIONS :

TRT-RES :
création du résultat prolog

DECOMP-CART-PROD :
création du résultat prolog
correspondant à un résultat RSL
de type "produit-cartésien"

CONSULT-UNIVERS :
recherche de l'explicitation de la structure
d'un type d'objet

IN, TUPLESEL, TYPE-OF : in specbase

PRECONDITION :

RESULTAT-A-DECOMPOSER :
prédicat indiquant si le résultat prolog
à générer est à décomposer sous
forme de liste

OUTLINE :

TYPES :

res-prlg : LISTE-PRLG

res-rsl : CHARST

assertions-de-typage : LIST-OUTLI-OBJ

univers : LIST-OBJ-TYPE-BLOCK

type-res-rsl : CHARST

explicitation-type-res : TYPE-EXPLICITATION

op-explicitation : OP-EXPLICITATION

OPERATIONS :

TRT-RES : CHARST X CHARST X LIST-OUTLI-OBJ
X LIST-OBJ-TYPE-BLOCK
X OP-EXPLICITATION
-> LISTE-PRLG

DECOMP-CART-PROD : LIST-OUTLI-OBJ
X TYPE-EXPLICITATION
-> LISTE-PRLG

CONSULT-UNIVERS : CHARST X LIST-OBJ-TYPE-BLOCK
-> TYPE-EXPLICITATION

PRECONDITIONS :

RESULTAT-A-DECOMPOSER : CHARST X LIST-OUTLI-OBJ
X LIST-OBJ-TYPE-BLOCK
X OP-EXPLICITATION
-> BOOLEEN

FORMAL SPEC :

RESULT : res-prlg

ARGUMENTS : res-rsl, assertions-de-typage,
univers, type-res-rsl,
op-explicitation

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
res-prlg
   = DECOMP-RESULTAT ( res-rsl, type-res-rsl,
                       assertions-de-typage,
                       univers, op-explicitation )
  |->
   ( ( TYPE-OF (REUSE-EXPLICITATION,
                explicitation-type-res)
       or
       TYPE-OF (SPECIALIZE-EXPLICITATION,
                explicitation-type-res) )
     and
     res-prlg = TRT-RES ( res-rsl,
                          explicitation-type-res,
                          assertions-de-typage,
                          univers,
                          op-explicitation )
   )
  or
   ( TYPE-OF (CART-PROD-EXPLICITATION,
              explicitation-type-res)
     and
     res-prlg
       = DECOMP-CART-PROD(assertions-de-typage,
                          explicitation-type-res)
   )
  or
   ( TYPE-OF (SEQ-EXPLICITATION,
              explicitation-type-res)
     and
     res-prlg = "[ Tres | Qres ]"
   )

   with

     explicitation-type-res
        = CONSULT-UNIVERS (type-res-rsl, univers)
```

```
EXPLICITATION OF PRECONDITIONS :

  RESULTAT-A-DECOMPOSER ( type-res-rsl,
                          assertions-de-typage,
                          univers,
                          op-explicitation )
   |->

    ( TYPE-OF ( REUSE-EXPLICITATION,
                explicitation-type-res)  )

   or

    ( TYPE-OF ( SPECIALIZE-EXPLICITATION,
                explicitation-type-res)  )

   or

    ( TYPE-OF ( CART-PROD-EXPLICITATION,
                explicitation-type-res)
      and
      ∀ type : IN (type, explicitation-type-res) :
      ∃ obj, i : TUPLESEL (TYPE,
                           ITH (assertions-de-typage, i)
                    = type  )

   or

    ( TYPE-OF (SEQ-EXPLICITATION,
               explicitation-type-res)
      and
      TYPE-OF (FORALL-EXPLICITATION, op-explicitation)

   with
        explicitation-type-res
          = CONSULT-UNIVERS (type-res-rsl, univers)
```

SPECIFICATION OF OPERATION TRT-OP-EXPL

LEXICON :

OBJECTIVE :

traduction d'un sous-arbre RSL correspondant à une explicitation d'opération (op-explicitation) (partie droite de |->) en un sous-arbre correspondant à un corps de procédure PROLOG

OBJECTS :

body : sous-arbre body PROLOG

op-expl : sous-arbre op-explicitation RSL

OPERATIONS :

TRT-TERMINAL-EXPL : traduction d'un sous-arbre RSL "explicitation terminale" (nombre, constante, expr.arithmétique, ...) en un sous-arbre "body" PROLOG

TRT-AND-EXPL : traduction d'un sous-arbre RSL "explicitation and" en un sous-arbre "body" PROLOG

TRT-OR-EXPL : traduction d'un sous-arbre RSL "explicitation or" en un sous-arbre "body" PROLOG

TRT-WITH-EXPL : traduction d'un sous-arbre RSL "explicitation with" en un sous-arbre "body" PROLOG

TRT-GUARDED-EXPL : traduction d'un sous-arbre RSL "explicitation guardée" en un sous-arbre "body" PROLOG

TRT-PARENTH-EXPL : traduction d'un sous-arbre RSL "parenth-explicitation" en un sous-arbre "body" PROLOG

OUTLINE :

TYPES :

body : BODY

op-expl : OP-EXPLICITATION

OPERATIONS :

TRT-OP-EXPL : OP-EXPLICITATION -> BODY

TRT-TERMINAL-EXPL : TERMINAL-EXPLICITATION -> BODY

TRT-AND-EXPL : AND-EXPLICITATION -> BODY

TRT-OR-EXPL : OR-EXPLICITATION -> BODY

TRT-WITH-EXPL : WITH-EXPLICITATION -> BODY

TRT-GUARDED-EXPL : GUARDED-EXPLICITATION -> BODY

TRT-PARENTH-EXPL : PARENTH-EXPLICITATION -> PARENTH-BODY

TYPE-OF : in specbase

FORMAL SPEC :

RESULT : body

ARGUMENTS : op-expl

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
body = TRT-OP-EXPL (op-expl)

|->

   ( TYPE-OF ( TERMINAL-EXPL, op-expl )
     and
     body = TRT-TERMINAL-EXPL (op-expl) )
  or
   ( TYPE-OF ( AND-EXPL, op-expl )
     and
     body = TRT-AND-EXPL (op-expl) )
  or
   ( TYPE-OF ( OR-EXPL, op-expl )
     and
     body = TRT-OR-EXPL (op-expl) )
  or
   ( TYPE-OF ( WITH-EXPL, op-expl )
     and
     body = TRT-WITH-EXPL (op-expl) )
  or
   ( TYPE-OF ( GUARDED-EXPL, op-expl )
     and
     body = TRT-GUARDED-EXPL (op-expl) )
  or
   ( TYPE-OF ( PARENTH-EXPL, op-expl )
     and
     body = TRT-PARENTH-EXPL (op-expl) )
```

SPECIFICATION OF OPERATION TRT-TERMINAL-EXPL

LEXICON :

OBJECTIVE :

traduction d'un sous-arbre RSL correspondant à une explicitation terminale (nom d'objet, nombre, texte, expression arithmétique, terme entre parenthèses) en un sous-arbre body PROLOG

OBJECTS :

res : sous-arbre "body" PROLOG

terminal-expl : sous-arbre "explicitation terminale" RSL

obj_name : sous-arbre "object-name" RSL

terme-rsl : sous-arbre "terme" RSL

operateur : opérateur PROLOG

terme-prl : sous-arbre "terme" RSL

OPERATIONS :

TRT-OBJ-NAME : traduction d'un sous-arbre "object-name" RSL en un sous-arbre variable PROLOG

TRT-NUMBER : traduction d'un sous-arbre "number" RSL en un sous-arbre "number" PROLOG

TRT-ARITHM-EXPR : traduction d'un sous-arbre "expression arithmétique" RSL en un sous-arbre "terme infixé" PROLOG

TRT-PARENTH-TERM : traduction d'un sous-arbre "terme entre parenthèses" RSL en un sous-arbre "parenth-body" PROLOG

TRT-TEXTE : traduction d'un sous-arbre "texte" RSL en un sous-arbre "atome entre quotes" PROLOG

TUPLEFORM, TUPLESEL, TYPE-OF : opérations de la bibliothèque

OUTLINE :

TYPES :

res : BODY

terminal-expl : TERMINAL-EXPLICITATION

obj_name : OBJ-NAME

terme-rsl : TERM-RSL

operateur : CHARST

terme-prl : TERM_PRL

OPERATIONS :

TRT-TERMINAL-EXPLICITATION : TERMINAL-EXPLICITATION -> BODY

TRT-OBJ-NAME : OBJ-NAME -> VARIABLE

TRT-NUMBER : NUMBER -> NUMBER

TRT-TEXTE : NUMBER -> QUOTED-ATOM

TRT-ARITHM-EXPR : ARITHM-EXPR -> INFIX-TERM

TRT-PARENTH-TERM : PARENTH-TERM -> PARENTH-TERM

TUPLEFORM, TUPLESEL, TYPE-OF : in specbase

FORMAL SPEC :

RESULT : res

ARGUMENTS : terminal-expl

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
res = TRT-TERMINAL-EXPL (terminal-expl)

|->

res = TUPLEFORM (T-OBJ-NAME (obj-name),
                 operateur, terme-prl )
with

obj-name = TUPLESEL ( OBJ-NAME,
                      terminal-expl)
and

( ( TYPE-OF ( OBJ-NAME, terme-rsl)
    and
    operateur = "=" )
    and
    terme-prl = TRT-OBJ-NAME (terme-rsl)
 or
  ( TYPE-OF ( NUMBER, terme-rsl)
    and
    operateur = "is"
    and
    terme-prl = TRT-NUMBER ( terme-rsl) )
 or
  ( TYPE-OF ( TEXTE, terme-rsl)
    and
    operateur = "="
    and
    terme-prl = TRT-TEXTE ( terme-rsl) )
 or
  ( TYPE-OF ( ARITH-EXPR, terme-rsl)
    and
    operateur = "is"
    and
    terme-prl = TRT-ARITHM-EXPR ( terme-rsl) )
 or
  ( TYPE-OF ( PARENTH-TERM, terme-rsl)
    and
    operateur = "is"
    and
    terme-prl = TRT-PARENTH-TERM (terme-rsl))
 with
  terme-rsl = TUPLESEL ( TERM, terminal-expl)
```

SPECIFICATION OF OPERATION TRT-AND-EXPL

LEXICON :

OBJECTIVE :

traduction d'un sous-arbre RSL correspondant à une "explicitation and"
(res1 = OP1 (...)
and
res2 = OP2 (...))
en un sous-arbre correspondant à une conjonction PROLOG
(op1(Res1,...), op2(Res2,...))

OBJECTS :

conj : sous-arbre "conjonction" PROLOG

and-explicitation :
sous-arbre "and-explicitation" RSL

op-expl1 : sous-arbre "op-explicitation" RSL (première partie de l'explicitation and)
op-expl2 : sous-arbre "op-explicitation" RSL (seconde partie de l'explicitation and)

body1 : sous-arbre "body" PROLOG (première partie de la conjonction à générer)
body2 : sous-arbre "body" PROLOG (seconde partie de la conjonction à générer)

OPERATIONS :

TRT-OP-EXPL :
traitement d'un sous-arbre RSL correspondant à une explicitation d'opération (partie droite de |->)

TUPLESEL : opération de la bibliothèque sélectionnant un élément dans un produit cartésien

OUTLINE :

TYPES :

conj : CONJUNCTION

body1 : BODY1
body2 : BODY2

and-explicitation : AND-EXPLICITATION

op-expl1 : OP-EXPL1
op-expl2 : OP-EXPL2

OPERATIONS :

TRT-AND-EXPL : AND-EXPLICITATION -> CONJUNCTION

TRT-OP-EXPL : OP-EXPLICITATION -> BODY

TUPLESEL : in specbase

FORMAL SPEC :

RESULT : conj

ARGUMENTS : and-explicitation

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

conj = TRT-AND-EXPL (and-explicitation)

|->

body1 = TRT-OP-EXPL (op-expl1)

and

body2 = TRT-OP-EXPL (op-expl2)

with

op-expl1 = TUPLESEL (OP-EXPL1, and-explicitation)

and

op-expl2 = TUPLESEL (OP-EXPL2, and-explicitation)

SPECIFICATION OF OPERATION TRT-OR-EXPL

LEXICON :

OBJECTIVE :

traduction d'un sous-arbre RSL correspondant à une "explicitation or"
(res1 = OP1 (...)
or
res2 = OP2 (...))
en un sous-arbre correspondant à une disjonction PROLOG
(op1(Res1,...) ; op2(Res2,...))

OBJECTS :

disj : sous-arbre "disjonction" PROLOG

or-explicitation :
sous-arbre "explicitation or" RSL

op-expl1 : sous-arbre "op-explicitation" RSL (première partie de l'explicitation or)
op-expl2 : sous-arbre "op-explicitation" RSL (seconde partie de l'explicitation or)

body1 : sous-arbre "body" PROLOG (première partie de la disjonction à générer)
body2 : sous-arbre "body" PROLOG (seconde partie de la disjonction à générer)

OPERATIONS :

TRT-OP-EXPL :
traitement d'un sous-arbre RSL correspondant à une explicitation d'opération (partie droite de |->)

TUPLESEL : opération de la bibliothèque sélectionnant un élément dans un produit cartésien

OUTLINE :

TYPES :

conj : CONJUNCTION

body1 : BODY1
body2 : BODY2

or-explicitation : OR-EXPLICITATION

op-expl1 : OP-EXPL1
op-expl2 : OP-EXPL2

OPERATIONS :

TRT-OR-EXPL : OR-EXPLICITATION
-> DISJUNCTION

TRT-OP-EXPL : OP-EXPLICITATION
-> BODY

TUPLSEL : in specbase

FORMAL SPEC :

RESULT : disj

ARGUMENTS : or-explicitation

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

disj = TRT-OR-EXPL (or-explicitation)

|->

body1 = TRT-OP-EXPL (op-expl1)

and

body2 = TRT-OP-EXPL (op-expl2)

with

op-expl1 = TUPLESEL (OP-EXPL1, or-explicitation)

and

op-expl2 = TUPLESEL (OP-EXPL2, or-explicitation)

SPECIFICATION OF OPERATION TRT-WITH-EXPL

LEXICON :

OBJECTIVE :

traduction d'un sous-arbre correspondant à une explicitation with RSL en un sous-arbre correspondant à une conjonction PROLOG

OBJECTS :

conj : sous-arbre "conjonction" PROLOG

with-explicitation :
sous-arbre "explicitation with" RSL

body1 : sous-arbre "body" PROLOG (première partie de la conjonction à générer)

body2 : sous-arbre "body" PROLOG (seconde partie de la conjonction à générer)

OPERATIONS :

TRT-OP-EXPL :
traitement d'un sous-arbre RSL correspondant à une explicitation d'opération (partie droite de |->)

TUPLSEL :
opération de la bibliothèque sélectionnant un élément dans un produit cartésien

OUTLINE :

TYPES :

conj	: CONJUNCTION
with-explicitation	: WITH-EXPLICITATION
op-expl1	: OP-EXPL1
op-expl2	: OP-EXPL2
body1	: BODY1
body2	: BODY2

OPERATIONS :

TRT-WITH-EXPL	: WITH-EXPLICITATION -> CONJUNCTION
TRT-OP-EXPL	: OP-EXPLICITATION -> BODY
TUPLESEL	: in specbase

FORMAL SPEC :

RESULT : conj

ARGUMENTS : with-explicitation

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

conj = TRT-WITH-EXPL (with-explicitation)

|->

res1 = TRT-OP-EXPL (op-expl2)

and

res2 = TRT-OP-EXPL (op-expl1)

with

op-expl1
= TUPLESEL (OP-EXPL1, with-explicitation)

and

op-expl2
= TUPLESEL (OP-EXPL2, with-explicitation)

SPECIFICATION OF OPERATION TRT-GUARDED-EXPL

LEXICON :

OBJECTIVE :

traduction d'un sous-arbre correspondant à une explicitation gardée RSL en un sous-arbre correspondant à une conjonction PROLOG

OBJECTS :

res : sous-arbre correspondant à une conjonction PROLOG

guarded-explicitation : sous-arbre correspondant à une explicitation gardée RSL

body1 : première partie de la conjonction PROLOG

body2 : deuxième partie de la conjonction PROLOG

predicate-rsl : sous-arbre correspondant à un prédicat RSL

op-expl : sous-arbre RSL correspondant à une explicitation d'opération

OPERATIONS :

TRT-OP-EXPL : traitement d'un sous-arbre RSL correspondant à une explicitation d'opération (partiedroite de |->)

TRT-PREDICAT : traitement d'un sous-arbre correspondant à un prédicat RSL

TUPLESEL : in specbase

OUTLINE :

TYPES :

res : CONJUNCTION

guarded-explicitation : GUARDED-EXPLICITATION

body1 : BODY1

body2 : BODY2

predicat-rsl : PREDICATE

op-expl : OP-EXPLICITATION

OPERATIONS :

TRT-GUARDED-EXPL : GUARDED-EXPLICITATION -> CONJUNCTION

TRT-OP-EXPL : OP-EXPLICITATION -> BODY

TRT-PREDICAT : PREDICATE -> BODY

TUPLESEL : in specbase

FORMAL SPEC :

RESULT : res

ARGUMENTS : guarded-explicitation

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
res = TRT-GUARDED-EXPL(guarded-explicitation)
|->
body1 = TRT-PREDICAT ( predicat-rsl )
and
body2 = TRT-OP-EXPL ( op-expl )
with
  predicat-rsl
    = TUPLESEL (PREDICATE,
                guarded-explicitation)
  and
  op-expl
     = TUPLESEL (OP-EXPL,
                 guarded-explicitation)
```

SPECIFICATION OF OPERATION TRT-PARENTH-EXPL

LEXICON :

OBJECTIVE :

traduction d'un sous-arbre correspondant à une explicitation RSL entre ( ) en un sous-arbre correspondant à un parenth-body PROLOG

OBJECTS :

res : sous-arbre correspondant à un parenth-body PROLOG

parenth-explicitation : sous-arbre correspondant à une explicitation RSL entre ( )

OPERATIONS :

TRT-OP-EXPL : traduction d'un sous-arbre RSL correspondant à une explicitation d'opération (partie droite de |->) en un sous-arbre PROLOG correspondant à un "body"

OUTLINE :

TYPES :

res : PARENTH-BODY

parenth-explicitation : PARENTH-EXPLICITATION

OPERATIONS :

TRT-PARENTH-EXPL : PARENTH-EXPLICITATION -> PARENTH-BODY

TRT-OP-EXPL : OP-EXPLICITATION -> BODY

FORMAL SPEC :

RESULT : res

ARGUMENTS : parenth-explicitation

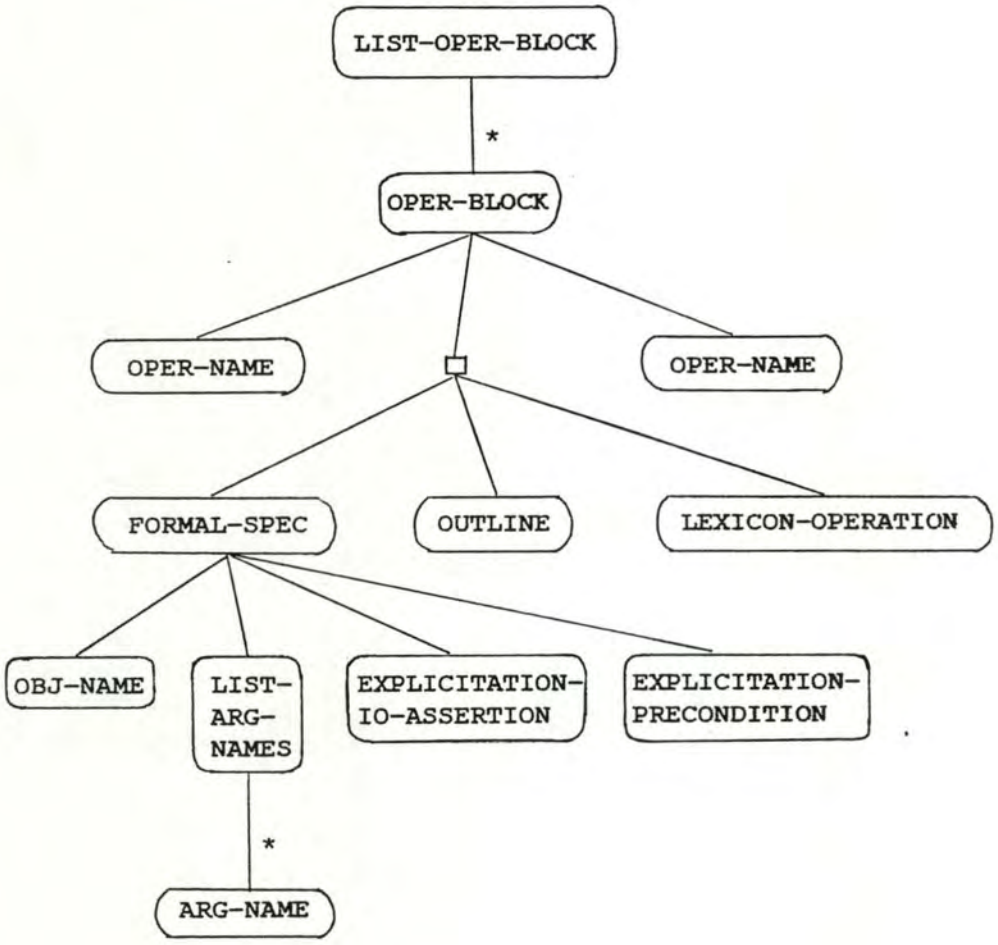
EXPLICITATION OF INPUT-OUTPUT ASSERTION :

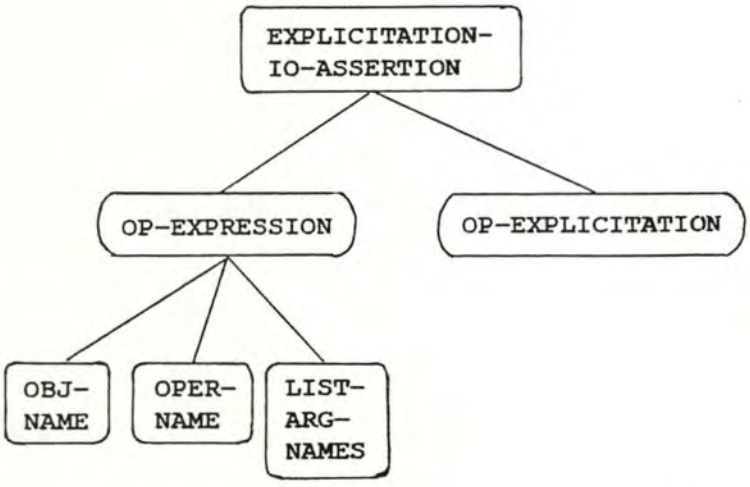
res = TRT-PARENTH-EXPL (parenth-explicitation)

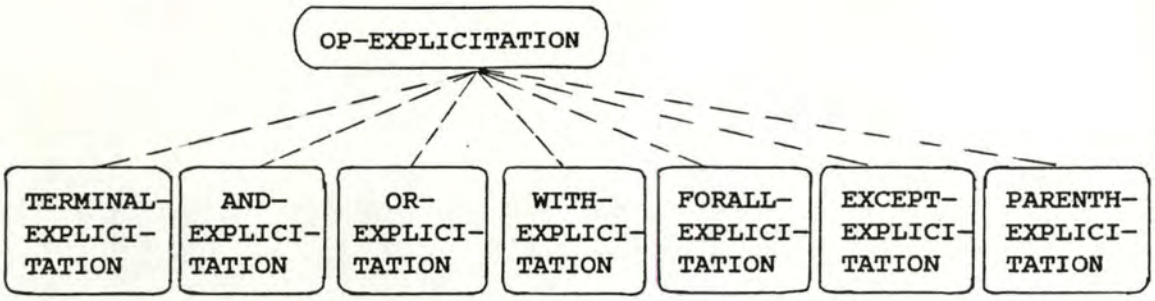
|->

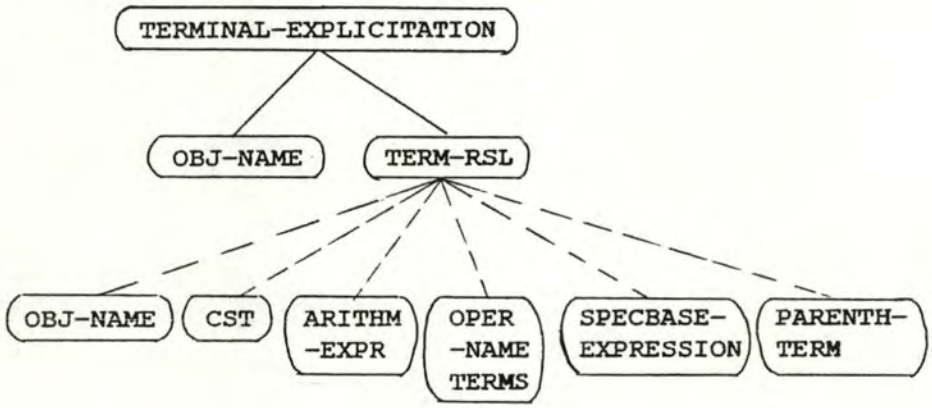
res = TRT-OP-EXPL (parenth-explicitation)

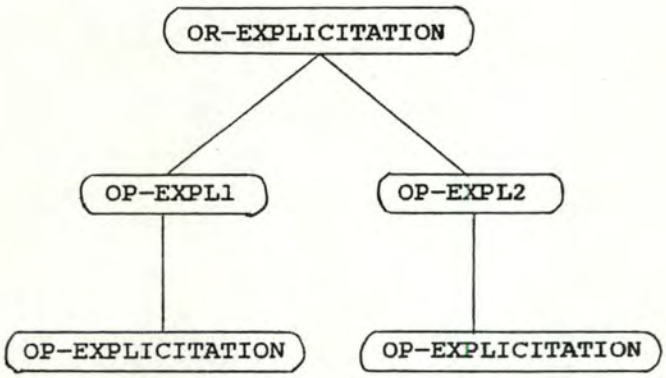
Représentation graphique d'une partie de l'univers :

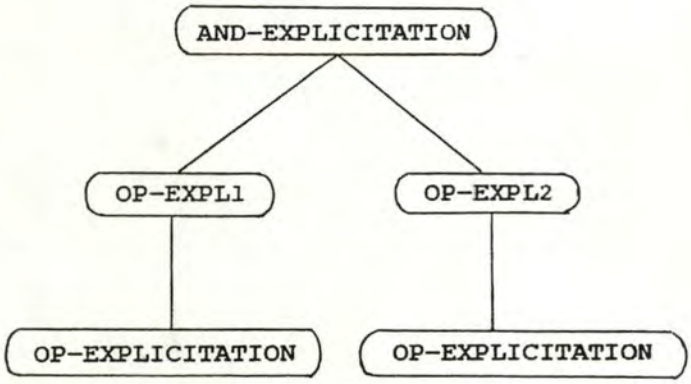


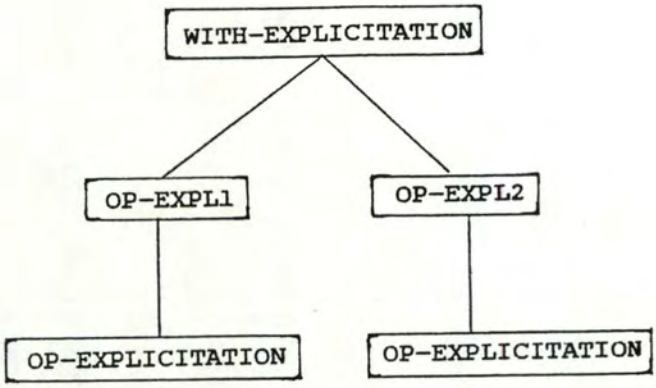


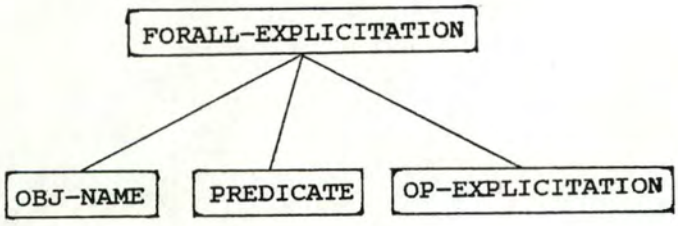

OP-EXPLICITATION
TERMINAL-EXPLICI-TATION
AND-EXPLICI-TATION
OR-EXPLICI-TATION
WITH-EXPLICI-TATION
FORALL-EXPLICI-TATION
EXCEPT-EXPLICI-TATION
PARENTH-EXPLICI-TATION

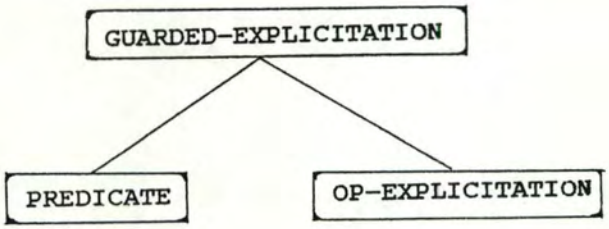

TERMINAL-EXPLICITATION
OBJ-NAME
TERM-RSL
OBJ-NAME
CST
ARITHM-EXPR
OPER-NAME TERMS
SPECBASE-EXPRESSION
PARENTH-TERM

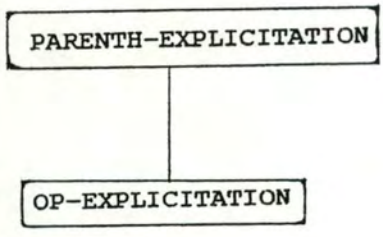

OR-EXPLICITATION
OP-EXPL1
OP-EXPL2
OP-EXPLICITATION
OP-EXPLICITATION

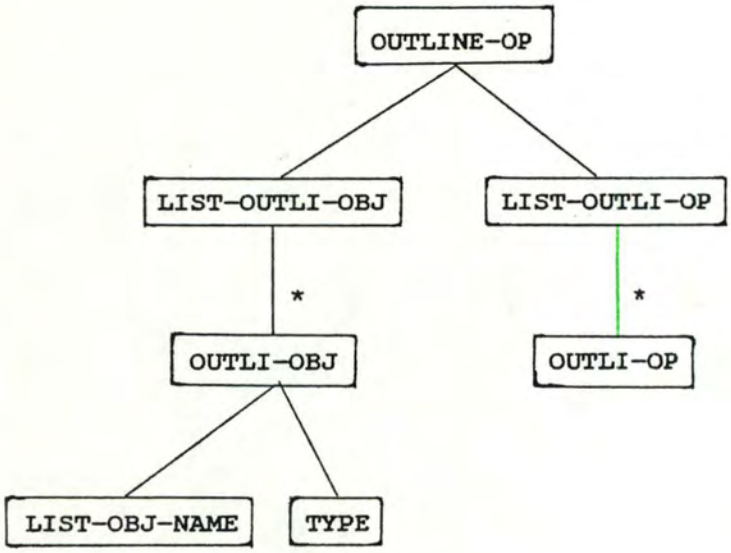

AND-EXPLICITATION
OP-EXPL1
OP-EXPL2
OP-EXPLICITATION
OP-EXPLICITATION

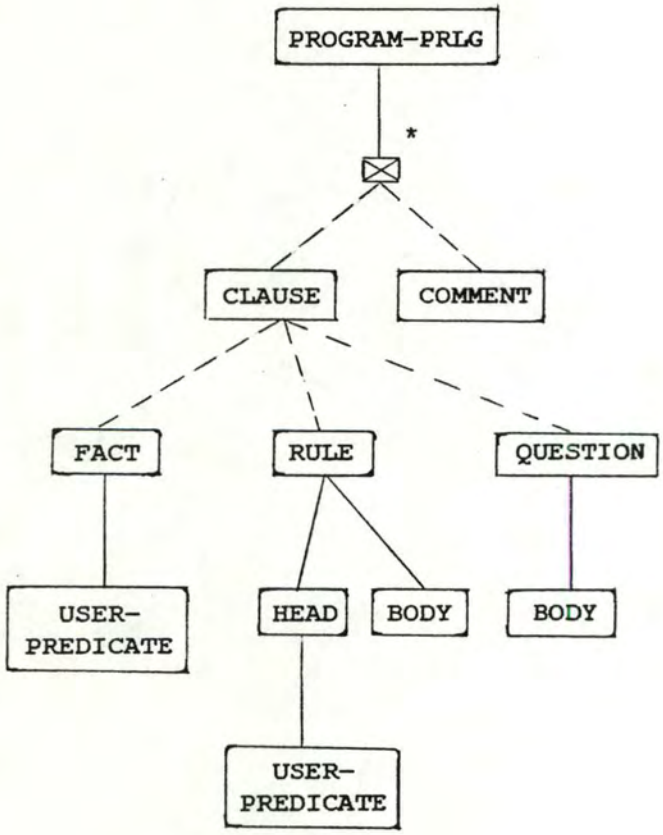

WITH-EXPLICITATION
OP-EXPL1
OP-EXPL2
OP-EXPLICITATION
OP-EXPLICITATION

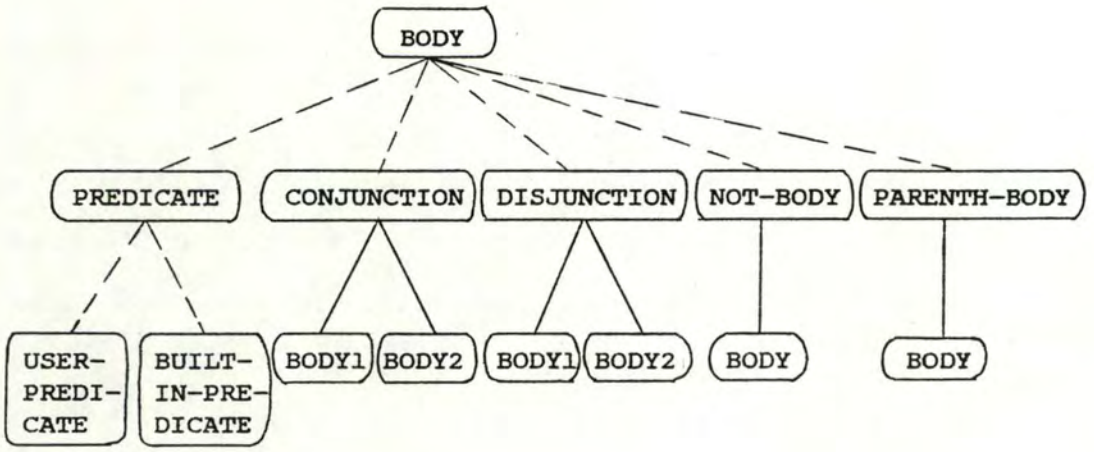

FORALL-EXPLICITATION
OBJ-NAME
PREDICATE
OP-EXPLICITATION

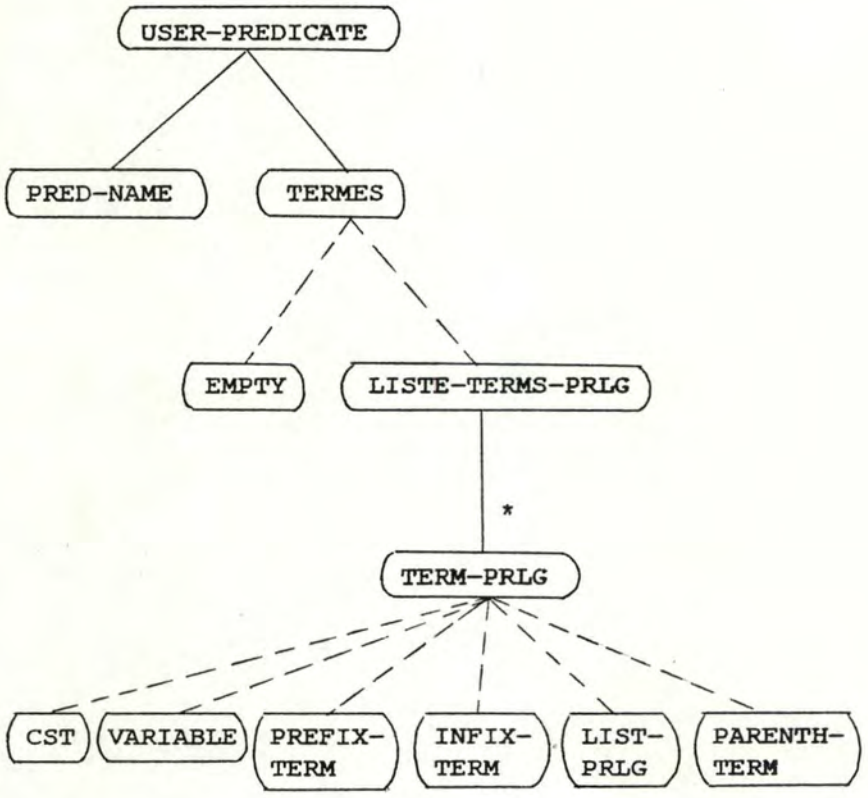

GUARDED-EXPLICITATION
PREDICATE
OP-EXPLICITATION


PARENTH-EXPLICITATION
OP-EXPLICITATION

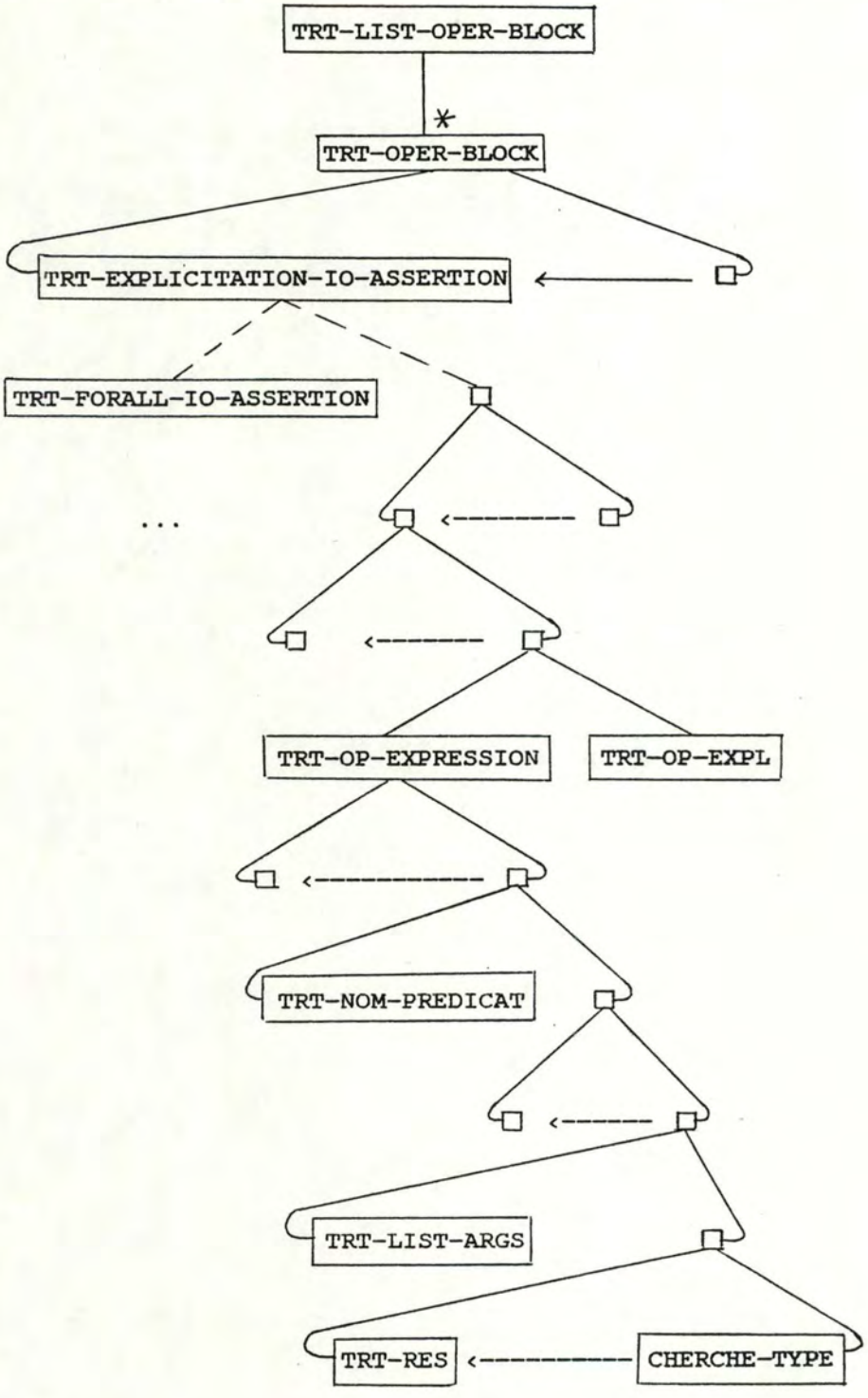

OUTLINE-OP
LIST-OUTLI-OBJ
LIST-OUTLI-OP
*
*
OUTLI-OBJ
OUTLI-OP
LIST-OBJ-NAME
TYPE

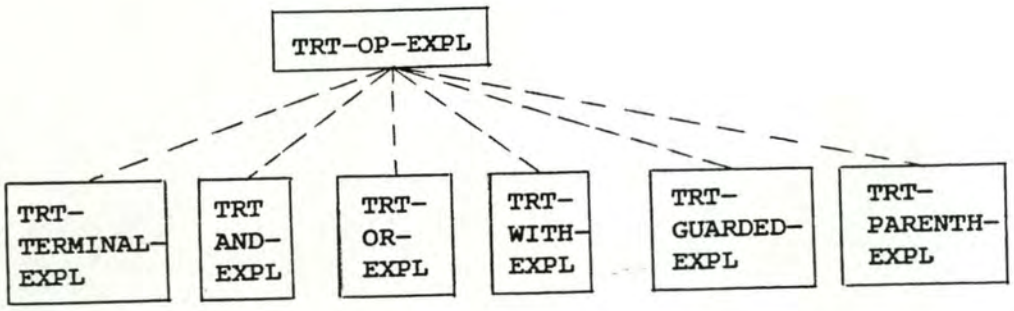

PROGRAM-PRLG
*
CLAUSE
COMMENT
FACT
RULE
QUESTION
USER-
PREDICATE
HEAD
BODY
BODY
USER-
PREDICATE

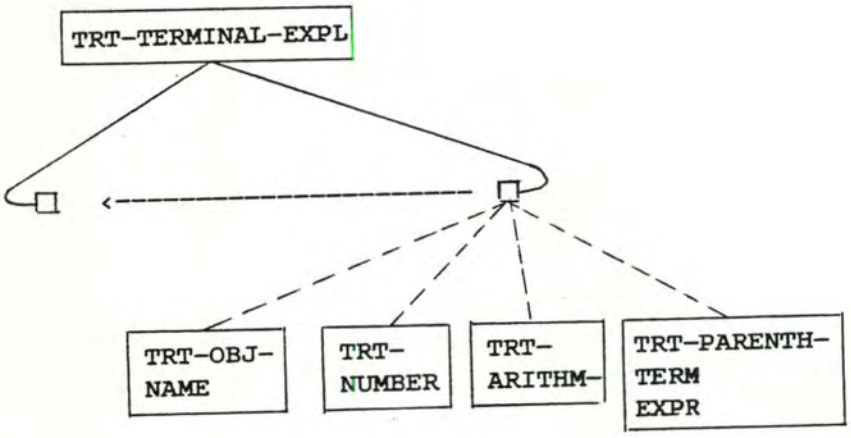

BODY
PREDICATE
CONJUNCTION
DISJUNCTION
NOT-BODY
PARENTH-BODY
USER-PREDI-CATE
BUILT-IN-PRE-DICATE
BODY1
BODY2
BODY1
BODY2
BODY
BODY

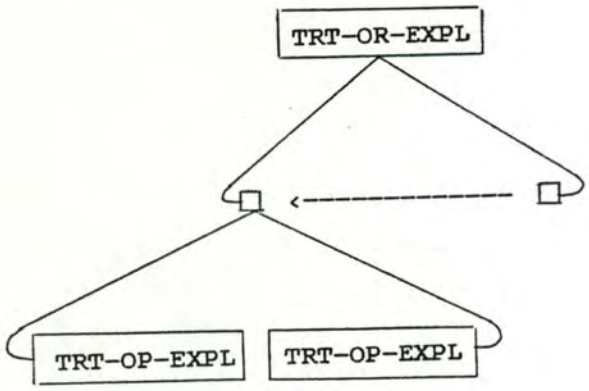

USER-PREDICATE
PRED-NAME
TERMES
EMPTY
LISTE-TERMS-PRLG
*
TERM-PRLG
CST
VARIABLE
PREFIX-TERM
INFIX-TERM
LIST-PRLG
PARENTH-TERM

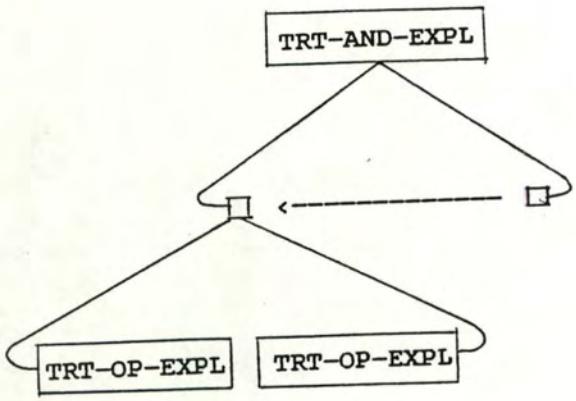
Représentation graphique d'une partie de l'énoncé :

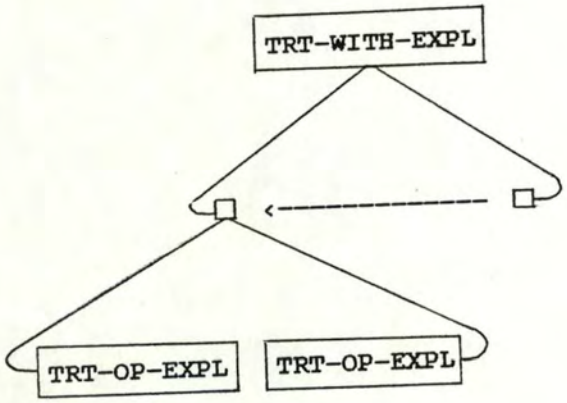


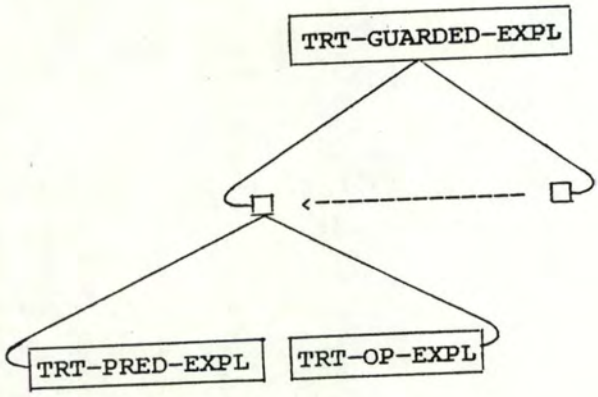

TRT-OP-EXPL
TRT-TERMINAL-EXPL
TRT AND-EXPL
TRT-OR-EXPL
TRT-WITH-EXPL
TRT-GUARDED-EXPL
TRT-PARENTH-EXPL


TRT-TERMINAL-EXPL
TRT-OBJ-NAME
TRT-NUMBER
TRT-ARITHM-
TRT-PARENTH-TERM EXPR


TRT-OR-EXPL
TRT-OP-EXPL
TRT-OP-EXPL


TRT-AND-EXPL
TRT-OP-EXPL
TRT-OP-EXPL


TRT-WITH-EXPL
TRT-OP-EXPL
TRT-OP-EXPL


TRT-GUARDED-EXPL
TRT-PRED-EXPL
TRT-OP-EXPL

3. Construction du programme C à partir de la spécification RSL

Le programme est implémenté en C sur UNIX BSD 4.2., VAX 750. Le procédé de construction que nous avons utilisé consiste à se baser sur la structure de la spécification RSL pour obtenir la structure du programme. A un bloc de spécification d'opération correspond dans notre traducteur une procédure. A une explicitation "or" correspond la plupart du temps une construction alternative "case". A une explicitation "forall" correspond une construction itérative (for i = 0 ; ... ; ++i). A une explicitation gardée correspond une construction alternative "if-then" ou "if-then-else".

4. Etat actuel du programme de traduction

La traduction complète d'une explicitation terminale est implémentée. Ceci comprend les cas suivants :

```
res = OP ( arg1, ..., argn)     ->   op( Res, Arg1, ..., Argn)

|->                             ->   :-

- res = obj-name                ->   Res = Obj-name

- res = number                  ->   Res is number

- res = 'texte'                 ->   Res = 'texte'


- res = expression arithm.      ->   Res is expression-arithm.

- res = (expression arithm.)    ->   Res is (expression-arithm.)


- res = OP'( arg1, ..., argn)   ->   op'( Res, Arg1, ..., Argn)


- res = OP'( arg1, ...,         ->   Res_int is expression_arithm.,
             expression-arithm.,      op'( Res, Arg1, ... Res_int,
             argn)                         ..., Argn)


- res = OP'( arg1, ...,         ->   op2( Res_int, ...),
             OP2 ( ...),              op( arg1,..., Res_int,..., Argn)
             ..., argn)


- res = OP'( arg1, ...,         ->   eval( Res_int, biblio-op(...)),
             BIBLIO-OP (...),         op( Res, arg1, ..., Res_int,
             ..., argn)                   ..., argn)


- res = BIBLIO-OP( arg1, ...,  ->  eval( Res, biblio-op( arg1, ...,
                   ..., argn )                           argn ) )


- res = BIBLIO-OP( arg1, ...,  ->  eval( Res, biblio-op( arg1, ...,
                   pred. simple,                         fonction,
                   ..., argn )                           argn ) )
```

```
- res =  BIBLIO-OP                    ->  Res_int is expr_arithm,
         ( arg1, ...,                     eval( Res, biblio-op( arg1, ...,
           expr-arithm.,                                        Res_int,
           ..., argn )                                           argn ) )

- res = BIBLIO-OP( arg1, ...,  ->      op2( Res_int,...),
                   OP2(...),           eval( Res, biblio_op( Arg1, ...,
                   ..., argn )                               Res_int,
                                                             ..., Argn )

- res  = BIBLIO-OP1                   ->  eval( Res,
          ( arg1, ...,                         biblio_op1( Arg1, ...,
            BIBLIO-OP2(...),                               biblio-op2(...),
            ..., argn )                                    ..., Argn))
```

Ceci constitue le noyau du programme.

CHAPITRE 3 : EVALUATION DE L'ENVIRONNEMENT

Dans ce chapitre, nous allons procéder à l'évaluation détaillée de l'environnement de prototypage présenté au chapitre 2. Nous commencerons par une évaluation de RSL sur base de notre expérience de spécification, ensuite nous discuterons des avantages et inconvénients de PROLOG dans le contexte du prototypage, et nous terminerons par l'outil de prototypage lui-même, avec l'adéquation des outils d'ALMA, et l'évaluation de notre programme de traduction RSL-PROLOG.

1 Evaluation de RSL

Dans cette section, nous présentons l'application que nous avons spécifiée dans le cadre de notre stage au C.I.G. (Bruxelles), ensuite nous dégagerons les qualités que devraient remplir une méthode et un langage de spécification, qualités qui serviront de base à une évaluation détaillée de RSL en fonction de notre expérience.

1.1 Notre expérience de spécification

1.1.1 Présentation de l'application traitée

CBR est un groupe comprenant, entre autres, une cimenterie. La cimenterie regroupe en fait 5 usines disséminées dans toute la

Belgique.

L'application traitée est l'application "Ventes" de la cimenterie. Cette application consiste en la gestion de fichiers (tiers, contrats), l'enregistrement de livraisons, la valorisation, la facturation comprenant l'édition de divers documents ainsi que des statistiques.

Les différentes fonctionnalités sont reprises ci-après.

1. Gestion du fichier des tiers

Le fichier des tiers reprend les clients, les fournisseurs et les commissionnaires de la société. Il est trié sur un indicatif comprenant le numéro de tiers. Ce dernier est identifiant dans la société. La gestion de ce fichier consiste en :

- la création de tiers,
- la modification d'informations concernant des tiers,
- l'annulation de tiers,

ainsi que l'édition des documents suivants :

- les cartons-tiers permettant d'avoir une image à jour du contenu du fichier des tiers
- les fiches clients
- les fiches fournisseurs
- les fiches clients classées par numéro de TVA.

Avant chaque opération, l'ordinateur effectuera des vérifications sur les éléments introduits afin de conserver la cohérence du système.

2. Gestion du fichier des contrats

Le fichier des contrats reprend tous les contrats passés entre la société et ses clients. Le concept de contrat correspond, chez C.B.R., au concept traditionnel de commande passée par un client, à quelques nuances près : la quantité et la qualité du ciment commandé peuvent rester indéterminées, ... La gestion de ce fichier consiste en :

- la création de contrats,
- la modification de contrats,
- l'annulation de contrats,
- l'édition de fiches contrats normales et de fiches contrats abrégées.

Tout contrat nouvellement créé fait l'objet d'une commande fictive afin de le tester. Si on arrive à la valoriser, on obtient un prix de référence pour la suite.
L'annulation d'un contrat se fait en deux phases dans le cas où le contrat a servi :

- neutralisation : suppression logique,
- annulation définitive au 31 décembre de l'année en cours.

Pendant la période de suppression logique, une réouverture du contrat est possible. Si aucune livraison n'a été enregistrée depuis le début du contrat, l'annulation est définitive. Le contrat est retiré physiquement du fichier.

Les contrats particuliers (concernant des clients occasionnels) font l'objet d'un traitement différent. Sous un même numéro de contrat de client particulier, on peut passer n'importe quelle livraison. Le traitement consiste en

l'enregistrement d'un contrat par siège, créé une fois pour toutes et ne donne lieu à aucune édition de fiches. Ce contrat contient en constantes les codes invariables pour tout client particulier. Le seul traitement ultérieur sur ce contrat consistera à mettre à jour, lors de l'enregistrement des livraisons, des zones utiles aux statistiques. Ces contrats donnant lieu à un paiement comptant, il n'y a pas de facturation.

Deux autres fichiers sont nécessaires aux traitements de l'application "Ventes" :

- le fichier des produits, propre à chaque usine et limité à ses produits. On distingue plusieurs types de produits : le produit principal, le conditionnement et le transport. Chaque produit a un code et une appellation différents selon le régime linguistique et le pays où il est livré.

  Ce fichier est un fichier peu mouvant et chaque usine peut en faire la mise-à-jour quand elle le désire. Dans le système actuel, le contrôle des produits existants se fait par le biais des barèmes lors de la valorisation de la livraison. Ces barèmes se trouvent en ordinateur et sont mis à jour régulièrement.

- le fichier transporteur qui répertorie tous les transporteurs. On y accède par un code.

3. Enregistrement des livraisons

Cette fonction "enregistrement des livraisons" consiste en différentes sous-fonctions :

- créer (ou modifier) des livraisons dans le fichier des livraisons et les valoriser,
- éditer les bons de livraison,
- éditer les "bordereaux d'expédition présents" c'est-à-dire la liste de toutes les livraisons enregistrées pendant la journée.

La création d'une livraison nécessite l'utilisation d'informations relatives aux contrats, produits et transporteurs ainsi qu'une interaction avec l'utilisateur. L'ordinateur vérifie dans la mesure du possible les données introduites au clavier.

La valorisation de la livraison est très complexe (le prix du ciment n'est pas fixe : il varie en fonction du contrat passé avec le client, de la qualité du ciment, de son mode de conditionnnement, ...).

L'édition des bons de livraison se fait suite à l'enregistrement d'une livraison. Un bon de livraison est édité pour un client, un transporteur et un contrat donnés.

4. Facturation courante

La procédure de facturation courante consiste à éditer tous les 15 jours, les différentes factures et notes de crédit, les notes de commission, les notes de provision, les relevés de transport ainsi que les tableaux de contrôle et les documents particuliers.

Toutes les livraisons concernant un même client sont facturées sur un même document sauf si le client a explicitement demandé que ce ne soit pas le cas.

Toutes les informations apparaissant sur les factures sont tirées du fichier des livraisons.

5. Facturation diverse

Cette procédure est utilisée pour la facturation des ventes en grande exportation et des ventes de sacherie, ainsi que pour certaines corrections de la facturation courante et pour des ventes occasionnelles de produits et services à comptabliser en résultats divisionnaires ou en récupération de charges. Il s'agit d'un moyen pour facturer tout ce qui ne passe pas dans la facturation normale. Elle couvre donc un large domaine. Cette facturation diverse va rejoindre la facturation normale pour les statistiques.

6. Statistiques

Sur base des données enregistrées ou préparées au cours du processus de facturation, divers états comptables et statistiques sont établis pour permettre :

- la comptabilisation des ventes,
- le contrôle budgétaire des ventes,
- l'analyse des ventes selon divers critères utiles à la gestion.

1.1.2 La démarche suivie

Désirant, dans un premier temps, tester l'applicabilité de la méthode de spécification et la puissance d'expression du langage de spécification sous-jacent, nous avons spécifié les principales fonctionnalités de l'application "Ventes" de la cimenterie : la gestion du fichier des tiers, des contrats, et des livraisons, ainsi que la valorisation, la facturation et une statistique.

L'approche régressive a été autant que possible privilégiée. Nous sommes parties de la liste des documents et résultats produits par l' application et nous en avons décrit la structure. Partant de là, les opérations nécessaires à la production de ces résultats ont été déterminées. La décomposition des opérations a été guidée par la structure des résultats correspondants, à l'exception de quelques cas de bas niveau où la décomposition des opérations a été guidée par la structure des arguments.

Il a cependant été constaté que la plupart du temps, on ne peut se limiter à une seule stratégie (purement régressive, progressive ou de réutilisation des connaissances) lors de l'application du méta-algorithme. La décomposition structurelle avec mise en évidence d'une suite guide qui est une stratégie mixte régressive/progressive illustre bien ce fait. Nous avons également remarqué que bien souvent, il n'existe pas une manière unique de spécifier l'application. Des choix entre des formulations équivalentes sont à prendre.

On trouvera en Annexe 1 les blocs de spécification pour la gestion du fichier des tiers.

1.2 Qualités d'un langage de spécification

La spécification d'un système informatique est une définition précise du problème que ce système doit résoudre, des tâches qu'il doit accomplir [MEYER 85]. C'est une description de ce qui est désiré plutôt que de la façon de le réaliser [BALZER 79]. La spécification est un outil de communication

- entre des non-informaticiens (demandeurs et futurs utilisateurs), et des informaticiens (réalisateurs du système). La spécification est en effet la base du contrat liant les deux parties, et
- entre informaticiens : la plupart des logiciels nécessitent la collaboration de plusieurs équipes chargées de la conception, de l'implémentation et de la maintenance du système. La spécification est un lien entre ces différentes équipes.

L'écriture de spécifications constitue une étape critique dans le développement d'un système informatique important [BOEHM 82]. D'une part, parce qu'il est très difficile d'écrire de bonnes spécifications, d'autre part parce qu'une erreur de spécification est souvent détectée tard et donc coûteuse à corriger. La seule spécification en langage naturel présente certains écueils qui rendent le produit résultant, sinon inacceptable, du moins insuffisant pour un développement rigoureux de logiciel [MEYER 85]. C'est pourquoi de nombreux langages de spécification sont actuellement proposés - ou font l'objet de recherches - afin de garantir certaines qualités de la spécification. L'accent y est souvent mis sur des modèles, des langages ou des outils de

spécification plutôt que sur des méthodes. Or la spécification d'un système complexe n'est pas une simple séquence linéaire de déclarations formelles ou informelles. C'est un texte bien structuré dont la rédaction exige beaucoup de créativité [DUBOIS 86]. C'est pourquoi le processus de construction et de stucturation d'une telle spécification est un élément important pour assurer des qualités telles que la facilité d'expression des besoins, la consistance, la complétude, et la possibilité de réutiliser la spécification. Le langage de spécification, quant à lui, doit permettre d'écrire une spécification minimale et concise, communicable, vérifiable, extensible, et facile à traduire en un programme correspondant. Ces critères sont une synthèse de [PARNAS 77], [BALZER 79] et [MEYER 85]. Ils seront exposés avant d'évaluer RSL sur base de notre expérience de spécification de l'application évoquée à la section 1.1 et de la spécification du traducteur RSL-PROLOG.

(1) Une méthode de spécification devrait favoriser la construction d'une spécification

- complète (absence de silence). L'expression "spécification complète" peut être perçue dans deux sens légèrement différents :
  - comme une spécification englobant absolument toutes les caractéristiques du problème. Cette propriété critique est pratiquement impossible à vérifier. "Tout au plus peut-on espérer détecter des présomptions de descriptions incomplètes" [BODART

83],

- comme une spécification dans laquelle tous les concepts utilisés sont définis.

- cohérente. La méthode de spécification devrait permettre d'écrire une spécification sans contradictions ni ambiguités.

- réutilisable : une bonne technique de spécification devrait favoriser la réutilisation d'éléments de spécification déjà écrits : définis une fois pour toutes dans une "boite à outils" ou précédemment définis par le spécifieur [MEYER 85].

(2) Une méthode de spécification devrait faciliter et guider le processus de structuration des spécifications

(3) Un langage de spécification devrait permettre d'écrire une spécification

- minimale et concise (absence de bruit). Le langage devrait permettre de construire des spécifications décrivant les propriétés intéressantes du problème et rien de plus. La redondance incontrôlée ainsi que les éléments n'apportant aucune information pertinente pour le problème obscurcissent souvent le texte et risquent d'introduire de nouvelles erreurs.

- communicable , c'est-à-dire précise, simple, et compréhensible à la fois par des non-informaticiens

(clients) et des informaticiens. Une personne familiarisée avec la notation utilisée devrait être capable de lire une spécification avec un minimum de difficulté.

- vérifiable : le langage devrait permettre de vérifier facilement que les spécifications sont une bonne traduction des désirs exprimés par l'utilisateur, qu'elles ne souffrent pas d'incomplétudes ou d'incohérences, ...

- modifiable, extensible : une légère modification d'une caractéristique du problème ne devrait donner lieu qu'à une légère modification de la spécification : l'information doit être localisée. On doit également pouvoir ajouter ou enlever des parties de la spécification sans grande perturbation de la structure existante.

- facilitant la construction et la vérification du programme correspondant.

(4) Un langage de spécification devrait posséder une grande puissance d'expression et être ainsi adaptable à une grande variété de problèmes : un langage de spécification permet souvent de décrire certains concepts de façon naturelle et directe. Par contre, d'autres concepts ne pourront être décrits, dans le même langage, qu'avec difficulté - ou pas du tout. Plus les concepts faciles à décrire sont nombreux, plus le langage est utile [LISKOV 75].

1.3 Evaluation de RSL

(1) La méthode de spécification RSL favorise-t-elle la construction d'une spécification

- complète (englobant toutes les caractéristiques du problème).

  Un avantage de traduire une spécification informelle des besoins en une spécification formelle est d'amener le spécifieur à se poser certaines questions qui, dans une approche informelle, pourraient rester dans l'ombre, et donc sans réponse [MEYER 85]. Ceci favorise la complétude des spécifications obtenues. Mais ne la garantit pas. Cette qualité est en effet pratiquement impossible à vérifier.

  De plus, la possibilité offerte par RSL d'utiliser une démarche régressive, guidée par la structure des résultats, réduit fortement les risques d'oublis de résultats, sans introduire de risques d'oublis d'arguments. En effet, en utilisant une telle démarche, on identifie les résultats que le système doit produire. Leur structure permet d'identifier des résultats intermédiaires, ... On arrivera en fin de processus aux arguments nécessaires pour produire les résultats voulus.

- réutilisable

  La méthode RSL favorise la réutilisation de parties de spécifications précédemment définies, grâce à la

philosophie de la bibliothèque, et au mécanisme de réutilisation des connaissances (cfr. chapitre 2).

La définition d'une bibliothèque est très intéressante car elle permet d'une part un gain de temps (les opérations de base n'ont plus à être spécifiées), et d'autre part, de réduire les risques d'introduction d'erreurs, ...

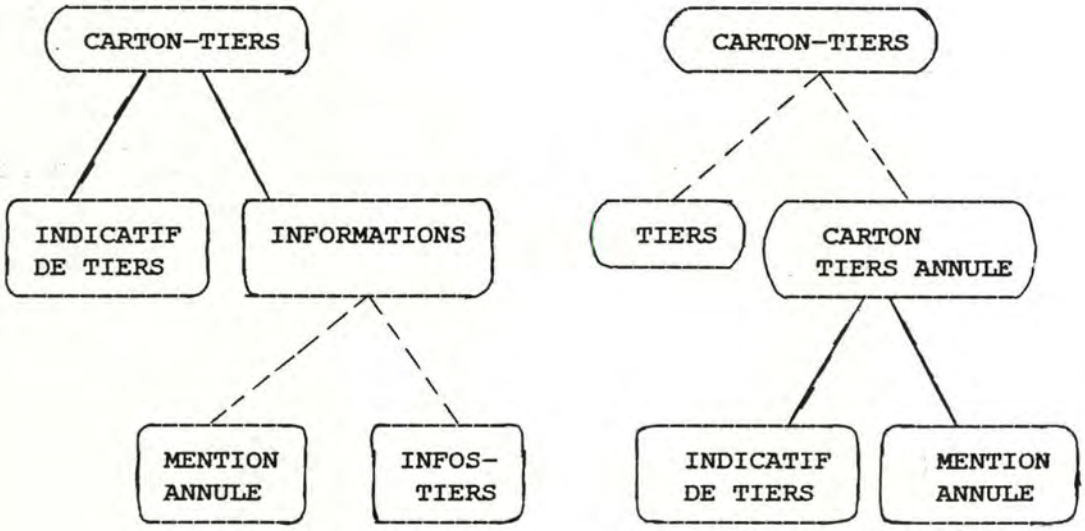
(2) La méthode de spécification RSL facilite-t-elle le processus de construction et de structuration des spécifications ?

La méthode facilite le processus de structuration des spécifications car, ayant choisi une stratégie, le spécifieur est guidé dans sa tâche par un méta-algorithme de spécification. De plus, la méthode de spécification, tout en guidant le spécifieur dans sa démarche, lui laisse une grande liberté quant aux stratégies (orientées-fonction, orientées-objet, ...) et mécanismes (décomposition, spécialisation et classification) à appliquer.

Un problème se situe cependant au niveau des explicitations de type : plusieurs alternatives peuvent se présenter.

Exemple tiré de la sous-application gestion du fichier des tiers : Un carton tiers reprend des informations concernant un tiers mis à jour (ajouté, modifié ou supprimé). Dans le cas d'un tiers supprimé, ce carton se limite à l'indicatif du tiers et la mention "ANNULE". Dans les autres cas, il reprend toutes les

informations concernant un tiers. Un choix de structuration du type est dès lors possible :



La solution choisie influencera le reste du développement (décomposition des opérations, structure du programme).
Ce problème semble ne pas être propre à RSL. Nous l'avons aussi rencontré dans le cadre du cours d'analyse fonctionnelle de première licence (modèle E/A). Il n'existe pas une seule formalisation correcte.

(3) Le langage RSL permet-il d'écrire une spécification minimale et concise ?

Une spécification RSL est assez concise car ne sont pas pris en compte des aspects relatifs à l'implantation des structures de données, à l'allocation des ressources, ...

Cependant, les deux modèles utilisés pour structurer une spécification, l'énoncé et l'univers, sont duaux, en ce sens que chaque unité de spécification dans un modèle a sa

contrepartie dans l'autre modèle. Ce principe introduit une certaine forme de redondance, mais il est très utile pour effectuer des vérifications de cohérence [DUBOIS 86].

D'autre part, le fait d'expliciter tout produit cartésien comme un n-uple de types

T |-> CART-PROD [t1, ..., tn],

conjugué au fait que la procédure de sélection d'un élément dans un objet de type produit cartésien se fait sur base du type de l'élément

élément = TUPLESEL (TYPE-ELEMENT, objet)

font que le spécifieur sera obligé de définir autant de types intermédiaires qu'il y a d'éléments de même type dans le n-uple. Ces types intermédiaires devant à leur tour être spécifiés chacun dans un bloc de spécification où ils seront explicités par le constructeur de type IS.

Exemple tiré de la spécification du traducteur : le type AND-EXPLICITATION RSL devrait être explicité comme un

CART-PROD [OP-EXPLICITATION, OP-EXPLICITATION]

Cependant, pour expliciter l'opération de traitement d'une "explicitation and", il faut sélectionner chacune des "op-explicitations" et la traiter.

op-expl1 = TUPLESEL (OP-EXPLICITATION,
and-explicitation)

ne fonctionne pas car les deux éléments de l'explicitation and sont de même type. On est donc obligé de définir des types intermédiaires OP-EXPL1 et OP-EXPL2. AND-EXPLICITATION sera alors explicité comme un

CART-PROD [OP-EXPL1, OP-EXPL2]

avec OP-EXPL1 et OP-EXPL2 définis chacun dans un bloc et explicités comme suit :

OP-EXPLi |-> IS [OP-EXPLICITATION]

- communicable

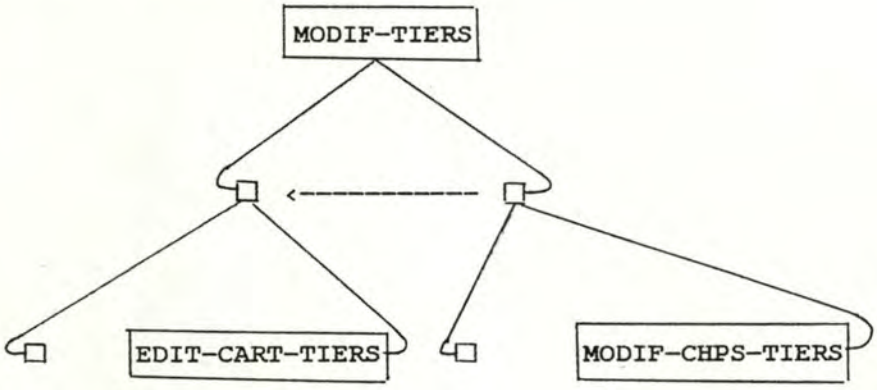
Le grand avantage d'une spécification RSL est de combiner une description formelle et une description informelle du problème (cfr. la structure des blocs de spécification, chapitre 2 point 1.2.5). On tire ainsi tous les avantages d'une spécification formelle (précision, rigueur, ...) et ceux du langage naturel (communicable et compréhensible par des non-informaticiens), tout en éliminant les inconvénients de chacune d'entr'elles prise isolément. On reproche en effet aux spécifications formelles pures (descriptions axiomatiques, définitions algébriques, ...) leur manque de lisibilité ([MEYER 85] et [LISKOV 75]), et aux spécifications en langage naturel, leur manque de précision, de rigueur ainsi que leur ambiguité [MEYER 85].

Un autre élément favorisant la communicabilité des spécifications RSL est la description graphique simplifiée -

sous forme de forêt d'arbres - de l'application. Cette description permet d'avoir une vue globale et récapitulative du système spécifié.

Cependant, un problème que nous souhaitons mentionner concernant cette description graphique est le suivant : il se peut qu'à un certain niveau, on obtienne, dans une explicitation, à la fois des opérations non terminales, et des opérations terminales (opérations de la bibliothèque ou combinaisons de celles-ci), ces dernières étant représentées dans l'arbre par des ||. De telles situations occasionnent un arbre moins lisible.

Exemple tiré de la sous-application gestion du fichier des tiers : la description graphique de l'opération de modification de tiers :



- vérifiable.

La structure d'une spécification RSL (l'ensemble des blocs) permet de vérifier aisément que tout concept employé est défini. En effet, pour contrôler si tous les types d'objet

intervenant dans le domaine ou le co-domaine d'une opération sont définis, il suffit de parcourir les blocs de l'univers. De même, on peut vérifier que toute opération intervenant dans une explicitation est définie dans un bloc.

De plus, les spécifications formelles aident à mettre en évidence les ambiguités et contradictions parce qu'elles obligent le spécifieur à décrire précisément et avec rigueur les caractéristiques du problème. Dans le cadre de RSL, les ambiguités éventuellement présentes dans la partie informelle d'une spécification devraient être levées par les descriptions formelles.

On peut envisager l'automatisation de telles vérifications, notamment en utilisant les outils d' ALMA. Des prototypes d'outils de contrôle de ce genre ont déjà été réalisés [BUISE 83], ou sont en cours de réalisation dans le cadre d'un projet parallèle développé au CRIN (Nancy), le projet SACSO.

De plus, étant donné la bonne compréhensibilité d'une spécification RSL, l'utilisateur pourra facilement valider celle-ci (conformité par rapport aux besoins). D'autre part, l'aspect aisé de la traduction vers des langages exécutables permet d'obtenir assez rapidement un prototype qui servira d'outil de validation de la spécification. Et enfin, le principe de dualité entre les deux modèles de la spécification, ainsi que les mécanismes de structuration (and, or, forall, ...) facilitent la mise en évidence des classes d'équivalence pour le concepteur de jeux de tests.

- extensible et modifiable.

  Un changement dans un concept de la spécification (type ou opération) est facilement répercuté en RSL et localisé. Ainsi l'identification d'une nouvelle opération utilisant des types déjà spécifiés donnera lieu à

  - la création du bloc la définissant,
  - la mise à jour d'anciens blocs de l'univers, quand cette opération s'insère dans l'explicitation d'une ou plusieurs autres opérations, et
  - la mise à jour d'anciens blocs de l'énoncé (blocs définissant les domaines et co-domaines de la nouvelle opération).

  Exemple : dans la spécification du traducteur, la prise en considération du traitement d'une "explicitation excep"t donnerait lieu à la création du bloc la définissant, à la mise à jour du bloc spécifiant l'opération de traitement d'une explicitation d'opération, ainsi que la mise à jour du bloc spécifiant le type OP-EXPLICITATION.

- facilitant la construction du programme correspondant.

  La complexité du processus de construction d'un programme est avant tout fonction de la complexité du problème spécifié. Cependant, la structure d'une spécification RSL nous semble faciliter, sinon influencer, la structure du programme correspondant. Ainsi, la tentation est grande de faire correspondre une procédure du programme à un bloc spécifiant une opération, une structure "case" ou "if" à une explicitation-or, une structure "while" ou "for" à une

explicitation-forall, ... comme nous l'avons fait pour le traducteur.

(4) RSL est-il adaptable à une grande variété de problèmes ?

La bibliothèque est extensible à volonté et donc adaptable à l'application. Cependant, les spécifications de performances en termes de temps, d'espace et de droit d'accès n'ont pas (encore) été prises en considération [DUBOIS 85]. RSL n'est donc pas adapté aux applications faisant intervenir du parallélisme, des conditions d'activation, ... telles que des applications d'OS., de réseaux, de contrôle de processus, ... D'autre part, une limite actuelle de RSL est qu'il ne permet pas de spécifier l'interface utilisateur. Or, tout système interactif nécessite des interfaces qui devront être spécifiés à l'aide d'un autre langage si l'on veut faire du prototypage d'interface. Des extensions dans ce sens sont actuellement en cours d'investigation.

RSL nous parait particulièrement bien adapté pour des applications de gestion. Il permet de définir assez facilement les structures de données de l'application, et les opérations s'y rapportant. Cependant, d'autres applications nous semblent difficiles à spécifier en RSL. C'est le cas pour les systèmes de traitement de textes qui nécessitent d'exprimer des déplacements de curseur, ...

D'autre part, une limite à la puissance d'expression de RSL, dans la version que nous avons utilisée, est le fait que les "ou" sont exclusifs. Il est parfois nécessaire d'avoir des "ou" inclusifs, notamment dans le cas des traitements d'erreurs.

Exemple tiré de la sous-application gestion du fichier des tiers :

```
( ERREUR1 ( arg )
  and
  mess = TRT-ERREUR1 ( arg ) )
or
( ERREUR2 ( arg )
  and
  mess = TRT-ERREUR2 ( arg ) )
or
 ...
```

où plusieurs erreurs sont possibles dans une même situation. Une précondition ne doit dès lors pas nécessairement être décomposée en préconditions exclusives.

D'autre part, un problème non pris en considération par RSL est celui des éléments facultatifs : comment expliciter un produit cartésien dont certains éléments ne sont pas obligatoires ? Ainsi, pour expliciter un type d'objet PERSONNE composé d'un certain nombre d'éléments dont la situation de service militaire, facultative, plusieurs solutions sont possibles :

- Soit créer deux types distincts :

```
PERSONNE |-> UNION [ HOMME, FEMME ]

avec

HOMME |-> CART-PROD [ NOM, SITUATION-MILITAIRE,
                      ADRESSE, ... ]

FEMME |-> CART-PROD [ NOM, ADRESSE, ... ].
```

Cette solution est lourde dans le cas de produits cartésiens complexes (tels que les contrats de notre spécification de l'application "ventes" de C.B.R., constitués d'une centaine d'éléments, dont certains sont facultatifs).

- Soit donner une valeur par défaut. Se pose alors le problème du choix de cette valeur : 'XXXXXXX' dans notre exemple du type PERSONNE.

- Soit incorporer dans la bibliothèque un type "ABSENCE DE VALEUR" :

```
PERSONNE
    |-> CART-PROD [ NOM, SITUATION-MILITAIRE,
                   ADRESSE, ... ]

SITUATION-MILITAIRE
    |-> UNION [ ABSENCE DE VALEUR, CHARST ].
```

1.4 Conclusion

RSL nous semble un bon compromis entre un langage naturel "pur" et un langage formel "pur".

Une spécification RSL nous est apparue assez facile à construire, à comprendre, à traduire, à modifier et à réutiliser. De plus, on peut aisément, voire automatiquement, vérifier qu'elle

est complète (au sens énoncé) et cohérente.

Cependant, un inconvénient majeur de RSL est son "applicabilité" limitée : il ne permet de formuler que les spécifications fonctionnelles. Les aspects de performances, sécurité, ... ne peuvent être intégrés dans la version actuelle. Cet inconvénient n'est sans doute que temporaire, puisque la méthode de spécification et le langage RSL font encore l'objet de recherches.

2 Evaluation de PROLOG dans le cadre du prototypage

2.1 Aspects positifs :

Les raisons du choix de PROLOG comme langage de programmation dans le cadre du prototypage sont les suivantes :

Tout d'abord, PROLOG est un langage de programmation théoriquement déclaratif. L'interpréteur de PROLOG contenant sa propre stratégie de résolution, il est (théoriquement) possible de décrire un problème sans faire référence à une stratégie de contrôle ou à un algorithme particulier pour les enchainements.

Ensuite, PROLOG offre une grande concision syntaxique ainsi qu'une grande puissance d'expression. Les programmes PROLOG sont environ dix fois plus courts que des programmes équivalents écrits en Pascal, Basic ou Fortran.

Exemple : soit un programme de concaténation de deux listes. On a, en PROLOG :

```
conc( L, [], L).
conc( [X | L3], [X | L1], L2 ) :- conc( L3, L1, L2 ).
```

où conc (A,B,C) signifie que A est la concaténation des listes B et C.

Et en Pascal :

```
procedure conc (var l3 : liste; l1 : liste; l2 : liste);

var i, j : integer ;

begin
     i = 0 ;
     j = 0 ;
     while ( l1[i] <> END_OF_ARRAY ) do
     begin
          l3[i] := l1[i] ;
          i := i + 1
     end ;
     while ( l2[j] <> END_OF_ARRAY ) do
     begin
          l3[i] := l2[j] ;
          i := i + 1 ;
          j := j + 1
     end ;
     l3[i] := END_OF_ARRAY
end ;
```

où le type liste doit avoir été défini plus tôt, ainsi que la constante END_OF_ARRAY

En outre, PROLOG offre une grande facilité de création et de manipulation uniforme de structures de données : aucune déclaration de type n'est nécessaire et les manipulations de structures de données complexes sont aisées, notamment grâce à la représentation sous forme de listes.

Ces trois qualités, permettent d'écrire facilement et rapidement des prototypes en PROLOG.

De plus, tout repose, en PROLOG, sur le concept de relation entre objets : un prédicat exprime une relation entre les divers arguments. C'est également le cas en RSL où une expression d'opération représente une relation entre des arguments et un résultat. Il y a donc une grande similitude entre les deux formalismes. Ainsi "X est le double de Y" sera exprimé, en PROLOG

comme suit : double(X, Y), et en RSL, de la façon suivante : x = DOUBLE (y). Cette similitude facilite fortement la traduction d'une spécification RSL en un programme PROLOG.

Un autre avantage théorique de PROLOG est la réversibilité : il n'y a pas, a priori, en PROLOG, de notion de variable d'entrée ou de sortie. Toute variable est instanciée par unification et utilisée, tantôt comme variable d'entrée, tantôt comme variable de sortie selon le mode désiré. D'où l'idée d'utiliser PROLOG pour la génération de jeux de tests. On désire, dans ce cas, déduire, pour une opération et un résultat donnés, les (des) arguments susceptibles de produire ce résultat. Ceci ne va pas sans problème, notamment à cause de l'utilisation des opérations arithmétiques et des prédicats prédéfinis extra-logiques. La réversibilité est donc partielle.

Exemple :

```
op1(Res, Arg) :- Res is 3 * Arg.

op(Res, Arg1, Arg2) :- op1( Res, Arg2).

?- op(Res, 4, 6)
   Réponse :  Res = 18, yes

?- op(21, X, Y)

   Réponse :  op1(21,Y) Error in arithmetic expression :
                        (Y is) not a number
```

Cette possibilité de générer des jeux de tests fait actuellement l'objet d'études, notamment aux F.N.D.P. [HABRA 86].

Toutes les qualités énoncées ci-dessus sont des avantages de PROLOG par rapport aux langages de programmation impérative tels que Pascal, Fortran, ... Parmi les langages de programmation

déclaratifs, Lisp semble également un bon candidat pour la génération automatique de prototypes. Cependant, il ne possède pas la qualité de réversibilité : il faut définir explicitement quels sont les résultats et quels sont les arguments.

2.2 Aspects négatifs :

Un certain nombre de reproches peuvent être faits à PROLOG :

Un programme PROLOG n'est pas purement déclaratif. Il est aussi procédural : on ne peut négliger la stratégie utilisée par le moteur d'inférence, ainsi que les caractéristiques extra-logiques de PROLOG pour construire un programme correct. Il faut donc en tenir compte dans la génération de prototypes.

Le manque de performances : les mécanismes mis en oeuvre dans PROLOG sont très puissants mais coûteux en temps, en CPU et en place mémoire. Ce défaut n'est pas primordial dans le cadre d'un outil de prototypage où le critère d'efficacité n'est pas prépondérant. Cependant, les temps de réponses peuvent être très longs, ce qui rend l'évaluation du prototype désagréable pour le client. Il est possible d'améliorer les performances d'un programme PROLOG en ayant recours à des procédures extra-logiques (le cut), Cependant, l'utilisation de ces procédures nous éloigne de la programmation logique : l'emploi du cut nécessite la prise en compte de la façon dont l'interpréteur travaille. Pour cette raison, on risque également d'introduire des erreurs dans le programme.

Les principales lacunes de PROLOG, pour le prototypage, résident au niveau des entrées/sorties. Les prédicats prédéfinis sont peu nombreux et ne permettent pas grand chose. Aussi le critère de convivialité que devrait satisfaire un prototype n'est pas rempli de prime abord. De plus, les interfaces homme/machine du prototype seront trop différents des interfaces du système final pour qu'un prototype PROLOG puisse servir d'outil d'apprentissage. Dans notre expérience manuelle, une part importante du développement a été consacrée à l'écriture des modules de dialogue et de production de documents. Il semble que dans ce domaine la situation soit en train d'évoluer et que de nouvelles versions de PROLOG comprennent diverses facilités du point de vue des entrées/sorties (TURBO-PROLOG, ...).

2.3 Conclusion

PROLOG permet une notation concise et compacte. Ses caractéristiques déclaratives et la grande facilité de création et de manipulation de structures de données en font un langage particulièrement adapté pour le prototypage. Cependant, la pauvreté de l'environnement de PROLOG constitue un handicap si l'on désire prototyper une application très interactive, et le manque de performances peut être gênant pour de grosses applications. Ces deux considérations font qu'un prototype PROLOG ne sera pas réutilisable dans le produit final, et devra être jeté. Néanmoins, ceci n'est qu'une conclusion à court terme. En effet, divers travaux en cours tentent de remédier à ces inconvénients en offrant un meilleur environnement de programmation.

3 Evaluation de la génération d'un prototype PROLOG

Pour évaluer la génération d'un prototype PROLOG à partir d'un texte exprimé en RSL, nous commencerons par les règles de transformation, ensuite, nous aborderons l'automatisation partielle de ces règles, (adéquation des outils d'ALMA et programme de traduction).

3.1 Les règles de transformation.

1. Eléments positifs :

- Simplicité : les règles de transformation sont simples et font bien apparaitre les correspondances entre les éléments RSL et PROLOG. Elles sont donc faciles à appliquer.

- Automatisation. Les règles de transformation étant simples et assez systématiques, il est possible de les automatiser partiellement.

- Validité du programme PROLOG : les règles garantissent la correction du programme PROLOG obtenu quand ce dernier est utilisé suivant la directionnalité

  { in (var, ground) } P { out (ground, ground) },

  c'est-à-dire en mode d'exploitation normal d'un prototype.

2. Limites :

- Les règles de transformation ne sont pas complètes : le traitement du "except" n'a pas été formalisé. Il s'agit d'un mécanisme permettant d'exprimer le fait qu'un objet est identique à un autre excepté pour quelques éléments.

  Exemple : tiers-modifié = tiers-ancien

  except TUPLESEL (NOM, tiers-modifié) = 'Dupont'

- Le programme obtenu ne sera pas réversible : utilisé dans le sens { in (ground, var) } P { out (ground, ground) }, le programme ne fonctionnera pas pour les raisons abordées précédemment dans ce travail.

- Des problèmes se poseront quand une spécification RSL comprendra des forall imbriqués dans d'autres structures, en raison de la récursivité à introduire en PROLOG. Il faudra dans ce cas solliciter l' intervention de l'utilisateur , de façon à introduire des opérations intermédiaires dans la spécification RSL.

  Exemple :

```
TYPE |-> SEQ [ SEQ [TYPEX]]

argument, resultat        : SEQ
element, element-resultat : TYPE

resultat = OP ( argument)

|->

for all sous-seq :
     IN ( sous-seq, argument) :

   for all element :
       IN ( element, sous-seq :
          element-resultat = 3 * elt.
```

n'est pas traduisible directement en PROLOG. Il faut introduire une opération intermédiaire :

```
resultat = OP ( argument)

|->

for all sous-seq :
     IN ( sous-seq, argument) :
   resultat-int = OP'(sequence-int)
```

avec OP' défini dans un autre bloc de spécification :

```
resultat = OP' ( sequence-int )

|->

for all element :
    IN ( element, sequence-int) :
         element-resultat = 3 * element.
```

- Rappelons qu'une intervention de l'utilisateur sera également nécessaire pour skolemiser et hornifier des préconditions ou des invariants

Remarque : nous avons choisi de représenter les opérations RSL non terminales par des prédicats PROLOG

```
res = OP (arglist)          op( Res, Arglist),
```

et les opérations terminales par des termes.

```
res = BIBLIO-OP (arglist)   eval ( Res, biblio_op( Arglist)).
```

Une autre solution eût été de représenter toutes les opérations par des termes :

```
res = OP (arglist)          eval( Res, op(Arglist))
```

Notre choix a pour inconvénient de devoir passer d'une représentation à l'autre en cours de transformation d'un texte RSL vers un programme PROLOG, mais il garantit une plus grande efficacité du programme PROLOG résultant. En effet, représenter toutes les opérations par des termes donnerait lieu à un programme constitué d'une seule et énorme procédure récursive.

3.2 Adéquation des outils d'ALMA.

ALMA offrant, entre autres, des outils génériques de manipulation de textes formels, il semblait intéressant d'utiliser certains de ces outils afin de profiter des facilités offertes par une représentation arborescente correspondant à la syntaxe abstraite d'un texte. Non seulement une telle représentation est débarassée des mots et symboles réservés du formalisme, mais elle permet des manipulations aisées sur un texte : on peut facilement accéder à telle partie (sous-arbre) du texte, remplacer un sous-arbre par un autre, ajouter ou supprimer un sous-arbre, ... Alors qu'un texte représenté comme une suite de caractères ne permet que des déplacements et manipulations élémentaires en termes de caractères.

Exemple : un noeud de l'arbre abstrait étant représenté en C par une structure composée des références aux noeuds père, fils, frère droit, du code du noeud, ainsi que d'une zone libre, il est facile, par le mécanisme de pointeur, de se deplacer dans l'arbre et d'accéder aux informations qui nous intéressent. Ainsi, l'accès au noeud "assertion d'explicitation" à partir de la référence du noeud "bloc

de spécification" se fait très simplement comme suit :

ref_bloc_de_specification->fils->frere->fils->fils->frere->frere

Alors que si le texte était représenté comme une suite de caractères, il faudrait le parcourir caractère par caractère jusqu'à rencontrer les mots-clés "EXPLICITATION INPUT-OUTPUT ASSERTION". Il faut alors traiter le flux de caractères jusqu'au mot-clé suivant.

De plus, les primitives de construction d'arbres (création d'un noeud zéroaire, ...) figurant dans la boite à outils, il suffisait de les utiliser.

En outre, l'éditeur visuel, bien que n'intervenant pas directement dans la production automatique d'un prototype PROLOG s'avère très pratique pour écrire la spécification RSL. En effet, il permet à l'utilisateur de ne pas se soucier des nombreux mots-clés, ni de la correction syntaxique de son texte.

Quand au décompilateur, il permet d'obtenir directement le programme Prolog correspondant à l'arbre généré.

Pour ces raisons, les outils d'ALMA semblent donc appropriés à nos besoins.

Néanmoins, la façon de procéder de YACC nous a obligées à introduire des délimiteurs supplémentaires, et à remplacer certains délimiteurs par d'autres dans le syntaxe LDF de RSL. Ceci a pour inconvénient que le texte en entrée doit être introduit via l'éditeur syntaxique si l'on veut qu'il soit conforme à la syntaxe LDF.

3.3 Le programme de transformation RSL - PROLOG

1. Etat actuel : Pour rappel, l'implémentation actuelle du programme de transformation se limite aux procédures de traduction d'une explicitation terminale. (voir le chapitre 2 point 3.2.2.)

2. Limites :

- Le texte RSL reçu en entrée est supposé sémantiquement correct :
  - le nombre d'arguments pour chaque opération doit être exact, pour que le programme PROLOG généré fonctionne correctement,

    Exemple : pas d'argument pour l'opérateur de création d'une liste vide : SEMPTY ()

  - les arguments sont supposés être de type adéquat

    Exemple : le premier argument d'un FILTER doit être de type séquence et le second de type prédicat RSL

  - il n'y a pas de cycle dans la spécification tel que

```
res = OP ( arglist1)
|->
res = OP'(arglist2)
```

et

```
res = OP'( arglisti)
|->
res = OP (arglistj)
```

Cet exemple donnerait lieu en PROLOG à une boucle

infinie à l'appel de op ou de op'.

- Un prédicat RSL argument d'une opération de la bibliothèque doit être un prédicat simple, à cause de la syntaxe LDF actuelle de PROLOG qui différencie le niveau terme du niveau prédicat. Pour résoudre ce problème, il suffirait d'introduire ";" et "," comme foncteurs infixés.

3. Perspectives d'amélioration :

(1) Il faudrait effectuer un certain nombre de vérifications sur le texte RSL reçu en argument :

- l'absence de cycles (dans les opérations et les types),
- le nombre et le type des arguments,
- tout objet est typé,
- tout type non terminal est explicité dans un bloc de spécification,
- toute opération non terminale est explicitée dans un bloc de spécification,
- toute opération figure dans le bloc de spécification de son co-domaine et de ses domaines,
- toute opération figurant dans un bloc de spécification d'un type est explicitée dans l'énoncé du problème,
- tout(e) précondition (invariant) non terminale est explicitée,
- tout type est explicité une seule fois,
- toute opération est explicité une seule fois,
- ...

(2) Il reste à implémenter les procédures de traitement d'une explicitation and, or, forall, with, et gardée dont les spécifications figurent dans le chapitre précédent.

(3) On pourrait améliorer les performances du programme de traduction en remplaçant l'allocation statique par de l'allocation dynamique.

3.4 Conclusion

Quelles que soient les améliorations qui pourraient être apportées au programme, une automatisation totale de la traduction d'un texte RSL en un programme PROLOG parait utopique, notamment pour les raisons suivantes :

- tout n'est pas représentable sous forme de clause de Horn (équivalence, ...)
- une expression RSL complexe peut nécessiter l'introduction d'opérations intermédiaires (forall imbriqués).

Comme nous l'avons souligné à plusieurs reprises, l'intervention de l'utilisateur de l'outil pourra toujours être requise, mais à des endroits, et pour des tâches bien précis.

CONCLUSION

De nombreuses recherches portent actuellement sur les moyens de réduire les coûts de développement de logiciels. Le prototypage, permettant une validation anticipée de l'adéquation des spécifications par rapport aux besoins du demandeur, s'inscrit dans ce cadre. Les principales approches de prototypage proposées actuellement sont : la production "manuelle" d'un prototype dans un langage de programmation de haut niveau, l'écriture de spécifications exécutables, et la production (semi-)automatique d'un prototype à partir de la spécification du système.

L'environnement de prototypage présenté ici relève de la troisième approche. Le passage d'une spécification RSL à un programme PROLOG est réalisé via des règles de transformation partiellement automatisables. Dans le cadre de ce mémoire, seule une partie de ces règles a été automatisée, l'objectif premier étant de tester la faisabilité de l'outil, ainsi que l'applicabilité du langage et de la méthode de spécification RSL. Pour ce faire, nous avons spécifié, lors de notre stage au C.I.G. (Bruxelles) une application de gestion. Celle-ci présentant une grande variété quant aux traitements et aux données manipulées nous a permis de mettre en lumière les qualités et limites de RSL. Ce langage nous semble un bon compromis entre un langage naturel "pur" et un langage formel "pur". De plus, la méthodologie sous-jacente rend plus aisée la construction des spécifications. Une partie de l'application spécifiée a été traduite manuellement en un programme PROLOG, faisant ressortir la possibilité et la nécessité d'une

automatisation.

La démarche de prototypage consiste à spécifier le système en RSL (éventuellement à l'aide d'un éditeur syntaxique), à transmettre la spécification au programme de prototypage, ce dernier générant un programme PROLOG exécutable, tout en nécessitant parfois des interventions ponctuelles de l'utilisateur de l'outil. La traduction (semi-)automatique passe par trois étapes principales :

- l'analyse syntaxique du texte RSL, avec production de l'arbre abstrait correspondant,
- la création, à partir de l'arbre abstrait RSL, d'un arbre abstrait PROLOG,
- la décompilation de l'arbre abstrait PROLOG en un programme PROLOG.

L'analyse syntaxique et la décompilation ont été réalisées par des outils développés dans le cadre du projet ALMA.

Les avantages de cet outil de prototypage sont que le prototype peut être obtenu rapidement et au moindre coût. A condition toutefois que la spécification du système soit déjà exprimée en RSL.

Néanmoins, cet environnement souffre de certains défauts non négligeables :

- RSL ne prenant pas en compte la spécification des interfaces usagers, ils doivent être programmés manuellement. A ceci

s'ajoute la pauvreté de l'environnement PROLOG, avec comme conséquence une perte de temps, et un allongement important du délai d'obtention du prototype.

- Le manque d'efficacité de PROLOG, allié à la pauvreté de son environnement conduisent à un prototype dont l'évaluation sera fastidieuse.

- L'impossibilité d'automatisation totale est due au fait que PROLOG travaille sur des clauses de Horn (on ne peut tout exprimer sous cette forme), et à la grande liberté laissée au spécifieur RSL (tous les cas ne peuvent être prévus).

Dans sa forme actuelle, notre programme de traduction nous paraît donc adéquat pour des applications de taille raisonnable, et peu interactives. Cependant, ceci n'est peut-être que temporaire. En effet, RSL fait encore l'objet d'études. De plus, des recherches sur le prototypage d'interface dans le même cadre sont en cours. Et enfin, les versions récentes de PROLOG semblent bénéficier d'améliorations au niveau des performances et de l'environnement.

Une direction de recherche dans le prolongement de l'outil de prototypage présenté serait de développer un environnement assez riche, comprenant, entre autres,

- des outils d'aide à la spécification RSL (vérifications de complétude, cohérence, ...), et
- la gestion des interactions lorsqu'une intervention humaine est nécessaire en cours de production du prototype.

BIBLIOGRAPHIE

[ALMA 86] BOUQUELLE J.P., BUYSE M., CHAMPAGNE R., DELCOURT B., DELOR E., ERVIER M., NISOLE P., SCHAYES M.C., SELDESLACHT J.,VAN LAMSWEERDE A., The Kernel of a Generic Software Development Environment. To appear in Proceedings Second ACM Software Engineering Symposium on Practical Software Development Environments, Palo Alto (Calif.), Dec. 9-11, 1986.

[BALZER 79] BALZER R., GOLDMAN N., Principles of Good Software Specification and their Implications for Specification Language, Proceedings, Specifications of Reliable Software, IEEE Catalog no. 79CH 1401-9C, pp 58-67, 1979.

[BODART 83] BODART F., PIGNEUR Y., Conception assistée des applications informatiques 1. Etude d'opportunité et analyse conceptuelle, Masson et Presses universitaires de Namur, 1983.

[BOEHM 82] BOEHM B., Les facteurs de coût du logiciel TSI vol. 1, numero 1.

[BUDDE 84] BUDDE R., KUHLENKAMP K., MATHIASSEN L., ZULLIGHOVEN H., Approaches to Prototyping, Springer-Verlag, 1984.

[BUYSE 83] BUYSE M., et VANHEMELRYCK P., Création d'un environnement de spécification pour le langage SPES. Mémoire de fin d'études, Institut d'Informatique, FNDP Namur, Juin 1983.

[CLOCKIN 81] CLOCKSIN W.F., MELLISH C.S., Programming in PROLOG, Springer-Verlag, 1981.

[DARLINGTON 85] Darlington J., Field A.J., Pull H., The Unification of Functional and Logic Languages, 1985.

[DUBOIS 84] DUBOIS E., Cadre et Méthode de Spécification de Systèmes d'Information Fondés sur les Types de Données, Thèse de Docteur-Ingénieur en Informatique, Institut National Polytechnique de Lorraine, Nancy, 1984.

[DUBOIS 85] DUBOIS E., FINANCE JP., VAN LAMSWEERDE A., A Constructive Approach to Requirements Specification Report M101 Philips Research Laboratory, Brussels, 1985.

[DUBOIS 86] DUBOIS E., VAN LAMSWEERDE A.,
Making Specification Processes Explicit,
Position paper for the 4th International Workshop on
Software Specification and Design, IEEE,
Monterey (Calif.), March 87.

[GOGUEN 78] GOGUEN J.A., THATCHER J.W., WAGNER E.G.,
An Initial Algebra Approach to the Specification,
Correctness and Implementation of Abstract Data Type,
in Current Trends in Programming Methodology, Vol. 4,
R. Yeh (Ed.), Prentice Hall, 1978.

[HABRA 86] HABRA N.,
Prototypage en prolog de spécifications
fonctionnelles : une étude de cas,
rapport technique, 1986, FNDP Namur.

[HENDERSON 86] HENDERSON P.,
Functionnal Programming, Formal Specification
and Rapid Prototyping,
IEEE Transactions on software engineering,
vol SE-12 no 2, February 86, pp 241-250.

[HOGGER 84] HOGGER C.J.,
Introduction to logic programming
Academic Press, 1984.

[KOWALSKI 79a] KOWALSKI R.,
Logic for problem solving
(Artificial Intelligence Series, vol.7),
Elsevier-North Holland-New York.

[KOWALSKI 79b] KOWALSKI R.,
Algorithm = logic + control,
Communications of the ACM 22, pp 424-431, 1979

[KOWALSKI 74] KOWALSKI R.,
Predicate logic as a programming language,
Proc. of I.F.I.P., North Holland,
Amsterdam, pp 569-574, 1974.

[LEE 85] LEE S.,
On executable models for rule-based prototyping,
IEEE, 1985.

[LISKOV 75] LISKOV B.H., ZILLES S.P.,
Specification techniques for data abstractions,
IEEE Transactions on software engineering,
vol SE-1, no 1, March 75, pp 7-19.

[MEYER 85] MEYER B.,
On formalism in specifications,
IEEE software, January 1985, pp 6-26.

[PARNAS 77] PARNAS D.L.,
The use of precise specifications in the development of software, Information Processing 77, IFIP, North-Holland Publishing Company 1977, pp 861-867.

[SEN 82] Software engineering notes,
Special issue on rapid prototyping
Volume 7, Number 5, Dec 1982.

[VLA 82] VAN LAMSWEERDE A.,
Les outils d'aide au développement de logiciels : un aperçu des tendances actuelles. Proc JIIA, 1982, Paris, Juin 1982.

[VLA 86] VAN LAMSWEERDE A.,
Syntaxe concrète de RSL,
Rapport technique, FNDP Namur, 1986.

Facultés Universitaires Notre-Dame de la Paix (Namur)

Institut d'informatique

ETUDE ET EVALUATION D'UN OUTIL DE PROTOTYPAGE

(ANNEXES)

Mémoire présenté par

Fabienne LANGELEZ

Cécile PARIS

en vue de l'obtention

du titre de

Licencié et Maitre en Informatique

Année académique 1985-1986

TABLE DES MATIERES

A N N E X E 1 :

BLOCS DE SPECIFICATION DE LA SOUS-APPLICATION

" GESTION DU FICHIER DES TIERS "

DE L'APPLICATION " VENTES " DE C.B.R.

ET SYNTHESE GRAPHIQUE

SPECIFICATION OF OPERATION GEST-F-TIERS

LEXICON :

OBJECTIVE :

gestion du fichier des tiers

OBJECTS :

res-g-f-tiers	: résultat global de la gestion du fichier des tiers
fich-tiers	: fichier des tiers
date-du-jour	: date du jour
ddes-maj-tiers	: demandes de mise à jour du fichier des tiers
demande-edit-f-c	: booléen vrai s'il y a une demande d'édition des fiches clients
demande-edit-f-c-ntva	: booléen vrai s'il y a une demande d'édition des fiches clients triées par numéro de TVA
res-majs-tiers	: résultats des diverses mises à jour du fichier des tiers
fiches-fourn	: fiches fournisseurs
fiches-cli	: fiches clients
fiches-cli-ntva	: fiches clients classées par numéro de TVA

OPERATIONS :

MAJS-F-TIERS	: opérations de mises à jour du fichier des tiers
EDIT-F-F	: édition des fiches fournisseurs
EDIT-F-C	: édition des fiches clients
EDIT-F-C-NTVA	: édition des fiches clients triées par numéro de TVA

OUTLINE :

TYPES :

res-g-f-tiers	: RES-GEST-F-TIERS
fich-tiers	: FICH-TIERS
date-du-jour	: DATE
ddes-maj-tiers	: DDES-MAJ-TIERS
demande-edit-f-c	: BOOLEAN
demande-edit-f-c-ntva	: BOOLEAN
res-majs-tiers	: RES-MAJS-F-TIERS
fiches-fourn	: FICH-TIERS
fiches-cli	: FICH-TIERS
fiches-cli-ntva	: FICH-TIERS

OPERATIONS :

GEST-F-TIERS	: FICH-TIERS X DATE X DDES-MAJ-TIERS X BOOLEAN X BOOLEAN -> RES-GEST-F-TIERS
MAJS-F-TIERS	: FICH-TIERS X DDES-MAJ-TIERS X DATE -> RES-MAJS-TIERS
EDIT-F-F	: FICH-TIERS -> FICH-TIERS
EDIT-F-C	: FICH-TIERS -> FICH-TIERS
EDIT-F-C-NTVA	: FICH-TIERS -> FICH-TIERS

FORMAL SPEC :

RESULT : res-g-f-tiers

ARGUMENTS : fich-tiers, date-du-jour,
ddes-maj-tiers,
demande-edit-f-c,
demande-edit-f-c-ntva

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
res-g-f-tiers
  =  GEST-F-TIERS (fich-tiers, date-du-jour,
                   ddes-maj-tiers,
                   demande-edit-f-c,
                   demande-edit-f-c-ntva)

 |->

   res-majs-tiers  =  MAJS-F-TIERS (fich-tiers,
                                    ddes-maj-tiers,
                                    date-du-jour)

   and

   ( JJ (: date-du-jour)  =  01
     and
     fiches-fourn  =  EDIT-F-F (fich-tiers) )

   and

   ( demande-edit-f-c
     and
     fiches-cli  =  EDIT-F-C (fich-tiers) )

   and

   ( demande-edit-f-c-ntva
     and
     fiches-cli-ntva  = EDIT-F-C-NTVA (fich-tiers))
```

SPECIFICATION OF OPERATION EDIT-F-F

LEXICON :

OBJECTIVE :

Edition des fiches fournisseurs, reprenant toutes les informations concernant les fournisseurs de la société (code qualité à 100)

OBJECTS :

fiches-fourn	:	fiches fournisseurs
fich-tiers	:	fichier des tiers
tiers	:	un tiers
fiche-fourn	:	fiche comprenant toutes les informations sur un fournisseur de la société

OPERATIONS :

IN	:	in specbase
TUPLESEL	:	in specbase

OUTLINE :

TYPES :

fiches-fourn	:	FICH-TIERS
fich-tiers	:	FICH-TIERS
tiers	:	TIERS
fiche-fourn	:	TIERS

OPERATIONS :

EDIT-F-F	:	PICH-TIERS -> PICH-TIERS
IN	:	in specbase
TUPLESEL	:	in specbase

FORMAL SPEC :

RESULT : fiches-fourn

ARGUMENTS : fich-tiers

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

fiches-fourn = EDIT-F-F (fich-tiers)

|->

for all tiers :

IN (tiers, fich-tiers)

and

TUPLESEL(QUAL-T, tiers) = 100 :

fiche-fourn = tiers

SPECIFICATION OF OPERATION EDIT-F-C

LEXICON :

OBJECTIVE :

Edition des fiches clients
reprenant toutes les informations
concernant les clients de la
société (code qualité = 1000)

OBJECTS :

fiches-cli	:	fiches clients
fich-tiers	:	fichier des tiers
tiers	:	un tiers
fiche-cli	:	fiche comprenant toutes les informations concernant un client

OPERATIONS :

IN	:	in specbase
TUPLESEL	:	in specbase

OUTLINE :

TYPES :

fiches-cli	:	FICH-TIERS
fich-tiers	:	FICH-TIERS
tiers	:	TIERS
fiche-cli	:	TIERS

OPERATIONS :

EDIT-F-C	:	FICH-TIERS -> FICH-TIERS
IN	:	in specbase
TUPLESEL	:	in specbase

FORMAL SPEC :

RESULT : fiches-cli

ARGUMENTS : fich-tiers

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
fiches-cli = EDIT-F-C (fich-tiers)
|->
for all tiers :
        IN (tiers, fich-tiers )
        and
        TUPLESEL ( QUALI-T, tiers ) = 1000 :

    fiche-cli = tiers
```

SPECIFICATION OF OPERATION EDIT-F-C-NTVA

LEXICON :

OBJECTIVE :

Edition des fiches clients reprenant
toutes les informations concernant
les clients de la société
(code qualité = 1000),
triées par numéro de TVA,

OBJECTS :

fiches-cli-ntva	:	fiches clients triées par numéro de TVA
fich-tiers	:	fichier des tiers
tiers	:	un tiers
fich-tiers-trie-ntva	:	fichier des tiers trié par numéro de TVA
fiche-cli-ntva	:	fiche comprenant toutes les informations concernant un client

OPERATIONS :

IN	:	in specbase
TUPLESEL	:	in specbase
SORT	:	in specbase

OUTLINE :

TYPES :

fiches-cli-ntva	:	PICH-TIERS
fich-tiers	:	PICH-TIERS
tiers	:	TIERS
fich-tiers-trie-ntva	:	PICH-TIERS
fiche-cli-ntva	:	TIERS

OPERATIONS :

EDIT-F-C-NTVA	:	FICH-TIERS -> FICH-TIERS
IN	:	in specbase
TUPLESEL	:	in specbase
SORT	:	in specbase

FORMAL SPEC :

RESULT : fiches-cli-ntva

ARGUMENTS : fich-tiers

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

fiches-cli-ntva = EDIT-F-C-NTVA (fich-tiers)
|->
for all tiers :
IN(tiers, fich-tiers-trie-ntva)
and
TUPLESEL (QUALI-T, tiers) = 1000 :

fiche-cli-ntva = tiers

with
fich-tiers-trie-ntva = SORT (fich-tiers, NUM-TVA)

SPECIFICATION OF OPERATION MAJS-F-TIERS

LEXICON :

OBJECTIVE :

mettre à jour le fichier des tiers,
en fonction d'un certain nombre
de demandes de mise à jour

OBJECTS :

fich-tiers	:	fichier des tiers
res-majs-tiers	:	résultat de mises à jour du fichier des tiers
ddes-maj-tiers	:	demandes de mises à jour de tiers
date-du-jour	:	date du jour
dde-maj-tiers	:	une demande de mise à jour de tiers
res-maj-tiers	:	résultat d'une mise à jour du fichier des tiers

OPERATIONS :

MAJ-F-TIERS	:	une opération de mise à jour du fichier des tiers

OUTLINE :

TYPES :

fich-tiers	:	FICH-TIERS
res-majs-tiers	:	RES-MAJS-F-TIERS
ddes-maj-tiers	:	DDES-MAJ-TIERS
date-du-jour	:	DATE
dde-maj-tiers	:	DDE-MAJ-TIERS
res-maj-tiers	:	RES-MAJ-F-TIERS

OPERATIONS :

MAJS-F-TIERS	:	FICH-TIERS X DDES-MAJ-TIERS X DATE -> RES-MAJS-F-TIERS
MAJ-F-TIERS	:	FICH-TIERS X DDE-MAJ-TIERS -> RES-MAJ-TIERS

FORMAL SPEC :

RESULT : res-majs-tiers

ARGUMENTS : fich-tiers, ddes-maj-tiers, date-du-jour

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

res-majs-tiers = MAJS-F-TIERS (fich-tiers,
ddes-maj-tiers,
date-du-jour)

|->

for all dde-maj-tiers :

IN (dde-maj-tiers, ddes-maj-tiers) :

res-maj-tiers = MAJ-F-TIERS (fich-tiers,
dde-maj-tiers,
date-du-jour)

SPECIFICATION OF OPERATION MAJ-F-TIERS

LEXICON :

OBJECTIVE :

effectuer une mise à jour du fichier des tiers (création, modification ou annulation d'un tiers)

OBJECTS :

res-maj-tiers	:	résultat d'une mise à jour du fichier des tiers
fich-tiers	:	fichier des tiers
dde-maj-tiers	:	demande de mise à jour du fichier des tiers (ajout, modification ou suppression d'un tiers)
date-du-jour	:	date du jour
mess-err-maj-tiers	:	message d'erreur
res-maj-tiers-ok	:	résultat d'une mise à jour correcte du fichier des tiers

OPERATIONS :

DEF-MESS-ERR-MAJ-TIERS	:	définition de message d'erreur
MAJ-TIERS-OK	:	mise à jour correcte du fichier des tiers

OUTLINE :

TYPES :

res-maj-tiers	:	RES-MAJ-F-TIERS
fich-tiers	:	FICH-TIERS
dde-maj-tiers	:	DDE-MAJ-TIERS
date-du-jour	:	DATE
mess-err-maj-tiers	:	CHARST
res-maj-tiers-ok	:	RES-MAJ-TIERS-OK

OPERATIONS :

MAJ-F-TIERS	:	FICH-TIERS X DDE-MAJ-TIERS X DATE -> RES-MAJ-F-TIERS
DEF-MESS-ERR-MAJ-TIERS	:	FICH-TIERS X DDE-MAJ-TIERS X DATE -> CHARST
MAJ-TIERS-OK	:	FICH-TIERS X DDE-MAJ-TIERS -> RES-MAJ-TIERS-OK

FORMAL SPEC :

RESULT : res-maj-tiers

ARGUMENTS : fich-tiers, dde-maj-tiers,
date-du-jour

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
res-maj-tiers = MAJ-F-TIERS (fich-tiers,
                             dde-maj-tiers,
                             date-du-jour )

|->

  ( not  OK-MAJ-TIERS  ( fich-tiers,
                         dde-maj-tiers,
                         date-du-jour )

    and

    mess-err-maj-tiers
      =  DEF-MESS-ERR-MAJ-TIERS ( fich-tiers,
                                  dde-maj-tiers,
                                  date-du-jour ))

 or

  ( OK-MAJ-TIERS  (fich-tiers, dde-maj-tiers,
                   date-du-jour)

    and

    res-maj-tiers-ok
       = MAJ-TIERS-OK ( fich-tiers,
                        dde-maj-tiers))
```

SPECIFICATION OF OPERATION MAJ-TIERS-OK

LEXICON :

OBJECTIVE :

effectuer une mise à jour du fichier des tiers (création, modification, ou annulation d'un tiers)

OBJECTS :

res-maj-tiers-ok	:	résultat d'une mise à jour correcte du fichier des tiers
fich-tiers	:	fichier des tiers
dde-maj-tiers	:	demande de mise à jour du fichier des tiers (ajout, modification ou suppression d'un tiers)

OPERATIONS :

CREA-TIERS	:	création d'un tiers
MODIF -TIERS	:	modification d'un tiers
ANNU-TIERS	:	annulation d'un tiers
TYPE-OF	:	in specbase

PRECONDITIONS :

OK-MAJ-TIERS	:	prédicat indiquant si la demande de mise à jour est correcte
OK-CREA-TIERS	:	prédicat indiquant si la demande de création est correcte
OK-MODIF-TIERS	:	prédicat indiquant si la demande de modification est correcte
OK-ANNU-TIERS	:	prédicat indiquant si la demande d'annulation est correcte

OUTLINE :

TYPES :

res-maj-tiers-ok	:	RES-MAJ-TIERS-OK
fich-tiers	:	FICH-TIERS
dde-maj-tiers	:	DDE-MAJ-TIERS

OPERATIONS :

```
MAJ-TIERS-OK : FICH-TIERS X DDE-MAJ-TIERS
                  -> RES-MAJ-TIERS-OK

CREA-TIERS   : FICH-TIERS  X  TIERS
                  ->  RES-MAJ-TIERS-OK

MODIF-TIERS  : FICH-TIERS  X  DDE-MODI-TIERS
                  ->  RES-MAJ-TIERS-OK

ANNU-TIERS   : FICH-TIERS  X  INDIC-T
                  ->  RES-MAJ-TIERS-OK
```

PRECONDITIONS :

```
OK-MAJ-TIERS    : FICH-TIERS  X  DDE-MAJ-TIERS
                                 -> BOOLEEN

OK-CREA-TIERS   : FICH-TIERS  X  TIERS
                                 -> BOOLEEN

OK-MODIF-TIERS  : FICH-TIERS  X  DDE-MODI-TIERS
                                 -> BOOLEEN

OK-ANNU-TIERS   : FICH-TIERS  X  INDIC-T
                                 -> BOOLEEN
```

FORMAL SPEC :

RESULT : res-maj-tiers-ok

ARGUMENTS : fich-tiers, dde-maj-tiers

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
res-maj-tiers-ok  =  MAJ-TIERS-OK (fich-tiers,
                                   dde-maj-tiers)
|->
   ( TYPE-OF (TIERS, dde-maj-tiers)
     and
     res-maj-tiers-ok
      = CREA-TIERS (fich-tiers, dde-maj-tiers))
or
   ( TYPE-OF (DDE-MODI-TIERS, dde-maj-tiers)
     and
     res-maj-tiers-ok
      = MODIF-TIERS (fich-tiers, dde-maj-tiers))
or
   ( TYPE-OF (INDIC-T, dde-maj-tiers)
     and
     res-maj-tiers-ok
      = ANNU-TIERS (fich-tiers,dde-maj-tiers))
```

EXPLICITATION OF PRECONDITIONS :

```
OK-MAJ-TIERS (fich-tiers, dde-maj-tiers,
              date-du-jour)
|->
   ( TYPE-OF (TIERS, dde-maj-tiers)
     and
     OK-CREA-TIERS (fich-tiers, dde-maj-tiers))
or
   ( TYPE-OF (DDE-MODI-TIERS, dde-maj-tiers)
     and
     OK-MODIF-TIERS (fich-tiers, dde-maj-tiers))
or
   ( TYPE-OF (INDIC-T, dde-maj-tiers)
     and
     OK-ANNU-TIERS ( fich-tiers, dde-maj-tiers,
                     date-du-jour ) )
```

SPECIFICATION OF OPERATION CREA-TIERS

LEXICON :

OBJECTIVE : effectuer la création d'un tiers

OBJECTS :

res-maj-tiers-ok	: résultat d'une mise à jour correcte du fichier des tiers
fich-tiers	: fichier des tiers
dde-maj-tiers	: demande de mise à jour du fichier des tiers (ajout d'un tiers)
fich-tiers'	: fichier des tiers après la création
carton-tiers	: carte reprenant toutes les informations concernant le tiers nouvellement créé
nouv-tiers	: tiers à insérer dans le fichier

OPERATIONS :

EDIT-CART-TIERS	: édition du carton tiers
SORT	: in specbase
APPEND	: in specbase
TUPLESEL	: in specbase

PRECONDITIONS :

OK-CREA-TIERS	: prédicat indiquant si la demande de création est correcte
EXIST-TIERS	: prédicat indiquant si le tiers est présent dans le fichier
COMPLETUDE-TIERS	: prédicat indiquant si la demande est complète
OK-CODES-TIERS	: prédicat indiquant si les codes introduits sont corrects

OUTLINE :

TYPES :

res-maj-tiers-ok	: RES-MAJ-TIERS-OK
fich-tiers	: FICH-TIERS
dde-maj-tiers	: TIERS
fich-tiers'	: FICH-TIERS
carton-tiers	: CARTE-TIERS
nouv-tiers	: TIERS

OPERATIONS :

CREA-TIERS	: FICH-TIERS X TIERS -> RES-MAJ-TIERS-OK
EDIT-CART-TIERS	: TIERS -> CARTE-TIERS
SORT	: in specbase
APPEND	: in specbase
TUPLESEL	: in specbase

PRECONDITIONS :

OK-CREA-TIERS	: FICH-TIERS X TIERS -> BOOLEEN
EXIST-TIERS	: FICH-TIERS X INT -> BOOLEEN
COMPLETUDE-TIERS	: TIERS -> BOOLEEN
OK-CODES-TIERS	: TIERS -> BOOLEEN

FORMAL SPEC :

RESULT : res-maj-tiers-ok

ARGUMENTS : fich-tiers, dde-maj-tiers

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
res-maj-tiers-ok  =  CREA-TIERS (fich-tiers,
                                 dde-maj-tiers )
|->
  fich-tiers' = SORT (APPEND (fich-tiers, nouv-tiers),
                      INDIC-T )
and
  carton-tiers = EDIT-CART-TIERS (nouv-tiers)
with
  nouv-tiers  =  dde-maj-tiers
    except  TUPLESEL (LIB-MAJ, nouv-tiers) = 'crea'
      and  TUPLESEL (DERN-SOLD-0, nouv-tiers)
                    = 00/00/00
```

EXPLICITATION OF PRECONDITIONS :

```
OK-CREA-TIERS ( fich-tiers, dde-maj-tiers )
|->
  not EXIST-TIERS (fich-tiers,
                   TUPLESEL (INDIC-T ,
                             TUPLESEL (NUM-TIE,
                                       dde-maj-tiers)
                            )
                   )
and
  COMPLETUDE-TIERS ( dde-maj-tiers )
and
  OK-CODES-TIERS ( dde-maj-tiers )
```

SPECIFICATION OF OPERATION MODIF-TIERS

LEXICON :

OBJECTIVE :

effectuer une modification d'un tiers

OBJECTS :

res-maj-tiers-ok	: résultat d'une mise à jour correcte du fichier des tiers
fich-tiers	: fichier des tiers
dde-maj-tiers	: demande de mise à jour du fichier des tiers (modification d'un tiers)
fich-tiers'	: fichier des tiers après modification
carton-tiers	: carte reprenant toutes les informations concernant le tiers modifié
fich-tiers-filtre	: partie du fichier des tiers comprenant le tiers à modifier
tiers-modifie	: tiers modifié

OPERATIONS :

EDIT-CART-TIERS	: édition d'un carton tiers
SORT	: in specbase
APPEND	: in specbase
FILTER	: in specbase
MODIF-CHPS-TIERS	: modification des champs d'un tiers

PRECONDITIONS :

OK-MODIF-TIERS	: prédicat indiquant si la demande de modification est correcte
EXIST-TIERS	: prédicat indiquant si le tiers est présent dans le fichier
OK-CODES-TIERS	: prédicat indiquant si les codes introduits sont corrects

OUTLINE :

TYPES :

res-maj-tiers-ok	: RES-MAJ-TIERS-OK
fich-tiers	: FICH-TIERS
dde-maj-tiers	: DDE-MODI-TIERS
fich-tiers'	: FICH-TIERS
carton-tiers	: CARTE-TIERS
fich-tiers-filtre	: FICH-TIERS
tiers-modifie	: TIERS

OPERATIONS :

MODIF-TIERS	: FICH-TIERS X DDE-MODI-TIERS -> RES-MAJ-TIERS-OK
EDIT-CART-TIERS	: TIERS -> CARTE-TIERS
MODIF-CHPS-TIERS	: FICH-TIERS X DDE-MODI-TIERS -> TIERS
APPEND	: in specbase
SORT	: in specbase
FILTER	: in specbase

PRECONDITIONS :

OK-MODIF-TIERS	: FICH-TIERS X DDE-MODI-TIERS -> BOOLEEN
EXIST-TIERS	: FICH-TIERS X INT -> BOOLEEN
OK-CODES-TIERS	: TIERS -> BOOLEEN

FORMAL SPEC :

RESULT : res-maj-tiers-ok

ARGUMENTS : fich-tiers, dde-maj-tiers

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
res-maj-tiers-ok = MODIF-TIERS (fich-tiers,
                                dde-maj-tiers )
|->
   fich-tiers' = SORT ( APPEND (fich-tiers-filtre,
                                tiers-modifie),
                        INDIC-T)
and
  carton-tiers = EDIT-CART-TIERS (tiers-modifie)

with

   fich-tiers-filtre
     = FILTER ( fich-tiers, TUPLESEL (INDIC-T, tiers)
                <> TUPLESEL (INDIC-T, dde-maj-tiers))
and

   tiers-modifie
     = MODIF-CHPS-TIERS ( fich-tiers, dde-maj-tiers )
```

EXPLICITATION OF PRECONDITIONS :

```
  OK-MODIF-TIERS ( fich-tiers, dde-maj-tiers )

|->
  EXIST-TIERS(fich-tiers,
              TUPLESEL(INDIC-T,TUPLESEL(NUM-TIE,
              )        )                dde-maj-tiers)
and
   ( forall chp-a-modifier :
       IN ( chp-a-modifier,
            TUPLESEL (CHPS-A-MODIFIER, dde-maj-tiers))
     TUPLESEL (NOM-CHAMP, chp-a-modifier ) <> 'INDIC-T'
   )
and

   OK-CODES-TIERS ( tiers-modifie )

   with tiers-modifie
        = MODIF-CHPS-TIERS ( fich-tiers, dde-maj-tiers)
```

SPECIFICATION OF OPERATION ANNU-TIERS

LEXICON :

OBJECTIVE :

effectuer l'annulation d'un tiers

OBJECTS :

res-maj-tiers-ok	: résultat d'une mise à jour correcte du fichier des tiers
fich-tiers	: fichier des tiers
dde-maj-tiers	: demande de mise à jour du fichier des tiers (suppression d'un tiers)
fich-tiers'	: fichier des tiers après annulation
carton-tiers	: carte reprenant des informations concernant le tiers annulé
tiers-annule	: tiers annulé

OPERATIONS :

EDIT-CART-TIERS	: édition d'un carton tiers
FILTER	: in specbase
APLAT	: in specbase
TUPLESEL	: in specbase

PRECONDITIONS :

OK-ANNU-TIERS	: prédicat indiquant si la demande d'annulation est correcte
EXIST-TIERS	: prédicat indiquant si le tiers est présent dans le fichier

OUTLINE :

TYPES :

res-maj-tiers-ok	: RES-MAJ-TIERS-OK
fich-tiers	: FICH-TIERS
dde-maj-tiers	: INDIC-T
fich-tiers'	: FICH-TIERS
carton-tiers	: CARTE-TIERS
tiers-annule	: TIERS

OPERATIONS :

ANNU-TIERS	: FICH-TIERS X INDIC-T -> RES-MAJ-TIERS-OK
EDIT-CART-TIERS	: TIERS -> CART-TIERS
FILTER	: in specbase
APLAT	: in specbase
TUPLESEL	: in specbase

PRECONDITIONS :

OK-ANNU-TIERS	: FICH-TIERS X INDIC-T X DATE -> BOOLEEN
EXIST-TIERS	: FICH-TIERS X INT -> BOOLEEN

FORMAL SPEC :

RESULT : res-maj-tiers-ok

ARGUMENTS : fich-tiers, dde-maj-tiers

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
res-maj-tiers-ok  =  ANNU-TIERS ( fich-tiers,
                                  dde-maj-tiers)
 !->

  fich-tiers'
    = FILTER (fich-tiers,
              TUPLESEL (INDIC-T,tiers) <> dde-maj-tiers)
 and
    carton-tiers = EDIT-CART-TIERS ( tiers-annule )
 with
  tiers-annule
   = APLAT (FILTER (fich-tiers,TUPLESEL (INDIC-T, tiers)
                                  = dde-maj-tiers ) )

      except TUPLESEL (LIB-MAJ, tiers-annule) = 'annu'
```

EXPLICITATION OF PRECONDITIONS :

```
OK-ANNU-TIERS ( fich-tiers, dde-maj-tiers,
                date-du-jour)
 !->

  EXIST-TIERS ( fich-tiers, TUPLESEL ( NUM-TIE,
                                       dde-maj-tiers))
and
 ( TUPLESEL ( MM, date-du-jour ) = 12
   and
   TUPLESEL ( JJ, date-du-jour ) = 31  )
and
 ( TUPLESEL (AA, TUPLESEL(DERN-SOLD-0,tiers-a-annuler)
                 <  TUPLESEL (AA, date-du-jour) - 1 )
  or
  (TUPLESEL (AA, TUPLESEL(DERN-SOLD-0,tiers-a-annuler)
                 =  TUPLESEL (AA, date-du-jour) - 1 )
   and
   TUPLESEL (MM, TUPLESEL(DERN-SOLD-0,tiers-a-annuler)
                    <= 06 ))  )
with tiers-a-annuler
       = APLAT (FILTER (fich-tiers,
                        TUPLESEL(INDIC-T, tiers)
                             = dde-maj-tiers))
```

SPECIFICATION OF OPERATION MODIF-CHPS-TIERS

LEXICON :

OBJECTIVE :

effectuer une modification de champs d'un tiers

OBJECTS :

tiers-modifie	: tiers modifié
fich-tiers	: fichier des tiers
dde-maj-tiers	: liste des modifications à effectuer
chp-a-modifier	: un champ à modifier

OPERATIONS :

APLAT	: in specbase
FILTER	: in specbase
TUPLESEL	: in specbase

OUTLINE :

TYPES :

tiers-modifie	: TIERS
fich-tiers	: FICH-TIERS
dde-maj-tiers	: DDE-MODI-TIERS
chp-a-modifier	: CHP-A-MODIFIER

OPERATIONS :

MODIF-CHPS-TIERS	: FICH-TIERS X DDE-MODI-TIERS -> RES-MAJ-TIERS-OK
APLAT	: in specbase
FILTER	: in specbase
TUPLESEL	: in specbase

FORMAL SPEC :

RESULT : tiers-modifie

ARGUMENTS : fich-tiers, dde-maj-tiers

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
tiers-modifie  =  MODIF-CHPS-TIERS (fich-tiers,
                                    dde-maj-tiers)

|->

   ( forall chp-a-modifier :

       IN (chp-a-modifier,
           TUPLESEL (CHPS-A-MODIFIER, dde-maj-tiers)):

     tiers-modifie  =  tiers-modifie

     except TUPLESEL(TUPLESEL (NOM-CHP, dde-maj-tiers),
                     tiers-modifie)

                 =  TUPLESEL (NOUV-VAL, dde-maj-tiers)

         and

             TUPLESEL (LIB-MAJ, tiers-modifie) = 'modi'
   )

 with

   tiers-modifie

     = APLAT (FILTER ( fich-tiers,
                       TUPLESEL (INDIC-T, tiers)
                     = TUPLESEL (INDIC-T,dde-maj-tiers)
                       )
              )
```

SPECIFICATION OF OPERATION EDIT-CART-TIERS

LEXICON :

OBJECTIVE :

effectuer l'édition d'un carton tiers

OBJECTS :

tiers	: un tiers
carton-tiers	: un carton tiers

OPERATIONS :

TUPLESEL	: in specbase
TUPLEFORM	: in specbase

OUTLINE :

TYPES :

carton-tiers	: CARTE-TIERS
tiers	: TIERS

OPERATIONS :

EDIT-CART-TIERS	: TIERS -> CARTE-TIERS
TUPLESEL	: in specbase
TUPLEFORM	: in specbase

FORMAL SPEC :

RESULT : carton-tiers

ARGUMENTS : tiers

EXPLICITATION OF INPUT-OUTPUT ASSERTION :

```
carton-tiers = EDIT-CART-TIERS ( tiers )
|->
( TUPLESEL (LIB-MAJ, tiers) <> 'annu'
  and
  carton-tiers = tiers
)
or
( TUPLESEL (LIB-MAJ, tiers) = 'annu'
  and
  carton-tiers
     = TUPLEFORM (TUPLESEL (INDIC-T, tiers),
                  'ANNULE' )
)
```

SPECIFICATION OF OBJECT TYPE RES-GEST-F-TIERS

LEXICON :

OBJECTIVE :

résultat des diverses mises à jour du fichier des tiers (fichier des tiers mis à jour, messages d'erreur éventuels, ainsi que les fiches clients, fiches fournisseurs et les fiches clients classées par numéro de TVA)

OPERATIONS :

GEST-F-TIERS : gestion du fichier des tiers

TYPES :

RES-MAJS-F-TIERS : résultats de mises à jour du fichier des tiers

FICHES-FOURN : fichier reprenant toutes les informations concernant les fournisseurs de la société

FICHES-CLI : idem, mais pour les clients de la societe

FICHES-CLI-NTVA : idem, mais pour les clients de la société et par ordre croissant de numéro de TVA

OUTLINE :

OPERATIONS :

GEST-F-TIERS : FICH-TIERS X DATE X DDES-MAJ-TIERS X BOOLEAN X BOOLEAN -> RES-GEST-F-TIERS

TYPES :

res-g-f-tiers : RES-GEST-F-TIERS

FORMAL SPEC :

ASSOCIATED OPERATIONS :

GEST-F-TIERS

EXPLICITATION OF TYPE STRUCTURE :

RES-GEST-F-TIERS

|->

CART-PROD [RES-MAJS-F-TIERS,
FICH-TIERS,
FICH-TIERS,
FICH-TIERS]

SPECIFICATION OF OBJECT TYPE RES-MAJS-F-TIERS

LEXICON :

OBJECTIVE :

résultats de mises à jour du fichier des tiers

OPERATIONS :

MAJS-F-TIERS : opérations de mise à jour du fichier des tiers

TYPES :

RES-MAJ-F-TIERS : résultat d'une mise à jour du fichier des tiers

OUTLINE :

OPERATIONS :

MAJS-F-TIERS : PICH-TIERS X DDES-MAJ-TIERS X DATE -> RES-MAJS-F-TIERS

TYPES :

res-majs-tiers : RES-MAJS-F-TIERS

FORMAL SPEC :

ASSOCIATED OPERATIONS :

MAJS-F-TIERS

EXPLICITATION OF TYPE STRUCTURE :

RES-MAJS-F-TIERS

|-> SEQ [RES-MAJ-F-TIERS]

SPECIFICATION OF OBJECT TYPE RES-MAJ-F-TIERS

LEXICON

OBJECTIVE

résultat d'une mise à jour du fichier des tiers (message d'erreur ou résultat d'une mise à jour correcte)

OPERATIONS

MAJ-F-TIERS : une opération de mise à jour du fichier des tiers

TYPES

RES-MAJ-TIERS-OK : résultats d'une mise-à-jour correcte

MESS-ERR-MAJ-TIERS : message d'erreur

OUTLINE

OPERATIONS

MAJ-F-TIERS : FICH-TIERS X DDE-MAJ-TIERS X DATE -> RES-MAJ-F-TIERS

TYPES

res-maj-tiers : RES-MAJ-F-TIERS

FORMAL SPEC

ASSOCIATED OPERATIONS

MAJ-F-TIERS

EXPLICITATION OF TYPE STRUCTURE

RES-MAJ-F-TIERS

|->

UNION [RES-MAJ-TIERS-OK,

CHARST]

SPECIFICATION OF OBJECT TYPE RES-MAJ-TIERS-OK

LEXICON

OBJECTIVE

résultat d'une mise à jour correcte du fichier des tiers (fichier mis à jour et carton-tiers reprenant les informations concernant le tiers crée, modifiée ou annulé)

OPERATIONS

MAJ-TIERS-OK : mise à jour correcte du fichier des tiers

CREA-TIERS : création d'un tiers

MODIF-TIERS : modification d'un tiers

ANNU-TIERS : annulation d'un tiers

TYPES

FICH-TIERS : fichier des tiers

CARTE-TIERS : carton tiers

OUTLINE

OPERATIONS

MAJ-TIERS-OK : FICH-TIERS
X DDE-MAJ-TIERS
-> RES-MAJ-TIERS-OK

CREA-TIERS : FICH-TIERS
X TIERS
-> RES-MAJ-TIERS-OK

MODIF-TIERS : FICH-TIERS
X DDE-MODI-TIERS
-> RES-MAJ-TIERS-OK

ANNU-TIERS : FICH-TIERS
X INDIC-T
-> RES-MAJ-TIERS-OK

TYPES

res-maj-tiers-ok : RES-MAJ-TIERS-OK

FORMAL SPEC

ASSOCIATED OPERATIONS

MAJ-TIERS-OK, CREA-TIERS,

MODIF-TIERS, ANNU-TIERS

EXPLICITATION OF TYPE STRUCTURE

RES-MAJ-TIERS-OK

|->

CART-PROD [FICH-TIERS,

CARTE-TIERS]

SPECIFICATION OF OBJECT TYPE FICH-TIERS

LEXICON :

OBJECTIVE :

fichier de tous les tiers avec lesquels la société est en contact (clients, fournisseurs, intermédiaires, ...)

OPERATIONS :

GEST-F-TIERS : gestion du fichier des tiers

MAJS-F-TIERS : opérations de mise à jour du fichier des tiers

EDIT-F-F, EDIT-F-C, EDIT-F-C-NTVA : éditions des fiches fournisseurs, clients et clients par numéro de TVA

MAJ-F-TIERS : opération de mise à jour du fichier des tiers

MAJ-TIERS-OK : opération de mise à jour correcte du fichier des tiers

DEF-MESS-ERR-MAJ-TIERS : définition de message d'erreur

CREA-TIERS : opération de création d'un tiers

MODIF-TIERS : opération de modification d'un tiers

ANNU-TIERS : opération d'annulation d'un tiers

TYPES :

TIERS : tiers pour la société (client, fournisseur, intermédiaire, ...)

OUTLINE :

OPERATIONS :

GEST-F-TIERS : FICH-TIERS X DATE X DDES-MAJ-TIERS X BOOLEAN X BOOLEAN
-> RES-GEST-F-TIERS

MAJS-F-TIERS : FICH-TIERS X DDES-MAJ-TIERS X DATE
-> RES-MAJS-TIERS

EDIT-F-F, EDIT-F-C, EDIT-F-C-NTVA : FICH-TIERS
-> FICH-TIERS

MAJ-F-TIERS : FICH-TIERS X DDE-MAJ-TIERS X DATE
-> RES-MAJ-TIERS

MAJ-TIERS-OK : FICH-TIERS X DDE-MAJ-TIERS
-> RES-MAJ-TIERS-OK

DEF-MESS-ERR-MAJ-TIERS : FICH-TIERS X DDE-MAJ-TIERS X DATE
-> CHARST

CREA-TIERS : FICH-TIERS X DDE-MAJ-TIERS
-> RES-MAJ-TIERS-OK

MODIF-TIERS : FICH-TIERS X DDE-MAJ-TIERS
-> RES-MAJ-TIERS-OK

ANNU-TIERS : FICH-TIERS X DDE-MAJ-TIERS
-> RES-MAJ-TIERS-OK

...

TYPES :

fich-tiers : fichier des tiers

fiches-fourn : FICH-TIERS

fiches-cli : FICH-TIERS

fiches-cli-ntva : FICH-TIERS

FORMAL SPEC :

ASSOCIATED OPERATIONS :

GEST-F-TIERS, MAJS-F-TIERS,

EDIT-F-F, EDIT-F-C,

EDIT-F-C-NTVA, MAJ-F-TIERS,

MAJ-TIERS-OK, DEF-MESS-ERR-MAJ-TIERS,

CREA-TIERS, MODIF-TIERS,

ANNU-TIERS, ...

EXPLICITATION OF TYPE STRUCTURE :

FICH-TIERS

|-> SEQ [TIERS]

SPECIFICATION OF OBJECT TYPE TIERS

LEXICON

OBJECTIVE

informations concernant un tiers avec lequel la société est en contact (client, fournisseur ou intermédiaire)

OPERATIONS

CREA-TIERS	: création d'un tiers
EDIT-CART-TIERS	: édition d'un carton tiers
OK-CREA-TIERS	: prédicat vrai si la demande de création d'un tiers est correcte

TYPES

INDIC-T	: indicatif de tiers
NOM-TIE	: nom de tiers
TITRE	: titre d'un tiers
ADR-TIE	: adresse d'un tiers (= nom de rue suivi du numero)
CAN-POS	: canton postal
LOCALITE	: localité
CODE-INS-P-AR	: code INS (pays + arrondissement)
REG-LING	: régime linguistique
NO-AS-TVA	: numéro d'assujetti à la TVA
CODE-BANQ	: code bancaire
NOM-ORG-FIN	: nom de l'organisme financier du tiers
CAN-POS-FI	: canton postal de l'organisme fin.
LOCALITE-FI	: localité de l'organisme fin.
N-CP-OF	: numéro de compte
NOM-OF-V	: nom de l'organisme financier pour virement
BEN-L1	: nom de bénéficiaire
BEN-L2	: adresse de bénéficiaire (nom de rue suivi du numéro)
BEN-L3	: canton postal et localité du bénéficiaire
BEN-L4	: communication
TAUX-INT	: code indiquant si un taux d'intérêt doit être appliqué en cas de recouvrement par traites
COND-PAI	: conditions de paiement
QUALITE-T	: qualité d'un tiers (client, fournisseur, ...)
HONORABILITE	: code d'honorabilité d'un tiers
ANCIEN-NO-VA	: ancien numéro de client de va
LIBELLE-MAJ	: libellé de mise à jour
DATE-DERNIER-SOLDE-NUL	: date de dernier solde nul

OUTLINE

OPERATIONS

CREA-TIERS	: FICH-TIERS X TIERS -> RES-MAJ-TIERS-OK
EDIT-CART-TIERS	: TIERS -> CARTE-TIERS
OK-CREA-TIERS	: FICH-TIERS X TIERS -> BOOLEEN

TYPES

tiers	: TIERS
tiers-modifie	: TIERS

FORMAL SPEC

ASSOCIATED OPERATIONS

CREA-TIERS, EDIT-CART-TIERS, OK-CREA-TIERS

EXPLICITATION OF TYPE STRUCTURE

TIERS

|->

CART-PROD [INDIC-T, NOM-TIE, TITRE,
ADR-TIE, CAN-POS, LOCALITE,
CODE-INS-P-AR, REG-LING,
NO-AS-TVA, CODE-BANQ,
NOM-ORG-FIN, CAN-POS-FI,
LOCALITE-FI, N-CP-OF, NOM-OF-V,
BEN-L1, BEN-L2, BEN-L3,
BEN-L4, TAUX-INT, COND-PAI,
QUALITE-T, HONORABILITE,
ANCIEN-NO-VA, LIBELLE-MAJ,
DATE-DERNIER-SOLDE-NUL]

SPECIFICATION OF OBJECT TYPE DDES-MAJ-TIERS

LEXICON

OBJECTIVE

demandes de mise à jour du fichier des tiers (création, modification ou suppression de tiers)

OPERATIONS

GEST-F-TIERS : gestion du fichier des tiers

MAJS-F-TIERS : opérations de mise à jour du fichier des tiers

TYPES

DDE-MAJ-TIERS : une demande de mise à jour du fichier des tiers (création, modification ou suppression de tiers)

OUTLINE

OPERATIONS

GEST-F-TIERS : FICH-TIERS X DATE
X DDES-MAJ-TIERS
X BOOLEAN X BOOLEAN
-> RES-GEST-F-TIERS

MAJS-F-TIERS : FICH-TIERS X DDES-MAJ-TIERS
X DATE
-> RES-MAJS-TIERS

TYPES

ddes-maj-tiers : DDES-MAJ-TIERS

FORMAL SPEC

ASSOCIATED OPERATIONS

GEST-F-TIERS, MAJS-F-TIERS

EXPLICITATION OF TYPE STRUCTURE

DDES-MAJ-TIERS

|->

SEQ [DDE-MAJ-TIERS]

SPECIFICATION OF OBJECT TYPE DDE-MAJ-TIERS

LEXICON

OBJECTIVE

une demande de mise à jour du fichier des tiers (création, modification ou suppression d'un tiers)

OPERATIONS

MAJ-F-TIERS : une opération de mise à jour du fichier des tiers

DEF-MESS-ERR-MAJ-TIERS : définition de message d'erreur

MAJ-TIERS-OK : une opération de mise à jour correcte du fichier des tiers

TYPES

TIERS : (demande de création d'un) tiers

DDE-MODI-TIERS : demande de modification d'un tiers

INDIC-T : demande d'annulation d'un tiers (indicatif)

OUTLINE

OPERATIONS

MAJ-F-TIERS : FICH-TIERS X DDE-MAJ-TIERS X DATE
-> RES-MAJ-TIERS

DEF-MESS-ERR-MAJ-TIERS : FICH-TIERS X DDE-MAJ-TIERS X DATE
-> CHARST

MAJ-TIERS-OK : FICH-TIERS X DDE-MAJ-TIERS
-> RES-MAJ-TIERS-OK

TYPES

dde-maj-tiers : DDE-MAJ-TIERS

FORMAL SPEC

ASSOCIATED OPERATIONS

MAJ-F-TIERS, DEF-MESS-ERR-MAJ-TIERS,

MAJ-TIERS-OK

EXPLICITATION OF TYPE STRUCTURE

DDE-MAJ-TIERS

|->

UNION [TIERS, DDE-MODI-TIERS, INDIC-T]

SPECIFICATION OF OBJECT TYPE DDE-MODI-TIERS

LEXICON

OBJECTIVE

Demande de modification d'un tiers
(indicatif suivi des champs à
modifier(nom du champ suivi de
la nouvelle valeur))

OPERATIONS

MODIF-TIERS : modification d'un tiers

OK-MODIF-TIERS : prédicat indiquant si la demande
de modification est correcte

MODIF-CHPS-TIERS : modification de champs de tiers

TYPES

INDIC-T : indicatif d'un tiers

CHP-A-MODIFIER : nom du champ à modifier

NOUV-VAL : nouvelle valeur

OUTLINE

OPERATIONS

MODIF-TIERS : FICH-TIERS
X DDE-MODI-TIERS
-> RES-MAJ-TIERS-OK

OK-MODIF-TIERS : FICH-TIERS
X DDE-MODI-TIERS
-> BOOLEEN

MODIF-CHPS-TIERS : FICH-TIERS
X DDE-MODI-TIERS
-> TIERS

TYPES

dde-modi-tiers : DDE-MODI-TIERS

FORMAL SPEC

ASSOCIATED OPERATIONS

MODIF-TIERS, OK-MODIF-TIERS,

MODIF-CHPS-TIERS

EXPLICITATION OF TYPE STRUCTURE

DDE-MODI-TIERS

|->

CART-PROD [INDIC-T,
CHPS-A-MODIFIER]

SPECIFICATION OF OBJECT TYPE CHPS-A-MODIFIER

LEXICON

OBJECTIVE

liste des champs à modifier
(nom de champ + nouvelle valeur)

OPERATIONS

MODIF-CHPS-TIERS : modification de champs de tiers

TYPES

NOM-CHP : nom du champ à modifier

NOUV-VAL : nouvelle valeur

OUTLINE

OPERATIONS

MODIF-CHPS-TIERS : PICH-TIERS
X DDE-MODI-TIERS
-> RES-MAJ-TIERS-OK

TYPES

FORMAL SPEC

ASSOCIATED OPERATIONS

MODIF-CHPS-TIERS

EXPLICITATION OF TYPE STRUCTURE

CHPS-A-MODIFIER

|->

SEQ [CART-PROD [NOM-CHP,
NOUV-VAL]]

SPECIFICATION OF OBJECT TYPE CARTE-TIERS

LEXICON

OBJECTIVE

informations concernant un tiers
(toutes les informations disponibles
pour un tiers créé ou modifié,
et l'indicatif suivi de la mention
"ANNULE" pour un tiers supprimé)

OPERATIONS

EDIT-CART-TIERS : édition des cartes tiers

TYPES

TIERS : un tiers

OUTLINE

OPERATIONS

EDIT-CART-TIERS : TIERS
-> CARTE-TIERS

TYPES

carton-tiers : CARTE-TIERS

FORMAL SPEC

ASSOCIATED OPERATIONS

EDIT-CART-TIERS

EXPLICITATION OF TYPE STRUCTURE

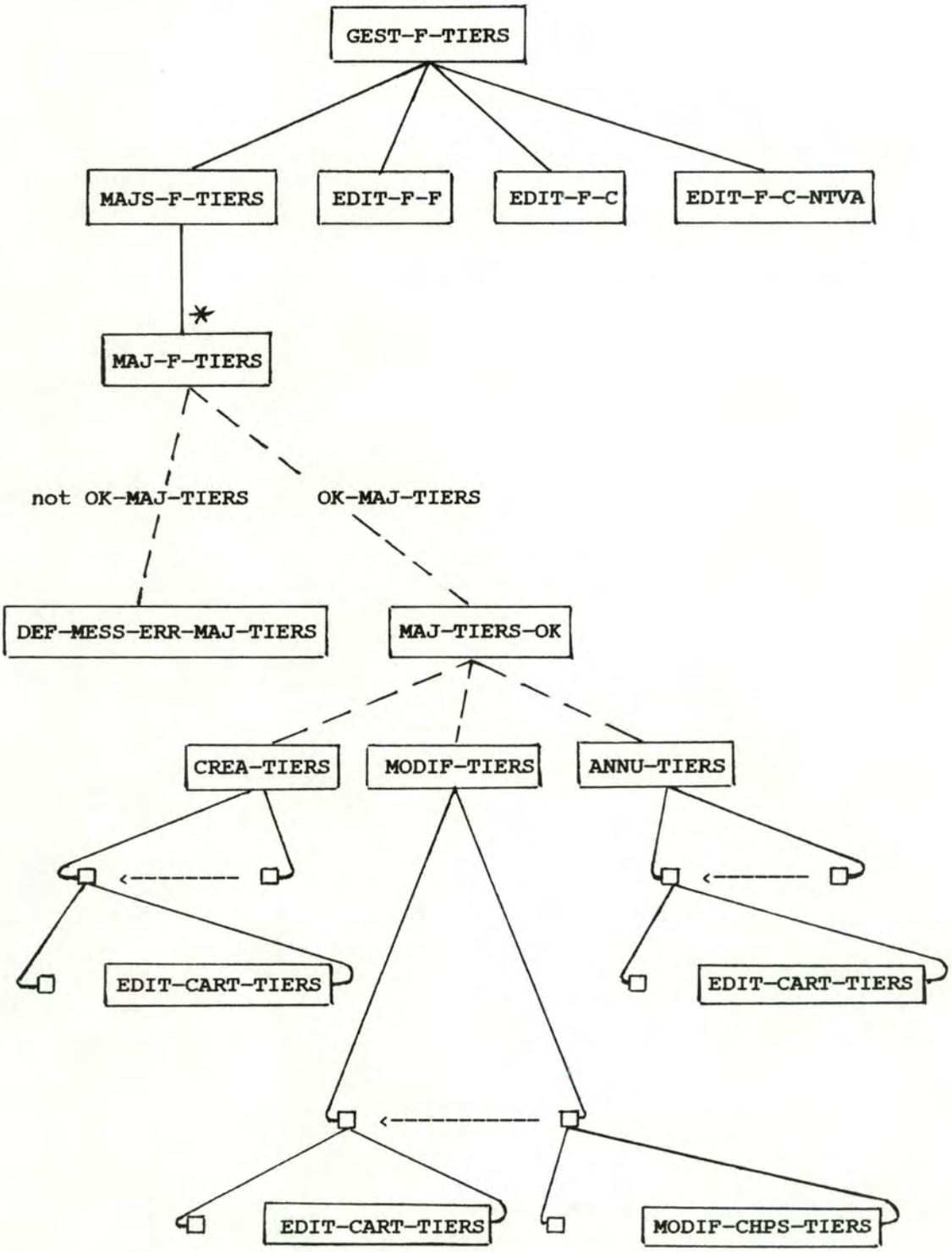
CARTE-TIERS

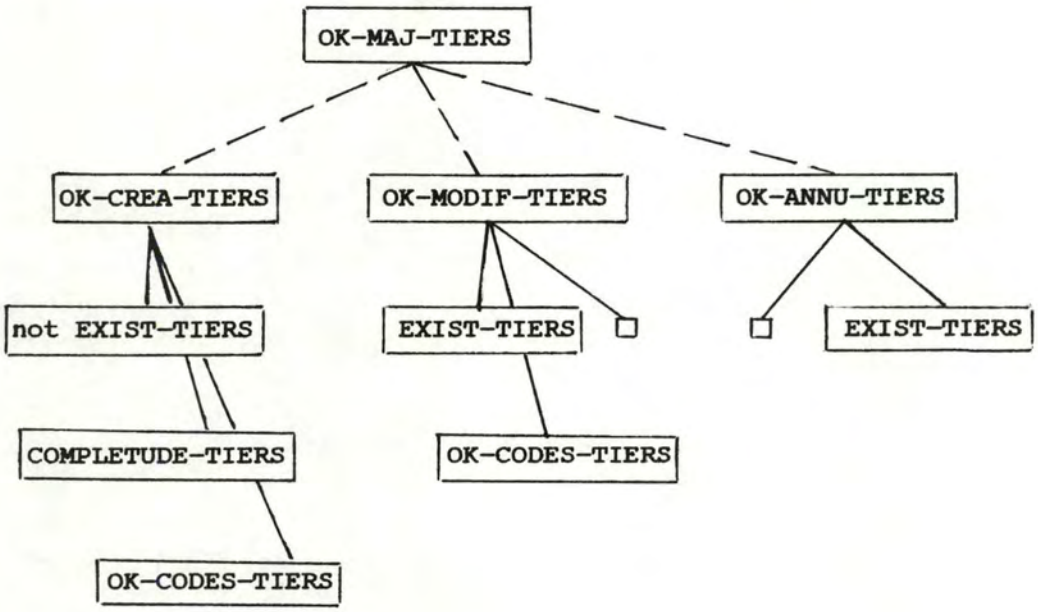
|->

UNION [TIERS,

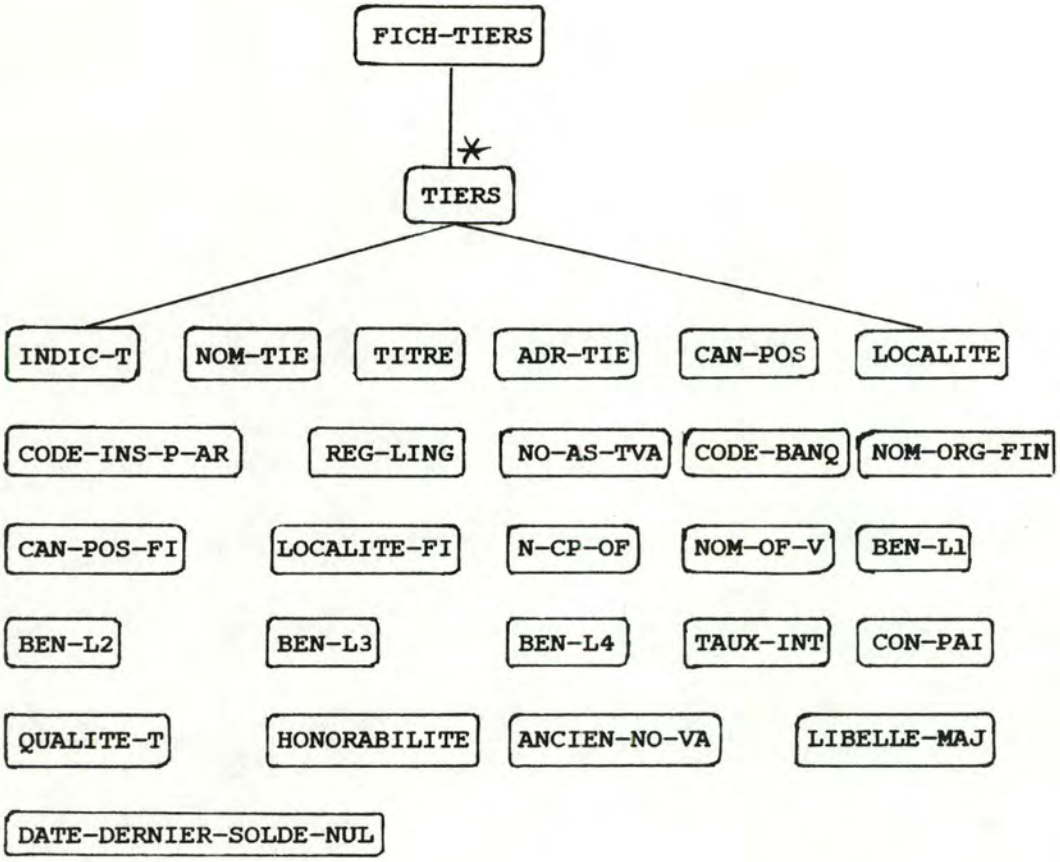
CART-PROD [INDIC-T,
"ANNULE"]

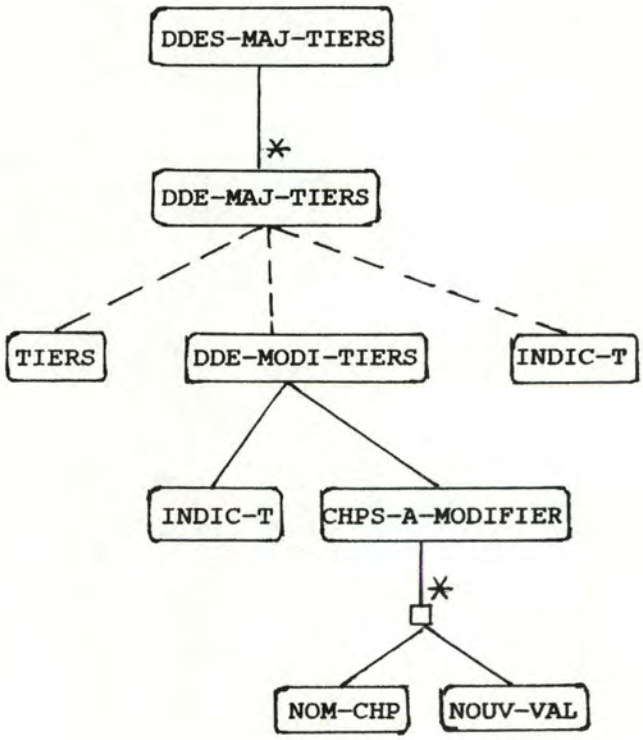
Schémas de l'énoncé du problème

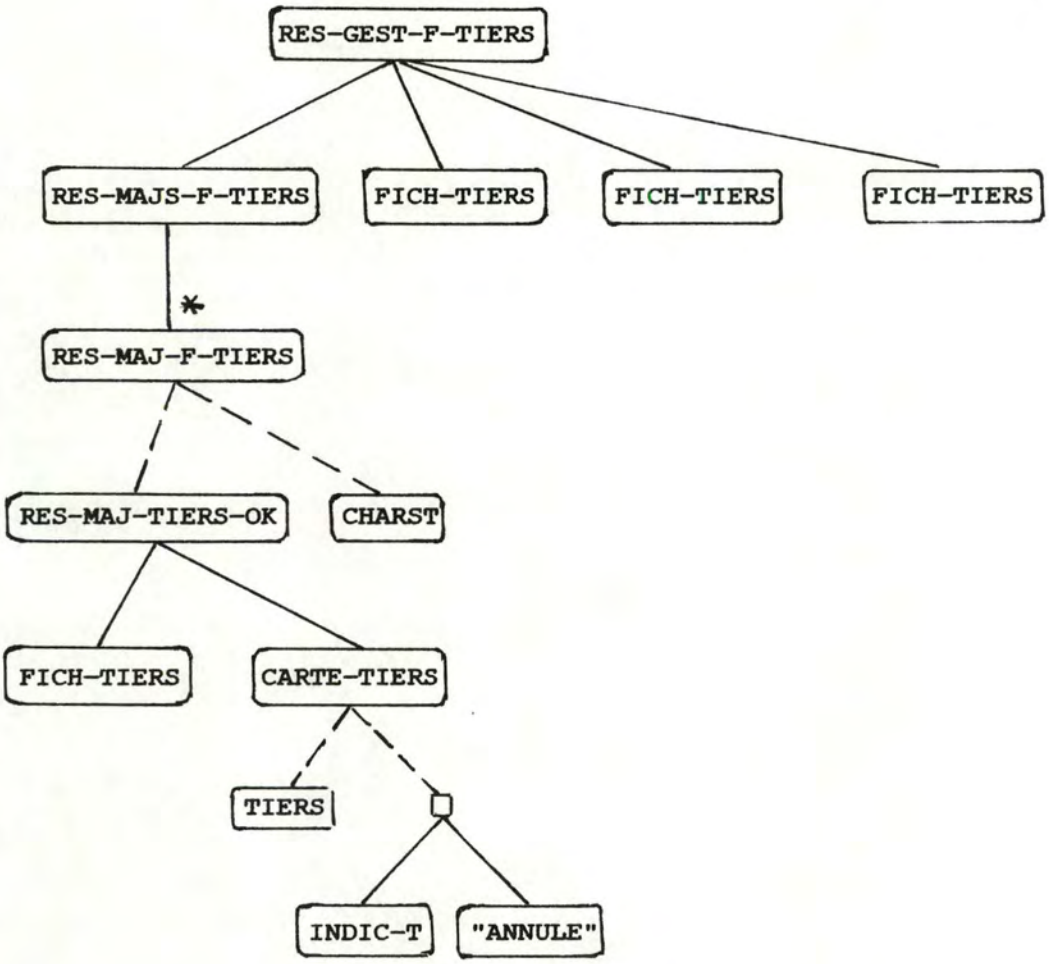



OK-MAJ-TIERS
OK-CREA-TIERS
OK-MODIF-TIERS
OK-ANNU-TIERS
not EXIST-TIERS
EXIST-TIERS
EXIST-TIERS
COMPLETUDE-TIERS
OK-CODES-TIERS
OK-CODES-TIERS

Schémas de l'univers du problème






RES-GEST-F-TIERS
RES-MAJS-F-TIERS
FICH-TIERS
FICH-TIERS
FICH-TIERS
*
RES-MAJ-F-TIERS
RES-MAJ-TIERS-OK
CHARST
FICH-TIERS
CARTE-TIERS
TIERS
INDIC-T
"ANNULE"

A N N E X E 2 :

PROGRAMME PROLOG RESULTANT DE LA TRADUCTION MANUELLE

Programme PROLOG résultant de la traduction manuelle des spécifications RSL de la gestion du fichier des tiers. Ce programme constitue une petite partie de l'implémentation réalisée.

Relevons la correspondance entre les blocs de spécifications présentés en Annexe 1 et les procédures prolog présentées ici.

Au bloc GEST-F-TIERS correspond la procédure gest_f_tiers

EDIT-F-F	edit-f-f
EDIT-F-C	edit-f-c
EDIT-F-C-NTVA	edit-f-c-ntva
MAJS-F-TIERS	majs-f-tiers
MAJ-F-TIERS	maj-f-tiers
MAJ-TIERS-OK	maj-tiers-ok
CREA-TIERS	crea-tiers
MODIF-TIERS	modif-tiers
MODIF-CHPS-TIERS	modif-chps-tiers
ANNU-TIERS	annu-tiers
EDIT-CART-TIERS	edit-cart-tiers

```
/*****************************************************************/
/*              GESTION  DU  FICHIER  DES  TIERS                 */
/*_______________________________________________________________*/


gest_f_tiers([ Res_majs_tiers, Fiches_fourn, Fiches_cli,
               Fiches_cli_ntva], Fich_tiers, Date_du_jour,
               Ddes_maj_tiers, Dde_edit_f_c, Dde_edit_f_c_ntva )

  :- (( eval( 1, jj( Date_du_jour)),
        edit_f_f( Fiches_fourn,Fich_tiers));
      edit_f_f( Fiches_fourn,[])),

     (( Dde_edit_f_c = 'vrai',
        edit_f_c( Fiches_cli, Fich_tiers));
      edit_f_c( Fiches_cli,[]) ),

     (( Dde_edit_f_c_ntva = 'vrai',
        edit_f_c_ntva( Fiches_cli_ntva, Fich_tiers) ) ;
      edit_f_c_ntva( Fiches_cli_ntva, [] ) ), !,

     majs_f_tiers( Res_majs_tiers, Fich_tiers, Ddes_maj_tiers,
                   Date_du_jour).


/*****************************************************************/
/*      E D I T I O N S   D E S   F I C H E S   T I E R S        */
/*_______________________________________________________________*/


edit_f_f([],[]).

edit_f_f( [ Tiers | Fiches_fourn], [Tiers | Seq_tiers])
   :-  eval( 100, quali_t( Tiers)),
       edit_f_f(Fiches_fourn, Seq_tiers).

edit_f_f( Fiches_fourn, [ Tiers | Seq_tiers])
   :-  edit_f_f( Fiches_fourn, Seq_tiers).

/*---------------------------------------------------------------*/

edit_f_c([],[]).

edit_f_c( [ Tiers | Fiches_cli], [Tiers | Seq_tiers])
   :-  eval( 1000, quali_t( Tiers)),
       edit_f_c( Fiches_cli, Seq_tiers).

edit_f_c( Fiches_cli, [ Tiers | Seq_tiers])
   :-  edit_f_c( Fiches_cli, Seq_tiers).

/*---------------------------------------------------------------*/
```

```
edit_f_c_ntva( [ ], [ ] ).

edit_f_c_ntva( Fiches_cli_ntva,Fich_tiers)
  :- eval( Fich_tiers_trie,
           sort( Fich_tiers, n_as_tva ) ),
     edit_f_c_ntva_2( Fiches_cli_ntva, Fich_tiers_trie).

edit_f_c_ntva_2( [ ], [ ] ).
edit_f_c_ntva_2( [ Tiers | Fiches_cli_ntva], [Tiers | Seq_tiers])
   :-  eval( 1000, quali_t( Tiers)),
       edit_f_c_ntva_2( Fiches_cli_ntva, Seq_tiers).

edit_f_c_ntva_2( Fiches_cli_ntva, [Tiers|Seq_tiers])
   :-  edit_f_c_ntva_2( Fiches_cli_ntva, Seq_tiers).


/*****************************************************************/

edit_cart_tiers( Carton_tiers, Tiers )
  :- eval( Lib_maj, lib_maj( Tiers)),
     ( ( Lib_maj = annu,
         eval( Indic, indic_t( Tiers)),
         Carton_tiers = [Indic, 'ANNULE' ] ) ;
       ( Lib_maj == annu,
         Carton_tiers = Tiers ) ).


/*****************************************************************/
/*          M I S E S   A   J O U R   D E   T I E R S            */
/*****************************************************************/


majs_f_tiers( [[ X , [ ] ]], X, [], _).

majs_f_tiers( [Res_maj_tiers| Res_majs_tiers], Fich_tiers,
              [Dde_maj_tiers| Ddes_maj_tiers], Date_du_jour)

  :- maj_f_tiers( Res_maj_tiers, Fich_tiers, Dde_maj_tiers,
                  Date_du_jour),

     Res_maj_tiers = [ Fich_tiers_res, _],

     majs_f_tiers( Res_majs_tiers, Fich_tiers_res, Ddes_maj_tiers,
                   Date_du_jour).
```

```
/*****************************************************************/
/*            M I S E   A   J O U R   D E   T I E R S            */
/*****************************************************************/


maj_f_tiers( Res_maj_tiers_ok, Fich_tiers, Dde_maj_tiers,
             Date_du_jour)
  :- ok_maj_tiers( Fich_tiers, Dde_maj_tiers, Date_du_jour),
     maj_tiers_ok(Res_maj_tiers_ok,Fich_tiers,Dde_maj_tiers),! .


maj_f_tiers( [ Fich_tiers, Mess_err_maj_tiers], Fich_tiers,
             Dde_maj_tiers, Date_du_jour)
  :- def_mess_err_maj_tiers( Mess_err_maj_tiers, Fich_tiers,
                             Dde_maj_tiers, Date_du_jour).


/*****************************************************************/
/*                  MISE A JOUR  CORRECTE  DE  TIERS             */
/*****************************************************************/


maj_tiers_ok( Res_maj_tiers_ok, Fich_tiers, Dde_maj_tiers)
  :- eval( true, type_of( tiers, Dde_maj_tiers)), !,
     crea_tiers( Res_maj_tiers_ok, Fich_tiers, Dde_maj_tiers).


maj_tiers_ok( Res_maj_tiers_ok, Fich_tiers, Dde_maj_tiers)
  :- eval( true, type_of( dde_modi_tiers, Dde_maj_tiers)), !,
     modif_tiers( Res_maj_tiers_ok, Fich_tiers, Dde_maj_tiers).


maj_tiers_ok( Res_maj_tiers_ok, Fich_tiers, Dde_maj_tiers)
  :- eval( true, type_of( indic_t, Dde_maj_tiers)), !,
     annu_tiers( Res_maj_tiers_ok, Fich_tiers, Dde_maj_tiers).
```

```
/*****************************************************************/
/*                    CREATION  D' UN  TIERS                     */
/*****************************************************************/


crea_tiers( [Fich_tiers_res, Carton_tiers], Fich_tiers,
           [Indic_t, Nom_tie, Titre, Adr_tie, Can_pos, Localite,
            Ins_p_ar, Escompte, Reg_lin, N_as_tva, Cod_ban,
            Nom_o_fi, Can_pos_fi, Localite_fi, N_cp_of,
            Nom_of_v, Ben_l_1, Ben_l_2, Ben_l_3, Ben_l_4,
            Taux_int, Con_pai, Quali_t, Honor_t, Anc_n_va,
            Lib_maj, Dern_solde_0 ] )

  :- Nouv_tiers =  [ Indic_t, Nom_tie, Titre, Adr_tie, Can_pos,
                     Localite, Ins_p_ar, Escompte, Reg_lin,
                     N_as_tva, Cod_ban, Nom_o_fi, Can_pos_fi,
                     Localite_fi, N_cp_of, Nom_of_v, Ben_l_1,
                     Ben_l_2, Ben_l_3, Ben_l_4, Taux_int,
                     Con_pai, Quali_t, Honor_t, Anc_n_va,
                     'crea', 000000 ],

     edit_cart_tiers( Carton_tiers, Nouv_tiers),

     insert( Fich_tiers_res, Fich_tiers, Nouv_tiers, indic_t).


/*****************************************************************/
/*                   MODIFICATION  D' UN  TIERS                  */
/*****************************************************************/


modif_tiers( [ Fich_tiers_res, Carton_tiers], Fich_tiers,
             [ Indic_dde| Chps_a_modifier])
  :- modif_chps_tiers( Tiers_modifie, Fich_tiers,
                       [Indic_dde|Chps_a_modifier]),

     edit_cart_tiers( Carton_tiers, Tiers_modifie),

     eval( Fich_tiers_filtre,
           filter( Fich_tiers,
                   not( eval( Indic_dde, indic_t( x)) ) )),

     insert( Fich_tiers_res, Fich_tiers_filtre, Tiers_modifie,
             indic_t ).


/*****************************************************************/
```

```
modif_chps_tiers( [], [], _).

modif_chps_tiers( Tiers_modifie, Fich_tiers,
                  [Indic_dde|Chps_a_modifier])
  :-  eval( Tiers_a_modifier,
            aplat( filter( Fich_tiers,
                           eval( Indic_dde, indic_t(x)) ) )),
      modif_chps( Tiers_modifie, Tiers_a_modifier,
                  [[lib_maj, modi] | Chps_a_modifier]).


/*****************************************************************/
/*                    ANNULATION  D' UN  TIERS                   */
/*****************************************************************/


annu_tiers([Fich_tiers_res, Carton_tiers], Fich_tiers, Indic_t )
  :- eval( Tiers_annule_int,
           aplat( filter( Fich_tiers,
                          eval( Indic_t, indic_t( x))) )),
     modif_chp( Tiers_annule, Tiers_annule_int,[lib_maj, annu]),
     edit_cart_tiers( Carton_tiers, Tiers_annule),
     eval( Fich_tiers_res,
           filter( Fich_tiers,
                   not( eval( Indic_t, indic_t( x)) ) )).


 ...
```

Procédure Prolog "eval" implémentant les opérations de la bibliothèque de spécification (filtre d'une suite, concaténation de deux suites, ...).

```
:- op(800, xfy, :).

/*************************************************************/
/*              E V A L U A T E U R                          */
/*___________________________________________________________*/

eval( X, X)                 :- functor(X, _, 0), !.

eval( .(Vx,Vy), .(X,Y) ) :- eval( Vx, X),
                             eval( Vy, Y), !.

eval( :(Vx,Vy), :(X,Y) ) :- eval( Vx, X),
                             eval( Vy, Y).

eval( Val, time(X,Y))       :- eval( Vx, X), eval( Vy, Y),
                               ptime( Val, Vx, Vy).

eval( Val, access(X,Y))     :- eval( Vx, X), eval( Vy, Y),
                               paccess( Val, Vx, Vy).

eval( Val, conspc( L ))     :- eval( Val, L).


/*-----------------------------------------------------------*/
/*     Evaluation de la somme des éléments d'une liste       */
/*-----------------------------------------------------------*/

eval( Val, sum( [Val])) :- number( Val).

eval( Val, sum( Liste_num))
  :- eval( [ Tete | Queue ], Liste_num),
     eval( Sum_queue, sum( Queue)), Val is Tete + Sum_queue.


/*-----------------------------------------------------------*/
/*      Evaluation de la longueur d'une liste                */
/*-----------------------------------------------------------*/

eval( 0, lgth( L )) :- eval( [], L).

eval( L, lgth( Seq ) ) :- eval( [ Tete | Queue], Seq),
                          eval( L_queue, lgth( Queue)),
                          L is L_queue + 1, !.

/*-----------------------------------------------------------*/

eval( Val, conc(X,Y)) :- eval( Vx, X), eval( Vy, Y),
                         pconc( Val, Vx, Vy).
```

```
/*-------------------------------------------------------------*/

eval( Val, aplat( Seq)) :- eval( Vseq, Seq), !,
                           paplat( Val, Vseq).

/*-------------------------------------------------------------*/

eval( Val, app(X,Y)) :- eval( Vx, X),  eval( Vy, Y), !,
                        papp( Val, X, Y).


/*-------------------------------------------------------------*/

eval( Val, jj( Date))  :-  Val is Date // 10000.
eval( Val, mm( Date))  :-  Val is ( Date // 100) mod 100.
eval( Val, aa( Date))  :-  Val is Date mod 100.

/*-------------------------------------------------------------*/

eval( Tete, ith( L, X)) :- eval( 1, X), eval( [Tete | _], L).

eval( Val, ith( L, X))
  :- eval( I, X), eval( [Tete | Queue], L), J is I - 1,
     eval( Val, ith( Queue, J)).

/*-------------------------------------------------------------*/

eval( Val, filter( Seq, Pre))  :-  eval( Vseq, Seq), !,
                                   pfilter( Val, Vseq, Pre, x).

/*-------------------------------------------------------------*/

eval( Val, sort( Seq, Cle))  :-  eval( Vseq, Seq),
                                 psort( Val, Seq, Cle).

/*-------------------------------------------------------------*/

eval( X, max( X, Y))  :-  eval( Vx, X), eval( Vy, Y), Vx >= Vy.
eval( Y, max( X, Y))  :-  eval( Vx, X), eval( Vy, Y), Vx =< Vy.


/*-------------------------------------------------------------*/
/*       Evaluation de l'égalité entre deux éléments           */
/*-------------------------------------------------------------*/

eval( true, eq(X,Y)) :- eval( Val, X), eval( Val, Y).
eval( false, eq(X,Y)) :- not( eval( true, eq(X,Y))).


/*************************************************************/

ptime(X,Y,Z) :- integer(Y), integer(Z), !, X is Y * Z.
ptime(X,Y,Z) :- integer(X), integer(Z), !, Y is X / Z.
ptime(X,Y,Z) :- integer(Y), integer(Z), !, Z is X / Y.

/*-------------------------------------------------------------*/
```

```
psum(X,Y,Z) :- integer(Y), integer(Z), X is Y + Z, !.
psum(X,Y,Z) :- integer(X), integer(Z), Y is X - Z, !.
psum(X,Y,Z) :- integer(X), integer(Y), Z is X - Y, !.

/*-----------------------------------------------------------------*/

paccess( Val, [Cle : Val | _], Cle).

paccess( Val, [ _ | L ], Cle)  :-  paccess( Val, L, Cle).

/*-----------------------------------------------------------------*/

paplat( X, [X]).

paplat( [ Tete | Queue], [[ Tete | X] | Y])
 :- ( X == [], paplat( Queue, [ X | Y ]) )  ;
    ( X = [], paplat( Queue, Y) ).

/*-----------------------------------------------------------------*/

papp( [X], [], X).

papp( [Tete | Queue_res], [Tete | Queue], Elt)
     :- papp( Queue_res, Queue, Elt).

/*-----------------------------------------------------------------*/

pconc( L, [], L).

pconc( [X | L3], [X | L1], L2 ) :- pconc( L3, L1, L2).

/*-----------------------------------------------------------------*/

pfilter( [], [], _, _ ).

pfilter( [El | Sres], [El | Sarg], Pre, Var)
  :- substitute( El, Var, Pre, Goal), Goal, !,
     pfilter( Sres, Sarg, Pre, Var).

pfilter( Sres, [ El | Sarg], Pre, Var)
  :- pfilter( Sres, Sarg, Pre, Var).
```

```
/*-------------------------------------------------------------*/
/*   substitute( New, Old, Val, Newval)                        */
/*   <=> remplacer Old par New dans Val donne Newval           */
/*-------------------------------------------------------------*/

substitute( New, Old, Old, New) :- !.

substitute( New, Old, Val, Val) :- atomic( Val), !.

substitute( New, Old, Val, Newval)
   :- functor( Val, Fn, N),
      functor( Newval, Fn, N),
      subst_args( N, New, Old, Val, Newval).

/*-------------------------------------------------------------*/

subst_args( 0,_,_,_,_) :- !.

subst_args( N, New, Old, Val, Newval)
  :- arg( N, Val, Oldarg),
     arg( N, Newval, Newarg),
     substitute( New, Old, Oldarg, Newarg),
     N1 is N-1,
     subst_args( N1, New, Old, Val,Newval).


/*-------------------------------------------------------------*/
/*  Tri par insertion : psort( Res, Arg, Cle)                  */
/*   <=> Res est le resultat du tri de Arg suivant la          */
/*       clé Cle, ou Cle est un nom de champ                   */
/*-------------------------------------------------------------*/

psort( [], [], _).

psort( Seq_res, [Tete | Queue], Cle)
  :- psort( Queue_triee, Queue, Cle),
     insert( Seq_res, Queue_triee, Tete, Cle).

/*-------------------------------------------------------------*/

insert( [Tete | Queue_res], [ Tete| Queue_arg], Elt, Cle)
  :- eval( true, avant( Tete, Elt, Cle)),
     insert( Queue_res, Queue_arg, Elt, Cle).

insert( [ Elt | Liste], Liste, Elt, Cle).
```

```
/*************************************************************/
/*      ACCES  AUX  DIFFERENTS  CHAMPS  D'UN  TIERS           */
/*___________________________________________________________*/

eval( Val, indic_t(X))
 :- eval( [ Val,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,
            _,_,_,_], X).

eval( Val, app_sap(X))
 :- eval( [[ Val,_,_,_],_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,
           _,_,_,_,_,_,_,_], X).

eval( Val, num_soc(X))
 :- eval( [[_,Val,_,_],_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,
           _,_,_,_,_,_,_,_], X).

eval( Val, num_ger(X))
 :- eval( [[_,_,Val,_],_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,
           _,_,_,_,_,_,_,_], X).

eval( Val, num_tie(X))
 :- eval( [[_,_,_,Val],_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,
           _,_,_,_,_,_,_,_], X).

eval( Val, nom_tie(X))
 :- eval( [[_,_,_,_],Val,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,
           _,_,_,_,_,_,_,_], X).

eval( Val, titre(X))
 :- eval( [[_,_,_,_],_,Val,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,
           _,_,_,_,_,_,_,_], X).

eval( Val, adr_tie(X))
 :- eval( [[_,_,_,_],_,_,Val,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,
           _,_,_,_,_,_,_,_], X).

 ...

/*************************************************************/
```

A N N E X E 3 :

DEFINITION LDF DE LA SYNTAXE DU LANGAGE

DE SPECIFICATION RSL

La définition ci-dessous est plus complète que le modèle présenté au chapitre 2. Elle reprend la définition des différents blocs de spécification.

```
(* ldf *)
definition de rsl;


<point_d_entree>  ::= <list_oper_block> <list_obj_type_block> ;

L <list_oper_block>  ::= <oper_block>+ ;

<oper_block>     ::= @ "SPECIFICATION OF OPERATION" <oper_name>
                     $ "--------------------------"
                     # <body_oper_block>
                     @ "END" <oper_name> ^"." ;


<body_oper_block>        ::= $ "FORMAL SPEC :"
                               # <formal_spec_op>
                           @ "OUTLINE :"
                               # <outline_op>
                           @ "LEXICON :"
                               # <lexicon_op> ;



L <list_obj_type_block> ::= <obj_type_block>+ ;

<obj_type_bloc>  ::= @ "SPECIFICATION OF OBJECT TYPE" <type_name>
                     $ "----------------------------"
                     # <body_obj_type_block>
                     @ "END" <type_name> ^"." ;


<body_obj_type_block>    ::= $ "FORMAL SPEC :"
                               # <formal_spec_obj>
                           @ "OUTLINE :"
                               # <outline_obj>
                           @ "LEXICON :"
                               # <lexicon_obj> ;



SC <formal_spec_op>  ::= <probl_tree_op> | <predef_op> ;

<probl_tree_op>    ::=   "RESULT :" <obj_name>
                       $ "ARGUMENTS :" <list_arg_names>
                       $ "EXPLICITATION OF INPUT-OUTPUT ASSERTION :"
                       # <explicitation_io_assertion>
                       $ <explicitation_precondition> ;


SC <explicitation_precondition> ::= <empty> | <expl_precond_nv> ;

<expl_precond_nv>  ::=   "EXPLICITATION OF PRECONDITION :"
                       # <predicate>
                       # "|->"
                       # <predicate> ;
```

```
<predef_op>          ::= <list_predef_op> ";" ;

L <list_predef_op>   ::= <equation>+ ";" % ;

<explicitation_io_assertion> ::=   <op_expression>
                                 $ "|->"
                                 $ <op_explicitation> ;

<op_expression>   ::= <obj_name> "="
                      <oper_name> ^"(" ^ <list_arg_names> ^")" ;

SC <op_explicitation>    ::=   <terminal_explicitation>
                             | <or_explicitation>
                             | <and_explicitation>
                             | <with_explicitation>
                             | <forall_explicitation>
                             | <guarded_explicitation>
                             | <except_explicitation>
                             | <op_explicitation_parenth> ;

<terminal_explicitation> ::= <obj_name> "=" <term> ;

<or_explicitation>       ::=   <op_explicitation>
                             $ "or"
                             $ <op_explicitation> ;

<and_explicitation>      ::=   <op_explicitation>
                             $ "and"
                             $ <op_explicitation> ;

<with_explicitation>     ::=   <op_explicitation>
                             $ "with"
                             $ <op_explicitation> ;

<forall_explicitation>   ::=  "for all" <obj_name> ":"
                                   <pred_in_forall>
                                # <op_explicitation> ;

SC <pred_in_forall>      ::= <empty> | <pred_in_forall_nv> ;

<pred_in_forall_nv>      ::= "{" <predicate> "}" <empty> ":" ;

SC <guarded_explicitation> ::=   <pred_and_op_explicitation>
                               | <if_op_explicitation> ;

<pred_and_op_explicitation>
         ::= "{" <predicate> "}" <empty> # "and"
            # <op_explicitation> ;

SC <if_op_explicitation> ::=   <if_op_expl_with_otherwise>
                             | <if_op_expl_without_otherwise> ;

<if_op_expl_with_otherwise> ::=   <l_if_pred_op_expl>
                                $ <otherwise_op_expl> ;

<if_op_expl_without_otherwise> ::=  <l_if_pred_op_expl> ;
```

```
L <l_if_pred_op_expl>    ::= <if_pred_op_expl>+ ^"," % ;

<if_pred_op_expl>        ::= <op_explicitation>
                             # "if" <empty> "{" <predicate> "}" ;

<otherwise_op_expl>      ::= <op_explicitation> "otherwise" ;

<except_explicitation>   ::=   <obj_name> "=" <obj_name>
                             $ "except" <empty>
                               "(" <decomp_except> ")" ;

SC <decomp_except>       ::=   <except_expression>
                             | <and_except>
                             | <with_except>
                             | <decomp_except_parenth> ;

<except_expression>      ::= <specbase_expression> "=" <term> ;

<and_except>             ::= <decomp_except>
                                 "and" <decomp_except> ;

<with_except>            ::= <decomp_except>
                                 "with" <op_explicitation> ;

<decomp_except_parenth>  ::= "(" <decomp_except> ")" ;

<op_explicitation_parenth> ::= "[" <op_explicitation> "]" ;


<formal_spec_obj> ::=   "ASSOCIATED OPERATIONS :"
                        <list_oper_names>
                      $ "EXPLICITATION OF TYPE STRUCTURE :"
                      # <type_name>
                      # "|->"
                      # <type_explicitation>
                      $ <explicitation_of_invariant> ;

SC <explicitation_of_invariant> ::= <empty>
                                  | <explicitation_of_inv_nv> ;

<explicitation_of_inv_nv> ::= "EXPLICITATION OF INVARIANT :"
                                # <predicate>
                                # "|->"
                                # <predicate> ;

SC <type_explicitation> ::=   <type>
                            | <union_expl>
                            | <cartesian_product_expl>
                            | <sequence_expl>
                            | <set_expl>
                            | <table_expl>
                            | <reuse_expl>
                            | <specialize_expl> ;
```

```
L <list_type_explicitation> ::= <type_explicitation>+ ^"," % ;

<union_expl>  ::= "UNION ["^ <list_type_explicitation> ^"]" ;

<cartesian_product_expl>
              ::= "CART-PROD ["^ <list_type_explicitation> ^"]" ;

<sequence_expl> ::= "SEQ ["^ <type_explicitation> ^"]" ;

<set_expl>      ::= "SET ["^ <type_explicitation> ^"]" ;

<table_expl>    ::= "TABLE ["^ <type_explicitation> "|->"
                               <type_explicitation> ^"]" ;

<reuse_expl>      ::= "IS ["^ <type_explicitation> ^"]" ;

<specialize_expl> ::= "IS_A ["^ <type_explicitation> ^"]" ;


SC <predicate>          ::=   <atom>
                            | <not_predicate>
                            | <or_predicate>
                            | <and_predicate>
                            | <with_predicate>
                            | <implic_predicate>
                            | <equiv_predicate>
                            | <forall_predicate>
                            | <existence_predicate>
                            | <parenth_predicate> ;

<not_predicate>         ::= "not" <predicate> ;

<or_predicate>          ::=   <predicate>
                            $ "or"
                            $ <predicate> ;

<and_predicate>         ::=   <predicate>
                            $ "and"
                            $ <predicate> ;

<with_predicate>        ::=   <predicate>
                            $ "with"
                            # <op_explicitation> ;

<implic_predicate>      ::=   <predicate>
                            $ "=>"
                            $ <predicate> ;

<equiv_predicate>       ::=   <predicate>
                            $ "<=>"
                            $ <predicate> ;

<forall_predicate>      ::=   "( for all" <obj_name> ")"
                                # <predicate> ;
```

```
<existence_predicate>    ::=   "( exist" <obj_name> ")"
                                 # <predicate> ;

<parenth_predicate>      ::= "{" <predicate> "}" ;

SC <atom>                ::=   <true> | <false>
                             | <oper_name_terms>
                             | <specbase_expression>
                             | <relational_expr> ;

<oper_name_terms>        ::= <oper_name> ^"(" ^ <list_terms> ^")" ;

L <list_terms>           ::= <term>* ^"," % ;

SC <term>                ::=   <obj_name>
                             | <constant>
                             | <arithm_expr>
                             | <oper_name_terms>
                             | <specbase_expression>
                             | <term_parenth> ;

<term_parenth>           ::= "(" <term> ")" ;

SC <constant>            ::= <number> | <text> ;

<arithm_expr>            ::= <term> <arithm_oper> <term> ;

SC <arithm_oper>         ::= <add> | <subst> | <mult> |
                             <div> | <mod>   | <intdiv> ;

<relational_expr>        ::= <term> <relational_oper> <term> ;

SC <relational_oper>     ::= <eq> | <diff> | <inf> | <sup>
                           | <eqinf> | <eqsup> ;

SC <specbase_expression> ::=   <spec_exp_simple>
                             | <spec_exp_filter>
                             | <spec_exp_group>
                             | <spec_exp_merge>
                             | <spec_exp_in>
                             | <spec_exp_insert>
                             | <spec_exp_access>
                             | <spec_exp_project>
                             | <spec_exp_tupleform>
                             | <spec_exp_tuplesel>
                             | <spec_exp_sempty>
                             | <spec_exp_append>
                             | <spec_exp_lgth>
                             | <spec_exp_ith>
                             | <spec_exp_conc>
                             | <spec_exp_sort>
                             | <spec_exp_aplat>
                             | <spec_exp_delete>
                             | <spec_exp_member>
                             | <spec_exp_type_of> ;
```

```
<spec_exp_simple>    ::= "SIMPLE("^     <list_specbase_args> ^")" ;
<spec_exp_filter>    ::= "FILTER("^     <list_specbase_args> ^")" ;
<spec_exp_group>     ::= "GROUP("^      <list_specbase_args> ^")" ;
<spec_exp_merge>     ::= "MERGE("^      <list_specbase_args> ^")" ;
<spec_exp_in>        ::= "IN("^         <list_specbase_args> ^")" ;
<spec_exp_insert>    ::= "INSERT("^     <list_specbase_args> ^")" ;
<spec_exp_access>    ::= "ACCESS("^     <list_specbase_args> ^")" ;
<spec_exp_project>   ::= "PROJECT("^    <list_specbase_args> ^")" ;
<spec_exp_tupleform> ::= "TUPLEFORM("^ <list_specbase_args> ^")" ;
<spec_exp_tuplesel>  ::= "TUPLESEL("^   <list_specbase_args> ^")" ;
<spec_exp_sempty>    ::= "SEMPTY("^     <list_specbase_args> ^")" ;
<spec_exp_append>    ::= "APPEND("^     <list_specbase_args> ^")" ;
<spec_exp_lgth>      ::= "LGTH("^       <list_specbase_args> ^")" ;
<spec_exp_ith>       ::= "ITH("^        <list_specbase_args> ^")" ;
<spec_exp_conc>      ::= "CONC("^       <list_specbase_args> ^")" ;
<spec_exp_sort>      ::= "SORT("^       <list_specbase_args> ^")" ;
<spec_exp_aplat>     ::= "APLAT("^      <list_specbase_args> ^")" ;
<spec_exp_delete>    ::= "DELETE("^     <list_specbase_args> ^")" ;
<spec_exp_type_of>   ::= "TYPE-OF("^    <list_specbase_args> ^")" ;
<spec_exp_member>    ::= "MEMBER("^     <list_specbase_args> ^")" ;

L <list_specbase_args>  ::= <specbase_arg>* ^"," % ;

SC <specbase_arg>       ::=   <obj_name>
                            | <constant>
                            | <type_name>
                            | <predicate> ;

SC <equation>           ::=   <simple_equation>
                            | <conditional_equation> ;

<simple_equation>       ::= <specbase_expression> "="
                            <specbase_expression> ;

SC <conditional_equation> ::=   <cond_equ_with_otherwise>
                              | <cond_equ_without_otherwise> ;

<cond_equ_with_otherwise> ::=   <l_if_pred_equation>
                              $ <otherwise_pred_equation> ;

<cond_equ_without_otherwise> ::=   <l_if_pred_equation> ;

L <l_if_pred_equation>  ::= <if_pred_equation>+ ^"," % ;

<if_pred_equation>      ::= <simple_equation>
                              # "if" <predicate> ;

<otherwise_pred_equation> ::= <simple_equation> "otherwise" ;

<outline_op>     ::=   "TYPES :" <list_outli_obj> ";" <empty>
                     $ "OPERATIONS :" <list_outli_op> ";" ;

<outline_obj>    ::=   "OPERATIONS :" <list_outli_op> ";" <empty>
                     $ "TYPES :" <list_outli_obj> ";" ;

L <list_outli_obj>  ::= <outli_obj>* ";" % ;
```

```
L <list_outli_op>    ::= <outli_op>* ";" % ;

<outli_obj>          ::= <list_obj_names> ":" <type> ;
<outli_op>           ::= <list_oper_names> ":" # <op_def> ;

SC <op_def>          ::= <in_specbase> | <op_def_nv> ;

<in_specbase>        ::= "IN SPECBASE" ;

<op_def_nv>          ::= <list_types> $ "->" $ <type> ;

L <list_types>       ::= <type>+ "X" % ;

SC <type>            ::= <type_name> | <basic_type> ;
SC <basic_type>      ::= <int> | <nat> | <char> | <charst>
                       | <bool> ;

<lexicon_op>         ::= "OBJECTIVE :" <text>
                         <objects_lexicon_op>
                         <operations_lexicon_op> ;

SC <objects_lexicon_op> ::= <empty> | <objects_lexicon_op_nv> ;

SC <operations_lexicon_op> ::= <empty>
                             | <operations_lexicon_op_nv> ;

<objects_lexicon_op_nv> ::= $ "OBJECTS :" <list_lex_obj> ";" ;

<operations_lexicon_op_nv>
                 ::= $ "OPERATIONS :" <list_lex_oper> ";" ;

<lexicon_obj>    ::= "OBJECTIVE :" <text>
                     <operations_lexicon_obj>
                     <types_lexicon_obj>
                     <invariants_lexicon_obj> ;

SC <operations_lexicon_obj> ::= <empty>
                              | <operations_lexicon_obj_nv> ;

SC <types_lexicon_obj>  ::= <empty> | <types_lexicon_obj_nv> ;

SC <invariants_lexicon_obj> ::= <empty>
                              | <invariants_lexicon_obj_nv> ;

<operations_lexicon_obj_nv>
                     ::= $ "OPERATIONS :" <list_lex_oper> ";" ;

<types_lexicon_obj_nv>  ::= $ "TYPES :" <list_lex_types> ";" ;

<invariants_lexicon_obj_nv> ::= $ "INVARIANTS :" <text> ;

L <list_lex_obj>         ::= <lex_obj>* ";" % ;
L <list_lex_oper>        ::= <lex_oper>* ";" % ;
L <list_lex_types>       ::= <lex_type>* ";" % ;

<lex_obj>                ::= <obj_name> ":" <text> ;
```

```
<lex_oper>                ::= <oper_name> ":" <text> ;
<lex_type>                ::= <type_name> ":" <text> ;

L <list_oper_names>       ::= <oper_name>+ ^"," % ;
L <list_obj_names>        ::= <obj_name>+ ^"," % ;
L <list_arg_names>        ::= <obj_name>* ^"," % ;

<type_name>               ::= <ident_m> ;
<oper_name>               ::= <ident_m> ;
<obj_name>                ::= <ident> ;

<empty>                   ::= ;

<true>   ::= "true" ;
<false>  ::= "false" ;

<int>    ::= "INT" ;
<nat>    ::= "NAT" ;
<char>   ::= "CHAR" ;
<charst>::= "CHARST" ;
<bool>   ::= "BOOL" ;

<add>    ::= "+" ;
<subst>  ::= "-" ;
<mult>   ::= "*" ;
<div>    ::= "/" ;
<mod>    ::= "MOD" ;
<intdiv>::= "DIV" ;

<eq>     ::= "==" ;
<diff>   ::= "<>" ;
<inf>    ::= "<" ;
<sup>    ::= ">" ;
<eqinf>  ::= "<=" ;
<eqsup>  ::= ">=" ;


GEN <ident>    ::= "[a-z][a-z0-9_]*" ;

GEN <ident_m>  ::= "[A-Z][A-Z0-9_]*" ;

GEN <number>   ::= "[-+]?[0-9]+(""."[0-9]+(""E""[-+]?[0-9]+)?)?" ;

GEN <text>     ::= "´" "COMMENT-LIKE" "´" .
```

A N N E X E 4 :

DEFINITION LDF DE LA SYNTAXE DU

LANGAGE PROLOG

La définition ci-dessous est différente de certaines syntaxes présentées dans la littérature :

- La distinction y est faite entre un terme et un prédicat,
- "not", "," et ";" sont considérés comme des symboles distingués plutôt que des opérateurs,
- Nous ne considérons pas la priorité des opérateurs,
- ...

Cette syntaxe semble mieux répondre à nos besoins : une disjonction dans la syntaxe prolog correspond à une or-explicitation dans la syntaxe RSL, une conjonction à une and-explicitation, une parenth-body à une op-explicitation-parenth, ...

```
(* ldf *)

definition de prolog  ;

<point_d_entree>  ::=    <program>        ;

L <program>       ::=    <clause_com>+    ;

SC <clause_com>   ::=    <clause>
                       | <comment_like>    ;

SC <clause>       ::=    <fact>   |   <rule>
                       |  <question>    ;


<fact>            ::=    $<user_predicate>   "."$  ;

<rule>            ::=    $<head>  ":-"$   <body>   "."$ ;

 <head>           ::=    <user_predicate>    ;

<question>        ::=    $"?-"    <body>    "."$    ;


SC <body>   ::=     <predicate>      |    <conjunction>
                  | <disjunction>    |    <not_body>
                  | <parenth_body>   ;

<conjunction>        ::=    <body>    ","$    <body>  ;

<disjunction>        ::=    <body>    ";"$    <body>  ;

<not_body>           ::=    "not"    <parenth_body>   ;

<parenth_body>       ::=    "("      <body>      ")"  ;

SC <predicate>       ::=    <built_in_predicate>
                          | <user_predicate>  ;

SC <built_in_predicate> ::=   <prefix_bip>
                            |  <infix_bip>      ;

<prefix_bip>   ::=   <predef_pref_pred_name>  <terms>  ;
```

```
<infix_bip>     ::=   <term>   <rel_operator>   <term>  ;

<user_predicate>  ::=   <pred_name>^  <terms>     ;

SC <terms>          ::=    <empty>
                         | <ne_terms>  ;

 <ne_terms>         ::=  ^"("  <list_terms>  ")" ;

  L <list_terms>  ::=    <term>*    ","    ;

SC <term>       ::=    <constant>     |  <variable>
                     |  <prefix_term>  |  <infix_term>
                     |  <list>         |  <parenth_term>  ;

<prefix_term>     ::=    <functor>^  <terms>  ;

<infix_term>      ::=    <term>   <functor>   <term>   ;

<parenth_term>    ::=   "("  <term>  ")"  ;

SC <functor>      ::=  <predef_functor>  |  <user_functor> ;

SC <list>         ::=    <pointed_list>
                       |  <list_of_elts>
                       |  <list_of_head_tail>   ;

 <pointed_list>        ::=   ".("  <term>  ","  <term>   ")" ;

 <list_of_elts>        ::=   "["   <list_terms>   "]"   ;

 <list_of_head_tail>  ::=   "[" <term>  "|"  <term>  "]" ;


SC <predef_pref_pred_name> ::=    <fail>     |   <true>
                                |  <write>    |   <nl>
                                |  <writeq>   |   <read>
                                |  <get>      |   <put>
                                |  <consult>  |   <cut>
                                |  <see>     |   <seen>
                                |  <tell>     |   <told>
                                |  <rel_operator> ;

SC <rel_operator> ::=    <inf>    |   <sup>    |  <eqinf>
                       |  <eqsup>  |   <eq>     |  <diff>
                       |  <is>       ;
```

```
SC <predef_functor>     ::=    <predef_pref_pred_name>
                             | <arithm_operator>  ;

SC <arithm_operator>    ::=    <add>      |    <mult>
                             |  <exp>     |    <mod>
                             |  <subst>   |    <div>
                             |  <intdiv>  |    <sin>    ;


<user_functor> ::=   <ident>  ;

<pred_name>    ::=   <ident>  ;

<empty>        ::=    ;

SC <constant>  ::=    <atom>
                    | <number>    ;

SC <atom>      ::=    <ident>          |   <atom_symbols>
                    | <quoted_atom>   |   <string>   ;


<cut>      ::=    "!"     ;

<fail>     ::=  "fail"     ;
<true>     ::=  "true"     ;
<write>    ::=  "write"    ;
<writeq>   ::=  "writeq"   ;
<read>     ::=  "read"     ;
<nl>       ::=  "nl"       ;
<get>      ::=  "get"      ;
<put>      ::=  "put"      ;
<system>   ::=  "system"   ;
<consult>  ::=  "consult"  ;
<see>      ::=  "see"      ;
<seen>     ::=  "seen"     ;
<tell>     ::=  "tell"     ;
<told>     ::=  "told"     ;

<add>      ::=    "+"     ;
<mult>     ::=    "*"     ;
<exp>      ::=    "^"     ;
<subst>    ::=    "-"     ;
<div>      ::=    "/"     ;
<mod>      ::=    "mod"   ;
<intdiv>   ::=    "//"    ;
<sin>      ::=    "sin"   ;

<inf>      ::=    "<"     ;
<sup>      ::=    ">"     ;
<eqinf>    ::=    "=<"    ;
<eqsup>    ::=    ">="    ;
<eq>       ::=    "="     ;
<diff>     ::=    "=="   ;
```

```
<is>          ::=    "is"     ;

GEN <ident>          ::=    "[a-z][_a-z0-9]*"                ;
GEN <quoted_atom>    ::=    "'"      "COMMENT-LIKE"     "'"  ;
GEN <string>         ::=    "
GEN <comment_like>   ::=    "/*"     "COMMENT-LIKE"     "*/" ;
GEN <number>         ::=    "[-]?[0-9]+[.]?[0-9]*[e]?[0-9]*" ;
GEN <atom_symbols>   ::=    "[+-<>=@#?]+"      ;
GEN <variable>       ::=    "[_A-Z][_A-Za-z0-9]*"    .
```

A N N E X E 5 :

CODE C : PARTIE PRINCIPALE DU PROGRAMME

DE TRADUCTION D'UN TEXTE RSL EN UN PROGRAMME PROLOG

Programme automatisant - partiellement - le processus de transformation d'un texte RSL en un programme Prolog (dans son état actuel).

Vous trouverez ci-après :

```
                                /* codes des noeuds prolog     */
 ...
# define P_RULE          5      /*  rule                       */
# define P_CONJ          6      /*  conjunction                */
# define P_DISJ          7      /*  disjunction                */
 ...
# define P_FACT         16      /*  fact                       */
# define P_HEAD         17      /*  head                       */
# define P_QUEST        18      /*  question                   */
# define P_BODY         19      /*  body                       */
# define P_N_BODY       20      /*  not_body                   */
# define P_P_BODY       21      /*  parenth_body               */
# define P_PRED         22      /*  predicate                  */
# define P_TERM         26      /*  term                       */
 ...
# define P_VAR          75      /*  variable                   */

                                /* codes des noeuds RSL        */
 ...
# define R_OPERB        22      /*  oper_block                 */
# define R_OP_EXP       26      /*  op_expression              */
# define R_FORALL       27      /*  forall_explicitation       */
 ...
# define R_AR_EXP       30      /*  arithm_expr                */
 ...
# define R_EXPL_IO      37      /*  expl_io_assertion          */
# define R_TERMIN       38      /*  terminal_explicitation     */
# define R_OR           39      /*  or_explicitation           */
# define R_AND          40      /*  and_explicitation          */
# define R_WITH         41      /*  with_explicitation         */
 ...
# define R_O_PRED       49      /*  or_predicate               */
# define R_A_PRED       50      /*  and_predicate              */
# define R_W_PRED       51      /*  with_predicate             */
 ...
# define R_OPNT         56      /*  oper_name_terms            */
 ...
# define R_PRED         86      /*  predicate                  */
# define R_N_PRED       87      /*  not_predicate              */
# define R_P_PRED       88      /*  parenth_predicate          */
# define R_ATOM         89      /*  atom                       */
 ...
# define R_SP_EX        95      /*  specbase_expression        */
# define R_SIMPLE       96      /*  spec_exp_simple            */
# define R_FILTER       97      /*  spec_exp_filter            */
 ...
# define R_MEMBER      115      /*  spec_exp_member            */
 ...
# define R_OB_N        136      /*  obj_name                   */
 ...
# define R_ADD         146      /*  add                        */
# define R_IDIV        151      /*  integer division           */
 ...
# define R_NUMB        159      /*  number                     */
# define R_TEXT        160      /*  text                       */
```

```
/*****************************************************************/
/*****************************************************************/
/*****************************************************************/
/********                                                 *******/
/******                                                    *****/
/*****           PROGRAMME  DE  TRANSFORMATION              ****/
/*****     D'UN  TEXTE  RSL  EN   UN PROGRAMME  PROLOG      ****/
/******                                                    *****/
/********                                                 *******/
/*****************************************************************/
/*****************************************************************/
/*****************************************************************/


# include "ctype.h"
# include "Anoeud.inc"
# include "FDsimu.str"
# include "stdio.h"
# include "data.c"    /* fichier de donnees                      */
# include "crea.c"    /* procedures de creation de sous-arbres   */
                      /* prolog                                  */
# include "bn.c"      /* procedures de traduction de sous-arbres*/
                      /* RSL de bas niveau                       */
# include "pred.c"    /* procedures de traduction de sous-arbres*/
                      /* predicats RSL                           */

# define SUPVAR  7    /* longueur maximum des variables generees*/



int n ;               /* variable globale utilisee pour generer */
                      /* des variables PROLOG                    */

int FDGdep ;
static struct noeud *sommet,*ref;
char *Adatdir;
FILE *Fmsgerr ;
FILE *fichin ;
FILE *fopen( );

struct noeud *tab_refs[SUP] ;
```

```
main (argc,argv)
 int argc;
 char *argv[];


/*****************************************************/
/*****************************************************/
/***                                               ***/
/**        DEBUT  DU  PROGRAMME  PRINCIPAL         **/
/***                                               ***/
/*****************************************************/
/*****************************************************/

{ int i, fin ;

  struct noeud *ref_pde, *ref_oper_block, *ref_rule ;
  struct noeud  *ref_fact, *somres, *ref_prog ;

  struct noeud *ansynt ( );
  struct FDuples *start,*FDgtst( );

  n = 0 ;

  fichin = fopen( argv[1], "r" ) ;

  if ( fichin == NULL )
    {   fprintf( stderr, "Usage : proto filename ..0 ) ;
        exit( ) ;     }

  init ( ) ;

            /** * * * * * * * * * * * * * * * * * * * * * * * **/
            /*  ANALYSE  SYNTAXIQUE  DU  FICHIER  ARGUMENT    */
            /** * * * * * * * * * * * * * * * * * * * * * * * **/

  sommet = ansynt ( fichin ) ;

  if ( sommet == NULL )
    printf( "No reference to a top  ....0 ) ;

  else

   {      /** * * * * * * * * * * * * * * * * * * * * * * * * **/
          /*   TRADUCTION  DE L'ARBRE RSL EN UN ARBRE PROLOG    */
          /** * * * * * * * * * * * * * * * * * * * * * * * * **/

     for ( i = 0  ;  i < SUP  ;  ++i )
        tab_refs[i] =  0 ;

     ref_pde = sommet->fils->frere ;
```

```
                /*   BOUCLE DE TRAITEMENT DES OPER-BLOCKS      */
                /* avec creation d'un tableau de references    */
                /* aux faits et regles constituant le program- */
                /* me PROLOG                                   */

    ref_oper_block = ref_pde->fils->fils ;
    i = 0 ;
    fin = 1 ;

    while (fin != 0)

      {   t_operb ( ref_oper_block, &ref_rule, &ref_fact ) ;

          if (ref_fact != 0)
            { tab_refs[i] = ref_fact ;
              ++i ; }

          if (ref_rule != 0)
            { tab_refs[i] = ref_rule ; }

          if (ref_oper_block->frere == 0)
            fin = 0 ;

            else { ++i ;
                   ref_oper_block = ref_oper_block->frere ; }
      } ;

    ref_prog = crea_prog ( tab_refs );

    somres = crea_arbre ( ref_prog );


                /** * * * * * * * * * * * * * * * * * * * **/
                /*    DECOMPILATION DE L'ARBRE PROLOG     */
                /** * * * * * * * * * * * * * * * * * * * **/

    FDGdep = 10 ;
    FDarbvisu ( somres ) ;
    FDufil ( somres, 80, stdout ) ;

   }

  fclose ( fichin ) ;
}

/*****************************************************/
/*****************************************************/
/***                                               ***/
/**        FIN   DU   PROGRAMME  PRINCIPAL         **/
/***                                               ***/
/*****************************************************/
/*****************************************************/
```

```
/* fonction t_operb ( traitement d'un sous-arbre oper_block RSL )
*
*  - input   : ref_op_bl : reference de l'oper_block RSL a
*                          traiter
*
*  - outputs :
*       - ref_rule : reference d'un sous-arbre prolog "rule"
*       - ref_fact : reference d'un sous-arbre prolog "fact"
*                    (nulle si aucun fait n'est a generer)
*
*  - principe : si l'oper_block est relatif a une "formal_spec_
*               op", on la traite ; sinon, on renvoie 0 dans
*               ref_rule et ref_fact  (cas non encore traite)
*/


t_operb ( ref_op_bl, ref_rule, ref_fact )

 struct noeud *ref_op_bl, *(*ref_rule), *(*ref_fact) ;

 { struct noeud *ref_formal ;

   ref_formal = ref_op_bl->fils->frere->fils ;
   if ( ref_formal->code = R_PTOP )
     t_probl ( ref_formal, ref_rule, ref_fact ) ;

     else { *ref_rule = 0 ;
            *ref_fact = 0 ; }
 } ;
```

```
/* fonction t_probl ( traitement d'un sous-arbre problem_tree_op
*                    RSL )
*
* - input   : ref_probl_tree_op : reference du noeud "problem_
*                                 tree_op" RSL a traiter
*
* - outputs :
*      - ref_rule : reference d'un sous-arbre "rule" prolog
*      - ref_fact : reference d'un sous-arbre "fact" prolog
*                   (eventuellement nulle)
*
* - principe : traiter la premiere partie du problem_tree_op
*              (op_expression), traiter ensuite la seconde partie
*              (op_explicitation), creer une regle prolog dont la
*              tete (head) est le resultat du traitement de l'op_
*              expression et dont le corps (body) est le resultat
*              du traitement de l'op_explicitation.
*
*/


t_probl ( ref_probl_tree_op, ref_rule, ref_fact )

 struct noeud *ref_probl_tree_op, *(*ref_rule), *ref_fact ;

 { struct noeud *ref_op_expr, *ref_head, *ref_op_expl, *ref_body;
   struct noeud *crea_rule_pr( ), *t_oexpr( );

   ref_op_expr = ref_probl_tree_op->fils->frere->frere->fils ;
   ref_op_expl = ref_op_expr->frere ;

   switch ( ref_op_expl->code )

   { case R_TERMIN : ref_head = t_oexpr ( ref_op_expr ) ;
                     ref_fact = 0 ;
                     t_oexpl (ref_op_expl, ref_head, &ref_body);
                     *ref_rule = crea_rule_pr(ref_head,ref_body);
                     break;

     default   : printf ("op-explicitation non implementee 0);
                 break;
   };

 } ;
```

```
/* fonction t_oexpr ( traitement d'un sous-arbre op_expression
*                     RSL )
*
* - input    : ref_op_expression : reference du sous-arbre "op_
*                                  expression" RSL a traiter
*
* - outputs  :
*     - ref_head : reference d'un sous-arbre "head" PROLOG
*
* - principe : construire un tableau dont le premier element
*              est la partie resultat (obj_name) de l'op_expres-
*              sion et dont les elements suivants sont les argu-
*              ments de la partie list_arg_names de l'op_expres-
*              sion ;
*              transformer tous ces elements en variables prolog;
*              transformer la partie oper_name de l'op_expression
*              en predicate_name prolog; construire le sous-arbre
*              "list_terms" prolog a partir du tableau des varia-
*              bles, construire le sous-arbre "predicate_name"
*              prolog, construire les sous-arbres "user_predicate"
*              et "head"
*
*/


struct noeud *t_oexpr ( ref_op_expression )

 struct noeud *ref_op_expression ;

 { char val_oper_name[SUP], tab_arg_names[SUP] [SUP] ;
   char tab_vars[SUP] [SUP], val_pred_name[SUP] ;
   struct noeud *ref_list, *ref_arg, *tab_rvar[SUP], *ref_head ;
   struct noeud *ref_list_t, *ref_ne_terms, *ref_pred_name ;
   struct noeud *crea_list_terms( ), *crea_ne_terms( ) ;
   struct noeud *crea_pname( ), *crea_userp( ), *crea_head( ) ;
   struct noeud *ref_userp ;

   int fin, i, j ;

   for ( i = 0  ;  i < SUP  ;  ++i )
      { for ( j = 0  ;  j < SUP  ;  ++j )
         {  tab_arg_names[i][j] = ' ' ;
            tab_vars[i][j]      = ' ' ; }
        tab_rvar[i] = 0 ; }

   getval ( ref_op_expression->fils->frere->fils, val_oper_name);
   getval ( ref_op_expression->fils->fils, tab_arg_names[0] ) ;

   ref_list = ref_op_expression->fils->frere->frere ;
   ref_arg = ref_list->fils ;
   fin = 0 ;

   for ( i = 1  ;  (fin == 0)  &&  ( ref_arg != 0 )  ;  ++i )

      { getval ( ref_arg->fils, tab_arg_names[i]) ;
```

```
        if ( ref_arg->frere == 0 ) fin = 1 ;
          else   ref_arg = ref_arg->frere ;
      }


   for ( i = 0  ; tab_arg_names[i][0] != ' '  ; ++i )
      min_to_var ( tab_arg_names[i] , tab_vars[i] ) ;

   maj_to_min ( val_oper_name, val_pred_name ) ;

   for ( i = 0  ;  tab_vars[i][0] != ' '  ; ++i )
      tab_rvar[i] = constrgen ( P_VAR, tab_vars[i],
                                strlen (tab_vars[i]));

   ref_list_t = crea_list_terms ( tab_rvar );
   ref_ne_terms = crea_ne_terms ( ref_list_t );

   ref_pred_name = crea_pname ( val_pred_name );
   ref_userp = crea_userp ( ref_pred_name, ref_ne_terms );

   ref_head = crea_head ( ref_userp );
   return ( ref_head );
} ;
```

```
/* fonction t_oexpl ( traitement d'un sous-arbre op_explici-
*                    tation RSL )
*
* - inputs :
*     - ref_op_explicitation : reference du noeud "op_expli-
*                              citation" RSL a traiter
*     - ref_head : reference a un sous-arbre "head" prolog
*
* - outputs :
*     - ref_body : reference d'un sous-arbre "body" prolog
*
* - principe : traiter le noeud op_explicitation en fonction
*              de son "type" : terminal_explicitation, and_
*              explicitation, ...
*/


t_oexpl ( ref_op_expl, ref_head, ref_body )

 struct noeud *ref_op_expl, *ref_head, *(*ref_body) ;

 { struct noeud *t_term_expl( ) ;

   switch ( ref_op_expl->code )

    { case R_TERMIN :  *ref_body = t_term_expl (ref_op_expl) ;
                       break ;

      default       :  printf ("         non implemente 0) ;
    }
  } ;
```

```
/* fonction t_term_expl ( traitement d'un sous-arbre
*                        terminal_explicitation RSL )
*
* - input    : ref_term_expl : reference du noeud "terminal_
*                              explicitation" a traiter
*
* - output   : reference a un sous-arbre body prolog
*
* - principe : traiter la terminal_explicitation en fonction
*              de son "type" : obj_name, number, ...
*
*/


struct noeud *t_term_expl ( ref_term_expl )

 struct noeud *ref_term_expl ;

 { struct noeud *ref_obj_name, *ref_term,  *ref1, *ref2 ;
   struct noeud *ref_op, *ref_pred ;
   struct noeud *t_obj_name( ), *crea_is( ), *creaistr( ) ;
   struct noeud *t_spec_ex( ), *t_ont( ), *t_ar_exp( ) ;

   ref_obj_name = ref_term_expl->fils ;
   ref_term = ref_obj_name->frere ;

   switch ( ref_term->code  )

    { case R_OPNT  : ref_pred = t_ont ( ref_obj_name,
                                        ref_term );
                     return ( ref_pred );

      case R_SIMPLE :
      case R_FILTER :
      case R_GROUP  :
      case R_MERGE  :
      case R_IN     :
      case R_INSERT :
      case R_ACCESS :
      case R_PROJ   :
      case R_T_FORM :
      case R_T_SEL  :
      case R_SEMPTY :
      case R_APP    :
      case R_LGTH   :
      case R_ITH    :
      case R_CONC   :
      case R_SORT   :
      case R_APLAT  :
      case R_DEL    :
      case R_MEMBER :
      case R_TY_OF  :  ref_pred = t_spec_ex ( ref_obj_name,
                                              ref_term ) ;
                       return ( ref_pred ) ;
```

```
      default       : { switch ( ref_term->code  )

            { case R_OB_N    :  ref2 = t_obj_name ( ref_term );
                                ref_op = crea_eq ( ) ;
                                break ;
              case -R_TEXT   :  ref2 = t_txt ( ref_term );
                                ref_op = crea_eq ( ) ;
                                break;

              default        : { switch ( ref_term->code )

                { case -R_NUMB :  ref2 = t_number (ref_term);
                                  break ;
                  case R_AR_EXP:  ref2 = t_ar_exp (ref_term);
                                  break;
                  case R_PTERM :  ref2 = t_pterm (ref_term);
                                  break;
                  default  :
                    printf (" erreur dans terminal_expl 0) ;
                } ;

                 ref_op = crea_is ( ) ; } ;
            } ;
          ref1 = t_obj_name ( ref_obj_name );
          ref_pred = creaistr ( ref1, ref_op, ref2 );

          return (ref_pred) ;
        } ;
      };
} ;
```

```
/* fonction t_ar_exp ( traitement d'un sous-arbre arithm_
*                      expression RSL )
*
* - input    : ref_a_exp : reference du sous-arbre arithm_
*                          expression RSL a  traiter
*
* - output   : reference d'un sous-arbre infix_term prolog
*
*/


struct noeud *t_ar_exp ( ref_a_exp )

 struct noeud *ref_a_exp;

 { struct noeud *ref_t1, *ref1, *ref_t2, *ref2, *ref_t3 ;
   struct noeud *ref3, *ref_int, *crea_add( ), *crea_subst( ) ;
   struct noeud *crea_mult( ), *crea_mod( );
   struct noeud *crea_div( ), *crea_idiv( ), *crea_iterm( );

   ref_t1 = ref_a_exp->fils;
   switch( ref_t1->code )

    { case R_OB_N    : ref1 = t_obj_name ( ref_t1 );
                       break;

      case -R_NUMB   : ref1 = t_number ( ref_t1 );
                       break;

      case R_AR_EXP  : ref1 = t_ar_exp ( ref_t1 );
                       break;

      case R_PTERM   : switch ( ref_t1->fils->code )

                     { case R_OB_N    :
                       case -R_NUMB   :
                       case R_AR_EXP  : ref1 = t_pterm ( ref_t1 );
                                        break;
                       default :
                       printf(" le terme de arithm_exp errone 0);
                       break ;
                     } ;
                       break ;
      default   : printf (" le terme de arithm_exp errone 0);
     };

   ref_t3 = ref_t1->frere->frere;
   switch( ref_t3->code )

    { case R_OB_N    : ref3 = t_obj_name ( ref_t3 );
                       break;

      case -R_NUMB   : ref3 = t_number ( ref_t3 );
                       break;
```

```
      case R_AR_EXP : ref3 = t_ar_exp ( ref_t3 );
                      break;

      case R_PTERM  : switch ( ref_t3->fils->code )

                    { case R_OB_N   :
                      case -R_NUMB  :
                      case R_AR_EXP : ref3 = t_pterm ( ref_t3 );
                                      break;
                      default :
                      printf(" 2e terme de arithm_exp errone 0);
                      break ;
                    } ;
                      break ;
      default     :
          printf (" 2e terme de arithm_exp errone 0);
    };

    ref_t2 = ref_t1->frere;
    switch ( ref_t2->code )

    { case R_ADD    : ref2 = crea_add ( );
                      break;
      case R_SUBST  : ref2 = crea_subst ( );
                      break;
      case R_MULT   : ref2 = crea_mult ( );
                      break;
      case R_DIV    : ref2 = crea_div ( );
                      break;
      case R_MOD    : ref2 = crea_mod ( );
                      break;
      case R_IDIV   : ref2 = crea_idiv ( );
                      break;

      default  : printf (" operateur non implemente 0);
    };

    ref_int = crea_iterm ( ref1, ref2, ref3 );

    return ( ref_int );
};
```

```
/* fonction t_pterm ( traitement d'un sous-arbre parenth_
*                    term RSL )
*
* - input  : ref_pterm : reference du sous-arbre parenth_
*                        term RSL a traiter
*
* - output : reference d'un sous-arbre parenth_term prolog
*
*/


struct noeud *t_pterm ( ref_pterm )

 struct noeud *ref_pterm;

 {  struct noeud *ref_t_rsl, *ref_t_prl, *ref_int;
    struct noeud *crea_pterm( );

    ref_t_rsl = ref_pterm->fils;
    switch (ref_t_rsl->code)

    { case R_OB_N   :  ref_t_prl = t_obj_name ( ref_t_rsl );
                       break ;

      case -R_NUMB  :  ref_t_prl = t_number ( ref_t_rsl );
                       break ;

      case -R_TEXT  :  ref_t_prl = t_txt ( ref_t_rsl );
                       break;

      case R_AR_EXP :  ref_t_prl = t_ar_exp ( ref_t_rsl );
                       break;

      case R_PTERM  :  ref_t_prl = t_pterm ( ref_t_rsl );
                       break;

      default       :  printf ("erreur dans t_pterm 0) ;
    } ;

    ref_int = crea_pterm ( ref_t_prl );
    return (ref_int);
  };
```

```
/* fonction t_ont ( traitement d'un sous-arbre oper_name_
*                  terms RSL )
*
* - inputs :
*      - ref_obj_name : reference d'un sous-arbre obj_name
*                       RSL (= le resultat de l'operation
*                       a traiter)
*      - ref_term     : reference du sous-arbre oper_name_
*                       terms RSL a traiter
*
* - output : reference d'un sous-arbre body prolog
*
*/


struct noeud *t_ont ( ref_obj_name, ref_term )

 struct noeud *ref_obj_name, *ref_term;

 { char val_opername[SUP], val_pname[SUP], res_int[SUPVAR] ;
   char res_int2[SUPVAR];
   struct noeud *ref_pname, *tab_rtprl[SUP], *ref_arg ;
   struct noeud *ref_ne_terms, *ref_userp, *tab_ri[SUP] ;
   struct noeud *ref_is, *ref_v1, *ref_v2, *ref_list_t ;
   struct noeud  *ref1, *ref2, *ref_int ;

   int i, j, n ;

   for ( i = 0  ;  i < SUP  ; ++i )
      tab_rtprl[i] = 0 ;

   for ( i = 0  ;  i < SUP  ; ++i )
      tab_ri[i] = 0 ;

   getval ( ref_term->fils->fils, val_opername );
   maj_to_min ( val_opername, val_pname );
   ref_pname = crea_pname ( val_pname);

   if ( ref_obj_name != 0 )
     { tab_rtprl[0] = t_obj_name (ref_obj_name) ;
       n = 1;
     }
     else n = 0;

   j = 0 ;
   ref_arg = ref_term->fils->frere->fils ;

   if ( ref_arg == 0 && n == 0)
     ref_ne_terms = crea_empty ( );

     else { for ( i = n  ;  ref_arg != 0  ;  ++i )
          { switch ( ref_arg->code  ) {

          case R_OB_N  :  tab_rtprl[i] = t_obj_name (ref_arg);
                          break ;
```

```
case -R_NUMB   :  tab_rtprl[i] = t_number (ref_arg);
                  break ;

case -R_TEXT   :  tab_rtprl[i] = t_txt (ref_arg);
                  break;

case R_PTERM   :  tab_rtprl[i] = t_pterm (ref_arg);
                  break;

case R_AR_EXP :
      gen_var (res_int) ;
      ref_v1 = constrgen(P_VAR,res_int,
                         strlen(res_int));
      tab_rtprl[i] = ref_v1 ;

      ref_v2 = constrgen(P_VAR,res_int,
                         strlen(res_int));
      ref_is = crea_is( ) ;
      ref_int = t_ar_exp ( ref_arg ) ;
      tab_ri[j] = creaistr (ref_v2, ref_is,
                            ref_int) ;
      ++j ;
      break ;

case R_OPNT :
      gen_var (res_int) ;
      tab_rtprl[i] = constrgen(P_VAR,res_int,
                               strlen(res_int));
      ref_int = crea_obj_name ( res_int) ;

      tab_ri[j] = t_ont ( ref_int, ref_arg) ;
      ++j ;
      break ;

case R_SIMPLE  :
case R_FILTER  :
case R_GROUP   :
case R_MERGE   :
case R_IN      :
case R_INSERT  :
case R_ACCESS  :
case R_PROJ    :
case R_T_FORM  :
case R_T_SEL   :
case R_SEMPTY  :
case R_APP     :
case R_LGTH    :
case R_ITH     :
case R_CONC    :
case R_SORT    :
case R_APLAT   :
case R_DEL     :
```

```
        case R_MEMBER  :
        case R_TY_OF   :
              gen_var (res_int) ;
              tab_rtprl[i] = constrgen(P_VAR,res_int,
                                       strlen(res_int));
              ref_int = crea_obj_name ( res_int) ;
              FDdmptr (ref_int) ;
              tab_ri[j] = t_spec_ex ( ref_int, ref_arg) ;
              ++j ;
              break ;

         default      :
            printf ("argument oper_name_terms errone 0) ;
        } ;

        ref_arg = ref_arg->frere ;
      } ;

    ref_list_t = crea_list_terms ( tab_rtprl ) ;
    ref_ne_terms  = crea_ne_terms ( ref_list_t ) ;

    } ;

  ref_userp = crea_userp ( ref_pname, ref_ne_terms ) ;

  if ( j == 0 )  return ( ref_userp );
   else { ref1 = creactr ( tab_ri, --j ) ;
          ref2 = crea_conj ( ref1, ref_userp ) ;
          return ( ref2 ) ; } ;
} ;
```

```
/* fonction t_spec_ex ( traitement d'un sous-arbre specbase_
*                      expression RSL )
*
* - inputs :
*      - ref_obj_name : reference du sous-arbre resultat
*                       de la specbase_expression
*      - ref_spec : reference du sous-arbre specbase_
*                   expression RSL a traiter
*
* - output : reference d'un sous-arbre user_predicate prolog
*
*/


struct noeud *t_spec_ex ( ref_obj_name, ref_spec )

 struct noeud *ref_obj_name, *ref_spec ;

 { struct noeud *ref_pname, *tab_rtprl[SUP], *ref_arg ;
   struct noeud *ref_ne_terms, *ref_userp, *ref_pred, *ref_int ;
   struct noeud *t_type_n( ), *t_predicat( ), *ref_list_t ;
   int n, i ;

   for ( i = 0  ;  i < SUP  ; ++i )
      tab_rtprl[i] = 0 ;

   ref_arg = ref_spec->fils->fils ;

   for ( i = 0  ;  ref_arg != 0  ;  ++i )
        { switch ( ref_arg->code  ) {

          case R_OB_N    :  tab_rtprl[i] = t_obj_name (ref_arg);
                            break ;

          case -R_NUMB   :  tab_rtprl[i] = t_number ( ref_arg );
                            break ;

          case -R_TEXT   :  tab_rtprl[i] = t_txt ( ref_arg );
                            break;

          case R_TY_N    :  tab_rtprl[i] = t_type_n ( ref_arg );
                            break ;


          case R_TRUE    :
          case R_FALSE   :
          case R_REL_EXP :
          case R_OPNT    :
          case R_SIMPLE  :
          case R_FILTER  :
          case R_GROUP   :
          case R_MERGE   :
          case R_IN      :
          case R_INSERT  :
          case R_ACCESS  :
```

```
        case R_PROJ     :
        case R_T_FORM   :
        case R_T_SEL    :
        case R_SEMPTY   :
        case R_APP      :
        case R_LGTH     :
        case R_ITH      :
        case R_CONC     :
        case R_SORT     :
        case R_APLAT    :
        case R_DEL      :
        case R_MEMBER   :
        case R_TY_OF    :
        case R_N_PRED   :
        case R_P_PRED   :
        case R_EX_PRED : tab_rtprl[i] = t_predicat (ref_arg );
                         break ;

        case R_O_PRED   :
        case R_A_PRED   :
        case R_W_PRED   :
        case R_I_PRED   :
        case R_E_PRED   :
        case R_F_PRED   :
          printf ("argument specbase_expr non traite 0) ;
          break ;

        default       :
          printf ("argument specbase_expr errone 0) ;
       } ;

     ref_arg = ref_arg->frere ;
  } ;

if ( tab_rtprl[0] == 0 )
  ref_ne_terms = crea_empty( ) ;

  else { ref_list_t = crea_list_terms ( tab_rtprl ) ;
         ref_ne_terms  = crea_ne_terms ( ref_list_t ) ;
       } ;

switch ( ref_spec->code )

 { case R_SIMPLE  : ref_pname = crea_pname ("simple");
                    break ;
   case R_FILTER  : ref_pname = crea_pname ("filter");
                    break ;
   case R_GROUP   : ref_pname = crea_pname ("group");
                    break ;
   case R_MERGE   : ref_pname = crea_pname ("merge");
                    break ;
   case R_IN      :
   case R_MEMBER  : ref_pname = crea_pname ("member");
                    break ;
   case R_INSERT  : ref_pname = crea_pname ("insert");
                    break ;
```

```
      case R_ACCESS  : ref_pname = crea_pname ("access");
                       break ;
      case R_PROJ    : ref_pname = crea_pname ("project");
                       break ;
      case R_T_FORM  : ref_pname = crea_pname ("tupleform");
                       break ;
      case R_T_SEL   : ref_pname = crea_pname ("tuplesel");
                       break ;
      case R_SEMPTY  : ref_pname = crea_pname ("sempty");
                       break ;
      case R_APP     : ref_pname = crea_pname ("append");
                       break ;
      case R_LGTH    : ref_pname = crea_pname ("lgth");
                       break ;
      case R_ITH     : ref_pname = crea_pname ("ith");
                       break ;
      case R_CONC    : ref_pname = crea_pname ("conc");
                       break ;
      case R_SORT    : ref_pname = crea_pname ("sort");
                       break ;
      case R_APLAT   : ref_pname = crea_pname ("aplat");
                       break ;
      case R_DEL   : ref_pname = crea_pname ("delete");
                     break ;
      case R_TY_OF : ref_pname = crea_pname ("type_of");
                     break ;
     } ;

  if ( ref_obj_name != 0 )
    { tab_rtprl[0] = t_obj_name (ref_obj_name) ;
      tab_rtprl[1] =  crea_userp ( ref_pname, ref_ne_terms ) ;

      for ( i = n  ;  i < SUP  ; ++i )
         tab_rtprl[i] = 0 ;

      ref_list_t = crea_list_terms ( tab_rtprl ) ;
      ref_ne_terms  = crea_ne_terms ( ref_list_t ) ;

      ref_pred = crea_pname ("eval") ;

      ref_int = crea_userp ( ref_pred, ref_ne_terms ) ;

    }

    else ref_int = crea_userp (ref_pname, ref_ne_terms) ;

  return ( ref_int );
} ;
```

```
/* procedure gen_var ( generation d'une variable PROLOG de
*                     SUPVAR caracteres max )
*
*/


gen_var (var)

char var[SUPVAR] ;

 { char s[3], t[5] ;
   int i ;

   for ( i = 0 ; i < 5 ; ++i )
     t[i] = ' ' ;

   if ( n < 10 )
        t[0] = '0' + n ;

   if ( n > 9  &&  n < 100 )
        { t[0] = '0' + (n / 10) ;
          t[1] = '0' + (n % 10) ; } ;

   if ( n > 99  &&  n < 1000 )
        { t[0] = '0' + (n / 100) ;
          t[1] = '0' + ((n % 100) / 10) ;
          t[2] = '0' + (n % 10) ; } ;

   if ( n > 999  &&  n < 10000 )
        { t[0] = '0' + (n / 1000) ;
          t[1] = '0' + ((n / 100) % 10) ;
          t[2] = '0' + (( n / 10) % 10) ;
          t[3] = '0' + (n % 10) ; } ;

    ++n ;

    s[0] = 'Z' ;
    s[1] = 'z' ;
    s[2] = ' ' ;

    conc ( var, s, t) ;
  } ;
```

```
/* fonction conc (concatenation de 2 strings )
*
*/


conc ( res, t1, t2 )

 char t1[30], t2[30], res[60] ;

 { int i, j ;

   i = 0 ;
   j = 0 ;

   while ( t1[i] != ' ')
        { res[i] = t1[i] ;
          ++i ; } ;

   while ( t2[j] != ' ' )
        { res[i] = t2[j] ;
          ++i ;
          ++j ; } ;
  res[i] = ' ' ;
 } ;
```

```
/* fonction getval ( acces a la valeur d'un generique )
*
*  - input : node : reference du noeud generique
*
*  - output : valgen : valeur du generique
*/

getval ( node, valgen )

  struct noeud *node ;        /*  ref. du noeud generique dont  */
                              /*  on veut la valeur             */
  char valgen[SUP] ;

  { int i, length ;
    char *curchar ;
    struct memter *curstr ;

    for ( i = 0  ;  i < SUP  ; ++i )
       valgen[i] = ' ' ;
    curchar = (char *) node->fils->pere ;
    curstr = (struct memter *) node->fils->frere ;
    for( i = 0 ; i < node->fils->code ; i++)
       {  if (*curchar == ' ')
             { curstr = curstr->suiva ;
               curchar = &(curstr->chcar[0]) ;
             };
          valgen[i] = *curchar ;
          curchar ++ ;
       }
    valgen[i] = ' ' ;
  } ;


/*fonction maj_to_min  ( transformation d'un string en
*                        majuscules (cc_arg) en un string
*                        en minuscules (cc_res) )
*/

maj_to_min ( cc_arg, cc_res )

char  cc_arg[SUP], cc_res[SUP] ;

 { int  i       ;

   for ( i = 0  ;  i < SUP  ;  ++i )
      cc_res[i] = ' ' ;
   for ( i = 0 ;  cc_arg[i] != ' '  ;  ++i )
     { if ( ( isupper ( cc_arg[i] )) != 0 )
         cc_res[i] =  tolower ( cc_arg[i] ) ;
         else if ( cc_arg[i] == '-' )
                cc_res[i] = '_' ;
              else  cc_res[i]  =  cc_arg[i] ;
     }
 } ;
```

```
/*fonction min_to_var  ( transformation d'un string en
*                       minuscules (c_arg) en un string dont
*                       la premiere lettre est majuscule et
*                       dont le reste est en minuscules (c_res))
*/

min_to_var ( c_arg, c_res )

char c_arg[SUP], c_res[SUP] ;

 { int  i       ;

   for ( i = 0  ;  i < SUP  ;  ++i )
      c_res[i] = ' ' ;

   if ( ( islower( c_arg[0] )) != 0 )
     c_res[0] = toupper ( c_arg[0] ) ;
     else c_res[0] = c_arg[0] ;

   for ( i = 1 ;  c_arg[i] != ' '  ;  ++i )

     { if ( isupper ( c_arg[i] ) )

         c_res[i] =   tolower( c_arg[i])  ;
         else if ( c_arg[i] == '-' )
             c_res[i] = '_' ;
             else  c_res[i]  =  c_arg[i] ;
     }
 } ;


/* fonction t_obj_name ( traitement d'un sous-arbre obj_
*                       name RSL )
*
* - input  : ref_obj_name : reference du noeud obj_name
*                           a traiter
*
* - output : reference a un sous-arbre "variable" prolog
*
*/

struct noeud *t_obj_name (ref_obj_name)

 struct noeud *ref_obj_name ;

 { char val_ident[SUP], val_var[SUP] ;
   struct noeud *ref_var ;
   int i;

   getval ( ref_obj_name->fils, val_ident ) ;
   min_to_var ( val_ident, val_var ) ;

   ref_var = constrgen ( P_VAR, val_var, strlen (val_var) ) ;
   return (ref_var );
 } ;
```

```
/* fonction t_number (traitement d'un generique number RSL)
*
* - input  : ref_num : reference du generique RSL a traiter
*
* - output : reference d'un generique number prolog
*
*/

struct noeud *t_number ( ref_num )

 struct noeud *ref_num ;

 { char val_num[SUP] ;
   struct noeud *ref_int;

   getval ( ref_num, val_num ) ;
   ref_int = constrgen ( P_NUMB, val_num, strlen(val_num));
   return (ref_int);
 } ;


/* fonction t_txt ( traitement d'un generique "text" RSL )
*
* - input  : ref_txt : reference du generique "text" RSL a
*                      traiter
*
* - output : reference d'un generique "quoted_atom" prolog
*
*/

struct noeud *t_txt ( ref_txt )

 struct noeud *ref_txt ;

 { char val_txt[SUP] ;
   struct noeud *ref_int;

   getval ( ref_txt, val_txt ) ;
   ref_int = constrgen ( P_Q_ATOM, val_txt, strlen(val_txt));
   return (ref_int);
 } ;
```

```
/* fonction t_type_n ( traitement d'un sous-arbre type_name
*                     RSL )
*
* - input  : ref_type_n : reference du sous-arbre type_name
*                         RSL a traiter
*
* - output : reference d'un generique ident prolog
*
*/

struct noeud *t_type_n ( ref_type_n )

 struct noeud *ref_type_n ;

 { struct noeud *ref_int ;
   char val_type[SUP], val_cst[SUP] ;

   getval ( ref_type_n->fils, val_type ) ;
   maj_to_min ( val_type, val_cst ) ;

   ref_int = constrgen ( P_IDENT, val_cst, strlen(val_cst));

   return ( ref_int ) ;
 } ;
```

```
/* fonction crea_prog ( création d'un sous-arbre "program" PROLOG
*                      à partir d'un tableau contenant des réfé-
*                      rences à des règles et/ou des faits PROLOG
*
* - input    : tab_refs : tableau de références à des sous-arbres
*                         "rule" ou "fac"t PROLOG
*
* - output   : référence à un sous-arbre "program" PROLOG
*
* - principe : créer les liens frères entre les différents noeuds
*              rules et facts, puis créer le noeud liste propre-
*              ment dit.
*/

 struct noeud *crea_prog ( tab_refs )

  struct noeud *tab_refs[SUP] ;

  { struct noeud *ref_prog ;
    int i ;

    for ( i = 1  ;  tab_refs[i] != 0  ;  ++i )
      tab_refs[i] = clien ( tab_refs[i-1], tab_refs[i] ) ;

    ref_prog = hliste ( P_PROG, tab_refs[i-1] ) ;
    return( ref_prog);
  } ;


/* fonction crea_arbre ( création d'un arbre PROLOG à partir
*                       d'une référence à un sous-arbre "program"
*                       PROLOG
*
* - input    : ref_prog : référence à un sous-arbre "program"
*                         PROLOG
* - output   : référence au sommet d'un arbre PROLOG
*
* - principe : construction des sous-arbres "point_d_entree" et
*              "formalisme", et construction du noeud père de
*              ces 2 sous-arbres
*/

 struct noeud *crea_arbre ( ref_prog )

  struct noeud *ref_prog ;

  { struct noeud *ref_pde, *ref_formalisme, *ref_arbre ;

    ref_pde = nun ( P_PDE, ref_prog ) ;
    ref_formalisme = constrgen ( 1, "prolog", 6 ) ;

    ref_arbre =  nbin ( 0, ref_formalisme, ref_pde ) ;
    return( ref_arbre ) ;
  } ;
```

```
/* fonction crea_rule_pr ( création d'un sous-arbre rule PROLOG
*                         à partir d'une référence à un
*                         sous-arbre "head" et d'une référence
*                         à un sous-arbre "body" PROLOG)
*
* - inputs :
*     - ref_head : référence d'un sous-arbre "head" PROLOG
*     - ref_body : référence d'un sous-arbre "body" PROLOG
*
* - output : référence à un sous-arbre "rule" PROLOG
*
*/

struct noeud *crea_rule_pr ( ref_head, ref_body )

 struct noeud *ref_head, *ref_body ;

 { struct noeud *ref_rule ;

   ref_rule =  nbin ( P_RULE, ref_head, ref_body ) ;
   return ( ref_rule);
 } ;



/* fonction crea_list_terms
*      ( création d'un sous-arbre list_terms PROLOG à
*        partir d'un tableau des références des
*        différents termes de la liste )
*
* - input : tab_refs : tableau de références à des termes PROLOG
*
* - output : référence au sous-arbre "list_terms" PROLOG
*
* - principe : création des liens frères entre les éléments
*              de la  liste, puis création du noeud list_terms
*              proprement dit
*/

struct noeud *crea_list_terms ( tab_refs )

 struct noeud *tab_refs[SUP] ;

 { struct noeud *ref_list_t ;
   int i ;

   for ( i = 1  ; tab_refs[i] != 0  ;  ++i )
      tab_refs[i] = clien ( tab_refs[i-1], tab_refs[i] ) ;

   ref_list_t = hliste ( P_LTERM , tab_refs[i-1] ) ;
   return ( ref_list_t ) ;
 } ;
```

```
/* fonction crea_ne_terms ( création d'un sous-arbre ne_terms
*                          PROLOG à partir de la référence
*                          à une liste de termes )
*
* - input  : ref_list_t : référence d'une liste de termes
*                         PROLOG
*
* - output : référence à un sous-arbre "ne_terms" PROLOG
*
*/

struct noeud *crea_ne_terms ( ref_list_t )

 struct noeud *ref_list_t ;

 { struct noeud *ref_ne_t ;

   ref_ne_t =  nun ( P_NE_TER, ref_list_t ) ;
   return ( ref_ne_t ) ;
 } ;


/* fonction crea_pname ( création d'un sous-arbre predicate-
*                        -name PROLOG à partir d'un nom de
*                        predicat )
*
* - input  : val_pred_name : nom de predicat
*
* - output : référence à un sous-arbre "predicate_name"
*            PROLOG
*
*/

struct noeud *crea_pname ( val_pred_name )

 char val_pred_name[SUP] ;

 { struct noeud *ref_ident, *ref_pname;

   ref_ident = constrgen ( P_IDENT, val_pred_name,
                           strlen (val_pred_name)) ;

   ref_pname = nun ( P_PR_N, ref_ident ) ;
   return ( ref_pname );
 } ;
```

```
/* fonction crea_userp ( création d'un sous-arbre "user_predicate"
*                       PROLOG à partir d'une référence à un
*                       noeud "pred_name", et d'une référence
*                       à un noeud "terms" PROLOG
*
* - inputs :
*     - ref_pname : référence à un sous-arbre "predicate_name"
*                   PROLOG
*     - ref_terms : référence à un sous-arbre "terms" PROLOG
*
* - output  : réference à un sous-arbre "user_predicate" PROLOG
*
*/

struct noeud *crea_userp ( ref_pname, ref_terms )

 struct noeud *ref_pname, *ref_terms ;

 { struct noeud *ref_userp ;

   ref_userp = nbin ( P_USERP, ref_pname, ref_terms ) ;
   return ( ref_userp );
 } ;


/* fonction crea_head ( création d'un sous-arbre head PROLOG
*                      à partir d'une référence à un noeud
*                      user_predicate )
*
* - input  : ref_userp : référence à un sous-arbre "user_
*                      predicate" PROLOG
*
* - output : référence à un sous-arbre "head" PROLOG
*
*/

struct noeud *crea_head ( ref_userp )

 struct noeud *ref_userp ;

 { struct noeud *ref_head ;

   ref_head = nun ( P_HEAD, ref_userp ) ;
   return ( ref_head );
 } ;
```

```
/* fonction creaistr ( création d'un sous-arbre "infix_structure"
*                      PROLOG )
*
* - inputs :
*     - ref_term1 : référence du premier terme de la structure
*                   infixée a creer
*     - ref_pname : référence d'un noeud "built_in_predicate_
*                   name"
*     - ref_term2 : référence du second terme de la structure
*                   infixée à créer
*
* - output : référence à un sous-arbre "predicate" PROLOG
*
*/

struct noeud *creaistr ( ref_term1, ref_pname, ref_term2 )

 struct noeud *ref_pname, *ref_term1, *ref_term2;

 { struct noeud *ref_pred;

   ref_pred = nter ( P_I_BIP, ref_term1, ref_pname, ref_term2 ) ;
   return ( ref_pred );
 } ;


/* fonction crea_iterm ( création d'un sous-arbre "infix_term"
*                       PROLOG )
*
* - inputs :
*     - ref_term1 : référence du premier membre du terme
*                   infixé à créer
*     - ref_funct : référence d'un noeud "functor"
*     - ref_term2 : référence du second membre du terme
*                   infixé à créer
*
* - output : reference a un sous-arbre "term infixe" PROLOG
*
*/

 struct noeud *crea_iterm ( ref_term1, ref_funct, ref_term2 )

  struct noeud *ref_term1, *ref_funct, *ref_term2;

  { struct noeud * ref_int;

    ref_int = nter ( P_I_TER, ref_term1, ref_funct, ref_term2 );
    return ( ref_int );
  } ;
```

```
/* fonction crea_is ( création d'un noeud zéroaire "is" PROLOG )
*
* - output : référence à un noeud zéroaire "is" PROLOG
*/

struct noeud *crea_is ()

 { struct noeud *ref_is;

   ref_is = nzer ( P_IS );
   return ( ref_is );
 } ;


/* fonction crea_add ( création d'un noeud zéroaire "add"
*                      PROLOG )
*
* - output : référence à un noeud zéroaire "add" PROLOG
*/

struct noeud *crea_add ()

 { struct noeud *ref_add;

   ref_add = nzer ( P_ADD );
   return ( ref_add );
 } ;


/* fonction crea_mult ( création d'un noeud zéroaire "mult"
*                       PROLOG )
*
* - output : référence à un noeud zéroaire "mult" PROLOG
*/

struct noeud *crea_mult ()

 { struct noeud *ref_mult;

   ref_mult = nzer ( P_MULT );
   return ( ref_mult );
 } ;


/* fonction crea_div ( création d'un noeud zéroaire "div"
*                      PROLOG )
*
* - output : référence à un noeud zéroaire "div" PROLOG
*/

struct noeud *crea_div ()

 { struct noeud *ref_div;

   ref_div = nzer ( P_DIV );
   return ( ref_div ); };
```

```
/* fonction crea_subst ( création d'un noeud zéroaire "subst"
*                        PROLOG )
*
* - output : référence à un noeud zéroaire "subst" PROLOG
*/

struct noeud *crea_subst ( )

 { struct noeud *ref_subst;

   ref_subst = nzer ( P_SUBST );
   return ( ref_subst );
 } ;


/* fonction crea_mod ( création d'un noeud zéroaire "mod"
*                      PROLOG )
*
* - output : référence à un noeud zéroaire "mod" PROLOG
*/

struct noeud *crea_mod ( )

 { struct noeud *ref_mod;

   ref_mod = nzer ( P_MOD );
   return ( ref_mod );
 } ;


/* fonction crea_idiv ( création d'un noeud zéroaire "intdiv"
*                       PROLOG )
*
* - output : référence à un noeud zéroaire "intdiv" PROLOG
*
*/

struct noeud *crea_idiv ( )

 { struct noeud *ref_idiv;

   ref_idiv = nzer ( P_I_DIV );
   return ( ref_idiv );
 } ;
```

```
/* fonction crea_pterm ( création d'un sous-arbre "parenth_term"
*                       PROLOG)
*
* - input  : référence à un sous-arbre "term" RSL
*
* - output : référence à un sous-arbre "term" PROLOG
*
*/

struct noeud *crea_pterm ( ref_term )

 struct noeud *ref_term;

 {  struct noeud *ref_int;

    ref_int = nun ( P_P_TER, ref_term );
    return ( ref_int );
 };


/* fonction crea_empty ( création d'un noeud zéroaire "empty"
*                       PROLOG )
*
* - input  : /
*
* - output : référence à un noeud zéroaire "empty" PROLOG
*
*/

struct noeud *crea_empty ( )

 {  struct noeud *ref_int;

    ref_int = nzer ( P_EMPTY );
    return ( ref_int );
 };


/* fonction crea_true ( création d'un noeud zéroaire "true"
*                      PROLOG )
*
* - output : référence à un noeud zéroaire "true" PROLOG
*
*/

struct noeud *crea_true ( )

 {  struct noeud *ref_int;

    ref_int = nzer ( P_TRUE );
    return ( ref_int );
 };
```

```
/* fonction crea_fail ( création d'un noeud zéroaire "fail"
*                      PROLOG )
*
* - output : référence à un noeud zéroaire "fail" PROLOG
*
*/

struct noeud *crea_fail ( )

 {  struct noeud *ref_int;

    ref_int = nzer ( P_FAIL );
    return ( ref_int );
 };


/* fonction crea_not_body ( création d'un sous-arbre "not_
*                          body" PROLOG)
*
* - input  : référence à un sous-arbre "body" PROLOG
*
* - output : référence à un sous-arbre "not_body" PROLOG
*
*/

struct noeud *crea_not_body ( ref_body )

 struct noeud *ref_body ;

 { struct noeud *ref_int ;

   ref_int = nun ( P_N_BODY, ref_body );
   return ( ref_int ) ;
 } ;


/* fonction crea_par_body ( création d'un sous-arbre "par_
*                          body" PROLOG)
*
* - input : référence à un sous-arbre "body" PROLOG
*
* - output : référence à un sous-arbre "par_body" PROLOG
*
*/

struct noeud *crea_par_body ( ref_body )

 struct noeud *ref_body ;

 { struct noeud *ref_int ;

   ref_int = nun ( P_P_BODY, ref_body );
   return ( ref_int ) ; } ;
```

```
/* fonction crea_eq ( création d'un noeud zéroaire "eq"
*                     PROLOG )
*
* - output : référence à un noeud zéroaire "eq" PROLOG
*/

struct noeud *crea_eq ( )

 {  struct noeud *ref_int;

    ref_int = nzer ( P_EQU );
    return ( ref_int );
 } ;


/* fonction crea_diff ( création d'un noeud zéroaire "diff"
*                       PROLOG )
*
* - output : référence à un noeud zéroaire "diff" PROLOG
*/

struct noeud *crea_diff ( )

 {  struct noeud *ref_int;

    ref_int = nzer ( P_DIFF );
    return ( ref_int );
 } ;


/* fonction crea_sup ( création d'un noeud zéroaire "sup"
*                     PROLOG )
*
* - output : référence à un noeud zéroaire "sup" PROLOG
*/

struct noeud *crea_sup ( )

 {  struct noeud *ref_int;

    ref_int = nzer ( P_SUP );
    return ( ref_int );
 } ;
```

```
/* fonction crea_inf ( création d'un noeud zéroaire "inf"
*                      PROLOG )
*
* - output : référence à un noeud zéroaire "inf" PROLOG
*/

struct noeud *crea_inf ( )

 {  struct noeud *ref_int;

    ref_int = nzer ( P_INF );
    return ( ref_int );
 };


/* fonction crea_eqi ( création d'un noeud zéroaire "eqinf"
*                      PROLOG )
*
* - output : référence à un noeud zéroaire "eqinf" PROLOG
*/

struct noeud *crea_eqi ( )

 {  struct noeud *ref_int;

    ref_int = nzer ( P_EQI );
    return ( ref_int );
 } ;


/* fonction crea_eqs ( création d'un noeud zéroaire "eqsup"
*                      PROLOG )
*
* - output : référence à un noeud zéroaire "eqsup" PROLOG
*/

struct noeud *crea_eqs ( )

 {  struct noeud *ref_int;

    ref_int = nzer ( P_EQS );
    return ( ref_int );
 } ;
```

```
/* fonction crea_disj ( création d'un sous-arbre "disjonction"
*                       PROLOG)
*
* - inputs : ref1 : référence à un sous-arbre "body" PROLOG
*            ref2 : référence à un sous-arbre "body" PROLOG
*
* - output : référence à un sous_arbre "disjonction" PROLOG
*
*/

struct noeud *crea_disj ( ref1, ref2 )

 struct noeud *ref1, *ref2 ;

 { struct noeud *ref_disj ;

   ref_disj = nbin ( P_DISJ, ref1, ref2 ) ;
   return ( ref_disj );
 } ;


/* fonction crea_conj ( création d'un sous-arbre "conjonction"
*                       PROLOG)
*
* - inputs : ref1 : référence à un sous-arbre "body" PROLOG
*            ref2 : référence à un sous-arbre "body" PROLOG
*
* - output : référence à un sous_arbre "conjonction" PROLOG
*
*/

struct noeud *crea_conj ( ref1, ref2 )

 struct noeud *ref1, *ref2 ;

 { struct noeud *ref_conj ;

   ref_conj = nbin ( P_CONJ, ref1, ref2 ) ;
   return ( ref_conj );
 } ;
```

```
/* fonction creactr
*    ( création d'un sous-arbre "conjonction" PROLOG à
*      partir d'un tableau de références à des sous-arbres
*      "body" PROLOG )
*
* - inputs :
*    - tab_refs : tableau de références à des sous-
*                 arbres "body" PROLOG
*    - bsup     : longueur du tableau + 1
*
* - output : référence à un sous-arbre conjonction
*            PROLOG
*/

struct noeud *creactr (tab_refs, bsup)

 struct noeud *tab_refs[SUP] ;
 int bsup ;

 { struct noeud *ref1, *ref2 ;
   int k ;

   if (bsup == 0 ) return (tab_refs[0]) ;
     else {
            ref1 = tab_refs[bsup] ;
            for ( k = bsup-1 ; k >= 0 ; --k )
             { ref2 = crea_conj ( tab_refs[ k], ref1 ) ;
               ref1 = ref2 ; } ;
            return (ref1) ; } ;
 } ;


/* fonction crea_obj_name ( création d'un sous-arbre RSL
*                          obj_name à partir d'une valeur
*                          de obj_name )
*/

struct noeud *crea_obj_name ( val_ident )

 char val_ident[SUP] ;

 { struct noeud *ref_obj_name, *ref_ident ;

   ref_ident = constrgen(R_IDENT,val_ident,strlen(val_ident));

   ref_obj_name = nun ( R_OB_N, ref_ident ) ;
   return ( ref_obj_name );
 } ;
```

```
/* fonction t_predicat (traitement d'un sous-arbre predicat RSL )
*
* - input : ref_pred : reference du sous-arbre predicat RSL a
*                      traiter
*
* - output : reference a un sous-arbre body PROLOG
*
*/


struct noeud *t_predicat ( ref_pred )

 struct noeud *ref_pred;

 { struct noeud *ref_int;
   struct noeud *t_rel_exp( ), *crea_true( ), *crea_fail( );
   struct noeud *t_par_p( ), *t_or_p( ), *t_and_p( ), *t_with_p( );
   struct noeud *t_ont( ), *t_spec_ex( ), *t_not_p( );

   switch ( ref_pred->code )

    {     case R_TRUE     : ref_int = crea_true ( ) ;
                            break ;

          case R_FALSE    : ref_int = crea_fail ( ) ;
                            break ;

          case R_REL_EXP  : ref_int = t_rel_exp ( ref_pred );
                            break ;

          case R_OPNT     : ref_int = t_ont ( 0, ref_pred );
                            break ;

          case R_SIMPLE   :
          case R_FILTER   :
          case R_GROUP    :
          case R_MERGE    :
          case R_IN       :
          case R_INSERT   :
          case R_ACCESS   :
          case R_PROJ     :
          case R_T_FORM   :
          case R_T_SEL    :
          case R_SEMPTY   :
          case R_APP      :
          case R_LGTH     :
          case R_ITH      :
          case R_CONC     :
          case R_SORT     :
          case R_APLAT    :
          case R_DEL      :
          case R_MEMBER   :
          case R_TY_OF    : ref_int = t_spec_ex ( 0, ref_pred );
                            break ;
```

```
        case R_N_PRED  : ref_int = t_not_p ( ref_pred ) ;
                         break ;

        case R_P_PRED  : ref_int = t_par_p ( ref_pred ) ;
                         break ;

        case R_O_PRED  :
        case R_A_PRED  :
        case R_W_PRED  :
        case R_I_PRED  :
        case R_E_PRED  :
        case R_F_PRED  :
        case R_EX_PRED : ref_int = 0;
                         printf (" predicat non implemente 0);
                         break ;
    } ;
   return (ref_int) ;
 };



/* fonction t_not_p ( traitement d'un sous-arbre not_predicate
*                     RSL )
*
* - input  : reference du sous-arbre not_predicate RSL a traiter
*
* - output : reference d'un sous-arbre not_body prolog
*
*/

struct noeud *t_not_p ( ref_not_p )

 struct noeud *ref_not_p ;

 { struct noeud *ref_pred, *ref_int ;

   ref_pred = t_predicat( ref_not_p->fils );
   ref_int = crea_not_body ( ref_pred ) ;
   return ( ref_int ) ;
 } ;
```

```
/* fonction t_par_p ( traitement d'un sous-arbre parenth_predi-
*                    cate RSL )
*
* - input  : reference du sous-arbre parenth_predicate RSL a
*            traiter
*
* - output : reference d'un sous-arbre parenth_body prolog
*
*/

struct noeud *t_par_p ( ref_par_p )

 struct noeud *ref_par_p ;

 { struct noeud *ref_pred, *ref_int ;

   ref_pred = t_predicat ( ref_par_p->fils );
   ref_int = crea_par_body ( ref_pred ) ;
   return ( ref_int ) ;
 } ;


/* fonction t_rel_exp ( traitement d'un sous-arbre relational_
*                       expression RSL )
*
* - input  : ref_rel_exp : reference du sous-arbre relational_
*                          expression RSL a traiter
*
* - output : reference d'un sous-arbre infix_bip prolog
*/

struct noeud *t_rel_exp ( ref_rel_exp )

 struct noeud *ref_rel_exp;

 { struct noeud *ref_t1, *ref1, *ref_t2, *ref2, *ref_t3, *ref3;
   struct noeud *ref_int, *crea_eq( ), *crea_diff( ), *crea_inf( );
   struct noeud *crea_sup( ), *crea_eqi( ), *crea_eqs( );
   struct noeud *creaistr( ), *t_pterm( ), *t_ar_exp( );

   ref_t1 = ref_rel_exp->fils;
   switch( ref_t1->code )

    { case R_OB_N   : ref1 = t_obj_name ( ref_t1 );
                      break;

      case -R_NUMB  : ref1 = t_number ( ref_t1 );
                      break;

      case R_AR_EXP : ref1 = t_ar_exp ( ref_t1 );
                      break;
```

```
    case R_PTERM  : ref1 = t_pterm ( ref_t1 );
                    break;

    default : printf(" 1e terme de relational_expr errone 0);
   };

  ref_t3 = ref_t1->frere->frere;
  switch( ref_t3->code )

   { case R_OB_N   : ref3 = t_obj_name ( ref_t3 );
                    break;

     case -R_NUMB  : ref3 = t_number ( ref_t3 );
                    break;

     case R_AR_EXP : ref3 = t_ar_exp ( ref_t3 );
                    break;

     case R_PTERM  : ref3 = t_pterm ( ref_t3 );
                    break;

     default : printf(" 2e terme de relational_expr errone 0);
   };

   ref_t2 = ref_t1->frere;
   switch ( ref_t2->code )

   { case R_DIFF   : ref2 = crea_diff ( );
                    break;

     case R_INF    : ref2 = crea_inf ( );
                    break;

     case R_EQ     : ref2 = crea_eq ( );
                    break;

     case R_SUP    : ref2 = crea_sup ( );
                    break;

     case R_EQI    : ref2 = crea_eqi ( );
                    break;

     case R_EQS    : ref2 = crea_eqs ( );
                    break;

     default       : printf (" foncteur non implemente 0);
   };

   ref_int = creaistr ( ref1, ref2, ref3 );
   return ( ref_int );
};
```