Université de Nancy I — U.F.R. S.T.M.I.A

Centre de Recherche en Informatique de Nancy

Dérivation de programmes Ada par transformation de systèmes parallèles fondés sur la communication abstraite entre processus

THÈSE

soutenue publiquement le **14 décembre 1990**

pour l'obtention du

Doctorat de l'Université de Nancy I

(– Spécialité Informatique –)

par

Abderrafiaa KOUKAM

devant la commission d'examen

Président :	Jean-Claude DERNIAME
Rapporteurs :	Jean-Marc ADAMO
	Jean-Claude DERNIAME
Examinateurs :	Jean-Pierre FINANCE
	Jacques JULLIAND
	Guy-René PERRIN

Je remercie :

Jacques Julliand, maître de conférences à l'université de Franche-comté qui a suivi de près ce travail. Je le remercie aussi pour ses critiques, sa patience à relire le manuscrit et pour les discussions que nous avons eues.

Jean-Pierre Finance, professeur à l'université de Nancy I et directeur du CRIN, qui m' a accueilli dans son équipe. Ses conseils et ses précieuses remarques m'ont permis d'améliorer la qualité de ce document.

Gy-René Perrin, professeur à l'université de Franche-Comté qui a toujours répondu à mes questions. Je le remercie pour la clarté de ses réponses et les suggestions dont il m'a fait bénéficier.

Jean-Marc Adamo, professeur à l'école normale supérieure de Lyon et Jean-Claude Derniame, professeur à l'institut national polytechnique de Lorraine d'avoir bien voulu être les rapporteurs de cette thèse.

Université de Nancy I U.F.R. S.T.M.I.A

Centre de Recherche en Informatique de Nancy

Dérivation de programmes Ada par transformation de systèmes parallèles fondés sur la communication abstraite entre processus

THÈSE

soutenue publiquement le **14 décembre 1990**

pour l'obtention du

Doctorat de l'Université de Nancy I
(- Spécialité Informatique -)

par

Abderrafiaa KOUKAM

devant la commission d'examen

Président :	Jean-Claude DERNIAME
Rapporteurs :	Jean-Marc ADAMO
	Jean-Claude DERNIAME
Examinateurs :	Jean-Pierre FINANCE
	Jacques JULLIAND
	Guy-René PERRIN

RESUME :

Le travail présenté dans cette thèse s'inscrit dans le cadre d'une démarche méthodique pour la construction de systèmes parallèles. L'idée de base est de définir de tels systèmes comme un ensemble d'entités manipulant des suites de données, entre lesquelles des relations dites de communication sont spécifiées. Pour concevoir les solutions déduites de cette démarche, un langage d'expression de systèmes parallèles est proposé. Ce langage est caractérisé par :

- une expression abstraite des relations de communication en termes de types abstraits algébriques appelés types de communication.
- une conception modulaire fondée sur la séparation de la description des communications de l'écriture des processus.

Notre premier objectif est d'élaborer une démarche pour automatiser la transformation des types de communication en programmes Ada. Mais la distance est grande entre cette expression abstraite et statique de la communication et le programme final dynamique et efficace. Aussi avons-nous été conduit à décomposer le processus de transformation en plusieurs étapes fondées sur des règles de transformation. Deux classes de règles sont ainsi mises en évidence :

- les règles qui permettent de représenter un type de communication par des structures de données proches du langage Ada.
- les règles qui introduisent les outils nécessaires à l'implantation des types de communication dans un environnement parallèle.

Nous proposons ensuite de transformer les processus utilisant les types de communication en tâches Ada. Cette transformation est spécifiée en termes de règles d'inférence.

Notre étude s'est concrétisée aussi par l'implantation d'un système de transformation. Réalisé en Lisp, ce système met en oeuvre les règles de transformation et s'intègre au sein d'un environnement de programmation composé d'un éditeur syntaxique et d'un interprète de systèmes parallèles. Enfin, nous proposons une extension de cet environnement pour prendre en considération la réutilisation des types de communication.

MOTS CLES : Transformation, Parallélisme, Communication, Type Abstrait de Données, Réutilisabilité.

ABSTRACT :

The work presented in this PH.D thesis is concerned by a methodological approach for constructing parallel systems. The basic idea is to define such systems as a set of entities handling data series and to specify relations between them. In order to express the solution obtained by this approach, a language for designing parallel systems has been proposed. This language is caracterised by:

- a formal expression of communication in terms of algebraic abstract data types called communication types.
- a modular design based on separation of the communication and the processes description.

Our first aim is to develop a methodology for automatic transformation of communication types into Ada programs. Since a final efficient and dynamic program is far from the initial abstract and static description of the communication, we have been induced to decompose the transformation process into several steps based on transformation rules. Two classes of rules have been shown up:

- rules which allow representation of a communication type using data structures close to Ada language.
- rules which introduce primitives necessary for implementing the communication types in a parallel environment.

Afterwards, we transform the processes which use communication types into Ada tasks. This transformation is specified in terms of inference rules.

Our study has resulted in the implementation of a transformation system. Achieved in lisp language, this system implements the transformation rules and is integrated with a programming environment composed of syntactic editor and a parallel system interpreter. Finally, we propose an extension of this environment allowing the reuse of previously defined communication types.

KEY WORDS : Transformation, Parallelism, Communication, Abstract Data Type, Reuse.

Table des matières

Introduction

Le parallélisme qui longtemps a constitué le terrain privilégié des systèmes d'exploitation s'étend depuis quelques années à d'autres domaines d'application. Cette extension a été motivée essentiellement par l'apparition de nouvelles classes de problèmes qui sont par essence parallèles, jointe au besoin d'améliorer les solutions proposées pour d'autres problèmes volumineux en informations et en calculs.

Ainsi voit-on se développer des études sur le parallélisme que l'on peut regrouper en deux thèmes. Le premier concerne la conception d'architectures parallèles. Les progrès technologiques ont conduit à l'aboutissement de plusieurs projets de recherche dans ce domaine : les machines SIMD et MIMD, les tableaux de processeurs, les architectures systoliques et récemment les machines à base de transputers pour ne citer qu'eux.

Le deuxième thème concerne la conception et la programmation de systèmes parallèles. La recherche dans ce domaine regroupe l'étude des problèmes liés à l'aspect méthode de construction, à la définition de langages d'expression et de mise en oeuvre de systèmes parallèles. Nous incluons aussi dans ce thème l'étude des modèles mathématiques du parallélisme dans lesquels est définie la sémantique des langages parallèles et les systèmes de preuves associés. Les recherches menées dans le cadre du projet COMETE (COMmunication et TEmps) auquel nous participons s'articulent autour de ce dernier thème.

1 Cadre de l'étude

La construction d'un système parallèle suppose la décomposition d'un problème en un ensemble de processus et la définition de leurs coopérations. Chacun des processus réalise un but partiel et participe, par la communication et la synchronisation avec les autres, à la réalisation de la solution globale du problème posé.

Comme dans le cadre de la programmation séquentielle, il est nécessaire d'adopter une démarche méthodique si l'on veut maîtriser la complexité du processus de construction de tels systèmes, tant au niveau de la conception que de la mise en oeuvre. Le but du projet COMETE est de proposer un mode de raisonnement et des outils pour la conception et la mise en oeuvre de systèmes parallèles. L'idée de base est de concevoir de tels systèmes

comme un ensemble d'entités manipulant des suites de données, entre lesquelles des relations de communication sont spécifiées. La notion de relation de communication a été ainsi introduite comme outil de raisonnement pour la conception d'une solution parallèle [PER 83,FIN 83,JUL 83,PER 85] et a conduit aux résultats suivants:

- définition d'un cadre formel pour la spécification de la communication en termes de relations entre les suites de données échangées par les processus. Ces suites sont interprétées dans un modèle asynchrone, en leur associant une horloge,
- expression constructive de ces relations en termes de types abstraits de données appelés types de communication,
- définition d'un langage d'expression de systèmes parallèles en termes de processus communicants. Ce langage que nous appelons par la suite Lesp intègre la notion de type de communication pour l'expression des relations entre processus.
- définition d'une sémantique fonctionnelle, puis opérationnelle du langage Lesp et qui a donné lieu à la réalisation d'un interprète.

Le langage Lesp a été proposé dans [PER 85] pour permettre la validation de la notion de communication comme outil de conception de systèmes parallèles. L'originalité de ce langage réside essentiellement dans l'expression abstraite de la communication entre processus grâce à la notion de type de communication. Pour communiquer, les processus utilisent deux opérations fournies par le type de communication : *produire* pour émettre une donnée, *consommer* pour recevoir une donnée. Aucun énoncé de synchronisation n'est introduit quant à la définition d'un type de communication. Les autres constructions du langage sont relatives à la définition des processus et trouvent leurs équivalents sous une autre forme dans les langages impératifs tels que Pascal et CSP [HOA 78]. Une description fonctionnelle de systèmes parallèles utilisant les types de communication est également proposée dans [JUL 90]. On trouve aussi dans [JUL 83], à travers des exemples, une analyse des outils de communication proposés dans les langages et l'apport de la notion de type de communication dans la construction de programmes parallèles.

Il reste cependant à obtenir à partir de l'énoncé Lesp, résultat de la conception, un programme exécutable. Ceci nécessite au préalable :

- l'étude des méthodes qui permettent la transformation des types de communication et des processus en un programme exécutable,
- la détermination d'outils aptes à mettre en oeuvre ces méthodes.

Ces méthodes et ces outils qui constituent ce que nous appelons des techniques de transformation de programmes, sont l'objet de notre étude.

2 Objectifs

L'objectif principal de cette thèse peut être résumé par la formule suivante:

Définir des méthodes et des outils permettant de transformer un système parallèle en un programme exécutable.

Pour préciser cette formule, il faut s'intéresser aux mots-clés *transformer*, *système parallèle* et *programme exécutable*:

- *Les systèmes parallèles* qui constituent le point de départ de notre étude sont des énoncés exprimés dans le langage Lesp. Ces énoncés introduisent deux sortes d'entités: les processus qui décrivent des calculs, et les types de communication qui spécifient les échanges de données entre les processus.
- *Les programmes* auquels nous voulons aboutir sont des énoncés exprimés dans le langage Ada, qui propose la notion de processus et le concept de rendez-vous pour exprimer la communication et la synchronisation.
- *Transformer*, c'est définir le passage d'un système Lesp à son implantation en Ada.

Nous abordons séparément le problème de la transformation des types de communication et celui des processus.

2.1 Transformation des types de communication

De la spécification d'un type de communication, il s'agit de dériver un programme Ada. Mais la distance est grande entre cette spécification abstraite et statique, et le programme final dynamique et efficace. Précisons d'avantage.

La spécification d'un type de communication est exprimée en terme de type abstrait de données. Elle introduit des objets et des fonctions qui expriment les propriétés logiques de la communication. Par exemple, on manipule à ce niveau des objets abstraits tels que des suites infinies, des fonctions, et ces objets ne sont pas toujours exprimables directement dans un langage de programmation. Nous qualifions donc cette spécification d'abstraite. Nous utilisons ici le qualificatif statique pour exprimer qu'une telle spécification n'introduit pas les contraintes nécessaires à l'implantation de la communication telles que: une opération n'est pas toujours activable, les opérations de communication doivent s'exécuter en exclusion mutuelle dans certaines situations. Ce sont, entre autres, ces contraintes qui confèrent au programme final le qualificatif dynamique, et que l'on désignera dans la suite par les caractéristiques de l'environnement des types de communication.

Aussi est-il raisonnable de déterminer des étapes intermédiaires pour transformer un type de communication en un programme. D'où l'idée de décomposer l'objectif "transformation" en deux sous-objectifs: représentation et passage à un niveau dynamique.

Représentation

Le but de cette étape est de transformer un type de communication pour aboutir à des structures de données et fonctions plus proches de celles qui sont utilisées dans un langage de programmation. En général, la représentation d'un type abstrait de données ne s'effectue pas en une seule étape. D'où l'idée de la décomposer en plusieurs étapes élémentaires et d'introduire des règles de transformation pour réaliser ces étapes. C'est ainsi que raisonnent et travaillent les partisans de l'approche transformationnelle [BUR 77] [BRO 81] qui a inspirée ce travail. Il est souhaitable aussi de prouver les règles de transformation que nous introduisons pour assurer la correction de la représentation. A ce sujet, nous considérons une règle de transformation comme une fonction de représentation d'un type abstrait par un autre, et utilisons les résultats des travaux sur les preuves de représentation [FIN 79] [GAU 80]. Nous exprimons le résultat de la représentation dans un style applicatif (fonctionnel), et laissons le passage au style impératif (qui est la caractéristique du langage cible Ada) à la charge de l'étape suivante.

Passage à un niveau dynamique

Pour diverses raisons, la représentation construite à l'étape précédente ne constitue pas l'implantation finale:

- la représentation est rédigée dans un style applicatif permettant d'écrire des définitions telles que : f_2=*enfiler*(f_1, e), où f_1 et f_2 sont des files.
  Cette définition peut conduire à la construction effective en mémoire d'une nouvelle file lors de l'appel de la fonction *enfiler*. Or dans la pratique, la file f_2 est la même que f_1, dont on a simplement modifié le contenu. Pour des raisons d'efficacité, le passage à un langage de programmation tel que Ada va nous conduire à introduire les notions de variable et d'affectation qui sont des notions dynamiques [FIN 78].

- nous avons jusqu'à présent considéré un type de communication comme un type abstrait de données classique (au sens de la programmation séquentielle). Or l'utilisation d'un tel type dans un environnement parallèle conduit à tenir compte, pour son implantation, des caractéristiques de cet environnement.

En conclusion, pour aboutir au programme final, nous introduisons des règles de transformation qui dépendent du langage cible et en l'occurrence Ada. Nous les qualifions de

règles de compilation. Fondées sur des schémas de programmes Ada prédéfinis, ces règles visent à traduire la représentation en un module Ada. Elles introduisent des primitives permettant de mettre en oeuvre les règles de synchronisation impliquées par la relation de communication.

Il est intéressant de constater que les caractéristiques du langage cible n'interviennent que lors de cette dernière étape. Par conséquent, l'obtention d'une implantation dans un autre langage de programmation peut se faire en modifiant uniquement les règles de compilation. Ceci permet de bénéficier de l'étape de représentation. D'où l'intérêt de distinguer ces deux étapes (représentation, passage à un niveau dynamique) dans la transformation d'un type de communication.

2.2 Transformation des processus

En plus des types de communication, la notion de processus est également proposée par le langage Lesp. Les processus formant un système parallèle sont séquentiels et en nombre statiquement défini. Leur définition se fait à l'aide de constructions similaires à celles que l'on trouve dans les langages de programmation tels que Pascal, CSP, et Ada. On peut cependant relever une différence quant à la communication entre processus. En Lesp, celle-ci est exprimée par l'utilisation des opérations fournies par les types de communication. Elle est indirecte et symétrique. Alors qu'en Ada, la communication est fondée sur le concept de rendez-vous et est directe et asymétrique.

Nous identifions la transformation des processus Lesp à une simple compilation , avec des règles de traduction parfois complexes du fait de la différence soulevée en matière d'expression de la communication. Ces règles de traduction seront spécifiées sous forme de règles d'inférence à l'aide de la méthode dite sémantique naturelle [KAH 87]. Cette spécification constitue un guide d'implantation du compilateur.

La communication en Ada est fondée sur un outil de synchronisation qui est le rendez-vous. Pour exprimer une communication asynchrone dans le langage, il faut la programmer en intercalant entre les interlocuteurs un processus intermédiaire. L'approche type de communication jointe à la dérivation systématique de programmes Ada correspondant permettent de spécifier de tels programmes et d'automatiser leur mise en oeuvre. Ceci justifie l'intérêt de cette étude du point de vue de la programmation parallèle en Ada.

2.3 Les outils

Développer des outils logiciels apportant une aide dans cette approche est une nécessité non seulement pour faciliter la tâche du programmeur mais aussi pour valider et montrer l'intérêt pratique des concepts proposés. Parmi ces outils on peut citer bien entendu les

éditeurs syntaxiques, les interprètes mais aussi les outils d'aide à la spécification ou à la transformation et la preuve de programmes. Ceci nous conduit à concevoir et mettre en oeuvre un système de transformation qui constitue le deuxième objectif de notre travail. Ce système vient enrichir l'environnement de programmation COMEDIE [JUL 85] composé d'un éditeur syntaxique et d'un interprète conçus pour le langage Lesp. En se restreignant à l'aspect communication entre processus, il nous est apparu intéressant d'étudier la possibilité de réutiliser des types de communication prédéfinis, et prêts à être intégrés dans un programme. Cette étude s'est concrétisée par la proposition d'une structure d'accueil où seront stockées non seulement la spécification et l'implantation des types de communication, mais aussi les relations pouvant exister entre eux. Ces dernières apportent un plus pour l'organisation et la facilité d'exploitation de cette structure.

3 Plan de l'étude

Suivant les objectifs dégagés au paragraphe 2, nous décomposons cette thèse en trois parties. La première est consacrée à la spécification et l'implantation de la communication, et aux langages de programmation parallèles. La seconde étudie le problème de la transformation des systèmes parallèles Lesp en programmes Ada. La troisième partie est consacrée aux outils permettant de mettre en oeuvre cette transformation.

3.1 Première partie

Dans *le chapitre 1*, nous présentons la spécification de la communication dans trois langages: les expressions de chemins avec prédicats, les modules abstraits et les types de communication. Nous les analysons du point de vue de l'expression de la communication. Au terme de cette analyse, nous constatons que les spécifications qu'ils permettent d'exprimer ne se situent pas au même niveau d'abstraction. Les expressions de chemins et les modules abstraits proposent des spécifications orientées "mise en oeuvre", par opposition à la spécification orientée "problème" qu'offrent les types de communication. Nous montrons alors le lien entre ces deux niveaux et ceci dans le cadre d'une démarche allant de la spécification à l'implantation de la communication.

Au *chapitre 2*, nous commençons par donner une vue d'ensemble de l'expression du parallélisme dans les langages de programmation. Nous insistons particulièrement sur les outils de communication qu'ils proposent. Nous décrivons ensuite le langage d'expression de systèmes parallèles Lesp et sa sémantique. Ce langage intègre la notion de types de communication comme outil de coopération entre les processus. Enfin, nous présentons le langage Ada que nous avons choisi pour implanter les systèmes parallèles Lesp.

3.2 Deuxième partie

Le chapitre 3 pose le problème de la transformation des types de communication en programmes Ada. Deux grandes étapes seront ainsi introduites pour effectuer cette transformation. La première étape dite de représentation, a pour objectif de représenter un type de communication par une structure de données et des fonctions facilement exprimables dans un langage de programmation. La seconde étape dite de passage à un niveau dynamique, construit autour de la représentation un module du langage Ada prenant en compte le parallélisme de l'environnement des types de communication.

C'est l'étape de représentation qui fait l'objet d'une description détaillée au *chapitre 4*. Les règles de transformation qui en constituent le fondement ainsi que leurs preuves y sont présentées. Cette étape identifie les types de communication à des types abstraits de données évoluant dans un environnement séquentiel.

Le chapitre 5 concerne, quant à lui, l'étude de l'étape de passage au niveau dynamique. On y décrit comment aboutir à un programme Ada à partir de la représentation d'un type de communication. En particulier, la synchronisation nécessaire à la mise en œuvre de la communication est introduite par l'utilisation d'outils fournis par le noyau parallèle du langage Ada.

Le chapitre 6 présente la spécification de la compilation du langage Lesp vers le langage Ada par l'intermédiaire d'un système de règles d'inférence. L'objectif de cette spécification est de mettre en évidence les règles de correspondance entre les constructions du langage Lesp et celles du langage Ada. La spécification constitue une sorte de guide d'implantation du compilateur.

3.3 Troisième partie

Pour étendre les fonctionnalités offertes par l'environnement de programmation auquel nous contribuons, nous proposons, parallèlement au mode construction des types de communication, un autre mode fondé sur la réutilisation. Pour cela, *le chapitre 7* présente une structure d'accueil où seront stockées la spécification, l'implantation Ada des types de communication, et les relations pouvant exister entre eux. De telles relations permettent d'établir une hiérarchie entre les types et facilitent par là même leur réutilisation.

Enfin *le chapitre 8* est consacré à l'exposé de nos résultats pratiques en matière d'outils logiciels. Nous y présentons le système de transformation que nous avons implanté en CEYX [HUL 83]. Ce système s'intègre dans un environnement de programmation com-

posé d'un éditeur syntaxique et d'un interprète de programmes Lesp. Nous montrons aussi comment implanter en CEYX les règles de transformation et de compilation. Une présentation de deux systèmes de transformation de programme clôture ce chapitre.

Partie I

Langages de spécification de la coopération et langages d'expression de systèmes parallèles

Au chapitre 1, nous présentons et analysons du point de vue de la communication: les expressions de chemins avec prédicats, les modules abstraits et les types de communication. Nous dégageons de cette analyse les étapes qui conduisent à l'implantation en Ada des types de communication en passant par les modules abstraits. Ces derniers permettent de préciser la sémantique des opérations d'un type de communication.

Le chapitre 2 commence par donner une vue d'ensemble sur les langages de programmation parallèles. L'expression de la communication dans ces langages sera également analysée ainsi que nous l'avons fait au chapitre 1. La présentation du langage Lesp, de sa sémantique et du langage Ada occupera la suite de ce chapitre.

Chapitre 1

Analyse de quelques langages de spécification de la coopération

1 Introduction

D'une manière très générale, La coopération entre processus d'un système parallèle peut être décrite, soit par l'accès à des objets partagés, soit par des échanges de messages ou communication.
Les langages de spécification basés sur le premier mode de coopération permettent d'exprimer les règles de synchronisation auxquelles est soumis l'accès à un objet dans un environnement parallèle. Pour exprimer la communication, ces langages obligent l'utilisateur à énoncer sa spécification en terme de contraintes de synchronisation d'accès à l'objet partagé; support de la communication. La spécification obtenue est celle de la mise en oeuvre [RAY 82]. En effet ce qui rend ces langages impropres à la spécification de la communication, c'est à la fois un trop grand niveau de détail et leur caractère essentiellement dynamique.

Nous présentons et analysons dans la suite deux exemples de tels langages: les expressions de chemins [AND 79] et les modules abstraits [KRO 87]. Ensuite nous présentons un langage où la communication est spécifiée explicitement en deux étapes [PER 85]. La première définit la communication en terme de relation entre les suites de valeurs échangées par les processus. La seconde étape représente cette relation par un type abstrait de données appelé type de communication. L'une des caractéristiques essentielles de ce langage est que, grâce à la démarche progressive qu'il propose pour spécifier la communication, les questions qui surgissent au cours des étapes sont les questions véritablement importantes pour une compréhension complète du problème. Ceci évite la confusion déjà signalée entre la communication et la synchronisation nécessaire à sa mise en oeuvre.

Nous analysons ces trois langages du point de vue de l'expression de la communication. Nous constatons au terme de cette analyse que les spécifications qu'ils permettent

d'exprimer ne se situent pas au même niveau d'abstraction. Les expressions de chemins et les modules abstraits proposent des spécifications orientées mise en oeuvre, par opposition à la spécification orientée problème qu'offrent les relations et les types de communication. Nous montrons alors le lien entre ces deux niveaux et ceci dans le cadre d'une démarche prenant en compte le processus complet de spécification et de mise en oeuvre de la communication. Cela nous permettra de jeter les bases de l'implantation des types de communication qui sera étudiée dans la deuxième partie.

2 Expressions de chemins

Proposées initialement par Campbell et Haberman dans [CAM 74], les expressions de chemins permettent de spécifier le séquencement des opérations d'accès à une ressource partagée. Cette spécification repose sur la séparation de la partie synchronisation de la partie active du système. Depuis, plusieurs extensions ont vu le jour dans le but d'augmenter la capacité d'expression du langage initial. Nous présentons la partie commune à toutes ces extensions, puis nous nous intéressons aux expressions de chemins avec prédicats, extension proposée dans [AND 79] ou sous une forme syntaxique différente dans [JAN 87].

2.1 Expressions de chemins classiques

Le langage des expressions de chemins est formé à l'aide de noms de procédures d'accès de l'objet à contrôler, et des opérateurs suivants:

- la séquence notée p ; q, spécifie que l'exécution de la procédure q ne peut commencer que si p a été entièrement exécutée.
- la sélection notée p + q, spécifie que les procédures p et q s'exécutent en exclusion mutuelle, et ce dans n'importe quel ordre.
- la simultanéité notée {p}, indique que plusieurs exécutions parallèles de la procédure p sont possibles.
- la répétition notée **chemin** EC **fin**, spécifie que zéro ou plusieurs exécutions de l'expression de chemin EC sont possibles. L'expression EC est construite à partir des opérateurs définis précédemment.

Comme il a été montré dans [RAY 82], ces opérateurs sont insuffisants pour spécifier certains schémas de synchronisation tels que l'asservissement et la priorité. Ceci a fait naître un grand nombre de propositions parmi lesquelles:

- Les expressions de chemins numériques [FLO 76], essentiellement pour faciliter l'expression de l'asservissement. Le langage COSY [LAU 79] inclut des constructions allant dans ce sens.
- Les expressions de chemins avec prédicats [AND 79], qui élargissent le champ d'application des expressions de chemins classiques en les unifiant avec le concept de module de contrôle [ROB 77].

Nous présentons la deuxième proposition, le lecteur intéressé peut se rapporter à [RAY 82] où on trouve en particulier à travers des exemples très connus une analyse critique et progressive.

2.2 Expressions de chemins avec prédicats

Le langage initial est étendu afin de rendre plus fin le contrôle des opérations composant une expression de chemins. Cette extension repose essentiellement sur l'association de prédicats à chaque opération à contrôler.

En plus des règles de synchronisation qu'expriment les opérateurs de contrôle déjà vus, un prédicat associé à une opération p spécifie sa condition d'exécution. Une opération p est alors décomposée en trois phases correspondant aux événements significatifs du point de vue du contrôle des accès aux objets partagés: la requête, l'autorisation, et la terminaison. Ces événements sont ainsi ordonnés dans le temps:

requête(p) $\longrightarrow$ autorisation(p) $\longrightarrow$ terminaison(p)

Le langage utilisé pour exprimer les prédicats est formé de constantes, et des compteurs req(p), aut(p), term(p). Ces compteurs mémorisent le nombre des événements cités ci-dessus, et représentent l'histoire des accès aux objets.
La sémantique d'une expression de chemins avec prédicats est définie en terme de restriction sur l'ordre partiel entre les événements associés aux opérations à contrôler.

2.3 Exemple

Considérons un couple de processus où l'un, le producteur envoie des valeurs à l'autre, le consommateur. Les valeurs produites non encore consommées sont stockées dans des tampons au nombre de n.

Bien qu'il s'agisse d'un problème faisant référence au niveau même de l'énoncé à la communication, pour le spécifier en utilisant les expressions de chemins, il faut exprimer les règles de synchronisation liées au support de la communication. En effet, la spécification est celle du contrôle des accès à un tampon de n cases. Les règles d'accès au tampon sont les suivantes:

1. le processus producteur ne peut déposer une valeur si le tampon est plein; il est alors bloqué jusqu' à une nouvelle consommation.

2. le processus consommateur ne peut prélever une valeur si le tampon est vide; il est alors bloqué jusqu' à une nouvelle production.

3. le producteur et le consommateur ne peuvent agir simultanément sur la même case (exclusion mutuelle).

Ces règles sont spécifiées par le module suivant:

Module prod-cons:
libre= n + **term**(consommer) − **aut**(produire)
occupe= **term**(produire) − **aut**(consommer)
chemin produire[libre > 0] + consommer[occupe > 0] **fin**
fin prod-cons

Dans ce module, l'opérateur + spécifie l'exclusion mutuelle entre les procédures produire et consommer (règle 3), alors que les prédicats [libre > 0] et [occupe > 0] spécifient respectivement les règles d'accès 1 et 2.

Du point de vue de la communication

Le problème du producteur-consommateur est lié fortement à la communication. Malheureusement dans la spécification précédente, on a spécifié le protocole nécessaire à la mise en oeuvre de celle-ci à travers une variable partagée. A aucun moment, on n' a fait référence aux valeurs échangées entre les processus, mais seulement aux contraintes qui régissent ces échanges. Nous dirons qu'il s'agit là d'une spécification orientée vers la mise en oeuvre par opposition à la spécification orientée problème.

Cette sur-spécification est dûe au fait que le concept d'histoire des expressions de chemins avec prédicats est adapté à la description de la synchronisation plutôt que de la la communication. Pour spécifier la communication, il faut garder l'histoire des valeurs échangées (valeurs produites, valeurs consommées) plutôt que l'histoire des événements nécessaires pour exprimer certaines contraintes de synchronisation (variables req, aut et term). Une réponse est apportée au dernier paragraphe où l'on définit la notion de type de communication. Cette notion permet de spécifier la communication comme une relation entre l'histoire des valeurs produites par un processus producteur et l'histoire des valeurs consommées par un processus consommateur. On en déduira la synchronisation nécessaire pour garantir l'exécution correcte de la communication spécifiée. L'intérêt des types de

communication est de s'abstraire du support et de séparer les problèmes.

3 Les modules abstraits

Les modules abstraits [KRO 87] s'inscrivent dans le cadre des nombreux travaux sur la logique temporelle. C'est pourquoi nous commencons tout d'abord par la présenter brièvement.

3.1 A propos de la logique temporelle

L'objet de la logique temporelle est de proposer un système formel permettant de raisonner sur les programmes et de spécifier leurs comportements. Ces comportements sont décrits par les propriétés que doivent vérifier les séquences de calculs associées aux programmes. Dans la littérature nombreuse sur le sujet [MAN 82] [OWI 82] [RAM 83] [LAM 83], on peut distinguer deux classes de propriétés :

- Les propriétés d'invariance ou de sûreté spécifient que certains événements ne doivent jamais se produire. La correction partielle pour les programmes séquentiels et l'absence de blocage dans les programmes parallèles font partie de cette classe.

- Les propriétés de vivacité garantissent l'occurrence de certains événements. La terminaison et l'équité en sont des exemples.

Pour exprimer de telles propriétés, différents opérateurs temporels sont proposés en plus des opérateurs de la logique classique. Nous en examinons quelques uns à travers le langage que nous présentons dans la suite de ce paragraphe.

Au delà de l'expression et de la preuve des propriétés que nous venons de citer, la logique temporelle a été aussi appliquée aux domaines de la sémantique et de la synthèse de programmes. A ce sujet, signalons par exemple la sémantique axiomatique et compositionnelle proposée dans [LAM 85], et la synthèse de programmes CSP présentée dans [MAN 84].

Nous avons choisi de présenter le langage de spécification proposé dans [KRO 87] et ceci pour plusieurs raisons. D'une part il inclut deux concepts importants dans la spécification des systèmes informatiques : les types abstraits de données et la logique temporelle. D'autre part les propriétés sont spécifiées de facon modulaire incluant des mots-clés qui facilitent la compréhension des spécifications. Cette dernière qualité est souvent négligée dans les langages de spécification fondés sur la logique temporelle, alors que c'est l'un des moyens pour mettre à la portée des programmeurs des techniques de spécification de haut niveau.

3.2 Le langage des modules abstraits

Parmi les nombreux modèles développés pour la spécification formelle des systèmes informatiques, les types abstraits de données et la logique temporelle occupent une place importante. Les spécifications obtenues avec ces modèles sont cependant assez différentes: les types abstraits de données permettent de spécifier des propriétés statiques tandis que la logique temporelle permet surtout de rendre compte de phénomènes temporels ou à caractère dynamique.
Les modules abstraits présentés dans [KRO 87] unifient ces deux aspects:

- Les types abstraits de données spécifiés algébriquement décrivent la partie fonctionnelle du module. Dans cette spécification, les propriétés des opérateurs des types sont exprimées par un ensemble d'axiomes ayant la forme d'équations [GUT 78] [GOG 79] [DER 79].
- La partie opérationnelle spécifie les règles que doivent respecter les procédures du module dans un environnement concurrentiel. Ces règles sont spécifiées à l'aide de la logique temporelle.

Nous ne présentons dans la suite que le langage de spécification de la partie opérationnelle des modules. Le lecteur intéressé par la spécification algébrique des types abstraits de données pourra consulter les références citées. Nous terminons par le même exemple que précédement; cela nous permettra de comparer les deux langages de spécification.

3.3 Le langage temporel

Avant de décrire le langage temporel utilisé, nous présentons figure 1.1 les différents états qui caractérisent l'exécution d'une procédure p quelconque d'un module abstrait. Nous notons "condp" la condition d'exécution de la procédure p: condition devant être vraie avant toute exécution de p.

instp, waitp, enablep, inp, startp, termp sont les symboles de prédicats qui vont permettre de savoir dans quel état se trouve le contrôle au moment de l'exécution de la procédure p.

On appelle formule atomique du langage temporel L_M, les expressions définies par les clauses suivantes:

1. $p(t_1, \ldots, t_k)$ où p est un symbole de prédicat.
2. init: signifie que le module est dans son état initial
   nil: signifie qu'aucune procédure du module n'est en cours d'exécution.

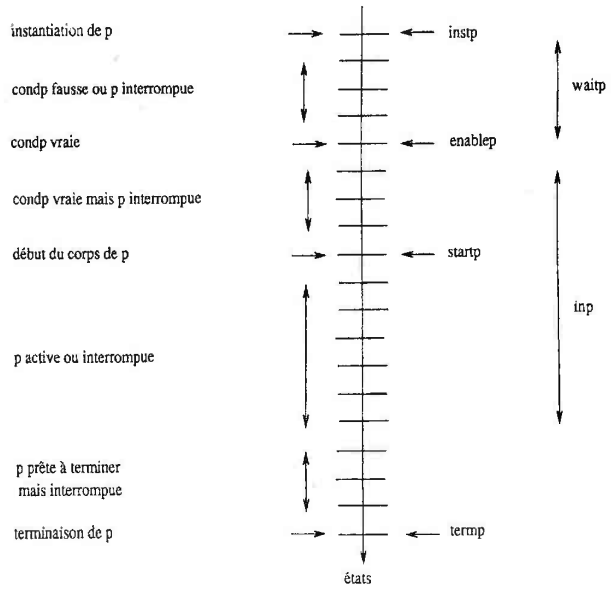


Figure 1.1: Etats caractérisant l'exécution d'une procédure p

3. les symboles décrits dans la figure 1 avec leurs paramètres, c-a-d $instp_i(t_1, \ldots, t_m)$, $\ldots, termp_i(t_1, \ldots, t_m)$.

Les formules du langage L_M sont alors définies par induction sur les formules atomiques en utilisant les opérateurs de la logique classique et les opérateurs de la logique temporelle suivants:

- $\circ$ A : A sera vrai à l'état immédiatement suivant.
- $\Box$ A : A est vrai maintenant et toujours.
- $\Diamond$ A : A est vrai maintenant ou plus tard
- A **atnext** B : A sera vrai à l'état immédiatement suivant celui ou B est vrai.
- A **until** B : A sera vrai jusqu' à ce que B le devienne.
- A **before** B : si B devient vrai dans un état futur, alors A sera vrai avant cet état.

Pour des raisons de clarté, certaines formules couramment utilisées ont été abrégées. Nous donnons ci-dessous celles qui seront utilisées dans l'exemple qui suit:

- $startp_i(t_1, \ldots, t_n) \wedge y = y_0 \rightarrow \Box(termp_i(t_1, \ldots, t_n) \rightarrow \circ\, y = h(\ldots y_0 \ldots))$
  Cette formule exprime que si avant l'exécution du corps de l'instance p_i la variable y a la valeur y_0, alors après la terminaison de p_i la variable y aura la valeur $h(\ldots y_0 \ldots)$ où h est un symbole de fonction. Ce qui est abrégé par:
  $startp_i(t_1, \ldots, t_n) \rightarrow new\ y = h(\ldots y \ldots)$
- $y = y_0 \rightarrow \circ(y = h(\ldots y_0 \ldots))$
  exprime que si y a comme valeur y_0, alors dans l'état immédiatement suivant y aura la valeur $h(\ldots y_0 \ldots)$. Cette formule devient:
  next $y = h(\ldots y \ldots)$
- $(v_1, \ldots, v_n)$ only changed by $(\sigma_i p_i(t_1, \ldots, t_n)\ ,\ \sigma_j g_j(s_1, \ldots, s_m)\ ,\ \ldots)$
  exprime que seules les actions $\sigma_i p_i, \sigma_j g_j, \ldots$ mentionnées dans la formule modifient les variables v_i. Les σ_k peuvent être inst, enable, term ou exec.

3.4 Exemple

La spécification de la communication entre les processus producteur et consommateur est donnée par le module abstrait Buffer. Ce module est basé sur le type abstrait algébrique *queue* spécifié ci-dessous. Dans le module on spécifie d'une part le service rendu par

les procédures *produire* et *consommer* à partir des opérations sur le type *queue*, et d'autre part les règles garantissant le bon fonctionnement de ces procédures.

type queue(element): empty, top, rest,append, isempty;

sorts : queue, element;

functions :
empty : $\rightarrow$ queue,
top : queue $\rightarrow$ element,
rest : queue $\rightarrow$ queue,
append : queue * element $\rightarrow$ queue,
isempty: queue $\rightarrow$ boolean,

laws :
q : queue, e : element
isempty(empty) = true,
isempty(append(q,e)) = false,
top(append(empty,e)) = e,
top(append(q,e)) = top(q), if q $\neq$ empty,
rest(append(empty,e)) = empty,
rest(append(q,e)) = append(rest(q),e), if q $\neq$ empty

endoftype

```
module Buffer(element): produire,consommer;
  based on : queue(element);
  variables : b: queue, r: element;
  procedures : produire(in: element), consommer(out: element);
  laws :
    (a) co modification de b oc
    b only changed by (execproduire,execconsommer);
    (b) co correction partielle oc
    startproduire(e) → new b=append(b,e),
    startconsommer(r) → new r=top(b) and new b=rest(b);
    (c) co terminaison oc
    startproduire → ◇ termproduire,
    startconsommer → ◇ termconsommer;
    (d) co condition d'exécution oc
    startproduire → true;
    startconsommer → not isempty(b);
    (e) co exclusion mutuelle oc
    not (inproduire_i ∧ inproduire_j) pour i ≠ j
    not(inconsommer_i ∧ inconsommer_j) pour i ≠ j
    not (inproduire ∧ inconsommer)
    (f) co priorité oc
    pour i ≠ j
    inproduire_i before inproduire_j → startproduire_i before startproduire_j
    inconsommer_i before inconsommer_j → startconsommer_i before startconsommer_j
  initial b=empty
endofmodule
```

Cette spécification appelle quelques remarques:

- Les parties (b), (d), (e) spécifient des propriétés de sûreté, tandis que les parties (c), (f) spécifient des propriétés de vivacité.
- La propriété (f) indique que les instances d'une même procédure s'exécutent dans l'ordre fifo; c'est une propriété d'équité.
- les procédures du module sont définies à partir des opérations sur le type *queue*. On reconnait là une démarche de spécification à deux niveaux, l'un concernant la partie fonctionnelle donnée par la spécification algébrique du type *queue*, l'autre utilisant la logique temporelle pour exprimer des propriétés temporelles.

Du point de vue de la communication

Comme nous l'avons signalé, la spécification d'une communication doit établir explicitement la relation entre les valeurs échangées par les processus. De ce point de vue, le module abstrait buffer peut être considéré comme une représentation abstraite de la relation "égalite" (toute valeur émise est reçue une et une seule fois dans l'ordre des émissions). Les règles d'accès spécifiées dans le module représentent alors les contraintes de synchronisation nécessaires à la mise en oeuvre d'une telle relation. Il est important pour nous de dissocier la spécification de la communication de sa mise en oeuvre; aussi nous éliminons dans notre langage les énoncés tel l'exclusion mutuelle. Toutefois, nous sommes conscients de l'utilité de tels énoncés que nous considérons seulement au niveau de l'implantation. Cette démarche est couramment utilisé dans la construction de programmes. Elle permet de procéder par étape et de repousser le plus loin possible les choix d'implantation et les contraintes dues à l'environnement (il s'agit ici d'un environnement parallèle).
Nous reviendrons sur ce point dans le paragraphe suivant.

4 Spécification de la communication

Dans les langages précédents, la spécification du système producteur-consommateur est exprimée en terme de relations entre les opérations produire et consommer. La relation entre les valeurs échangées par les processus n'apparait pas ou peu dans la spécification. Au contraire, le modèle proposé dans [PER 85] prend en compte l'échange de données entre les processus dès le niveau spécification. Il constitue, même la base d'une démarche méthodique et modulaire pour la construction de systèmes parallèles. En effet, l'objectif de cette démarche est de construire des solutions asynchrones permettant le parallélisme "maximal" tout en limitant le couplage entre les processus du système. Pour atteindre cet objectif, on introduit la notion de relation de communication, qui permet de regrouper en des points précis et spécifiques les échanges de données entre les processus du système.

4.1 Relations de communication

Intuitivement, une relation de communication entre deux processus p1 et p2 d'un système parallèle est une relation entre la suite de valeurs produites par p1 et la suite de valeurs consommées par p2. Cette suite est déduite de la première par modification du nombre d'occurrences et de l'ordre de ses termes sans en modifier les valeurs.

Une première spécification des relations de communication consiste à interpréter ces suites dans un modèle du parallélisme asynchrone fondé sur la notion de suites temporelles.

Cette notion introduite dans [PER 85] permet de prendre en compte les instants des requêtes d' opérations pour définir une relation de communication.

Notion de suites temporelles

Soit CHRONO un ensemble infini, dénombrable, et totalement ordonné par une relation d'ordre notée $\leq$ Etant donné un ensemble quelconque ELT, on note respectivement "*valeur*" et "*instant*" les opérations de projection définies sur le produit cartésien ELT*CHRONO.

Définition

On appelle suite temporelle de type ELT, toute application S de N dans le produit cartésien ELT*CHRONO, telle que pour tout i et j dans le domaine de S:

$$i < j \Rightarrow instant(S(i)) \leq instant(S(j))$$

La projection N $\rightarrow$ CHRONO d'une suite temporelle S est appelée l'horloge de S.

Relations de communication

Une relation de communication R (de type ELT) est une relation binaire sur l'ensemble des suites temporelles de type ELT qui vérifie:

Soient p et c deux suites temporelles de domaines P, C. (p,c)$\in$R si et seulement si :
$\exists\ \varphi$, application: C $\rightarrow$ P telle que:

$$\forall j \in C, \forall i \in P,\ \varphi(j) = i \implies valeur(c(j)) = valeur(p(i))$$
$$instant(c(j)) \geq instant(p(i))$$
$$\tau\ (p,c,i,j),\ \text{où } \tau \text{ est une relation qui caractérise R}$$

Etant données une relation de communication R, une suite temporelle p, il existe une infinité de suites temporelles c, telle que (p,c)$\in$R. Rappelons qu'il s'agit ici de définir des relations de communication entre processus. Aussi, on choisit l'une de ces suites en introduisant la suite de requêtes de consommation notée δ. Ceci nous amène à restreindre la définition précédente en lui ajoutant la propriété suivante:

$$instant(c(j)) = max(\delta(j), instant(p(i))$$

où δ est la suite de requêtes de consommation qui est une horloge, c la suite temporelle consommée relativement à la suite temporelle produite p. Enfin pour illustrer l'expressivité de ce formalisme, nous reprenons l'exemple traité précédement. Il s'agit d'une communication, où toute valeur produite est consommée une et une seule fois dans l'ordre des productions. Cette relation τ appelée égalité est spécifiée tout simplement comme suit:

$$égalite(p,c,i,j) \equiv (i \in P \wedge j \in C \wedge i = j)$$

On trouve dans [JAR 88], une présentation formelle de la notion de suite temporelle et son utilisation pour spécifier les systèmes réactifs.

4.2 Type de communication

La spécification d'une relation de communication en terme de suites temporelles, précise les propriétés du résultat (la suite consommée) mais ne donne aucun moyen pour l'obtenir. Ceci correspond d'ailleurs bien à ce que l'on peut attendre d'une première spécification [FIN 79]. Cependant, il est plus judicieux de proposer, dans une deuxième étape, une expression constructive et statique des relations de communication.

Etant données une relation R, une suite produite p, cette expression doit permettre de construire explicitement la suite consommée c, telle que (p,c) $\in$ R. Pour cela on propose dans [PER 85] une expression des relations de communication sous la forme d'un type abstrait de données appelé type de communication.

Définition

La définition d'un type de communication fait intervenir trois ensembles:

- L'ensemble des valeurs des éléments de la suite temporelle produite, désigné par SP,
- L'ensemble des valeurs de la suite temporelle consommée, désigné par SC,
- Un sous ensemble des valeurs de la suite temporelle produite appelé ensemble consommable et noté EA. Il représente l'ensemble des valeurs dont on peut disposer pour construire l'ensemble consommé. Il mémorise en fait des états partiels du type de communication et permet ainsi de définir l'opération "consommer", sans manipuler les instants des suites temporelles.

Nous notons <SP,EA,SC> ce triplet d'ensembles. Un type de communication est muni de trois constructeurs:

- init qui initialise un objet type de communication,

- produire, qui exprime la construction de la suite temporelle produite par adjonction d'une valeur aux ensembles désignés par SP et EA. Dans certains types de communication, l'adjonction de la valeur produite à l'ensemble consommable est soumise à une condition (pre-prod). Celle-ci s'interprète par l'élimination des valeurs qui ne pourront jamais être consommées.

- consommer, qui exprime la construction de la suite temporelle consommée conformément à la relation de communication. Cette construction se fait par extraction d'une valeur de l'ensemble désigné par EA, modification de cet ensemble et adjonction de cette valeur à l'ensemble désigné par SC.

Outre les opérations de projection qui permettent d'accéder à chacune des composantes d'un objet type de communication, les opérations suivantes permettent de constater son évolution:

- pre-prod: détermine la condition minimale que doit vérifier la valeur produite pour qu'elle fasse partie des données prêtes à être consommées.

- pre-cons: précondition sur la consommation. Elle restreint le domaine de définition de l'opération consommer.

- val-cons: détermine la valeur de la donnée communiquée prise en compte par l'opération de consommation.

- post-cons: post-condition de la consommation. Elle définit l'état de l'ensemble consommable, résultant d'une opération de consommation.

Les constructeurs sont définis de la même manière pour tous les types, alors que ces quatre dernières opérations sont spécifiques à chaque type de communication.

Avant de présenter la spécification des types de communication, permettons-nous une parenthèse pour préciser les types abstraits de données qu'elle fait intervenir. Nous introduisons le type suite pour définir les suites produites et consommées. On peut justifier le choix de ce type en examinant la définition des opérations de communication: on ne fait que concaténer des éléments à la suite produite (respectivement la suite consommée) par l'opération "produire" (respectivement "consommer") sans altérer les éléments existants. Par contre, l'ensemble consommable noté EA sera définit comme un objet de type Table, car la définition de l'opération consommer nécessite parfois un accès aléatoire aux éléments de cet ensemble.

Spécification

On appelle type de communication un type abstrait noté TYPECOM, paramétré par le type des valeurs communiquées noté ELT et défini par la spécification pré-post suivante:


```
type TYPECOM(ELT)=<SP:Suite(ELT),EA:Table(ELT),SC:Suite(ELT)>
  notation : tc=<SP,EA,SC>
  opérations :
    INIT () tc : TYPECOM
      pre : vrai
      post: tc=<creer(),t-creer(),creer()>
    PRODUIRE (tc : TYPECOM, e : ELT) tc' : TYPECOM
      pre : vrai
      post : tc'=<SP',EA',SC>
        avec SP'=ajouter(SP,e)
        EA'= si pre-prod((<SP,EA,SC>),e) alors t-ajouter(EA,long(SP)+1,e)
             sinon EA fsi
    CONSOMMER (tc : TYPECOM) tc' : TYPECOM
      pre : pre-cons(<SP,EA,SC>)
      post : tc'=<SP,EA',SC'>
        avec EA'=post-cons(<SP,EA,SC>)
          SC'=ajouter(SC,val-cons(<SP,EA,SC>))
    Opérations propres à chaque type de communication
    pre-prod (tc : TYPCOM) b : booleen
    pre-cons (tc : TYPCOM) b : booleen
    val-cons (tc : TYPCOM) e : ELT
    post-cons (tc : TYPCOM) t : TABLE
fin TYPECOM.
```

Cette définition appelle quelques remarques:

- Elle utilise le constructeur de type "produit cartésien" noté $< >$. Etant donnés n types $T_1, \ldots, T_n$ et n symboles $a_1, \ldots, a_n$, ce constructeur associe le type $T=< a_1 : T_1, \ldots, a_n : T_n >$ qui est le type produit cartésien des types T_i relativement aux sélecteur a_i. Dans notre cas les a_i définissent les fonctions de projection suivantes:

  SP,SC : TYPCOM $\rightarrow$ Suite, EA : TYPCOM $\rightarrow$ Table.

  Nous confondrons volontairement dans la suite ces fonctions et les trois éléments d'un objet de type TYPCOM, que nous noterons tc=<SP,EA,SC>.

- Il y est fait usage des types Suite, Table et de certains de leurs opérations, notamment les opérations creer respectivement t-creer (crée une suite vide respectivement

une table vide), ajouter respectivement t-ajouter (ajoute un élément à une suite respectivement à une table) et long (donne la longueur d'une suite).

- L'opération CONSOMMER est définie sous la condition pre-cons. Dans l'univers parallèle, la non définition de l'opération CONSOMMER est interprétée par une attente que la condition pre-cons devienne vraie.
- Contrairement à l'opération pre-cons, la condition pre-prod n'introduit pas d'attente. Elle spécifie seulement la condition minimale que doit vérifier la valeur produite pour qu'elle fasse partie des données prêtes à consommer.
- Pour spécifier un type TYPECOM, il suffit de définir les opérations pre-prod, pre-cons, val-cons et post-cons que nous appelons dans la suite opérations caractéristiques d'un type de communication.

Nous donnons ci-dessous la spécification algébrique du type Suite et du type Table. Nous notons Entier>0 l'ensemble des entiers strictement positifs, $+\infty$ le plus grand élément de cet ensemble, et *indef* l'élément de type ELT qui dénote la valeur indéfinie.

type Table (Entier>0,ELT)	**type** Suite (ELT)
opérations	**opérations**
t-creer ():Table	creer ():Suite
t-ajouter(Table,Entier>0,ELT):Table	ajouter(Suite,ELT):Suite
retirer(Table,Entier>0):Table	debut(Suite):Suite
extraire(Table,Entier>0):ELT $\cup\{indef\}$	reste(Suite):Suite
t-vide(Table):Booléen	vide(Suite):Booléen
max(Table):Entier>0	premier(Suite):ELT $\cup\{indef\}$
min(Table):Entier>0	dernier(Suite):ELT $\cup\{indef\}$
card(Table):Entier	long(Suite):Entier
axiomes	**axiomes**
soient t:Table, e1,e2:ELT, i1,i2:Entier>0	Soient s:Suite, e:ELT
retirer(t-creer(),i1)=t-creer()	debut(creer())=creer
retirer(t-ajouter(t,i1,e1),i2)= **si** i1=i2 **alors** t	debut(ajouter(s,e))=s
sinon t-ajouter(retirer(t,i2),i1,e1) **fsi**	reste(creer())=creer
extraire(t-creer(),i1)=*indef*	reste(ajouter(s,e))=ajouter(reste(s),e)
extraire(t-ajouter(t,i1,e1),i2)= **si** i1=i2 **alors** e1	vide(creer())=vrai
sinon extraire(t,i2) **fsi**	vide(ajouter(s,e))=faux
t-vide(t-creer())=vrai	premier(creer())=*indef*
t-vide(t-ajouter(t,i1,e1))=faux	premier(ajouter(s,e))=**si** s=creer() **alors** e
max(t-creer())=1	**sinon** premier(s) **fsi**
max(t-ajouter(t,i1,e1))= **si** max(t)<i1 **alors** i1	dernier(creer())=*indef*
sinon max(t) **fsi**	dernier(ajouter(s,e))=e
min(t-creer())=$+\infty$	long(creer())=0
min(t-ajouter(t,i1,e1))= **si** i1<min(t) **alors** i1	long(ajouter(s,e))=long(s)+1
sinon min(t) **fsi**	**fin** Suite
fin Table.	

Nous enrichissons le type Table par les opérations suivantes:

- t-premier, t-dernier: Table → ELT,
  t-premier(t)=extraire(t,min(t)), t-dernier(t)=extraire(t,max(t)).
- decapite, supder : Table → Table,
  decapite(t)=retirer(t,min(t)), supder(t)=retirer(t,max(t)).

4.3 Exemple

Le type égalite

Nous reprenons l'exemple de la communication égalitaire. La spécification des opérations caractéristiques est la suivante:

pre-prod(<SP,EA,SC>) = vrai
pre-cons(<SP,EA,SC>) = non t-vide(EA)
val-cons(<SP,EA,SC>) = extraire(EA,min(EA))
post-cons(<SP,EA,SC>) = retirer(EA,min(EA))

Pour exprimer une variante de ce type de communication, dans laquelle la réception n'est pas bloquante, il suffit de modifier l'opération pre-cons de la façon suivante: pre-cons(<SP,EA,SC>) = vrai. Les autres opérations restent inchangées. Le processus consommateur peut recevoir dans ce cas la valeur indéfinie notée *indef* lorsqu'aucune nouvelle valeur n'est émise.

Le type croissante

On suppose que le type ELT est muni d'une relation d'ordre notée "<". Ce type de communication est tel que les valeurs émises sont reçues une seule fois dans l'ordre de leur production, à la condition de former une suite croissante de valeurs de type ELT.

pre-prod(<SP,EA,SC>) = sup(SP,e)
pre-cons(<SP,EA,SC>) = non t-vide(EA)
val-cons(<SP,EA,SC>) = extraire(EA,min(EA))
post-cons(<SP,EA,SC>) = retirer(EA,min(EA))

Où sup est une nouvelle opération définie sur le type Suite. Cette opération vérifie si un élément donné est supérieur à tous les éléments d'une suite. Elle est définie par:

```
sup : Suite * ELT → booléen
sup (s,e) = si vide(s) alors vrai
                  sinonsi dernier(s) < e alors sup(debut(s),e)
                                          sinon faux
                  fsi
            fsi
```

Du point de vue de la communication

La communication constitue le fondement même de ce langage. Aussi l'utilisation des suites de valeurs échangées entre les processus dans la spécification a permis de définir la communication comme une relation entre ces suites. Il en résulte que les valeurs communiquées apparaissent explicitement dans la spécification, permettant ainsi à l'utilisateur de s'assurer que c'est la valeur envoyée qui sera reçue et non une autre. Ce dernier point est très important. Pour nous en rendre compte prenons le cas des expressions de chemins avec prédicats. En dehors du contrôle des accès, rien n'est précisé: Une mise en oeuvre d'un système ainsi spécifié peut délivrer des valeurs incohérentes sans mettre en défaut la spécification [RAY 81].

5 De la spécification à la mise en oeuvre

Au terme de cette analyse, nous constatons que les spécifications obtenues dans les langages précédents ne se situent pas au même niveau d'abstraction. Les expressions de chemins et les modules abstraits proposent des spécifications orientées mise en oeuvre, par opposition à la spécification orientée problème qu'offrent les relations et les types de communication.

Nous nous proposons dans ce paragraphe d'établir le lien entre ces deux niveaux , et ceci dans le cadre d'une démarche prenant en compte le processus complet de spécification et de mise en oeuvre de la communication. Cette démarche est schématisée par la figure 1.2, dans laquelle nous avons précisé pour chaque étape, l'expression obtenue et le langage support associé. Nous étudions ici le lien entre les types de communication et les modules abstraits.

En partant des relations de communication, la figure 1.2 montre les étapes permettant d'en dériver une mise en oeuvre dans un langage de programmation. Ainsi les flèches reliant les étapes peuvent être lues comme "représenté par" ou "implanté par".

Nous abordons les deux dernières étapes, en montrant à travers l'exemple de la communication égalité les différents problèmes rencontrés. La représentation des relations par

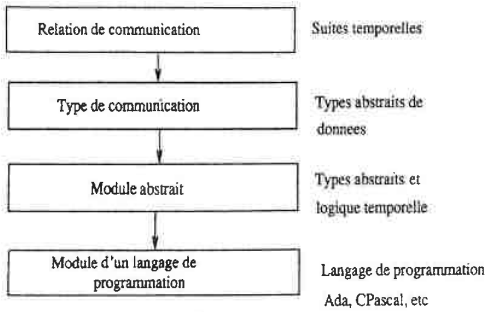


Figure 1.2: schéma d'implantation d'une relation de communication

les types de communication a été étudiée dans [PER 85] puis dans [MAR 89].

5.1 Type de communication-Module abstrait

Nous nous proposons dans ce paragraphe de définir l'implantation d'un type de communication par un module abstrait en introduisant deux étapes intermédiaires.

La première permet de représenter un type de communication par un type abstrait de données plus adapté à la définition des opérations de communication. La deuxième étape associe au type représentant les règles prenant en compte les caractéristiques de l'environnement parallèle des types de communication. De telles règles définissent la sémantique des opérations "produire" et "consommer" en utilisant la logique temporelle.

Nous obtenons alors au terme de cette deuxième étape un module abstrait fondé sur le type représentant et dont l'interface est constitué des opérations de communication.

Représentation

De manière intuitive, la représentation d'un type TS (type source) par un type TC (type cible) est une application qui définit les objets et les opérations du type TS à l'aide des objets et des opérations de type TC.

Pour qu'elle soit correcte, une représentation doit transporter tout ou partie des propriétés du type source. Nous nous intéressons ici aux représentations faibles [FIN 79] qui ne conservent que les propriétés des opérations externes : les opérations dont le codomaine n'est pas le type source.

Aussi pour prouver qu'une représentation faible ρ est correcte, il suffit de montrer

que tout théorème du type TS de la forme op(ts,t_1,...,t_n)=t_{n+1}, avec op une opération externe de profils TS*T_1 * ... * T_n → T_{n+1}, se transforme en un théorème de TC de la forme ρ(op)(ρ(ts),$\rho(t_1)$,...,$\rho(t_n)$) = $\rho(t_{n+1})$.

Nous allons étudier maintenant la représentation du type égalité en introduisant deux étapes élémentaires. Chaque étape construit un nouveau type plus concret que le type initial. Rappelons tout d'abord la spécification de ce type.

pre-prod(<SP,EA,SC>) = vrai
pre-cons(<SP,EA,SC>) = non t-vide(EA)
val-cons(<SP,EA,SC>) = extraire(EA,min(EA))
post-cons(<SP,EA,SC>) = retirer(EA,min(EA))

Etape 1

Une première représentation a pour but de transformer un type de communication en vue de simplifier sa structure. Rappelons que la structure d'un objet de tel type est définie par un produit cartésien de composantes de type suite et de type Table. Certaines de ces composantes introduisent parfois une redondance dans la spécification. Par exemple, les composantes SP et SC du type égalité n'interviennent pas dans la définition des opérations caractéristiques. Les seules opérations qui leurs sont appliquées sont les constructeurs **creer** et **ajouter**. Aussi on ne fait que construire de telles composantes sans accéder à leurs valeurs.

Nous définissons alors une représentation notée ρ_1 qui transforme le type égalité en un type noté TCS tel que:

Pour tout tc:égalité avec tc=<SP,EA,SC> on a ρ_1(tc)=tcs avec tcs=EA et tcs:TCS.

Cette définition montre que le type TCS n'est autre qu'un renommage du type Table. La représentation ρ_1 associe aux opérations du type égalité des opérations du type TCS de même nom. Notons que la représentation est l'identité sur les opérations caractéristiques (pre-prod, pre-cons, val-cons et post-cons). Seules les opérations de communication sont transformées par ρ_1. Leur représentation est donnée par les opérations suivantes.

PRODUIRE (tcs : TCS, e : ELT) tcs' : TCS	CONSOMMER (tcs : TCS) tcs' : TCS
pre : vrai	**pre** : pre-cons(tcs)
post : tcs'=ajouter(tcs,max(tcs)+1,e)	**post** : tcs'=post-cons(tcs)

Propriété: ρ_1 est une représentation faible du type égalité par le type TCS.

Preuve: Il suffit de montrer que les propriétés des opérations externes définies sur le type égalité sont conservées par ρ_1. Ceci est évident puisque ρ_1 est l'identité sur ces opérations. Prenons par exemple le cas de l'opération val-cons et établissons la preuve:

Soit tc:égalité, avec tc=<SP,EA,SC>, nous avons val-cons(tc)=extraire(EA,min(EA))

ρ_1(val-cons(tc)) = ρ_1(extraire(EA,min(EA)))
= extraire(EA,min(EA)), puisque ρ_1 est l'identité sur le type Table
=val-cons(ρ_1(tc))
=ρ_1(val-cons)(ρ_1(tc))

Le fait que ρ_1 soit une représentation faible nous assure que le type TCS définit la même stratégie de communication que le type égalité. L'étape suivante permet de représenter le type TCS par le type queue.

Etape2

L'analyse des opérations du type TCS montre que les objets de ce type sont des tables gérées comme des files. En effet, pour un objet tcs les adjonctions se font à l'entrée la plus grande notée max(tcs), les accès, et les suppressions ont lieu à l'entrée la plus petite notée min(tcs). Nous choisissons donc de représenter le type TCS par le type queue, déjà introduit au paragraphe 3.2. Définissons maintenant la fonction de représentation que nous notons ρ_2:

ρ_2(TCS) = queue
ρ_2(t-creer()) = empty()
ρ_2(t-ajouter(tcs,e)) = append(ρ_2(tcs),e)
ρ_2(t-vide(tcs)) = is-empty(ρ_2(tcs))
ρ_2(post-cons(tcs)) = ρ_2(retirer(tcs,min(tcs))) = rest(ρ_2(tcs))
ρ_2(val-cons(tcs)) = ρ_2(extraire(tcs,min(tcs))) = top(ρ_2(tcs))
ρ_2(pre-cons(tcs)) = non ρ_2(t-vide(tcs)) = non is-empty(ρ_2(tcs))

Les opérations de communication sont représentées par des opérations du type queue de même nom suivantes:

produire(q:queue,e:elt) q':queue	consommer(q:queue) q':queue
pre : vrai	pre : non is-empty(q)
post : q'=append(q,e)	post : q'=rest(q)

Propriété : ρ_2 est une représentation faible du type TCS par le type queue.

Preuve : Nous devons montrer que les propriétés des opérations externes sont conservées par ρ_2. Nous donnons une preuve par récurrence sur les objets de type TCS pour l'opération pre-cons. L'élaboration de la preuve pour les autres opérations suit le même principe.

Soit tcs un objet de type TCS,

- tcs=t-creer(), pre-cons(tcs)=faux

  ρ_2(pre-cons(tcs)) =non is-empty(ρ_2(tcs))
  =non is-empty(empty()), puisque ρ_2(tcs)=empty()
  =faux, *par définition du type queue page 19*

- tcs=t-ajouter(tcs',max(tcs)+1,e), nous avons dans ce cas pre-cons(tcs)=vrai

  ρ_2(pre-cons(tcs)) =non is-empty(ρ_2(tcs))
  =non is-empty(append(ρ_2(tcs'),e)), puisque ρ_2(tcs)=append(ρ_2(tcs'),e)
  =vrai, *par définition du type queue page 19*

Nous avons donc pour tout objet tcs:TCS, pre-cons(tcs)=ρ_2(pre-cons(tcs)) . (CQFD)

Avec cette deuxième représentation, nous clôturons la première étape pour le passage du type de communication au module abstrait qui l'implante.

Construction du module abstrait égalité

L'objet de cette étape est de construire le module abstrait égalité, ayant comme support le type queue et comme interface les opérations de communication.

Pour cela, il suffit de donner les règles qui garantissent l'utilisation correcte des opérations produire et consommer par les processus. Nous ne donnons pas une nouvelle fois de telles règles qui sont identiques à celles définies dans le module Buffer (paragraphe 3.2). Rappelons toutefois que l'on y trouve entre autres:

- une règle qui exprime que l'activation de l'opération consommer est assujettie à la condition pre-cons (règle (d)),
- une règle qui exprime que l'activation simultanée des opérations de communication est régie par le principe de l'exclusion mutuelle (règle (e)),
- une règle qui exprime que les appels d'une même opération de communication sont pris en compte selon leurs ordres d'arrivées (règle (f)).

Ces trois règles restent valables pour tous les types de communication. Notons qu' à cette étape les modules abstraits permettent d'exprimer différentes stratégies de synchronisation, contrairement aux types de communication.

Il s'agit maintenant d'implanter le module abstrait égalité dans un langage de programmation. Cette implantation doit respecter la sémantique des opérations de communication exprimée à l'aide des règles précédentes.

5.2 Module abstrait-programme

Nous proposons dans cette dernière étape une implantation du module abstrait égalité. Nous avons choisi d'utiliser pour cela le langage Ada [ICH 79] qui offre des possibilités de parallélisme (tâche, rendez-vous etc) et des outils d'abstraction (paquetage, paramétrisation).

L'implantation est construite de la manière suivante:

- Au type queue est associé un paquetage de même nom qui exporte le type file et les opérations associées. L'annexe c en donne une définition complète,
- le module égalité est implanté par un paquetage paramétré par le type des données communiquées et dont l'interface déclare les opérations de communication. Notons que ce paquetage n'exporte pas de type et encapsule l'état des objets. Nous donnons ci-dessous la partie spécification et la partie corps de ce paquetage.

generic	**with** queue;
type elt **is private**	**package body** egalite **is**
package egalite **is**	**package** file_elt **is new** queue(elt);
procedure produire(e:**in** elt);	b : file_elt.file;
procedure consommer(e:**out** elt);	-implémentation des opérations produire et consommer
end egalite;	**end** egalite;

Nous nous intéressons maintenant à l'implantation des opérations de communication. Nous devons pour cela tenir compte des règles (exclusion mutuelle, condition d'exécution,..) exprimées dans le module abstrait égalité. C'est pourquoi nous allons introduire la notion de point d'entrée définie dans les tâches Ada pour mettre en oeuvre ces opérations.

Une solution consiste à définir une tâche à deux entrées, chacune implante une opération de communication. Ci-dessous nous donnons l'interface et le corps de cette tâche ainsi que l'implantation des opérations produire et consommer.

```
--spécification de la tache inter
task inter is
 entry p (x:in elt);
 entry c ( x:out elt);
end inter;

--implémentation de l'opération produire
procedure produire (e: in elt) is
begin
 inter.p(e);
end produire;

--implémentation de l'opération consommer
procedure consommer (e: out elt) is
begin
 inter.c(e);
end consommer;
```

```
--corps de la tache inter
task body inter is
 v : elt;
begin
 loop
  select
   accept p (x:in elt) do
    v := x;
   end p;
   append(b,v);
  or
   when not empty(b) =>
   accept c (x:out elt) do
    x := top(b);
   end c;
   rest(b);
  end select;
 end loop;
end inter;
```

Etablissons maintenant la correspondance entre les règles spécifiées dans le module abstrait égalité et la solution proposée:

- L'exécution de l'opération consommer est assujettie à la condition pre-cons précisée dans la clause **when**. D'où la réalisation de la règle (d) du module,
- l'exclusion mutuelle entre les opérations produire et consommer est assurée par construction dans la tâche inter. En effet toutes les entrées d'une tâche Ada s'exécutent en exclusion mutuelle. Ceci réalise donc bien la règle (e) du module abstrait,
- les appels à une même entrée d'une tâche Ada sont toujours pris en compte selon leur ordre d'arrivée. D'où la réalisation de la règle d'équité (f) du module.

Nous remarquons donc que l'implantation Ada que nous avons associée au module abstrait respecte bien les règles spécifiées dans ce module.

5.3 Qu' avons-nous fait ?

Tout au long de ce paragraphe, nous nous sommes efforcés de définir le processus d'implantation des types de communication. Un tel processus comme nous l'avons vu, est décrit par une succession d'étapes élémentaires permettant de progresser vers un programme.

Ainsi nous avons introduit des étapes de représentation qui consistent à remplacer un type par un autre plus concret ou plus propice à l'implantation. Ensuite nous avons associé au type représentant un module abstrait dans lequel sont définies les règles qui expriment

la sémantique des opérations de communication. C'est seulement à cette étape que nous avons pris en compte les règles de synchronisation nécessaires à la mise en oeuvre de la communication. Enfin l'implantation de ce module en Ada, comme on a pu s'en rendre compte ne pose pas de problèmes ardus.

Toutes ces étapes seront étudiées de manière plus détaillée dans dans la deuxième partie où nous abordons le problème de leur automatisation.

6 Conclusion

Nous avons présenté et analysé dans ce chapitre quelques propositions de langages pour la spécification des relations entre processus d'un système parallèle.

Les expressions de chemins et les modules abstraits permettent de spécifier essentiellement le concept de synchronisation, soit par la définition explicite de relations entre des occurrences d'opérations, soit par le respect de certaines contraintes d'accès à des objets partagés. Aussi les deux langages comme nous l'avons signalé ne traitent pas explicitement la communication en tant que tel: sa spécification est confondue avec celle de sa mise en oeuvre à travers une variable partagée.

Les type de communication que nous avons présentés à la fin de ce chapitre évitent cette confusion en proposant un modèle où la communication est spécifiée en terme de relation entre les suites de valeurs échangées par les processus. La spécification obtenue écarte par là même les énoncés parasites tel l'exclusion mutuelle; qui ne relèvent normalement que de l'implantation de la communication.

Enfin l'utilisation effective des types de communication nécessite leur mise en oeuvre, pour cela un certain nombre de langages sont proposés. Nous étudions dans le chapitre suivant comment exprimer des systèmes parallèles en termes de processus communicants à l'aide d'un langage intégrant la notion de type de communication. Ensuite, nous présentons le langage Ada que nous avons choisi pour mettre en oeuvre ces systèmes.

Chapitre 2

Langages d'expression de systèmes parallèles

1 Introduction

Après avoir présenté le concept de type de communication, nous nous intéressons dans ce chapitre à son utilisation comme outil de communication dans un langage d'expression de systèmes parallèles baptisé Lesp.

Proposé dans [PER 85] comme langage de conception de systèmes parallèles, Lesp exprime un parallélisme fondé sur la coopération entre processus non déterministes, et communicant de manière asynchrone par des ports (instances de types de communication). Deux sortes de modules sont fournies par le langage:

- les types de communication qui spécifient les relations entre processus, indépendamment de tout souci de synchronisation,
- les processus qui décrivent des suites de calculs, et utilisent les opérations définies dans les types de communication pour échanger des données.

L'introduction de ces deux modules permet de séparer dans un système parallèle la définition des processus de celle de leur communication. Cette séparation est fondée sur une typologie qui permet de diviser les problèmes, afin d'améliorer l'évolutivité des systèmes obtenus.

Pour concevoir et mettre en oeuvre les programmes Lesp, un environnement de programmation appelé COMEDIE propose: un éditeur syntaxique, un interprète et un compilateur vers le langage Ada. C'est à ce problème de compilation que nous consacrons les deux prochains chapitres. Nous nous contentons pour l'instant de présenter les concepts fondamentaux pour l'expression du parallélisme proposés par les deux langages Lesp et Ada. Mais auparavant, il convient de donner une vue d'ensemble sur l'expression du parallélisme dans les langages de programmation. Nous insistons plus particulièrement sur les

relations de coopération qu'ils proposent de la même manière que nous l'avons fait pour les langages de spécification (cf *chapitre 1*).

2 Expression du parallélisme dans les langages de programmation

2.1 Généralités

Longtemps réservée à la programmation des systèmes d'exploitation, l'utilisation du parallélisme est devenue une nécessité pour résoudre d'autres classes de problèmes tels que les calculs scientifiques, les systèmes de contrôle de procédés industriels, etc.

Cette nécessité a fait naître plusieurs propositions pour l'expression du parallélisme, notamment dans le domaine de la conception des langages de programmation. A ce sujet, deux concepts sont introduits entre autres dans les langages pour ainsi rendre compte d'une façon claire et structurée du parallélisme:

- le concept de processus qui permet de maîtriser la notion d'activité, et vient enrichir les différents modules déjà existants dans les langages,
- le concept de coopération qui représente les relations entre processus: relation de concurrence, de communication ou de compétition.

La structure et la mise en oeuvre de tels concepts dans les langages parallèles sont nombreuses, et il serait difficile de les présenter toutes. Aussi nous nous contentons ici d'une synthèse permettant de classer les langages en fonction de deux critères.

Le premier est fondé sur le mode de coopération qu'ils proposent, et sera abordé dans le paragraphe suivant. Le second critère concerne la classe de systèmes à laquelle ils sont dédiés [PER 87] :

- les systèmes transformationnels qui se présentent comme des fonctions définissant des résultats à partir des données, indépendamment de leurs environnements,
- les systèmes réactifs qui, contrairement aux précédents, ont pour principale fonction de maintenir une certaine relation avec leurs environnements.

Pour un système transformationnel, la mise en œuvre d'une solution parallèle consiste à répartir les fonctions à réaliser en un ensemble de processus coopérants. Les langages dédiés à cette classe de systèmes adoptent généralement un style impératif. Citons par exemple CPascal [HAN 75], CSP [HOA 78], Ada [ICH 79], LC3 [CER 85]. Deux autres catégories de langages offrent aussi des solutions pour cette classe de systèmes : les langages data-flow et les langages vectoriels.

Pour les systèmes réactifs, la mise en œuvre d'une solution doit tenir compte de l'indépendance de certains événements, et permettre éventuellement leur prise en compte de manière simultanée. Les langages dédiés à cette classe adoptent généralement une démarche consistant à exprimer le comportement des systèmes. Citons par exemple LUSTRE [CAS 87], ESTEREL [BER 85], SIGNAL [GUE 84], C-Net [ADA 89].

2.2 La coopération

Comme nous l'avons dit plus haut, le concept de coopération constitue un autre critère à partir duquel on peut établir une classification des langages. Par coopération, nous entendons toutes les relations pouvant exister entre les processus composant un système parallèle: relation de concurrence, de communication ou de compétition. Nous écartons la relation de compétition qui exprime une coopération implicite entre processus, non pas pour la réalisation d'une tâche commune mais pour l'utilisation de ressources telles que l'unité centrale, la mémoire etc. Les deux autres types de relation expriment une coopération explicite et permettent de distinguer deux classes de langages que nous présentons maintenant.

Relation de concurrence

Il s'agit du cas où les processus composants un système utilisent et mettent à jour un objet partagé. Pour cela, ils disposent d'opérations de consultation et de modification définies sur cet objet.

Le problème ici est d'assurer le contrôle de ces opérations afin de maintenir la cohérence de l'état de l'objet partagé. Les langages fondés sur ce mode de coopération proposent des outils de synchronisation dont le but est de mettre en oeuvre l'exclusion mutuelle entre les opérations. Citons le langage CPascal, SIMONE [HER 80], et Portal [SCH 87] qui introduisent le concept de moniteur. Un moniteur est un module qui regroupe l'objet partagé et les procédures qui le manipulent. Parmi celles ci, certaines sont internes au moniteur, d'autres sont publiques et peuvent être appelées de l'extérieur. A chacune des procédures publiques est associée une file d'attente des processus appelants, qui sont réveillés lorsque l'activation est possible. La mise en attente ou l'activation des processus est assurée par l'utilisation de primitives de synchronisation delay (ou wait) et continue (ou signal).

Si on se place du point de vue de la communication, nous dirons qu'elle ne constitue pas un concept de base de cette famille de langages. L'expression de toute forme de communication doit être programmée en terme d'un contrôleur d'accès à l' objet partagé qui sert de support aux valeurs communiquées.

Signalons aussi l'exclusion mutuelle imposée systématiquement par le moniteur et qui

n'est pas toujours nécessaire. Nous retrouvons encore ici, les inconvénients déjà cités dans le chapitre précédent concernant les expressions de chemins et les modules abstraits.

Relation de communication

Il s'agit du cas où les processus s'échangent des données de manière explicite. Pour cela, ils disposent de deux primitives de communication: l'envoi et la réception d'une donnée.

Les langages qui prennent en compte ce type de relation se distinguent essentiellement par le mode de désignation des processus interlocuteurs et la synchronisation qu'ils présupposent.

Mode de désignation

L'identité des interlocuteurs peut être mentionnée ou non dans les primitives de communication. On dit que la communication est directe lorsque les processus se désignent mutuellement par leur nom, comme dans CSP et PLITS. Autrement la communication est dite indirecte et peut être:

- symétrique : les processus utilisent le nom d'un objet intermédiaire de type boîte aux lettres ou canal. Par exemple LC3, OCCAM [MAY 83], COOL [ADA 86]
- asymétrique: seul l'identité du processus appelé est précisée comme c'est le cas dans le langage Ada.

La communication directe rend non modulaire l'écriture des processus et limite parfois leurs possibilités de communication (voir pour cela l'analyse faite à propos du langage CSP dans [GUE 80]). Par ailleurs, il n'est pas possible de créer une bibliothèque de processus serveurs réalisant des fonctions couramment utilisées. En effet, la définition d'un processus serveur nécessite de connaître l'identité précise de ses interlocuteurs. Or ces derniers ne sont pas généralement connus lors d'une telle définition.

Synchronisation

Selon la synchronisation mise en jeu lors de la communication, nous distinguons deux situations:

- communication synchrone : l'envoi et la réception d'une donnée sont bloquants. C'est le cas des langages fondés sur le concept de rendez-vous : CSP, OCCAM, ADA,

- communication asynchrone : les processus communiquent à travers un tampon. La seule synchronisation qui existe entre l'envoi et la réception est celle qu'implique la règle de causalité et éventuellement la taille du tampon. Citons par exemple LC3 et GYPSY.

Bien que la communication soit un concept de base de ces deux familles de langages, elle reste toutefois limitée soit à une communication synchrone de type rendez-vous, ou asynchrone de type séquence.

Dans les langages à rendez-vous, l'obtention d'une communication asynchrone n'est pas immédiate et nécessite l'introduction de processus intermédiaires. Ces derniers jouent alors le rôle de gestionnaire des tampons de communication. C'est seulement ainsi que l'on peut atteindre les solutions offertes directement par la famille de langages à communication asynchrone. Cependant, il est aussi regrettable que la stratégie de communication de type séquence (communication égalitaire ou FIFO) soit la seule proposée par cette deuxième famille de langages. Pourtant des exemples de systèmes parallèles tels que ceux présentés dans [JUL 83] [PER 85], montrent l'intérêt de l'expression d'autres stratégies de communication que la séquence. En introduisant les types de communication, le langage que nous présentons maintenant, évite cette lacune.

3 Langage d'expression de systèmes parallèles

Le langage Lesp a été proposé dans [PER 85] pour l'écriture de systèmes parallèles. Par rapport à la classe de langages impératifs asynchrones, il propose le concept de type de communication. D'autre part il inclut les constructions existantes dans les langages de programmation, tels que CPascal, Ada et CSP. L'idée originale est d'abstraire l'expression de la communication entre processus en introduisant trois concepts : port, donnée communiquée et type de communication.

Le langage présuppose une approche de conception où l'on doit analyser clairement ce qui dans le problème spécifié, ressort de la définition d'une relation de communication, et ce qui ressort de la définition des processus. Cette approche conduit naturellement le concepteur à séparer dans un système, l'expression de la communication du traitement des valeurs communiquées. Aussi le langage est bien adapté aux problèmes où la communication est l'essence même de l'expression d'une solution à un problème.

Ci-dessous nous présentons successivement les concepts fondamentaux permettant l'expression du parallélisme dans le langage, puis leur sémantique. Nous présentons ensuite comment concevoir dans le langage certains schémas de composition de processus tels que la diffusion, la divergence et la convergence.

3.1 Les processus

Les processus formant un système parallèle sont séquentiels et en nombre statiquement défini. Un processus a une structure analogue à celle d'une tâche Ada: il possède un nom, un interface et un corps décrivant son comportement. Comme nous l'avons signalé, un système parallèle est décrit par un ensemble hiérarchisé de processus communicants. Selon l'emplacement d'un processus dans cette hiérarchie, nous distinguons:

- Les processus élémentaires qui constituent les feuilles de la hiérarchie. Un processus élémentaire est constitué de son interface qui décrit l'utilisation des ports de communication, et d'un corps décrivant son comportement indépendamment des autres processus.

- Les processus composés qui regroupent des processus internes s'exécutant en parallèle et constituent un véritable système. A la différence des processus élémentaires, les processus composés n'exécutent qu'une seule action: l'activation en parallèle des processus internes notée //.

Un processus se présente de la manière suivante:

```
processus p ::
  définitions des données communiquées
  entree : ...
  sortie : ...
  (1) déclarations des entités locales au processus
  corps
  ...
fin p;
```

Les clauses entrée et sortie constituent l'interface du processus. Dans le cas du processus principal, racine de la hiérarchie, cette partie est vide.

La partie (1) définit des entités locales au processus : déclaration de variables, de types et de sous-programmes. Dans un processus composé communicant, on trouve en plus la définition des processus internes et leurs relations de communication.

Nous allons maintenant examiner la définition et l'utilisation des moyens de communication entre processus.

3.2 Expression de la communication

Les ports

L'outil primitif sur lequel repose la communication est le port. Un port constitue un objet, support de communication à travers lequel les processus peuvent envoyer ou recevoir des valeurs.

Dans Lesp, les ports sont des instances de types de communication et sont utilisés pour définir l'interface des processus d'un système parallèle. Les ports sont toujours déclarés dans un processus composé et utilisés dans les clauses entrée et sortie des processus qui lui sont internes. Comme exemple de déclaration, soit le port p véhiculant des valeurs entières selon la relation de communication spécifiée dans le type de communication TC:

port p : TC (entier)

Dans cette déclaration, TC représente un modèle de communication auquel est fourni en paramètre effectif le type des données communiquées définies dans l'interface des processus. L'instanciation de ce modèle produit alors un type de communication effectif, muni des opérations de communication "produire" et "consommer".

Interface du processus

L'interface d'un processus décrit l'utilisation de ports de communication définis dans le processus englobant. Cette utilisation est exprimée dans les clauses entrée et sortie:

- Une clause entrée est constituée de noms de données typées, et pour chaque donnée le port utilisé pour recevoir les valeurs de cette donnée.

- Une clause sortie est constituée de noms de données typées, et pour chaque donnée, la liste des ports utilisés pour transmettre les valeurs de cette donnée.

Les données figurant dans une liste d'entrée ou une liste de sortie sont appelées données communiquées. C'est sur ces données que sont définies les actions de communication "produire" et "consommer" utilisées dans le corps des processus. Contrairement à Ada et CSP, il n'y a pas de désignation de processus interlocuteurs dans les actions de communication, ce qui rend modulaire l'écriture des processus.

Pour illustrer l'expression de la communication dans le langage moyennant le concept de port et de données communiquées, nous reprenons l'exemple du système producteur-

consommateur déjà introduit au *chapitre 1.* Il s'agit d'un système composé de deux processus nommés prod et cons qui communiquent selon la relation spécifiée par le type égalité (communication FIFO). Supposons pour cela que le type de communication "égalite" a été spécifié et intéressons-nous ici au schéma de communication entre les deux processus et à son expression dans le langage Lesp.

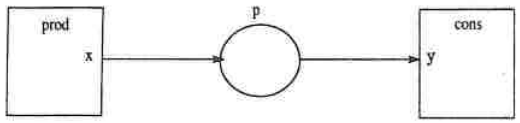


Figure 2.1: schéma de communication bipoint

Ce schéma se traduit en Lesp par un processus composé, où sont déclarés les deux processus prod et cons ainsi que le port p.

```
processus prod-cons ::
  port p : égalité (entier);
  processus prod ::
    sortie x : vers p;
    corps
    .....
    produire(x);
    .....
  fin prod;
  processus cons ::
    entree y : via p;
    corps
    .....
    consommer(y);
    .....
  fin cons;
  corps
  prod // cons;
fin prod-cons;
```

La structuration de la communication ainsi offerte par le concept de port et de données communiquées permet de séparer la description de la stratégie de communication de la description des processus. Ceci favorise la modularité et facilite la modification des programmes: chaque processus est indépendant du nombre, du nom et de la structure des autres processus avec lesquels il communique.

3.3 Structure de choix non déterministe

Outre les instructions de contrôle classiques (itération, conditionnelle,...), le langage offre une construction dérivée des commandes gardées [DIJ 75] introduisant le non-déterminisme:

select
  c_1...$enonce_1$
ou
:
:
ou
  c_n...$enonce_n$
fin select

Chaque composante de l'instruction **select** est formée d'une garde suivie d'une suite d'instructions. Une garde (c_i) est une expression booléenne qui, lorsqu'elle n'est pas vérifiée a pour effet de supprimer la prise en compte de l'énoncé associé.

Dans le cas où plusieurs gardes sont vérifiées, le choix de l'énoncé à exécuter se fait arbitrairement: c'est le non déterminisme. L'exécution d'une instruction "select" produit un état d'erreur lorsqu' aucune garde n'est satisfaite.

3.4 A propos de la sémantique

Une sémantique formelle du langage Lesp a été présentée dans [PER 85], puis dans [MAR 89] en utilisant une approche qualifiée d'opérationnelle. Cet approche définit un programme comme une machine abstraite ou automate, généralement sous la forme d'un système de transitions. Une démarche plus récente fondée sur l'approche opérationnelle utilise un système de règles d'inférence pour exprimer les règles de transition de la machine abstraite [PLO 81]. C'est cette démarche qui a été adoptée dans [MAR 89] et qui a conduit à la réalisation d'un interprète du langage Lesp. Notons cependant l'hypothèse faite dans cette réalisation concernant l'introduction d'un référentiel de temps global et l'association d'une horloge à chaque processus. L'explicitation du temps permet ainsi, dans le cadre de l'interprétation, de modéliser réellement la simultanéité d'actions.

Nous considérons pour notre part une sémantique opérationnelle non déterministe, qui définit un système parallèle par l'entrelacement des actions de chacun des processus qui le composent. Nous nous contentons ici d'une approche intuitive permettant de comprendre la signification des constructions importantes du langage Lesp. Pour cela, nous com-

mencons par la définition de la sémantique des déclarations des ports de communication, des entrées et des sorties. Par la suite, nous présentons la sémantique des opérations de communication et de la composition parallèle des processus.

Déclaration de port

port p : TC(t)

TC représente un modèle de communication auquel est fourni en paramètre effectif le type des données communiquées désigné par t. Cette déclaration a pour effet d'instancier ce modèle, produisant ainsi un type de communication effectif désigné par p, muni des opérations de communication "produire" et "consommer".

Déclaration d'entrée

entree e : via p

Notons t le type des données communiquées mentionné dans la déclaration du port p

L'effet de cette déclaration dépend en partie de la nature du processus dans lequel elle apparait. Soit Q un tel processus:

- si Q est un processus élémentaire, il y a création d'une variable locale à ce processus, de nom e et de type t,
- si Q est un processus composé, il y a création d'une variable locale à chacun de ses processus internes, de nom e et de type t.

Dans ces deux cas, il y a association entre la (les) variable(s) locale(s) et le port de nom p. Cette association va permettre de déterminer le port concerné par l'activation de l'énoncé "consommer(e)".

Déclaration de sortie

sortie s : vers p

Cette déclaration produit le même effet que la déclaration d'entrée, hormis le fait que la (les) variable(s) locale(s) créée(s) doivent être associées à tous les ports désignés par $p_1, \ldots, p_n$. Une telle association va permettre de déterminer les ports concernés par l'activation de l'énoncé "produire(s)".

Enoncés de communication

Ces énoncés correspondent aux opérations "produire" et "consommer" définies sur les ports et activées par les processus.

Notons $\underline{x}$ la valeur associée à la donnée x, $\underline{p}$=<SP,EA,SC> l'état mémoire du port p=<SP,EA,SC> où SP, EA, et SC représentent, respectivement, la suite produite, l'ensemble consommable et la suite consommée du port p.

1. Production : **produire**(s), où s est une sortie associée aux ports $p_1, \ldots, p_n$

   cette opération a pour effet d'appliquer simultanément l'opération "produire" du type TYPECOM sur les objets désignés par $p_1, \ldots, p_n$. Le nouvel état mémoire $\underline{p_i}$' $\underline{p_i}$'=<$\underline{SP_i}$',$\underline{EA_i}$',$\underline{SC_i}$'> du port p_i est défini de la manière suivante :
   $\underline{SP_i}$' = ajouter($\underline{SP_i}$,$\underline{s}$),
   $\underline{EA_i}$' = **si** pre-prod($\underline{p_i}$) **alors** i-ajouter($\underline{EA_i}$,long($\underline{SP_i}$)+1,$\underline{s}$) **sinon** $\underline{EA_i}$ **fsi**
   $\underline{SC_i}$' = $\underline{SC_i}$.

   L'état mémoire du processus producteur reste inchangé.

2. Consommation : **consommer**(e) , où e est une entrée associée au port p.

   - si la valeur pre-cons($\underline{p}$) est vraie, l'opération "consommer" est activée et a pour effet:
     - de modifier l'état mémoire du processus consommateur comme suit:
       $\underline{e}$ = val-cons($\underline{p}$)
     - de modifier l'état $\underline{p}$ par l'application de l'opération définie par le type de communication associé au port p. Nous obtenons un nouvel état $\underline{p}$'=<$\underline{SP}$',$\underline{EA}$',$\underline{SC}$'> défini de la manière suivante :
       $\underline{SP}$' = $\underline{SP}$, $\underline{EA}$' = post-cons($\underline{p}$), $\underline{SC}$' = ajouter($\underline{SC}$,val-cons($\underline{p}$))
   - si la valeur pre-cons($\underline{p}$) est fausse, l'activation de l'opération consommer est retardée jusqu'à ce que l'état $\underline{p}$ soit tel que pre-cons($\underline{p}$) devienne vrai. On dit que le processus consommateur est mis en attente.

Composition parallèle

L'opérateur noté // a pour effet d'activer en parallèle les corps des processus composant un système. Nous lui donnons une sémantique non déterministe : les actions des processus activés seront entrelacées tout en respectant les attentes liées à l'activation de l'opération consommer (lorsque sa pré-condition n'est pas satisfaite).

Nous illustrons ces idées sur un exemple de système parallèle composé de deux processus nommés prod et cons. Nous définissons la sémantique de ce système par l'ensemble des séquences d'actions construites par l'entrelacement indéterministe des actions engagées par les deux processus. Pour cela nous distinguons deux cas selon que les actions engagées contiennent ou non des opérations de communication.

1. soient prod et cons deux processus définis par les séquences d'actions suivantes :
   prod = x ; y, cons = z ; t
   Où ";" dénote la séquence et x, y, z, t sont des actions atomiques qui ne sont pas des actions de communication.

   L'ensemble des séquences d'actions possibles du système "prod // cons" est le suivant: { x;y;z;t , x;z;y;t , x;z;t;y , z;t;x;y , z;x;t;y , z;x;y;t }.
   L'exécution du système "prod // cons" consiste donc à exécuter l'un de ces six entrelacements.

2. considérons maintenant le cas où les processus prod et cons comportent des actions de communication : prod = x ; p ; y, cons = z ; c ; t
   Où p est l'opération produire, c est l'opération consommer définies sur le même port de communication.

   Pour construire l'ensemble des séquences d'actions du système "prod // cons", il faut prendre en compte le fait que l' opération consommer n'est pas toujours activable. En effet, l'activation de cette opération nécessite que sa pré-condition (pre-cons) soit vérifiée. Nous allons examiner ce cas en supposant que le port de communication est de type égalité. Rappelons que toute activation de l'opération consommer n'est possible que si l'ensemble consommable n'est pas vide (pre-cons(tc)=non t-vide(EA)). Dans les séquences d'actions du système "prod // cons", l'action p doit nécessairement précéder l'action c. Ceci résulte du fait que l'action c du processus cons ne peut être activée (sa pré-condition est fausse) avant l'action p du processus prod. Aussi l'ensemble des actions possibles pour ce système est le suivant:
   { x;p;y;z;c;t , x;p;z;y;c;t , x;p;z;c;y;t , x;p;z;c;t;y , x;z;p;y;c;t , x;z;p;c;y;t , x;z;p;c;t;y , z;x;p;y;c;t , z;x;p;c;y;t , z;x;p;c;t;y }

Considérons, pour terminer, la composition parallèle "p // c" de l'action produire et de l'action consommer définies sur le même port. Lorsque la pré-condition de l'action c est fausse, nous retrouvons le cas 2 çi-dessus. Il en résulte que "p;c" est la seule séquence possible. En revanche, lorsque cette pré-condition est vérifiée avant l'activation de l'opération p, nous aurons deux séquences d'actions possibles "p;c" et "c;p". Notons pour ce dernier cas, une différence essentielle avec la sémantique présentée dans [MAR 89], qui décrit l'exécution simultanée grâce à l'introduction du temps. Elle privilégie la production

sur la consommation, et transforme la composition parallèle "p // c" en la séquence "p;c" considérée comme une action atomique.

3.5 Composition de processus

La communication décrite dans le système présenté par la figure 2.1 établit une liaison entre un processus producteur et un processus consommateur. Cette liaison qualifiée de bipoint n'est pas suffisante dans de nombreuses situations où plus de deux interlocuteurs sont en communication. Nous nous intéressons ici à trois situations typiques de la communication entre processus: la diffusion, la divergence et la convergence.

Pour chacune d'elles, nous donnons d'une part un schéma indiquant les chemins de communication entre processus, et d'autre part le système Lesp correspondant à ce schéma.

La diffusion :

La valeur émise par un producteur est diffusée vers n consommateurs. Les ports utilisés à cet effet peuvent être ou non de même type de communication. Cette situation est illustrée par la figure 2.2.

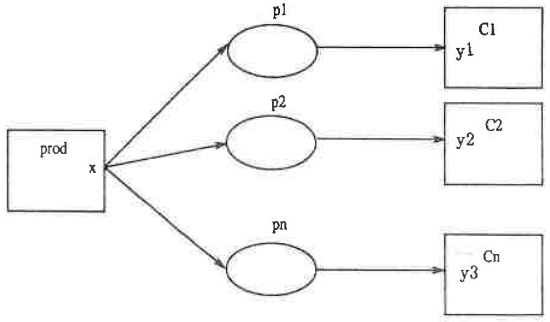


Figure 2.2: schéma de diffusion

Comme nous le remarquons sur ce schéma, la diffusion peut être considérée comme un ensemble de communications bipoints. La figure 2.3 présente Le système Lesp correspondant pour n=3.

```
processus diff ::
  port
    p1,p2 : tc1(type);p1 et p2 sont de même type
    p3 : tc2(type);
  processus prod ::
    sortie x : vers p1,p2,p3;
    corps
    produire(x);
  fin prod;
  processus C1 ::
    entree y1 : via p1;
    corps
    consommer(y1);
  fin C1;
  processus C2 ::
    entree y2 : via p2;
    corps
    consommer(y2);
  fin C2;
  processus C3 ::
    entree y3 : via p3;
    corps
    consommer(y3);
  fin C3;
  corps
  prod // C1 // C2 // C3;
fin diff;
```

Figure 2.3: composition de processus par diffusion

La divergence:

Toutes les valeurs émises par un producteur sont consommées par un des processus consommateurs, au hasard de leur activation. D'où le schéma de la figure 2.4.

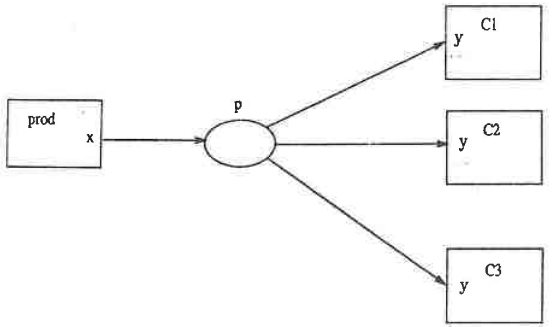


Figure 2.4: schéma de divergence

La conception du système Lesp associée à ce schéma suit la démarche suivante :

- regroupement des processus "C_i" en un système cons dont l'interface se réduit à la déclaration de l'entrée y. Ce système définit en fait un processus communicant mettant en composition parallèle les processus élémentaires "C_i". L'entrée y ainsi déclarée peut apparaître dans une opération de communication "consommer" activée depuis l'un de ces processus,

- le type du port est un type de communication généralisé, dont la structure est composée:

  - d'une suite produite,
  - d'une suite consommable,
  - d'un n-uplet de suites consommées, chacune étant associée à un des processus "C_i".

On obtient ainsi le système illustré par la figure 2.5.

```
processus div ::
  port p : tc(type);
  processus prod ::
    sortie x : vers p;
    corps
    produire(x);
  fin prod;
  processus cons ::
    entree y : via p; entrée globale à tous les processus
    processus Ci ::
      corps
      consommer(y);
    fin Ci;
    corps
    // Ci , i = 1,n
  fin cons;
  corps
  prod // cons;
fin div;
```

Figure 2.5: composition de processus par divergence

La convergence:

A chaque activation de l'opération de consommation, une parmi l'entrelacement des valeurs émises par les processus producteurs est consommée par un processus consommateur. Le schéma de la figure 2.6 illustre cette situation.

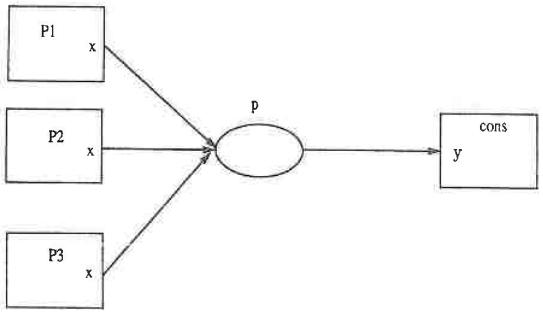


Figure 2.6: schéma de convergence


```
processus conv ::
  port p : tc(type);
  processus prod ::
    sortie x : vers p;
    processus P1 ::
      corps
      produire(x);
    fin P1;
    processus P2 ::
      corps
      produire(x);
    fin P2;
    corps
    P1 // P2;
  fin prod;
  processus cons ::
    entree y : via p;
    corps
    consommer(y);
  fin cons;
  corps
  prod // cons;
fin conv;
```

Figure 2.7: composition de processus par convergence

Pour obtenir le système Lesp associé à ce schéma, nous regroupons les processus P_i en un sous-système producteur dont l'interface est constitué de la déclaration de la sortie x. Ce sous-système assure ainsi la composition parallèle des processus P_i qui réalisent les opérations de production de la donnée x au hasard de leurs activations. La figure 2.7 présente un exemple de système Lesp associé à ce schéma:

4 Le langage Ada

4.1 Présentation générale

Ada [DOD 83] est un langage de programmation adapté à l'écriture d'applications générales, notamment en environnement temps réel.

Un programme Ada est composé d'un ensemble d'unités de programmes, pouvant être compilées séparément. Une unité de programme peut être l'un des modules suivants :

- une procédure ou une fonction qui correspondent à la notion habituelle de sous-

programme,

- un paquetage qui permet d'encapsuler un groupe d'entités logiquement reliées. Ce type de module supporte directement les principes de modularité, d'abstraction, de localisation et de dissimulation d'informations,

- une tâche qui définit un traitement pouvant s'exécuter en parallèle avec d'autres tâches, et matérialise la notion de processus,

- une unité générique qui définit un modèle de sous programmes ou de paquetages, accompagné de paramètres génériques et donne ainsi la possibilité d'ajuster ce modèle aux besoins particuliers au moment de l'instantiation.

Toutes ces unités possèdent une structure similaire en deux parties, constituée d'une spécification et d'un corps. La spécification d'une unité identifie l'information visible pour ses clients, alors que le corps contient les détails d'implantation de l'unité qui peuvent être logiquement et textuellement cachés aux clients.

Comme la spécification et le corps peuvent être compilés séparément, il est facile de construire la spécification tôt dans le développement d'une application et d'ajouter le corps plus tard [BOO 84]. C'est en fait l'essence même de l'essai de l'utilisation d'Ada en tant que langage de conception. Cependant, la diversité des constructions proposées par le langage et leur puissance d'expression nécessitent non seulement une excellente connaissance du langage, mais aussi des concepts qui en sont fondateurs. Ce n'est qu'ainsi que de telles constructions peuvent être appliquées de manière appropriée. Le tableau ci-dessous recense les différentes applications de chacune des unités de programmes dans la construction d'une solution en Ada.

Sous-programme :	programme principal, définitions d'opérations sur des types.
Paquetage :	ensemble de déclarations, groupement d'unités de programmes reliées, types abstraits de données, machines abstraites.
Tâche :	actions concurrentes, routages de messages, contrôle de ressources, gestion des interruptions.

Les deux premières unités peuvent être paramétrées par des variables, des types et des sous-programmes.

4.2 Expression du parallélisme

Parmi les objectifs fixés pour la définition du langage Ada, figurait la nécessité de pouvoir programmer directement des systèmes parallèles et en particulier des applications temps réel. Cet objectif s'est concrétisé par l'introduction des constructions suivantes:

- le concept de tâche qui matérialise la notion de processus,

- la communication et la synchronisation exprimées à l'aide du mécanisme de rendez-vous,

- les interruptions et les clauses de représentation,

- la gestion du temps et, dans une certaine mesure, des priorités.

Les tâches

Une tâche est une unité de programmation pouvant s'exécuter en parallèle avec d'autres tâches. Elle ne peut être isolée : une tâche dépend de son parent qui est essentiellement l'unité dans laquelle la tâche a été déclarée. L'activation des tâches se fait au début de la partie corps du parent dans un ordre indéterminé; il n'y a pas d'activation explicite comme dans Lesp.

La partie spécification d'une tâche comprend la déclaration des entrées qui définissent les services de coopération offerts aux autres unités de programmes. La déclaration et l'utilisation d'une entrée sont similaires à celles d'une procédure. A toute spécification de tâche est associé un corps, qui doit contenir pour chaque entrée au moins une instruction **accept** définissant le traitement à effectuer lors de la prise en compte d'un appel sur cette entrée. L'exemple qui suit présente une solution synchronisée du système producteur-consommateur:

```
task prod;                          task cons is
                                      entry consommer (x : in elt);
task body prod is                   end cons;
  x : elt;
begin                               task body cons is
  ...                               begin
  cons.consommer(x);                  ...
  ...                                 accept consommer (x : in elt) do
end prod;                             ...
                                      end consommer;
                                      ...
                                    end cons;
```

Expression de la communication

La communication est assurée par le mécanisme de rendez-vous. Sur l'exemple du système précédent, la communication a lieu lorsque "prod" et "cons" élaborent respectivement l'appel "consommer" et l'énoncé "accept" associé. La valeur de la donnée "x" est alors transférée de "prod" à "cons" qui exécute le traitement contenu dans l'énoncé accept et les deux tâches continuent en parallèle.

Comme nous le voyons immédiatement, il n'y a pas ici, contrairement à Lesp, de communication asynchrone et les tâches ne sont pas banalisées les unes vis-à-vis des autres.

Si plusieurs tâches appellent la même entrée, les appels sont rangés dans une file d'attente associée à cette entrée.

Expression de l'indéterminisme

En plus des structures de contrôle classiques, les tâches sont dotées d'une structure non déterministe qui leur permet de sélectionner une communication parmi plusieurs. Comme le rendez-vous en Ada n'est pas symétrique, on distingue pour cette structure plusieurs formes selon son utilisation dans une tâche appelée : attente sélective, ou dans une tâche appelante : appel sélectif.

Attente sélective

C'est une instruction composée dont chacune des composantes est constituée d'une garde éventuellement vide suivie d'une branche. Parmi ces composantes, il doit y avoir toujours au moins une branche dite branche de rendez-vous, commençant par une instruction "accept". Nous illustrons l'utilisation d'une telle instruction par un exemple de corps de tâche permettant de temporiser les messages échangés entre deux processus.

```
select
when not plein => accept produire (e : in elt) do
                  ...
                  end produire;
or
when not vide => accept consommer (e : out elt) do
                  ...
                  end consommer;
end select;
```

En plus de ces branches dites de rendez-vous, on peut avoir des branches de temporisation, de terminaison ou une partie **else**.

Une branche est dite ouverte si la garde associée est vérifiée ou si cette garde est vide.

Voyons maintenant comment fonctionne une telle instruction :

1. toutes les gardes sont d'abord évaluées pour déterminer les branches ouvertes,
2. une branche ouverte commençant par une instruction "accept" est sélectionnée si le rendez-vous correspondant est possible. L'instruction "accept" et les éventuelles instructions suivantes sont alors exécutées.
3. si aucune branche de rendez-vous ne peut être sélectionnée, plusieurs cas sont envisageables :
   - attente simple : la tâche attend jusqu'au premier appel de l'un quelconque des points d'entrée associées aux branches de rendez-vous ouvertes,
   - attente avec délai : correspond au cas où au moins une branche temporisée est ouverte. Si aucun rendez-vous n'a lieu avant l'expiration du délai spécifié, les instructions de la branche indiquant ce délai sont exécutées,
   - attente avec terminaison : correspond au cas où il existe une branche de terminaison ouverte. L'attente est alors limitée par la durée de vie de la tâche qui dépend de son environnement.
   - rendez-vous immédiat : correspond au cas où il existe une instruction avec une partie **else**. Si aucun rendez-vous n'est possible immédiatement, la partie **else** est exécutée et l'instruction **select** est terminée.

Notons qu'un cas d'erreur est détecté lorsqu'aucune branche n'est ouverte au moment de l'exécution d'une instruction **select** sans partie **else**.

Appel sélectif

Cette instruction permet à une tâche appelante d'annuler une demande de rendez-vous lorsque celui-ci n'est pas possible immédiatement (appel conditionnel), ou après un intervalle de temps donné (appel avec délais).

Appel conditionnel:	Appel avec délais:
select	**select**
tampon.produire(x);	tampon.consommer(x)
...	...
else	**or delay** t
...	...
end select	**end select**

Ces deux formes de l'instruction **select** interdisent la présence des gardes dans les appels sélectifs.

5 Conclusion

La différence essentielle entre les deux langages que nous venons de présenter réside dans leurs modes d'expression de la coopération et en particulier de la communication entre processus.

Bien qu'il soit utilisé pour exprimer la communication en Ada, le rendez-vous est avant tout un outil de synchronisation. L'expression de toute forme asynchrone oblige le programmeur à gérer les tampons nécessaires en introduisant des processus intermédiaires. Pour que ces intermédiaires soient une mise en œuvre fidèle des formes de communication voulues, nous pensons qu'ils doivent être programmés en référence à des expressions abstraites et non pas les introduire artificiellement.

En introduisant les types de communication comme outil pour l'expression abstraite de la communication entre processus, le langage Lesp est plus adapté à la conception d'une première solution proche du problème à résoudre. Une telle solution peut ensuite être traduite en Ada [KOU 86]. Les chapitres suivants présentent un système permettant d'automatiser cette traduction.

Partie II

Etude de l'implantation automatique des types de communication et des systèmes parallèles Lesp en Ada

Le chapitre 3 pose le problème de la transformation des types de communication, justifie et met en évidence l'intérêt de la décomposer en deux grandes étapes.

La première dite de représentation fait l'objet du chapitre 4. Elle permet de représenter un type de communication par des structures de données et des fonctions facilement exprimables dans un langage de programmation.

Le chapitre 5 quant à lui présente la deuxième étape dite de compilation. Celle-ci construit autour de la représentation un module du langage Ada.

Au chapitre 6, nous présentons la spécification de la compilation des systèmes Lesp en programmes Ada. Celle-ci est définie par un système formel à base de règles d'inférence.

Chapitre 3

Implantation des types de communication

1 Introduction

En partant de la spécification d'un type de communication, l'objectif de ce chapitre est d'étudier les principales stratégies et techniques pour automatiser son implantation dans le langage Ada. Cette étude soulève deux problèmes que l'on peut résumer en deux points :

- le problème de la représentation : la spécification d'un type de communication est exprimée en termes de types abstraits de données. Elle introduit des objets et des opérations de types Suite et Table, spécifiés sous forme algébrique. Le passage à un programme pose alors le problème désormais classique, de la représentation de ces objets et de ces opérations en termes de structures de données et de fonctions connues par un langage de programmation.
- le problème de l'environnement : nous appelons environnement d'un type l'ensemble des processus qui l'utilisent pour la communication. Cette utilisation est soumise à des contraintes de synchronisation dûes à l'activation en parallèle des opérations de communication. Par conséquent, l'implantation doit en plus du problème de la représentation, prendre en compte le caractère parallèle de cet environnement.

L'énoncé de ces deux problèmes montre bien que l'implantation (qu'elle soit automatique ou non) d'un type de communication ne peut se faire de manière directe et en une seule étape. D'importantes transformations sont alors nécessaires pour obtenir un programme exécutable. Ces constatations nous ont conduit à faire appel à des techniques de transformation de programmes selon une approche dite transformationnelle. Dans cette approche, l'implantation d'un type de communication est vue comme une succession d'étapes élémentaires. Le passage d'une étape à une autre se fait par l'application d'une

règle de transformation soit sur les données soit sur les opérations manipulant ces données. En suivant cette approche, nous avons entrepris l'écriture d'un système d'implantation des types de communication en Ada. Ce système procède en deux grandes étapes correspondant aux deux problèmes soulevés précédemment:

- une étape de représentation qui transforme un type de communication pour aboutir à des types de données et fonctions proches d'un langage de programmation. Cette étape identifie les types de communication à des types abstraits de données classiques (au sens de la programmation séquentielle). Elle fait donc abstraction de leur environnement.
- une étape de passage au niveau dynamique qui construit autour de la représentation du type un module du langage Ada prenant en compte les contraintes d'implantation dûes au parallélisme de l'environnement.

Nous consacrons ce chapitre à l'étude de l'étape de représentation. La transformation vers le langage Ada fera l'objet du chapitre suivant. Nous présentons en premier lieu le langage d'expression des types de communication. Cette présentation va nous permettre de bien préciser les énoncés qui constituent la donnée du système d'implantation des types de communication. Ensuite, nous donnons une description générale de l'étape de représentation avant de présenter les règles de transformation qui en constituent le fondement.

2 Langage d'expression des types de communication

Comme dans la majorité des travaux sur la transformation de programmes, il est important de caractériser le langage des énoncés sur lesquels vont porter les transformations. Ainsi nous consacrons ce paragraphe au langage d'expression des types de communication.

2.1 Présentation générale

Nous avons présenté les types de communication sous la forme d'un type abstrait de données, muni de deux opérations "produire" et "consommer" (**cf** *chapitre 1*). La spécification de ce type est fondée sur les types abstraits Suite et Table et possède les caractéristiques suivantes:

- il est défini par un produit cartésien,
- les opérations "produire" et "consommer" sont définies à l'aide des opérations caractéristiques d'une relation de communication que sont:
  - pre-prod : restreint l'ensemble des données produites,
    pre-prod : TYPCOM × ELT → booléen

  - pre-cons : restreint le domaine de définition de l'opération de consommation, pre-cons : TYPCOM → booléen
  - val-cons : définit la valeur de la donnée communiquée prise en compte par l'opération de consommation, val-cons : TYPCOM → ELT
  - post-cons : définit l'état de l'ensemble consommable résultant d'une opération de consommation, post-cons : TYPCOM → Table

L'ensemble constitué des types Suite, Table, booléen, entier et ELT (types externes) constitue ce que nous appellerons, par la suite, l'univers d'un type de communication.

Comme nous l'avons mentionné auparavant, spécifier un type de communication revient à définir ces quatre opérations. Le langage utilisé à cet effet est un langage fonctionnel permettant de manipuler essentiellement des suites et des tables. La classe de fonctions qu'il décrit est définie inductivement à partir des fonctions prédéfinies sur les types Suite et Table, par introduction de la conditionnelle, la composition fonctionnelle et la récursion.

Nous présentons tout d'abord les fonctions prédéfinies, puis nous précisons la forme des fonctions que l'on peut construire dans le langage.

2.2 Les fonctions prédéfinies

Elles constituent les primitives du langage. Les tableaux suivants dressent la liste de telles fonctions en présentant leurs profils et leurs significations.

opérations	profils	signification
creer	→ Suite	crée une suite vide
ajouter	Suite × elt → Suite	adjonction d'un élément
debut	Suite × entier → Suite	suppression du dernier élément
reste	Suite → Suite	supprime le premier élément d'une suite
vide	Suite → booleen	teste si une suite est vide
long	Suite → entier	nombre d'éléments d'une Suite
premier	Suite → elt	accès au premier élément d'une suite
dernier	Suite → elt	accès au dernier élément d'une suite

opérations	profils	signification
t-creer	→ Table	crée une table vide
t-ajouter	Table × Entier × elt → Table	adjonction d'un élément
retirer	Table × Entier → Table	suppression d'un élément
extraire	Table × entier → elt	accès à un élément
t-vide	Table → booleen	teste si une table est vide
card	Table → entier	nombre d'éléments d'une table
min	Table → entier	l'entrée la plus petite d'une table
max	Table → entier	l'entrée la plus grande d'une table
t-dans	Table × entier → booleen	teste si un entier est une entrée
decapite	Table → Table	supprime l'élément dont l'indice est le plus petit d'une table
supder	Table → Table	supprime l'élément dont l'indice est le plus grand d'une table
t-premier	Table → elt	accès à l'élément dont l'indice est le plus petit d'une table
t-dernier	Table → elt	accès à l'élément dont l'indice est le plus grand d'une table

2.3 Définition de nouvelles fonctions

L'ensemble des fonctions prédéfinies sur les types Suite et Table s'avère parfois insuffisant pour spécifier certains types de communication. Pour cela, le langage offre la possibilité d'enrichir cet ensemble en introduisant de nouvelles fonctions.

La définition d'une nouvelle fonction consiste en deux parties:

- le profil qui précise le domaine et le codomaine de la fonction,
- la définition proprement dite de la fonction exprimée à l'aide de:
  - la récursion qui est un schéma élaboré à partir de la conditionnelle et de la composition fonctionnelle,
  - les opérations prédéfinies sur les types Suite et Table.

Pour construire une fonction, on peut adopter une première stratégie fondée sur une définition inductive des types Suite et Table. Classiquement, le domaine de définition d'une fonction peut être décrit inductivement à l'aide d'un ensemble de fonctions appelées constructeurs. De telles fonctions permettent d'engendrer tout les objets de ce domaine. Par exemple { **creer, ajouter** } est un ensemble de constructeurs permettant d'engendrer toutes les suites d'éléments de même type. Cette décomposition du domaine conduit à définir une fonction par induction sur les constructeurs à l'aide d'une suite d'équations. C'est ainsi que nous avons décrit les fonctions prédéfinies sur les types Suite et Table dans *le chapitre 1.*

Une autre définition plus constructive et qui dérive directement de la précédente consiste à exprimer les opérations par récurrence sur le type Suite, en utilisant des fonctions appelées observateurs (les opérations dont le codomaine n'est pas le type Suite) et simplificateurs (les opérations dont le codomaine est le type Suite et qui ne sont pas des constructeurs). Par exemple les fonctions **debut** et **reste** sont des simplificateurs, les fonctions **vide, dernier** et **premier** sont des observateurs. La définition d'une fonction peut alors s'exprimer sous la forme suivante:

f(s,d) = **si** vide(s) **alors** ....
sinon h(f(debut(s),d),...)
fsi

Ce schéma n'est qu'un exemple pour illustrer cette seconde stratégie, on peut imaginer d'autres observateurs et simplificateurs pour construire des opérations plus complexes. C'est cette approche que nous avons adoptée. En un sens, ce type de définition correspond à la notion de schéma de décomposition utilisée dans le système CATY [BID 87] pour construire des algorithmes sur des types abstraits de données.

3 Etape de représentation

Nous avons décomposé l'implantation d'un type de communication en deux grandes étapes:

- une étape de représentation qui a pour but de représenter un type de communication par des types de données proches d'un langage de programmation,
- une étape de passage au niveau dynamique qui construit autour de la représentation un module du langage Ada.

La difficulté essentielle dans l'implantation des types de communication réside dans l'étape de représentation. Il en est de même en ce qui concerne les types abstraits de données en général. C'est pour cette raison que nous énonçons tout d'abord le problème de représentation, en précisant les options que nous avons choisies. Nous donnons ensuite une description générale de l'étape de représentation des types de communication, après quoi nous l'abordons de manière plus détaillée.

3.1 le problème de représentation

Le problème de la représentation des types abstraits apparaît généralement lors du processus de transformation d'une spécification en vue de l'obtention d'un programme.

Dans ce processus, la représentation consiste à exprimer les types de la spécification à l'aide de structures de données ou types de bases connus par le langage de programmation cible. Sur un plan pratique, la représentation d'un type ne s'effectue pas en une seule étape, le passage par des représentations intermédiaires est souvent nécessaire. Précisons maintenant cette notion de représentation.

Très schématiquement, on peut dire qu'un type TC est une représentation d'un type TS, s'il existe une application ρ de l'ensemble des objets de TS dans celui des objets de TC que l'on note ρ(TS)=TC.

Cependant, il est évident que n'importe quel type TC ne peut être une représentation correcte de TS. Aussi on exige que l'application ρ vérifie certaines conditions:

- ρ doit associer à chaque axiome du type TS un théorème du type TC. On parle dans ce cas de représentation **généralisée** [FIN 79] : toutes les propriétés du type TS sont conservées par l'application ρ,

- ρ ne doit conserver que les propriétés des opérations externes du type TS. On parle dans ce deuxième cas de représentation **faible** [FIN 79,LEV 84] : seul le comportement du type TS vis-à-vis des opérations externes (les opérations dont le codomaine n'est pas le type TS) est pris en compte par ρ.

La représentation généralisée introduit une contrainte importante dans la mesure où elle demande le transport de toutes les propriétés du type source. De ce fait, elle limite beaucoup les possibilités de représentation. En revanche, la notion de représentation faible est moins restrictive, ce qui d'un point de vue pratique la rend plus intéressante que la précédente. C'est cette notion que nous utilisons dans ce travail. Nous en donnons une définition plus précise.

Définition

Une représentation faible ρ d'un type TS par un type TC est une application qui transforme tout théorème de TS de la forme $\mathrm{op}(\mathrm{ts},t_1,\ldots,t_{n-1}) = t_n$ avec op une opération externe sur TS de profil TS $*T_1 * \ldots T_{n-1} \rightarrow T_n$
en un théorème de TC de la forme $\rho(\mathrm{op})(\rho(\mathrm{ts}),\rho(t_1),\ldots,\rho(t_{n-1})) = \rho(t_n)$.

Les types T_i constituent l'univers de la spécification du type TS. Il arrive fréquemment que la représentation ρ soit l'identité sur ces types ($\rho(T_i) = T_i$). C'est le cas des représentations que nous serons amené à traiter dans la suite de ce travail. La définition précédente peut alors s'énoncer de la manière suivante:

Une application ρ est une représentation faible d'un type TS par un type TC si pour toute opération externe op sur TS de profil TS $*T_1 * \ldots T_{n-1} \rightarrow T_n$, on a:

$\mathrm{op}(\mathrm{ts},t_1,\ldots,t_{n-1}) = \rho(\mathrm{op})(\rho(\mathrm{ts}),t_1,\ldots,t_{n-1})$.

Rappelons pour terminer, les travaux qui ont concerné le problème de la représentation des types abstraits de données. Nous en distinguons deux catégories. La première regroupe les études qui ont permis de cerner essentiellement les propriétés des représentations et les preuves de correction associées [GUT 78] [GAU 78] [FIN 79] [REM 82]. Dans la plupart de ces travaux, il s'agit d'inventer une représentation et d'en faire une preuve à posteriori. Signalons toutefois que dans certains cas, on essaye de synthétiser la représentation des opérations du type source connaissant le type cible [GAU 78].

La deuxième catégorie concerne l'automatisation de la construction de représentation. Nous abordons plus en détail certains de ces travaux au *chapitre 8*. Contentons nous pour l'instant d'une brève présentation. Le système PSI [BAR 79] utilise une base de connaissances très riche en règles de nature variée, parmi lesquelles certaines concernent la représentation des types de données. Le système APE [BAR 81] produit des programmes INTERLISP, réalisant des types abstraits algébriques et des algorithmes portant sur ces types. Ces derniers sont spécifiés sous forme de règles de réécriture conditionnelles. Enfin le système SAIDA [GRA 88] qui, à partir d'un texte Ada portant sur des objets abstraits et dont la représentation n'est pas donnée, permet de les représenter par des structures de données concrètes. Notons que ce système a été conçu pour l'enseignement assisté par ordinateur de la programmation. La caractéristique commune de ces trois travaux est l'utilisation de systèmes experts à base de règles de production.

3.2 Principe

Le passage d'une spécification (un type abstrait ou l'énoncé d'un problème) à un programme se fait rarement de manière directe. Des transformations sont le plus souvent nécessaires. Par une approche transformationnelle il est possible de réécrire progressivement une spécification en un programme. Cette approche signifie que l'on procède par application de règles de transformation supposées valides afin d'assurer la correction des programmes obtenus à chaque étape. C'est ainsi que raisonnent et travaillent les partisans de l'approche transformationnelle [BUR 77] [BRO 81] [FEA 86]. La représentation des types de communication suit une telle approche. Comme il s'agit ici de représenter des types abstraits de données, les règles de transformation s'identifient à des règles de représentation. Nous voulons que de telles règles soient valides afin d'assurer la correction du résultat de leur application. Pour cela, nous devons les placer dans un cadre formel, qui seul peut nous permettre d'établir des preuves. Dans ce but, nous considérons une règle de transformation comme une fonction de représentation (au sens du paragraphe précédent) d'un type abstrait par un autre. La validité d'une règle consiste alors en la preuve de correction d'une telle fonction. Nous utilisons les résultats des travaux sur les

preuves de représentation que nous avons présenté plus haut.

3.3 Les étapes

Nous avons insisté à plusieurs reprises sur l'importance de la notion d'étape dans l'organisation du processus d'implantation. Nous présentons ici les étapes principales qui composent l'étape de représentation des types de communication. Nous nous contentons d'une brève description dont le but est de donner une vue intuitive et globale sur l'organisation d'une telle étape au sein du système d'implantation des types de communication. Les règles de transformation qui en constituent le fondement ainsi que leurs preuves seront présentées dans les paragraphes suivants. Nous prenons comme exemple illustratif le type de communication TC défini par la spécification suivante:

pre-prod(<SP,EA,SC>,e) = meme(SP,e) < p pre-cons(<SP,EA,SC>) = non t-vide(EA) val-cons(<SP,EA,SC>) = extraire(EA,min(EA)) post-cons(<SP,EA,SC>) = retirer(EA,min(EA))

L'opération "meme" est définie sur le type Suite de la manière suivante:

meme : Suite × ELT → Entier
meme (s,e) =
si vide(s) **alors** 0
sinon si dernier(s)=e **alors** meme(supder(s),e) + 1
sinon 0 **fsi**
fsi

En partant du dernier élément d'une suite s, l'opération "meme" calcule le nombre d'éléments consécutifs de s, identiques à un élément donné e.

Analyse

Cette étape prend en entrée la spécification d'un type de communication défini par les quatre opérations caractéristiques (pre-prod, pre-cons, val-cons, post-cons) et rend en sortie pour chaque composante du type, l'ensemble des opérations qui lui sont appliquées. On ajoute à cet ensemble les constructeurs qui sont appliqués sur les composantes de tous les types de communication (voir les opérations "produire" et "consommer"). Se sont les propriétés de ces opérations qui vont guider la représentation.

L'intérêt de cette étape est de simuler le recueil d'informations nécessaires qu'utilise habituellement le programmeur pour implanter une spécification. Elle évite par là-même

au système de demander à l'utilisateur des informations pouvant être extraites automatiquement de la spécification. Par exemple, l'étape d'analyse associe au type TC les ensembles suivants:

- pour la composante SP : {creer, ajouter, meme},
- pour la composante EA : {t-creer, t-ajouter, t-vide, extraire, retirer},
- pour la composante SC : {creer, ajouter},

Ces ensembles vont être utilisés par les étapes suivantes.

Simplification

Cette étape a pour but de transformer la spécification d'un type en vue d'en éliminer la redondance. De manière intuitive, la simplification consiste à supprimer de la spécification d'un type de communication la ou les composantes dont les valeurs n'interviennent pas dans cette spécification, autrement dit, les composantes qui ne sont manipulées par aucune opération d'accès (ou opération externe). Pour le type TC, la composante SC en est un exemple. Dans la spécification de ce type, on ne fait que construire cette composante par les constructeurs "creer" et "ajouter" sans utiliser leurs valeurs. La simplification du type TC produit les résultats suivants:

- simplification de la structure du type:
  Avant la simplification:
  type TC(ELT)=<SP:Suite(ELT),EA:Table(ELT),SC:Suite(ELT)>,
  Après la simplification:
  type TC(ELT)=<SP:Suite(ELT),EA:Table(ELT)>.
- les opérations caractéristiques restent inchangées, puisqu'elles n'utilisent pas la composante SC,
- simplification des opérations "init", "produire" et "consommer" en ôtant de leurs définitions les opérations relatives à la manipulation de la composante SC. Par exemple, pour l'opération consommer nous aurons:

Avant la simplification:	Après la simplification:
Notation tc=<SP,EA,SC>	Notation tc=<SP,EA>
consommer(tc:TC)tc':TC	consommer(tc:TC)tc':TC
pré: pre-cons(tc)	**pré**: pre-cons(tc)
post: tc'=<SP,EA',SC'>	**post**: tc'=<SP,EA'>
avec	**avec**
EA'=post-cons(tc),SC'=ajouter(SC,val-cons(tc))	EA'=post-cons(tc)

Nous présentons formellement la règle de transformation utilisée dans cette étape au *chapitre 4*. Nous montrons aussi que cette transformation est une représentation faible.

Remarque

La suite produite notée SP, de même que la suite consommée notée SC jouent le rôle de composantes d'histoire dans la spécification des types de communication. Nous retrouvons leurs équivalents dans [OWI 79] sous le nom de variables auxiliaires ou encore variables d'histoire dans [HOW 76], où elles sont utilisées seulement pour faire des preuves. Toutefois, dans certains types de communication, en plus de l'aspect preuve [LAZ 89] ces variables sont utilisées en mode consultation pour spécifier les opérations caractéristiques. Autrement, elles seront supprimées dans l'étape de simplification.

Construction d'une représentation concrète

Une telle étape prend en entrée le type résultat de la simplification et rend en sortie une représentation de ce type définie par une structure de données et des fonctions que l'on peut traduire facilement en Ada. Cette étape est guidée par deux stratégies fondées sur la forme des types de communication:

- une stratégie de décomposition qui consiste à définir la représentation concrète d'un type en terme de celle de ses composantes. Cette stratégie est fondée sur le fait qu'un type de communication est défini par un produit cartésien.
- une stratégie ascendante qui consiste à définir la représentation d'une opération à partir des opérations qui la composent. Cette deuxième stratégie est fondée sur le fait que les opérations d'un type sont construites à partir des opérations caractéristiques, qui sont elles mêmes définies à partir des opérations des types Suite et Table.

Nous montrons l'application de ces deux stratégies sur le type TC.

Stratégie de décomposition

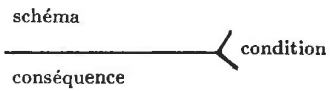
Cette stratégie a pour conséquence de décomposer la représentation d'un type en d'autant d'étapes qu'il y a de composantes. Par exemple pour le type TC nous aurons les deux étapes suivantes:

1. représentation de la composante SP,
2. représentation de la composante EA.

Ces deux étapes sont guidées par les propriétés des opérations fournies par l'analyse de la spécification. Nous examinons maintenant le déroulement de chaque étape.

1. Représentation de la composante SP

L'analyse de la spécification du type TC fait état de trois opérations : les constructeurs "creer", "ajouter" et l'opération externe "meme". L'absence de simplificateurs (les opérations qui suppriment des éléments) parmi ces opérations fait que la taille de cette composante croit indéfiniment. Cette remarque est valable pour les composantes d'histoire de tous les types de communication. Par conséquent, l'élaboration d'une représentation généralisée (conservant toutes les propriétés) a pour inconvénient de conduire à une structure de données de taille infinie. D'un point de vue pratique, ce qui importe dans un objet est son "comportement" vis-à-vis des opérations externes [FIN 79]. La représentation à construire ne doit alors conserver que les propriétés de ces opérations. Cette idée constitue la clé de voûte de la démarche que nous développons au paragraphe 3 du *chapitre 4* pour représenter les composantes d'histoire. Les règles de représentation qui en constituent le fondement pourraient se présenter sous la forme suivante:



et se lit : lorsqu'une opération externe donnée s'unifie avec le schéma d'une règle et que la condition est vérifiée, alors la conséquence est instanciée, produisant ainsi une représentation de la composante considérée.

C'est la règle II (page 92) que nous utilisons pour représenter la composante SP du type TC. L'instantiation de cette règle produit la représentation suivante:

type REP(ELT)=<valmeme:Entier,dernier:ELT, vide:booléen> **opérations** repcreer : → REP repajouter : REP × ELT → REP repmeme : REP × ELT → entier **notation**: reps:REP, reps=<valmeme,dernier,vide> repcreer()=<0,indef,vrai> repajouter(reps,e)=<repmeme(reps,e),e,faux> repmeme(reps,e)= **si** reps.vide **alors** 0 **sinon** **si** reps.dernier=e **alors** reps.valmeme+1 **sinon** 0 **fsi** **fsi**	**Commentaires** : -repop est l'opération du type REP qui représente l'opération de nom op du type Suite -dernier représente le dernier élément ajouté -vide est un champ qui permet de tester si un objet de type REP est vide -valmeme est un entier qui permet de calculer l'opération meme par récurrence, à chaque adjonction d'un élément

Cette règle représente la composante SP par un objet de type REP. Nous montrons au *chapitre 4*, qu'il s'agit ici d'une représentation faible du type Suite (type de la composante SP) restreint aux trois opérations *creer*, *ajouter* et *meme* par le type REP.

2. Représentation de la composante EA

L'analyse de la spécification fait état de cinq opérations:

- les constructeurs t-creer et t-ajouter,
- les opérations externes t-vide et extraire,
- le simplificateur retirer.

Contrairement aux composantes d'histoire, remarquons la présence d'un simplificateur parmi ces opérations. C'est une remarque valable pour tous les types de communication. Cette présence apporte un complément d'information permettant de nous orienter vers une représentation par des structures de données connues telles que les piles, les files, etc. La démarche que nous adoptons dans ce cas est fondée sur la réutilisation de représentations prédéfinies. Elle s'appuie sur l'utilisation d'une bibliothèque de spécifications et d'implantations en Ada de types abstraits de données. Chaque type est muni d'un ensemble de propriétés qui, lorsqu'elles sont vérifiées par les opérations appliquées sur l'ensemble consommable, permettent de choisir le type comme représentation.

Par exemple, les opérations de la composante EA du type TC vérifient les propriétés du type File que l'on peut énoncer intuitivement de la manière suivante :
Une table t (en l'occurrence la composante EA) est représentée par une file, si les constructeurs se font à l'entrée la plus grande de t notée max(t), et les opérations d'accès, les simplificateurs ont lieu à l'entrée la plus petite notée min(t).

Nous représentons donc la composante EA par un objet de type File. Le tableau suivant présente sur la même ligne l'opération appliquée sur la composante EA et l'opération qui la représente dans le type File.

t-creer	creer : → File, creé une file vide
t-ajouter	enfiler : File × ELT → File, ajoute un élément en queue de la file
t-vide	vide : File → booléen, teste si une file est vide
extraire	tete : File → ELT, retourne l'élément de tete d'une file
retirer	defiler : File → File, supprime l'élément de tete d'une file.

La représentation des composantes étant faite, on déduit facilement celle du type de communication grâce à la stratégie de décomposition. En notant REP-TC la représentation du type TC, nous aurons:

type REP-TC(ELT) = <SP:REP(ELT),EA:FILE(ELT) >

Il reste maintenant à définir les opérations du type de communication sur la représentation ainsi construite. Pour cela nous appliquons la stratégie ascendante.

Application de la stratégie ascendante

Cette stratégie stipule que la représentation des opérations d'un type se définit automatiquement à partir de celle des opérations appliquées sur ses composantes. Il suffit de remplacer dans le type de communication ces dernières opérations par leurs représentations. Par exemple pour le type TC, les opérations caractéristiques seront représentées par des opérations du type REP-TC de même nom :

pre-prod(<SP,EA>,e)=repmeme(SP,e) < p
pre-cons(<SP,EA>)=non vide(EA)
val-cons(<SP,EA>)=tete(EA)
post-cons(<SP,EA>)=defiler(EA)

Remarquons qu'on a seulement remplacé dans les opérations caractéristiques, les opérations appliquées sur les composantes SP et EA (meme, t-vide, etc) par leurs représentations (repmeme, vide, etc). Les opérations "init", "produire" et "consommer" seront représentées par des opérations de même nom, en suivant le même principe utilisé pour les opérations caractéristiques:

type REP-TC(ELT)=<SP:REP(ELT),EA:FILE(ELT)>
  notation : tc=<SP,EA>
  opérations :
    INIT () tc : REP-TC
      pre: vrai
      post: tc=<repcreer(),creer()>
    PRODUIRE (tc : REP-TC, e : ELT) tc' : REP-TC
      pre: vrai
      post: tc'=<SP',EA'>
        avec SP'=repajouter(SP,e)
        EA'= **si** pre-prod(tc,e) **alors** enfiler(EA,e)
            sinon EA **fsi**
    CONSOMMER(tc : REP-TC) tc' : REP-TC
      pre: pre-cons(tc)
      post: tc'=<SP,EA'>
        avec EA'=defiler(EA)

Les deux stratégies que nous venons de présenter ramènent le problème de la représentation d'un type de communication à celui de la représentation de ses composantes. C'est donc à ce dernier problème que nous consacrons une grande partie du chapitre suivant.

Remarquons que la représentation ainsi construite est exprimée dans un style applicatif. C'est l'étape de passage au niveau dynamique qui introduira les règles pour traduire cette représentation en Ada (style impératif), tout en tenant compte des caractéristiques de l'environnement parallèle des types de communication.

Chapitre 4

Etude de la représentation des types de communication

1 Introduction

Nous avons introduit et justifié les étapes qui composent le processus de transformation des types de communication en programmes Ada. La transition entre ces étapes est assurée par des règles de transformation, que nous avons présentées de manière informelle.

L'objet de ce chapitre est de spécifier les règles qui constituent le fondement de l'étape de représentation. Cette spécification va nous conduire à présenter la preuve de correction de telles règles.

Ainsi au paragraphe 2, nous présentons la transformation de simplification. Puis nous décrivons les approches et les règles de transformation que nous avons développées pour représenter les composantes d'histoire &3 et l'ensemble consommable &4.

2 Transformation de simplification

La transformation de type que nous allons introduire permet de simplifier la structure d'un type tout en préservant les propriétés de ses opérations caractéristiques. Le type résultat de la transformation doit exprimer la même stratégie de communication que le type de départ. De manière intuitive, simplifier un type de communication consiste à éliminer de sa structure la ou les composantes dont la valeur n'est pas nécessaire à la définition des opérations caractéristiques.

Nous allons illustrer par un exemple une telle transformation avant d'en donner une définition formelle.

2.1 Exemple

Considérons le type de communication noté TC exprimant la stratégie de communication suivante : toute valeur reçue est la dernière émise, à la condition de former une suite décroissante de valeurs de type ELT. Pour faciliter la compréhension de la transformation, nous considérons la spécification complète du type TC et non seulement celle de ses opérations caractéristiques.

```
type TC (ELT)=<SP:Suite(ELT),EA:Table(ELT),SC:Suite(ELT)>
  notation : tc=<SP,EA,SC>
  opérations :
    init() tc : TC
      pre : vrai
      post : tc=<creer(),t-creer(), creer()>
    produire (tc : TC, e : ELT) tc' : TC
      pre : vrai
      post : tc'=<SP',EA',SC>
        avec SP'=ajouter(SP,e)
        EA'= si pre-prod(tc,e) alors t-ajouter(EA.long(SP)+1,e)
          sinon EA fsi
    consommer (tc : TC) tc' : TC
      pre : pre-cons(tc)
      post : tc'=<SP,EA',SC'>
        avec EA'=post-cons(tc)
          SC'=ajouter(SC,val-cons(tc))
    pre-prod (tc : TC, e : ELT) b : booleen
      pre : vrai
      post : b = inf(SP,e)
    pre-cons (tc : TC) b : booleen
      pre : vrai
      post : b = non t-vide(EA)
    val-cons (tc : TC) e : ELT
      pre : vrai
      post : e = t-dernier(EA)
    post-cons (tc : TC) t : table
      pre : vrai
      post : t = supder(EA)
fin TC
```


Notons l'introduction de l'opération inf qui porte sur le type Suite et constitue un enrichissement de celui-ci :

$$\text{inf} : \text{Suite} \times \text{ELT} \rightarrow \text{booleen}$$

Etant donnés une suite s et un élément e, l'opération "inf" vérifie que e est plus petit que tous les éléments de s :

```
inf(s,e): si vide(s) alors vrai
                sinon si dernier(s) > e alors inf(debut(s),e)
                                        sinon faux
                      fsi
          fsi
```

La composante SC du type TC n'intervient pas dans la spécification des opérations caractéristiques. Les seules opérations qui lui sont appliquées sont les constructeurs **creer** et **ajouter.** Ainsi on ne fait que construire une telle composante sans accéder à sa valeur. On dira que la composante SC introduit une redondance dans le type TC. La transformation de simplification permet d'éliminer cette redondance en transformant le type TC en un autre type noté TCS tel que TCS=<SP:Suite(ELT),EA:Table(ELT)>. Cette transformation est une application que nous notons ρ et qui associe à tout objet tc=<SP,EA,SC> de TC l'objet tcs=<SP,EA> de type TCS.

Notons tout de suite que ρ est l'identité sur les types constituant l'univers de la spécification d'un type de communication :

ρ(Suite)=Suite
ρ(Table)=Table
ρ(Booléen)=Booleen
ρ(ELT)=ELT
ρ(Entier)=Entier

L'application ρ associe aux opérations sur le type TC, des opérations sur le type TCS de même nom dans lesquelles la composante SC et les opérations associées sont ignorées. Nous présentons ci-dessous la spécification du type TCS.

```
type TCS (ELT)=<SP:Suite(ELT),EA:Table(ELT)>
  notation : tcs=<SP,EA>
  opérations :
    init() tcs : TCS
      pre : vrai
      post : tcs=<creer(),t-creer()>
    produire (tcs : TCS, e : ELT) tcs' : TCS
      pre : vrai
      post : tcs'=<SP',EA'>
        avec SP'=ajouter(SP,e)
        EA'= si pre-prod(tcs,e) alors t-ajouter(EA,long(SP)+1,e)
          sinon EA fsi
    consommer (tcs : TCS) tcs' : TCS
      pre : pre-cons(tc)
      post : tcs'=<SP,EA'>
        avec EA'=post-cons(tcs)
    pre-prod (tcs : TCS, e : ELT) b : booleen
      pre : vrai
      post : b = inf(SP,e)
    pre-cons (tcs : TCS) b : booléen
      pre : vrai
      post : b = non t-vide(EA)
    val-cons (tcs : TCS) e : ELT
      pre : vrai
      post : e = t-dernier(EA)
    post-cons (tcs : TCS) t : table
      pre : vrai
      post : t = supder(EA)
fin TCS
```

Propriété :

L'application ρ est une représentation faible du type TC par le type TCS.

Preuve :

Il suffit de montrer que les propriétés des opérations externes définies sur le type TC sont conservées par ρ. Cette preuve est évidente puisque ρ est l'identité sur les opérations caractéristiques (les opérations externes d'un type de communication). Prenons par

exemple le cas de l'opération pre-cons et établissons la preuve.

Soit tc:TC, avec tc=<SP,EA,SC>. Nous avons pre-cons(tc)=non t-vide(EA) et ρ(tc)=<SP,EA>.

ρ(pre-cons(tc)) = ρ(non t-vide(EA))
= non t-vide(EA), car ρ est l'identité sur le type Table
=ρ(pre-cons)(ρ(tc))

Le fait que ρ soit une représentation faible nous assure que le type TCS définit la même stratégie de communication que le type TC. Nous allons définir la règle de simplification comme une représentation faible d'un type par un autre.

2.2 Règle de simplification

Soit T un type abstrait de données défini par un produit cartésien de types $T_1, \ldots, T_n$ relativement aux fonctions de projection $t_1, \ldots, t_n$ que l'on note: T=< t_1 : $T_1, \ldots, t_n$: T_n >.

Nous définissons les ensembles suivants :
Σ^i_T : ensemble des opérations définies sur le type T_i et utilisées dans la spécification de T,
$\Sigma^e_{T_i}$: ensemble des opérations externes définies sur le type T_i.
Si $\Sigma^i_T \cap \Sigma^e_{T_i} = \emptyset$ **alors** l'application ρ définie par :

- ρ(T)=TS avec TS=< $t_1 : T_1, \ldots, t_{i-1} : T_{i-1}, t_{i+1} : T_{i+1}, \ldots, t_n : T_n$ >, et pour tout type externe T_e, $\rho(T_e)=T_e$,
- pour tout objet t=< $t_1, \ldots, t_n$ > de T, alors ρ(t)=ts avec ts=< $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$ > et ts un objet de type TS,
- pour toute opération op du type T, ρ associe une opération de même nom du type TS telle que :
  - si op est une opération externe définie par op(t)=t_e avec t un objet de type T et t_e un objet d'un type externe alors ρ(op(t))=t_e. ρ est l'identité sur les opérations externes.
  - si op est un constructeur du type T définie par op(t)=t_c avec t=< $t_1, \ldots, t_n$ > et t_c=< $op_1(t_1), \ldots, op_n(t_n)$ > deux objets de type T, et op_i est un constructeur sur le type T_i alors ρ(op(t))=< $op_1(t_1), \ldots, op_{i-1}(t_{i-1}), op_{i+1}(t_{i+1}), \ldots, op_n(t_n)$ >.

est une simplification du type T par le type TS.

Propriété

Une simplification est une représentation faible du type T par le type TS.

Preuve

Il suffit de montrer que les propriétés des opérations externes du type T sont conservées par l'application ρ. Or nous avons pour toute opération externe op : ρ(op(t))=op(t) avec t:T. L'application ρ est donc l'identité sur les opérations externes, ce qui assure la conservation de leurs propriétés.

Dans le cadre de ce travail, T est un type de communication, T_1=Suite, T_2=Table, T_3=Suite. Pour l'exemple que nous avons présenté précédemment, nous avons :

- la composante SP : Σ^1_{TC}={ajouter,creer,inf} et $\Sigma^1_{TC} \bigcap \Sigma^e_{Suite} = \{inf\}$
- la composante EA : Σ^2_{TC}={t-ajouter,t-creer,t-vide,t-dernier,supder} et $\Sigma^2_{TC} \bigcap \Sigma^2_{Table} = \{t-vide, t-dernier\}$
- la composante SC : Σ^3_{TC}={ajouter,creer} et $\Sigma^3_{TC} \bigcap \Sigma^e_{Suite} = \emptyset$

Nous avons donc représenté le type TC=<SP:Suite(ELT),EA:Table(ELT),SC:Suite(ELT)> par le type TCS=<SP:Suite(ELT),EA:Table(ELT)>. Le fait que la transformation soit une représentation faible nous assure l'équivalence de comportement des deux types vis-à-vis des opérations externes (les opérations caractéristiques). Il en résulte que le type résultat de la transformation définit la même stratégie de communication que le type initial.

Cette règle trouve son équivalent dans [WIL 81] sous le nom de variables mortes et est utilisée pour optimiser les programmes. Cette optimisation repose sur l'élimination des variables dont les valeurs ne sont pas exploitées dans les programmes. Certains analyseurs sémantiques de langages de programmation utilisent aussi, sous une autre forme, la règle de simplification pour signaler les variables utilisées en mode "construction" (partie gauche d'une affectation) et non en mode "accès" (partie droite d'une affectation). Dans ce cas, on peut alors considérer l'affectation comme un constructeur, et l'accès à la valeur d'une variable comme une opération externe.

3 Représentation des composantes d'histoire

Comme nous l'avons signalé, la représentation de ces variables est sans intérêt si on ne dispose pas de fonction d'accès à leur valeur. Ceci est exprimé dans la règle précédente. Autrement les opérations appliquées sur ces objets ne peuvent être que les constructeurs primitifs du type Suite (ajouter, creer) et les opérations externes. Si on essaie de construire une représentation généralisée de ces objets, on aboutira à une structure de données de taille infinie. Aussi nous étudions les représentations comme moyen d'optimiser la structure de ces objets en ne conservant que leurs propriétés observables.

Nous retrouvons encore ici la notion de représentation faible. Ces remarques nous amènent à adopter la stratégie suivante :

Simplifier la structure de ces objets, en ne représentant que la partie nécessaire à la définition des opérations externes qui leurs sont appliquées.

Cette stratégie constitue la clé de voûte de la démarche que nous illustrons par un exemple avant d'en expliciter les principes.

3.1 Exemple

Considérons de nouveau le type de communication décroissante présenté au paragraphe précédent. Nous supposons que ce type a été transformé par la règle de simplification produisant ainsi un nouveau type que nous avons noté TCS. Rappelons que les objets de ce type sont définis par un produit cartésien d'une suite et d'une table : la suite produite notée SP et l'ensemble consommable noté EA. Nous nous intéressons ici à la représentation de la composante SP de ces objets. Pour cela nous nous appuyons sur les propriétés des opérations qui lui sont appliquées dans la spécification et en particulier, les opérations externes. L'analyse du type TCS fait état de trois opérations :

- les constructeurs **creer** et **ajouter** du type Suite,
- l'opération externe **inf** définie aussi sur le type Suite.

La représentation que nous cherchons peut être considérée comme la représentation du type Suite (type de la composante SP) restreint aux trois opérations précédentes.

Intuitivement, la règle que nous utilisons pour construire cette représentation est la suivante. Soit s une suite, pour vérifier qu'un élément e est inférieur à tous les éléments de s, il suffit de vérifier que e est inférieur au plus petit élément de s. En réduisant la table s à son plus petit élément, nous définirons l'opération inf de manière plus simple. Le type REP que nous présentons ci-dessous permet de représenter le type Suite, en s'appuyant sur cette règle.

```
type REP (ELT)=<elem:ELT,vide:booléen>
opérations:
repcreer: → REP
repajouter: REP × ELT → REP
repinf : REP × ELT → booléen
soit reps:REP, e:ELT
notation: reps=<elem,vide>
repcreer() = <indef,vrai>
repajouter(reps,e) = si reps.vide ou reps.elem>e1 alors
<e1,faux> sinon <reps.elem,faux> fsi
repinf(reps,e1)= si reps.vide alors vrai
                 sinon si reps.elem > e alors vrai
                                       sinon faux
                 fsi
          fsi
```

le type REP est un produit cartésien dont les objets sont des couples composés :

- d'un booléen qui indique si un objet est vide,
- d'un élément de type ELT qui représente intuitivement le plus petit élément d'une table.

Nous notons ρ la représentation définie par :

ρ(Suite)=REP
ρ(creer)=repcreer
ρ(ajouter)=repajouter
ρ(inf)=repinf

Pour prouver la correction de cette représentation, il suffit de montrer que les propriétés de l'opération inf sont conservées. Autrement dit, pour tout objet s de type Suite avec ρ(s)=reps, il faut montrer que la propriété inf(s,e)=repinf(reps,e) est toujours vérifiée. Cette preuve sera présentée dans la règle III du paragraphe 3.3.

Nous présentons maintenant de manière plus précise la démarche utilisée pour représenter les composantes d'histoire.

3.2 Démarche

Partant des remarques précédentes, nous avons adopté une démarche que l'on peut qualifier de synthèse de représentation. Celle-ci est fondée sur les propriétés des opérations externes appliquées sur une composante d'histoire dans la spécification. Cette démarche comprend deux étapes :

Soit X une composante d'histoire, $f_1, f_2, \ldots, f_n$ les opérations externes appliquées sur X dans la spécification d'un type de communication.

- Une étape dite de **transformation** qui permet de définir pour chaque opération f_i appliquée sur X :
  - une représentation de X par une structure de données ne contenant que les informations nécessaires et suffisantes pour implanter l'opération f_i,
  - une définition de l'opération f_i et des constructeurs primitifs (creer et ajouter) sur cette représentation que nous notons repf_i, *repcreer*$_i$ et *repajouter*$_i$.
- une étape dite de **composition** qui construit la représentation finale de X par "regroupement" des structures de données et définitions engendrées pour chaque opération f_i lors de l'étape précédente.

Etablissons tout de suite le lien avec l'exemple que nous avons développé précédemment:

- X est la composante SP du type de communication TCS. La seule opération externe appliquée sur cette composante est l'opération inf,
- l'étape de transformation a donné lieu à la représentation REP et à la définition des opérations creer, ajouter et inf sur cette représentation,
- Comme la composante SP est manipulée par une seule opération externe, la structure de données REP est la représentation finale de cette composante : l'étape de composition n'est pas nécessaire.

Nous allons maintenant expliciter chacune de ces deux étapes à travers les règles qu'elles utilisent.

3.3 Etape de transformation

Cette étape utilise un ensemble de règles de transformation correspondant à des classes particulières d'opérations. Ces classes se différencient entre elles par la forme des opérations externes utilisées dans la spécification. Lorsqu'une opération externe donnée f appartient à une classe, la règle de transformation associée à cette classe est appliquée pour construire une représentation de la composante d'histoire et une définition de l'opération f ainsi que les constructeurs primitifs sur cette représentation. Une règle de transformation est définie par :

- un schéma d'opérations qui caractérise la classe des opérations admissibles. Le langage d'expression des schémas est fonctionnel. Il est constitué de la conditionnelle,

des symboles de fonctions prédéfinies sur le type Suite et les symboles de fonctions définies par l'utilisateur.

- une condition d'applicabilité qui a pour rôle de mieux préciser les particularités du schéma d'opérations et qui restreint par conséquent le domaine de définition de la règle. Elle porte essentiellement sur les objets constituant le schéma.
- une conséquence qui décrit le résultat de l'application de la règle de transformation sur une opération f admissible (appartenant à la classe d'opérations spécifiée par le schéma et la condition d'applicabilité) :
  - représentation de la composante d'histoire argument de l'opération par une structure de données. Nous notons **REP** le type abstrait qui définit cette structure.
  - définition de l'opération f et des constructeurs sur cette représentation que nous notons respectivement **repf**, **repcreer** et **repajouter**.

Présentée ainsi, une règle de transformation n'est autre que la représentation du type Suite restreint à l'opération f et aux constructeurs primitifs par le type **REP**.

Le domaine de définition d'une règle, e.g. l'ensemble des opérations sur lesquelles la règle peut être appliquée, est défini par le schéma d'opérations et la condition d'applicabilité. Nous distinguons parmi ces règles celles relatives à certaines opérations prédéfinies sur le type Suite. Les schémas associés se réduisent à la définition d'une seule opération. De telles règles seront dites locales, contrairement aux règles globales où les schémas spécifient des classes d'opérations.

Rappelons que l'objectif d'une règle de transformation est de trouver une représentation des composantes d'histoire en fonction des opérations externes qui leurs sont appliquées dans la spécification des opérations caractéristiques d'un type de communication. Nous avons précisé les raisons de ce choix. Il en résulte que la représentation d'une composante est la composition des conséquences des transformations appliquées sur les opérations prises séparément.

Règles globales

REGLE I :

Considérons l'opération "card" qui calcule le nombre d'éléments d'une suite donnée :

card(s) = **si** vide(s) **alors** 0
sinon card(debut(s)) + 1
fsi

Nous cherchons à construire une représentation REP du type Suite permettant de conserver les propriétés de l'opération card. Une solution commode serait que le cardinal d'une suite figure dans sa représentation. Pour cela nous introduisons dans la représentation REP un objet de type entier qui est incrémenté à chaque adjonction d'un nouvel élément. L'opération card va être représentée par l'accès à la valeur de cet objet. Nous définissons le type REP de la manière suivante :

type REP (ELT)=<valcard:Entier>
opérations:
repcreer : → REP
repajouter : REP × ELT → REP
repcard : REP → entier
notation: reps:REP, reps=<valcard>
repcreer () = <0>
repajouter(reps,e) = <reps.valcard+1>
repcard(reps) = reps.valcard

Remarquons que la définition récursive de l'opération card est transformée en une récurrence sur l'objet valcard exprimée dans l'opération repajouter. En fait nous avons placé dans la représentation le résultat de l' opération qu'on voulait exécuter facilement.

Nous donnons maintenant le schéma général de cette règle.

Schéma:	Profils:
f(s) :	f : Suite → τ,
si vide(s) **alors** a	a : → τ
sinon h(f(debut(s)),b)	h :$\tau \times \tau\prime \rightarrow \tau$
fsi	b : → $\tau\prime$
Condition d'applicabilité: vrai	

Conséquence:
type REP(ELT) = <valf: τ >
opérations:
repcreer : → REP
repajouter : REP × ELT → REP
repf : REP → τ
Soit reps:REP, e:ELT
notation : reps=<valf>
repcreer() = <a>
repajouter(reps,e) = <h(reps.valf,b)>
repf(reps) = reps.valf
fin type

Preuve

La règle de transformation peut être considérée comme une fonction de représentation du type Suite par le type REP. Comme il s'agit ici d'une représentation faible, la preuve

de correction d'une telle règle consiste alors à montrer que la fonction de représentation conserve les propriétés de l'opération f. Autrement dit, les opérations f et repf fournissent le même résultat. Nous devons pour cela montrer que f(s,e)=repf(reps,e) pour tout objet s de type Suite, reps est l'objet de type REP qui représente s. Nous établissons une preuve par induction sur s:

- s=creer, reps=repcreer nous avons:
  f(s)=a, car s=creer et d'après la spécification du type Suite vide(creer)=vrai
  repf(reps)=reps.valf=a, par définition de repcreer et de repf dans la conséquence de la règle. D'où f(s)=repf(reps),

- s=ajouter(s1,e), reps=repajouter(reps1,e)=<h(reps1.valf,b)>
  f(s)=h(f(s1),b), car vide(s)=faux, puisque s=ajouter(s1,e)
  repf(reps)=reps.valf=h(reps1.valf,b), par définition de repf dans la conséquence de la règle.
  Pour que l'égalité f(s)=repf(reps) soit vérifiée, il suffit de montrer que f(s1)=reps1.valf. Or nous avons reps1.valf=repf(reps1) et par hypothèse d'induction f(s1)=repf(reps1), d'où f(s1)=reps1.valf.

Cette preuve montre bien que la règle de transformation conserve les propriétés de l'opération f.

REGLE II

Considérons l'opération "même" définit comme suit :

```
meme(s,e)= si vide(s) alors 0
                      sinon
                      si dernier(s)=e alors
                                        meme(debut(s),e)+1
                                        sinon 0
                      fsi
           fsi
```

En partant du dernier élément d'une suite s, l'opération meme calcule le nombre d'éléments consécutifs de s identiques à un élément donné e.

Intéressons nous à la partie de cette opération qui correspond au cas dernier(s)=e. L'appel récursif interne peut être remplacé par le terme meme(s,dernier(s)) (puisque e=dernier(s)). En introduisant un objet de type entier, nous pouvons calculer meme(s,dernier(s)) par récurrence sur cet objet à chaque adjonction d'un nouvel élément. La connaissance du

dernier élément de la suite s suffit pour réaliser ce calcul. Ainsi nous représentons le type Suite par le type REP donné ci-dessous et dont les objets sont composés de trois champs :

- valmeme est un champ de type entier permettant de transformer la définition récursive de l'opération meme en une récurrence (ainsi que nous l'avons fait dans la règle I). Ce champ représente meme(s,dernier(s)) et sera construit par récurrence à chaque adjonction d'un nouvel élément.

- dernier représente le dernier élément d'une suite. Seul cet élément est nécessaire à la réalisation de l'opération meme.

```
type REP(ELT)=<valmeme:Entier,dernier:ELT,vide:booléen>
opérations:
repcreer : → REP
repajouter : REP × ELT → REP
repmeme : REP × ELT → entier
notation: reps:REP, reps=<valmeme,dernier,vide>
repcreer()=<0,indef,vrai>
repajouter(reps,e)=<repmeme(reps,e),e,faux>
repmeme(reps,e)=
si reps.vide alors 0
            sinon
            si reps.dernier=e alors
                              reps.valmeme+1
                              sinon 0
            fsi
fsi
```

Nous présentons maintenant le schéma de la règle et sa conséquence.

Schéma:	Profils:
f(t,e)= si vide (t) alors a sinon si p(dernier(t),e) alors h(f(debut(t),e),b) sinon c fsi fsi	f : Suite × ELT → τ p : ELT × ELT → booléen h : $\tau \times \tau' \rightarrow \tau$ a : → τ, b : → τ, c : → τ
Condition d'applicabilité: symétrique(p) ∧ transitive(p)	

symétrique(p) et transitive(p) expriment respectivement le fait que p est une relation binaire symétrique et transitive sur le type ELT.

Conséquence:
type REP (ELT) = <valf:τ,dernier:ELT,vide:booléen>
opérations:
repcreer : $\rightarrow$ REP
repajouter : REP $\times$ ELT $\rightarrow$ REP
repf : REP $\times$ ELT $\rightarrow \tau$
Soit reps:REP, e:ELT
notation: reps=<valf,dernier,vide>
repcreer () = <a,indef,vrai>
repajouter(reps,e) = <repf(reps,e),e,faux>
repf(reps,e) = **si** reps.vide **alors** a
sinon si p(reps.dernier,e) **alors** h(reps.valf,b)
sinon c
fsi
fsi

Preuve

Nous montrons que f(s,e)=repf(reps,e) pour tout objet s de type Suite, reps étant l'objet de type REP représentant s. Ainsi que nous l'avons fait pour la règle I, nous établissons une preuve par induction sur s.

- s=creer, reps=repcreer=<a,indef,vrai>
  f(s,e)=a, car vide(s)=vrai
  repf(reps,e)=a,
  D'où f(s,e)=repf(reps,e).

- s=ajouter(s1,e1) , reps=repajouter(reps1,e1)=<repf(reps1,e1),e1,faux> avec reps1 l'objet de type REP qui représente l'objet s1 de type Suite.

  f(s,e)= **si** p(e1,e) **alors** h(f(s1,e),b) , car vide(s)=faux
  sinon c
  fsi

  repf(reps,e)= **si** p(reps.dernier,e) **alors** h(reps.valf,b) , car reps.vide=faux
  sinon c
  fsi

  Nous avons reps.dernier=e1 et reps.valf=repf(reps1,e1), d'où:

  repf(reps,e)= **si** p(e1,e) **alors** h(repf(reps1,e1),b)
  sinon c
  fsi

Pour que l'égalité f(s,e)=repf(reps,e) soit vérifiée, il suffit de montrer que si p(e1,e) alors f(s1,e)=repf(reps1,e1). Or par hypothèse d'induction, nous avons f(s1,e)=repf(reps1,e); par conséquent, il suffit de montrer que si p(e1,e) alors repf(reps1,e)=repf(reps1,e1).
Nous établissons de nouveau une preuve par induction sur reps1 :

- reps1=repcreer, repf(reps1,e)=a et repf(reps1,e1)=a
  d'où repf(reps1,e)=repf(reps1,e1)
- reps1=repajouter(reps2,e2) :

  repf(reps1,e)= **si** p(e2,e) **alors** h(repf(reps2,e),b)
  sinon c
  fsi

  repf(reps1,e1)= **si** p(e2,e1) **alors** h(repf(reps2,e1),b)
  sinon c
  fsi

  Rappelons les hypothèses que nous avons, à savoir :
  p(e1,e) et repf(reps2,e)=repf(reps2,e1) (hypothèse d'induction).
  Pour que repf(reps1,e)=repf(reps1,e1), il suffit que :
  (1) p(e2,e1) $\wedge$ p(e1,e) $\Rightarrow$ p(e2,e)
  et
  (2) p(e2,e) $\wedge$ p(e1,e) $\Rightarrow$ p(e2,e1)
  Si nous exigeons que p soit transitive, alors nous aurons la propriété (1), et si en plus elle est symétrique, nous aurons la propriété (2).

Ce qui justifie les conditions sur p dans la définition de la règle de transformation (transitive(p) et symétrique(p)).

REGLE III

Considérons l'opération "inf" qui vérifie qu' un élément donné e est inférieur à tous les éléments d'une suite donnée :

inf(s,e)=
si vide(s) **alors** vrai
sinonsi dernier(s)>e **alors**
inf(debut(s),e)
sinon faux
fsi
fsi

En réduisant la suite s à son plus petit élément, on peut réaliser l'opération inf(s,e) en comparant tout simplement cet élément à e. Nous représentons le type Suite par le type REP donné ci-dessous et qui associe à toute suite s son plus petit élément noté elem.

type REP (ELT)=<elem:ELT,vide:booléen>
opérations:
repcreer: → REP
repajouter: REP × ELT → REP
repinf : REP × ELT → booléen
soit reps:REP, e:ELT
notation: reps=<elem,vide>
repcreer() = <indef,vrai>
repajouter(reps,e) = **si** reps.vide ou reps.elem>e **alors** <e1,faux>
sinon <reps.elem,faux> **fsi**
repinf(reps,e)=
si reps.vide **alors** vrai
sinonsi reps.elem > e **alors** vrai
sinon faux
fsi
fsi

le type REP est un produit cartésien dont les objets sont des couples composés:

- d'un booléen qui indique si un objet est vide.
- d'un élément de type ELT qui représente intuitivement le plus petit élément d'une suite.

Nous présentons maintenant le cas général.

Schéma:	Profils:
f(s,e): **si** vide(s) **alors** a(e) **sinon si** p(dernier(s),e) **alors** f(debut(s),e) **sinon** b(e) **fsi** **fsi**	f : Suite × ELT → τ a : ELT → τ b : ELT → τ p : ELT × ELT → booléen
Condition d'applicabilité: transitive(p) ∧ transitive(non p)	

transitive exprime le fait qu'un prédicat est une relation transitive sur le type ELT.

Conséquence:
type REP(ELT) = <elem:ELT,vide:booléen >
opérations:
repcreer : → REP
repajouter : REP × ELT → REP
repf : REP × ELT → τ
Soit reps:REP, e:ELT
Notation : reps=<elem,vide>
repcreer () = < vrai,indef >
repajouter (reps,e) =
si reps.vide **alors** <faux,e>
sinonsi P(reps.elem,e) **alors** <faux,e>
sinon <faux,reps.elem>
fsi
fsi
repf(reps,e) =
si reps.vide **alors** a(e)
sinonsi P(reps.elem,e) **alors** a(e)
sinon b(e)
fsi
fsi

Preuve

Il faut prouver que f(s,e)=repf(reps,e) pour tout objet s de type Suite, reps étant l'objet de type REP qui représente s. Par induction sur s, nous avons à examiner les cas suivants :

- s=creer, reps=repcreer=<indef,vrai>
  f(s,e)=a(e), car vide(s)=vrai
  repf(reps,e)=a(e), car reps.vide=vrai
  D'où f(s,e)=repf(reps,e).
- s=ajouter(creer,e1), reps=repajouter(repcreer,e1)=<e1,faux>
  f(s,e)=**si** p(e1,e) **alors** f(creer,e) **sinon** b(e) **fsi**
  repf(reps,e)=**si** p(reps.elem,e) **alors** a(e) **sinon** b(e) **fsi**
  nous avons f(creer,e)=a(e) et reps.elem=e1. En remplacant f(creer,e) par a(e) dans f(s,e) et reps.elem par e1 dans repf(reps,e) nous déduisons que f(s,e)=repf(reps,e).
- s=ajouter(s1,e1), reps=repajouter(reps1,e1)
  f(s,e)=**si** p(e1,e) **alors** f(s1,e) **sinon** b(e) **fsi**
  repf(reps,e)=**si** p(reps.elem) **alors** a(e) **sinon** b(e) **fsi**
  Par hypothèse d'induction, nous avons f(s1,e)=repf(reps1,e). Nous remplacons le terme f(s1,e) par repf(reps1,e) dans f(s,e) et nous obtenons:
  f(s,e)=**si** p(e1,e) et p(reps1.elem,e) **alors** a(e) **sinon** b(e) **fsi**.
  Pour montrer que f(s,e)=repf(reps,e) il suffit alors de montrer que :

(1) p(e1,e) et p(reps1.elem,e) $\Longleftrightarrow$ p(reps.elem,e)
et
(2) non(p(e1,e) et p(reps1.elem,e)) $\Longleftrightarrow$ non p(reps.elem,e).

De la preuve de la première équivalence on peut déduire la seconde (car a $\Longleftrightarrow$ b = non a $\Longleftrightarrow$ non b).

Preuve de (1) :

- p(e1,e) et p(reps1.elem,e) $\Longrightarrow$ p(reps.elem,e).
  calculons reps.elem :
  - **si** p(reps1.elem,e1) **alors** reps.elem=e1.
    Dans ce cas nous avons p(reps.elem,e) car p(e1,e) est vérifiée.
  - **si** non p(reps1.elem,e1) **alors** reps.elem=reps1.elem.
    Dans ce cas nous avons aussi p(reps.elem) car p(reps1.elem,e) est vérifiée.

  Nous déduisons donc la preuve de la première implication.

- p(reps.elem,e) $\Longrightarrow$ p(e1,e) et p(reps1.elem,e).
  Nous distinguons les deux cas suivants pour le calcul de reps.elem :
  - p(reps1.elem,e1), reps.elem=e1 es donc p(e1,e) (puisque par hypothèse nous avons p(reps.elem,e)). Il reste à montrer que p(reps1.elem,e) est vérifiée. Si on exige que p soit transitive nous aurons p(reps1.elm,e) (car p(reps1.elem,e1) et p(e1,e) sont vérifiées).
  - non p(reps1.elem,e1), reps.elem=reps1.elem et donc p(reps1.elem,e) (puisque par hypothèse nous avons p(reps.elem,e)). Il reste à montrer que p(e1,e) est vérifiée. Pour cela nous établissons une preuve par l'absurde. Nous supposons que non p(e1,e) est vérifiée. Si on exige que non p est transitive, nous déduisons que non p(reps1.elem,e) est vérifiée (car non p(reps1.elem,e1) et non p(e1,e)) ce qui est en contradiction avec l' hypothèse que nous avons à savoir p(reps1.elem,e). Nous déduisons donc p(e1,e).

Par cette preuve, nous concluons que la transformation est une représentation faible du type Suite par le type REP.

Règles locales

Les règles locales définissent les transformations des opérations prédéfinies sur le type Suite. Contrairement aux règles globales, les règles locales ne portent pas sur des classes d'opérations, mais sur une opération bien définie.

- Règle VIDE

  L'opération vide teste la vacuité d'un objet de type Suite.

Définition : vide(s) **si** s=creer() **alors** vrai **sinon** faux **fsi**	Profils : vide : Suite → booleen creer : → Suite
condition d'applicabilité: vrai	

Conséquence:
TYPE REP (ELT) = <vide:booléen >
Opérations :
repcreer : → REP
repajouter : REP × ELT → REP
repvide : REP → booléen
Soit reps : REP, e : ELT
Notation reps=<vide>

repcreer() = <vrai>

repajouter(reps,e) = <faux>

repvide (reps) = reps.vide

- Règle DERNIER

  L'opération dernier extrait la dernière valeur ajoutée à une suite.

Définition : dernier(creer())=*indef* dernier(ajouter(s,e))=e	Profil: dernier : Suite → ELT ∪{*indef*}
condition d'applicabilité:vrai	

Conséquence:
type REP (ELT) = <dernier:ELT>
Opérations :
repcreer : → REP
repajouter : REP × ELT → REP
repdernier : REP → ELT ∪{*indef*}
Soit reps : REP, e : ELT
Notation reps=<dernier>
repcreer() = < *indef* >
repajouter(reps,e) = < e >
repdernier(reps) = reps.dernier

- Règle PREMIER

L'opération premier extrait la plus ancienne valeur ajoutée (en date) à une suite.

Définition :	Profil:
premier(creer())=*indef* premier(ajouter(s,e))=**si** vide(s) **alors** e **sinon** premier(s) **fsi**	premier : Suite → ELT ∪{*indef*}
condition d'applicabilité:vrai	

Conséquence:
type REP (ELT) = <premier:ELT,vide:booléen >
Opérations : repcreer : → REP
repajouter : REP × ELT → REP
reppremier : REP → ELT ∪{*indef*}
Soit rep:REP, e:ELT
Notation : reps=<premier,vide>
repcreer() = <*indef*,faux >
repajouter(reps,e) = **si** reps.vide **alors** < e,faux >
sinon <reps.premier,faux>
fsi
reppremier(reps) = reps.premier

Après transformation, le corps des opérations vide, dernier et premier se réduit à l'accès aux champs de la représentation. Nous avons choisi de supprimer la définition de ces opérations et de remplacer directement leur utilisation par l'accès direct aux champs qui les représentent. Nous les avons gardées ici pour des raisons de conformité de présentation.

- Règle DANS

L'opération dans teste si un entier donné appartient au domaine d'une suite.

Définition :	Profil:
dans (s,i) = **si** vide(s) **alors** faux **sinon si** i=long(s) **alors** vrai **sinon** dans(debut(s),i) **fsi** **fsi**	dans: Suite×entier→booléen
condition d'applicabilité:vrai	

La transformation de cette opération n'introduit pas de nouvelle structure de données, sinon celle engendrée par la transformation de l'opération long conformément à la règle I.

Conséquence:
type REP(ELT) = <vallong:entier,vide:booléen>
opérations :
repcreer : → REP
repajouter : REP × ELT → REP
replong : REP → entier
repdans : REP × entier → booléen
Soit reps:REP, e:ELT, i:entier
Notation : reps=<vallong,vide>
repcreer() = <0,vrai>
repajouter(reps,e) = <replong(reps),vrai>
replong(reps) = **si** reps.vide **alors** 0
sinon reps.vallong + 1
fsi

repdans (reps,i) = **si** reps.vide **alors** faux
sinon si 1 ≤ i ≤ replong(reps) **alors** vrai
sinon faux
fsi
fsi

Nous nous intéressons maintenant à l'étape de composition.

3.4 Etape de composition

Les règles que nous venons de présenter concernent la transformation des opérations appliquées sur une composante d'histoire. Chaque transformation d'une opération f génère une structure de données et une définition des constructeurs primitifs sur cette structure.

Lorsqu'une composante d'histoire est manipulée par plusieurs opérations, l'étape de composition a pour but de regrouper les résultats de leurs transformations. Plus précisément cette étape construit une structure de données REP et les opérations repcreer et repajouter à partir des représentations REP_i et des opérations $repcreer_i$ et $repajouter_i$ générées pour chaque opération f_i. Nous allons illustrer cette étape sur un exemple afin d'en dégager les principes généraux.

Exemple

Soit à représenter la variable d'histoire x sur laquelle sont appliquées les deux opérations "meme" (exemple de la règle II) et "dans" (la règle locale **dans**) dans une spécification de type de communication. Comme nous l'avons vu, la transformation de ces opérations produit séparément les résultats suivants pour les constructeurs :

type REP_1(ELT) = <valmeme:Entier,dernier:ELT,vide: booléen>
Soit reps:REP_1
notation: reps=<valmeme,dernier,vide>
$repcreer_1$() = <0,indef,vrai>
$repajouter_1$(reps,e) = <repmeme(reps,e),e,faux>

type REP_2(ELT) = <vallong:Entier,vide:booléen>
Soit reps:REP_2
notation: reps=<vallong,vide>
$repcreer_2$() = <0,vrai>
$repajouter_2$(reps,e) = <reps.vallong+1,faux>

De ces deux types, nous déduisons deux représentations pour la variable d'histoire x notées respectivement rep_1-x et rep_2-x définies ainsi :

- rep_1-x=<valmeme,dernier,vide> avec rep_1-x:REP_1

- rep_2-x=<vallong,vide> avec rep_2-x:REP_2,

La représentation finale rep-x de la variable x est définie de la manière suivante :

rep-x =**composer**(rep_1-x,rep_2-x)

Où **composer** est un opérateur qui exprime l'union syntaxique de la liste des champs constituant les structures rep_i-x, à savoir :

rep-x=<valmeme,vallong,dernier,vide> avec rep-x:REP et
type REP(ELT)=**composer**(REP_1,REP_2)
=<valmeme:Entier,vallong:Entier,dernier:ELT,vide:booléen>

Remarquons que le champ "vide" appartenant aux deux structures rep_1-x et rep_2-x n'est déclaré qu'une fois dans rep-x. En fait ce champ représente la même information sémantique pour les deux structures. On dira que le champ "vide" est un champ commun, contrairement aux champs "valmeme" et "vallong " que nous appellerons champs propres à chaque structure. Nous présentons maintenant le type REP en donnant la liste de ses opérations :

opérations:
repcreer : → REP
repajouter : REP × ELT → REP
repmeme : REP → entier
repdans : REP × entier → booléen
notation: reps=<valmeme,vallong,dernier,vide>
repcreer() =<0,0,indef,vrai>
repajouter(reps,e)=<repmeme(reps,e),vallong+1,e,faux>
les opérations repmeme, replong et repdans restent inchangées

La définition des constructeurs repcreer et repajouter est réalisée par dépliage des opérations $repcreer_i$ et $repajouter_i$. Cependant, le calcul des champs communs (par exemple le champ vide) n'est fait qu'une seule fois. Nous présentons maintenant les principes de l'étape de composition en définissant de manière plus précise l'opérateur **composer**.

Vers une formalisation de l'opérateur composer

Pour simplifier les notations, nous nous limitons au cas où l'étape de transformation a donné lieu à deux représentations: REP_1 (respectivement REP_2) qui résulte de la transformation de l'opération externe f_1 (respectivement f_2). Rappelons que chaque type REP_i est munit de trois opérations : les deux constructeurs $repcreer_i$, $repajouter_i$ et l'opération externe $repf_i$. Nous notons :

$$\text{type } REP_1 =< p_1^1 : P_1^1, \ldots, p_1^n : P_1^n, c_1 : C_1, \ldots, c_k : C_k >$$
$$\text{type } REP_2 =< p_2^1 : P_2^1, \ldots, p_2^n : P_2^m, c_1 : C_1, \ldots, c_k : C_k >$$

Où les p_i^j dénotent les champs propres au type REP_i, et les c_l dénotent les champs communs à REP_1 et REP_2.

Le type REP=**composer**(REP_1,REP_2) qui est le résultat de l'étape de composition, hérite des propriétés des deux représentations: il possède deux constructeurs *repcreer*, *repajouter* et deux opérations externes $repf_1$, $repf_2$. Nous définissons ces opérations après avoir donné la structure des objets de type REP.

1. structure des objets de type REP:
   Soient rep_1:REP_1, rep_2:REP_2, rep:REP

   rep=**composer**(rep_1,rep_2) $\Longrightarrow$ rep.p_1^j=$rep_1.p_1^j$, pour j=1, ... ,n
   rep.p_2^j=$rep_2.p_2^j$, pour j=1, ... ,m
   rep.c_j=$rep_1.c_j$=$rep_2.c_j$, pour j=1, ... ,k

2. définition des constructeurs du type REP:
   repcreer : $\rightarrow$ REP, repajouter : REP $\times$ ELT $\rightarrow$ REP

   repcreer()=**composer**(rep_1,rep_2),
   avec rep_1=$repcreer_1$() et rep_2=$repcreer_2$()

   repajouter(rep,e)=**composer**(rep_1,rep_2),
   avec rep_1=$repajouter_1$(rep_1',e)
   rep_2=$repajouter_2$(rep_2',e) et
   rep=**composer**(rep_1',rep_2').

3. définition des opérations externes :
   $repf_i$: REP $\times$ X_1 $\rightarrow$ T_1, $repf_2$: REP $\times$ X_2 $\rightarrow$ T_2

   $repf_1$(rep,x_1)=t_1,
   avec t_1=$repf_1$(rep_1,x_1), et rep.p_1^j=$rep_1.p_1^j$, pour j=1, ... ,n,
   et rep.c_j=$rep_1.c_j$, pour j=1, ... ,k,

   $repf_2$(rep,x_2)=t_2,
   avec t_2=$repf_2$(rep_2,x_2), et
   rep.p_2^j=$rep_2.p_2^j$, pour j=1, ... ,m et
   rep.c_j=$rep_2.c_j$, pour j=1, ... ,k.

L'opérateur **composer** construit donc un nouveau type à partir des représentations fournies par l'étape de transformation. Un tel type spécifie alors la représentation finale de la composante d'histoire considérée.

3.5 Remarque

Les schémas que nous venons d'étudier ont la caractéristique commune d'utiliser les composantes d'histoire comme paramètre d'induction. Cependant, il existe d'autres opérations manipulant les variables d'histoire sans que celles-ci ne jouent le rôle de paramètre d'induction.

Voici un exemple de telles opérations :

```
f(s,y) : entier
si vide(s) ou vide(y) alors min(y)
        sinon si premier(y) ≠ dernier(s) alors min(y)
                        sinon f(s,decapite(y))
        fsi
fsi
```

l'opération f calcule le plus petit indice de la suite y dont l'élément associé est différent du dernier élément de la composante d'histoire s.

En vertue de la stratégie ascendante que nous avons énoncé, la transformation de l'opération f consiste en deux étapes:

- transformation de chaque opération externe (vide, dernier),
- application de la règle de composition sur les conséquences de ces transformations.

Ces deux étapes ayant été déjà spécifiées auparavant, nous nous contentons ici de donner le résultat obtenu :

```
Conséquence:
type REP(ELT) = <dernier:ELT,vide:booléen>
opérations:
repcreer : → REP
repajouter : REP × ELT → REP
soit reps:REP, e:ELT
notation: reps=<dernier,vide>
repcreer()=< indef,vrai>
repajouter(reps,e)=<e,faux>
f (reps,y)=
si reps.vide ou vide(y) alors min(y)
                sinon
                si premier(y) ≠ reps.dernier alors min(y)
                                        sinon f(reps,decapite(y))
                fsi
fsi
```

4 Représentation de l'ensemble consommable

Les règles de représentation que nous venons de présenter s'appuient sur la nature des opérations appliquées sur les composantes d'histoire dans la spécification. Ces opérations sont de deux sortes: les constructeurs (creer et ajouter) et les opérations externes. Outre ces opérations, l'ensemble consommable est manipulé par des simplificateurs; les opérations

qui suppriment des éléments dans une table. Aussi, la représentation de cet ensemble doit en plus prendre en compte les caractéristiques de ces nouvelles opérations.

4.1 Démarche

Au lieu d'adopter une démarche de synthèse de représentation comme nous l'avons fait précédemment, nous utilisons ici une démarche fondée sur la réutilisation de représentations prédéfinies. En effet, cette démarche s'appuie sur l'utilisation d'une bibliothèque de spécifications et d'implantations en Ada de types abstraits de données. Chaque type définit une représentation particulière de l'ensemble consommable. Le choix d'un type de la bibliothèque dépend du nombre d'éléments que peut contenir l'ensemble consommable et des propriétés des opérations du type Table appliquées sur cet ensemble dans la spécification. Ces deux caractéristiques nous ont permis de structurer le processus de représentation en plusieurs étapes :

- Etape1 : calcul de la taille du tampon
  En fonction de la taille de l'ensemble consommable, cette étape permet de choisir entre deux structures de données :
  - une structure simple permet de mémoriser une donnée et correspond à un tampon de taille un. L'ensemble consommable ne peut contenir qu'un élément.
  - une structure composée permet de mémoriser une suite de données, et correspond à un tampon de taille n (n est une valeur fixe ou variable). L'ensemble consommable peut contenir n éléments.

  Le calcul de la taille ne peut se faire de manière automatique. C'est une information que doit fournir l'utilisateur au système d'implantation des types de communication. En effet la taille d'un objet est liée à une dynamique d'exécution (et en particulier à la vitesse d'exécution des processus) et ne peut être obtenue par une simple analyse de la spécification.

  La structure simple est définie par un unique composant dans la bibliothèque. Aussi le choix d'une telle structure dans cette étape achève le processus de représentation. En revanche, la structure composée désigne plusieurs composants. L'étape suivante permet d'en choisir un comme représentation du tampon de communication.

- Etape2 : représentation de la structure composée
  La structure composée est définie par plusieurs types abstraits de données en bibliothèque. Chaque type est muni d'un ensemble de propriétés qui lorsqu'elles sont vérifiées par les opérations appliquées sur l'ensemble consommable, le type est choisi comme représentation. Cependant, un type peut posséder plusieurs implantations

sous forme de modules Ada. Par exemple le type pile peut être implanté soit par un tableau (implantation contigue) soit par une liste chaînée (représentation par chaînage). Le choix entre ces deux implantations est fonction de la taille du tampon qui peut être fixe ou variable.

Nous présentons la structure simple comme représentation d'un tampon de taille un, et étudions les règles de choix de représentation pour une structure composée. De telles règles constituent le fondement de la dernière étape.

4.2 Structure simple

La structure simple définit la représentation de l'ensemble consommable par un tampon de taille un. Intuitivement, Cette représentation restreint l'ensemble des entrées d'une table à un singleton; correspondant à la dernière valeur ajoutée. A cette structure est associé un seul module Ada en bibliothèque. La représentation concrète encapsulée dans ce module consiste en un enregistrement composé :

- d'un champ de type ELT qui représente la dernière valeur ajoutée,
- d'un champ de type booléen qui représente l'opération vide,
- d'un champ de type entier qui représente l'opération long.

Nous donnons la description complète de ce module en Annexe.

4.3 Structure composée

La structure composée est définie par plusieurs types abstraits de données en bibliothèque. Chaque type est muni d'un ensemble de propriétés qui lorsqu'elles sont vérifiées par les opérations manipulant l'ensemble consommable, le type est choisi comme représentation. Cependant, un type peut posséder plusieurs représentations sous forme de module Ada. Une deuxième étape doit permettre d'en choisir une. Ces deux étapes correspondent en fait à deux niveaux d'abstraction différents. Dans la première, il s'agit de la représentation du type abstrait Table par un autre type tout en gardant le même niveau d'abstraction. Dans la seconde, il s'agit de la représentation d'un type abstrait en termes de types concrets d'un langage de programmation.

L'analyse d'un certain nombre d'exemples de types de communication nous a amené à retenir quatre types : Pile, File, File à double accès et Table. Chaque type décrit une représentation particulière du type Table relativement aux trois catégories d'opérations. Pour chaque type, nous donnons

- une spécification algébrique,

- la liste des opérations du type Table pouvant être représentées que nous appelons opérations admissibles, suivies par les conditions éventuelles qui restreignent leurs domaines de définition : lorsque ces opérations sont appliquées sur l'ensemble consommable en respectant les conditions associées, le type est alors choisi comme représentation. Par exemple, pour choisir une pile comme représentation d'une table t, il faut que les opérations t-ajouter, retirer et extraire se fassent à l'entrée la plus grande de t notée max(t). Ce que nous notons dans la suite :

  t-ajouter(t,i,e), retirer(t,i), extraire(t,i) : i=max(t).

  Par contre, les autres opérations telles que card, vide peuvent être représentées par des opérations du type pile sans aucune condition.

- la définition de la fonction de représentation que nous notons ρ sur les opérations du type Table.

- le choix de la représentation concrète.

Type File

Une table t est utilisée en file lorsque les constructeurs se font à l'entrée la plus grande de t, et que les opérations d'accès, les simplificateurs ont lieu à l'entrée la plus petite notée min(t).

type File (ELT) **opérations**: creer : → File enfiler : File × ELT → File defiler : File → File tete : File → ELT ∪{*indef*} vide : File → booléen card : File → entier **axiomes**: Soit f:File,e:ELT defiler(enfiler(f,e))=**si** f=creer **alors** f **sinon** enfiler(defiler(f),e) **fsi** tete(creer)=*indef* tete(enfiler(f,e))=**si** f=creer **alors** e **sinon** tete(f) **fsi** vide(creer)=vrai vide(enfiler(f,e))=faux card(creer)=0 card(enfiler(f,e))=card(f)+1 **fin** file	**opérations admissibles**: t-creer, t-ajouter, retirer, decapite, extraire, t-premier, t-vide, card. **conditions**: t-ajouter(t,i,e) : i=max(t) extraire(t,i), retirer(t,i) : i=min(t) **définition de la représentation**: ρ(t-creer())=creer() ρ(t-ajouter(t,i,e))=enfiler(ρ(t),e) ρ(retirer(t,i))=ρ(decapite(t))=defiler(ρ(t)) ρ(extraire(t,i))=ρ(t-premier(t))=tete(ρ(t)) ρ(t-vide(t))=vide(ρ(t)) ρ(card(t))=card(ρ(t))

Représentations concrètes: File_b, File_v

- File_b pour un tampon de taille variable : un tableau où sont rangés les éléments de la file, et deux variables contenant respectivement l'indice du premier et du dernier élément,

- File_v pour un tampon de taille variable : les éléments de la file sont chaînés entre eux et deux pointeurs contenant respectivement l'adresse du premier et du dernier élément.

Nous prouvons que l'application ρ est une représentation faible du type Table par le type File, sous les conditions énoncées. Pour cela, il faut montrer que les propriétés des opérations externes du type Table sont conservées par ρ. Nous le ferons pour les deux opérations t-vide et t-premier, le même principe s'applique aux autres opérations.

Pour tout t:Table, rept:File avec ρ(t)=rept, nous devons montrer que les deux propriétés P1 et P2 suivantes sont vérifiées :

P1: t-vide(t)=vide(rept), P2: t-premier(t)=tete(rept).

Nous établissons une preuve par induction sur t.

- t=t-creer, rept=creer, nous avons par définition :
  t-vide(t)=vide(rept)=vrai et t-premier(t)=tete(rept)=*indef*. D'où P1 et P2.

- t=t-ajouter(t1,max(t1)+1,e1), rept=enfiler(rept1,e1), avec ρ(t1)=rept1
  t-vide(t1)=vide(rept1), et t-premier(t1)=tete(rept1)(hypothèse d'induction).
  t-vide(t)=vide(rept)=faux, d'où P1.
  t-premier(t)=**si** t1=t-creer **alors** e1 **sinon** t-premier(t1) **fsi**
  tete(rept)=**si** rept1=creer **alors** e1 **sinon** tete(rept1) **fsi**.
  on établit une preuve par cas sur t1:

  - t1=t-creer, donc rept1=creer et t-premier(t)=tete(rept)=e1. D'où P2.
  - t1≠t-creer, donc rept1≠creer, t-premier(t)=t-premier(t1), tete(rept)=tete(rept1).
    Or par hypothèse d'induction t-premier(t1)=tete(rept1). D'où t-premier(t)=tete(rept) et donc P2.

Les mêmes preuves peuvent être établies pour les représentations que nous donnons dans la suite et selon le même principe. Aussi, nous ne les présentons pas.

Type Pile

Une table t est utilisée en pile lorsque les trois catégories d'opérations se font à l'entrée la plus grande de la table notée max(t).

type Pile (ELT) **opérations**: creer : → Pile empiler : Pile × ELT → Pile depiler : Pile → Pile sommet : Pile → ELT ∪{*indef*} vide : Pile → booléen card : Pile → entier **axiomes** : Soient p:Pile, e:ELT depiler(empiler(p,e))=p sommet(creer)=*indef* sommet(empiler(p,e))=e vide(creer)=vrai vide(empiler(p,e))=faux card(creer)=0 card(empiler(p,e))=card(p) + 1 **fin** pile	**opérations admissibles**: t-creer, t-ajouter, retirer, supder, extraire, t-dernier, t-vide, card. **conditions**: t-ajouter(t,i,e), retirer(t,i), extraire(t,i): i=max(t) **définition de la représentation**: ρ(t-creer())=creer() ρ(t-ajouter(t,i,e))=empiler(ρ(t),e) ρ(retirer(t,i))=ρ(supder(t))=depiler(ρ(t)) ρ(extraire(t,i))=ρ(t-dernier(t))=sommet(ρ(t)) ρ(t-vide(t))=vide(ρ(t)) ρ(card(t))=card(ρ(t))

Représentations concrètes : Pile_b, Pile_v

- Pile_b pour un tampon de taille bornée : un tableau où sont rangés les éléments de la pile, et une variable contenant l'indice du sommet,
- Pile_v pour un tampon de taille variable : les éléments de la pile sont chaînés entre eux, et un pointeur contenant l'adresse du sommet.

Type File à double accès

Une table t est utilisée en file à double accès lorsque les trois catégories d'opérations se font à l'entrée la plus grande ou la plus petite de t.

type File-double (ELT) **opérations**: creer : → File-double ajouter : File-double × ELT → File-double retirert,retirerq : File-double → File-double extrairet,extraireq : File-double → ELT ∪{*indef*} vide : File-double → booléen card : File-double → entier **axiomes**: Soit fd:File-double,e:ELT retirert(ajouter(fd,e))=**si** fd=creer **alors** fd **sinon** ajouter(retiret(fd),e) **fsi** retirerq(ajouter(fd,e))=fd extrairet(creer)=*indef* extrairet(ajouter(fd,e))=**si** fd=creer **alors** e **sinon** extrairet(fd) **fsi** extraireq(creer)=*indef* extraireq(ajouter(fd,e))=**si** fd=creer **alors** e **sinon** e **fsi** vide(creer)=vrai vide(ajouter(fd,e))=faux card(creer)=0 card(ajouter(fd,e))=card(fd)+1 **fin** File-double	**opérations admissibles**: t-creer, t-ajouter, retirer, decapite, supder extraire, t-premier, t-dernier, t-vide, card **conditions**: ajouter(t,i,e) : i=max(t) extraire(t,i), retirer(t,i) : i=min(t) **ou** i=max(t) **définition de la représentations**: ρ(t-creer())=creer() ρ(t-ajouter(t,max(t),e))=ajouter(ρ(t),e) ρ(retirer(t,min(t)))=ρ(decapite(t)) =retirert(ρ(t)) ρ(retirer(t,max(t)))=ρ(supder(t)) =retirerq(ρ(t)) ρ(extraire(t,min(t)))=ρ(t-premier(t)) =extrairet(ρ(t)) ρ(extraire(t,max(t)))=ρ(t-dernier(t)) =extraireq(ρ(t)) ρ(t-vide(t))=vide(ρ(t)) ρ(card(t))=card(ρ(t))

Représentations concrètes : Filed_b, Filed_v

- Filed_b pour un tampon de taille bornée identique à celle du type file,
- Filed_v pour un tampon de taille variable : double chaînage des éléments et deux pointeurs pour repérer respectivement le premier et le dernier élément.

Type Table

Lorsqu'aucun des types présentés précédemment ne vérifie les règles de choix associées, la représentation choisie est le type Table considéré comme représentation par défaut.

Ce type n'impose en effet aucune restriction en ce qui concerne les opérations utilisées dans la spécification : la fonction de représentation est l'identité. Le seul problème ici est de choisir la représentation concrète la plus adéquate, permettant d'implanter de manière efficace les trois catégories d'opérations. Rappelons quelques caractéristiques communes à tous les types de communication, et qui doivent être prises en considération pour le choix d'une implantation :

- les indicatifs de la table EA (ensemble consommable) sont des entiers,

- l'adjonction d'un élément se fait toujours à l'entrée la plus grande de la table (voir l'opération produire).

- contrairement à l'opération précédente, la suppression et l'accès peuvent se faire à une entrée quelconque de la table EA.

Nous proposons une représentation qui partage la table EA (dite table principale) en sous-table, dans lesquelles on procède par un adressage associatif. La table principale est représentée de manière contigue : à l'indice k, on trouve la sous-table regroupant les entrées i telles que le reste de la division de i par l (l est la taille de la table principale) est k. Notons qu'avec ce partage, les ensembles d'entrées des diverses sous-table sont disjoints. Ceci permet de minimiser le temps d'accès moyen dans la table principale. Par ailleurs, les éléments d'une sous-table sont triés par construction sur les entrées : c'est le résultat des adjonctions qui se font toujours à l'entrée la plus grande de la table.

5 Conclusion

Nous nous sommes contentés de décrire dans ce chapitre les règles de transformation permettant de construire la représentation d'un type de communication. L'automatisation de cette étape nécessite dans certaines situations une intervention de l'utilisateur, notamment dans les cas suivants :

- pour vérifier la condition d'application d'une règle de transformation telle que les prédicats symétrique et transitive. Ces prédicats font appel à certaines propriétés du type paramètre (que nous avons noté ELT) des types de communication. Pour établir une telle vérification, il faut nécessairement connaitre ces propriétés. Or dans la définition des types de communication, les propriétés du type paramètre ne sont pas énoncées. Par ailleurs, même en disposant de ces propriétés, la preuve d'une condition d'applicabilité ne peut être automatisée dans tous les cas. Nous retrouvons ici les problèmes de preuve dans les spécifications paramétrées.

- pour fournir la taille de l'ensemble consommable qui est une information ne pouvant pas être extraite simplement de la spécification d'un type de communication.

Nous reviendrons sur ces problèmes au *chapitre 8*, où l'on décrira le système de transformation. Il reste maintenant à traduire la représentation ainsi construite en Ada. Pour cela le chapitre suivant propose des règles de compilation.

Chapitre 5

Passage au niveau dynamique

1 Introduction

Le chapitre précédent décrit la première étape du processus d'implantation des types de communication. Pour un type donné, cette étape construit une représentation en termes de structures de données et de fonctions facilement exprimables dans un langage de programmation. Cependant, la représentation ainsi construite ne prend pas en considération les règles qui garantissent l'exécution correcte des opérations de communication dans un environnement parallèle. Rappelons quelques unes de ces règles :

- l'activation en parallèle des opérations produire et consommer est régie par le principe de l'exclusion mutuelle,

- l'activation de l'opération consommer est assujettie à la condition pre-cons. L'appelant doit se mettre en attente lorsque cette condition est fausse.

C'est en particulier à la mise en œuvre de telles règles que nous consacrerons ce chapitre en présentant l'étape de passage au niveau dynamique. L'objet de cette deuxième étape est de construire autour de la représentation, un module Ada prenant en compte le caractère parallèle de l'environnement des types de communication.

2 Les principes

L'implantation des types de communication doit se faire da manière à minimiser les interférences avec l'expression des processus. Cette propriété est fondamentale pour l'évolution ultérieure des systèmes construits; car on doit pouvoir modifier la définition de la stratégie de communication sans toucher aux processus et vice versa. On retrouve ici la notion d'abstraction, le type de communication apparait aux processus comme un module défini par son interface et non par sa réalisation. Dans le langage Ada, le paquetage est l'outil d'abstraction le mieux adapté à ce type de situation. Cet outil est souvent

utilisé pour implanter les types abstraits de données, comme c'est le cas des types de communication.

Sans vouloir faire ici une présentation détaillée sur les différentes utilisations des paquetages en Ada, il est cependant nécessaire de rappeller deux points :

- Un paquetage est une unité de base du langage Ada qui permet de regrouper des entités logiquement reliées. Comme tout module Ada, le paquetage se décompose en deux parties :
  - un interface contenant les informations qui doivent être visibles des autres unités,
  - un corps décrivant les détails d'implémentation qui n'ont pas besoin d'être accessibles par d'autres unités.
- on peut paramètrer un paquetage par des objets, des types et des sous-programmes.

Nous utilisons donc le paquetage, comme outil pour encapsuler la représentation d'un type de communication. Il reste cependant à définir les outils permettant de prendre en compte le parallélisme de l'environnement. Plus précisement, nous aurons besoin d'outils de synchronisation qui permettent d'exprimer l'attente conditionnelle, l'exclusion mutuelle. Ces outils sont fournis par le noyau parallèle du langage Ada. Nous les présentons lors de l'implantation des opérations de communication. Nous présentons séparément la construction de l'interface et du corps du paquetage traduisant les types de communication.

3 Construction de l'interface

Pour construire la partie interface d'un paquetage, il faut décider de ce qui doit être visible par les clients ou les utilisateurs de ce paquetage. Dans le cas des types de communication; les seules opérations visibles par les clients que sont les processus sont les opérations de communication. les autres opérations servent à implanter l'interface et doivent donc être cachées aux processus. A chaque opération de communication est associée un sous-programme dans la partie interface dont le profil est constitué d'un seul paramètre. Ce paramètre représente la donnée communiquée et peut être :

- un paramètre qui joue le rôle d'une donnée. C'est le cas de l'opération produire.
- un paramètre qui joue le rôle d'un résultat. C'est le cas de l'opération consommer.

Un type de communication est paramètré par le type des données communiquées que nous notons elt. Le paquetage correspondant doit alors déclarer ce type comme paramètre générique. Il arrive aussi que la spécification d'un type de communication utilise des

opérations supposées prédéfinies sur le type elt. De telles opérations doivent à leur tour être déclarées comme paramètres génériques, sous forme de sous-programmes Ada. Nous donnons ci-dessous l'interface du paquetage "typcom" (nom du type de communication).

```
generic
   type elt is private;
   [with function nomop (données) return résultat;]
package typcom is
   procedure produire ( x : in elt );
   procedure consommer ( x : out elt );
end typcom ;
```

- "PRIVATE" indique que le type des données communiquées peut être n'importe quel type autorisant l'affectation et le test d'égalité,
- la partie [**with**...] indique la déclaration éventuelle des opérations du type elt, utilisées par le type de communication. Par exemple, le type croissante utilise dans sa spécification l'opération notée "<" sur les objets de type elt. La partie **with** du paquetage croissante sera définie de la manière suivante :
  with function "<" (e1,e2:elt) **return** boolean
- les attributs "IN" et "OUT" indiquent le mode de passage des paramètres, "IN" pour un paramètre donné, "OUT" pour un paramètre résultat.

Notons que contrairement à la solution classique utilisée en Ada pour implanter les types abstraits de données, le paquetage correspondant à un type de communication est considéré comme une machine abstraite. Il n'exporte pas de type et l'état interne des objets types de communication est encapsulé dans le corps du paquetage.

4 Construction de la partie corps

La partie corps correspond à la définition de la représentation interne des objets types de communication et de l'implantation des sous programmes constituant l'interface. Nous distinguons dans une telle définition les trois parties suivantes :

- la déclaration de la représentation interne qui définit l'état d'un type de communication,
- l'implantation des opérations caractéristiques (pre-prod, pre-con,s val-cons, post-cons),

- l'implantation de l'interface qui regroupe les opérations de communication produire et consommer.

Ces différentes parties sont construites à partir des résultats de l'étape de transformation. Nous présentons maintenant les règles de compilation pour chaque partie.

4.1 Représentation interne

La structure de données associée à un type de communication est formée des composantes d'histoire et du tampon de communication. Nous avons fait correspondre à chacune de ces entités une variable dont le type a été défini lors du processus de représentation :

- les variables d'histoire sont déclarées respectivement de type **REP_SP**, **REP_SC**. Comme nous l'avons noté dans la transformation de simplification, ces déclarations peuvent ne pas avoir lieu pour certains types de communication,

- la variable tampon est déclarée de type **REP_EA**, définit dans le composant choisi dans la bibliothèque de représentation,

- la représentation interne est déclarée de type REP_TYPCOM (TYP_COM est le nom du type de communication) que nous avons définis par le produit cartésien des types représentant les composantes d'histoire et le tampon de communication.

La traduction de ces déclarations en Ada ne pose aucun problème particulier comme nous le montre l'exemple suivant. Considérons le type de communication TC pour lequel l'étape de représentation produit les types suivants :

1. type REP_SP=<elem:ELT,vide:booléen>, qui définit la représentation de la composante SP.

2. type FILE, qui définit la représentation du tampon de communication (composante notée EA).

3. type REP_TC=<SP:REP_SP,EA:FILE>, qui définit la représentation du type de communication TC.

Le principe de traduction de ces types de données en Ada est le suivant :

- le produit cartésien noté < > se traduit par un type enregistrement. Ainsi nous obtenons pour les types REP_SP et REP_TC les déclarations Ada suivantes :

type REP_SP **is record** elem:ELT; vide:boolean; **end record**;	**type** REP_TC **is record** EP:REP_SP; EA:FILE; **end record**;

- le type FILE est définie dans le module Ada FILE_V de la bibliothèque de représentation. Pour rendre accessible ce type, il faut **importer** le module FILE_V par la clause **with**, puis l'**instancier** afin de produire un exemplaire. L'instantiation est nécessaire puisque les modules de la bibliothèque sont génériques. Nous obtenons ainsi les déclarations suivantes :
  with FILE_V; importation du module,
  package instance **is new** FILE_V (ELT); instantiation du module,
  use instance; pour accéder au type FILE et à ses opérations sans les préfixer par le mot instance qui est le nom de l'exemplaire.

La représentation interne des objets types de communication est une variable de type REP_TC.

4.2 Les opérations internes

Après avoir définit la représentation interne d'un type de communication, nous présentons maintenant la compilation des opérations qui permettent l'accès à cette représentation : les opérations caractéristiques (pre-prod, pre-cons, val-cons, post-cons) et les opérations définies sur les composantes d'un type de communication. Nous avons volontairement utilisé jusqu'à présent un langage applicatif pour décrire ces opérations et leurs représentations. Ceci permet, par exemple d'écrire dans la représentation d'un type de communication des définitions telles que :
EA'=enfiler(EA,e), où EA' et EA sont des objets de type file, enfiler est l'opération qui ajoute un élément en queue d'une file.

Dans un langage de programmation, cette définition conduit à la construction effective en mémoire d'une nouvelle file lors de l'appel de l'opération enfiler. Or dans la pratique, la file EA' qui résulte de l'opération enfiler est la même que la file initiale EA, dont on a simplement modifié le contenu. Cette remarque nous conduit à traduire l'opération enfiler en une procédure de même nom, qui modifie son premier argument. La même remarque peut être faite à propos des opérations qui jouent le rôle de constructeur ou de simplificateur. Ainsi nous associons à chaque opération interne, un sous-programme Ada qui peut être :

- une procédure, lorsque l'opération est un constructeur ou un simplificateur.

- une fonction, lorsque l'opération est un observateur (opération d'accès).

Le profil d'une opération permet de définir la liste des paramètres formels du sous-programme correspondant. Cependant, pour les opérations caractéristiques nous supprimons le type de communication de la liste des paramètres. En effet, celui-ci est définit par sa représentation interne qui est une variable globale à toutes ses opérations. Par exemple à l'opération :

pre-cons: TC → booléen, on associera : **function** pre_cons **return** boolean.

Nous donnons ci-dessous la compilation des opérations internes du type TC, en rappelant tout d'abord sa représentation et celle de la composante SP.

```
type REP_SP(ELT)
      =<elem:ELT,vide:booléen>
notation: reps:REP_SP,reps=<elem,vide>
repcreer: → REP_SP
repajouter: REP_SP×ELT → REP_SP
repinf: REP_SP×ELT→ entier
repcreer()=< indef,vrai>
repajouter(reps,e)=
si reps.vide alors <e,faux>
   sinonsi reps.elem>e alors <e,faux>
         sinon <reps.elem,faux>
   fsi
fsi
repinf(reps,e)=
si reps.vide alors faux
   sinonsi reps.dernier=e alors vrai
         sinon faux
   fsi
fsi
fin type
```

```
type REP_TC(ELT)
      =<SP:REP_SP,EA:FILE>
notation tc:REP_TC,tc=<SP,EA>
init() tc:REP_TC
pre: vrai
post tc=<repcreer(),creer()>
produire(tc:REP_TC,e:ELT)tc':REP_TC
pre: vrai
post: tc'=<SP',EA'> avec
SP'=repajouter(SP,e)
EA'= si pre_prod(tc,e)
      alors enfiler(EA,e)
      sinon EA
      fsi
consommer(tc:REP_TC)tc':REP_TC
pre: pre_cons(tc)
post: tc'=<SP,EA'>
avec EA'=post_cons(tc)
pre_prod(tc:REP_TC,e:ELT)b:booleen
pre: vrai
post: b=repinf(SP,e)
pre_cons(tc:REP_TC)b:booleen
pre: vrai
post: b=non vide(EA)
val_cons(tc:REP_TC)e:ELT
pre: vrai
post: e=tete(EA)
post_cons(tc:REP_TC)t:Table
pre: vrai
post: t=defiler(EA)
fin type
```

Le tableau suivant donne l'implantation Ada du type TC.


```
with FILE_V;
package body TC is
package instance is new FILE_V(ELT);
use instance;
type REP_SP is record
            elem:ELT;
            vide:boolean;
            end record;
type REP_TC is record
            SP:REP_SP;
            EA:FILE;
            end record;
rep : REP_TC;
procedure repcreer(reps:outREP_SP) is
begin
reps.vide:=false;
end repcreer;
procedure repajouter(reps:in outREP_SP;e:ELT) is
begin
if reps.vide then reps:=(e,false);
         else if reps.elem > e then reps:=(e,false);
                              else reps:=(reps.elem,false); end if;
end if;
end repajouter;
function repinf(reps:REP_SP;e:ELT) return boolean is
begin
if reps.vide then return true;
         else if reps.elem > e then return true;
                              else return false;
         end if;
end if;
end repinf;
procedure init is
begin
repcreer(rep.sp); creer(rep.EA);
end init;
function pre_prod (e:ELT) return boolean is
begin
return(inf(rep.SP,e));
end pre_prod;
procedure prod (e:ELT) is
begin
repajouter(rep.SP,e);
if pre_prod(e) then enfiler(rep.EA,e); end if;
end prod;
function pre_cons return boolean is
begin
return(not vide(rep.EA));
end pre_cons;
```

```
function val_cons return ELT is
begin
return(tete(rep.EA));
end val_cons;
procedure post_cons is
begin
defiler(rep.EA);
end post_cons;
Implantation des opérations de communication
begin
init;
end TC;
```

5 Compilation des opérations de communication

5.1 Aspect général

Ces opérations mettent en relation le type de communication avec son environnement qui peut être séquentiel ou parallèle. Si l'environnement est séquentiel, le type de communication n'est autre qu'un module au sens de [PAR 78]: il n'y a dans la mise en œuvre aucune synchronisation nécessaire pour retarder l'exécution d'une opération en attendant qu'une condition devienne vraie. En revanche, dans un environnement parallèle (comme c'est le cas ici) la mise en œuvre doit prendre en compte les règles de synchronisation impliquées par la relation de communication. Ces dernières expriment les cas d'attente des processus dans certaines situations. Plus précisement l'implantation doit respecter la sémantique des opérations de communication déjà présentée dans la première partie de cette thèse. Nous la rappelons ici à travers les deux règles suivantes :

- l'activation de l'opération consommer est assujettie à la condition pre_cons : le processus qui active cette opération doit se mettre en attente quand cette condition est fausse.

- l'activation simultanée des opérations de communication est régie par le principe de l'exclusion mutuelle.

Nous présentons une première solution qui dérive directement de la représentation des opérations de communication et des règles que nous venons d'énoncer. Nous constatons que cette solution impose une exclusion mutuelle totale sur l'activation de ces opérations [KOU 88]. Nous décrivons ensuite une deuxième solution qui affaiblit la condition d'exclusion mutuelle pour certains types de communication.

5.2 Première solution

Pour chacune des règles précédentes, nous allons définir les outils Ada permettant de la mettre en œuvre. L'intégration de l'ensemble de ces outils permet alors de construire la solution finale.

- Règle 1 : en associant à l'opération consommer une entrée d'une tâche Ada, soit c on peut implanter cette règle grâce à l'énoncé suivant :
  when pre_cons => **accept** c **do** ...**end** c;
  Nous réalisons ainsi l'attente conditionnelle.

- Règle 2 : on peut assurer l'exclusion mutuelle en associant à chaque opération une entrée d'une même tâche Ada. L'exclusion mutuelle sera assurée par construction, car les entrées d'une tâche s'exécutent toujours en exclusion mutuelle.

En partant de ces remarques, nous pouvons maintenant décrire la compilation des opérations de communication. Nous introduisons une tâche passive inter à deux points d'entrée (Règle 2) : ce type de tâche n'exécute que des instructions **accept**, éventuellement incluses dans des instructions **select** et des boucles et n'exécute donc aucune action propre (e.g action non associée à un rendez-vous). La partie spécification de la tâche inter est définie par le module suivant :

```
task inter is
   entry p ( x : in elt);--Réalise l'opération produire
   entry c ( x : out elt);--Réalise l'opération consommer
end inter;
```

La partie corps contient pour chaque entrée, le traitement à effectuer lors de la prise en compte d'un appel sur cette entrée. Ces traitements sont construits à partir de la représentation des opérations de communication comme le montre le tableau suivant.

```
task body inter is
v : elt;
begin
 loop
  select
   accept p (x : in elt) do
        v := x;
       end p;
       prod(v);
  or
   when pre_cons ⇒ accept c (x : out elt) do
        x := val_cons;
       end c;
       post_cons;
  end select;
 end loop;
end inter;
```

Commentaires :

- l'instruction **select** exprime l'indéterminisme : lorsque les deux entrées p et c sont activables, l'une d'entre elles sera choisie de façon arbitraire, puis exécutée,
- rappelons que pendant l'exécution du traitement inclu dans la construction **accept**, l'appelant est mis en attente. Nous avons placé les instructions prod (respectivement post_cons) à l'extérieur de la construction **accept** correspondant à une production (respectivement à une consommation) pour minimiser cette attente,
- lors de la compilation d'un type de communication, nous introduisons toujours une procédure nommée "prod" pour implanter la partie **post** de l'opération produire. Ceci nous permet de ne pas référencer directement la représentation d'un type dans l'implantation de l'interface (voir le corps de cette procédure dans le paquetage précédent).

Les opérations produire et consommer sont alors réalisées par les procédures suivantes :

```
procedure produire (e: in elt) is
begin
   inter.p(e);
end produire;
```

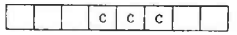
```
procedure consommer (e: out elt) is
begin
   inter.c(c);
end consommer;
```

La tâche inter joue le rôle d'un processus qui gère le médium de communication. On retrouve un tel processus chaque fois qu'il s'agit d'implanter une communication asynchrone dans les langages fondés sur le concept de rendez-vous. Si l'on se place dans le cas

d'un tampon à n cases, cette solution n'autorise pas l'exécution simultanée des opérations produire et consommer qui pourtant demeure possible dans certaines situations (il existe des cases vides pour produire et des données prêtes à être consommées). Ceci résulte de l'exclusion mutuelle totale entre les entrées p et c de la tâche inter. Nous allons examiner maintenant s'il est possible de remédier à cet exclusion mutuelle totale en proposant une deuxième solution.

5.3 Deuxième solution

Nous pouvons remarquer, qu'il n'est pas nécessaire de placer entièrement les opérations produire et consommer en exclusion mutuelle. En effet, considérons le schéma suivant :



Ce schéma illustre l'état du tampon à l'instant t :

- Les cases marquées par "c" contiennent des données non encore consommées.
- Les cases vides sont prêtes à recevoir des données.

Si à l'instant t, le consommateur (respectivement le producteur) appelle la procédure consommer (respectivement produire), la première solution impose que l'un des deux processus se mette en attente. Or, les deux procédures peuvent s'exécuter simultanément sur des éléments distincts du tampon (les cases marquées par "c" pour une consommation, les cases vides pour une production). Pour rendre cette exécution possible, il faut transformer la tâche inter de manière à ce que l'exclusion mutuelle soit définie sur une case et non sur la totalité du tampon de communication. En notant $produire_i$ (respectivement $consommer_i$) l'action de production (respectivement l'action de consommation) dans la case i. la remarque précédente conduit à affaiblir la règle d'exclusion mutuelle (Règle 2) en la redéfinissant de la manière suivante :

> l'activation simultanée des actions produire et consommer doit se faire en exclusion mutuelle seulement dans le cas où produire=$produire_i$ et consommer=$consommer_i$.

La mise en œuvre de cette nouvelle règle consistera donc à :

- définir les opérations $produire_i$ et $consommer_i$ sur la case i, considérée comme un tampon de taille un, et assurer l'exclusion mutuelle entre ces deux opérations.

- définir l'opération produire (respectivement consommer) à l'aide des opérations $produire_i$ (respectivement $consommer_i$).

Nous allons proposer dans un premier temps une solution pour le type de communication égalité, représenté par une file de taille n. Puis nous examinons si une telle solution peut être envisagée pour d'autres types de communication.

Commencons par implanter les opérations de communication $produire_i$ et $consommer_i$ définies sur la case i du tampon. Comme ces deux opérations doivent s'exécuter en exclusion mutuelle, la seule solution possible est de les implanter par deux entrées d'une même tâche Ada. Cependant, ceci nous conduit à déclarer n tâches, où chacune correspond à la gestion d'une case du tampon. Pour éviter de multiplier ces déclarations, nous introduisons la notion de type tâche : un modèle que l'on peut utiliser pour créer plusieurs tâches exécutant le même traitement. D'où le type inter_un suivant qui définit un modèle de tâches permettant de gérer des tampons de taille un.

```
task type inter_un is
   entry p (x: in ELT);
   entry c (x: out ELT);
end inter_un;
```

```
task body inter_un is
tamp:ELT;
vide:boolean:=true;
begin
loop
 select
  when vide => accept p (x: in ELT) do
                  tamp:=x;
                  end p;
                  vide:=false;
  or
  when not vide => accept c (x: out ELT) do
                      x:=tamp;
                      end c;
                      vide:=true;
 end select;
end loop;
end inter_un;
```

Le tableau tampon suivant déclare les n tâches qui correspondent à la gestion des cases du tampon :

tampon : **array** (1..n) **of** inter_un;

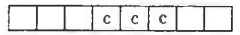
Ainsi l'opération $produire_i$ (respectivement $consommer_i$) définit sur la case i sera nommée tampon(i).p (respectivement tampon(i).c). Les opérations produire et consommer

sont alors réalisées par les procédures suivantes :

procedure produire (e: **in** elt) **is**
begin
  tampon(ip).p (e);
  ip:=ip **mod** n + 1;
end produire;
	procedure consommer (e: **out** elt) **is**
	begin
	  tampon(ic).c (e);
	  ic:=ic **mod** n + 1;
	end consommer;

Les variables ip et ic doivent être déclarées au même niveau que le tampon, de la manière suivante : ip,ic:**integer** := 1.

Notons que la modification de ces variables par les opérations de communication est faite de manière à ce que le tampon soit géré comme une file. Ceci permet à l'implantation d'être conforme à la stratégie de communication spécifiée par le type égalité. Pour montrer de manière intuitive que les opérations produire et consommer peuvent s'exécuter simultanément sur deux cases différentes du tampon, nous reprenons le schéma donné au début de ce paragraphe :



et qui illustre l'état du tampon à un instant donné t. Sur ce schéma la variable ip (respectivement ic) a comme valeur 7 (respectivement 4). L'activation simultanée des opérations produire et consommer entrainera celle de l'entrée p de la tâche d'indice 7 et de l'entrée c de la tâche d'indice 4. Le fait que ces deux entrées soient déclarées dans deux tâches différentes rend possible leur exécution de manière simultanée et en conséquence celle des opérations de communication. Nous remédions ainsi à l'exclusion mutuelle introduite par la première solution. Cependant, la création de n tâches pour gérer uniquement le tampon peut entrainer une surcharge au niveau du système à l'exécution et surtout lorsque le nombre n est grand. Mais au delà de ce détail technique et néanmoins important, on peut se demander s'il est possible de généraliser cette solution à tous les types de communication. Nous répondons négativement à cette question et ce pour les raisons suivantes :

- les n tâches n'ont aucune connaissance de l'état global du tampon de communication. Or pour certains types de communication, la vérification de la pré-condition de consommation et même le choix de la valeur à consommer nécessitent de connaitre cet état. Par exemple, consommer la valeur la plus petite de toutes celles que contiennent le tampon.

- même si le problème précédent est résolu, la décomposition des opérations produire et consommer de manière à les définir sur chaque case du tampon n'est pas immédiate.

En effet cette décomposition change le niveau d'observation auquel se placait la spécification initiale : on passe du contrôle des opérations produire et consommer à celui des opérations $produire_i$ et $consommer_i$. Ces dernières n'apparaissent pas dans la définition des types de communication.

Pour toutes ces raisons, nous préférons la première solution, qui ne pose aucune restriction quant à la classe des types de communication considérée. Il n'est d'ailleurs pas étonnant que la mise en œuvre d'une communication asynchrone dans les langages à rendez-vous se ramène toujours à cette solution. Nous présentons dans l'annexe C d'autres exemples d'implantations de types de communication.

Chapitre 6

Spécification d'un compilateur Lesp-Ada

1 Introduction

Ce chapitre présente la spécification d'un compilateur du langage Lesp vers le langage Ada. Le principe de base qui a guidé la conception de ce compilateur est le suivant: nous voulons que tout système parallèle observant certaines contraintes raisonnables puisse être traduit en un programme Ada compilable sans erreur et exécutable avec la même sémantique que dans [PER 85].

A cette fin, nous avons examiné le langage Lesp et tenté de définir pour chacune des constructions qu'il contient l'équivalent en Ada qui nous semblait le plus simple. Dans certains cas, cette garantie d'équivalence nous a mené vers des constructions Ada assez complexes. Nous traduisons donc tout système Lesp correct, en un programme Ada correct ayant le même comportement à l'exécution.

La spécification est exprimée à l'aide de la méthode dite sémantique naturelle [CLE 85] [KAH 87]. L'objectif d'une telle spécification est de fournir un énoncé formel comme fondement à l'étape de conception et de réalisation du compilateur.

La première partie de ce chapitre introduit la méthode sémantique naturelle. La partie suivante explicite les objets manipulés dans la spécification, la syntaxe et la signification des formules utilisées dans les règles de compilation. La troisième partie présente la spécification du compilateur et un exemple illustrant l'utilisation des règles de compilation.

2 Méthode de spécification: sémantique naturelle

Nous avons choisi d'exprimer la spécification du compilateur Lesp-Ada à l'aide de la méthode dite sémantique naturelle . Cette méthode a été destinée initialement à la défini-

tion de la sémantique des langages de programmation dans un style inférentiel. Le langage support de la méthode est appelé TYPOL [DES 86]. Les définitions ainsi construites dans ce langage peuvent être exécutées grâce au système CENTAUR [BOR 87]. Ce système permet de spécifier et de réaliser des environnements de programmation: interprète, compilateur, et vérificateur de la sémantique statique.

D'autres systèmes destinés à la génération de compilateurs ou d'environnements de programmation ont été proposés. Citons le système PERLUETTE [GAU 80] qui utilise une définition algébrique pour la spécification des langages de programmation. La compilation est formalisée par la représentation d'un type abstrait par un autre. Celui de L.Paulson [DES 84] utilise les grammaires attribuées et permet de générer des compilateurs pour une machine à pile. Ces systèmes se différencient essentiellement par la méthode de définition de la sémantique utilisée et le langage de description associé. Ils ont un champ d'application restreint et sont souvent trop spécifiques à une tâche donnée. Par ailleurs, et c'est là l'intérêt de la sémantique naturelle, ces systèmes ne possèdent pas en général un langage de spécification de haut niveau et facile à utiliser. Ceci est en particulier le cas de ceux qui utilisent les grammaires attribuées.

Les spécifications en sémantique naturelle sont définies en termes de règles d'inférence. De telles règles expriment comment une formule peut être déduite d'une ou de plusieurs autres formules. Dans la spécification du compilateur, les règles d'inférences vont permettre d'exprimer la compilation d'une construction du langage Lesp en fonction des sous-constructions qui la composent. Ainsi notre spécification sera donnée par la définition d'un système formel que nous décrirons en organisant ce chapitre de la manière suivante:

- nous commençons au paragraphe 3 par la définition des objets utilisés dans la spécification,
- nous présentons ensuite au paragraphe 4 les concepts de base de la sémantique naturelle: les prédicats, les formules et les règles d'inférence. Ces concepts sont d'une portée très générale, aussi nous ne les présentons que dans le cadre de la compilation.
- la nature et la syntaxe des formules de la spécification sont définies au paragraphe 5.
- les paragraphes 6 et 7 définissent les règles d'inférence composant la spécification du compilateur.
- le paragraphe 8 présente un exemple à travers lequel nous illustrons l'utilisation des règles d'inférence pour transformer un système parallèle Lesp en un programme Ada.

- enfin, nous concluons ce chapitre par des remarques sur la réalisation du compilateur.

3 Les objets

Les objets utilisés dans la spécification sont les éléments de la syntaxe abstraite des deux langages et l'environnement de la compilation.

3.1 Syntaxe abstraite

La sémantique naturelle est conçue pour décrire des définitions sémantiques de langage, il est donc naturel que les premiers objets manipulés soient des programmes ou plutôt leur représentation abstraite: les arbres de syntaxe abstraite.

Un arbre de syntaxe abstraite est une représentation interne des programmes. Il permet de conserver la structure des termes du langage qui sont syntaxiquement corrects tout en négligeant des informations telles que les signes de ponctuation et les mots-clés. Il nous faudra donc spécifier la syntaxe abstraite pour chacun des langages. Il est souhaitable, sinon indispensable si on veut pouvoir spécifier rigoureusement et facilement les choix de traduction, d'utiliser le même formalisme pour définir le langage source et le langage objet. Pour cela, nous nous sommes inspirés du formalisme METAL utilisé par le système CENTAUR [BOR 87]. La syntaxe abstraite est décrite à partir d'opérateurs et de phyla.

Opérateur

Un opérateur définit un type d'arbre de la syntaxe abstraite. Il est caractérisé par son nom, son arité (fixe ou variable), et les types de ses fils. Le type d'un fils est appelé phylum.

- opérateur d'arité fixe:

  Un opérateur d'arité fixe est défini par un nombre N fixé de phyla. Ces phyla peuvent être différents. Intuitivement, toute instance de cet opérateur est un nœud qui possède exactement N fils. Considérons l'exemple d'une sous-unité de compilation Ada, définie par les règles (BNF) suivantes:

  sub-unit ::= SEPARATE (ident) proper-body
  proper-body ::= pack-body | subp-body | task-body

  Nous pouvons associer à la première règle un opérateur binaire, dont le premier fils est un identificateur, le deuxième fils un corps de paquetage, de sous-programme ou

de tâche. La production correspondante de la syntaxe abstraite s'écrira ainsi sous forme fonctionnelle:

sub-unit : IDENT × PROPER-BODY → UNIT

Ceci signifie que sub-unit est un arbre binaire dont les deux fils appartiennent respectivement aux phyla IDENT et PROPER-BODY. Le type de cet arbre est le phyla UNIT.

- opérateur d'arité nulle:

  Un tel opérateur possède une valeur et représente un atome du langage. Il est caractérisé par le phyla auquel il appartient. Par exemple le phylum IDENT que nous avons introduit précédemment est le type d'un opérateur d'arité nulle:

  ident : → IDENT

  La valeur de cet opérateur est un identificateur.

- opérateur d'arité variable:

  C'est un opérateur qui possède un nombre variable de fils. Les phyla de ses fils doivent être identiques. Considérons l'exemple d'une suite d'instructions définies par l'opérateur **seq-stat**:

  seq-stat : STAT* → SEQ-STAT

  seq-stat définit un type d'arbres dont les instances sont des listes d'instructions (listes d'arbres dont le phylum est STAT). La notation **seq-stat**[] définie la liste vide, et l'opérateur + est utilisé pour désigner la concaténation des listes.

Phylum

Un phylum est un ensemble non vide d'opérateurs. Il définit les opérateurs greffables en un point donné d'un arbre. Par exemple le phylum STAT regroupe tous les opérateurs de la syntaxe abstraite représentant les différentes instructions du langage Ada. La définition de ce phylum sera définie ainsi:

STAT := **cond-stat, select-stat, loop-stat, null-stat,** etc

Une relation d'inclusion est définie sur l'ensemble des phyla. Par exemple:

PACK-BODY ⊂ PROPER-BODY, SUBP-BODY ⊂ PROPER-BODY,
TASK-BODY ⊂ PROPER-BODY

Présentée ainsi, la syntaxe abstraite est alors considérée comme une algèbre ordo-sortée [KAH 87], dont les sortes sont les phyla et dont les opérateurs sont les différentes compositions syntaxiques.

Notations

Dans la suite et pour rendre les définitions formelles plus assimilables, nous utiliserons les notations suivantes:

- Les opérateurs de la syntaxe abstraite seront décrits par une notation infixée. Les noms d'opérateurs sont en miniscules et en caractères gras, les noms des fils sont en majuscules et rappellent le phylum auquel doit appartenir le sous-arbre correspondant. Par exemple l'opérateur **cond-stat** qui représente l'instruction conditionnelle sera définit de la manière suivante:

  cond-stat(EXP,SEQ-STAT1,SEQ-STAT2)

  où EXP représente la partie condition et SEQ-STAT1 (respectivement SEQ-STAT2) représente la partie **alors** (respectivement la partie **sinon**).

- Le symbole + est utilisé pour dénoter la concaténation entre opérateurs de taille variable.

La syntaxe abstraite du langage Ada a été élaborée à partir de la syntaxe concrète du langage normalisé, présentée dans [THO 86]. Bien que nous n'ayons eu besoin que d'un sous ensemble du langage, nous avons pris en considération toutes les constructions importantes, hormis les clauses de représentation et les pragma qui sont des aspects liés à l'implantation du langage.

Par ailleurs, nous avons volontairement conservé les termes anglais afin de désigner les éléments de la syntaxe abstraite, pour plus de commodité avec le manuel de référence. Les syntaxes abstraites du langage Lesp et Ada sont données en annexe.

3.2 Environnement de la compilation

L'environnement sert à établir la liaison entre un identificateur et l'information qui lui est associée. Il définit ce que l'on appelle généralement la table des symboles pour un compilateur.

L'environnement dont on aura besoin dans la spécification du compilateur doit établir le lien entre les informations suivantes:

- le nom d'une entrée ou d'une sortie et le nom de port associé,

- le nom d'un processus composé et le nom du processus qui le déclare.

Pour définir un tel environnement, nous enrichissons les syntaxes abstraites des deux langages en introduisant un nouveau opérateur **env** à partir duquel nous pouvons créer et manipuler des environnements. Cet opérateur de phyla ENV est définit de la manière suivante:

env : ASSOCIATION* → ENV
ASSOCIATION := **port-assoc prog-assoc**
port-assoc : IDENT × IDENT → ASSOCIATION
prog-assoc : IDENT × IDENT → ASSOCIATION

Les opérateurs **port-assoc** et **prog-assoc** représentent les deux types de liens (ou association) que nous avons mis en évidence précédemment.

4 Les concepts de base

Le langage de spécification repose sur trois concepts de base: prédicat, formule et règle d'inférence.

4.1 Les prédicats

Le langage offre la possibilité de construire différentes sortes de prédicats. Certains symboles de prédicats sont prédéfinis et expriment une relation entre deux ou plusieurs objets. On peut leur attacher différentes significations selon le problème traité. Un exemple de tels prédicats noté → est utilisé par la suite sous la forme suivante:

C1 → C2

Avec C1 respectivement C2 des constructions appartenant au langage Lesp respectivement Ada. Ce prédicat signifie tout simplement que la construction C1 est traduite par la construction C2.

4.2 Les formules

Une formule exprime le fait que la preuve d'un prédicat nécessite la validité d'un certain nombre d'hypothèses. Elle est constituée de deux parties séparées par le symbole ⊢: les hypothèses et la conséquence qui doit être un prédicat. Les objets utilisés dans les formules sont les éléments de la syntaxe abstraite et les objets de type ENV représentant

l'environnement de compilation. Un exemple de formule est donné ci-dessous:

ENV ⊢ construction-Lesp → construction-Ada

- "construction-Lesp" est un élément de la syntaxe abstraite du langage Lesp,
- "construction-Ada" est un élément de la syntaxe abstraite du langage Ada,
- les objets de type ENV sont des listes de couples (x,y) où x est un identificateur, et y l'information associée.

4.3 Les règles d'inférence

Une règle d'inférence exprime comment une formule peut être déduite d'une ou plusieurs autres formules. Elle est constituée de deux parties séparées par une ligne horizontale: le numérateur est une collection de formules appelées prémisses de la règle, le dénominateur est formé d'une seule formule qui constitue la conclusion. La règle suivante spécifie la traduction de la conditionnelle Lesp en Ada:

$$\frac{\begin{array}{c}\text{ENV} \vdash \text{EXPL} \rightarrow \text{EXPA} \\ \text{ENV} \vdash \text{INSTSL1} \rightarrow \text{SEQ-STAT1} \\ \text{ENV} \vdash \text{INSTL2} \rightarrow \text{SEQ-STAT2}\end{array}}{\text{ENV} \vdash \textbf{sialorssinon}(\text{EXPL},\text{INSTL1},\text{INSTL2}) \rightarrow \textbf{cond-stat}(\text{EXPA}, \text{SEQ-STAT1},\text{SEQ-STAT2})}$$

Cette règle peut être paraphrasée en disant:

- si la compilation de la partie condition EXPL donne pour résultat l'expression EXPA,
- si la compilation de la partie **alors**, respectivement la partie **sinon**, donne comme résultat la suite d'instruction SEQ-STAT1, respectivement SEQ-STAT2,

alors la compilation de l'instruction **sialorssinon** donne comme résultat l'instruction **cond-stat** du langage Ada.

4.4 Modularité

Lorsque l'on spécifie une fonction sémantique à l'aide d'un système de règles d'inférence, il est intéressant de regrouper en module les règles selon leurs rôles dans la spécification. Ainsi, si on prend comme exemple la sémantique statique d'un langage de programmation, on trouve des règles pour la vérification des types, d'autres pour la gestion de la portée et la visibilité des identificateurs. La sémantique naturelle permet de regrouper ces différentes règles dans des modules appelés SET. Bien entendu, le module principal est l'ensemble de règles décrivant la fonction sémantique d'intérêt.

La syntaxe d'une définition de module est donnée ci-dessous:

SET nom IS
corps
END nom

où "corps" est un ensemble de règles d'inférence.

Si nous voulons de l'intérieur d'un système de règles, utiliser une fonction sémantique externe à ce système, nous devons nommer cette fonction par le nom du module où elle est définie. Ce dernier apparait sur le symbole $\vdash$ des formules faisant référence à ce module.

Du point de vue preuve, la référence à un module dans une règle, implique le passage d'un système d'inférence à un autre afin de prouver un prédicat. En ce qui nous concerne, nous utilisons au paragraphe 6.1 le concept de module pour la spécification de l'environnement.

5 La syntaxe des formules

Après avoir présenté l'ensemble des objets et concepts qui vont être utilisés dans la spécification, nous définissons maintenant la syntaxe des formules et leurs significations. Pour cela nous distinguons trois groupes de formules:

5.1 Formules de compilation

De telles formules spécifient la compilation d'une expression Lesp en Ada. La syntaxe d'une formule de compilation est la suivante:

ENV $\vdash$ cons-Lesp $\rightarrow$ list-cons-Ada

- cons-Lesp est un élément de la syntaxe abstraite du langage Lesp,
- list-cons-Ada est une liste d'éléments de la syntaxe abstraite du langage Ada séparés par des virgules. Notons que la compilation d'une construction Lesp produit dans certains cas plusieurs constructions Ada,
- ENV représente l'environnement de la compilation.

Les deux groupes de formules que nous allons présenter maintenant sont relatifs à la manipulation de l'environnement. Ils introduisent une surcharge du symbole $\vdash$ lorsqu'elles sont utilisées dans les règles d'inférence spécifiant la compilation. Le nom qui apparait sur le symbole $\vdash$ est alors le nom du module dans lequel ces formules doivent être prouvées.

5.2 Formules de construction de l'environnement

De telles formules spécifient la construction de l'environnement de compilation. Nous classons ces formules en deux sous groupes selon les informations qu'elles prennent en compte:

- ENV $\overset{\textbf{c-port}}{\vdash}$ exp-Lesp,**ident** p : ENV
  cette formule exprime la construction de l'environnement étant donnés une liste d'entrée ou de sortie et l'identificateur de port associé.
- ENV $\overset{\textbf{c-prog}}{\vdash}$ exp-Lesp,**ident** p : ENV
  Cette formule exprime la construction de l'environnement étant donnés une liste de processus composés et l'identificateur du processus qui les déclare.

5.3 Formules d'accès à l'environnement

Ces formules spécifient l'accès à l'environnement. Comme pour la construction, nous distinguons deux sous-groupes:

- ENV $\overset{\textbf{r-port}}{\vdash}$ **ident** x:**ident** p
  cette formule exprime l'accès au nom p du port associé à l'entrée ou la sortie de nom x,
- ENV $\overset{\textbf{r-prog}}{\vdash}$ **ident** x:**ident** p
  cette formule exprime l'accès au nom p du processus qui déclare le processus composé de nom x.

6 Spécification de l'environnement

La spécification de l'environnement est définie par quatre modules: les modules de construction et les modules d'accès.

6.1 Modules de construction

Le module c-port

Le module **c-port** spécifie la construction de l'environnement étant donnés une entrée ou une sortie et le port associé. La construction est effectuée à partir de l'interface des processus élémentaires. C'est dans l'interface que sont déclarées les entrées et les sorties d'un processus. Aussi les règles d'inférence qui composent le module **c-port** ont pour but de parcourir ces déclarations et de construire pas à pas l'environnement. Rappelons que

l'environnement ainsi construit, sera utilisé pour compiler la partie corps d'un processus et plus précisement les énoncés de communication (produire et conommer).

SET **c-port** IS

(1)
ENV ⊢ ENTRES,IDENT : ENV'
ENV' ⊢ SORTIES,IDENT : ENV"

ENV ⊢ **specif**(ENTRES,SORTIES),IDENT : ENV"

(2)
ENV ⊢ ENTRE,IDENT : ENV'
ENV' ⊢ ENTRES,IDENT : ENV"

ENV ⊢ **entres**(ENTRE+ENTRES),IDENT : ENV"

(3)
ENV ⊢ SORTIE,IDENT : ENV'
ENV' ⊢ SORTIES,IDENT : ENV"

ENV ⊢ **sorties**(SORTIE+SORTIES),IDENT : ENV"

(4)
ENV ⊢ **entres**[],IDENT : ENV

(5)
ENV ⊢ **sorties**[],IDENT : ENV

(6)
ENV ⊢ NOMVARS,**ident** p : ENV'

ENV ⊢ **entre**(NOMVARS,**ident** t,**ident** p), **ident** p : ENV'

(7)
ENV ⊢ NOMVARS,**ident** p : ENV'

ENV ⊢ **sortie**(NOMVARS,**ident** t,**ident** p), **ident** p : ENV'

(8)
ENV ⊢ NOMVARS,**ident** p : ENV'

ENV ⊢ **nomvars**[**ident** x + NOMVARS],**ident** p : env[**port-assoc**(**ident** x,**ident** p)+ENV']

(9)
ENV ⊢ **nomvars**[],IDENT : ENV

END **c-port**

Les opérateurs de la syntaxe abstraite du langage Lesp utilisés dans ce module sont définis de la manière suivante:

- **specif**: définit l'interface d'un processus élémentaire. Cet opérateur possède deux fils: la liste des déclarations d'entrées (notée ENTRES) et la liste des déclarations de sorties (notée SORTIES).

  specif: ENTRES × SORTIES → SPECIF

- **entre** (respectivement **sortie**) définit une déclaration d'entrée (respectivement de sortie).

  entre: NOMVARS × IDENT × IDENT → ENTRE
  sortie: NOMVARS × IDENT × IDENT → SORTIE

- **nomvars** définit une liste d'identificateurs :

  nomvars: IDENT* → NOMVARS

Nous expliquons maintenant le rôle de chaque règle du module:

- la règle (1) exprime la construction de l'environnement à partir de l'interface d'un processus élémentaire.
- la règle (2) (respectivement (3)) exprime la construction de l'environnement à partir d'une liste de déclaration d'entrées (respectivement de sorties).
- la règle (4) (respectivement (5)) exprime que l'environnement n'est pas modifié lorsqu'il s'agit d'une liste vide de déclaration d'entrées (respectivement de sorties).
- la règle (6) (respectivement (7)) exprime la construction de l'environnement à partir d'une déclaration d'entrées (respectivement de sorties).
- La règle (8) exprime la construction de l'environnement à partir d'une liste d'identificateurs d'entrées ou de sorties et d'un identificateur de port.
- L'axiome (9) exprime que l'environnement n'est pas modifié lorsqu'il s'agit d'une liste vide d'identificateurs d'entrées ou de sorties.

Le module **c-prog** suivant suit le même principe.

Le module c-prog

Ce module spécifie la construction de l'environnement étant donnés un processus composé et le nom du processus qui le déclare.

SET **c-prog** IS

(1)
ENV ⊢ DECL-PROGS,IDENT : ENV'

ENV ⊢ **decl-progs**[PROCESS+DECL-PROGS],IDENT :ENV'

(2)
ENV ⊢ PROCESS,IDENT : ENV

(3)
ENV ⊢ PROG,**ident** p : ENV'
ENV ⊢ DECL-PROGS,**ident** p : ENV"

ENV ⊢ **decl-progs**[PROG+DECL-PROGS],**ident** p :env[ENV'+ENV"]

(4)
ENV ⊢ **prog**(**ident** x,DECL-PROGS,PORTS),**ident** p :
env[**prog-assoc**(**ident** x,**ident** p)+ENV]

(5)
ENV ⊢ **decl-progs**[],IDENT : ENV

END **c-prog**

Les opérateurs de la syntaxe abstraite du langage Lesp utilisés dans ce module sont définis de la manière suivante:

- **decl-progs**: définit une liste de déclarations de processus (élémentaires et composés), et de types de communication:
  decl-progs: DECL-PROG* → DECL-PROGS
  DECL-PROG := **process prog tca**

- PROCESS correspond à une déclaration de processus élémentaire,

- **prog**: définit un processus composé,
  prog: IDENT × DECL-PROGS × PORTS → DECL-PROG
  PORTS := **port*** (liste de déclaration de ports)

6.2 Les modules d'accès

Le module r-port

Ce module spécifie la recherche dans l'environnement du port associé à une entrée ou une sortie donnée .

SET **r-port** IS

(1)
env[**port-assoc**(**ident** x,**ident** p)+ENV] ⊢ **ident** x : **ident** p

(2)
ENV ⊢ **ident** x1 : **ident** p1

---

env[**port-assoc**(**ident** x2,**ident** p2)+ENV] ⊢ **ident** x1 : **ident** p1
END **r-port**

- L'axiome (1) exprime l'accès au nom du port associé à une entrée ou une sortie.

- La règle (2) exprime le parcours de l'environnement pour l'accès au nom du port associé à une entrée ou une sortie donnée.

Le module r-prog

La spécification donnée par le module suivant suit le même principe que précédemment pour exprimer l'accès au processus qui déclare un processus composé donné.

SET **r-prog** IS

(1)
env[**prog-assoc**(**ident** x,**ident** p)+ENV] ⊢ **ident** x : **ident** p

(2)
env ⊢ **ident** x1 : **ident** p1

---

env[**prog-assoc**(**ident** x2,**ident** p2)+ENV] ⊢ **ident** x1 : **ident** p1

END **r-prog**

7 Spécification des règles de compilation

La spécification consiste en un ensemble de règles d'inférence, chacune décrivant la transformation d'un opérateur Lesp en un ou plusieurs opérateurs Ada. Cette spécification utilise les modules spécifiés précédemment pour construire et accéder à l'environnement.

7.1 Les ports

Un port constitue un objet, support de communication via lequel les processus peuvent communiquer des valeurs. Au niveau du langage Lesp, un port est défini comme une instance d'un type de communication. Nous avons vu que la construction Ada correspondant à un type de communication est un paquetage paramétré par le type des données communiquées, où seules les opérations de communication sont visibles aux processus. Nous

associons à chaque déclaration de port, un exemplaire du package générique réalisant le type de communication utilisé dans cette déclaration. L'instantiation est faite en passant comme paramètre effectif le type de la donnée communiquée associée au port.

Exemple:
Représentation textuelle:
port p : égalité(**entier**) → **package** p **is new** égalité (**integer**)
Représentation arborescente:
port("p","égalité","entier") → **pack-inst**("p","égalité","integer")

Une telle solution considère un port comme étant un objet abstrait dans lequel sont définies les actions de communication "produire" et "consommer". Leur réalisation est cachée aux processus.

ENV ⊢ **port**(**ident** p,**ident** tc,**ident** t) → **ident** tc , **pack-inst**(**ident** p,**ident** tc, **ident** t)

Lexique:

ident p	: nom du port
ident tc	: nom du type de communication
ident t	: nom du type des données communiquées
port	: opérateur définissant un port
pack-inst	: opérateur définissant l'instantiation de paquetage générique

Le nom du type de communication (**ident** tc) sera utilisé dans la clause d'importation (**with**) du programme principal. En effet, ces types sont implantés par des paquetages génériques et sont séparés du programme principal pour constituer des unités autonomes.

7.2 Les entrées et les sorties

Chaque entrée ou sortie est traduite par la déclaration d'une ou de plusieurs variables locales aux processus. Le nom et le type de ces variables sont identiques à ceux de l'entrée ou de la sortie qu'elles représentent. Toutefois, le nombre de variables engendrées dépend de la nature du processus déclarant l'entrée ou la sortie:

- cas d'un processus élémentaire : l'entrée ou la sortie est locale au processus, elle est traduite par une seule variable.
- cas d'un processus composé : l'entrée ou la sortie est visible par les processus internes. Il y a création d'une variable locale pour chacun de ces processus.

L'association des ports aux variables locales engendrées se fait dans la partie corps des processus, plus précisément au niveau des instructions de communication(produire et consommer). Ces dernières sont préfixées dans le corps des processus par le nom du port qui les définit.

Exemple:
Représentation textuelle: **sortie** s : **entier vers** p → s : **integer**
Représentation arborescente: **sortie**("s","entier","p") → **dvar**("s","integer")

ENV ⊢ **sortie**(NOMVARS,**ident** t,**ident** p) → **dvar**(NOMVARS,**ident** t)

Lexique

NOMVARS	: liste d'identificateurs
ident t	: nom du type des sorties
ident p	: nom de port
sortie	: opérateur définissant la déclaration d'une suite de sorties
dvar	: opérateur définissant une déclaration de variables

7.3 Les instructions de communication

Les instructions de communication au nombre de deux sont exprimées en Lesp par l'appel de l'opération *produire* ou de l'opération *consommer*. Ces opérations constituent l' interface des paquetages Ada résultant de la compilation des ports.

La compilation de l'appel *produire(s)* (respectivement *consommer(e)*) nécessite tout d'abord la connaissance du port sur lequel est définie cette opération. Cette information est obtenue en cherchant dans l'environnement le port associé à la sortie *s* (respectivement l'entrée *e*). Ensuite, l'instruction est traduite par un appel de procédure Ada préfixé par le nom du port.

Ainsi l'appel **produire(s)** dans le corps d'un processus se traduit par **p.produire(s)**, où **p** est le port associé à la sortie **s**. Nous ne donnons que la règle concernant l'opération *produire*, l'opération *consommer* suit le même principe.

$$\frac{\text{ENV} \vdash^{r=\textbf{port}} \textbf{ident}\ s{:}\textbf{ident}\ p}{\text{ENV} \vdash \textbf{produire}(\textbf{ident}\ s) \rightarrow \textbf{proc-call}(\textbf{ref-mod}(\textbf{ident}\ p,\text{"produire"}),\textbf{ident}\ s)}$$

Lexique

ident s	: nom de la donnée communiquée
ident p	: nom de port sur lequel est produite la donnée **ident** s
prod	: opérateur définissant en Lesp l'instruction de communication produire
proc-call	: opérateur définissant l'appel d'une procédure en Ada
ref-mod	: opérateur définissant la notation pointée en Ada pour préfixer une entité par le nom du module qui la déclare

7.4 Les processus élémentaires

Les processus élémentaires sont traduits par des tâches Ada. L'idée de séparer la partie spécification des tâches de leur partie corps imposée par le langage Ada, nous conduit à générer pour chaque tâche en plus de sa partie corps une partie spécification sans point d'entrée. Rappelons que dans le cas du langage Ada, les points d'entrée sont utilisés pour exprimer la communication et la synchronisation entre tâches. La communication est exprimée de manière directe et asymétrique; un processus nomme les points d'entrée d'un autre. Dans le cas de Lesp, la communication est indirecte; les processus communiquent en utilisant les ports. Cet "indirection" fait que la partie spécification des tâches réalisant les processus Lesp est vide.

Exemple:

Représentation textuelle:

process prod :: →	task prod; **task body** prod **is**
–SPECIF: déclaration des	–DEC-PART: partie déclarative
entrées et des sorties	d'un module Ada
sortie s : **entier vers** p;	s : **integer**;
–CORPS: partie corps	–SEQ-STAT: partie instruction
debut	**begin**
.....;	.....;
produire(s);	p.**produire**(s);
.....;	.....;
fin prod;	**end** prod;

Représentation arborescente:

process("prod",**specif**(**entres**[]**sortie** ("s","entier","p")),**corps**(**produire**("s")))
→
task-spec("prod"),
task-body("prod",**dvar**("s","integer"),**seq-stat**(**proc-call**(**ref-mod**("p","produire"),"s")))

$$\frac{\text{ENV} \vdash \text{ENTRES} \rightarrow \text{DVARS1} \quad \text{ENV} \vdash \text{SORTIES} \rightarrow \text{DVARS2}}{\text{ENV} \vdash \textbf{specif}(\text{ENTRES},\text{SORTIES}) \rightarrow \textbf{dvars}(\text{DVARS1}+\text{DVARS2})} \quad (1)$$

$$\frac{\text{ENV} \vdash \text{DECLS} \rightarrow \text{DEC-PART} \quad \text{ENV} \vdash \text{INSTRS} \rightarrow \text{SEQ-STAT}}{\text{ENV} \vdash \textbf{corps}(\text{DECLS},\text{INSTRS}) \rightarrow \text{DEC-PART}\ ,\ \text{SEQ-STAT}} \quad (2)$$

$$\frac{\text{ENV} \overset{\textbf{c-port}}{\vdash} \text{SPECIF},\text{IDENT} : \text{ENV}' \quad \text{ENV} \vdash \text{SPECIF} \rightarrow \text{DVARS} \quad \text{ENV}' \vdash \text{CORPS} \rightarrow \text{DEC-PART},\text{SEQ-STAT}}{\text{ENV} \vdash \textbf{process}(\textbf{ident}\ \text{p},\text{SPECIF},\text{CORPS}) \rightarrow \textbf{task-spec}(\textbf{ident}\ \text{p},()),\ \textbf{task-body}(\textbf{ident}\ \text{p},\text{DEC-PART}+\text{DVARS},\text{SEQ-STAT})} \quad (3)$$

Lexique:

ident p	: nom de processus
SPECIF	: déclaration des entrées et des sorties
CORPS	: déclaration des entités locales et définition de la partie instruction
DVARS	: déclaration de variables
DEC-PART	: partie déclarative d'un module Ada
SEQ-STAT	: séquence d'instructions Ada
process	: opérateur définissant les processus élémentaires Lesp
task-spec	: opérateur définissant la partie spécification des tâches Ada **task-spec** : IDENT × ENTRY-DECS → TASK-SPEC
task-body	: opérateur définissant la partie corps des tâches Ada **task-body** : IDENT × DEC-PART × SEQ-STAT → TASK-BODY
+	: concaténation de deux listes

- La règle (1) (respectivement (2)) spécifie la compilation de la partie interface (respectivement la partie corps) d'un processus élémentaire.
- La règle (3) spécifie la compilation d'un processus élémentaire. Notons que la première prémisse de cette règle introduit une surcharge du symbole $\vdash$. Une telle prémisse doit être prouvée en utilisant les règles spécifiées dans le module **c-port** présenté au paragraphe 6.1. Elle a pour but de définir le nouvel environnement dans lequel est compilé le corps du processus.

7.5 Compilation séparée

La nécessité de découper les textes d'un programme pour pouvoir compiler séparément est évidente, surtout lorsqu'il s'agit de programmes structurés en modules comme c'est le cas en Lesp. Cependant, les programmes Lesp ne font pas appel à la compilation

séparée, ils sont "monolithiques". Les facilités offertes par le langage Ada en matière de compilation séparée, nous ont conduit à décomposer les systèmes parallèles Lesp en modules compilables séparément. Ce découpage est pris en compte par le compilateur, produisant ainsi trois sortes de modules:

- la procédure représentant le processus principal,
- les paquetages représentant les types de communication,
- les tâches représentant les processus composés.

Les deux premiers modules constituent des unités de compilation Ada autonomes; elles ne sont incluses dans aucune unité et forment des modules de bibliothèque, tandis que les tâches représentant les processus composés constituent des sous-unités de compilation; normalement incluses dans d'autres unités et détachées de celles-ci pour la compilation. Remarquons que c'est la seule possibilité offerte par le langage Ada pour la compilation séparée des tâches. Nous pouvons adopter un découpage plus fin en considérant les processus élémentaires comme étant des sous-unités de compilation. Cependant, cette solution risque de rendre complexe la correspondance entre le programme Ada obtenu et le programme Lesp correspondant. Par ailleurs, nous voulons que l'auteur du programme original ait le plus de facilités pour faire évoluer le programme obtenu.

De ces constations, il résulte que le programme Ada obtenu est alors formé d'une procédure constituant le programme principal, et d'une liste d'unités de compilation. Cette liste inclut le corps des tâches représentant les processus composés, les parties spécifications et les parties corps des paquetages représentant les types de communication.

7.6 Les processus composés

Les processus composés sont traduits par des tâches Ada à l'exception du processus racine de la hiérarchie. Il est traduit par une procédure et constitue le programme principal d'un système parallèle.

Les processus composés n'exécutent aucune action propre, leur activité se résume à l'activation simultanée des processus internes. Une telle action est réalisée implicitement en Ada à la rencontre du mot clé BEGIN d'une procédure ou d'une tâche. Il en résulte que la partie corps se réduit à l'instruction vide du langage Ada.

Nous donnons ci-dessous un système Lesp qui exprime un schéma producteur/consommateur et le programme Ada correspondant.

```
--processus principal
process prod_cons ::
--PORTS: déclaration de ports
 port p : egalite(entier);
--DECL-PROGS: déclaration de processus
  et de types de communication
 process prod ::
  sortie s : entier vers p;
 debut
  produire(s);
 fin prod;
 process cons ::
  entre e : entier via p;
 debut
  consommer(e);
 fin cons;
 --spècification du type égalité
--CORPS: corps du processus prod_cons
 debut
 prod // cons
fin prod_cons;
```

```
--W-CLAUSE: importation du paquetage égalité
with egalite;
--PROC-BODY: programme principal
procedure prod_cons is
 DEC-PART: partie déclarative du programme
  package p is new egalite(integer);
  task prod;
  task cons;
  task body prod is
   s : integer;
  begin
   p.produire(s);
  end prod;
  task body cons is
   e : integer;
  begin
   p.consommer(e);
  end cons;
 SEQ-STAT: partie corps du programme
  begin
  null;
end prod_cons;
```

Nous présentons maintenant les deux règles de compilation concernant le processus principal et les processus composés.

processus principal

$$\frac{\text{env}[] \overset{\text{c-prog}}{\vdash} \textbf{ident p} : \text{ENV1} \quad \text{ENV1} \vdash \text{DECL-PROGS} \rightarrow \text{DEC-PART,LUNITS} \quad \text{ENV1} \vdash \text{PORTS} \rightarrow \text{W-CLAUSE,D}}{\text{env}[] \vdash \textbf{prog(ident p,DECL-PROGS,PORTS)} \rightarrow \textbf{unit-comp}(\text{W-CLAUSE,()},\textbf{proc-body}(\textbf{entete}(\textbf{ident p}, ()), \text{DEC-PART+D},\textbf{null})),\text{LUNITS}}$$

processus interne

$$\frac{\text{ENV} \overset{\text{r-prog}}{\vdash} \textbf{ident p1} : \textbf{ident p2} \quad \text{ENV} \overset{\text{c-prog}}{\vdash} \textbf{ident p1} : \text{ENV1} \quad \text{ENV1} \vdash \text{DECL-PROGS} \rightarrow \text{DEC-PART,LUNITS} \quad \text{ENV1} \vdash \text{PORTS} \rightarrow \text{W-CLAUSE,D}}{\text{ENV} \vdash \textbf{prog(ident p1,DECL-PROGS,PORTS)} \rightarrow \textbf{task-spec(ident p1,())},\textbf{task(ident p1)}, \textbf{unit-comp}(\text{W-CLAUSE,()},\textbf{sub-unit(ident p2,task-body (ident p1,DEC-PART+D,null))}, \text{LUNITS}}$$

Lexique :

prog	: opérateurs définissant les processus composés
ident p1	: nom du processus
ident p2	: nom du processus composé déclarant le processus de nom **ident p1**
DECL-PROGS	: déclarations de processus et types de communication
PORTS	: déclaration de ports
DEC-PART	: déclaration de la partie spécification et de la partie corps des processus internes
D	: liste d'instantiation de paquetage générique représentant les ports de communication utilisés dans la déclaration de ports
unit-comp	: opérateur définissant une unité de compilation autonome **unit-comp** : W-CLAUSE × U-CLAUSE × UNIT → UNIT-COMP
W-CLAUSE	: clause d'importation en Ada (**with**), mentionnant la liste des noms de types de communication utilisés dans la déclaration des ports
U-CLAUSE	: clause **use** du langage Ada
UNIT	: phylum regroupant toutes les unités Ada pouvant être compilées séparément
sub-unit	: opérateur définissant une sous-unité de compilation **sub-unit** : IDENT × PROPER-BODY → UNIT
PROPER-BODY	: phylum regroupant les sous-unités de compilation Ada: corps de sous-programmes, de paquetages ou de tâches
LUNITS	: liste d'unités de compilation contenant les corps de tâches représentant les processus composés et les paquetages représentant les types de communication
proc-body	: opérateur définissant un corps de procédure
entete	: opérateur définissant l'entête d'un sous programme
null	: opérateur définissant l'instruction vide en Ada
task	: opérateur définissant la déclaration d'une sous-unité de compilation; corps de tâche
env[]	: liste vide représentant l'environnement initial

7.7 Les types

Comme le langage Ada ne permet pas l'utilisation de types anonymes (sauf pour les tableaux dans certains contextes), nous définissons de tels types séparément, produisant de nouveaux identificateurs de types lorsque cela est nécessaire.

Exemple:

soit la structure S exprimée en Lesp de la manière suivante:

```
S : struct
    champ1 : entier;
    champ2 : car
    finstruct
```

le type de la variable S est anonyme, la compilation en Ada de cette déclaration est la suivante:

```
type type_s is
    record
    champ1 : integer;
    champ2 : character;
    end record;
    S : type_s;
```

$$\frac{\text{ENV} \vdash \text{NOMVARS} \rightarrow \text{ID-LIST} \quad \text{ENV} \vdash \text{TYPCONS} \rightarrow \text{IDENT , TYPE-DEF}}{\text{ENV} \vdash \textbf{dvar}(\text{NOMVARS,TYPCONS}) \rightarrow \textbf{type-dec}(\text{IDENT,TYPE-DEF}),\textbf{dvar}(\text{ID-LIST,IDENT})}$$

Lexique

NOMVARS	: liste de noms de variables en Lesp
ID-LIST	: liste de noms de variables en Ada
TYPCONS	: définition de type dans le langage Lesp
IDENT	: nom de type
TYPE-DEF	: définition de type dans le langage Ada
dvar	: opérateur définissant la déclaration de variables
type-dec	: opérateur définissant la déclaration de types en Ada

7.8 Les règles de liaison

Comme nous pouvons le constater, la compilation d'un opérateur Lesp donne lieu à plusieurs opérateurs Ada. Il s'agit ici de décrire les règles d'inférence permettant de lier ces opérateurs soit à la partie déclarative d'une unité de programme, soit à la liste des unités de compilation qui est syntaxiquement séparée du reste du programme. Ce problème de liaison est dû essentiellement à l'existence en Ada de deux parties (spécification,corps)

pour la définition des modules.

La première règle concerne le regroupement dans la même partie déclarative, des parties spécification et corps de la tâche représentant un processus élémentaire. Notons que la spécification doit précéder le corps:

$$\frac{\text{ENV} \vdash \text{PROCESS} \rightarrow \text{TASK-SPEC,TASK-BODY} \quad \text{ENV} \vdash \text{DECL-PROGS} \rightarrow \text{DEC-PART,LUNITS}}{\text{ENV} \vdash \textbf{decl-progs}[\text{PROCESS+DECL-PROGS}] \rightarrow \textbf{dec-part}[\text{TASK-SPEC+DEC-PART+TASK-BODY}],\text{LUNITS}}$$

La règle suivante précise d'une part le regroupement de la partie spécification et la déclaration de la partie corps de la tâche représentant un processus composé, d'autre part l'ajout de la sous-unité de compilation englobant le corps à la liste des unités de bibliothèque symbolisée par **lunits**:

$$\frac{\text{ENV} \vdash \text{PROG} \rightarrow \text{TASK-SPEC,TASK,SUB-UNIT} \quad \text{ENV} \vdash \text{DECL-PROGS} \rightarrow \text{DEC-PART,LUNITS}}{\text{ENV} \vdash \textbf{decl-progs}[\text{PROG+DECL-PROGS}] \rightarrow \textbf{dec-part}[\text{TASK-SPEC+DEC-PART+TASK}], \textbf{l-units}[\text{SUB-UNIT+LUNITS}]}$$

Enfin, la dernière règle concerne les types de communication. La spécification et le corps du paquetage représentant un type de communication sont des unités de bibliothèque et donc détachées du programme prinicipal:

$$\frac{\text{ENV} \vdash \text{TCA} \rightarrow \text{UNIT-COMP1,UNIT-COMP2} \quad \text{ENV} \vdash \text{DECL-PROGS} \rightarrow \text{DEC-PART,LUNITS}}{\text{ENV} \vdash \textbf{decl-progs}[\text{TCA+DECL-PROGS}] \rightarrow \text{DEC-PART}, \textbf{lunits}[\text{UNIT-COMP1+UNIT-COMP2+LUNITS}]}$$

Nous n'avons présenté que les règles de liaison concernant les modules, et il faut en faire de même pour la déclaration des variables dans le cas des types anonymes et pour les déclarations de ports. Ces règles ne posent aucune difficulté supplémentaire et suivent le même principe.

8 Exemple d'application des règles de compilation

La spécification que nous venons de présenter permet de formaliser la compilation des systèmes Lesp en Ada. Elle peut également constituer un programme TYPOL du système CENTAUR réalisant le compilateur. Bien que nous n'ayons pas approfondi cette possibilité, nous présentons toutefois un exemple de ce que pourrait être une interprétation des règles que nous avons spécifiées. Nous allons examiner comment obtenir un processus Ada à partir d'un processus élémentaire Lesp en appliquant les règles d'inférence sur le processus prod suivant:

```
process prod ::
-SPECIF: déclaration des entrées
et des sorties
 sortie s : entier vers p;
-CORPS: partie corps
 debut
 .....;
 produire(s);
 .....;
fin prod;
```

La représentation arborescente de ce processus est la suivante:

process("prod",**specif**(**entres**[],**sortie** ("s","entier","p")),**corps**(**produire**("s"))).

La compilation de ce processus est activée sur l'arbre abstrait donné ci-dessus par la règle suivante:

$$\frac{\text{ENV} \overset{\textbf{c-port}}{\vdash} \text{SPECIF,IDENT : ENV'} \quad \text{ENV} \vdash \text{SPECIF} \rightarrow \text{DVARS} \quad \text{ENV'} \vdash \text{CORPS} \rightarrow \text{DEC-PART,SEQ-STAT}}{\text{ENV} \vdash \textbf{process}(\textbf{ident}\ \text{p,SPECIF,CORPS}) \rightarrow \textbf{task-spec}(\textbf{ident}\ \text{p,()}), \textbf{task-body}(\textbf{ident}\ \text{p,DEC-PART+DVARS,SEQ-STAT})}$$

Cette activation a pour but de prouver le prédicat spécifié dans la partie basse de la règle en utilisant les prémisses qui en constituent la partie haute comme des hypothèses. Chaque prémisse est à son tour prouvée en choisissant la règle dont la partie basse commence par un opérateur Lesp pouvant s'unifier avec la tête de cette prémisse. Nous commençons donc par la preuve des trois prémisses de la règle précédente.

8.1 Preuve des prémisses

1. La prémisse ENV $\overset{c-port}{\vdash}$ SPECIF,IDENT: ENV'

SPECIF s'unifie avec le sous arbre **specif(entres[],sortie**("s","entier","p")), deuxième fils de l'arbre **process**.

La preuve de cette prémisse va se faire en utilisant les règles du module **c-port**. C'est la première règle de ce module qui sera activée (car sa conclusion s'unifie avec l'arbre **specif**). Rappelons la spécification de cette règle:

$$\frac{\text{ENV} \vdash \text{ENTRES,IDENT} : \text{ENV'} \quad \text{ENV'} \vdash \text{SORTIES,IDENT} : \text{ENV"}}{\text{ENV} \vdash \textbf{specif}(\text{ENTRES,SORTIES}),\text{IDENT} : \text{ENV"}}$$

Nous allons maintenant prouver chaque prémisse.

- la prémisse ENV ⊢ ENTRES,IDENT : ENV'
  ENTRES s'unifie avec le sous arbre **entres[]**, premier fils de l'arbre **specif**.

  Cette prémisse va activer l'axiome "ENV ⊢ **entres[]**,IDENT : ENV" du module **c-port** sur cet arbre. Cet axiome spécifie que l'environnement reste inchangé lorsqu'il s'agit d'une liste de déclarations d'entrée qui est vide. Il en résulte que l'environnement ENV' de la prémisse à prouver est l'environnement ENV donné en tête de cette prémisse.

- la prémisse ENV' ⊢ SORTIES,IDENT : ENV"
  SORTIES s'unifie avec l'arbre **sortie**("s","p","entier"), deuxième fils de l'arbre **specif**.

  Cette prémisse va activer la règle (7) du module **c-port** que nous rappelons ci-dessous:

$$\frac{\text{ENV} \vdash \text{NOMVARS,}\textbf{ident}\ \text{p} : \text{ENV'}}{\text{ENV} \vdash \textbf{sortie}(\text{NOMVARS,}\textbf{ident}\ \text{t,}\textbf{ident}\ \text{p}),\ \textbf{ident}\ \text{p} : \text{ENV'}}$$

  Notons que la conclusion de cette règle s'unifie avec l'arbre que nous avons à savoir **sortie**("s","p","entier"). Il faut de nouveau prouver la prémisse de cette règle sachant que NOMVARS s'unifie avec l'identificateur "s", et **ident** p avec l'identificateur "p". Pour cela nous appliquons l'axiome (10) du module **c-port** qui est le suivant:
  ENV ⊢ **ident** x,**ident** p : **env**{**port-assoc**(**ident** x,**ident** p)+ENV}
  Le résultat de l'application de cet axiome est l'adjonction de l'association **port-assoc**("s","p") à l'environnement existant.

Ceci termine la preuve de la prémisse **ENV** $\overset{c-port}{\vdash}$ **SPECIF,IDENT: ENV'**.
ENV' représente le nouvel environnement obtenu à partir de **ENV** en ajoutant l'association précédente.

2. La prémisse ENV ⊢ SPECIF → DVARS

SPECIF s'unifie avec l'arbre **specif(entres[], sortie**("s","entier","p")).

La preuve de cette prémisse va se faire en activant la règle suivante:

$$\frac{\text{ENV} \vdash \text{ENTRES} \rightarrow \text{DVARS1} \quad \text{ENV} \vdash \text{SORTIES} \rightarrow \text{DVARS2}}{\text{ENV} \vdash \textbf{specif}(\text{ENTRES,SORTIES}) \rightarrow \textbf{dvars}(\text{DVARS1+DVARS2})}$$

sur l'arbre précédent. Cette activation va consister en la preuve des deux prémisses de la règle:

- la prémisse ENV ⊢ ENTRES → DVARS1:
  ENTRES s'unifie avec le sous-arbre **entres[]**, premier fils de l'arbre **specif**. La preuve de cette prémisse active l'axiome suivant:
  ENV ⊢ **entres[]** → **dvars[]**
  Le résultat de cette activation produit l'arbre Ada **dvars[]** qui s'unifie avec DVARS1.

- la prémisse ENV ⊢ SORTIES → DVARS2:
  SORTIES s'unifie avec le sous arbre **sorties**("s","entier","p"), deuxième fils de l'arbre **specif**.

  Cette prémisse va activer l'axiome concernant la compilation des sorties que nous rappelons ci-dessous:
  ENV ⊢ **sortie**(NOMVARS,**ident** t,**ident** p) → **dvar**(NOMVARS,**ident** t)

  Cet axiome est activé sur le sous arbre **sortie**("s","p","entier") (car ce sous-arbre s'unifie avec la tête de cet axiome). Cette activation produit l'arbre Ada **dvar**("s","entier") sachant que NOMVARS s'unifie avec "s" et **ident** t avec "entier".

Ces deux prémisses étant prouvées, il ne reste plus qu' à instancier la conclusion de la règle (1) que nous rappelons ci-dessous:

ENV ⊢ **specif**(ENTRES,SORTIES) → **dvars**(DVARS1+DVARS2).

L'instantiation est faite en remplaçant DVARS1 par **dvars[]** et DVARS2 par **dvar**("s","entier"). Nous obtenons alors l'arbre Ada suivant: **dvars**(**dvar**("s","entier")).

qui traduit l'arbre **specif**(**entres[]**,**sorties**("s","entier","p")).

3. La prémisse ENV' ⊢ CORPS → DEC-PART,SEQ-STAT

CORPS s'unifie avec l'arbre **corps(decls[],instrs(produire**("s"))), troisième fils de l'arbre **process**.

La preuve de cette prémisse va se faire en activant la règle suivante:

$$\frac{\text{ENV} \vdash \text{DECLS} \rightarrow \text{DEC-PART} \quad \text{ENV} \vdash \text{INSTRS} \rightarrow \text{SEQ-STAT}}{\text{ENV} \vdash \textbf{corps}(\text{DECLS,INSTRS}) \rightarrow \textbf{dec-part , seq-stat}}$$

sur l'arbre précédent. Pour cela, il faut prouver chaque prémisse de cette règle:

- la prémisse ENV' ⊢ DECLS → DEC-PART
  DECLS s'unifie avec l'arbre **decls**[], premier fils de l'arbre **corps**.

  Cette prémisse active l'axiome ENV ⊢ **decls**[] → **dec-part**[].
  Le résultat de cette activation produit l'arbre Ada **dec-part**[] qui s'unifie avec DEC-PART.

- la prémisse ENV' ⊢ INSTRS → SEQ-STAT
  INSTRS s'unifie avec le sous-arbre **produire**("s"), deuxième fils de l'arbre **corps**.

  Cette prémisse va activer la règle **7.3** relative aux instructions de communication sur ce sous-arbre. Rappelons la spécification d'une telle règle:

$$\frac{\text{ENV} \overset{\text{r-port}}{\vdash} \textbf{ident s:ident p}}{\text{ENV} \vdash \textbf{produire}(\textbf{ident s}) \rightarrow \textbf{proc-call}(\textbf{ref-mod}(\textbf{ident p},\text{"produire"}),\textbf{ident s})}$$

  Cette règle comporte une seule prémisse : ENV $\overset{\text{r-port}}{\vdash}$ **ident** s:**ident** p
  La preuve de cette prémisse va se faire en utilisant le module **r-port**. Elle consiste à chercher dans l'environnement le nom du port associée à la sortie "s" qui s'unifie avec **ident** s. Rappelons que nous avons stocké le couple ("s","p") dans l'environnement. Par conséquent une telle preuve produit comme résultat l'identificateur "p" qui s'unifie avec **ident** p. Nous pouvons maintenant déduire la preuve de la conclusion qui produit l'arbre Ada suivant: **proc-call**(**ref-mod**("p","produire"),"s").

Ces deux prémisses étant prouvées, il ne reste plus qu' à instancier la conclusion de la règle (2) que nous rappelons ci-dessous:

ENV ⊢ **corps**(DECLS,INSTRS) → **dec-part,seq-stat**

L'instantiation produit les deux arbres Ada résultant de la preuve des deux prémisses précédentes:

dec-part[], qui est une instance de **dec-part**,

proc-call(**ref-mod**("p","produire"),"s") qui est une instance de **seq-stat**.

8.2 Preuve de la conclusion :

ENV ⊢ **process**(**ident** p,SPECIF,CORPS) → **task-spec**(**ident** p,()),
task-body(**ident** p,DEC-PART+DVARS,SEQ-STAT)

Nous venons de présenter la preuve des trois prémisses constituant la règle relative aux processus élémentaires. Nous pouvons maintenant déduire celle de sa conclusion par instanciation. Rappelons que cette règle a été activée sur l'arbre suivant:

process("prod",**specif**(**entres**[]**sortie** ("s","entier","p")),**corps**(**produire**("s")))

L'instanciation de la conclusion de cette règle produit les deux arbres Ada suivants:

task-spec("prod",()),

task-body("prod",**dvar**("s","integer"),**proc-call**(**ref-mod**("p","produire"),"s")).

Notons que ces deux arbres sont construits à partir de la preuve des trois prémisses que nous avons présentées précédemment. Leur décompilation produit le texte Ada suivant:

```
task prod;
task body prod is
-DEC-PART: partie déclarative
d'un module Ada
 s : integer;
-SEQ-STAT: partie instruction
 begin
 .....;
 p.produire(s);
 .....;
end prod;
```

9 Conclusion

Dans ce chapitre, nous nous sommes efforcés de définir formellement la compilation du langage Lesp vers le langage Ada à l'aide de la sémantique naturelle. La spécification obtenue est compacte et peut être exécutée par le système CENTAUR [BOR 87]. En effet, ce système permet d'interpréter les spécifications écrites dans le formalisme TYPOL. Cependant, l'apparition tardive de ce système (au début de la réalisation du logiciel

COMEDIE) ne nous a pas permis d'exploiter cette possibilité. Nous nous efforçons dans le chapitre consacré à la réalisation de mettre en évidence l'implantation d'une telle spécification à l'aide du système CEYX [HUL 83].

Partie III

Environnement de programmation

Au chapitre 7, nous présentons une structure d'accueil permettant de stocker des spécifications de types abstraits de données et leurs implantations en Ada. Nous nous attacherons particulièrement à l'intérêt de cette structure du point de vue de la réutilisation des types de communication.

Au chapitre 8, nous présentons le système de transformation, et sa place dans l'environnement COMEDIE. Nous montrons comment implanter en CEYX (environnement de réalisation) les règles de transformation et de compilation présentées dans la partie précédente. Nous cloturons ce chapitre par la présentation de quelques systèmes transformationnels.

Chapitre 7

Vers un système d'aide à la réutilisation des types de communication

1 Introduction

Nous avons utilisé dans le processus de représentation des types de communication une bibliothèque de types abstraits de données et de leurs implémentations en Ada. Une telle utilisation ne couvre qu'un des objectifs qui nous ont amenés à concevoir cette bibliothèque. En effet, notre objectif principal [KOU 90] est d'en faire une structure d'accueil permettant la réutilisation des types de communication. Le but étant de permettre à l'utilisateur de ne pas réécrire et développer des types ayant le même comportement que ceux de la bibliothèque. Ainsi, nous pouvons par là même étendre les fonctionnalités du système COMEDIE en fournissant parallèlement au mode construction déjà existant, un autre mode fondé sur la réutilisation.

Pour atteindre cet objectif, la bibliothèque doit fournir des moyens d'accès faciles à utiliser, simples et puissants. La facilité d'utilisation dépendra de son organisation, de l'outil qui la gère, et plus particulièrement des fonctions d'accès qu'elle fournit. Il arrive aussi parfois que le type que l'on veut réutiliser n'ait pas tout à fait le comportement souhaité. Une manière de le lui donner est d'appliquer une transformation. A cet égard, la bibliothèque ne doit pas être un organe passif; elle doit prendre en considération cette fonctionnalité en proposant des règles de transformation permettant de passer d'un type à un autre.

Ainsi au deuxième paragraphe, nous citons les inconvénients de l'approche bibliothèque telle qu'elle a été couramment utilisée pour stocker des unités de programmes en vue de leur réutilisation. Nous pallions à ces inconvénients en proposant au paragraphe 3 une structure plus riche fondée sur un ensemble de concepts : composant abstrait, composant concret,

et relation entre composants que nous examinons en détails au paragraphe 4. Enfin nous présentons au paragraphe 5 quelques primitives pour la gestion de cette structure.

2 Approche bibliothèque

La construction de bibliothèques de programmes prêts à l'emploi est pratiquée depuis longtemps, notamment au niveau des sous-programmes. Ces bibliothèques proposent un ensemble de fonctions ou procédures permettant de résoudre des problèmes précis dans des domaines d'application bien définis. Citons les plus connus : les domaines des statistiques, d'analyse numérique et de l'infographie.

Si cette approche a eu un succès dans certains domaines très particuliers, il n'en demeure pas moins que ses limites n'ont pas permis sa pratique dans d'autres :

- les sous-programmes ne peuvent suffire à constituer des composants logiciels généraux [MEY 86]. L'émergence d'un ensemble de concepts en génie logiciel tels que les types abstraits de données, les classes et les objets largement acceptés et employés dans les différentes phases de développement, font qu'un composant doit être organisé autour d'une structure de données et non pas seulement assurer une fonction unique,

- la pauvreté de la documentation sémantique (spécification) constitue une barrière empêchant une réutilisation massive et profitable. Il semble plus intéressant de se poser le problème de la réutilisation au niveau de la spécification. En effet, c'est au niveau le plus abstrait, qu'on définira l'ensemble des propriétés intéressantes d'un composant [PRO 82] [LIT 84],

- enfin, le caractère passif de ces bibliothèques qui fait que le composant ne peut être réutilisé qu'en tant que tel (réutilisation du code) sans la possibilité de le transformer pour l'adapter à de nouvelles applications.

La structure de ces bibliothèques repose généralement sur de simples fichiers avec des mécanismes d'accès fournis par les utilitaires du système hôte.

Afin de remédier globalement à l'ensemble de ces limites, nous introduisons dans notre bibliothèque non pas seulement l'implantation des types mais aussi leurs spécifications et les relations pouvant exister entre eux. Ces dernières vont permettre d'établir une sorte de hiérarchie entre les différents types permettant ainsi de faciliter leur recherche et par conséquent leur réutilisation.

3 Description sommaire

La bibliothèque de types peut être conceptuellement décomposée en deux sous-bibliothèques :

- la bibliothèque abstraite contient la spécification des types que nous appelons par la suite composant abstrait. Nous y distinguons deux sortes de composants : ceux définissant des structures de données (pile, file, etc), et ceux définissant les types de communication,

- la bibliothèque concrète définit une représentation de la bibliothèque abstraite, en ce sens qu'elle fait correspondre à chaque composant abstrait, une ou plusieurs implantations sous forme de paquetage Ada. Comme pour la bibliothèque abstraite, les composants sont classés selon l'abstraction qu'ils représentent (structure de données ou type de communication).

Pour permettre leurs utilisations dans des contextes variés, les composants sont paramétrés par des types de données et éventuellement les opérations que doivent posséder ces types.

Cette organisation en deux niveaux de description de composants (spécification, implantation) met en évidence une première relation "implanté par" entre les deux bibliothèques. D'autres relations seront présentées dans le but d'établir une hiérarchie entre composants.

Sans vouloir rappeler l'intérêt de la spécification dans le processus de développement de programmes, citons tout simplement les avantages de sa prise en compte dans tout système comprenant une bibliothèque de modules, et en particulier dans le cadre de cette étude :

- pour pouvoir réutiliser un composant paramétré, il est nécessaire de bien comprendre et respecter les propriétés que doivent posséder ses paramètres. Celles-ci fixent la classe de contextes dans laquelle le composant peut être réutilisé correctement [LIT 84]. Or les langages de programmation et en l'occurrence Ada, ne permettent pas d'exprimer de telles propriétés. Il devient alors nécessaire de les exprimer en terme d'un langage de spécification. Citons au passage la notion de théorie offerte par le langage OBJ [FUT 87] comme solution à ce problème,

- pour une même abstraction, on peut associer plusieurs réalisations. Il est alors intéressant de regrouper dans une même entité (la spécification) leur caractéristique commune.

- L'indépendance vis à vis d'un langage de programmation; on peut associer plusieurs bibliothèques concrètes à une même bibliothèque abstraite. Chacune est exprimée dans un langage de programmation donné.

Nous présentons maintenant les trois concepts que nous avons mis en évidence précédemment à savoir : composant abstrait, composant concret et relation.

4 Concepts de base

La bibliothèque repose sur un petit nombre de concepts : composant abstrait, composant concret et relation entre composants.

4.1 Composant abstrait

Un tel composant définit la spécification formelle d'une abstraction donnée sans faire référence à une implantation ou à une représentation particulière. Nous nous limitons ici à deux sortes d'abstractions :

- Les structures de données spécifiées à l'aide des types abstraits algébriques : composants structures,
- les types de communication spécifiés sous forme de types abstraits avec des pré et post conditions : composants communications.

La définition d'un composant abstrait est constituée de quatre rubriques :

- **générique** indique les paramètres du composant en précisant pour chaque paramètre les propriétés que doit posséder le paramètre effectif correspondant lors d'une instantiation,
- **import** précise la liste des entités importées par le composant,
- **interface** mentionne les opérations du type avec leur signature. Dans le cas d'un composant structuré nous séparons les opérations en trois sous-rubriques : constructeurs, modificateurs, observateurs.
- **propriétés** définit la partie sémantique du composant contrairement aux rubriques précédentes qui décrivent sa partie syntaxique. Ces propriétés sont décrites sous forme d'équations pour un composant structuré, et sous forme de pré et post conditions pour un composant communication.

Les rubriques définissent des points d'accès aux différentes parties d'un composant.

4.2 Composant concret

A chaque composant abstrait est associé une ou plusieurs implantations exprimées sous forme de modules Ada. Nous appelons composants concrets de tels modules. Ces

implantations diffèrent essentiellement par leurs caractéristiques internes : représentation en mémoire et complexité des algorithmes implantant les opérations du composant. Le langage Ada offre un ensemble de facilités pour la programmation de composants logiciels : modules paramètrés, séparation entre l'interface d'un module et son implantation, compilation séparée, etc. Pour une étude plus détaillée sur la programmation de composants logiciels en Ada, on peut se rapporter à [BOO 87]. Conformément à la structuration des composants sous forme de rubriques, un composant concret est composé de cinq rubriques. Chacune est considérée comme une implantation d'une rubrique abstraite :

- **import** mentionne le nom des composants importés,
- **générique** contient la déclaration des types paramètres et éventuellement les opérations accompagnant ces types. Pour les composants structures, le type paramètre n'est muni d'aucune opération. Cependant, il faut au minimum autoriser l'affectation et le test d'égalité entre les objets de ce type. Ces deux opérations sont les seules nécessaires pour l'implantation des opérations du composant. Ceci nous amène à utiliser la forme "private" pour la déclaration du type paramètre.

  Par ailleurs, il existe certain composant communication où le type paramètre doit posséder certaines opérations. La rubrique générique doit alors les mentionner sous forme de spécification de sous-programmes Ada.
- **interface** associe à chaque opération du composant abstrait, une spécification de sous-programme.
- **représentation** définit pour un composant sa représentation en termes de types de données du langage Ada.
- **corps** définit l'implantation des sous-programmes mentionnées dans l'interface. Cette rubrique est considérée comme l'implantation de la rubrique **propriété** d'un composant abstrait.

Nous donnons ci-dessous l'une des implantations du composant file à double accès à l'aide d'une représentation chaînée :

```
generic
  type elt is private;
package filed_v is
  type file_da is private;
  procedure creer (fd : out file_da);
  procedure ajoutert (fd : in out file_da; e : in elt);
  procedure ajouterq (fd : in out file_da; e : in elt);
  procedure retirert (fd : in out file_da);
  procedure retirerq (fd : in out file_da);
  function extrairet (fd : in file_da) return elt;
  function extraireq (fd : in file_da) return elt;
  function vide (fd : in file_da) return boolean;
  function card (fd : in file_da) return natural;
  fd_vide : exception;
private
  type element;
  type lien is access element;
  type element is record
                    val : elt;
                    suivant : lien;
                    end record;
  type file_da is record
                  nbelem : natural;
                  tete,queue : lien;
                  end record;
end filed_v;
```

Nous présentons dans le tableau suivant le corps de ce paquetage.

```
package body filed_v is
  procedure creer (fd : out file_da) is
  begin
  fd:=(nbelem=>0,tete=>null,queue=>null);
  end creer;
  procedure ajoutert (fd : in out file_da; e : in elt) is
  begin
  if vide(fd) then fd.tete:=new element '(e,null);
                   fd.queue:=fd.tete;
              else fd.tete:=new element'(e,fd.tete);
  end if;
  fd.nbelem:=fd.nbelem+1;
  end ajoutert;
```



```
procedure ajouterq (fd : in out file_da; e : in elt) is
begin
if vide(fd) then fd.tete:=new element '(e,null);
                 fd.queue:=fd.tete;
            else fd.queue.suivant := new element'(e,null);
                 fd.queue :=fd.queue.suivant;
end if; fd.nbelem:=fd.nbelem+1;
end ajouterq;
procedure retirert (fd : in out file_da) is
begin
if vide(fd) then raise fd_vide;
           else if fd.tete=fd.queue then fd.tete:=null; fd.queue:= null;
                                    else fd.tete:=fd.tete.suivant; end if;
end if; fd.nbelem:=fd.nbelem-1;
end retirert;
procedure retirerq (fd : in out file_da) is
p:lien:=fd.tete;
begin
if vide(fd) then raise fd_vide;
           else if fd.tete=fd.queue then fd.tete:=null; fd.queue:= null;
                                    else while p.suivant /= fd.queue loop
                                         p:=p.suivant;
                                         end loup;
                                         fd.queue:=p.; fd.queue.suivant:=null;
           end if;
end if; fd.nbelem:=fd.nbelem-1;
end retirerq;
function extrairet (fd : in file_da) return elt is
begin
if vide(fd) then raise fd_vide;
           else return fd.tete.val;
end if;
end extrairet;
function extraireq (fd : in file_da) return elt is
begin
if vide(fd) then raise fd_vide;
           else return fd.queue.val;
end if;
end extraireq;
function vide (fd : in file_da) return boolean is
begin
return fd.tete=null;
end vide;
function card (fd : in file_da) return natural is
begin
return fd.nbelem;
end card;
end filed_b;
```

Pour ne pas surcharger l'exemple, nous avons supprimé volontairement les commentaires de même que les informations concernant les caractéristiques internes du composant. Signalons toutefois que ces informations sont très utiles pour choisir entre des composants concrets implantant la même abstraction.

4.3 Relations

Les études sur les systèmes d'aide au développement de logiciels ont mis en évidence l'existence de relations entre les différents constituants d'une application [MEY 85] [KHA 90]. Ainsi pour mieux définir et exploiter le concept de relation, certains systèmes ont eu recours aux techniques de bases de données tels le modèle entité-association.

En conservant les relations entre les constituants d'une application, l'objectif est de garantir sa cohérence au fur et à mesure des modifications introduites par le processus de développement. Ce concept de relation se transporte naturellement dans le cadre de notre bibliothèque. Elle permet non seulement de structurer l'ensemble des composants, mais aussi d'introduire une possibilité d'accès supplémentaire.

Relation d'importation

Cette relation exprime le fait qu'un composant A utilise des ressources (type, opérations) fournies par un composant B que l'on note A **import** B. Comme exemple, tout composant abstrait de communication est lié par la relation d'importation au composant définissant le type "table". Cette relation est déduite des rubriques d'importation décrites dans la définition des composants. L'intérêt d'une telle relation est qu'elle permet de déterminer les conséquences d'une modification portant sur un composant et impose par là même un ordre pour la compilation des composants concrets.

Relation de généralisation

Un composant A est une généralisation d'un composant B que l'on note A **généralise** B, si l'ensemble des objets décrits par A contient l'ensemble de ceux décrits par B. De manière opposée, on peut définir B comme une spécialisation de A, que l'on note A **spécialise** B.

considérons les deux exemples suivants pour illustrer ces deux relations :

- le composant "liste triée", paramétré par le type de ses éléments et la relation d'ordre associée est une spécialisation du composant "liste", paramétrée par le type de ses éléments. Il s'agit ici d'une spécialisation par adjonction de propriétés propres au composant spécialisé.

- le composant "liste d'entiers" est une spécialisation du composant "liste" paramétré par le type de ses éléments. Il s'agit dans ce cas d'une spécialisation par instantiation du composant le plus général.

Généralisation et spécialisation sont deux formes d'abstraction couramment utilisées dans la représentation des connaissances en intelligence artificielle et dans les bases de données [HUL 87]. Ces deux formes se manifestent aussi dans les langages de programmation : la paramétrisation des modules, le sous-typage en Ada, et l'héritage dans les langages orientés objets en sont des exemples.

Relation d'enrichissement

Un composant A est un enrichissement d'un composant B que l'on note A **enrichi** B si A peut être construit à partir de B par ajout de nouvelles opérations. Nous appelons **restreint** la relation inverse.

Cette relation illustre un point de vue incrémental dans la construction des composants. Considérons quelques exemples de composants structurés bien connus pour illustrer ces deux relations :

le composant "file-double" est un enrichissement des composants "file" et "pile". De même, il est une restriction du composant liste.

L'enrichissement permet de construire des composants offrant plus de fonctionnalités à partir de composants simples. Il constitue de ce fait l'un des moyens permettant d'adapter un composant à certaines utilisations différentes de celles pour lesquelles il a été construit.

Relation d'implémentation

Cette relation exprime le fait qu'un composant concret A est une implantation d'un composant abstrait, que l'on note A **implémente** B. Cette relation définit le lien entre la spécification d'une abstraction et ses différentes implantations en Ada. Nous appelons **spécifie** la relation inverse.

Relations entre composants communication

Nous introduisons [PER 85] deux relations permettant d'une part de comparer les composants communication selon leur degré d'asynchronisme, d'autre part d'établir une hiérarchie de tels composants. Ces relations sont telles que si, pour une donnée communiquée on substitue un composant de communication tc1 à un composant communication tc2, tels que tc1 et tc2 soient ordonnés dans cet ordre par l'une de ces relations, le sys-

tème devient plus "asynchrone" au sens suivant : les situations d'attente du processus consommateur sont plus rares.

Bien que nous définissions ces relations au niveau des composants abstraits, elles sont héritées par les composants concrets correspondants.

Nous notons tc_n l'état du composant communication tc après une séquence de longueur n d'activations des opérations de production et de consommation. Bien évidemment, l'activation d'une opération de consommation n'est possible que si sa pré-condition "pre-cons" a pour résultat "vrai".

- **Relation $<_{as}$**

  Soient $tc1$ et $tc2$ deux composants communication. $tc2$ est dit plus asynchrone que $tc1$ que l'on note $tc1 <_{as} tc2$ si et seulement si pour tout n :

  $$\text{pre-cons}(tc1_n) \Rightarrow \text{pre-cons}(tc2_n)$$
  $$\text{ind-cons}(tc1_n) \leq \text{ind-cons}(tc2_n)$$

  où ind-cons(tc_n) est l'indice dans l'ensemble consommable de la valeur prise en compte par l'opération de consommation. Cette opération est propre à chaque type de communication.

  Notons que l'ensemble des valeurs consommées n'est pas nécessairement le même pour les deux types de communication contrairement à la relation suivante.

- **Relation $<_{op}$**

  Deux composants de communication $tc1$ et $tc2$ sont en relation $<_{op}$ que l'on note $tc1 <_{op} tc2$ si et seulement si pour tout n :

  $$\text{pre-cons}(tc1_n) \Rightarrow \text{pre-cons}(tc2_n)$$
  $$\text{ind-cons}(tc1_n) = \text{ind-cons}(tc2_n)$$

Propriété

Les relations $<_{op}$ et $<_{as}$ sont des relations d'ordre partiel sur l'ensemble des composants communication, telle que :

$$tc1 <_{op} tc2 \Rightarrow tc1 <_{as} tc2$$

Lorsqu'on substitue un type de communication $tc1$ à un type $tc2$ tel que $tc1 <_{as} tc2$, on obtient un programme "plus asynchrone" au sens suivant : toutes les valeurs produites

ne sont pas nécessairement consommées et les situations d'attente du processus consommateur sont plus rares. Cependant, le programme obtenu après cette substitution n'est pas défini sur les mêmes ensembles de données contrairement à la relation $<_{op}$.

Exemple

Considérons les deux types de communication Egalite et Aléatoire dont la spécification des opérations caractéristiques est donnée ci-dessous :

type Egalite	**type** Aleatoire
pre-cons(<SP,EA,SC>)=non t-vide(EA)	pre-cons(<SP,EA,SC>)=non t-vide(EA)
val-cons(<SP,EA,SC>)=extraire(EA,min(EA))	val-cons(<SP,EA,SC>)=extraire(EA,max(EA))
post-cons(<SP,EA,SC>)=retirer(EA,min(EA))	post-cons(< SP,EA,SC >)=finir(EA)

L'opération "finir" supprime tout les éléments d'indice inférieur à l'élément d'indice maximal de la table EA. L'opération ind-cons est définie de la manière suivante :

- pour le type égalité : ind-cons(< SP,EA,SC >) = min(EA),
- pour le type aléatoire : ind-cons(< SP,EA,SC >) = max(EA),

On a alors Egalite $<_{as}$ Aleatoire

On trouve dans [PER 85] des exemples de transformation de programmes parallèles par substitution d'un type de communication par un autre liés par l'une des relations. Par ailleurs, le système PAUSE [LAZ 89] permet de prouver l'existence de telles relations entre des types de communication.

5 Les outils

Dans ce qui précéde, nous nous sommes consacrés à la définition des composants et de leurs relations. Cependant, leur exploitation ne peut être assurée de manière effective que si elle est accompagnée d'un ensemble de primitives ou de fonctions d'aide. Ces primitives définissent les services que l'on peut demander à un gestionnaire de composants.

Compte tenu de notre intérêt pour les types de communication, qui constituent le sujet de cette étude, nous nous limitons à la gestion de tels composants. Toutefois les primitives que nous présenterons peuvent être étendues aux composants structure en suivant le même principe.

Nous avons retenu dans notre système les primitives suivantes :

- la création et le stokage des composants et de leurs relations,

- la recherche et la localisation de composants répondant à un besoin donné.

5.1 Création et stockage

La création d'un composant consiste à préciser toutes les rubriques qui le composent. L'outil le plus attendu pour accomplir cette tâche est un éditeur syntaxique. Pour cela, nous disposons de l'éditeur fourni par l'environnement COMEDIE, permettant de créer la spécification des composants communication. Le système de dérivation se charge de la construction de l'implémentation Ada correspondante.

Une fois le composant crée, on peut le stocker. On doit pour cela préciser ses relations éventuelles avec les composants existants. La définition des relations d'ordre entre les composants communication peut se faire de manière automatique par l'intermédiaire du système Pause [LAZ 89]. En effet ce système permet de prouver l'existence de telles relations entre deux composants communication en utilisant uniquement leurs spécifications. Les relations **implémente** et **import** sont déduites syntaxiquement. Quant aux autres relations, elles sont actuellement à la charge de l'utilisateur et doivent faire l'objet d'une étude approfondie.

5.2 Recherche et localisation

La recherche et par conséquent la localisation d'un composant peuvent se voir de deux facons :

- en éditant la liste des noms de composants existants et les commentaires associés à chacun d'eux, on peut explorer une telle liste afin de localiser le ou les composants répondant à un besoin donné. A cet égard, l'exploration de la hiérarchie établie par l'une des relations permet d'accélérer la recherche,
- étant donnée la spécification d'un type de communication, on cherche à localiser le ou les composants liés à cette spécification par l'intermédiaire d'une relation donnée. Lorsqu'il s'agit de l'une des relations d'ordre définies sur les composants communication, cette recherche peut se faire de manière automatique par le système PAUSE. Les composants ainsi localisés peuvent être candidats à se substituer au type de communication de départ dans un programme, en vue d'obtenir une solution plus asynchrone.

Bien évidemment, nous n'avons pas signalés toute les techniques de recherche que l'on peut envisager. A ce sujet, le système SPRAC [FOI 85] utilise une recherche fondée sur des questions prédéfinies incluant des mots-clés et des synonymes. L'inconvénient de cette manière de procéder est qu'on doit se familiariser avec le vocabulaire utilisé et

avec sa signification, sauf si on se place dans un domaine d'application très précis. Une autre approche consiste à définir des critères permettant de comparer les composants. Un exemple est donné dans [PRO 82], où le critère de comparaison est fondée sur la notion d'équivalence au nom près de types abstraits de données. Enfin, Wood, dans [MUR 88], propose une approche fondée sur les techniques utilisées dans le domaine du langage naturel pour représenter d'une part les informations concernant les composants et d'autre la manière de les localiser à partir de telles informations.

6 Conclusion

A la lumière de cette étude et des travaux sur le domaine de la réutilisation, il nous paraît ainsi se dégager les principes suivants :

- la nécessité de conserver non seulement l'implémentation des composants mais aussi leurs spécifications. Outre les raisons que nous avons signalées à ce sujet, vient s'ajouter la possibilité d'établir des preuves et d'envisager des transformations de composants en vue de les adapter à de nouvelles situations. Ces deux activités sont reconnues difficilement envisageables sur un composant codé dans un langage de programmation [CHE 84]. Dans ce contexte, Goguen [GOG 86] propose un langage incluant des opérateurs permettant de combiner des spécifications de composants pour en construire d'autres,
- l'importance de la notion de relation qui doit capturer non seulement des liens de nature syntaxique entre composants (relation d'importation par exemple) mais aussi de nature sémantique (relations d'enrichissement, de généralisation, etc),
- enfin, un autre point intéressant du point de vue de la réutilisation, et que nous n'avons pas abordé dans cette étude, concerne la conservation de l'historique et des choix de conception ayant conduit à la réalisation du composant [GRE 88].

Tel qu'il est utilisable actuellement, le système offre un environnement restreint par les primitives disponibles mais relativement complet par sa structure. Il importe de bien signaler que, pour qu'un tel système soit vraiment utile, il ne doit pas être isolé mais s'intégrer dans un environnement d'aide au développement de programmes. Le système peut alors être considéré comme une composante permettant de mettre en œuvre la tactique de réutilisation [GRA 86] au sein de l'environnement global.

Chapitre 8

Environnement de programmation COMEDIE

Le but de ce chapitre est de présenter le logiciel COMEDIE-ADA; un système de dérivation de programmes Ada à partir d'énoncés exprimés en Lesp. Construit autour des idées présentées dans les chapitres précédents, ce logiciel vient compléter l'environnement de programmation COMEDIE [JUL 85] qui est composé d'un éditeur syntaxique et d'un interprète de programmes Lesp.

Nous présentons dans un premier temps le système CEYX que nous avons utilisé à la base de notre réalisation. Ensuite nous donnons l'architecture générale de l'environnement COMEDIE, après quoi nous décrivons les différentes composantes du système COMEDIE-ADA.

1 Environnement de développement

1.1 Critères de choix

Lorsque l'on réalise un logiciel, il est pratique de réutiliser au maximum des environnements existants en ajoutant éventuellement des couches supplémentaires pour effectuer des fonctions supplémentaires. Il se pose alors le problème de choix d'un tel environnement.

Les critères à prendre en compte pour guider ce choix résident essentiellement dans la nature des objets que l'on désire manipuler et les traitements associés. Ajouté à cela les possibilités offertes par l'environnement de développement lui même : disponibilité d'un ensemble de constructions adéquats et facilement extensibles.

Les différentes composantes de COMEDIE-ADA que nous avons identifiées dans les chapitres précédents manipulent des textes de programmes ou plutôt leur représentation abstraite : arbre de syntaxe abstraite. Quant aux traitements, ils consistent généralement en des opérations de construction, de consultation et de parcours d'arbres.

Par ailleurs, signalons qu'au moment du lancement du projet COMEDIE, seuls les environnements MENTOR et CEYX étaient disponibles. La possibilité d'accéder immédiatement aux structures et primitives offertes par CEYX, leurs adéquations à la manipulation, la construction d'arbres ont conduit aux choix de cet environnement. De plus la programmation orientée objet supporté par CEYX permet de profiter de toutes les qualités de ce style de programmation. En effet on peut attacher aux opérateurs de la syntaxe abstraite des fonctions sémantiques (appelées aussi méthodes dans les langages orientés objets), par exemple d'affichage, de compilation, ce qui permet de vérifier de manière incrémental le résultat obtenu et facilite donc la mise au point.

1.2 L'environnement CEYX

CEYX a été développé à l'INRIA comme une aide à la conception de circuits VLSI [HUL 83]. Sur-système du langage Lisp, Ceyx offre à l'utilisateur un ensemble de fonctions Lisp permettant de créer et de manipuler des objets structurés. Ces objets résultent de la combinaison d'une structure de données avec un ensemble de propriétés sémantiques.

Les objets sont regroupés en famille de manière hiérarchique de sorte qu'ils héritent les propriétés de leurs ancêtres. Un style de programmation orienté objet comme dans SMALLTALK [GLO 83] est ainsi possible.

Pour implanter la syntaxe abstraite d'un langage, CEYX offre deux structures: deftree et defcons représentant respectivement les concepts de phyla et d'opérateurs que nous avons introduit pour définir la syntaxe abstraite.

- **deftree**

  cette construction permet de définir un univers d'arbres (phylum) éventuellement hiérarchisé. On peut lui associer des attributs sémantiques qui seront partagés par tous les arbres issus de cet univers.

- **defcons**

  Cette construction définit les opérateurs noeuds des arbres abstraits que l'on peut construire à partir de la grammaire abstraite d'un langage. Elle permet de spécifier comment sont structurés les fils de l'arbre auquel est rattaché un opérateur.

Par exemple, l'opérateur opack-body (corps d'un paquetage Ada) spécifié dans la syntaxe abstraite de la manière suivante:

pack-body : IDENT × DEC-PART × SEQ-STAT × EXCEP-HAND → PROPER-BODY

est implanté en CEYX par la construction **defcons** comme suite:


```
(defcons {PROPER-BODY}: pack-body
                        sons(vector IDENT DEC-PART
                                    SEQ-STAT EXCEP-HAND)
```

PROPER-BODY, phylum de l'opérateur **pack-body** est définit par la construction **deftree**:

(**deftree** {UNIT} : PROPER-BODY), où UNIT est le phylum qui regroupe tout les phyla décrivant les unités de programme Ada.

L'opérateur **pack-body** joue deux rôles: il définit d'une part un modèle d'arbre, d'autre part une fonction permettant de construire des instances de ce modèle. Comme il s'agit ici d'un opérateur d'arité fixe, on introduit le type (ou modèle) de base vector. Ce type représente l'opérateur **pack-body** par un tableau à quatre éléments dont les phylum sont respectivement IDENT, DEC-PART, SEQ-STAT et EXCEP-HAND.

L'utilisation des deux constructions **deftreer** et **defcons** permet de disposer de l'ensemble des fonctions de construction et de manipulation des arbres générés automatiquement par CEYX. On trouvera aux annexes A et B les syntaxes abstraites des langages Lesp et Ada, ainsi que leurs implantations en CEYX.

2 Architecture générale de l'environnement COMEDIE

COMEDIE est conçu comme un environnement de programmation, intégrant un ensemble d'outils qui utilisent et travaillent sur une structure de données commune : l'arbre de syntaxe abstraite.

Le noyau de l'environnement est bâti autour d'un éditeur syntaxique appelé COMEDIE. Les programmes construits avec cet éditeur sont écrits en Lesp. Lorsqu'un programme Lesp a été conçu à l'aide de COMEDIE, il convient en général de le mettre au point. Pour cela, l'environnement offre la possibilité de l'interpréter [MAR 89]. Enfin et c'est là ou se situe notre contribution sur le plan de la réalisation en offrant la possibilité de produire des programmes Ada par le système COMEDIE-ADA

Ces trois outils permettent de structurer l'environnement en trois modes distincts comme indiqué sur la figure 8.1.

Dans la suite, nous présentons brièvement l'éditeur syntaxique COMEDIE, et étudions plus en détails les modules fonctionnels composant le compilateur.

3 L' éditeur syntaxique COMEDIE

Cet éditeur est dirigé par la syntaxe du langage Lesp, et constitue le noyau de l'environnement. Nous ne reviendrons pas ici en détail sur l'utilité d'un tel outil dans le cadre de développe-

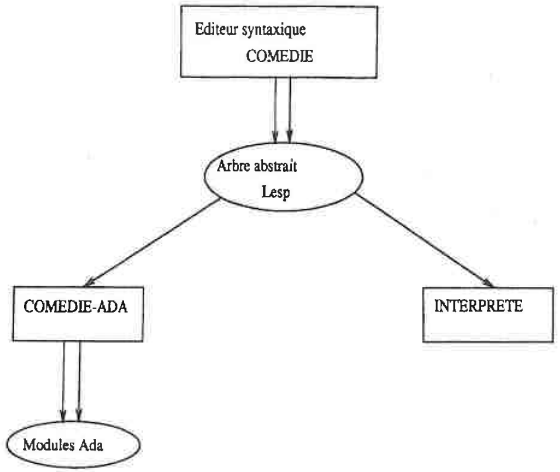


Figure 8.1: architecture générale de l'environnement COMEDIE

ment de programmes. Rappelons toutefois que souvent utilisés comme noyau d'environnement de programmation, les éditeurs syntaxiques par opposition aux éditeurs de textes classiques permettent de manipuler des documents en déchargeant l'utilisateur d'une partie des tâches de routine liées à la nécessité de fournir tous les détails de la syntaxe concrète des documents. Outre l'aspect syntaxique, l'éditeur opère un certain nombre de contrôles liés à la sémantique statique des programmes (vérification de type, visibilité des objets ...). Par ailleurs, l'utilisation par l'éditeur d'une structure de données normalisée (l'arbre de syntaxe abstraite) lui permet de servir de support à d'autres outils logiciels. Dans notre cas, l'interprétation et la compilation vers Ada.

Le développement d'un programme sous l'éditeur **COMEDIE** s'effectue essentiellement au moyen d'une opération de création de squelettes ou de parties de programmes. La partie crée est automatiquement insérée dans le programme de deux manières : soit au plus près de la partie courante du programme, soit à la place d'une partie abstraite non encore définie. Nous ne détaillons pas les différentes fonctions du système, nous nous contentons d'un exemple simplifié de développement d'un programme sous l'éditeur.

```
?(creeprogram)
      nom du program: prod-cons
  processus prod-cons
      port $port;
```



```
      $processus;
      $prog;
      $typcom
      composition
fin prod-cons
```

Etant donné ce squelette de programme, on peut créer un processus, un programme, un type de communication ou un port de communication permettant de connecter des processus.

4 Le système COMEDIE-ADA

Le système COMEDIE-ADA est un des composants essentiels de l'environnement COMEDIE. Il a pour rôle de construire pour un système parallèle Lesp le programme Ada qui lui correspond tout en respectant les règles de traduction et de transformation spécifiées dans la partie 2.

4.1 Architecture du système

Comme nous l'avons vu, certaines construction du langage Lesp peuvent être traduites directement en Ada, alors que d'autres en particulier celles relatives aux types de communication nécessitent des étapes intermédiaires. Ces étapes font intervenir des règles de transformation de programmes.

Pour cela, outre le traducteur et le décompilateur qui sont les composantes classiques d'un compilateur, le système COMEDIE-ADA inclut une base de connaissance et des outils opérant sur cette base. La base de connaissance est composée d'un ensemble de règles de transformation et de choix de représentation, ainsi qu'une bibliothèque de types abstraits de données et de leurs implantations en Ada.

L'organisation générale de ces différentes composantes est donnée par la figure 8.2.

Ce schéma montre les modules fonctionnels :

- Le pilote est le module principal. Il assure l'enchainement des autres modules du système.
- Le traducteur qui construit pour un arbre de la syntaxe abstraite du langage Lesp, l'arbre correspondant dans la syntaxe abstraite du langage Ada.
- Le décompilateur qui associe à chaque arbre de la syntaxe abstraite Ada, sa représentation textuelle.

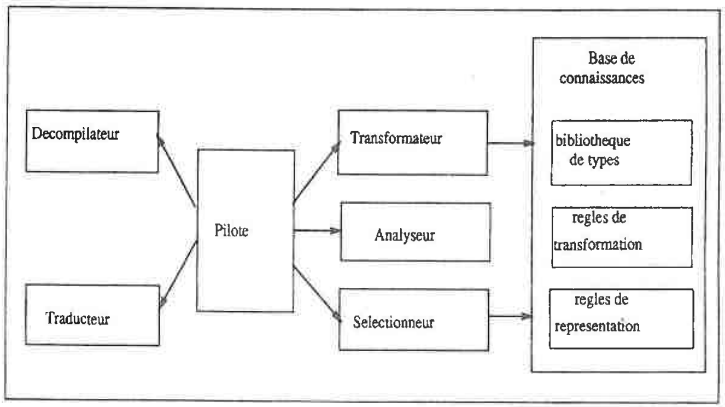


Figure 8.2: architecture générale du système COMEDIE-Ada

- L'analyseur permet d'extraire de la spécification d'un type de communication, les informations nécessaires à l'étape de représentation.
- Le transformateur qui réalise la simplification d'un type de communication et la représentation des composantes d'histoire.
- Le sélectionneur qui choisi dans la bibliothèque une représentation pour l'ensemble consommable.

Ces deux derniers modules utilisent les règles de transformation, les règles de choix de représentation, et la bibliothèque décrite au chapitre précédent.

4.2 Les connaissances du système

Les connaissances du système regroupent l'ensemble des informations utilisées dans le processus de transformation des types de communication. Ces connaissances sont constituées:

- d'un catalogue de règles de transformation,
- d'une bibliothèque de types abstraits de données et de leurs implantations en Ada,
- d'un catalogue de règles de choix de représentation.

La bibliothèque des types ayant fait l'objet du chapitre précédent, nous nous contentons de présenter dans la suite les catalogues des règles de transformation et de représentation.

Catalogue des règles de transformation

Nous regroupons dans ce catalogue l'ensemble des règles de transformation globales et locales. Rappelons que de telles règles ont été introduites dans le but de construire une représentation des variables d'histoire.

Une règle est représentée par un arbre ayant la forme 8.3:

Un arbre d'une règle est représenté en CEYX par les constructions **deftree** et **defcons**. Ainsi toutes les règles sont construites à partir de l'opérateur "regle" définit de la manière suivante:

(**defcons** {u-regle}: regle sons(vector schema cond conseq))

u-regle est l'univers à partir duquel sont définies toutes les règles et les opérateurs qui interviennent dans leurs définitions:

(**deftree** u-regle (nom string)), où nom est une chaine de caractères qui peut être le nom d'une règle, où un commentaire.

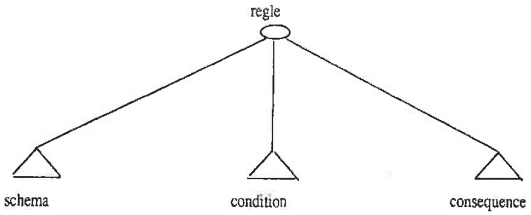


Figure 8.3: représentation arborescente d'une règle

4.3 Le module analyseur

Le système utilise un ensemble d'informations caractérisant la spécification d'un type de communication. Ces informations sont extraites de l'arbre abstrait d'un type de communication par le module analyseur et utilisées par le transformateur et le sélectionneur.

Deux listes sont construites ainsi par l'analyseur:

- la liste des opérations externes appliquées sur les variables d'histoire.
- la liste des caractéristiques relatives aux opérations appliquées sur l'ensemble consommable. Cette liste va permettre au module sélectionneur de choisir une représentation à cet ensemble en bibliothèque.

L'analyseur est implanté par une méthode **analyser** que nous avons associée aux opérateurs de la syntaxe abstraite des types de communication.

4.4 Le module transformateur

Le transformateur a pour but de construire le résultat de l'application d'une règle de transformation sur une opération manipulant une variable d'histoire. Pour cela, il procède de la manière suivante:

- choix d'une règle de transformation : étant donnée une opération *op*, il s'agit d'identifier la règle de transformation dont le schéma *filtre op* .
- instantiation de la conséquence de la règle : cet instantiation est guidée par l'identification (ou le filtre) établie entre le schéma de la règle et l'opération *op*.

Ainsi le résultat du module transformateur est construit tout simplement par instantiation de la partie conséquence de la règle choisie.

4.5 Le compilateur

Le compilateur opère à partir de l'arbre abstrait Lesp fourni par l'éditeur syntaxique COMEDIE. Cet arbre est traduit en un arbre abstrait Ada durant la phase de traduction. La forme textuelle est fournie ensuite par la phase de décompilation.

le module traducteur

Le rôle du traducteur est d'associer à chaque opérateur de la syntaxe abstraite Lesp, l'opérateur de la syntaxe abstraite Ada qui lui correspond.

Le traducteur est implanté par une méthode **traduire** que nous avons associée à chaque opérateur de la syntaxe abstraite Lesp. Une fois invoquée sur une instance d'un opérateur, **traduire** génère l'arbre Ada correspondant.

La traduction complète d'un système parallèle Lesp s'effectue en invoquant la méthode sur le noeud **oprog** représentant le programme principal. Le résultat de la traduction est composé:

- d'un arbre **ounit-comp** contenant la procédure Ada qui traduit le programme principal,
- d'une liste d'arbres contenant d'une part les sous-unités de compilation qui traduisent les processus internes, d'autre part les unités de compilation qui traduisent les types de communication.

Cette liste représente en fait la bibliothèque d'unités Ada utilisé par le programme principal et séparées de celui-ci pour la compilation séparée.

Nous nous proposons de montrer comment dériver la méthode **traduire** à partir des règles d'inférence spécifiées au chapitre 6. Nous commençons tout d'abord par présenter le principe d'implantation des règles relatives à l'environnement de compilation. Pour cela, nous prenons l'exemple du module **c-port.** Ensuite, nous présentons l'implantation des règles de compilation.

Implantation de l'environnement: le module c-port

Ce module construit une liste de couples composés de l'identificateur d'une entrée ou d'une sortie et du nom de port associé. Cette liste est construite à partir de l'interface d'un processus élémentaire (définit par l'opérateur **specif**). Nous présentons l'implantation de la première règle du module c-port. Cette présentation va nous permettre de dégager le principe d'implantation du module complet. Rappelons une telle règle:

$$\frac{\begin{array}{l}(1)\\ \text{ENV} \vdash \text{ENTRES,IDENT : ENV1}\\ \text{ENV} \vdash \text{SORTIES,IDENT : ENV2}\end{array}}{\text{ENV} \vdash \textbf{specif}(\text{ENTRES,SORTIES}),\text{IDENT : env[ENV1+ENV2]}}$$

Pour rendre très explicite le passage des règles d'inférence à leurs implantations fonctionnelles en CEYX, nous les réecrivons d'abord sous une forme clausale à la prolog. Puis nous présentons comment passer d'une telle forme à des fonctions.

Passage à une forme clausale

La règle précédente spécifie la preuve du prédicat ":" (que nous notons par la suite port pour plus de clarté). De manière générale, une règle d'inférence permet de décomposer un problème (qui est ici la preuve de la conclusion de la règle) en plusieurs sous-problèmes ou sous-buts (preuves des prémisses). Le problème forme la partie basse d'une règle, tandis que les sous-buts en constituent la partie haute. De là on peut établir une analogie avec le langage Prolog. En effet, un programme Prolog est composé d'une liste de clauses qui sont elles-mêmes des suites de littéraux. Ces littéraux s'apparentent aux prédicats de TYPOL (par exemple le prédicat port), tandis que les clauses s'apparentent aux règles de TYPOL. La règle précédente peut alors être considérée comme une clause, où la conclusion est le littéral de tête, les deux prémisses représentent les sous-buts.

Ainsi, nous associons à toute formule A ⊢ B,C : D, le littéral port(A,B,C,D) et à la règle précédente la clause que nous appelons C1 définie de la manière suivante:

```
port(ENV,specif(ENTRES,SORTIES),IDENT,env) ← port(ENV,ENTRES,IDENT,ENV1)
                                              port(ENV,SORTIES,IDENT,ENV2)
                                              append(ENV1,ENV2,env)
```

Nous avons introduit le prédicat *append* pour noter l'opérateur "+" qui figure dans les règles d'inférence.

Passage à une forme fonctionnelle

Nous nous inspirons des résultats présentés dans [ALE 88] sur la transformation de programmes logiques en programmes fonctionnelles. Les règles de transformation que nous appliquons sont les suivantes:

- **INTRO** : cette règle consiste à introduire des symboles fonctionnels. Soit S un ensemble de clauses, S1=**INTRO**(S;P,I) est un nouvel ensemble de clause dérivé de S, en remplaçant toute occurrence d'atome de la forme $P(t_1,\ldots,t_n)$ par : $EQ(P(t_1,\ldots,t_{i-1},t_{i+1},\ldots,t_n),t_i)$ Où EQ est un prédicat qui joue le rôle de l'égalité.

- **SIMP** : permet d'éliminer dans une clause un atome de la forme EQ(-,-).
- **FONC** : permet de passer dans un nouveau système formel dont les règles sont les règles d'évaluation des fonctions.

Nous ne donnons pas ici la description formelle de ces règles, ni leurs preuves qui sont présentées dans les références c itées. En revanche, nous montrons leur intérêt pratique dans la transformation des règles d'inférences en fonctions CEYX. Cette transformation procède en plusieurs étapes que nous décrivons sur la clause C1.

Etape 1 : Application de la règle **INTRO**

Remplaçons dans la clause C1 le symbole "port" par un nouveau symbole binaire noté "c-port", en choisissant le quatrième paramètre comme résultat et en introduisant le prédicat binaire EQ qui joue le rôle de l'égalité. Nous obtenons l'énoncé C2 suivant:

```
EQ(c-port(ENV,specif(ENTRES,SORTIES), IDENT),env) ←
                                              EQ(c-port(ENV,ENTRES,IDENT),ENV1)
                                              EQ(c-port(ENV,SORTIES,IDENT),ENV2)
                                              append(ENV1,ENV2,env)
EQ(t,t) ←
```

Etape 2 : application de la règle **SIMP**

Nous éliminons ENV1 et ENV2 en supprimant les atomes de la forme EQ(-,-) dans les prémisses de la première clause. D'où l'énoncé C3:

```
EQ(c-port(ENV,specif(ENTRES,SORTIES), IDENT),env) ←
                                              append(c-port(ENV,ENTRES,IDENT),
                                                     c-port(ENV,SORTIES,IDENT),
                                                     env)
EQ(t,t) ←
```

Etape 3 : Application successive de la règle **INTRO** et de la règle **SIMP** au symbole append pour obtenir le nouvel énoncé C4 :

```
EQ(c-port(ENV,specif(ENTRES,SORTIES),IDENT),
    append(c-port(ENV,ENTRES,IDENT),c-port(ENV,SORTIES,IDENT)) ←
EQ(t,t) ←
```

Ce dernier énoncé se transforme facilement par la règle **FONC** en une fonction c-port telle que :

c-port(ENV,**specif**(ENTRES,SORTIES),IDENT) = append(c-port(ENV,ENTRES,IDENT),
c-port(ENV,SORTIES,IDENT))

que nous réécrivons dans la syntaxe CEYX en adoptant les règles de traduction suivante:

- la fonction c-port est traduite en une méthode CEYX attachée à l'opérateur specif,
- ENTRES et SORTIES qui représentent les fils d'un arbre specif sont obtenus par la fonction *get-son* : étant donné un arbre A et un entier I, *get-son* fournit le Ième fils de A,
- l'appel interne à la fonction c-port (partie droite du symbole "=") se traduit par l'envoie du message c-port (**sendq** en CEYX) sur les fils d'un arbre **specif**.

D'où la fonction suivante:

```
(de {specif}: c-port(ENV SP IDENT); SP est un arbre instance de l'opérateur specif
    (append (sendq port ENV (get-son SP 1) IDENT); premier fils de pylum ENTRES
            (sendq port ENV (get-son SP 2) IDENT); deuxième fils de phylum SORTIES
    )
)
```

Ainsi, l'implantation de chaque règle du module c-port produit une méthode CEYX. Cette méthode est associée à l'opérateur se trouvant en tête de la règle.

Implantation d'une règle de compilation

En suivant le même principe que précédemment, nous allons montrer comment implanter les règles de compilation en termes de fonctions CEYX. Pour cela, nous prenons l'exemple de la règle de compilation de la partie interface des processus élémentaires, spécifiée de la manière suivante:

$$\frac{\text{ENV} \vdash \text{ENTRES} \rightarrow \text{DVARS1} \quad \text{ENV} \vdash \text{SORTIES} \rightarrow \text{DVARS2}}{\text{ENV} \vdash \textbf{specif}(\text{ENTRES},\text{SORTIES}) \rightarrow \textbf{dvars}(\text{DVARS1}+\text{DVARS2})}$$

Passage à une forme clausale

Les règles de compilation spécifient la preuve du prédicat "→" (que nous notons par la suite "trad"). Ainsi que nous l'avons fait pour les règles de l'environnement, nous associons à toute formule A ⊢ B → C, le littéral trad(A,B,C) et à la règle de compilation la clause C1 définie de la manière suivante :

```
trad(ENV,specif(ENTRES,SORTIES),dvars) ← trad(ENV,ENTRES,DVARS1)
                                         trad(ENV,SORTIES,DVARS2)
                                         append(DVARS1,DVARS2,dvars)
```


Passage à une forme fonctionnelle

Appliquons sur la clause C1 les règles de transformation (INTRO,SIMP,FONC) pour aboutir à un programme fonctionnel. Nous retrouvons ainsi les mêmes étapes que pour la transformation des règles de manipulation de l'environnement.

Etape 1 : Application de la règle INTRO.

Le symbole trad est remplacé par le symbole traduire, en choisissant le troisième paramètre comme résultat. Le prédicat EQ est introduit. D'où le nouvel énoncé C2:

```
EQ(traduire(ENV,specif(ENTRES,SORTIES)), dvars) ←
                                          EQ(traduire(ENV,ENTRES),DVARS1)
                                          EQ(traduire(ENV,SORTIES),DVARS2)
                                          append(DVARS1,DVARS2,dvars)
EQ(t,t) ←
```

Etape 2 : application de la règle SIMP. On élimine DVARS1 et DVARS2 dans la première clause et les atomes de la forme EQ(-,-) pour obtenir le nouvel énoncé C3 :

```
EQ(traduire(ENV,specif(ENTRES,SORTIES)),dvars)
    ← append(traduire(ENV,ENTRES),traduire(ENV,SORTIES),dvars)
EQ(t,t) ←
```

Etape 3 : application de la règle **INTRO** et de la règle **SIMP** au symbole append pour obtenir l'énoncé C4 :

```
EQ(traduire(ENV,specif(ENTRES,SORTIES)),
    append(traduire(ENV,ENTRES),traduire(ENV,SORTIES))) ←
EQ(t,t) ←
```

En appliquant la règle **FONC** on obtient le programme suivant :

```
traduire(ENV,specif(ENTRES,SORTIES))= append( traduire(ENV,ENTRES),
                                              traduire(ENV,SORTIES))
```

Nous traduisons ce programme en une fonction CEYX en suivant les règles de traduction suivantes :

- la fonction traduire est une méthode CEYX attachée à l'opérateur **specif**,
- ENTRES et SORTIES sont traduits par l'appel à la fonction *get-son* qui permet d'accéder aux fils d'un arbre.

D'où la fonction suivante :

```
(de {specif}: traduire (ENV SP)
    (append (sendq traduire ENV (get-son SP 1))
            (sendq traduire ENV (get-son SP 2)))
    )
)
```

Résumons maintenant le principe d'implantation des règles d'inférence en CEYX:

- chaque règle d'inférence se traduit en une méthode, de même nom que le prédicat spécifié dans la règle. Cette méthode est rattachée à l'opérateur se trouvant en tête de la règle,
- les prémisses constituent le corps de la méthode et se traduisent par l'activation de celle-ci sur les arbres fils de l'opérateur se trouvant en tête de la règle.

le module décompilateur

Ce module a pour rôle d'associer à chaque opérateur de la syntaxe abstraite Ada sa forme textuelle. Comme pour le traducteur, le décompilateur est implantée par une méthode **decompiler** que nous avons associée à chaque opérateur de la syntaxe abstraite Ada. L'algorithme de décompilation d'un arbre consiste en un parcours d'arbre en profondeur d'abord. Tout au long de ce parcours, les noeuds et les feuilles de l'arbre qui sont visités produisent un flux de caractères qui pas à pas construit la représentation textuelle de l'arbre décompilé. Considérons l'exemple de l'instruction itérative et sans condition d'arrêt définit dans la syntaxe abstraite Ada par l'opérateur **loop** suivant:

(**defcons**{STAT}: **loop** sons (vector SEQ-STAT))

L'opérateur **loop** possède un seul fils : la suite d'instructions à répéter dont le phylum est SEQ-STAT. La méthode **decompiler** est définit sur cet opérateur de la manière suivante:

```
(de {loop}: decompiler ( a )
    (print "LOOP")
    (sendq decompiler (get-son a 1))
    (print "END LOOP;")
)
```

Où le paramètre noté "a" est une instance de l'opérateur **loop**.

Notons l'envoie du message *decompiler* sur le fils de l'arbre passé en paramètre qui peut être n'importe quelle séquence d'instructions. Grâce au mécanisme de liaison dynamique, la méthode *decompiler* qui sera activée par cet envoie est celle rattachée à l'opérateur d'appartenance de ce fils.

Pour conclure, il est intéressant de revenir sur l'apport de la programmation orientée objet offerte par l'environnement CEYX. Ainsi que nous l'avons vu, l'algorithme de traduction et l'algorithme de décompilation sont répartis dans les différents opérateurs de la syntaxe abstraite à l'aide des méthodes *traduire* et *decompiler*. Il en résulte que chaque opérateur est traité séparément en faisant abstraction des autres. Du point de vue de la mise au point, cette répartition nous a permit de tester séparément les différentes méthodes et offre la possibilité de ne compiler qu'une partie d'un programme.

5 La programmation automatique

Nous présentons dans cette section deux systèmes permettant de transformer des spécifications en programmes exécutables : PSI (Program Synthesis System) et APE (Automating Programming Expert). Ces deux systèmes ne sont pas les seuls représentants de l'approche transformationnelle. Nous les avons choisi uniquement parce qu'ils introduisent des règles de transformation dont le but les rapproche le plus de celles utilisées dans l'implantation des types de communication. Le sytème SPES que nous avons présenté et utilisé pour montrer comment dériver systématiquement l'implantation CEYX des règles d'inférence, constitue lui aussi un autre exemple de systèmes transformationnels.

Les deux systèmes PSI et APE ne permettent de construire que des programmes séquentiels. En conséquence, leur lien avec notre travail se limite donc à la partie représentation des types de communication.

5.1 Le système PSI: Program Synthesis System

Développé par l'équipe de C. Green et D. Barstow [BAR 79,KAN 84], le système PSI permet de construire un programme Lisp à partir d'un algorithme informel donné par l'utilisateur. Le système se présente sous la forme d'un ensemble d'experts; chacun est dévolu à une tâche particulière du processus de construction d'un programme. Un premier groupe d'experts permet de construire une première spécification (appelée modèle de programme) décrit dans un langage de haut niveau à partir de phrases en Anglais: c'est la phase d'aquisition. Un second groupe a pour rôle de transformer cette spécification en un programme Lisp efficace: c'est la phase de synthèse.

Nous décrivons rapidement ces deux phases avant de les illustrer sur un exemple.

Phase d'acquisition

L'utilisateur peut formuler son problème à l'aide d'un langage de haut niveau et de phrases (dites sentences) utilisant un sous-ensemble de l'anglais. Pour cela le système intéragi avec l'utilisateur par l'intermédiaire de l'expert "Dialogue Moderator Expert" qui décide des questions à poser à l'utilisateur et gère le déroulement du dialogue. A partir des informations récoltées, l'énoncé du problème est traduit en des termes plus orientés programmes appelés *fragments*. Ces fragments sont ensuite assemblés pour produire un modèle de programme: un algorithme de très haut niveau pouvant contenir des annotations telles que des contraintes sur le programme ou des indications sur les données.

Les règles qui constituent la base de connaissance de cette phase sont diverses:

- les règles qui permettent de convertir des sentences anglaises en fragments, représentés sous forme de réseau sémantique. Elles utilisent des formes typiques d'usage

de l'anglais, mémorisées par le système,

- les règles qui permettent de produire des structures de données abstraites à partir des informations fournies par l'utilisateur,

- les règles qui permettent d'assembler les fragments pour construire un modèle de programme.

Phase de synthèse

Cette phase a pour but de dériver une implantation exprimée en Lisp, à partir du modèle de programme déjà obtenu. Elle procède par l'application de règles de transformation sur les structures de données abstraites et sur les opérations qui leurs sont associées. Le module PECOS qui assure ces transformations fait appel à l'expert LIBRA dont le rôle est de choisir une règle en fonction de critères d'efficacité en temps d'exécution et en espaces mémoires. Pour ce faire Libra utilise des techniques d'analyse d' algorithmes et des heuristiques fondées sur la taille des données et le côut des opérations. Une classification des règles de transformation, par type et par niveau d'abstraction permet aussi d'optimiser la recherche de règles applicables à une étape de la synthèse. On trouve trois types de règles:

- les règles qui décrivent des techniques générales de programmation. (analyse par cas, énumération, parcours de listes,..). Par exemple la règle qui permet de mettre en place une énumération s'énonce de la manière suivante: 1) trouver un ordre d'énumération, 2) trouver un moyen de conserver l'état courant de l'énumération, 3) expliciter une condition d'arrêt.

- les règles qui se chargent de la représentation des structures de données abstraites. Par exemple: Une collection de données peut être représentée par une liste chainée pour un traitement séquentiel.

- les règles relatives aux langages de programmation Lisp (dites règles de codage) qui utilisées dans les dernières étapes de la synthèse permettent de produire un programme Lisp.

Nous proposons un exemple pour illustrer le fonctionnement du système PSI.

Exemple

Considérons l'exemple du problème de test d'appartenance d'un élément à une collection de données, formulé de la manière suivante:

Structures de données:
X : *collection d'entiers*
Y : *entier*
Algorithme
is X *element de* Y

collection est une structure de données abstraite connue par le système. Elle possède plusieurs représentations concrètes.

A partir de l'énoncé précédent, le système procède aux transformations qui vont permettre de construire un programme Lisp exécutable. Il introduit alors les étapes suivantes:
Etape 1 : Cette étape conduit à représenter Y sous la forme d'une collection explicite d'entiers en utilisant la règle suivante:

une collection peut être représentée explicitement
On obtient l'énoncé suivant:

X (entier) is-element de Y (collection explicite d'entiers)

Pour saisir la différence entre une représentation implicite et une représentation explicite, prenons l'exemple d'une collection contenant les entiers compris entre 1 et 7 :

- une représentation implicite correspond au cas où la collection est représentée uniquement par le petit et le plus grand élément (1 respectivement 7),

- une représentation explicite correspond au cas où tous les entiers de la collection sont pris en compte dans la représentation.

Etape 2 : on choisit pour Y un rangement de tout les éléments en mémoire sous forme séquentielle. On obtient:

X (entier) *is-element de* Y (collection séquentielle d'entier)

Etape 3 : une collection séquentielle peut être représentée par une liste chainée ou un tableau. Si l'on choisi la première, on obtient:

X (entier) *is-element de* Y (liste chainée d'entiers)

Etape 4 : PECOS propose alors de représenter la liste en utilisant des cellules chainées. On obtient le nouvel énoncé suivant:

X (entier) *is-element de* Y (cellules chainées d'entiers)

Etape 5 : il s'agit maintenant d'implanter les cellules chainées par une liste Lisp. D'où :

X (entier) *is-element de* Y (liste Lisp d'entiers)

Etape 6 : PECOS propose de représenter l'opération IS-element par la fonction Lisp MEMBER (qui teste la présence d'un élément dans une liste), et les entiers par des entiers Lisp. On obtient alors la fonction Lisp suivante:
(MEMBER X Y)

Nous pouvons relever à travers cet exemple simple l'aspect très atomique et élémentaire de la transformation d'un énoncé. Pour traiter des énoncés de taille plus importante que le précédent, plusieurs dizaines d'étapes sont nécessaires, faisant intervenir un grand nombre de règles. Par ailleurs, le fait que le système a pour but d'intégrer tous les aspects de l'interface en langue naturelle jusqu'à l'optimisation du programme produit nécessite l'utilisation d'une base de connaissances très riche en règles de natures variées. On peut noter cependant le manque de stratégies plus globales dans le processus de transformation.

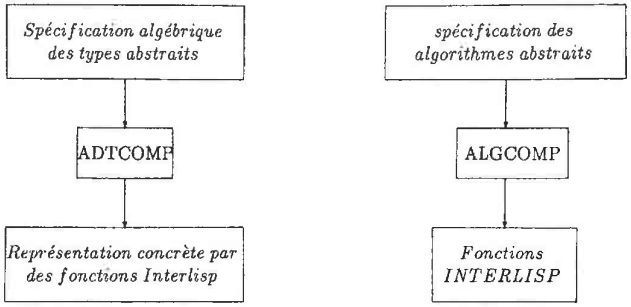
5.2 Le système APE: Automatic Programming Expert

APE est développé par Bartels, Olthoff et Raulefs [BAR 81,BAR 82]. Ce système construit un programme INTERLISP (langage dérivé de Lisp) exécutable à partir:

- de spécifications algébriques de types abstraits de données,
- d'algorithmes abstraits, présentés sous la forme de règles de réécritures conditionnelles. Les termes utilisés dans ces règles sont bâtis à partir des opérations intervenant dans la spécification des types.

Les règles de transformation sont codées dans une base de connaissances sous la forme de règles de production. Le système se limite à l'implantation des structures de données abstraites qui sont toutes des variantes du type liste. Quant aux algorithmes traités, on trouve principalement des algorithmes de tri et de recherche.

Contrairement au système PSI, APE sépare la transformation des types de données de celle des algorithmes qui les manipulent. Ceci a pour conséquence que les caractéristiques des algorithmes ne sont pas prises en compte dans le choix de la représentation des données. Nous retrouvons cette séparation dans la structure générale du système donnée par le schéma suivant:



Les deux sous-systèmes ADTCOMP et ALGCOMP correspondent respectivement à la représentation des types de données et à la transformation des algorithmes.

Nous montrons le fonctionnement du sous-système ADTCOMP, en prenant comme exemple le type abstrait de données *liste-ouverte* caractérisé par:

- les adjonctions, les suppressions et les accès se font par l'intermédiaire d'un indice de type entier (noté Nat),
- le dernier élément ajouté a comme indice un,
- les opérations *empty* et *cons* sont les constructeurs primitifs.

add : LO × NAT × ELT → LO add(empty,n,x)= **si** n=1 **alors** cons(empty,x) **sinon** undef **fsi** add(cons(l,x),n,y)= **si** n=1 **alors** cons(cons(l,x),y) **sinon** cons(add(l,n-1,x),y) **fsi**	read : LO × NAT → ELT read(empty,n)=undef read(cons(l,x),n)= **si** n=1 **alors** x **sinon** read(l,n-1) **fsi**

Le système fonctionne en sept étapes, que nous décrivons sur l'exemple précédent.

Initialisation : cette étape traduit la spécification donnée en une structure interne Lisp, et associe certaines propriétés aux opérations. Pour les opérations présentées précédemment, nous aurons:

add (IMPLEM), read (IMPLEM), cons (HIDEN)

qui exprime que les opérations add et read doivent être implantées, tandis que l'opération cons doit être cachée.

Fonctionnalités : analyse le profil des opérations (domaine et co-domaine). Les deux règles suivantes sont applicables :

R1: **si** *op a comme domaine LO* **et** *éventuellement TE* et *comme codomaine TR* **et** *LO* $\neq$ *TR* **alors** *op est une opération de lecture*
Résultat : read (lecture)

Une opération de lecture est tout simplement une opération d'accès.

R2: **si** *op a comme domaine LO* **et** *éventuellement TE* et *comme codomaine LO* **et** *LO* $\neq$ *TE* **alors** *op est une opération un-continue*
Résultat : add continue (un), cons continue (un)

Une opération est dite n-continue si elle construit un objet du type source à partir de n objets de ce type.

D'autres règles permettent de conclure que les entiers sont utilisées par les opérations comme critère d'accès aux éléments d'un objet de type LO. A la fin de cette étape, le système établit l'entête des fonctions INTERLISP, associées aux opérations:

add FUNCT (LAMBDA (D N X))
read FUNCT (LAMBDA (D N))

Axiome : analyse les axiomes de la spécification. La règle suivante est applicable:

R3: **si** *op1 est une opération qui manipule les objets de type LO à partir de termes construits par l'opération op2*
alors *op2 est un constructeur primitif*
Résultat : constr (constructeur primitif)

une autre règle permet de conclure quer les opérations add et read se font relativement au dernier élément ajouté par le constructeur primitif constr.

Représentation : c'est cette étape qui permet de choisir une représentation du type source. Elle repose sur les informations fournies par les étapes précédentes. Le fait que les objets du type LO soient manipulés par des opérations un-continues permet au système de les représenter par des listes. Il reste cependant à choisir entre deux structures de listes:

- liste linéaire : les opérations se font à un emplacement fixe en mémoire (en tête ou en queue),
- liste bilatère : les opérations peuvent se faire en tête et en queue.

L'application de d'autres règles permettent de choisir le type liste linéaire comme représentation : seule l'adresse du dernier élément suffit pour réaliser les adjonctions (opération add) et les accès (opération read). Remarquons que ce choix repose sur les informations fournies par l'étape **axiome**.

Implantation : cette étape définit la représentation des objets du type LO en terme de cellules *cons*, et associe aux opérations des fonctions INTERLISP. Ces dernières sont exprimées en termes de fonctions prédéfinies telles que *car* (accès au premier élément d'une liste) et *cdr* ().

Toutes les règles utilisées par le système APE se présentent sous la forme de règles de production identiques à celles que nous avons utilisées dans l'exemple précédent. A la lecture de ces règles, il apparaît très difficile de faire ressortir des principes généraux et une démarche globale pour la transformation de programmes.

6 Conclusion

Les deux systèmes que nous venons de présenter font apparaître certains points communs avec le système de transformation des types de communication. Ils ont tous pour objectif de dériver des programmes exécutables à partir de descriptions explicites de problèmes. Pour cela, ils procèdent non seulement à des transformations sur les opérations mais aussi à des représentations des structures de données employées. La notion d'étapes est également introduite par les trois systèmes comme moyen pour structurer le processus de transformation.

Cependant, le manque de stratégies, voir de démarches globales dans les systèmes PSI et APE rend difficile la compréhension des transformations. Notons également l'absence totale d'un cadre formel pour décrire les règles utilisées par ces deux systèmes. Par conséquent, aucune preuve n'a été donnée quant leurs corrections. A ce sujet, nous pensons que l'approche "système expert" ne doit être qu'une aide à l'implantation des systèmes de dérivation de programmes et non un cadre formel pour les définir.

Conclusion

Le concept de communication a constitué le centre de notre étude. Il intervient aussi bien dans la phase de décomposition d'un problème que dans la phase de conception d'une solution en termes de processus communicants.

Nous avons pu montrer au *chapitre 1*, à travers des exemples de langages de spécification, ce qui caractérise une expression abstraite de la communication par rapport à sa mise en oeuvre. Cette synthèse nous a permis de dégager clairement deux aspects fondamentaux:

- la communication doit être prise en compte au niveau de la spécification pour s'abstraire des synchronisations qui réalisent les contraintes nécessaires à la mise en oeuvre.

- qu'il s'agisse d'exprimer la communication ou la synchronisation, il est souhaitable, pour la modularité, de le faire indépendamment des processus concernés.

Ces deux aspects constituent les deux caractéristiques du langage d' expression de systèmes parallèles Lesp que nous avons présenté au *chapitre 2*.

Nous nous sommes consacrés ensuite à la dérivation de programmes Ada à partir de systèmes exprimés en Lesp. Nous avons voulu ainsi contribuer à la démarche définie dans [PER 85] en associant aux concepts proposés des mises en oeuvre dans un langage de programmation. Pour cela, nous avons proposé dans les *chapitres 3, 4 et 5* une démarche pour transformer les types de communication en programmes Ada. Cette démarche est fondée sur une approche transformationnelle [BRO 81] [FEA 86] et possède les particularités suivantes:

- l'énoncé de départ est la spécification d'un type abstrait de données,

- l'énoncé d'arrivée est un paquetage Ada,

- le passage du premier énoncé vers le second se fait par l'application de règles de transformation dont on a prouvé la validité.

Les règles de transformation jouent un rôle très important dans la dérivation, c'est pourquoi nous les rappellons ci-dessous tout en établissant le lien avec des travaux similaires:

- la règle de simplification permet de supprimer dans une spécification les objets qui ne sont pas utilisés en mode consultation. Une telle règle trouve son équivalent dans [WIL 81] sous le nom de variables mortes.

- certaines règles visent à construire une représentation éfficace en remplaçant des fonctions récursives par des variables informatiques et en simplifiant le domaine d'induction. Nous rejoignons en celà les transformations concernant la suppression de la récursivité [DAR 76] [ARS 82], quoique dans ce dernier cas le domaine d'induction reste inchangé,

- d'autres règles de représentation appliquent des heuristiques à partir de propriétés extraites de la spécification pour choisir une représentation en bibliothèque. Cette approche dite à base de connaissance est utilisée par les systèmes: PSI [BAR 79], APE [BAR 81], et SAIDA [GRA 88].

Il est intéressant de noter que dans cette dérivation, seules les règles de compilation introduisent des connaissances sur le langage cible. Aussi pour obtenir une dérivation dans un autre langage, il suffit de redéfinir de telles règles. Tous les travaux cités précédement et avec lesquels nous avons relevé des points communs concernent la transformation de spécification en vue de produire des programmes séquentiels. En se basant sur les objectifs assignés à notre travail, nous retrouvons aussi des études similaires dans le cadre de la programmation parallèle. Citons la transformation de spécifications exprimées sous forme de modules de contrôle en programmes SIMON [HER 80]. De tels modules spécifient les contraintes de synchronisation liées aux accés à des données partagés.

Nous avons ensuite proposé une spécification de la transformation des processus Lesp en programmes Ada à l'aide de la méthode dite "sémantique naturelle" [KAH 87]. Cette méthode a été souvent utilisée pour définir la sémantique des langages. Par cette spécification, nous donnons d'une part un exemple assez important (compte tenu de la richesse des deux langages) de son utilisation dans le cadre de la traduction des langages, d'autre part nous fournissons un bon support pour notre réalisation et pour la preuve de correction du traducteur.

Notre étude s'est concrétisée par la réalisation d'un système mettant en oeuvre la dérivation de programmes Ada. Ce système s'intègre au sein d'un environnement de programmation composé d'un éditeur syntaxique et d'un interprète de programmes Lesp. Nous avons aussi abordé le problème de la réutilisation en se limitant aux cas des types de communication. A ce sujet, nous avons proposé une structure fondée sur les trois concepts

de composant abstrait, composant concret et relation. L'implantation d'une telle structure constitue l'un des prolongements à cours terme de notre étude.

Enfin, la notion de type de communication, jointe à la dérivation de programmes Ada étudiées dans ce travail, ouvrent deux axes de recherche. Le premier concerne la preuve de correction des programmes Ada résultat de la dérivation. A ce sujet, le système de preuve proposé dans [GER 84] fournit un cadre formel bien adapté aux programmes parallèles que nous avons considéré dans notre travail.

Le second axe concerne l'aspect méthode de conception de programmes parallèles. Au delà des cadres linguistiques (langages de spécification, de conception et d'implantation) et théoriques (sémantique, preuve), le problème méthodologique de la décomposition d'un énoncé en termes de processus communicants reste difficile et fondamental. Dans ce contexte, nous pensons que des propositions telles que celles faites par le groupe Anna Gram [GRA 86] peuvent constituer un repère intéressant.

Annexe A

Syntaxe abstraite du langage Lesp et son implantation en CEYX

1 Syntaxe abstraite exprimée en terme d'opérateurs et de phyla

prog : IDENT × DECLS-PROG × PORTS → PROG
decls-prog : DECL-PROG* → DECL-PROGS
ports : PORT* → PORTS
port : NOMVARS × IDENT → PORT
process : IDENT × SPECIF × CORPS → DECL-PROG
specif : ENTRES × SORTIES → SPECIF
entres : ENTRE* → ENTRES
sorties : SORTIES* → SORTIES
entre : NOMVARS × IDENT × IDENT → ENTRE
sortie : NOMVARS × IDENT × NOMVARS → SORTIE
coprs : DECLS × INSTRS → CORPS
decls : DECL* → DECLS
dvar : NOMVARS × TYPE-IDENT → DECL
dtype : IDENT × TYPE-IDENT
entier : → TYPPREDEF
reel : → TYPPREDEF
car : → TYPPREDEF
booleen : → TYPPREDEF
tab : INTERVAL* × TYPE-IDENT → TYPE
interval : CST-NUM × CST-NUM → INTERVAL
struct : CHAMP* → TYPE
champ : NOMVARS × TYPE-IDENT → CHAMP
instrs : INSTR* → INSTRS
prod : IDENT → INSTR
cons : IDENT → INSTR
select : GARDE × INSTRS → INSTR
lire : NOMVARS → INSTR
ecrire : ARITH* → INSTR
sialors : LOG × INSTRS → INSTR
sialorsinon : LOG × INSTRS × INSTRS → INSTR
tantque : LOG × INSTRS → INSTR
affec : CHAMP-TAB-IDENT × EXP → INSTR

nomvars : IDENT* → NOMVARS
ident : → IDENT

2 implantation de la syntaxe abstraite à l'aide des constructions deftree et defcons

```
    ;;;;; le super univers racine de toute la hierarchie des univers;;;;;
;;;;; pour definir les commentaires sur chaque noeud ;;;;;
(deftree lu (dc~string "")(fc~string "")) ; dc:com pre ; fc : com post
(defmodel l-string(Predicate fl-string))
(de fl-string (l)
(while (and l(stringp(car l)))(nextl l))
(null l))
(deftree {lu}:decls-prog)
(deftree {decls-prog}:decl-prog)
(defcons {decls-prog}:odecls-prog sons~(List decl-prog))
(deftree {decl-prog}:prog (sem-ok~symbol nil) (tc-v~l-string nil)(tc-ut~l-string nil))
(deftree {lu}:nomvars)
(deftree {nomvars}:ident (validentificateur))
(deftree {lu}:sisole)
(deftree {sisole}:decls)
(defcons {prog}:oprog sons~(Vector ident decls-prog ports))
(defcons {ident}:oident)
(defcons {ident}:oid-int)
(modelunion mdecl(metavar decl))
(defcons {decls}:odecls sons~(List mdecl))
(defmodel {string}:identificateur(Predicate fidentificateur))
(de fidentificateur (id)
(or(omatchq ident-minus id)(omatchq ident-majus id)))
;;;;; declaration de variables ;;;;;
(deftree {decls}:decl)
(deftree {decl}:dvars)
(deftree {dvars}:dvar)
(deftree {lu}:type)
(deftree {type}:typpredef)
(modelunion mdvar(metavar dvar))
(defcons {dvars}:odvars sons~(List mdvar))
(defcons {dvar}:odvar sons~(Vector nomvars type-ident))
(defcons {nomvars}:onomvars sons~(List ident))
;;;;; declaration de types ;;;;;
(deftree {decl}:dtypes)
(deftree {dtypes}:dtype)
(modelunion mdtype(metavar dtype))
(defcons {dtypes}:odtypes sons~(List mdtype))
(defcons {dtype}:odtype sons~(Vector ident type-ident))
(defcons {typpredef}:oentier)
(defcons {typpredef}:oreel)
(defcons {typpredef}:obooleen)
(defcons {typpredef}:ocar)
(deftree {lu}:intervals)
(deftree {intervals}:interval)
(defcons {interval}:ointerval sons~(Vector cst-num cst-num))
(defcons {intervals}:ointervals sons~(List interval))
```



```
(deftree{type}:typcons)
(defcons {typcons}:otab sons~(Vector intervals type-ident))
(deftree {sisole}:champ)
(defcons {champ}:ochamp sons~(Vector nomvars type-ident))
(defcons {typcons}:ostruct sons~(List champ))
(modelunion type-ident(type ident metavar))
(modelunion cst-ident(ident cst-num cst-alpha arithsuite))
;;;;; declaration de processus ;;;;;
(deftree {sisole}:specif)
(deftree {sisole}:corps)
(deftree {lu}:entres)
(deftree {lu}:sorties)
(deftree {entres}:entre)
(deftree {sorties}:sortie)
(deftree {lu}:ports)
(deftree {decl-prog}:process (sem-ok~symbol nil)(var-v~l-string nil)(var-ut~l-string nil)
(typ-v~l-string nil)(typ-ut~l-string nil) (fonct-v~l-string nil)(fonct-ut~l-string nil)
(entree-v~l-string nil) (entree-ut~l-string nil)(sortie-v~l-string nil)(sortie-ut~l-string nil)
(port-v~l-string nil)(port-ut~l-string nil))
(defcons {process}:oprocess sons~(Vector ident specif corps))
(defcons {specif}:ospecif sons~(Vector entres sorties))
(modelunion mentre(metavar entre))
(modelunion msortie(metavar sortie))
(defcons {entres}:oentres sons~(List mentre))
(defcons {sorties}:osorties sons~(List msortie))
(defcons {entre}:oentre sons~(Vector nomvars ident ident))
(defcons {sortie}:osortie sons~(Vector nomvars ident nomvars))
(deftree {ports}:port)
(defcons {ports}:oports sons~(List port))
(defcons {port}:oport sons~(Vector nomvars ident))
;;;;; corps d'un processus ;;;;;
(deftree {lu}:instrs)
(defcons {corps}:ocorps sons~(Vector decls instrs ))
(modelunion minstr(metavar instr))
(defcons {instrs}:oinstrs sons~(List minstr))
(deftree {instrs}:instr)
(defcons {instr}:oprod sons~(Vector ident))
(defcons {instr}:ocons sons~(Vector ident))
(defcons {instr}:oselect sons~(List cdegardee))
(deftree {sisole}:cdegardee)
(defcons {cdegardee}:ocdegardee sons~(Vector garde instrs))
(modelunion garde(log ocons))
;;;;; declaration de fonctions ;;;;;
(deftree {decl}:dfonct (sem-ok~symbol nil)(var-v~l-string nil)
(var-ut~l-string nil)(typ-v~l-string nil)(typ-ut~l-string nil)
(fonct-v~l-string nil)(fonct-ut~l-string nil)(pf~l-string nil))
(defcons {dfonct}:odfonct sons~(Vector ident dvars ocorps type-ident ))
;;;;; EXPRESSIONS ARITHMETIQUES ET LOGIQUES ;;;;;
(deftree {sisole}:exps)
(deftree {lu}:arithmetique )
(deftree {lu}:logique )
(deftree {logique}:cst-bool)
(modelunion exp(arith log))
(modelunion arith (arithmetique ident destab deschamp metavar opint tampon valeur))
```

```
(modelunion log (logique ident destab deschamp metavar opint tampon valeur))
(defcons {arithmetique}:o+ sons~(Vector arith arith))
(defcons {arithmetique}:o- sons~(Vector arith arith))
(defcons {arithmetique}:o× sons~(Vector arith arith))
(defcons {arithmetique}:odiv sons~(Vector arith arith))
(defcons {arithmetique}:omod sons~(Vector arith arith))
(deftree {arithmetique}:uarith)
(defcons {uarith}:ou+ sons~(Vector arith))
(defcons {uarith}:ou- sons~(Vector arith))
(defcons {logique}:o= sons~(Vector arith arith))
(defcons {logique}:o<> sons~(Vector arith arith))
(defcons {logique}:o> sons~(Vector arith arith))
(defcons {logique}:o< sons~(Vector arith arith))
(defcons {logique}:o>= sons~(Vector arith arith))
(defcons {logique}:o<= sons~(Vector arith arith))
(deftree {logique}:bool2)
(defcons {bool2}:oet sons~(Vector log log))
(defcons {bool2}:oou sons~(Vector log log ))
(defcons {logique}:ounon sons~(Vector log))
(defcons {cst-bool}:ovrai)
(defcons {cst-bool}:ofaux)
(deftree {arithmetique}:cst-alpha( val~string))
(defcons {cst-alpha}:ocst-alpha)
(deftree {arithmetique}:cst-num (val~integer))
(defcons {cst-num}:ocst-ent)
(defcons {cst-num}:ocst-reel)
(deftree {sisole}:destab)
(deftree {sisole}:deschamp)
(modelunion destab-ident(destab ident))
(modelunion destab-deschamp-ident(deschamp destab ident))
(defcons {exps}:oexps sons~(List arith))
(modelunion exps-arith(exps arith))
(defcons {destab}:odestab sons~(Vector destab-ident exps-arith))
(defcons {deschamp}:odeschamp sons~(Vector destab-deschamp-ident destab-ident))
;;;;; CORPS D'UN PROCESSUS OU D'UNE FONCTION ;;;;;
(defcons {instr}:ocom sons~(Vector nomvars))
(defcons {instr}:olire sons~(Vector nomvars))
(defcons {instr}:oecrire sons~(List arith))
(modelunion minstrs(metavar instrs))
(defcons {instr}:osialors sons~(Vector log minstrs))
(defcons {instr}:osialorsinon sons~(Vector log minstrs minstrs))
(defcons {instr}:otantque sons~(Vector log minstrs))
(defcons {instr}:oaffec sons~(Vector destab-deschamp-ident exp))
;;;;; DECLARATION D'UN TYPE DE COMMUNICATION ;;;;;
(deftree {lu}:udef)
(deftree {udef}:defs)
(deftree {udef}:def)
(deftree {defs}:s-def)
(deftree {lu}:ints)
(deftree {ints}:int)
(defcons {decl-prog}:otca sons~(Vector ident log defs ints log))
(defcons {ints}:oints sons~(List int))
(modelunion choix-exp(choix exp))
(defcons {int}:ointid sons~(Vector ident choix-exp ident))
```



```
(deftree {lu}:choix)
(deftree {ident}:fonct)
(defcons {lu}:oprofil sons~(Vector nomvars ident))
(defcons {int}:ointop sons~(Vector fonct choix-exp oprofil))
(defcons {choix}:ochoix1 sons~(Vector log choix-exp))
(defcons {choix}:ochoix2 sons~(Vector log choix-exp choix-exp))
(deftree {lu}:tampon )
(deftree {tampon}:tfval)
(deftree {tampon}:tnval)
(deftree {tampon}:opint (validentificateur))
(defcons {opint}:oopint sons~(List arith))
(deftree {sisole}:l-arg)
(defcons {fonct}:ofonct sons~(Vector ident l-arg))
(defcons {l-arg}:ol-arg sons~(List exp))
(defcons {defs}:odefs sons~(List def))
(defcons {def}:odef sons~(Vector log s-def))
(defcons {s-def}:os-def sons~(Vector mvaleur mtampon))
(modelunion mtampon(metavar tampon opint choix))
(modelunion mvaleur(metavar valeur opint choix))
(deftree {lu}:valeur )
(deftree {valeur}:vfval)
(defcons {vfval}:opremier sons~(Vector tampon))
(defcons {vfval}:odernier sons~(Vector tampon))
(defcons {valeur}:oextrait sons~(Vector tampon arith))
(defcons {valeur}:oomega)
(defcons {tfval}:odecapite sons~(Vector tampon))
(defcons {tnval}:oenleve sons~(Vector tampon arith))
(defcons {tnval}:osuffixe sons~(Vector tampon arith))
(defcons {tfval}:ofin sons~(Vector tampon))
(defcons {tfval}:otue sons~(Vector tampon))
;;;;; complément aux expressions logiques ;;;;;
(defcons {logique}:ovide sons~(Vector tampon))
(defcons {logique}:odans sons~(Vector tampon arith))
(deftree {tampon}:neutre(val~nomsuite))
(defcons {neutre}:oneutre)
(defcons {neutre}:oneutre-argt)
(defmodel {string}:nomsuite (Predicate fnomsuite))
(de fnomsuite(x)
(or(equal x "A")(equal x "SP")(equal x "SC")
(equal x "a")(equal x "sp")(equal x "sc")))
;;;;; complément aux expressions arithmetiques ;;;;;
(deftree {arithmetique}:arithsuite)
(defcons {arithsuite}:ocard sons~(Vector tampon))
(defcons {arithsuite}:osup sons~(Vector tampon))
(defcons {arithsuite}:oinf sons~(Vector tampon))
;;;;; meta-variables et schemas predefinis ;;;;;;;;;;;;;;;;
(defmodel {string}:metaident(Predicate fmetaident))
(de fmetaident (id)
(let((l(explode id))) (and(eq(car l)(cascii '$))
(omatchq identificateur(string(implode(cdr l)))))))
(deftree {lu}:metavar (val~metaident))
(defcons {metavar}:ometavar)
```

Annexe B

Syntaxe abstraite du langage Ada et son implantation en CEYX

1 Syntaxe abstraite exprimée en termes d'opérateurs et de phya

unit-comp : W-CLAUSE × U-CLAUSE × UNIT → UNIT-COMP
sub-unit : IDENT × PROPER-BODY → .....
pack-spec : IDENT × BAS-DEC × PRIVATE → SPEC
proc-spec : ENTETE → SPEC
func-spec : ENTETE × IDENT → SPEC
task-spec : IDENT × ENTRY-DEC* → SPEC
gen-spec : GENFORM-PART × UNIT-SPEC → SPEC
pack-body : IDENT × DEC-PART × SEQ-STAT × EXCEP-HAND → BODY
proc-body : ENTETE × DEC-PART × SEQ-STAT × EXCEP-HAND → BODY
func-body : ENTETE × IDENT × DEC-PART × SEQ-STAT × EXCEP-HAND → BODY
task-body : IDENT × DEC-PART × SEQ-STAT × EXCEP-HAND → BODY

2 Implantation de la syntaxe abstraite à l'aide des constructions deftree et defcons

(**deftree** Ada)
(**deftree** {Ada}: unit)
(**deftree** {Ada}: unit-comp)
(**defcons** {unit-comp}: ounit-comp sons`(vector w-clause u-clause unit))
(**deftree** {unit}: spec)
(**deftree** {spec}: pack-spec)
(**deftree** {spec}: subp-spec)
(**deftree** {spec}: task-spec)
(**deftree** {spec}: gen-spec)
(**deftree** {spec}: gen-inst)
(**deftree** {unit}: body)
(**deftree** {body}: proper-body)
(**deftree** {proper-body}: subp-body)
(**deftree** {proper-body}: pack-body)
(**deftree** {proper-body}: task-body)
(**deftree** {unit}: sub-unit)

```
(defcons {sub-unit}: osub-unit sons~(vector ident proper-body))
(deftree {body}: body-stub)
(defcons {body-stub}: oproc sons~(vector entete))
(defcons {body-stub}: ofunc sons~(vector entete ident))
(defcons {body-stub}: opack sons~(vector ident))
(defcons {body-stub}: otask sons~(vector ident))
(deftree {Ada}: dec-part)
(defcons {dec-part}: odec-part sons~(list mdec-part))
(modelunion mdec-part (spec decls body))
(deftree {Ada}: bas-dec)
(defcons {bas-dec}: obas-dec sons~(list mbas-dec))
(modelunion mbas-dec (spec decls))
(defcons {pack-spec}: opack-spec sons~(vector ident bas-dec private))
(defcons {subp-spec}: oproc-spec sons~(vector entete))
(defcons {subp-spec}: ofunc-spec sons~(vector entete ident))
(defcons {task-spec}: otask-spec sons~(vector ident entrys-dec))
(defcons {gen-spec}: ogen-spec sons~(vector genform-part unit-spec))
(modelunion unit-spec (pack-spec subp-body))
(defcons {gen-inst}: ogpack sons~(vector ident ident l-arg))
(defcons {gen-inst}: ogproc sons~(vector ident ident l-arg))
(defcons {gen-inst}: ogfunc sons~(vector ident ident l-arg))
(defcons {subp-body}: oproc-body sons~(vector entete dec-part seq-stat excep-hand))
(defcons {subp-body}: ofunc-body sons~(vector entete ident dec-part seq-stat excep-hand))
(defcons {pack-body}: opack-body sons~(vector ident dec-part seq-stat excep-hand))
(defcons {task-body}: otask-body sons~(vector ident dec-part seq-stat excep-hand))
(deftree {Ada}: entete)
(defcons {entete}: oentete sons ~(vector ident form-part))
(deftree {Ada}: form-part)
(defcons {form-part}: oform-part sons~(list param-spec))
(deftree {form-part}: param-spec)
(defcons {param-spec}: oparam-spec sons~(vector dvar mode ))
(deftree {Ada}: mode)
(defcons {mode}: omodin)
(defcons {mode}: omodout)
(defcons {mode}: omodinout)
(deftree {Ada}: entrys-dec)
(defcons {entrys-dec}: oentrys-dec sons ~(list entry-dec))
(deftree {entrys-dec}: entry-dec)
(defcons {entry-dec} : oentry-dec sons~(vector entete))
(deftree {Ada}: genform-part)
(defcons {genform-part}: ogenform-part sons~(list genpart-dec))
(modelunion genpart-dec (subp-spec privtyp-dec))
(deftree {Ada}: seq-stat)
(defcons {seq-stat}: oseq-stat sons~(list instr))
(defcons {instr}: onull)
(defcons {instr}: oraise sons~(vector ident))
(deftree {instr}: delay)
(defcons {delay}: odelay sons~(vector arithmetique))
(defcons {instr}: oloop sons~(vector seq-stat))
(deftree {instr}: entproc-call)
(defcons {entproc-call}: oentproc-call sons~(vector ref-mod l-arg))
(deftree {Ada}: fonc-call)
(defcons {fonc-call}: ofonc-call sons~(vector ref-mod l-arg))
(deftree {Ada}: ref-mod)
```



```
(defcons {ref-mod}: oref-mod sons~(vector ident ident))
(defcons {instr}: oreturn sons~(vector exp))
(deftree {instr}: accept)
(defcons {accept}: oaccept sons~(vector entete seq-stat))
(deftree {instr}: select)
(deftree {Ada}: select-walt)
(deftree {select}: select-wait)
(defcons {select-wait}: oselect-wait sons~(list select-alt))
(deftree {Ada}: select-alt)
(defcons {select-alt}: oselect-alt sons~(vector log select-walt))
(deftree {select-walt}: accept-alt)
(defcons {accept-alt}: oaccept-alt sons~(vector accept seq-stat))
(deftree {select-walt}: delay-alt)
(defcons {delay-alt}: odelay-alt sons~(vector delay seq-stat))
(defcons {select-walt}: oterminate)
(deftree {select}: select-else)
(defcons {select-else}: oselect-else sons~(vector accept-alts seq-stat))
(deftree {select}: condent-call)
(defcons {condent-call}: ocondent-call sons~(vector entproc-call seq-stat seq-stat))
(deftree {select}: timent-call)
(defcons {timent-call}: otiment-call sons~(vector entproc-call seq-stat delay))
(deftree {Ada}: accept-alts)
(defcons {accept-alts}: oaccept-alts sons~(list accept-alt))
(deftree {Ada}: private)
(defcons {private}: oprivate sons~(vector bas-dec))
(deftree {typpredef}: privtyp-dec)
(defcons {privtyp-dec}: oprivtyp-dec sons~(vector ident))
(defcons {typpredef}: oexception)
(deftree {Ada}: excep-hand)
(defcons {excep-hand}: oexcep-hand sons~(list excep))
(deftree {excep-hand}: excep)
(defcons {excep}: oexcep sons~(vector choice seq-stat))
(deftree {Ada}:choice)
(defcons {choice}: ochoices (vector nomvars))
(defcons {choice}: oothers)
(deftree {decls}: u-clause)
(defcons {u-clause}: ou-clause sons~(vector nomvars))
(deftree {Ada}: w-clause)
(defcons {w-clause}: ow-clause sons~(vector nomvars))
(defcons {Ada}: repst sons~(vector typcons subp-body seq-stat))
```

Annexe C

Les modules Ada de la bibliothèque

Chaque module définit l'implantation en Ada d'un type abstrait de données. La spécification algébrique de tels types est présentée au *chapitre 5*. Il s'agit des types : Pile, File, File à double accès et Table.

1 Le type Pile

1.1 Représentation contiguë

```
generic
  type elt is private;
package pile_b is
  type pile (taille:positive:=50) is private;
  procedure creer (p:out pile);
  procedure empiler (p:in out pile;e:in elt);
  procedure depiler (p:in out pile);
  function sommet (p:in pile) return elt;
  function card (p:in pile) return natural;
  function vide (p:in pile) return boolean;
  pile_vide,pile_plein : exception;
private
  type tab is array (positive range <>) of elt;
  type pile (taille:positive:=50) is
  record
  haut:natural;
  valeur:tab(1..taille);
  end record;
end pile_b;
```

```
package body pile_b is
 procedure creer (p:out pile) is
 begin
 p.haut:=0;
 end creer;
 procedure empiler (p:in out pile;e:in elt) is
 begin
 if p.haut=p.taille then raise pile_plein;
                    else p.haut:=p.haut+1;
                         p.valeur(p.haut):=e;
 end if;
 end empiler;
 procedure depiler (p:in out pile) is
 begin
 if vide(p) then raise pile_vide;
            else p.haut:=p.haut-1;
 end if;
 end depiler;
 function sommet (p:in pile) return elt is
 begin
 if vide(p) then raise pile_vide;
            else return p.valeur(p.haut);
 end if;
 end sommet;
 function card (p:in pile) return natural is
 begin
 return p.haut;
 end card;
 function vide (p:in pile) return boolean is
 begin
 return p.haut=0;
 end vide;
end pile_b;
```


1.2 Représentation chainée

```
generic
  type elt is private;
package pile_v is
  type pile is private;
  procedure creer (p:out pile);
  procedure empiler (p:in out pile;e:in elt);
  procedure depiler (p:in out pile);
  function sommet (p:in pile) return elt;
  function card (p:in pile) return natural;
  function vide (p:in pile) return boolean;
  pile_vide : exception;
private
  type element;
  type lien is access element;
  type pile is
  record
  nbelem:natural;
  valeur:lien;
  end record;
end pile_v;
```

```
package body pile_v is
 type element is
 record
 val:elt;
 suivant:lien;
 end record;
 procedure creer (p:out pile) is
 begin
 p:=(nbelem=>0,valeur=>null);
 end creer;
 procedure empiler (p:in out pile;e:in elt) is
 begin
 p.nbelem:=p.nbelem+1;
 p.valeur:=new element'(e,p.valeur);
 end empiler;
 procedure depiler (p:in out pile) is
 begin
 if vide(p) then raise pile_vide;
            else p.valeur:=p.valeur.suivant;
                 p.nbelem:=p.nbelem-1;
 end if;
 end depiler;
 function sommet (p:in pile) return elt is
 begin
 if vide(p) then raise pile_vide;
            else return p.valeur.val;
 end if;
 end sommet;
 function card (p:in pile) return natural is
 begin
 return p.nbelem;
 end card;
 function vide (p:in pile) return boolean is
 begin
 return p.nbelem=0;
 end vide;
end pile_v;
```

2 Le type File

2.1 Représentation contiguë

```
generic
 type elt is private;
package file_b is
 type file (taille:positive:=50) is private;
 procedure creer (f:out file);
 procedure enfiler (f:in out file;e:in elt);
 procedure defiler (f:in out file);
 function tete (f:in file) return elt;
 function card (f:in file) return natural;
 function vide (f:in file) return boolean;
 file_vide,file_plein:exception;
private
 type tab is array (positive range <>) of elt;
 type file (taille:positive:=50) is
 record
 nbelem,tete,queue:natural;
 valeur:tab(1..taille);
 end record;
end file_b;
```

```
package body file_b is
 procedure creer (f:out file) is
 begin
 f.tete:=0; f.queue:=0; f.nbelem:=0;
 end creer;
 procedure enfiler (f:in out file;e:in elt) is
 begin
 if f.nbelem=f.taille then raise file_plein;
   else
   if f.queue=f.taille then f.queue:=1;
     else f.queue:=f.queue+1;
   end if;
   f.valeur(f.queue):=e; f.nbelem:=f.nbelem+1;
 end if;
 end enfiler;
 procedure defiler (f:in out file) is
 begin
 if vide(f) then raise file_vide;
   else
   if f.tete=f.taille then f.tete:=1;
     else f.tete:=f.tete+1;
   end if;
   f.nbelem:=f.nbelem-1;
 end if;
 end defiler;
 function tete (f:in file) return elt is
 begin
 if vide(f) then raise file_vide;
   else return f.valeur(f.tete);
 end if;
 end tete;
 function card (f:in file) return natural is
 begin
 return f.nbelem;
 end card;
 function vide (f:in file) return boolean is
 begin
 return f.nbelem=0;
 end vide;
end file_b;
```


2.2 Représentation chainée

```
generic
 type elt is private;
package file_v is
 type file is private;
 procedure creer (f:out file);
 procedure enfiler (f:in out file;e:in elt);
 procedure defiler (f:in out file);
 function tete (f:in file) return elt;
 function card (f:in file) return natural;
 function vide (f:in file) return boolean;
 file_vide:exception;
private
 type element;
 type lien is access element;
 type file is
 record
 nbelem:natural;
 tete,queue:lien;
 end record;
end file_v;
```

```
package body file_v is
 type element is
 record
 val:elt;
 suivant:lien;
 end record;
 procedure creer (f:out file) is
 begin
 f:=(nbelem=>0,tete=>null,queue=>null);
 end creer;
 procedure enfiler (f:in out file;e:in elt) is
 begin
 if f.tete=null
   then f.tete:=new element'(e,null);
        f.queue:=f.tete;
   else f.queue.suivant:=new element'(e,null);
        f.queue:=f.queue.suivant;
 end if;
 f.nbelem:=f.nbelem+1;
 end enfiler;
 procedure defiler (f:in out file) is
 begin
 if vide(f) then raise file_vide;
   else f.tete:=f.tete.suivant;
        f.nbelem:=f.nbelem-1;
 end if;
 end defiler;
 function tete (f:in file) return elt is
 begin
 if vide(f) then raise file_vide;
          else return f.tete.val;
 end if;
 end tete;
 function card (f:in file) return natural is
 begin
 return f.nbelem;
 end card;
 function vide (f:in file) return boolean is
 begin
 return f.nbelem=0;
 end vide;
end file_v;
```

3 Le type File à double accès

3.1 Représentation contiguë

```
generic
 type elt is private;
package filed_b is
 type file_da (taille:positive:=50) is private;
 procedure creer (fd:out file_da);
 procedure ajoutert (fd:in out file_da;e:in elt);
 procedure ajouterq (fd:in out file_da;e:in elt);
 procedure retirert (fd:in out file_da);
 procedure retirerq (fd:in out file_da);
 function extrairet (fd:in file_da) return elt;
 function extraireq (fd:in file_da) return elt;
 function vide (fd:in file_da) return boolean;
 function card (fd:in file_da) return natural;
 fd_vide, fd_plein : exception;
private
 type tab is array (positive range <>) of elt
 type file_da (taille:positive:=50) is
 record
 queue:natural;
 valeur:tab(1..taille);
 end record;
end filed_b;

package body filed_b is
 procedure creer (fd:out file_da) is
 begin
 fd.queue:=0;
 end creer;
 procedure ajoutert (fd:in out file_da;e:in elt) is
 begin
 if fd.queue=fd.taille then raise fd_plein;
 else
 fd.valeur(2..fd.queue+1):=fd.valeur(1..fd.queue);
 fd.queue:=fd.queue+1;fd.valeur(1):=e;
 end if;
 end ajoutert;
 procedure ajouterq (fd:in out file_da; e:in elt) is
 begin
 if fd.queue=fd.taille then raise fd_plein;
   else fd.valeur(fd.queue+1):=e;
        fd.queue:=fd.queue+1;
 end if;
 end ajouterq;
 procedure retirert (fd:in out file_da) is
 begin
 if vide(fd) then raise fd_vide;
 else if fd.queue=1 then fd.queue:=0;
 else
 fd.valeur(1..fd.queue-1):=fd.valeur(2..fd.queue);
 end if;
 fd.queue:=fd.queue-1;
 end if;
 end retirert;
 procedure retirerq (fd : in out file_da) is
 begin
 if vide(fd) then raise fd_vide;
   else fd.queue:=fd.queue-1;
 end if;
 end retirerq;
 function extrairet (fd:in file_da) return elt
 is
 begin
 if vide(fd) then raise fd_vide;
   else return fd.valeur(1);
 end if;
 end extrairet;
 function extraireq (fd:in file_da) return elt
 is
 begin
 if vide(fd) then raise fd_vide;
   else return fd.valeur(fd.queue);
 end if;
 end extraireq;
 function vide (fd:in file_da) return boolean is
 begin
 return fd.queue=0;
 end vide;
 function card (fd:in file_da) return natural is
 begin
 return fd.queue;
 end card;
end filed_b;
```


3.2 Représentation chainée

```
generic
 type elt is private;
package filed_v is
 type file_da is private;
 procedure creer (fd:out file_da);
 procedure ajoutert (fd:in out file_da; e:in elt);
 procedure ajouterq (fd:in out file_da; e:in elt);
 procedure retirert (fd:in out file_da);
 procedure retirerq (fd:in out file_da);
 function extrairet (fd:in file_da) return elt;
 function extraireq (fd:in file_da) return elt;
 function vide (fd:in file_da) return boolean;
 function card (fd:in file_da) return natural;
 fd_vide : exception;
private
 type element;
 type lien is access element;
 type element is record
                 val : elt;
                 suivant : lien;
                 end record;
 type file_da is record
                 nbelem : natural;
                 tete,queue : lien;
                 end record;
end filed_v;

package body filed_v is
 procedure creer (fd:out file_da) is
 begin
 fd:=(nbelem=>0,tete=>null,queue=>null);
 end creer;
 procedure ajoutert (fd:in out file_da;e:in elt) is
 begin
 if vide(fd) then fd.tete:=new element '(e,null);
                 fd.queue:=fd.tete;
   else fd.tete:=new element'(e,fd.tete);
 end if;
 fd.nbelem:=fd.nbelem+1;
 end ajoutert;
 procedure ajouterq (fd:in out file_da;e:in elt) is
 begin
 if vide(fd) then fd.tete:=new element '(e,null);
                 fd.queue:=fd.tete;
   else fd.queue.suivant:=new element'(e,null);
                 fd.queue:=fd.queue.suivant;
 end if;
 fd.nbelem:=fd.nbelem+1;
 end ajouterq;
 procedure retirert (fd:in out file_da) is
 begin
 if vide(fd) then raise fd_vide;
   else
   if fd.tete=fd.queue then fd.tete:=null;
                            fd.queue:= null;
     else fd.tete:=fd.tete.suivant;
   end if;
   fd.nbelem:=fd.nbelem-1;
 end if;
 end retirert;
 procedure retirerq (fd:in out file_da) is
 p:lieu:=fd.tete;
 begin
 if vide(fd) then raise fd_vide;
   else
   if fd.tete=fd.queue then fd.tete:=null;
                            fd.queue:= null;
     else
     while p.suivant /= fd.queue loop
     p:=p.suivant;
     end loup;
     fd.queue:=p;fd.queue.suivant:=null;
   end if;
   fd.nbelem:=fd.nbelem-1;
 end if;
 end retirerq;
 function extrairet (fd:in file_da) return elt is
 begin
 if vide(fd) then raise fd_vide;
   else return fd.tete.val;
 end if;
 end extrairet;
 function extraireq (fd:in file_da) return elt is
 begin
 if vide(fd) then raise fd_vide;
   else return fd.queue.val;
 end if;
 end extraireq;
 function vide (fd:in file_da) return boolean is
 begin
 return fd.tete=null;
 end vide;
 function card (fd:in file_da) return natural is
 begin
 return fd.nbelem;
 end card;
end filed_v;
```

4 Le type Table

```
generic
   type elt is private;
package gest_table is
   type table (taille:positive:=50) is private;
   procedure creer (t:out table);
   procedure ajouter (t:in out table; i:in positive; e:in elt);
   procedure retirer (t : in out table; i:in positive);
   function extraire (t : in table; i:in positive) return elt;
   function vide (t : in table) return boolean;
   function card (t : in table) return natural;
   function dans (t:in table;i:in positive) return boolean;
   t_vide,inexistant : exception;
private
   type element;
   type liste is access element;
   type tab is array (positive range <>) of liste;
   type table (taille:positive:=50) is
   record
   nbelem:natural;
   valeur:tab(1..taille);
   end record;
   type element is
   record
   indice:natural;
   val:elt;
   suivant:liste;
   end record;
end gest_table;
```



```
package body gest_table is
procedure creer (t:out table) is
begin
t.nbelem:=0; t.valeur:=(1..t.taille=>null);
end creer;
procedure ajouter (t:in out table; i:in positive; e:in elt) is
place:natural:=(i mod t.taille)+1;
begin
if t.valeur(place)=null then t.valeur(place):=new element'(i,e,null);
                        else t.valeur(place):=new element'(i,e,t.valeur(place));
end if;
t.nbelem:=t.nbelem+1;
end ajouter;
procedure cherche (t:in table;trouve:out boolean; preced:in out liste;i:in positive) is
place:natural:=(i mod t.taille)+1;
p:liste;
begin
trouve:=false; preced:=null; p:=t.valeur(place);
while p/=null and not trouve loop
      if p.indice=i then trouve:=true;
                    else if p.indice<i then p:=null;
                                       else preced:=p;
                                            p:=p.suivant;
                    end if;
      end if;
end loop;
end cherche;
procedure retirer (t:in out table; i:in positive) is
trouve:boolean;
preced:liste;
begin
cherche(t.trouve,preced,i);
if trouve=false then raise inexistant;
  else if preced=null
         then t.valeur((i mod t.taille)+1):=t.valeur((i mod t.taille)+1).suivant;
         else preced.suivant:=preced.suivant.suivant;
  end if;
t.nbelem:=t.nbelem-1;
end if;
end retirer;
```

```
function extraire (t:in table; i:in positive) return elt is
trouve:boolean;
preced:liste;
begin
cherche(t,trouve,preced,i);
if trouve=false then raise inexistant;
  else if preced=null
         then return t.valeur((i mod t.taille)+1).val;
         else return preced.suivant.val;
  end if;
end if;
end extraire;
function vide (t : in table) return boolean is
begin
return (t.nbelem=0);
end vide;
function card (t : in table) return natural is
begin
return t.nbelem;
end card;
function dans (t:in table;i:in positive) return boolean is
trouve:boolean;
preced:liste;
begin
cherche(t,trouve,preced,i);
return trouve;
end dans;
end gest_table;
```

Bibliographie

[ADA 86] J.M. ADAMO et J. BONNEVILLE. Cool : manuel de référence. Rapport Université de Lyon, 1986.

[ADA 89] J.M. ADAMO. A c++ based language for distributed and real-time programming. *11th OUG meeting*, Edimbourg, 1989.

[ALE 88] F. ALEXANDRE, J.P. FINANCE, et A. QUERE. Spes : un système de transformation de programmes logiques. *Colloque de Programmation Logique*, Trégastel, 1988.

[AND 79] A. ANDLER. Predicate path expressions. *6th Annual ACM Symposium on Principles of Programming Languages*, pages 226–236, 1979.

[ARS 82] J. ARSAC et Y. KODRATOFF. Some techniques for recursion removal from recursive functions. *ACM Transactions on Programming Languages and Systems*, 4(2):295–322, 1982.

[BAR 79] D. BARSTOW. *Knowledge-Based Program Construction.* Elsevier North Holland, 1979.

[BAR 81] U. BARTELS, W. OLTHOFF, et P. RAULEFS. Ape: An expert system for automatic programming from abstract specifications of data types and algorithms. MEMO SEKI-BN-81-01, Institut fur Informatik, Universitat Kaiserslautern, 1981.

[BAR 82] U. BARTELS, W. OLTHOFF, et P. RAULEFS. An expert system for implementing abstract sorting algorithms on parametrized abstract data types. *7th IJCAI*, Vancouver, Canada, 1982.

[BER 85] G. BERRY et L. COSSERAT. The esterel synchronous programming language and its mathematical semantics. *LNCS Springer Verlag*, 197, 1985.

[BID 87] M. BIDOIT et al. Asspro : un environnement de programmation interactif et intégré. *TSI*, 6(1):21–40, 1987.

[BOO 84] G. BOOCH. *Software Engineering with Ada.* Benjamin/Cummings Pub. Comp, 1984.

[BOO 87] G. BOOCH. *Software Components with Ada: Structures, Tools and Subsystems.* Benjamin/Cummings Pub. Comp, 1987.

[BOR 87] P. BORRAS et al. Centaur: the system. INRIA Report 777, 1987.

[BRO 81] M. BROY et P. PEPPER. Program development as a formal activity. *IEEE Transaction on Software Engineering*, SE-7(1):14–22, 1981.

[BUR 77] R.M. BURSTALL et J. DARLINGTON. A transformation system for developping recursive programs. *Journal of the ACM*, 24(1), 1977.

[CAM 74] R.H CAMPBELL et A.N HABERMAN. The specification of process synchronizations by path expression. *Lectures Notes in Computer Sciences*, 16, 1974.

[CAS 87] P. CASPI et al. Lustre : a declarative language for programming synchronous systems. *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, ACM. Janvier, 1987.

[CER 85] P. Le CERTEN. Lc3, un langage de programmation parallèle. *Actes des journées AFCET-GROPLAN*, 1985.

[CHE 84] T.E. CHEATHAM. Reusability through program transformations. *IEEE Transaction on Software Engineering*, SE-10(5):589–594, 1984.

[CLE 85] D. CLEMENT et al. Natural semantics on the computer. INRIA Report 416, 1985.

[DAR 76] J. DARLINGTON et R.M. BURSTALL. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

[DER 79] J.C. DERNIAME et J.P. FINANCE. Types abstraits de données : spécification, utilisation et réalisation. Ecole d'été de L'AFCET Monastir, Rapport CRIN 79.E.57, 1979.

[DES 84] T. DESPEYROUX. Spécifications sémantiques dans le système mentor. Thèse de 3ème cycle, université de Paris-Sud, ORSAY, 1984.

[DES 86] T. DESPEYROUX. Typol : a formalism to implement natural semantics. INRIA Report 94, 1986.

[DIJ 75] E.W. DIJKSTRA. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8):354–457, 1975.

[DOD 83] DOD. Reference manual for the ada programming language. ANSI/MIL-std 1815-a. U.S. Department of Defense, 1983.

[FEA 86] M.S. FEATHER. A survey and classification of some program transformation, approaches and techniques. *IFIP PC2 Working Conference on Program Specification and Transformation*, BAD-TOLZ, FRG, 1986.

[FIN 78] J.P. FINANCE. De la spécification abstraite d'une donnée à sa représentation en mémoire: les états successifs d'une information. *Actes du congrès AFCET, Editions Hommes et Techniques*, 1978.

[FIN 79] J.P FINANCE. Etude de la construction des programmes: méthodes et langages de spécification et de résolution de problèmes. Thèse d'etat, Université de Nancy I, 1979.

[FIN 83] J.P. FINANCE, J. JULLIAND, et G.R. PERRIN. Conception et expression de types de communication entre processus. Rapport de recherche CRIN, 83-R-069, 1983.

[FLO 76] L. FLON et N. HABERMAN. Towards the construction of verifiable software systems. *Sigplan Notices*, 2:141–148, 1976.

[FOI 85] J. FOISSEAU et al. Le système sprac : expression et gestion de spécifications, d'algorithmes et de représentations. *TSI*, 4(2):237–254, 1985.

[FUT 87] K. FUTATSUGI, J.A. GOGUEN, J. MESEGUER, et K. OKADA. Prameterized programming in obj2. *ACM*, pages 51–60, USA, 1987.

[GAU 78] M.C. GAUDEL et G. TERRINE. Synthèse de la représentation d'un type abstrait par des types concrets. *Congrès AFCET panorama de l'informatique*, pages 434–445, Paris, 1978.

[GAU 80] M.C GAUDEL. Génération et preuve de compilateurs basées sur une sémantique formelle des langages de programmation. Thèse d'etat, Institut national polytechnique de Lorraine, 1980.

[GER 84] R. GERTH et W.P DE ROEVER. A proof system for concurrent ada programs. *Science of Computer Programming*, 4(2), 1984.

[GLO 83] A. GLODBERG et A. ROBSON. *Smalltalk80 : the Language and its Implementation.* Addison Wesley, 1983.

[GOG 79] J.A. GOGUEN et J. TARDO. An introduction to obj : a language for writing and testing software specification. Specification of reliable software IEEE Catalog CH-1401-96., 1979.

[GOG 86] J.A. GOGUEN. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16–28, 1986.

[GRA 86] A. GRAM. *Raisonnner pour programmer*. Dunod, 1986.

[GRA 88] M. GRANDBASTIEN. Une approche à base de connaissances pour l'enseignement de la programmation, conception et réalisation de saida: système d'aide a' l'implantation de données abstraite. Thèse d'etat, Université de Nancy I, 1988.

[GRE 88] C. GRESSE. Représentation, communication et formalisation de développements de programmes: une etude synthétique. *Actes des journées internationalles, le génie logiciel et ses applications*, pages 1285–1302, Toulouse, France, 1988.

[GUE 80] P. LE GUERNIC et M. RAYNAL. L'expression de la communication dans les langages : des analyses ct des prpopositions. Rappport de Recherche IRISA 129, 1980.

[GUE 84] P. LE GUERNIC, A. BENVENISTE, et T. GAUTIER. Signal : un langage pour le traitement du signal. *Traitement du Signal*, 1(1), 1984.

[GUT 78] J.V. GUTTAG et J.J HORNING. The algebraic specifications of abstract data types. *Acta Informatica*, 10:27–62, 1978.

[HAN 75] P. BRINCH HANSEN. The programming language concurrent pascal. *IEEE Transaction on Software Engineering*, SE-1(2), 1975.

[HER 80] D. HERMAN. Compilation d'un langage de haut niveau pour la programmation des synchronisations. Bulletin Bigre 20, 1980.

[HOA 78] C.A.R. HOARE. Communicating sequential processes. *CACM*, 21(8):666-677, 1978.

[HOW 76] J.H. HOWARD. Proving monitors. *CACM*, 19(5):273–279, 1976.

[HUL 83] J.M HULLOT. Ceyx, a multiformalism programming environment. *IFIP 83*, Paris, 1983.

[HUL 87] R. HULL et R. KING. Semantic database modeling: Surveys, applications and researche issues. *ACM Computing Survey*, 19(3):201–260, 1987.

[ICH 79] J.D. ICHBIAH et al. Rationale for the design of ada programming language. *Sigplan Notices*, 14(6), juin 1979.

[JAN 87] R. JANICKI. A formal semantics for concurrent systems with a priority relation. *Acta Informatica*, 24(1):33–55, 1987.

[JAR 88] J. JARAY. Timde Specifications for the Development of Real-Time Systems. *Symp. on Formal Techniques in Real-Time and Fault Tolerance Systems*, Warwick, Septembre 1988.

[JUL 83] J. JULLIAND. Spécification algébrique de la communication entre processus parallèles. *TSI*, 2(4):257–269, 1983.

[JUL 85] J. JULLIAND. Comedie: manuel d'utilisation. Rapport CRIN 85-R-075, 1985.

[JUL 90] J. JULLIAND et G.R. PERRIN. Asynchronous functional parallel programs. *International Conference on Computing and Information*, Ontario, Canada, 1990.

[KAH 87] G. KAHN. Naturals semantics. *Proc. of Symp. on Theorical Aspects of Computer Science*, pages 21–28, Passau, Germany, 1987.

[KAN 84] E. KANT et D.R. BARSTOW. The refinement paradigm : the interaction of coding and efficiency knowledge in program synthesis. Interactive Programming Environments, Mac Graw Hill, New York, 1984.

[KHA 90] T. KHAMMACI et N. BOUDJLIDA. An object-constructor database model to software process modeling. *Fifth International Symposium On Cumputer and Information Sciences*, Cappadocia, Turkey, 1990.

[KOU 86] A. KOUKAM, J.P. FINANCE, J. JULLIAND, et G.R. PERRIN. Une méthode de dérivation de programmes ada. Rapport CRIN R-86-73, 1986.

[KOU 88] A. KOUKAM, J.P. FINANCE, J. JULLIAND, et G.R. PERRIN. Spécification et réalisation des types de communication. *Actes du Premier Séminaire International de Génie Logiciel*, Oran, Algérie, 1988.

[KOU 90] A. KOUKAM. A hierarchical structure for reusing software components. *International Conference on Computing and Information*, Ontario, Canada, 1990.

[KRO 87] F. KROGER. Abstract modules: combining algebraic and temporal logic specification means. *TSI*, 6(6), 1987.

[LAM 83] L. LAMPORT. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 3(2), 1983.

[LAM 85] L. LAMPORT. An axiomatic semantics of concurrent programming languages. *Nato ASI Series, Logics and Models of Cocurrent Systrems*, F13, 1985.

[LAU 79] P.E LAUER, P.R TORRIGIAN, et M.W SHIELDS. Cosy : A system specification language based on paths and process. *Acta Informatica*, 12(2):109–158, 1979.

[LAZ 89] N. LAZRAK. Pause : A theorem prover for elementary properties in equational specifications. *Proceedings of the Workshop on Computer-aided Development of Proofs, Theories, Programs, and Circuits*, pages 17–19, Universidad Complutense de Madrid, 1989.

[LEV 84] N. LEVY. Outils d'aide à la construction et transformation de types abstraits algébriques. Thèse de 3ème cycle, université de Nancy I, 1984.

[LIT 84] S.D. LITVINTCHOUK et A.S. MATSUMOTO. Design of ada systems yielding reusable components: An approach using structured algebraic specification. *IEEE Transactions on Software Engineering*, SE-10(5):544–551, 1984.

[MAN 82] Z. MANNA et A. PNUELI. Verification of concurrent programs : a temporal proof system. *Proc. 4th School on advanced programming*, pages 10–14, Holland, 1982.

[MAN 84] Z. MANNA et P.L. WOLPER. Synthesis of communicating process from temporal logic specifications. *ACM Toplas*, 6(1):68–93, 1984.

[MAR 89] M. MARMONIER. Spécification et validation de système de processus communicants. Doctorat de l'université de Nancy I, 1989.

[MAY 83] D. MAY. The occam language. *Sigplan Notice*, 18(4), 1983.

[MEY 85] B. MEYER. The software knowledge base. *Proceedings of the 8th International Conference on Software Engineering*, London, 1985.

[MEY 86] B. MEYER. Software reusability : the case for object-oriented design. Report TR-86-06, Interactive Software Engineering, Santa Barbara (California), 1986.

[MUR 88] M. MURRAY et I. SOMMERVILLE. An informatioon retrieval system for software components. *Software Engineering Journal*, September:198–207, 1988.

[OWI 79] S. OWICKI. Specifications and proofs for abstract data types in concurrent programs. *Springer Verlag*. LNCS 69:174–197, 1979.

[OWI 82] S. OWICKI et L. LAMPORT. Proving livenes properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.

[PER 83] G.R. PERRIN. Specification and verification of communications of processes. Rapport de recherche CRIN, 82-R-020, 1983.

[PER 85] G.R. PERRIN. La communication: un outil pour la spécification, la construction et la vérification de système parallèles. Thèse d'etat, Université de Nancy I, 1985.

[PER 87] G.R. PERRIN. Programmation parallèle : point de vue sur les langages et les méthodes. *TSI*, 6(2), 1987.

[PLO 81] G. PLOTKIN. A structural approach to operational semantics. Report DAIMI FN. 19, Computer Science Dpt Aarhus Univ, 1981.

[PRO 82] K. PROCH. Orsec : un outil de recherche de spécifications equivalentes par comparaison d'exemples. Thèse de trosième cycle, Université de Nancy I, 1982.

[RAM 83] K. RAMAMRITHAM et R.M. KELLER. Specification of synchronizing processes. *IEEE Transactions on Software Engineering*, SE-9(6):722–733, 1983.

[RAY 81] R. RAYNAL. Contribution aà l'étude de la coopération entre processus dans les langages et les sytèmes informatiques. Thèse d'Etat, Université de Rennes, 1981.

[RAY 82] M. RAYNAL. Une analyse de la spécification de la coopération entre processus par variables partagées. *TSI*, 1(3):201–210, 1982.

[REM 82] J.L REMY. Etude de systèmes de réécriture conditionnelle et applications aux types abstraits algébriques. Thèse d'etat, Institut national polytechnique de Lorraine, 1982.

[ROB 77] P. ROBERT et J.P VERJUS. Towards autonomous descriptions of synchronization modules. *Proc. IFIP congress*, pages 981–986, NORTH HOLLAND, 1977.

[SCH 87] A. SCHIPER. Programmation concurrente. Presses Polytechniques Romandes, 1987.

[THO 86] M. THORIN. *Manuel Ada, Langage Normalisé Complet.* MASSON, 1986.

[WIL 81] D.S. WILE. Type transformations. *IEEE Transactions on Software Engineering*, SE-7(1):32–39, 1981.

NOM DE L'ETUDIANT : KOUKAM Abderrafiaa

NATURE DE LA THESE : DOCTORAT de l'UNIVERSITE NANCY I INFORMATIQUE

VU, APPROUVE ET PERMIS D'IMPRIMER

NANCY, le 12 DEC. 1990

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

M. BOULANGE

RESUME :

Le travail présenté dans cette thèse s'inscrit dans le cadre d'une démarche méthodique pour la construction de systèmes parallèles. L'idée de base est de définir de tels systèmes comme un ensemble d'entités manipulant des suites de données, entre lesquelles des relations dites de communication sont spécifiées. Pour concevoir les solutions déduites de cette démarche, un langage d'expression de systèmes parallèles est proposé. Ce langage est caractérisé par :

- une expression abstraite des relations de communication en termes de types abstraits algébriques appelés types de communication.
- une conception modulaire fondée sur la séparation de la description des communications de l'écriture des processus.

Notre premier objectif est d'élaborer une démarche pour automatiser la transformation des types de communication en programmes Ada. Mais la distance est grande entre cette expression abstraite et statique de la communication et le programme final dynamique et efficace. Aussi avons-nous été conduit à décomposer le processus de transformation en plusieurs étapes fondées sur des règles de transformation. Deux classes de règles sont ainsi mises en évidence :

- les règles qui permettent de représenter un type de communication par des structures de données proches du langage Ada.
- les règles qui introduisent les outils nécessaires à l'implantation des types de communication dans un environnement parallèle.

Nous proposons ensuite de transformer les processus utilisant les types de communication en tâches Ada. Cette transformation est spécifiée en termes de règles d'inférence.

Notre étude s'est concrétisée aussi par l'implantation d'un système de transformation. Réalisé en Lisp, ce système met en oeuvre les règles de transformation et s'intègre au sein d'un environnement de programmation composé d'un éditeur syntaxique et d'un interprète de systèmes parallèles. Enfin, nous proposons une extension de cet environnement pour prendre en considération la réutilisation des types de communication.

MOTS CLES : Transformation, Parallélisme, Communication, Type Abstrait de Données, Réutilisabilité.