ARCHE : un langage parallèle à objets fortement types

Marc Benveniste, Valérie Issarny

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1992

25ème anniversaire

N° 1646

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

ARCHE : UN LANGAGE PARALLÈLE À OBJETS FORTEMENT TYPÉS

Marc BENVENISTE
Valérie ISSARNY

Mars 1992

RR-1646

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Arche : un langage parallèle à objets fortement typés

Arche: a strongly-typed parallel object-oriented language

Marc Benveniste[1]
Valérie Issarny[2]
IRISA/INRIA-Rennes
Campus de Beaulieu, 35042 Rennes Cedex, FRANCE
e-mail: mbenveni/issarny@irisa.fr

Publication Interne n°642, 132 pages - Programme 1
17 mars 1992

Résumé

ARCHE est le successeur du langage POLYGOTH, fruit des premiers travaux de l'équipe Langages et Systèmes Parallèles de l'IRISA dans l'approche objet de la programmation parallèle. Cette première expérience visait à exprimer la gestion explicite de la distribution et de la réplication, tant des données permanentes que des calculs. Le langage POLYGOTH propose une intégration du parallélisme et des objets originale mais mal adaptée aux mécanismes de synchronisation, d'héritage et de traitement d'exceptions.

Les objets dans le langage ARCHE sont plus simples. Ils s'inscrivent dans la lignée de l'école scandinave. Cette simplicité facilite l'intégration des différents mécanismes nécessaires à l'approche objet de la programmation, et à la gestion explicite du parallélisme. Parmi les points saillants de ARCHE, remarquons les originalités suivantes :

- intégration du parallélisme, de l'héritage et des types abstraits dans un langage impératif fortement typé,
- généralisation de l'invocation de méthode à une séquence d'objets,
- introduction d'un mécanisme de traitement d'exceptions adapté aux contextes parallèles imbriqués.

La première partie de ce rapport présente le concept de multiprocédure, son intégration dans le langage POLYGOTH, et souligne les difficultés nommées ci-dessus. La deuxième partie expose les principales caractéristiques de ARCHE. Tout d'abord, les objets, les types et les classes sont définis, puis trois chapitres présentent respectivement le parallélisme, l'héritage de classe ainsi que la spécialisation de type, et le traitement d'exceptions.

[1]Bourse du XIIème programme franco-mexicain CONACYT-CEFI.
[2]Bourse INRIA-Région.

Abstract

ARCHE is the POLYGOTH language successor, designed in the "Langages et Systèmes Parallèles" team at IRISA as the result of a first effort towards understanding concurrent object-oriented programming. The design of POLYGOTH was aimed at the explicit management of both data and computation distribution and replication. A novel approach for integrating parallelism and objects was proposed, although it proves to be poor as far as synchronization, inheritance and exception handling are concerned.

The ARCHE object is simpler and closer to those of the scandinavian school languages. This simple object model allows for a satisfactory integration of the mechanisms that both the object-oriented approach and the explicit management of concurrency require. Among ARCHE main aspects, the following novelties can be pointed out:

- integration of concurrency, inheritance and abstract types in an imperative and strongly-typed language,
- generalization of the method call to a sequence of objects,
- introduction of an exception handling mechanism that handles nested parallelism.

The first part of this report presents the multiprocedure concept and its integration in the POLYGOTH language, and shows the drawbacks mentioned above. The second part describes the main aspects of ARCHE. The notions of object, type and class are initially defined, then parallelism, inheritance and sub-typing, and exception handling are presented in three separate chapters.

Table des Matières

Avant-propos 5

Le contexte historique 7

I Multiprocédure 9

I.1 Déclaration et activation de multiprocédure 9

I.2 Composition de multiprocédure : l'appel coordonné 11

II Le langage POLYGOTH 13

II.1 Fragmentation des objets 13

II.2 Parallélisme 16

II.3 Types 17

Le langage ARCHE 19

III Objets fortement typés 21

III.1 Présentation 21

III.1.1 L'objet 21

III.1.2 La vue 22

III.1.3 La classe 22

III.1.4 Les unités de compilation 22

III.2 Types concrets 23

III.2.1 Types ordinaux 23

III.2.2 Ensembles 24

III.2.3 Structures 25

III.3 Type vue 26
III.3.1 Signature de méthode 27
III.3.2 Vues remarquables 28
III.4 Classe 33
III.4.1 Déclarations 36
III.4.2 Création et recopie 39
III.4.3 Commandes 40
III.4.4 Expressions 41
IV Parallélisme **45**
IV.1 Processus 45
IV.2 Communication 47
IV.3 Synchronisation 48
IV.4 Plus de disponibilité 55
IV.5 Plus de simultanéité 60
IV.5.1 Appel coordonné 66
V Sous-types et héritage **73**
V.1 Sous-typage 73
V.1.1 Relation de sous-typage 73
V.1.2 Spécialisation des vues 75
V.1.3 Relation d'affectabilité 78
V.2 Sous-classage 79
V.3 Discussion 82
VI Traitement d'exceptions **85**
VI.1 Modèle coopération de traitement des exceptions 85
VI.1.1 Traitement des exceptions globales 86
VI.1.2 Traitement des exceptions concertées 86
VI.2 Expression du traitement des exceptions 88
VI.2.1 Définition des exceptions 88
VI.2.2 Réalisation du traitement des exceptions 95
VI.2.3 Signalement des exceptions 100

VI.3 Discussion 103
VII Conclusion **105**

Annexes **113**
A Syntaxe **113**

Avant-propos

ARCHE est un langage de recherche, «grandeur nature», conçu dans le cadre du projet GOTHIC [1] qui s'inscrit dans la lignée des langages de type ALGOL. Plus précisément, il descend de MODULA-2 [2]. Il est le successeur du langage POLYGOTH [3, 4, 5], lui même fruit des premiers travaux de l'équipe Langages et Systèmes Parallèles de l'IRISA dans l'approche objet de la programmation parallèle.

Bien que la définition de ARCHE ne soit pas encore établie par une sémantique formelle, certaines idées novatrices méritent d'être rapportées. Il nous paraît difficile de présenter ARCHE sans le placer dans son contexte historique car certains choix de conception sont directement hérités de ses ancêtres. Aussi, nous avons structuré ce rapport en deux parties. La première présente le concept de multiprocédure et son intégration dans le langage POLYGOTH. La deuxième expose les principales caractéristiques de ARCHE. Le lecteur qui aurait suivi les développements du projet GOTHIC peut donc commencer par la deuxième partie de ce rapport.

Ce rapport s'adresse aux lecteurs intéressés par les problèmes que posent l'intégration du parallélisme et de l'héritage dans un langage de programmation. Il s'adresse aussi aux lecteurs qui voudraient mettre en œuvre le langage ARCHE. Enfin, les lecteurs trouveront dans ce rapport l'ébauche d'un manuel de programmation ARCHE. Il aurait été sans doute préférable de produire trois documents, chacun adressé à une catégorie de lecteurs. Nous avons manqué de temps et ce rapport est un compromis. En conséquence, nous anticipons quelques déceptions de la part de nos lecteurs.

Remerciements

La définition d'un langage de programmation demande un gros effort de réalisation et exige un travail d'équipe. Nous tenons à remercier Philippe Lecler, Isabelle Puaut et Jean-Paul Routeau du travail qu'ils ont effectué pour mettre en œuvre le langage ARCHE. La qualité de ce rapport s'est sans aucun doute améliorée grâce aux commentaires de Maurice Jégado et d'Hector Ruiz Barradas que nous remercions aussi.

Le contexte historique

Chapitre I

Multiprocédure

La notion de *multiprocédure* est le résultat d'une approche proposée dans [6] visant l'intégration du parallélisme et des procédures : la généralisation de l'imbrication de blocs [7]. La procédure permet d'abstraire la mise en œuvre d'un calcul et de n'en retenir que ses pré- et post-conditions. Elle favorise la définition paramétrique de celui-ci en fournissant un mécanisme de déclaration et de passage de paramètres. Le principal moyen de composition des procédures est l'imbrication. La multiprocédure retient ces propriétés dans le cadre du parallélisme. Elle permet de déclarer, de façon paramétrique, des calculs parallèles et de les composer par imbrication.

I.1 Déclaration et activation de multiprocédure

Une déclaration de multiprocédure comprend un en-tête et un corps. L'en-tête est identique à celui de la procédure. On y indique le nom, les paramètres et le résultat de la multiprocédure.

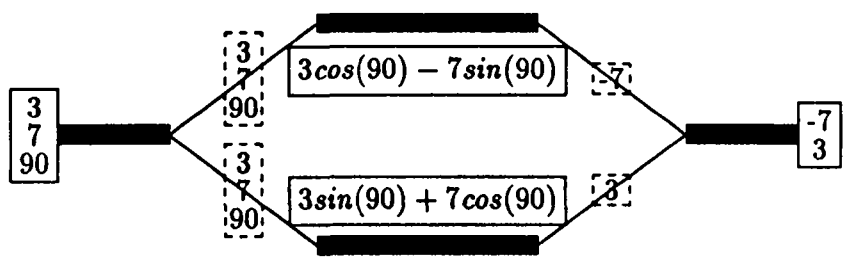
Le corps est fait d'un ensemble de blocs. Dans chaque bloc est précisé le sous-ensemble des paramètres utilisés ainsi que sa contribution au résultat. Un bloc, délimité par un couple «`BEGIN...END`», se compose d'un ensemble de déclarations (locales) et d'une séquence d'instructions. Si nous considérons que le passage de paramètres se fait par valeur, les blocs du corps sont totalement indépendants. Pour des raisons de simplicité, nous n'abordons pas l'imbrication statique de multiprocédures. A titre d'exemple, nous présentons la multiprocédure «`Rotation`» qui calcule la rotation d'un point dans le plan.

Exemple I.1

```
MPROC Rotation( x,y: FLOAT; alpha: Degree ):( u,v: FLOAT ) =
BEGIN ( x,y,alpha ): ( u )
 RETURN x * cos(alpha) - y * sin(alpha)
END ||
BEGIN ( x,y,alpha ): ( v )
 RETURN x * sin(alpha) + y * cos(alpha)
END;
```

L'activation d'une multiprocédure crée une *famille* de contextes d'exécution disjoints, un contexte par bloc composant. Nous appelons cette famille un *multicontexte.* Les valeurs des paramètres à l'appel sont distribués aux différents composants suivant la déclaration faite par chaque bloc. Les corps des composants sont exécutés en parallèle. Les résultats sont collectés d'après les déclarations, puis ils sont transmis à l'appelant. Le multicontexte est alors abandonné. Le schéma suivant illustre l'évaluation de «`Rotation( 3,7,90 )`» :

I.1...



Il est important de remarquer que la multiprocédure offre à la fois une structure de contrôle (création et changement de contexte d'exécution) et l'abstraction d'un schéma de communication à multiples partenaires (passage de paramètres et collecte de résultats). L'aspect communication est souligné dans la figure de l'exemple I.1 par les traits fins, liant les contextes, sur lesquels sont placées les valeurs correspondant à l'appel.

I.2 Composition de multiprocédure : l'appel coordonné

Tout comme les procédures, les multiprocédures se composent par imbrication. La généralisation de l'appel de procédure retenue traduit le fait qu'une multiprocédure constitue une unité. En effet, tous les composants d'une multiprocédure participent à un appel unique. L'appel partiel réalisé par chacun des composants induit une forte synchronisation de tous les composants, suivie de l'imbrication de l'appelée. Cet appel est l'*appel coordonné*. L'appel traditionnel en est un cas particulier. Le mécanisme de passage de paramètres et de retour de résultats devient une puissante abstraction de communication. Afin de l'illustrer, nous présentons une programmation originale de la famille de fonctions définies par les équations suivantes :

Exemple I.2

$$
\begin{aligned}
f(0) &= (p_0, q_0) && (1)\\
f(1) &= (p_1, q_1) && (2)\\
f(n+2) &= (h(p,q), g(p,q)) \text{ où } (p,q) = f(n) \text{ pour tout } n \geq 0 && (3)
\end{aligned}
$$

Nous montrons tout d'abord une programmation traditionnelle de cette famille de fonctions à l'aide de la commande «PARBEGIN...PAREND». Nous supposons que deux procédures «H» et «G» sont disponibles, toutes les deux ayant la signature «(INT,INT) : (INT)».

I.2...

```
PROC F(n: INT): (r,s : INT) =
 VAR p,q: INT;
     a,b: INT;
BEGIN
 IF    n = 0 THEN RETURN (p0,q0)
 ELSIF n = 1 THEN RETURN (p1,q1)
             ELSE (p,q) := F(n-2) ;
                  PARBEGIN  a := H(p,q) || b := G(p,q) PAREND ;
                  RETURN (a,b)
 END
END;
```

Nous pouvons remarquer que les variables «a» et «b» servent de liens entre le contexte d'exécution de la commande parallèle et celui de la procédure. Ci-après,

nous présentons une programmation à l'aide d'une multiprocédure.

I.2...

```
MPROC F(n : INT): (r,s : INT) =
BEGIN (n): (r)
 IF    n = 0 THEN RETURN (p0,_)
 ELSIF n = 1 THEN RETURN (p1,_)
             ELSE RETURN (G( F( n-2 ) ), _)
 END
END ||
BEGIN (n): (s)
 IF    n = 0 THEN RETURN (_,q0)
 ELSIF n = 1 THEN RETURN (_,q1)
             ELSE RETURN (_ , H( F( _ ) ))
 END
END;
```

Nous employons la même notation pour l'appel partiel et pour son retour. La correspondance entre une valeur fournie par un appel partiel et un paramètre formel est établie par l'identité de leur ordre respectif d'apparition textuelle. Le symbole « _ » sert à désigner les emplacements vides. Un ensemble d'appels partiels (resp. retours) forment un appel (resp. retour) coordonnée si, et seulement si, *(i)* chaque composant d'une même multiprocédure en fournit un élément, et *(ii)* il existe une bijection entre les valeurs fournies et les paramètres formels qui respecte leur correspondance [8, 9]. Ainsi, les appels partiels «`F(n - 2)`» et «`F( _ )`» forment un appel coordonné, et «`RETURN (p0,_)`» et «`RETURN (_,q0)`» forment un retour.

La multiprocédure permet la composition de phases de calculs parallèles au moyen de l'imbrication dynamique. Ceci simplifie grandement la tâche du programmeur en lui évitant la gestion explicite de variables partagées par imbrication statique, introduites pour permettre aux différentes phases parallèles de propager leurs calculs. L'exemple présenté n'est sans doute pas le meilleur pour montrer l'intérêt de cette gestion automatisée des contextes parallèles, sa récursivité étant terminale droite. Il suffit cependant de s'interroger sur le rôle tenu par les variables «`a`» et «`b`» dans l'exemple I.2 pour s'apercevoir de l'apport des multiprocédures.

Nous ne développons pas d'avantage cette présentation des multiprocédures. Elles sont le sujet de travaux en cours [9]. Nous présentons sommairement le langage POLYGOTH, fondé sur une extension du concept de multiprocédure à la notion d'objet. Cette extension est motivée par le rapprochement que l'on peut faire entre la notion d'objet et celle de procédure persistante [10]. Le chapitre suivant mène plus avant cette réflexion.

Chapitre II

Le langage Polygoth

Le langage POLYGOTH propose une intégration du parallélisme et des objets fondée sur le concept de multiprocédure. La conception de ce langage visait à exprimer la gestion explicite de la distribution et de la réplication, tant des données permanentes que des calculs, et à la fois, à fournir les notations permettant de s'en abstraire dans l'utilisation des dites données. La notion de module y est étendue de manière à pouvoir exprimer la création d'exemplaires et la distribution spatiale de son état. Cette extension est à l'origine de la classe et de son mécanisme de fragmentation. POLYGOTH introduit la notion d'objet *fragmenté* dont les méthodes sont des multiprocédures et l'état est composé de blocs parallèles appelés *fragments*. Dans les paragraphes suivants, nous limitons notre présentation de POLYGOTH aux trois aspects : fragmentation, parallélisme et système de types. Nous renvoyons le lecteur recherchant une description plus complète de ce langage à la bibliographie [3, 4, 5].

II.1 Fragmentation des objets

Le mécanisme de fragmentation est une extension de la notion de multiprocédure à celle de module. La notion de module peut être rapprochée de celle de procédure *persistante*. Un module peut être assimilé au contexte d'exécution d'une procédure qui serait accessible au terme de celle-ci. Les données d'un module seraient les variables locales de la procédure et les opérations seraient ses procédures. Un objet fragmenté est à la multiprocédure ce que le module est à la procédure. L'état d'un objet fragmenté est un ensemble de données logiquement réparties. Le traitement de ces données n'est effectué que par appel de méthode. Une méthode est tout naturellement réalisée par une multiprocédure. La multiprocédure permet à la fois d'exprimer une distribution des calculs qui épouse parfaitement la répartition des données traitées, et d'abstraire cette répartition

dans l'utilisation des calculs. Le traitement des données placées sur un fragment constitue un bloc de la multiprocédure réalisant le traitement global des données réparties. Lorsque le traitement ne peut pas être effectué localement, un appel coordonné permet l'accès aux informations détenues par les autres fragments.

La notation retenue dans le langage POLYGOTH pour exprimer cette distribution est la déclaration partielle des variables et des multiprocédures. Ainsi, dans chaque fragment figurent des déclarations privées et des déclarations partielles. Ces dernières sont distinguées syntaxiquement par la présence du mot-clef «FRAGT». Nous reproduisons l'exemple fourni dans [5, pp. 88] d'une base de donnée répartie sur deux machines :

Exemple II.1

```
IMPLEMENTATION CLASS DataBase;
 FROM Wherever IMPORT InfoType, QuerryType;
...
CONST N = ...;              (* nombre maximal d'enregistrements *)
TYPE
 ItemType     = RECORD free: BOOLEAN; info: InfoType END;
 DataBaseType = ARRAY [ 1 .. N ] OF ItemType;
...
FRAGMENT Machine1; || FRAGMENT Machine2;
...
END DataBase.
```

Cet exemple montre la distribution explicite de la base de données sur deux fragments, nommés «Machine1» et «Machine2», que nous détaillons ci-après.

II.1...

```
FRAGMENT Machine1;
 CONST Lo = 1; Hi = N DIV 2;
 VAR FRAGT data : DataBaseType[[Lo .. Hi]];
 VAR i : INTEGER;
 MULTIPROC FRAGT Insert ( this : ARRAY OF InfoType );
 ...
 BEGIN                    (* initialisation du fragment *)
  FOR i := Lo TO Hi DO data[i].free := TRUE END
 END Machine1;
```

Dans chacun des fragments, la variable «data» est une déclaration partielle de la base de données. La déclaration d'un bloc de la multiprocédure «insert» dans le fragment assure, par un mécanisme traditionnel de portée, son accès à ce composant de la donnée «data».

II.1...

```
FRAGMENT Machine2;
 CONST Lo = N DIV 2 + 1; Hi = N;
 VAR FRAGT data : DataBaseType[[Lo .. Hi]];
 VAR i : INTEGER;
 MULTIPROC FRAGT Insert ( this : ARRAY OF InfoType );
 ...
 BEGIN                    (* initialisation du fragment *)
  FOR i := Lo TO Hi DO data[i].free := TRUE END
 END Machine2;
```

L'utilisation d'un objet de la classe «DataBase» ne requiert pas d'avoir connaissance de sa mise en œuvre répartie. La base de donnée ne constitue qu'un seul objet pour l'utilisateur. Cet exemple montre bien les idées sous-jacentes à l'introduction du mécanisme de fragmentation dans le langage POLYGOTH. Le mécanisme de réplication est obtenu en rendant tous les fragments de la classe identiques.

Cependant, l'exemple que nous venons de présenter n'est pas un programme POLYGOTH. Il lui manque la partie *virtuelle* que nous reproduisons ci-après.

II.1...

```
VIRTUAL
 VAR data : DataBaseType;
 MULTIPROC Insert ( this : ARRAY OF InfoType );
 MULTIPROC Querry ( question : QuerryType ) : ARRAY OF InfoType;
END;
```

La partie virtuelle, délimitée par les mots-clefs «VIRTUAL... END», permet de donner une vue globale de la classe. Les déclarations partielles sont regroupées dans cette partie : les fragments de structure de donnée retrouvent leurs formes familières de tableaux et de structures. La partie virtuelle a été initialement introduite pour améliorer la lisibilité des classes et pour simplifier la tâche du compilateur [11]. Par la suite, elle est devenue l'expression d'une mémoire partagée.

Nous pensons que cette mémoire partagée remet en cause l'utilité de l'expression explicite de la fragmentation. Pourquoi demander au programmeur d'expliciter la «localité» des données par rapport aux calculs alors que ces calculs ont, par définition, accès à l'ensemble des données, le modèle de programmation étant celui des processus communicant par mémoire partagée. Dans ce modèle, le placement des données par rapport aux processus ne concerne pas le programmeur. Néanmoins, certains langages permettent de décrire explicitement l'architecture

virtuelle (anneau, grille etc.) ciblée [12, 13, 14]. Il revient alors au programmeur d'exprimer le placement des données. Mais le langage POLYGOTH n'offre pas de telles notations concernant l'architecture. De plus, le mécanisme de mémoire virtuelle généralisée fourni par le système GOTHIC [15] rend transparent le placement des unités de stockage, les *segments*. Par voie de conséquence, le programmeur est libéré du problème posé par la répartition des données. L'intérêt d'un placement explicite des données ne nous semble donc pas évident dans le cas où celles-ci sont partagées.

II.2 Parallélisme

POLYGOTH présente deux grains de parallélisme : un grain fin avec les fragments et un grain moyen avec les objets. Bien qu'ils soient intimement liés, nous tenons à les distinguer pour mieux comprendre certaines difficultés posées par cette différence. Nous les appelons respectivement parallélisme *intra-objet* et *inter-objet*.

Parallélisme intra-objet

Le parallélisme intra-objet provient de la fragmentation. La création d'un objet fragmenté entraîne la création d'une famille de processus, un par fragment. Ces processus ont la même durée de vie que celle de l'objet. Ils sont *actifs* pendant l'exécution de la partie initiale de la classe, puis lors du traitement d'un appel de méthode. Entre un appel de méthode et le suivant, les processus sont *passifs*. Nous rappelons que les méthodes offertes par un objet fragmenté sont réalisées par des multiprocédures. L'appel d'une multiprocédure est effectué par les processus de l'objet fragmenté dans lesquels ont été déclaré ses composants. Il n'y a pas de création de nouveaux processus. Il s'en suit qu'un appel de méthode n'est exécuté par l'objet que si les processus concernés sont passifs. Les appels non exécutables sont mis en attente. Nous reproduisons de [3, page 79] cette politique de synchronisation :

> "La sérialisation entre les méthodes associées à un même objet est déduite de la répartition de ces méthodes entre les fragments. Deux méthodes définies sur deux ensembles de fragments non disjoints sont sérialisés."

Cette politique s'avère insuffisante pour garantir l'exclusion mutuelle dans l'accès aux variables privées d'un fragment (*cf.* [4]). Nous ne voyons pas de solution à

ce problème hormis l'introduction de primitives de synchronisation plus fine, c'est-à-dire, l'emploie de verrous [16] ou de régions critiques gardées [17].

Parallélisme inter-objet

Le parallélisme inter-objet est une vue macroscopique du parallélisme intra-objet. Les processus de différents objets peuvent être rendus actifs simultanément par l'appel simple de méthode. Chaque composant d'une multiprocédure peut appeler une méthode d'un objet de façon indépendante. Plusieurs familles de processus se trouvent ainsi activées simultanément. Ce parallélisme est le parallélisme inter-objet. La création d'un objet ajoute une famille de processus à l'univers d'exécution des programmes POLYGOTH. Nous pouvons remarquer que deux objets ne sont que partiellement parallèle car l'appel de méthode et la création d'objet se déroulent de manière synchrone, suivant le schéma de l'appel de procédure. Il est difficile de caractériser ce parallélisme puisque ses entités d'exécution sont des sous-familles de processus qui ne disposent pas de nommage. La synchronisation sur condition n'a pas été abordée dans le langage POLYGOTH. La granularité de ce parallélisme et la politique de sérialisation des méthodes réclament que le conditionnement de la synchronisation soit exprimé au niveau des méthodes.

Difficultés

Le mariage des mécanismes nécessaires, d'une part à l'exclusion mutuelle entre processus d'un même objet, et de l'autre, aux synchronisations conditionnelles entre processus de différents objets, ne nous semble pas aisé. Si, de plus, nous considérons les interférences entre le mécanisme de synchronisation et celui d'héritage, essentiel aux langages à objets [18], l'entreprise paraît inabordable.

II.3 Types

Dans POLYGOTH, les objets ne sont pas des valeurs comme les autres : ils ne peuvent pas être rangés dans les structures de données traditionnelles (tableaux et structures). Le mot-clef «OBJECT» permet de déclarer une variable à valeurs dans l'ensemble des exemplaires d'une classe. Le nom de la classe sert à typer ces variables. Deux systèmes de types disjoints cohabitent. Le premier, hérité directement de MODULA-2, permet de typer les valeurs et les variables habituelles. Le second se limite aux objets. Le module de définition –ou interface– associé à la classe d'un objet lui sert de type. Nous disons que les objets de la classe

satisfont son interface. Le mécanisme de spécialisation, caractéristique de la programmation à base d'objets, est absent dans POLYGOTH. Le système de types de POLYGOTH interdit toute utilisation d'un objet $\mathcal{O}$ satisfaisant une interface $\mathcal{I}$ là où est attendue un objet de type $\mathcal{I}'$, bien que $\mathcal{O}$ satisfasse l'interface $\mathcal{I}'$.

Récapitulatif

En résumé, le langage POLYGOTH souffre de certaines contradictions dans sa conception :

- le mécanisme de fragmentation, élégant en l'absence de mémoire partagée, s'avère contrecarré par le modèle de communication retenu,
- le parallélisme, difficile à gérer de façon uniforme, paraît impossible à maîtriser en présence de l'héritage,
- le système de types reste trop pauvre pour tirer partie, à bon escient, de l'approche objet;

c'est pourquoi nous avons entrepris la conception de ARCHE.

Le langage Arche

Nous retenons un modèle d'objet dans la lignée de l'école scandinave en insistant d'avantage sur l'aspect masquage des informations. Nous sommes ainsi en mesure d'introduire, de façon intégrée, les différents mécanismes nécessaires à l'approche objet de la programmation et à la gestion explicite du parallélisme. Parmi les points saillants de ARCHE, remarquons les originalités suivantes :

- intégration du parallélisme, de l'héritage et des types abstraits dans un langage impératif fortement typé,
- généralisation de l'invocation de méthode à une séquence d'objets,
- introduction d'un mécanisme de traitement d'exceptions adapté aux contextes parallèles imbriqués.

Nous avons organisé la présentation de cette partie comme suit. Le chapitre III traite des objets, des types et des classes de ARCHE. Les types primitifs (ou concrets) sont tout d'abords rapidement présentés, puis les types abstraits. Les classes sont introduites et quelques exemples sont utilisés pour mettre en évidence la relation entre le type abstrait et la classe qui le réalise. Nous nous attardons sur le parallélisme et les mécanismes de communication et de synchronisation fournis par ARCHE dans le chapitre IV. Le chapitre V introduit deux relations d'ordre liées au mécanisme d'héritage. La première, définie pour les types, permet l'extension d'une interface par spécialisation. La deuxième, portant sur les classes, sert de support à la programmation par différences. Le chapitre VI présente le modèle pour le traitement des exceptions retenu, ainsi que son expression dans ARCHE. En conclusion, chapitre VII, nous décrivons l'état d'avancement de la mise en œuvre de ARCHE et nous indiquons quelques perspectives.

Chapitre III

Objets fortement typés

III.1 Présentation

Nous présentons trois notions essentielles dans ARCHE : l'objet, la classe et la vue. Dans un souci de clarté, nous définissons ces notions en employant la terminologie associée aux langages procéduraux classiques. Nous utilisons principalement les notions de variable, de type, de procédure et de module. Dans un premier temps, les aspects liés au parallélisme, à l'héritage et au traitement d'exceptions de ARCHE ne sont pas abordés. Les trois chapitres suivants leur sont consacrés.

III.1.1 L'objet

Un objet est un exemplaire de module. Les procédures exportées par ce module sont les méthodes de l'objet. Elles correspondent aux différentes opérations susceptibles d'être appliquées à son état. L'état d'un objet est l'ensemble des valeurs des variables de premier niveau du module. Un objet n'est pas directement dénotable dans le langage ARCHE. Un objet n'est présent dans le langage que sous la forme d'une *référence*. Une référence correspond à une forme de nommage de l'objet transparente au programmeur. Une référence est une valeur à part entière : elle peut être rangée dans une structure et elle peut être passée en paramètre à d'autres objets. Par abus de langage, les termes objet et référence d'un objet sont employés indifféremment. ARCHE est un langage hybride en ce sens que toutes les valeurs ne sont pas des objets. Les valeurs de ARCHE sont les valeurs habituelles (les booléens, les entiers, les caractères, etc.) et les objets. Toutes les variables et les valeurs de ARCHE sont typées. Les variables le sont de façon explicite par le programmeur. En particulier, les variables contenant des objets sont typés par une famille de types, les *vues*, issus d'un même

constructeur.

III.1.2 La vue

La vue est l'entité ARCHE qui donne accès aux objets. Une vue est un type voisin d'une structure dont les champs sont des procédures. En tant que type d'un objet, une vue indique les méthodes fournies par celui-ci. En tant que type d'une variable, une vue détermine les méthodes que doivent offrir les objets susceptibles d'être affectés à celle-ci. ARCHE fait partie des langages qui peuvent toujours garantir, à la compilation, que l'objet concerné par un appel fournit la méthode appelée.

III.1.3 La classe

La classe est l'entité ARCHE qui permet de déclarer les familles d'objets. Une classe est un module dont seules certaines procédures sont exportées. La famille d'objets déclarés par une classe est formée par l'ensemble de ses exemplaires. Une classe est toujours associée à une vue. On dit alors qu'elle *réalise* cette vue. Il revient au programmeur d'expliciter la vue réalisée par la classe qu'il écrit. Ce choix détermine les procédures exportées par la classe ainsi que le type des objets exemplaires de celle-ci.

III.1.4 Les unités de compilation

Les unités de compilation de ARCHE sont la classe et la *bibliothèque*. Une bibliothèque est un module qui regroupe un ensemble de déclarations de type. Un module bibliothèque peut utiliser les déclarations faites dans d'autres bibliothèques ARCHE. Ces modules, délimités par les mots-clefs «LIBRARY»... «END», servent de support à l'organisation des types. A titre d'exemple, une bibliothèque peut regrouper tous les types relatifs aux pièces mécaniques d'un appareil alors que ceux qui ont attrait aux circuits pneumatiques sont décrits dans un autre module :

Exemple III.1

```
LIBRARY Mecanique =                LIBRARY Pneumatique =
 Poulie = ...                       Valve = ...
 ...                                ...
 Force = ...                        Force = ...
 ...                                ...
END Mecanique;                     END Pneumatique;
```

Afin d'éviter des conflits dans le nommage des types, le mécanisme d'utilisation d'une bibliothèque offre trois modes d'opération : le nommage préfixé, le nommage direct et le renommage.

III.1...

```
        LIBRARY Appareil =
(1)      USES Mecanique;
(2)      FROM Mecanique USES Force;
(3)      FROM Pneumatique USES Force AS Pression;
         ...Mecanique.Poulie...
         ...Force...                (* Mecanique.Force *)
         ...Pression...             (* Pneumatique.Force *)
        END Appareil;
```

Ces modes sont illustrés dans l'exemple III.1 par les clauses d'utilisation, numérotées *(1)*, *(2)* et *(3)*, du module «`Appareil`»

Une déclaration de type associe un nom à une expression de type. Nous en faisons une brève présentation dans les paragraphes suivants.

III.2 Types concrets

Nous retrouvons dans ARCHE la plupart des types présents dans MODULA-2. Nous les présentons très rapidement au moyen de quelques exemples.

III.2.1 Types ordinaux

Une *énumération* est définie par une suite d'identificateurs. Par exemple :

```
TYPE
 Sexe   = ( masculin, feminin);
 Fruits = ( banane, orange, pamplemousse, pomme );
```

Un *intervalle* est défini par ses bornes inférieure et supérieure. Elles doivent appartenir au même type ordinal. Par exemple :

```
TYPE
 Dizaine         = [ 1 .. 10 ];
 FruitsAPepins = Fruits [ orange .. pomme ];
```

Les fonctions accompagnant les types ordinaux sont :

- «ORD» rend l'ordre de l'ordinal dans son type,
- «VAL» rend la valeur qui a pour ordre son premier argument dans le type indiqué par son second argument, ou signale l'exception ERRTYPE (*cf.* § VI),
- «LOWER» rend la première valeur du type ordinal donné en argument,
- «UPPER» rend la dernière valeur du type ordinal donné en argument,
- «SIZE» rend le nombre d'éléments du type ordinal donné en argument.

Le type «INTEGER» est le type contenant tous les entiers représentés par la machine. Les autres types ordinaux prédéfinis peuvent s'exprimer à l'aide de celui-ci comme suit :

```
CARDINAL  = [ 0 .. UPPER( INTEGER ) ];
CARDINAL1 = [ 1 .. UPPER( INTEGER ) ];
BOOLEAN   = ( FALSE, TRUE );
CHAR      = ( <ascii>  + <special> );
```

Enfin, nous disposons des valeurs à virgule flottante double précision («FLOAT»).

Les opérations initialement fournies pour ces types concrets sont décrites dans le paragraphe consacré à la présentation des expressions III.4.4, page 41. Cette liste n'est ni exhaustive, ni définitive ; l'utilisation du langage et ses futures révisions pourraient la modifier.

III.2.2 Ensembles

Il est possible de définir des ensembles d'éléments de type ordinal au moyen du constructeur «SET OF». Voici un exemple d'utilisation des ensembles. Soit la déclaration :

```
TYPE
 Corbeille = SET OF FruitsAPepins;
```

alors, une variable de type «Corbeille» peut prendre les valeurs :

```
{ }, {orange},{pamplemousse},{pomme},
{orange, pomme},{orange, pamplemousse}, {pamplemousse, pomme},
{orange, pamplemousse, pomme}
```

III.2.3 Structures

Le constructeur «RECORD» permet de regrouper plusieurs éléments de types éventuellement différents pour former une *structure*. Un composant de structure est appelé un *champ* et il est déclaré à l'aide d'un identificateur suivi du type associé. Deux champs ne peuvent pas avoir le même nom. Les valeurs de la structure sont donc des associations *nom–valeur* dans lesquelles la valeur appartient au type du champ nommé. Par exemple, une date de naissance peut être décrite à l'aide de la déclaration suivante :

```
TYPE
 DateNaissance = RECORD
                  annee : [0 .. 99];
                  mois  : Mois
                 END;
```

Ce constructeur permet aussi d'étendre la définition d'une structure par préfixage. La présentation de ce mécanisme est reportée dans le paragraphe qui traite des sous-types (*cf.* § V). A titre d'exemple, la description d'un numéro INSEE peut être déclaré comme suit :

```
TYPE
 Insee          = DateNaissance
                  RECORD
                   sex   : Sexe;
                   dpt   : Departement;
                   clef  : [0 .. 99]
                  END;
```

Une variable de type «Insee» inclut les champs déclarés par le type «DateNaissance». Les champs d'une structure sont accédés au moyen de la notation pointée :

```
MonNumero: Insee;
...
MonNumero.clef := ...; ... MonNumero.mois;
...
```

III.3 Type vue

Le type vue correspond à l'intégration de la notion d'interface de classe dans un système de types. Nous avons adopté une position extrême dans le degré de protection des données d'un objet. L'accès à son état n'est possible que par appel de méthode.

Prenons le traditionnel exemple du point. Dans un espace à deux dimensions, un point peut présenter l'interface suivante :

Exemple III.2

```
Point    = VIEW ( x, y: FLOAT )
             GetPos: ()(x,y: FLOAT);
             Move  : ( dx,dy: FLOAT)();
             Rotate: ( alpha: FLOAT)();
           END;
```

La vue «`Point`» est paramétrique en ses coordonnées initiales, et offre trois méthodes : «`GetPos`» qui ne prend pas de paramètre et rend la position courante du point en coordonnées cartésiennes ; «`Move`» qui prend en paramètre le vecteur de translation et transporte le point par effet de bord ; et «`Rotate`» qui prend l'angle de rotation en paramètre et effectue cette rotation par effet de bord.

Nous pensons qu'il est important de séparer la description de l'état d'un objet de la déclaration des signatures de ses méthodes. Cette scission rend indépendante l'utilisation d'un objet, ne serait-ce que syntaxiquement, de sa mise en œuvre effective. Le type vue est à rapprocher d'un type abstrait de donnée. Il faut toutefois remarquer que ce rapprochement ne concerne que l'aspect syntaxique. Un type vue ne précise que les signatures des opérations fournies par les objets qu'il décrit. Bien qu'il existe des techniques pour décrire la sémantique de ces opérations de façon plus ou moins indépendante de la représentation concrète de l'état de l'objet [19], l'utilisation de cette description dans un langage de programmation reste encore problématique. Si la description sémantique des méthodes d'un objet faisait partie du type vue, il faudrait que le compilateur ARCHE vérifie que les procédures de la classe réalisant ce type possèdent la sémantique précisée par la vue. Nous ne connaissons pas d'algorithme qui permette d'effectuer cette vérification. Les descriptions sémantiques ne seraient pas plus profitables aux programmeurs que de simple commentaires. Loin de décourager l'écriture de ces descriptions, nous pensons que les langages de spécification formelle, tels que LARCH [20], VDM [21], Z [22] ou MAUDE [23] entre autres, sont mieux adaptés à cette tâche. La formalisation des liens existant entre les langages de spécification et ceux de programmation reste du domaine de la recherche [24, 25, 26].

III.3.1 Signature de méthode

La signature d'une méthode ARCHE est déclarée en indiquant le nom de la méthode suivi de deux noms de type. Le premier type correspond à la déclaration d'une structure dont les champs sont les paramètres formels. Similairement, les résultats de la méthode sont les champs de la structure nommée par le deuxième. Cette façon de déclarer la signature d'une méthode permet d'accéder, en bloc, ses paramètres ou ses résultats. Elle présente néanmoins l'inconvénient de forcer le programmeur à déclarer explicitement les types pour chaque signature. Pour pallier cette lourdeur d'expression, nous introduisons une convention d'écriture qui rend implicite la déclaration des deux structures.

Exemple III.3

La déclaration suivante :

```
meth: (e1: TE1,...,en: TEn)(s1: TS1,...,sr: TSr);
```

est une abréviation des déclarations :

```
methIN  = RECORD e1: TE1; ...; en: TEn END;
methOUT = RECORD s1: TS1; ...; sr: TSr END;
meth: ( methIN )( methOUT );
```

Cette convention permet d'offrir au programmeur plus de souplesse dans l'utilisation des appels de méthode. Une liste de valeurs correctement typées, chacune correspondant à un champ, est acceptée en remplacement d'une structure. Cette souplesse rend aisée la composition de deux appels de méthode dont les résultats du premier appel sont les paramètres du second. Nous illustrons nos propos par un exemple :

Exemple III.4

```
meth : (e1: TE1,...,en: TEn)(s1: TS1,...,sr: TSr);
meth': (f1: TS1,...,fr: TSr)();
...
VAR result : RECORD q1: TS1; ...; qr: TSr END;
    other  : RECORD p1: TE1; ...; pn: TEn END;
...
(1) result := meth(exp1, ..., expn);
(2) meth'(meth(exp1, ..., expn));
(3) result := meth(other);
...
```

L'expression à la ligne *(1)* (resp. *(2)*, *(3)*) est une abréviation des lignes *(1')* (resp. *(2')*, *(3')*) :

III.4...

```
VAR actual : methIN;
    temp   : methOUT;
    actual': meth'IN;
...
(1')  actual.e1, ..., actual.en := exp1, ..., expn;
(1')  temp := meth(actual);
(1')  result.q1, ..., result.qr := temp.s1, ..., temp.sr;
...
(2')  actual.e1, ..., actual.en := exp1, ..., expn;
(2')  temp := meth(actual);
(2')  actual'.f1, ..., actual'.fr := temp.s1, ..., temp.sr;
(2')  meth'( actual' );
...
(3')  actual.e1, ..., actual.en := other.p1, ..., other.pn;
(3')  temp := meth(actual);
(3')  result.q1, ..., result.qr := temp.s1, ..., temp.sr;
```

La convention, duale de celle-ci, portant sur le mécanisme de retour de procédure, et sur l'accès à ses paramètres, est exposée au paragraphe III.4.1.4, lors de la description des procédures.

III.3.2 Vues remarquables

Il existe dans ARCHE plusieurs vues prédéfinies qui ont un status spécial : «ANY», «NULL», «CURRENT» et «SEQUENCE».

III.3.2.1 Extrêmes

Le type «ANY» est la vue la plus pauvre en ce sens qu'elle ne précise aucune méthode. Le type «NULL» correspond à la vue universelle. Il permet de typer l'objet «NIL». Cet objet sert de valeur initiale à toute variable typée par une vue. Tout appel des méthodes de «NIL» rend l'exception prédéfinie «FAILURE».

III.3.2.2 Réflexive

La vue «CURRENT» tient lieu d'auto-référence. Elle signifie le type courant. Son usage est fortement limité afin de conserver la sûreté du système de types. Dans la signature d'une méthode ou d'une procédure, la vue «CURRENT» ne peut

être utilisée que pour typer un résultat. Nous respectons ainsi la monotonie des méthodes par rapport à la spécialisation des types [27].

La vue «CURRENT» ne va pas sans rappeler un cas particulier de la déclaration de type par association introduite dans EIFFEL [28]. Il s'agit de la déclaration servant à désigner le type de la classe courrante, «like *Current*», qui reste valide pour toutes les possibles extensions de la classe par héritage. En effet, tout comme cette déclaration novatrice [29], la vue «CURRENT» sert essentiellement à typer la valeur associée à l'expression «SELF» (cf. § III.4.4). L'information fournie par ce type s'avère nécessaire, dans un langage fortement typé, pour améliorer l'exploitation d'appel de méthode ayant cette expression pour résultat [30, 31]. Bien que nous n'ayons pas encore présenté le mécanisme de spécialisation de type ni celui d'héritage, nous nous servons de l'exemple classique du pixel et de sa version colorée –certes un peu artificiel (cf. [31])– pour illustrer l'utilisation de «CURRENT». Soient les vues :

Exemple III.5

```
Pixel       = VIEW ( x, y: CARDINAL )
                Move  : ( dx,dy: INTEGER )( new: CURRENT );
                ...
              END;

PixelCouleur = VIEW Pixel
                ChangeColor : ( c: Color )();
                ...
              END;
```

La vue «PixelCouleur» *spécialise* la vue «Pixel» en fournissant des méthodes supplémentaires. «Pixel» est un préfixe de «PixelCouleur» et les méthodes de la première sont aussi offertent par la dernière. La vue «CURRENT» permet d'indiquer que la méthode «Move» dans la vue «PixelCouleur» rend un objet de type «PixelCouleur». Sans cette indication, la construction par préfixation nous aurait contraint à *perdre* la couleur des pixels déplacés par la méthode «Move».

Nous devons remarquer que la procédure associée à la méthode «Move» dans la classe réalisant le type «Pixel» peut être directement réutilisée, via l'héritage, dans la classe réalisant le sous-type «PixelCouleur». Soient «UnPixel» et «UnPixelCouleur» les classes associées à ces types. La procédure «UnPixel.Move» peut être le corps de la méthode «PixelCouleur.Move» car le type de son résultat est «CURRENT». Ce nonobstant, une variable de type «CURRENT» ne peut référencer que des exemplaires de classe créés de façon auto-référentielle, c'est-à-dire au moyen de l'expression «NEW CURRENT» (cf. § IV.1, page 45). La procédure «UnPixel.Move» s'écrit comme suit :

III.5...

```
CLASS UnPixel IMPLEMENTS Exemple.Pixel =
 VAR X: CARDINAL := x; Y: CARDINAL := y;
...
PROCEDURE Move = BEGIN
  RETURN (NEW CURRENT(X+dx,Y+dy));
 END Move;
...
END UnPixel;
```

La procédure «UnPixelCouleur.Move» peut s'écrire, grâce à l'héritage, comme suit :

III.5...

```
CLASS UnPixelCouleur IMPLEMENTS Exemple.PixelCouleur =
 INHERITS UnPixel;
 REDEFINES Move;
VAR
 MaCouleur: Color;
...
PROCEDURE Move =
  VAR r: PixelCouleur;
 BEGIN
  r :=  SUPER(dx,dy); r!ChangeColor(MaCouleur);
  RETURN (r);
 END Move;
...
END UnPixelCouleur;
```

La clause «REDEFINES» permet d'indiquer qu'une nouvelle procédure est fournie pour réaliser la méthode «Move». L'expression «SUPER» dénote l'appel à la procédure «Move» de la classe incluse par héritage. En l'occurence, cet appel rend un exemplaire de la classe courrante bien que la procédure appelée soit «UnPixel.Move». Nous n'expliquons pas davantage cet exemple. Une compréhension intuitive de la vue «CURRENT» et des expressions «NEW CURRENT» et «SUPER» suffit à cette introduction. Ces notions sont traitées plus avant dans le chapitre consacré à l'héritage et aux sous-types. Nous abordons à présent la dernière des vues remarquables.

III.3.2.3 Séquences

Le lecteur aura remarqué l'absence des tableaux, «ARRAY OF», dans la présentation des types concrets au paragraphe III.2. Nous n'avons pas inclu ce constructeur de type pour des raisons historiques.

Dans un premier temps, nous avons retenu une abstraction du tableau. Il s'agit de la séquence. Une séquence est une fonction finie des entiers vers un ensemble donné, nommé son *type de base*, dont le domaine est un intervalle $[1\,..\,k]$ pour un entier k. Grossièrement, nous pouvons dire qu'une séquence est un tableau dynamique «sans trous». Un index n'appartient au domaine d'une séquence que s'il est associé à une valeur du type de base.

Le constructeur «SEQ OF» s'applique à tous les types de ARCHE. Plusieurs opérations lui sont associées parmi lesquelles figurent la concaténation, le retrait de tête ou de queue, ou encore la surécriture. Les séquences dont l'index est borné explicitement, au moyen d'un type intervalle, sont distinguées des séquences *ouvertes* dont la borne supérieure est implicite. Par exemple, soient «Kg» et «Chapitre» deux types quelconques dans la déclaration suivante :

```
TYPE
 Recolte = SEQ Fruits OF Kg;
 Livre   = SEQ OF Chapitre;
```

alors, une variable de type «Recolte» est une séquence dont l'index ne peut prendre que les valeurs du type «Fruits». En revanche, une variable de type «Livre» est une séquence dont l'index varie sur le type «CARDINAL1».

La notation d'accès aux éléments d'une séquence reste la paire de crochets. Soit une variable, «R», de type «Recolte», alors la récolte d'orange est donnée par l'expression «R[orange]». Dans le cas des séquences à multiple dimensions, les règles classiques de simplification d'écriture des tableaux restent valides :

```
SEQ I1,...,In OF T;  s[i1,...,in]
```

sont des abréviations pour

```
SEQ I1 OF ... OF SEQ In OF T; s[i1]...[in]
```

Nous avons présenté le type séquence comme un type prédéfini. Cependant, le type séquence correspond à une pseudo-vue réalisée par deux classes prédéfinies ; la première réalise les séquences ouvertes alors que la deuxième correspond aux séquences fermées. Aussi, nous avons adopté la syntaxe d'appel de méthode

pour les opérations[1] associées au type séquence afin de souligner que la séquence pourrait être définie par la vue suivante :

La pseudo-vue «SEQUENCE»[T]

```
VIEW
 CAT      : (s: SEQ OF T)();
 APPEND   : (e: T)();
 override : (i: CARDINAL1, e: T)();
 TAIL     : ()();
 FRONT    : ()();
 FILTER   : (s: SEQ OF T)();
 DOMRES   : (i: SEQ OF CARDINAL1)();
 LENGTH   : ()(l: CARDINAL);
 HEAD     : ()(e: T);
 LAST     : ()(e: T);
 index    : (i: CARDINAL1)(e: T);
END;
```

Une variable de type séquence dénote donc toujours une *référence* à un objet de l'une des deux classes prédéfinies. La création d'exemplaire est gérée de façon implicite dans le langage.

Nous appelons les opérations associées aux séquences des *pseudo-méthodes*. Nous en indiquons leur sens ci-après.

«CAT» prend en argument une séquence du même type que l'objet appelé et en fait son préfixe.

«APPEND» prend en argument un élément *affectable*[2] au type de base de l'objet et se l'ajoute à l'index successeur de sa taille.

«override» prend en arguments un index et un élément affectable au type de base de l'objet. Deux cas sont à considérer. Si l'index n'appartient pas au domaine de la séquence, c'est-à-dire qu'aucun élément lui est associé, alors deux sous-cas se présentent. Si l'index est le successeur de la taille de la séquence, alors l'élément est ajouté à cette position dans la séquence. Dans le cas contraire, l'exception d'erreur **«FAILURE»** est signalée à l'appelant (*cf.* § VI.2.1.4, page 93). Si l'index appartient au domaine de la séquence, l'élément associé à l'index est remplacé par le second argument.

[1] Les méthodes «index» et «override» constituent une exception à cette règle puisque nous utilisons les crochets et l'affectation, plus pratiques.

[2] La relation d'affectabilité est définie au paragraphe V.1.3, page 78.

«TAIL» l'objet appelé devient une séquence, identique à l'originale, à laquelle on aurait retiré le premier élément.

«FRONT» l'objet appelé devient une séquence, identique à l'originale, à laquelle on aurait retiré le dernier élément.

«FILTER» l'objet appelé devient une séquence, identique à l'originale, de laquelle on aurait uniquement conservé, dans le même ordre, tous les éléments appartenant à la séquence fournie en argument.

«DOMRES» l'objet appelé devient une séquence, identique à l'originale, de laquelle on aurait uniquement conservé, dans le même ordre, les éléments indexés par les éléments de la séquence fournie en argument.

«LENGTH» rend la taille de la séquence, c'est-à-dire le nombre d'éléments indexés.

«HEAD» rend le premier élément ou l'exception «FAILURE».

«LAST» rend le dernier élément ou l'exception «FAILURE».

«index» rend l'élément indexé par son argument ou l'exception «FAILURE».

Nous ne définissons pas les séquences au moyen d'une vue générique car nous avons choisi de ne pas introduire, dans un premier temps, la généricité dans le langage ARCHE. Par voie de conséquence, les séquences ne sont pas traitées de façon homogène par rapport aux autres types du langage. Elles sont un hybride de type abstrait et de type concret. Les constructeurs «SEQ OF» et «SEQ I OF» correspondent à des pseudo-classes qui ne peuvent pas figurer dans un graphe d'héritage, à différence des langages tels que POOL ou EIFFEL [32]. Néanmoins, elles bénéficient d'un mécanisme, proche de l'héritage, permettant de les étendre dans la dimension *temporelle* plutôt que dans celle *spatiale*. Nous présentons ce mécanisme dans le paragraphe IV.5, page 60.

III.4 Classe

Une vue peut être réalisée par plusieurs classes. Ainsi, deux objets de même type n'ont pas forcément la même représentation concrète. Cette flexibilité constitue une première source de polymorphisme dans ARCHE.

La description des objets est fournie dans l'entité ARCHE correspondant au module d'implantation de MODULA-2. Il s'agit de la *classe*. Une classe, délimitée par les mots-clefs «CLASS ...END», introduit la représentation concrète des objets et le corps des méthodes offertes. Nous disons qu'une classe *réalise* une vue. Le nom de cette vue figure explicitement dans l'en-tête de la déclaration de classe. Il

doit être préfixé du nom de la bibliothèque, «LIBRARY», qui le définit (*cf.* § III.3). Une classe a un nom qui sera utilisé pour en créer des exemplaires.

Une déclaration de classe comporte une liste de déclarations –types, constantes, variables et procédures– et un corps. Les variables déclarées constituent la représentation concrète de l'état des objets de la classe réalisée ; les procédures sont associées au traitement de ses méthodes ; le corps permet de décrire les actions à effectuer lors de la création et de la recopie de ses exemplaires.

Nous présentons tout d'abord un exemple, puis nous décrivons de manière plus détaillée les entités constituantes d'une classe.

L'interface exprimée par la vue «Point» de l'exemple III.2 (*cf.* page 26) ne contraint pas la représentation du point. Nous présentons deux classes *réalisant* ce type. La première montre une représentation cartésienne de l'état[3] :

Exemple III.6

```
CLASS CartesianPoint IMPLEMENTS Examples.Point =
 VAR X: FLOAT := x;  Y :FLOAT := y;
 PROCEDURE GetPos = BEGIN  RETURN ( X,Y ) END GetPos;
 PROCEDURE Move   = BEGIN  X := X+dx; Y := Y+dy END Move;
 PROCEDURE Rotate =
  VAR c,s : FLOAT;
 BEGIN
  c := COS(alpha); s:= SIN(alpha);
  X := X*c - Y*s; Y := X*s + Y*c
 END Rotate;
END CartesianPoint;
```

La deuxième représente un point par ses coordonnées polaires :

[3]Nous supposons dans cet exemple que nous disposons des fonctions trigonométriques et de certaines opérations sur les valeurs à virgule flottante. Un environnement de programmation ARCHE devrait fournir des objets chargés d'assurer ces fonctionnalités. Ces objets pourraient être réalisés directement par le compilateur de l'environnement.

III.6...

```
CLASS  PolarPoint IMPLEMENTS Examples.Point =
 VAR r   : FLOAT := SQRT(x*x + y*y);
     rho : FLOAT := ARCTG(y/x);
 PROCEDURE GetPos =
 BEGIN
  RETURN ( r*COS(rho), r*SIN(rho) )
 END GetPos;
 PROCEDURE Move =
  VAR p,q: FLOAT;
 BEGIN
  p := r*COS(rho)+dx; q := r*SIN(rho)+dy;
  r := SQRT(p*p + q*q); rho := ARCTG(q/p)
 END Move;
 PROCEDURE Rotate = BEGIN  rho := rho + alpha END Rotate;
END PolarPoint;
```

Nous pouvons remarquer que le lien entre le type et la classe est explicite. Le compilateur est chargé de vérifier que la classe réalise effectivement le type indiqué. Une classe réalise une vue si chacune des méthodes offertes par la première trouve une procédure de même nom et de même signature dans la dernière. On dit alors que la procédure est associée à la méthode ou que la procédure est le corps de la méthode. Une classe peut déclarer plus de procédures que la vue réalisée n'offre de méthodes. Ces procédures sont dites *locales* à la classe. Deux procédures ne peuvent pas avoir le même nom. En conséquence, la signature d'une procédure associée à une méthode ne doit pas obligatoirement figurer dans sa déclaration.

Les paramètres «x» et «y», communs à la vue réalisée et à la classe, ne font pas partie de l'état des objets ; leur portée est limitée au bloc de premier niveau. Malgré sa simplicité, cet exemple souligne bien la dépendance entre la représentation concrète de l'état d'un objet et l'écriture du corps de ses méthodes.

Afin d'illustrer l'utilisation de ces points, nous présentons une classe réalisant la vue «ANY», ce qui correspond à un programme principal dans les langages classiques.

III.6...

```
CLASS Main IMPLEMENTS STD.ANY =
 USES PolarPoint, CartesianPoint;
 FROM Examples USES Point;
 VAR UpperLeft, LowerRight: Point;
BEGIN
 UpperLeft := NEW PolarPoint(-1.0,2.0);
 LowerRight := NEW CartesianPoint(3.0,-2.0);
 UpperLeft!Move(LowerRight!GetPos());
 ...
END Main;
```

L'identificateur «STD» fait allusion à la bibliothèque standard des types primitifs de ARCHE parmi lesquels figure la vue «ANY». Les déclarations explicites d'utilisation de types ou de classes ne va pas sans rappeler le mécanisme d'importations de MODULA-2. Cependant, il serait erroné de leur attribuer la même signification. Les clauses d'utilisations de ARCHE sont des commentaires, à caractère obligatoire, décrivant les dépendances que le programmeur établit entre ses différentes unités de compilation. Ces simples commentaires permettent une meilleure organisation des composants d'un logiciel et facilitent l'automatisation de leur traitement (la mise à jour de versions par exemple). Le fait d'utiliser «PolarPoint» n'étend pas la portée de ses déclarations à la classe «Main». La déclaration rend explicite son utilisation. La forme d'utilisation sélective, «FROM...USES», est réservée aux bilbiothèques de types («LIBRARY»).

La classe «Main» déclare deux variables du même type «Point». Lors de la création de ces objets, le choix de représentation est explicitement fait par le programmeur. La primitive «NEW» permet de créer un objet à partir du nom de sa classe et des valeurs fournies en paramètre, ici la position initiale du point. L'appel de méthode est dénoté par le point d'exclamation, « ! », placé entre l'objet appelé et le nom de la méthode concernée. La composition des appels est simplifiée par la signification donnée aux signatures de méthodes au paragraphe III.3.

III.4.1 Déclarations

Les déclarations d'une classe sont structurées en quatre paragraphes, tous facultatifs. Chacun d'eux est précédé par chacun des mots-clefs suivant : «CONST», «TYPE», «VAR» et «PROCEDURE» Ces mots-clefs conservent leur sens habituel. La portée des déclarations est limitée à la classe dans laquelle elles apparaissent textuellement.

III.4.1.1 Constantes

Les constantes peuvent être typées explicitement. ARCHE présente deux notions de constante : la *constante de classe* et la *constante d'objet*. La valeur d'une constante de classe est définie au moyen d'une expression calculable à la compilation. Ces expressions sont formées à partir des littérales et des autres constantes de classe au moyen de quelques opérations primitives. La valeur d'une constante d'objet est définie par une expression calculable à la création d'un exemplaire de classe. Cette expression ne peut être constituée que de constantes de classe et de paramètres de classe. Les classes mettant en œuvre la vue «`Point`» (*cf.* page 34) illustrent l'utilisation des constantes d'objet dans l'expression de leurs états initiaux.

III.4.1.2 Types

De nouveaux types peuvent être déclarés dans les modules «`CLASS`». Un type est déclaré en associant une expression de type à un identificateur. Plusieurs exemples de déclaration de type ont été donnés au paragraphe III.2. Des vues peuvent être déclarées dans une classe. Toutefois, il n'est pas possible de déclarer de classes locales dans cette version du langage. Nous renvoyons à [33] le lecteur intéressé par les classes imbriquées. Nous avons tâché de ne pas introduire des mécanismes qui complexifient l'intégration, d'emblée ardue, du parallélisme et des objets polymorphes dans le monde des langages fortement typés. Un type déclaré dans un module «`LIBRARY`» peut être utilisé dans une classe grâce à la clause «`USES`» d'utilisation que nous avons présenté au paragraphe III.1.4.

III.4.1.3 Variables

Les variables sont explicitement typées. Chaque variable se voit affectée d'une valeur par défaut à la création de l'objet. Le type d'une variable détermine sa valeur par défaut. La table suivante indique cette correspondance :

Expression associée par défaut à une variable suivant son type	
Type	Expression
Ordinal	«`VAL(0)`»
Flottant	«`0.0`»
Ensemble	«`{ }`»
Séquence	«`< >`»
Structure	«`(`» défaut de chacun de ses champs «`)`»
Vue	«`NIL`»

Le programmeur peut remplacer ces valeurs par défaut soit à la déclaration par l'affectation d'une expression constante –de classe ou d'objet–, soit dans le corps de la classe par l'affectation d'une expression quelconque.

III.4.1.4 Procédures

Lorsqu'une procédure fournit le corps d'une méthode de la vue associée, l'entête n'a pas à être répétée. Les paramètres formels sont des variables locales de la procédure. Lorsque les paramètres ou les résultats sont déclarés de façon anonyme –c'est-à-dire par un type structure–, les champs de ces structure sont implicitement assimilés à des paramètres formels, comme nous l'avons déjà expliqué au paragraphe § III.3.1. Soient les déclarations suivantes :

Exemple III.7

```
data = RECORD e1: TE1; ...; en: TEn END;
...
meth: ( data )();
```

alors, dans le corps de la procédure correspondant à la méthode «meth», les déclarations suivantes sont implicitement faites :

III.7...

```
PROCEDURE meth =
 VAR e1: TE1; ...; en: TEn;
...
END meth;
```

Le passage de paramètre se fait toujours par valeur. Nous pouvons remarquer que le partage de référence, traditionnellement lié au passage de paramètre par référence, est inhérent à la définition des objets. En effet, la valeur d'un objet est une référence et son passage en paramètre, par valeur, est une forme déguisée de passage par référence de l'exemplaire de classe lui-même. Le partage de référence est présent jusque dans l'affectation. En effet, l'affectation d'un objet à une variable a précisément ce sens. En particulier, le passage d'un paramètre de type séquence entraîne le partage, par référence, de ses éléments. Il en est de même pour l'affectation d'une séquence : les éléments ne sont pas recopiés.

Les procédures peuvent fournir des déclarations locales de types, constantes, variables ou procédures. La portée, la visibilité et la durée de ces déclarations sont celles adoptées par MODULA-2. Une liste de commandes parenthésée par les mots-clefs «BEGIN...END» constitue le corps d'une procédure et son exécution se déroule de la même manière qu'en MODULA-2. Le retour de procédures

rendant un résultat est toutefois exprimé différemment. La commande «RETURN» suivie d'une expression indique la fin de la procédure ainsi que son résultat, nécessairement de type structure. Un sucre syntaxique, dual de celui présenté au paragraphe III.3.1, permet de fournir la valeur de chaque champ de la structure résultat au moyen d'une simple liste d'expressions. Nous illustrons cette facilité d'écriture par l'exemple suivant. Soit une méthode dont la signature est :

Exemple III.8

```
result = RECORD s1: TS1; ...; sr: TSr END;
meth : ()( result );
```

alors, dans le corps de la procédure correspondante les commandes de retour suivantes sont admises et revêtent le même sens :

III.8...

```
VAR local: RECORD t1: TS1; ...; tr: TSr END;
...
local.t1, ..., local.tr := exp1, ..., expr;
RETURN local;
...
RETURN (exp1, ..., expr);
...
```

Nous pouvons à présent décrire le corps d'une classe, et de ce fait, les commandes et les expressions du langage ARCHE. Tout comme nous l'avons fait dans la présentation générale et dans la partie concernant les déclarations, nous omettons, dans un premier temps, les aspects liés au parallélisme, à l'héritage et au traitement des exceptions qui sont respectivement traités aux chapitres IV, V et VI.

III.4.2 Création et recopie

Le corps d'une classe comporte deux paragraphes formés d'une liste de commandes : celui correspondant à la création est précédé du mot-clef «NEW», et celui de la recopie par le mot-clef «COPY». Tous deux sont facultatifs.

Lors de la création d'exemplaire, les commandes du paragraphe «NEW» sont exécutées de façon séquentielle. Ce bloc de commandes sert essentiellement à créer d'autres objets et à réaliser une suite d'opérations sur ces derniers pour établir un état initial correct de l'objet créé. Les paramètres de classe peuvent être référencés dans ce bloc. Il correspond aussi à la notion de programme en

PASCAL ou C (cf. encadré page 36). Dans le cas où ce bloc fait défaut, la création d'exemplaire n'entraîne que la mise à jour initiale des variables de l'objet créé.

Lors de la recopie d'un exemplaire, les commandes du paragraphe «COPY» sont exécutées de façon séquentielle par la copie. Les paramètres de classe sont hors de portée dans ce bloc. Dans le cas où ce bloc fait défaut, les variables d'état de l'exemplaire copié contiendront les mêmes valeurs que l'original.

Nous présentons les commandes de ARCHE, puis ses expressions. Dans une première lecture de ce rapport, les deux paragraphes suivants peuvent être ignorés.

III.4.3 Commandes

Nous ne présentons que les commandes «séquentielles».

- la commande d'affectation permet de remplacer le contenu d'une variable par la valeur d'une expression. Nous avons retenu la syntaxe habituelle pour cette commande :

  ```
  LHS := exp
  ```

  où «LHS» est un signifiant mutable auquel la valeur de «exp» est affectable (cf. V.1.3, page 78)
- l'appel de procédure permet d'imbriquer un calcul dans un autre en fournissant une gestion automatique de leurs contextes d'exécutions. Dans la syntaxe :

  ```
  Ident ( exp, ..., exp )
  ```

  l'identificateur doit désigner une procédure déclarée dans la classe ou procédure englobante. Les paramètres sont passés par valeur.
- nous avons retenu les commandes habituelles pour le branchement conditionné et les diverses itérations. Toutefois, l'index de la boucle *pour* est implicitement déclaré et sa portée est limitée au corps de la boucle. Dans la syntaxe :

  ```
  FOR Ident := exp TO exp BY constExp DO ... END
  ```

  l'occurrence de «Ident» équivaut à la déclaration suivante :

  ```
  VAR Ident : [init .. finit]
  ```

  où «init» est la valeur de l'expression figurant avant (resp. après) «TO» si la valeur du pas («constExp») est positive (resp. négative). «finit» est la valeur de l'autre expression.

III.4.4 Expressions

Une expression est syntaxiquement formée d'opérations et d'arguments, eux-mêmes des expressions. Un identificateur, une expression littérale ou encore un type sont des exemples d'argument. La valeur d'une expression est récursivement déterminée par l'application de l'opération aux valeurs des expressions arguments. L'ordre de cette évaluation est arbitraire sauf pour certaines opérations dont l'ordre est signalé dans la présentation.

III.4.4.1 Littérales

Les littérales adoptées sont, dans l'ensemble, tout à fait classiques. Nous fournissons quelques exemples et contre-exemples de littérales en indiquant leur type :

```
124          (* entier decimal   *)   1Q  (* ni char ni int          *)
1650 0677 B  (* entier en octal  *)   8B  (* pas de 8 en octal       *)
101 C        (* char(65)='A'     *) 700C  (* seul 8 bits comptent    *)
0AB1 H       (* entier hexadec.  *) FF09H (* doit commencer par 0-9  *)
12.5         (* float            *) .24   (* idem                    *)
56.E04       (* float 560 000.0  *) 5.E   (* incomplet               *)
56.024E-2    (* float .56024     *)
'A' "B"      (* caracteres A,B   *) 'A"   (* desequilibre guillemets *)
'toto'       (* sequence de char *)
"titi"       (* idem             *)
NIL          (* unique objet typ\'e par la vue NULL *)
```

III.4.4.2 Constantes de classe

Une constante de classe est une littérale, ou un identificateur introduit par un type énumération, ou une expression formée à l'aide des opérations primitives présentées dans l'annexe 1 appliquées à des constantes de classe, ou le nom d'une constante déclarée dont l'expression associée est une constante de classe. De plus, les trois constructeurs présentés ci-après permettent de définir des constantes de classe structurées.

- l'expression :

```
TypeOrdinal { exp , .., exp }
```

permet de dénoter un ensemble dont les éléments sont issus du type ordinal `TypeOrdinal`[4]. Ces éléments sont dénotés par la valeur des constantes de classe `exp`[5].

- l'expression :

```
(@ exp , ..., exp @)
```

permet de dénoter une structure dont la valeur de chaque champ est dénotée par une constante de classe. Afin d'éviter toute ambiguïté, la déclaration d'une constante structurée n'est possible que si son type est fourni explicitement. Par exemple, la déclaration suivante est autorisée :

```
CONST
  RevolutionURSS : DateNaissance = (@ 91, Aout @);
```

alors que celle-ci ne l'est pas :

```
CONST
  SeismeMexico : (@ 85, Septembre @);
```

- l'expression :

```
< exp , ..., exp >
```

permet de dénoter une séquence dont la valeur de chaque élément est dénotée par une constante de classe. Ces constantes doivent être typée de façon homogène. Cette expression est de type «`SEQ [1..k] OF T`», où `k` correspond au nombre d'expressions constituant la séquence, et `T` est le type commun à celles-ci.

III.4.4.3 Constantes d'objet

Une constante d'objet est une constante de classe, ou une expression formée à l'aide des opérations primitives (cf. ci-avant) appliquées à des constantes d'objet, ou le nom d'une constante déclarée dont l'expression associée est une constante d'objet, ou un paramètre de classe. De plus, toute classe déclare implicitement une constante d'objet dénotée par le mot-clef «`SELF`». Nous n'avons pas trouvé

[4]Les constantes introduites par les types ordinaux primitifs présentés au paragraphe III.2.1 ne sont pas préfixées par le nom des dits types.

[5]La notation d'intervalle, *e1* .. *en*, permet d'exprimer une suite de valeurs ordinales successives.

dans la littérature de nom français pour cette constante. Guidés par sa signification dans la programmation par objets, nous la nommons *identité*. L'identité d'un objet est de type vue, «`CURRENT`» pour être exact (cf. § III.3.2). Lors de la création d'un objet, son identité est affectée de la valeur de cet objet. La dénotation du mécanisme d'héritage présentée dans [34] montre que le rôle de cette constante lui est essentiel. Les constantes d'objet peuvent être structurées au moyen des constructeurs de structure et de séquence présentés au paragraphe précédent.

III.4.4.4 Signifiants

Un signifiant est une variable, ou une expression formée à partir d'une expression et de l'opération de sélection associée à la structure ou à la séquence :

- l'expression :

  ```
  exp . NomDeChamp
  ```

  permet d'adresser le champ `NomDeChamp` de la structure dénotée par `exp`. Elle a le type et la valeur du champ nommé.

- l'expression :

  ```
  exp [ exp1, ..., expK ]
  ```

  permet d'adresser l'élément de la séquence dénotée par `exp`, indéxé par les valeurs dénotées par les expressions `exp1` à `expK`. Son type et sa valeur sont ceux de l'élément séleccioné de la séquence.

Le constructeur de séquence :

```
< exp1, ..., expK >
```

appliqué à une liste d'expressions, de type homogène, est aussi un signifiant. Il permet d'adresser cette liste d'expressions en tant que séquence.

III.4.4.5 Expressions

Une expression est une constante d'objet, ou un signifiant, ou une expression formée à l'aide des opérations primitives appliquées à des expressions.

Le mot-clef «`NEW`», suivi d'un identificateur de classe et d'une liste d'expressions entre parenthèses, est une expression. Elle est du type réalisé par la classe et sa valeur est l'identité d'un nouvel exemplaire de cette classe (cf. III.4.2).

Le mot-clef «`COPY`», suivi d'une expression dénotant un objet, est une expression. Elle est du type de cet objet et sa valeur est l'identité de la copie de ce dernier (cf. III.4.2). Si «`COPY`» n'est pas suivi d'une expression, alors l'expression est équivalente à «`COPY SELF`»[6].

Un identificateur de procédure, suivi d'une liste d'expressions, est une expression. Son type et sa valeur sont ceux du résultat de la procédure.

Finalement, une expression de type référence, suivie du symbole « ! », d'un identificateur de méthode et d'une liste d'expressions, est une expression. Son type et sa valeur sont ceux du résultat de la méthode.

[6]Pour les raisons qui sont exposées dans le chapitre suivant, cette expression provoquerait un interblocage. C'est pourquoi l'expression alternative est retenue.

Chapitre IV

Parallélisme

IV.1 Processus

Le parallélisme explicite repose sur la notion de *processus*. Le terme processus, originaire des systèmes d'exploitation [35], désigne les aspects opératoires de l'exécution d'un programme. Cette notion est intimement liée au phénomène d'*observation* et à la définition d'*événements* considérés significatifs. L'exécution d'un programme dans un système informatique est généralement perçue comme une suite de changements d'état du dit système. Les événements significatifs d'une exécution sont les changements qu'un observateur est capable de discerner et qu'il choisit de retenir. Ils déterminent la façon d'observer le système et le vocabulaire propre à la description de son comportement. La littérature propose plusieurs définitions de processus [36, 37, 38, 39] parmi lesquelles nous retenons celle énoncée dans [39] :

Définition IV.1 *Un processus est la description du comportement d'un système*[1] *dont un ensemble fini d'événements observables a été choisi pour le caractériser.*

Cette définition peut servir de guide dans la détermination d'une entité correspondant au processus dans un langage de programmation. Pour ce faire, il convient de se demander quels sont les événements observables de l'exécution d'un programme écrit dans le langage considéré.

L'exécution d'un programme écrit en ARCHE donne lieu à la création d'objets, et provoque le déroulement d'appels de méthode de ces objets et de ceux rémanents de l'exécution d'autres programmes. Nous soulignons que l'état d'un

[1] La définition citée utilise le terme *objet* au lieu de système. Nous avons préféré ce dernier afin d'éviter toute confusion avec les *objets* du langage ARCHE.

objet n'est accessible que par l'appel de ses méthodes. Nous disons que la protection de l'état ainsi acquise constitue une *membrane* pour l'objet. Cette membrane introduit les notions d'*intérieur* et d'*extérieur* relatives à un objet. Les événements concernant l'exécution d'un programme écrit en ARCHE peuvent être regroupés suivant la localisation de l'observateur par rapport aux objets observés. Si l'observateur est placé à l'intérieur de l'un des objets, les événements significatifs sont ceux qui ont trait aux modifications des variables de cet objet, et aux changements dans son contrôle. Si l'observateur est placé à l'extérieur des objets, les événements significatifs sont les appels de méthode, et les variations dans l'ensemble des objets observés. Chacun de ces points de vue donne naissance à une entité dans le langage ARCHE correspondant à un processus : la classe et la vue. Les deux entités décrivent le comportement des objets ARCHE. Les descriptions diffèrent dans les événements observés.

Les deux formes alternatives de créer un processus en ARCHE sont la création d'exemplaire de classe, et la copie d'objet. Dans les deux cas le processus se comporte vis à vis de son état comme l'indique sa classe, et vis à vis des autres processus comme l'indique son type, c'est-à-dire la vue réalisée par sa classe.

La création d'un exemplaire de classe se fait au moyen de l'expression «`NEW <nom de la classe>(<parametres>)`». La valeur de cette expression est la référence à un nouvel objet, exemplaire de la classe indiquée. L'état de l'objet créé est déterminé par les valeurs initiales de ses variables. La création de cet objet provoque l'exécution du bloc «`NEW`» de sa classe, et ce en parallèle avec l'objet créateur. Une fois que la fin de ce bloc est rencontrée –l'absence de celui-ci étant assimilée à cette fin– le processus se met en attente d'appel de méthode.

La copie d'un objet se fait au moyen de l'expression «`COPY <objet>`». La valeur de cette expression est la référence à un nouvel objet, exemplaire de la même classe que l'objet indiqué. Si aucun objet n'est précisé après «`COPY`», alors une copie de l'objet courant est créée. L'état de l'objet créé est déterminé par les valeurs courantes de l'objet original. Il est donc extrêmement important de spécifier les états dans lesquels l'objet original permet cette copie. L'exclusion mutuelle dans l'exécution des méthodes et de la pseudo-méthode «`COPY`» est adoptée. La création d'une copie d'objet provoque l'exécution du bloc «`COPY`» de sa classe, et ce en parallèle avec l'objet copié. Une fois que la fin de ce bloc est rencontrée, le processus se met en attente d'appel de méthode. Il est important de remarquer que la copie se fait au premier niveau des valeurs –copie superficielle (*shallow copy*). Ainsi, la copie d'une référence n'entraîne pas la copie de l'objet référencé de façon automatique. Ce nonobstant, cette forme de copie –copie profonde (*deep copy*)– est aisément obtenue à partir des mécanismes décrits, moyennant une programmation explicite.

IV.2 Communication

Deux objets ARCHE communiquent par appel de méthode. Ce mécanisme est une abstraction d'un échange particulier de messages dans lequel les envois et les réceptions sont contraints de suivre le protocole suivant : l'envoi est toujours suivi d'une réception, la réception est toujours suivi d'un envoi. L'appel de méthode est une abstraction en ce sens qu'il libère le programmeur de l'écriture explicite des envois et des réceptions de messages. Pour le programmeur, l'appel de méthode correspond à un appel de procédure traditionnel. Nous ne parlons pas d'*appel de procédure à distance* (RPC) [40] bien que les deux notions soient très proches, car ce terme nous paraît trop lié à une mise en œuvre particulière de ce protocole dans les systèmes d'exploitation répartis. L'appel de méthode et le RPC déchargent le programmeur de la localisation des objets répartis. Le processus et l'objet sont indissociables. Ils sont rémanents à l'exécution de leur créateur. D'un point de vue conceptuel, objet et processus ne sont qu'un. Nous rejoignons sur ce point le modèle *acteur* de G. Agha [41, 42], exception faite des appels bloquants et du système de type présents dans ARCHE.

La primitive de communication offerte dans ARCHE traite de façon asymétrique l'appelant et l'appelé. L'appel de méthode, exprimé par «`<destinataire>!<methode>(<parametres>)`», peut figurer là où une expression est permise. Cette expression permet au programmeur de choisir le point du programme où une communication a lieu. L'appel de méthode exige que le correspondant soit explicitement désigné. Par contre, la réception d'un appel de méthode est implicite. Le programmeur ne peut pas choisir le point du programme où cette réception est effectuée : il est fixé dans la sémantique du langage. De plus, le programmeur ne peut pas désigner un appel particulier et il ignore l'identité de l'appelant.

Un appel de méthode est exécuté, du point de vue de l'appelant, comme un appel de procédure. Que la méthode rende ou non un résultat, l'appelant est bloqué jusqu'à ce que l'appel soit terminé. Nous pouvons envisager d'accroître le degré de parallélisme en autorisant une forme asynchrone d'appel dans le cas où la méthode ne rend pas de résultat. Si nous considérons que les effets de bord d'une méthode sont une forme de résultat, nous sommes obligés de conclure que les méthodes sans résultat sont inexistantes dans la pratique. Cette dernière remarque prend toute son importance dans le cadre d'un langage parallèle. La simplicité et l'habitude de l'utilisation de l'appel de procédure sont des atouts que nous avons choisi de ne pas abandonner.

L'exécution d'un appel de méthode, du point de vue de l'appelé, n'est pas explicitement commandée par le programmeur. L'intégration de la notion de processus à celle d'objet implique l'exclusion mutuelle dans l'exécution de ses

méthodes. Un seul appel peut être traité à la fois. Un appel de méthode est donc exécuté par l'objet appelé dès que nulle autre exécution n'est en cours. De ce point de vue, l'état d'un objet est protégé comme il le serait par un *moniteur* [43]. La sérialisation des appels de méthode est sans doute une politique draconienne, mais elle a le mérite de permettre la formulation de pré- et de post-conditions. Nous donnons plus de souplesse à cette politique au paragraphe IV.4.

IV.3 Synchronisation

Nous abordons à présent la synchronisation sur condition. Il est intéressant de s'interroger sur le sens de cette synchronisation dans le cadre que nous nous sommes fixé. Nous le faisons en répondant à deux questions :

1) Dans quelles circonstances le programmeur a-t-il besoin d'exprimer l'attente d'un objet ?

- Dans le cas où une «intervention extérieure à l'objet» s'avère nécessaire et qu'elle n'est pas disponible. D'après nos définitions, un objet ne peut coopérer qu'avec d'autres objets. Or ceux-ci n'interagissent que par appel de méthode. L'«intervention» dont il s'agit est simplement un appel de méthode.

2) Dans quels cas l'objet destinataire se trouverait-il dans l'impossibilité de traiter cet appel ?

- Deux scénarios sont possibles. Le premier se présente lorsque l'objet est engagé dans le traitement d'un autre appel. Dans ce cas, le programmeur n'a pas d'attente à exprimer : elle est implicite dans la politique de sérialisation des appels adoptée. Le second scénario fait intervenir un objet destinataire dont l'état des variables ne permet pas l'exécution de la méthode appelée (par exemple, l'opération de dépilement lorsque la pile est vide). Il revient au programmeur d'exprimer cette indisponibilité.

Ainsi, la synchronisation sur condition dans le langage ARCHE peut être identifiée à l'expression d'un conditionnement de l'exécution d'une méthode fondé sur l'état de l'objet.

Nous revenons sur la notion de type *vue* présentée précédemment (*cf.* § III.3). Une vue détermine la signature des méthodes offertes par l'ensemble des objets de ce type. Le système de type de ARCHE permet de garantir, à la compilation, que toute méthode appelée est bien offerte par l'objet destinataire et que les paramètres requis sont correctement fournis.

Mais nous venons d'indiquer que l'offre d'une méthode par un objet n'est pas valide *tout le temps*. Bien que le type de la variable contenant l'objet indique

que la méthode est fournie, l'appel peut échouer car l'état de l'objet interdit son exécution. Nous sommes confrontés à un problème similaire à celui des variables initialement non affectées et des pointeurs ballants.

Exemple IV.1

```
     VAR a,b: INT;                    VAR a,b: INT;
         p  : POINTER TO INT;             p  : POINTER TO INT;

(1)  a  := b + 1;                     b  := 5;
(2)  b  := 5;                         a  := b + 1;
(3)  p^ := b + 1;                     p  := ADR(a);
(4)  p  := ADR(a);                    p^ := b + 1;
```

Dans l'exemple IV.1, les deux fragments de programmes sont acceptés par un vérificateur de types car les opérations «+» et «^» sont respectivement autorisées pour les types «INT» et «POINTER». Cependant, les commandes de la colonne de gauche sont dépourvues de sens. En effet, à la ligne *(1)*, la variable «b» n'a pas de valeur affectée et l'opération d'addition n'a pas de sens. À la ligne *(3)*, le pointeur «p» est suivi alors qu'il est ballant. Les opérations «+» et «^», bien que syntaxiquement valides, ne peuvent pas être appliquées dans les états indiqués.

Afin de détecter et de rejeter ce genre de programmes lors de la compilation, la notion d'*état de type* à été proposée pour les langages fortement typés [44]. Ainsi, un type n'est plus seulement caractérisé par l'ensemble des opérations qu'il fournit, mais aussi par un automate dont les états indiquent les opérations applicables, et les transitions correspondent à l'application d'une opération. Une vérification de type plus poussée est alors possible.

Nous appliquons au type vue cette notion d'état de type afin d'exprimer la disponibilité des méthodes offertes par un objet. Nous proposons de définir la disponibilité d'une méthode comme suit :

Définition IV.2 *Une méthode est* passante *dans un état de l'objet si cet état permet un traitement immédiat de l'appel de celle-ci en l'absence d'autres appels. Dans le cas contraire, il s'agit d'un état blocant de la méthode. La méthode est alors* blocante *dans cet état.*

Une vue peut être enrichie d'un *ensemble* d'états de type que nous appelons *états de synchronisation*, ou *états de vue*, ou plus simplement *états* lorsqu'il n'y a aucun risque d'ambiguïté. Dans une vue, un état de synchronisation indique les méthodes passantes dans l'état où un objet typé par celle-ci est en attente d'appel. La clause «STATE» d'une vue permet de déclarer des états de synchronisation. Une déclaration d'état a la forme suivante :

```
STATE
 ...
 NOMi : { Mi1, .., Mik }
 ...
```

Cette déclaration associe à l'état de vue «NOMi» les méthodes «Mij» (0<j<k+1) de la vue.

Lors de sa création, un objet se trouve, par défaut, dans l'état de synchronisation dans lequel toutes les méthodes de son type sont passantes. Cependant, un ensemble d'états initiaux peut être indiqué dans la déclaration d'une vue. La liste de leur noms, entre accolades, doit alors figurée à la suite des paramètres de vue. Dans ce cas, le bloc «NEW» de la classe réalisant la vue détermine l'état de synchronisation initial, à l'aide de la commande «BECOME» définie par la suite.

L'exécution d'une méthode provoque un changement d'état que nous nommons une *transition*. Nous appelons l'état atteint par cette transition un *post-état* de la méthode. L'ensemble des post-états associé à une méthode détermine les transitions déclenchables par celle-ci. La clause «POST» permet de déclarer les post-états de chaque méthode. La forme de cette déclaration est la suivante :

```
POST
 ...
 Mi : { NOMi1, .., NOMik }
 ...
```

Cette clause indique que l'exécution de la méthode «Mi» conduit l'objet adressé à adopter un des états «NOMij» (0<j<k+1) pour le traitement de l'appel suivant.

La clause «POST» est obligatoire dès lors qu'une clause d'états de synchronisation, «STATE», a été déclarée. L'absence d'une méthode dans la clause «POST» traduit le fait que cette méthode ne change pas l'état de synchronisation de l'objet adressé. Dans le cas où la vue ne comporte pas de clause de synchronisation, les méthodes sont toujours passantes, et donc, aucune transition d'état de synchronisation n'accompagne leurs exécutions.

Lors d'une recopie, l'objet copie acquiert, par défaut et s'il y a lieu, l'état de synchronisation de l'objet copié prévalant au début du traitement de la pseudo-méthode «COPY». Cependant, un état de synchronisation différent peut être établi par le bloc «COPY», si la vue de l'objet indique que cet état figure dans les post-états de «COPY». Les éventuels post-états de «COPY» sont spécifiés come ceux des méthodes, dans la clause «POST». Toutefois, la transition décrite par cette déclaration ne concerne pas l'objet adressé, mais la copie créée par l'exécution de «COPY», contrairement aux autres méthodes.

Nous illustrons le mécanisme des états de synchronisation par l'exemple bien connu du sémaphore n-aire.

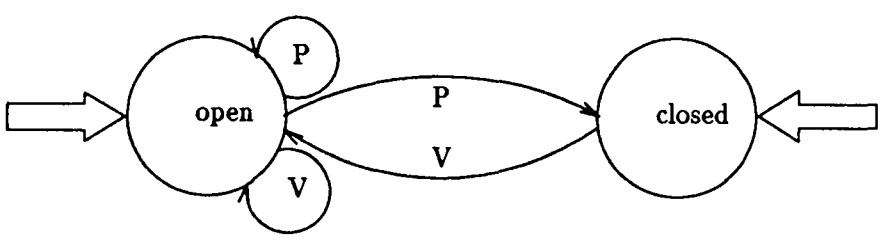
Exemple IV.2

```
LIBRARY Example =
 Semaphore = VIEW ( initially: CARDINAL ) { open, closed }
              P : () () ;
              V : () () ;
             STATE
              open   : { P, V };
              closed : { V };
             POST
              P: { open, closed };
              V: { open };
             END;
END Example;
```

La clause «`STATE`» introduit deux états de synchronisation : «`open`» pour dénoter un sémaphore ouvert, acceptant donc de participer aux appels «`P`» et «`V`», et «`closed`» pour un sémaphore fermé, bloquant pour tout appel «`P`». La clause «`POST`» indique qu'après l'exécution d'une opération «`P`», le sémaphore peut devenir fermé ou rester ouvert –cela dépend de sa valeur initiale, «`initially`», et de son histoire. Après une opération «`V`», il devient ouvert. Les accolades suivant les paramètres de vue permettent d'indiquer l'état de synchronisation initial des objets de type sémaphore. En l'occurence, la valeur initiale passée en paramètre lors de la création d'exemplaire détermine si le sémaphore se trouve ouvert ou fermé.

Nous pouvons représenter le type «`Semaphore`» comme un automate d'états fini non-déterministe :

IV.2...



Les transitions d'état de synchronisation indiquées par une vue donne une description plus précise, quoique non-déterministe, du comportement des objets

typés par celle-ci. Le type vue correspond à la description d'un processus tel que nous l'avons défini dans les paragraphes précédents. Les événements significatifs dans cette description sont les appels de méthodes. C'est pourquoi nous pouvons aussi retrouver le type «`Semaphore`» dans le processus tCSP [39][2] suivant :

IV.2...

$$
\begin{array}{lcl}
SEM & \triangleq & V \rightarrow OPEN \sqcap (P \rightarrow SEM \,[]\, V \rightarrow OPEN) \\
OPEN & \triangleq & P \rightarrow SEM \,[]\, V \rightarrow OPEN \\
\alpha\, SEM & = & \{\, P, V \,\} = \alpha\, OPEN
\end{array}
$$

Il est intéressant de remarquer que le type «`Semaphore`» reste une *approximation* du comportement d'un sémaphore n-aire. Il y est seulement indiqué qu'un appel «`P`» peut être forcé d'attendre un appel «`V`». Cependant, rien n'indique que cet appel sera effectivement bloqué, ou encore qu'il existe une relation de cause à effet entre la valeur initiale, «`initially`», le nombre d'appel «`V`» ou le nombre d'appel «`P`», et cette mise en attente ! Ainsi, l'occurence dans la clause «`POST`» d'une méthode ayant plus d'un post-état exprime un choix non-déterministe interne à l'objet, et donc indépendant de son environnement.

Les transitions sont commandées explicitement par le programmeur au moyen de l'instruction «`BECOME`» suivie du nom de l'état visé. Afin de l'illustrer, nous présentons une classe réalisant le type de l'exemple IV.2.

[2] Le formalisme tCSP (pour *theoretical CSP*) ne doit pas être confondu avec la proposition de Hoare formulée dans [45]. Il s'agit d'une algèbre de processus communicants et d'un ensemble de lois qui permettent de faire du calcul de programmes parallèles. L'opérateur de choix *externe* est dénoté par $\_\,[]\,\_$ et celui de choix *interne* par $\_\sqcap\_$. Les événements P et V forment l'alphabet des processus *SEM* et *OPEN*.

IV.2...

```
CLASS Counter IMPLEMENTS Example.Semaphore =
 VAR Value: INTEGER := initially;
 PROCEDURE P = BEGIN
   DEC( Value );
   IF Value = 0
    THEN BECOME closed
    ELSE BECOME open
   END
 END P;
 PROCEDURE V = BEGIN
   INC( Value );
   BECOME open
 END V;
BEGIN
 IF Value > 0
  THEN BECOME open
  ELSE BECOME closed
 END
END Counter;
```

Le compilateur est chargé de vérifier que les états de synchronisation accessibles par l'exécution d'une méthode, ou du corps de la classe, sont bien les post-états indiqués par la vue réalisée. Si plusieurs transitions sont commandées dans une suite d'instructions, la dernière exécutée prévaut. La commande «BECOME» indique l'état de synchronisation dans lequel l'objet traitera le prochain appel de méthode. Cet état n'est atteint que lorsque l'exécution courante est terminée.

Parmi les mécanismes de synchronisation proposés par d'autres langages à objets parallèles fortement typés, citons trois approches fondées sur le conditionnement de l'exécution des méthodes :

- le langage HYBRID [46] propose des files d'attente qui peuvent être associées aux méthodes. Une file peut contrôler l'accès de plusieurs méthodes, la réciproque étant interdite. Une file d'attente peut être ouverte ou fermée. Un appel de méthode est mis en attente lorsque sa file associée est fermée. Autrement, les appels sont traités l'un après l'autre. Deux commandes sont fournies pour gérer l'ouverture et la fermeture des files.
- la clause «CONTROL» du langage GUIDE [47], déclarée dans la classe, permet d'associer à chaque méthode une condition d'activation. Cette condition est une expression booléenne pouvant porter sur l'état de l'objet, les valeurs passées en paramètre à la méthode et les *compteurs de synchronisation* [48].

Ces derniers, associés à chaque méthode, indiquent le nombre d'appel reçu, le nombre d'appel accepté, le nombre d'appel exécuté, le nombre d'appel en cours d'exécution et le nombre d'appel mis en attente. Ils sont gérés automatiquement par la machine d'exécution.

- la commande de choix non-déterministe («`SELECT`») et l'acceptation explicite, «`ACCEPT(m1,..,mK)`», des appels de méthode dans la famille des langages POOL [49] permettent de conditionner l'exécution d'une méthode. Remarquons toutefois que les expressions de synchronisation sont disséminées dans le texte de la classe alors que les précédentes solutions adoptent un contrôle centralisé.

Nous trouvons deux inconvénients majeurs à ces propositions. Le premier est lié à la notion de sous-type, et le second au mécanisme d'héritage. N'ayant pas encore abordé ces sujets, nous ne pouvons qu'esquisser les dits inconvénients.

1) La notion de sous-type permet d'employer en toute sécurité un objet d'un type à la place d'un objet d'un autre type. Dans un cadre parallèle, cela revient à utiliser un processus à la place d'un autre. Les instructions de synchronisation déterminent en grande partie le comportement des processus. Les contraintes de synchronisation n'étant pas exprimées dans le type, il est difficile de définir une relation de sous-type intéressante, par exemple celle de *raffinement* [50].
2) Le mécanisme d'héritage permet de réutiliser le code d'autres classes, *sans le modifier*, dans l'écriture d'une classe que nous appelons l'*héritière*. Il arrive fréquemment que de nouvelles méthodes soient introduites par l'héritière. Se posent alors les difficultés suivantes :
   - HYBRID – comment exprimer l'ouverture ou la fermeture des nouvelles méthodes au sein des méthodes héritées sans modifier leur code ?
   - GUIDE – comment intégrer dans la clause «`CONTROL`» les conditions d'activation des nouvelles méthodes autrement qu'en utilisant l'opération couper-et-coller d'un éditeur de texte pour ensuite l'adapter ?
   - POOL – comment étendre les instructions de choix et d'acceptation d'appel de méthode disséminées dans toutes les méthodes héritées pour qu'elles reflètent la disponibilité des nouvelles méthodes ?

La difficulté posée par la définition d'un ordre «intéressant» sur les types en présence d'un mécanisme de synchronisation est rapportée dans [51]. Dans cet article, il est proposé d'étendre la notion de type afin qu'il constitue une spécification partielle du comportement des objets. Une notion de type pour les

objets–processus a été récemment proposée par les mêmes auteurs dans [52]. Nous tenons à signaler que le type vue du langage ARCHE a été conçu de façon indépendante à ces recherches.

L'interférence entre le mécanisme d'héritage et celui de synchronisation a d'abords été rapporté dans [18]. Cette même référence nous a guidé dans nos choix, en particulier dans l'utilisation de la commande «`BECOME`». Il faut néanmoins rappeler que les langages à *acteurs* sont dépourvus d'un système de type.

Nous convenons que le mécanisme de synchronisation retenu pour le langage ARCHE est strictement moins expressif que ceux des langages POOL ou GUIDE. Nous avons choisi de favoriser l'intégration des divers mécanismes proposés aux dépens de leur pouvoir d'expression individuel. Nous remettons au chapitre V le lecteur désirant plus de précisions au sujet de l'intégration synchronisation–type–héritage.

Nous revenons sur la politique de sérialisation présentée au paragraphe précédent. Cette politique est draconienne : tous les appels de méthode sont sérialisés. Nous cherchons à l'affiner sans renoncer à l'existence des pré- et post-conditions des méthodes. Rappelons que dans un cadre parallèle, l'existence d'une pré- et d'une post-condition est intimement liée à la notion d'*atomicité*. Nous disons qu'une liste de commandes est exécutée de façon atomique si l'environnement ne peut pas déceler d'état intermédiaire entre l'état initial et l'état final. Cette atomicité est obtenue pour les méthodes dès lors qu'objet et processus ne sont qu'un. Ce choix semble raisonnable pour des objets de petite taille. Cependant, lorsque l'objet est d'une taille importante et l'exécution de ses méthodes devient coûteuse en temps, une politique plus souple est souhaitable. Les deux paragraphes suivants présentent les deux notations fournies par le langage ARCHE pour exprimer un plus grand degré de parallélisme : la méthode *observatrice* et le *multi-appel*.

IV.4 Plus de disponibilité

Notre modèle de programmation repose sur les objets. L'introduction des objets dans le langage ARCHE est fortement motivée par la théorie des *types abstraits* [19]. Dans la définition d'un type abstrait, deux classes d'opérations sont distinguées d'après leurs *signatures*. La signature d'une opération est constituée de deux listes de types, la première indiquant le type des arguments, et la seconde celui des résultats. Nous les appelons respectivement membre gauche et membre droit de la signature. Lorsque le membre droit d'une signature se réduit au type en cours de définition nous disons que l'opération associée peut être une *génératrice* des valeurs de ce type. Lorsque le type en cours de définition ne figure

que dans le membre gauche d'une signature, nous disons que l'opération associée est une *observatrice* de ses valeurs.

Exemple IV.3

[*Elem*]
$\{\} : \rightarrow Bag$
insert, delete : $Elem \times Bag \rightarrow Bag$
$\_ \in \_ : Elem \times Bag \rightarrow Boolean$
count : $Elem \times Bag \rightarrow \mathbb{N}$
⊢
Bag **generated by** {}, insert
Bag **partitioned by** delete, $\in$, count
|
$\forall b : Bag;\ e, e' : Elem \bullet$
$\neg (e \in \{\})$
$e \in \text{insert}(e', b) \Leftrightarrow (e = e') \vee e \in b$
|
$\text{delete}(e, \{\}) = \{\}$
$\text{delete}(e, \text{insert}(e, b)) = b$
$\text{delete}(e', \text{insert}(e, b)) = \text{insert}(e, \text{delete}(e', b)) \Leftrightarrow e \neq e'$
|
$\text{count}(e, \{\}) = 0$
$\text{count}(e, \text{insert}(e, b)) = \text{count}(e, b) + 1$
$\text{count}(e, \text{insert}(e', b)) = \text{count}(e, b) \Leftrightarrow e \neq e'$

L'exemple IV.3 emprunte le langage de spécification LARCH [20] pour présenter le type multi-ensemble. Les génératrices retenues par cette spécification sont l'ensemble vide « {} » et l'opération d'ajout « insert »[3]. Les observatrices sont le test d'appartenance, « $\in$ », et le nombre d'occurrences « count ». Le rôle de ces opérations dans la description des valeurs appartenant au type est différent. Un certain nombre de génératrices servent à caractériser la forme de ces valeurs. D'autres génératrices servent à ramener les valeurs aux formes choisies et induisent, avec les observatrices, une relation d'équivalence comme le montre les équations.

Il est intéressant de rechercher des notions similaires dans le paradigme objet. Nous avons déjà signalé au paragraphe III.3 que le type vue ne va pas sans rappeler un type abstrait de donnée. Cependant, la signature des méthodes figurant dans une vue ne suffit pas à distinguer une génératrice d'une observatrice car les objets possèdent un état. Pour les distinguer, il nous faut examiner la

[3] D'après notre définition, l'opération « delete » peut aussi être une génératrice puisque sa signature est identique à celle de « insert ». En pratique, elle est beaucoup moins intéressante comme génératrice que comme *normalisatrice*.

classe. Nous appelons méthode *génératrice* celle dont la procédure associée modifie l'état de l'objet, et méthode *observatrice* celle dont la procédure associée le consulte. Nous admettons que l'état d'un objet est entièrement déterminé par le contenu de ses variables. Nous définissons une méthode observatrice comme suit :

Définition IV.3 *Une méthode est* observatrice *si les commandes d'affectation dans son corps n'ont pour membre gauche que des variables locales, s'il n'y figure aucune commande* «BECOME» *et si toute procédure appelée dans son corps correspond à une méthode observatrice. La pseudo-méthode* «COPY» *est une observatrice.*

À notre connaissance, seuls les langages de la famille POOL font la différence entre ces deux familles de méthodes (les «METHOD» et les «ROUTINE»). Cette discrimination permet d'exploiter, en toute sécurité[4], un plus grand degré de parallélisme.

remarque (peut être ignorée en première lecture)

Les appels de méthodes observatrices ne peuvent pas être traités comme s'il s'agissait d'une simple création de processus. Dans ce cas, tout se passerait comme si l'objet créait une copie de lui-même –sans exécuter les commandes associées au bloc «COPY»– pour ensuite lui *déléguer* l'appel. L'objet original serait immédiatement prêt à traiter d'autres appels. La copie, dont l'identité ne serait connue que de l'appelant –et ce à son insu– traiterait l'appel d'observatrice, rendrait éventuellement un résultat, puis terminerait. Cependant, quelques précautions sont nécessaires si nous ne voulons pas renoncer à la possibilité de formuler des pré- et des post-conditions pour ces méthodes. En effet, les observations ainsi recueuillies risquent fort d'être *incohérentes*. Avant de poursuivre notre raisonnement, nous définissons la notion de *partie privée* de l'état d'un objet et nous tâchons de préciser la notion de *cohérence* d'une observation.

Définition IV.4 *Une variable globale à un objet, ou locale à une de ses procédures, est* inaccessible *à l'environnement de cet objet si :*

(1) la variable ne dénote pas une référence, ou

(2) la référence dénotée n'a jamais été passée, de façon directe ou indirecte, à l'environnement.

[4]Nous précisons cette notion dans ce qui suit.

L'ensemble des variables d'un objet inaccessibles à son environnement constitue la partie privée *de son état.*

Cette définition n'est pas très utile d'un point de vue constructif, mais elle est néanmoins nécessaire à la définition suivante :

Définition IV.5 *L'observation d'un objet est dite* cohérente *par rapport à un environnement si l'observation récursive de la partie privée de son état correspond à l'une des observations que l'on recueuillerait en l'absence de toute autre activité de l'environnement.*

Autrement dit, nous considérons qu'une méthode observatrice doit rendre les mêmes (classes de) résultats, qu'elle soit exécutée en exclusion mutuelle des autres méthodes, ou qu'elle soit exécutée en parallèle avec celles-ci. Nous présentons un exemple pour illustrer notre propos. Soient les déclarations suivantes :

Exemple IV.4

```
Cumul =                    | CLASS Compteur IMPLEMENTS Exemple.Cumul =
 VIEW (init: CARDINAL)     |  VAR v : CARDINAL := init;
  succ: ()();              |  PROCEDURE succ = BEGIN INC(v) END succ;
 OBSERVER                  |  PROCEDURE val = BEGIN RETURN(v) END val;
  val : ()(r: CARDINAL);   | END Compteur;
 END;                      |
Multi = VIEW (f: Base)     | CLASS MultCompte IMPLEMENTS Exemple.Multi=
  maz  : ()();             |  VAR c : Cumul; b: Base := f;
  ajout: (c: Cumul)();     |      next: Multi := NIL;
 OBSERVER                  |  PROCEDURE maz = BEGIN
  sigma: ()(r:CARDINAL);   |   IF next # NIL THEN next!maz END;
 END;                      |   c := NEW Compteur(0);
-------------------------  |  END maz;
  ...
 PROCEDURE sigma = BEGIN
  IF next # NIL THEN r := next!sigma()*b ELSE r := 0 END;
  RETURN (r+c!val())
 END sigma;                END MultCompte;
```

Le scénario suivant nous fournit un exemple conflictuel par rapport à notre notion de cohérence : après avoir introduit deux compteurs dans une liste c à l'aide des commandes[5] :

[5] L'ajout se fait en queue.

```
...c := NEW MultCompte(10);
   c!ajout(NEW Compteur(4)); c!ajout(NEW Compteur(5)); ...
```

nous pouvons nous demander quelle est la valeur de r après une exécution parallèle des commandes :

```
... r := c!sigma() || c!maz; ...
```

Une observation cohérente de c veux que les valeurs possibles pour r soient 0 ou 54. Or la simple création de processus avec recopie de l'état de l'objet pourrait aussi donner à r la valeur 4.

fin de remarque

L'introduction des méthodes observatrices revient à relâcher la politique d'accès à l'état d'un objet. Nous demandons à présent une politique connue sous l'apellation « n-lecteurs un-rédacteur symétrique » dans les systèmes d'exploitation. Nous étendons le type vue d'une clause, «OBSERVER», qui permet de distinguer les observatrices des autres méthodes. Cette différence syntaxique nous permet de contrôler statiquement la nature des méthodes et des procédures associées, et ce avec les techniques de compilation séparée indispensables au langage ARCHE. Le compilateur est chargé de vérifier que les procédures associées aux observatrices satisfont la définition IV.5.

Nous pouvons nous interroger sur l'intérêt de la clause «OBSERVER». Une analyse statique de la nature des accès à l'état des objets –écriture ou lecture– portant sur l'ensemble des classes d'un programme peut conduire à une exploitation du parallélisme identique à celle-ci. Une politique d'accès encore plus libérale peut sans doute être obtenue à l'aide d'analyses plus poussées [53]. Nous voyons deux inconvénients majeurs à ces solutions : d'une part, le coût de ces analyses serait sans aucun doute prohibitif ; d'autre part, la persistance des objets ainsi que la confidentialité des textes sources réduirait lourdement la protée de l'analyse. Rappelons aussi que nous sommes concernés par la gestion explicite du parallélisme. Les notations que nous proposons permettent de le faire avec plus de souplesse.

La longue remarque de ce paragraphe aura sans doute montré le besoin d'isoler de son environnement un ensemble d'objets afin de leur appliquer un certain nombre d'opérations sans interférences indésirables. Nous traitons de ce sujet dans le suivant paragraphe.

IV.5 Plus de simultanéité

Nous revenons à présent sur les structures de données du langage ARCHE. Un objet peut être structuré au niveau de sa classe, par le mécanisme d'héritage (*cf.* § V), ou par l'utilisation d'autres objets dans la définition de son état. Nous nous intéressons au deuxième cas.

La structure la plus simple d'un état est donnée par la déclaration d'une liste de variables dont certaines sont typées par une vue. Ce simple moyen de structuration suffit pour obtenir toute sorte de graphes dont les nœuds sont d'arité fixe. Les nœuds d'arité variable sont souvent nécessaires dans la pratique. Citons en exemples une figure faite à base de polygones, ou une bibliothèque de livres. Bien qu'il puisse être simulé par une liste chaînée, un nœud d'arité variable facilite l'expression des appels de méthode de par l'accès direct à ses éléments. La séquence, présentée au paragraphe III.3.2.3, page 31, permet de structurer ce genre de nœuds.

Une communication dans le langage ARCHE étant réalisée par un appel de méthode bloquant, la structuration d'un objet est généralement accompagnée d'une sérialisation dans les appels de méthode à ses différents composants. Ceci constitue un goulet d'étranglement pour le parallélisme potentiel des objets structurés. Prenons un exemple pour l'illustrer. Soit «`Figure`» le type des séquences de polygones, dont la vue, «`Polygone`», offre une méthode de translation, «`Move`», et une observatrice d'affichage, «`Draw`».

Exemple IV.5

```
Polygone = VIEW ...
            Move: (POS)();
            ...
           OBSERVER
            Draw: ()();
            ...
           END;
Figure   = SEQ OF Polygone;
```

Nous voulons exprimer que le déplacement d'une figure peut être accompli par le déplacement de ses composants de façon solidaire. Les polygones constituant la figure peuvent bien évidemment être déplacés en parallèle. Tout ordre indiqué pour l'exécution de ces mouvements est superflu. Cependant, les notations présentées jusqu'ici ne permettent pas de l'éviter. Nous pouvons simuler un mode de communication asynchrone, ou appel non-bloquant, à l'aide de la création de processus. Nous sommes alors obligés de créer un certain nombre d'objets, messagers fugaces, chargés d'appeler, indépendamment, les différents polygones

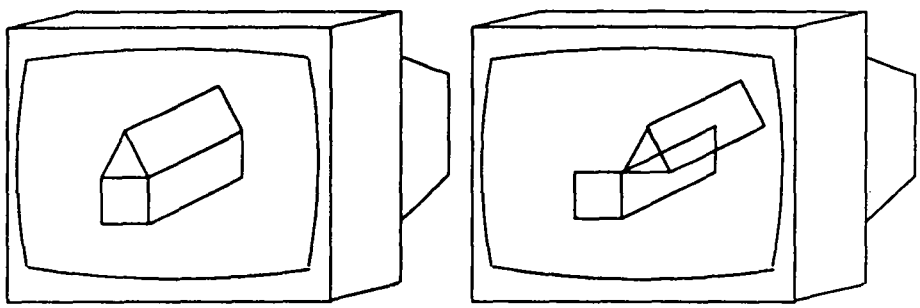
de la figure.

IV.5...

```
VAR F: Figure; vecteur: POS; m: SEQ OF MSG;
...
FOR i:= 1 TO F!LENGTH() DO
 m[i] := NEW Messenger(F[i])
END;
...
```

Remarquons qu'un ordonnancement superflu subsiste dans l'itération conduisant à la création des messagers. Mise à part la lourdeur d'expression, cette solution n'est pas entièrement satisfaisante. Pour l'expliquer, supposons qu'une fois déplacée, nous voulions mettre à jour la représentation de la figure sur un écran en affichant les polygones constituants. Le tracé des composants peut aussi exploiter un environnement parallèle. Nous employons à nouveau la technique des messagers fugaces pour commander l'exécution parallèle des méthodes d'affichage. La figure suivante illustre la faiblesse de la solution proposée :

IV.5...



Avant Après

Certains polygones ont été redessinés *avant* d'être déplacés ! Quelques messagers ont délivré les appels «`Draw`» avant que *tous* les appels «`Move`» n'ai été reçus. Nous ne pouvons faire aucune supposition sur les vitesses relatives d'exécution des différents processus et nous n'avons émis aucune hypothèse sur l'ordre des envois et des réceptions de messages.

Il est clair qu'en munissant les messagers d'une méthode servant d'accusé de reception, une troisième série d'appels viendrait à bout de notre problème. Il nous paraît plus simple d'introduire une forme d'appel, réservée aux objets appartenant à une séquence, que nous dénommons le *multi-appel.*

Le multi-appel d'une séquence d'objets consiste à créer et activer une multiprocédure (cf. § I) dont les composants possèdent des en-têtes identiques. Soit

le type vue «T» décrit ci-après.

Exemple IV.6

```
T = VIEW
     meth : (i1: IN1, .., in: INn)(o1: OUT1, .., ok: OUTk);
    END;
```

Alors le type «S = SEQ OF T» correspond à une pseudo-vue qui étend celle que nous avons associée au type séquence (voir encadré, page III.3.2.3) en lui ajoutant une pseudo-méthode de nom «meth» et de signature :

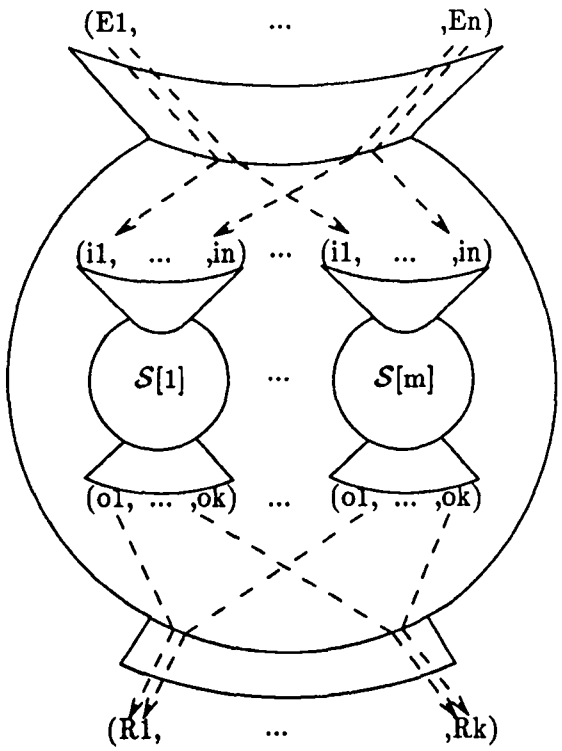
```
(i1: SEQ OF IN1,.., in: SEQ OF INn)(o1: SEQ OF OUT1,.., ok: SEQ OF OUTk);
```

La signature ci-avant est obtenue en remplaçant le type de chaque paramètre et de chaque résultat de la signature de «meth» dans la vue «T» par une séquence de ce type. Cette *vectorisation* correspond à la nécessité de fournir les paramètres aux différents composants de la multiprocédure associée à la pseudo-méthode «meth».

Similairement, la classe réalisant le type «S» est étendue par l'ajout de la multiprocédure «meth*» associée à cette nouvelle pseudo-méthode. Cette multiprocédure à pour composants les procédures associées aux méthodes «meth» des objets appartenant à la séquence appelée[6]. Soit $\mathcal{S}$ une séquence de type «S» ; l'expression «$\mathcal{S}$!meth(E1,..,En)» dénote l'appel de la multiprocédure «meth*» qui se déroule comme suit : la méthode «meth» de l'objet «$\mathcal{S}$[i]» est appelée avec les paramètres «E1[i]» à «En[i]». Lorsque l'exécution de toutes les méthodes est terminée, les résultats «o1[i]» à «ok[i]» prennent les valeurs délivrées par la méthode «meth» de l'objet «$\mathcal{S}$[i]». Le schéma de contrôle et le passage de paramètres que nous venons de décrire peuvent être représentés graphiquement comme suit :

[6]Nous n'appelons pas «meth*» une *multi-méthode* car cette appellation a déjà une autre acception. Dans les langages tels que CLOS [54], une multi-méthode est une méthode dont la procédure associée n'est pas uniquement déterminée par l'objet adressé (premier paramètre), mais par la totalité de ses paramètres.

Représentation de « (R1,..Rk) := $\mathcal{S}$!meth(E1,..,En) »



Il est intéressant de remarquer que le parallélisme découlant d'un multi-appel est *induit* par la structure des données (objets). La séquence munie du mécanisme de multi-appel permet de reconstituer l'objet à fragments identiques introduit par le langage POLYGOTH. Il est rapporté dans [1] que ces objets s'avèrent les plus utiles dans l'écriture d'applications réparties.

La composition de la séquence est fixée au moment de la reception du multi-appel. La politique de sérialisation « n-lecteurs un-rédacteur » appliquée aux séquences permet de garantir cette propriété. Les expressions « E1 » à « En » doivent être des séquences de même taille que $\mathcal{S}$, chacune formée d'éléments du type indiqué par le paramètre formel correspondant. Seule la vérification portant sur l'exactitude du nombre de paramètres fournis reste à la charge de la machine d'exécution. La vérification portant sur l'adéquation des types est effectuée à la compilation. Bien que fortement déconseillé, nous n'interdisons pas le partage de séquence d'objets. Il se peut qu'un multi-appel soit refusé par la machine d'exécution de ARCHE, comme par exemple, dans le cas suivant :

Exemple IV.7

Soient les déclarations,

```
TYPE
 V = VIEW ... m:(p:T)();...END;
(* m prend en argument un objet de type T *)
...
VAR s : SEQ OF V;
    p1,p2,p3 : T; (* trois objets de type T *)
...
```

alors, la commande suivante peut conduire au refus du multi-appel.

```
IF b THEN s := < o1, o2, o3 > ELSE s := < o1, o2 > END;
s!m(<p1,p2,p3>);          (* refus si NOT b *)
```

Un multi-appel est *passant* lorsque la méthode appelée est passante pour tous les objets de la séquence concernée, et ce de façon simultanée. Rappelons qu'une méthode est passante si elle figure dans l'état de synchronisation courant de l'objet adressé. L'appel n'est exécuté que si tous les objets de la séquence appelée prennent l'engagement d'y participer au même temps. Cette propriété induit d'importantes contraintes de réalisation dans un système distribué. Nous illustrons cette difficulté, propre à tout mécanisme de rendez-vous multiple, en présentant un scénario conflictuel.

Exemple IV.8

Soient les déclarations de l'encadré IV.7 ci-avant, altérées comme suit :

```
TYPE W = V VIEW n:()(); ... END;    (* ajout de la methode n *)
...
VAR  o1,..,o5: W;
...
```

La simlutanéité du multi-appel doit garantir que toute exécution parallèle des deux commandes suivantes :

```
< o1, o2, o4 >!m(< p1, p2, p3 >) || < o2, o3, o4, o5 >!n;
```

provoque le même ordre d'exécution pour les méthodes `m` et `n` des objets communs aux séquences appelées (o2,o4).

De plus, nous demandons de toute mise en œuvre du langage ARCHE une forte équité, au sens de [55], dans l'ordonnancement des multi-appels : un multi-appel qui serait passant infiniment souvent doit être exécuté une infinité de fois.

Les différentes études de la synchronisation et du contrôle dans les systèmes répartis rapportées dans la littérature, notamment [56], sont à la source des algorithmes nécessaires à la mise en œuvre de cette propriété. Une réalisation efficace d'un cas particulier de ce mécanisme de coopération est rapportée dans [57, 58]. Notons toutefois que le mécanisme mis en œuvre concerne seulement le cas, plus facile à traiter, où les objets et les séquences sont connus statiquement.

Revenons à l'exemple IV.5. Le déplacement des différents polygones de la figure en parallèle, suivi de l'affichage de la figure déplacée, toujours en parallèle, est obtenu en exécutant les commandes suivantes :

IV.5...
```
VAR F: Figure; v: POS;
...
F!Move(<v,v,v,v>);
F!Draw
...
```

Meilleure que la première, cette solution n'est pas tout à fait satisfaisante. Nous sommes obligés de faire correspondre le nombre de vecteurs de translation, «v», passés en paramètres au nombre de polygones de la figure. Cette contrainte est fort gênante car elle va à l'encontre de la généralité du code écrit : si la figure venait à être modifiée dans le nombre de polygones utilisés, l'appel à la méthode de déplacement devrait être *édité* et *recompilé*. Nous fournissons une simplification d'écriture qui permet d'exprimer, de manière générale, la *diffusion* d'un paramètre. Reprenons la vue de l'exemple IV.6 et la séquence S de type «SEQ OF T». Le mode diffusion du mécanisme de passage de paramètre que nous définissons ci-après confère la même sémantique aux commandes suivantes :

IV.6...
```
R := S!meth(E1,..,ei,..,En)
et
FOR j:= 1 TO S!LENGTH() DO Ei[j] := ei END;
R := S!meth(E1,..,Ei,..,En)
```

Définition IV.6 *Une expression de type «Q» fournie à la place d'un paramètre formel de type «SEQ OF Q» dans un multi-appel dénote la diffusion de la valeur de cette expression à tous les composants de la multiprocédure correspondant à la méthode appelée.*

Nous pouvons à présent apporter une solution satisfaisante à l'exemple du déplacement et affichage d'une figure structurée comme suit :

IV.5...
```
VAR F: Figure; v: POS;
...
F!Move(v);
F!Draw
...
```

IV.5.1 Appel coordonné

L'appel coordonné (*cf.* § I.2) sert à réaliser le dual d'une diffusion : un seul appel est issu d'une séquence d'objets. Dans la syntaxe, ce mode est indiqué en préfixant l'appel par le mot-clef «COCALL». L'intérêt d'un tel appel repose sur le fait qu'il fournit une abstraction pour un schéma de coopération courant : un calcul (séquentiel ou parallèle) a besoin d'un certain nombre de données. Ces données sont calculées indépendamment par un groupe d'objets. L'appel coordonné permet à ce groupe de coopérer au lancement du calcul à effectuer (*cf.* § I).

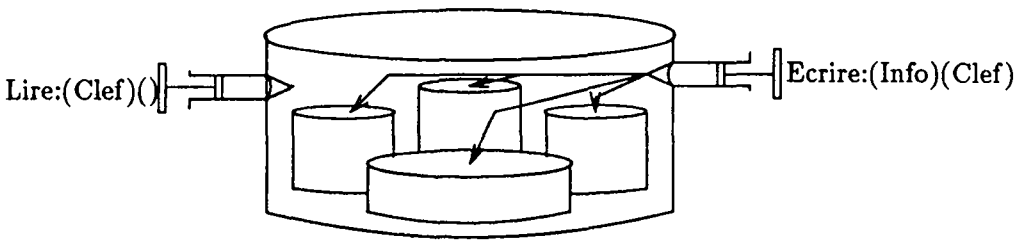
L'appel coordonné ne peut coordonner que les objets engagés dans une même multiprocédure. Ces objets appartiennent donc à une séquence, destinataire d'un multi-appel. Cette séquence est référencée par chaque objet dans la pseudo-variable «WE». La position occupée par chaque objet dans la séquence est référencée par la pseudo-variable «ME». Ainsi l'équation `WE[ME] = SELF` est vérifiée dans le corps de toute procédure. La sémantique de l'appel coordonné est fortement liée à l'objet mandataire et au destinataire :

- si l'objet mandataire ne participe pas à l'exécution d'une multiprocédure, l'appel coordonné se déroule comme un appel traditionnel. La vérification portant sur les paramètres doit être réalisée lors de l'appel ;
- si l'objet mandataire participe à l'exécution d'une multiprocédure, l'appel coordonné force la synchronisation de tous les composants. Si de plus, l'appel s'adresse à :
  - un objet destinataire commun à tout les composants, ou à une méthode du mandataire, les paramètres sont collectés en vérifiant leur *complétude*. Chaque participant à l'appel est libre de fournir un sous-ensemble quelconque des paramètres. Les valeurs fournies sont explicitement affectées aux paramètres formels de la pseudo-méthode appelée (voir exemple IV.9, page 70) ;

- un objet destinataire qui n'est pas le même pour tous les composants, ou à un ensemble de méthodes distinctes, alors une exception d'échec est levée (cf. § VI).

Nous ne pouvons pas faire une présentation exhaustive de ce mécanisme d'imbrication généralisée car il est toujours à l'étude [9]. Nous nous limitons à en montrer l'emploie dans un exemple suffisamment simple. Nous reprenons l'exemple d'un gestionnaire de disques répartis présenté dans [1].

Exemple IV.9



Considérons un ensembles de disques que nous voulons gérer comme s'il s'agissait d'un seul disque. Chaque disque est un objet typé par la vue «`Disque`» que nous détaillons ci-après.

IV.9...

```
Disque  = VIEW(Adm: Balance)
           Ecrire: (Inf:SEQ OF DATA)(Clef: KEY);
          OBSERVER
           Lire  : (Clef: KEY)(Inf: SEQ OF DATA);
           Libre : ()(Oct: CARDINAL);
          END;
Balance = VIEW
          OBSERVER
           Distribuer: (d: SEQ OF Dem)(a: SEQ OF Att);
          END;
Dem     = RECORD Cap,Tail: CARDINAL END;
Att     = SEQ OF CARDINAL;
```

Le paramètre de la vue est un objet destiné à équilibrer l'utilisation de l'espace libre des différents disques. La méthode «`Distribuer`» de cet objet prend en paramètre une séquence de *demandes* et rend une séquence d'*attributions*. Une demande est formée de la capacité courante du disque «`Cap`» et du nombre de données à stocker «`Tail`». Une attribution indique les éléments de la séquence «`Inf`» à ranger.

Un disque est muni de trois méthodes. La méthode «`Ecrire`» permet d'écrire sur le disque un ensemble d'informations typées. Elle rend une clef, nécessaire pour la lecture de ces données par la méthode observatrice «`Lire`». L'observatrice «`Libre`» permet d'obtenir l'espace libre du disque.

Dans ce qui suit, nous supposons que la pseudo-vue de l'encadré de la page 32 offre une opération supplémentaire. Cette opération, la concaténation distribuée («`DCAT`»), permet d'«écraser» une séquence de séquences pour obtenir une séquence simple. Sa signature est la suivante :

```
DCAT: (S: SEQ OF SEQ OF T)(s: SEQ OF T);      (* s <- S[1]^..^S!LAST *)
```

Nous présentons tout d'abords la classe, réalisant la vue «`Disque`», qui utilise une poule de disques.

IV.9...

```
CLASS PouleDisques IMPLEMENTS Exemple.Disque =
VAR
 DD : SEQ OF Disque; G: Balance := Adm;
PROCEDURE Coder  : (kk: SEQ OF KEY)(k: KEY) = ...
PROCEDURE Decoder: (k: KEY)(kk: SEQ OF KEY) = ...
...
```

L'état des objets de cette classe est composé d'une séquence de disques et d'un gestionnaire, désigné à la création d'exemplaire. Nous disposons aussi de deux procédures pour coder une séquence de clefs en une clef unique et vice versa.

Les procédures réalisant les différentes méthodes du disque utilisent le multi-appel introduit au paragraphe précédent.

IV.9...

```
...
PROCEDURE Ecrire: (Inf:SEQ OF DATA)(Clef: KEY) =
 VAR LesClefs: SEQ OF KEY;
BEGIN
 LesClefs := DD!Ecrire(Inf);
 RETURN Coder(LesClefs)
END Ecrire;
...
```

La procédure «`Ecrire`» diffuse l'information à écrire à tous les disques de la séquence et obtient une séquence de clefs. Les clefs sont codées en une clef unique, rendue en résultat.

IV.9...
```
...
PROCEDURE Lire: (Clef: KEY)(Inf: SEQ OF DATA) =
BEGIN
 RETURN (DD!Lire(Decoder(Clef))!DCAT)
END Lire;
...
```

La procédure «`Lire`» commence par décoder la clef fournie puis distribue une clef à chaque disque de la séquence. La correspondance entre les disques et les clefs est ici donnée par leur ordre respectif dans les deux séquences[7].

Nous supposons l'existence d'une procédure prédéfinie, «`SIGMA`», qui calcule la somme d'une séquence d'entiers.

IV.9...
```
...
PROCEDURE Libre: ()(Oct: CARDINAL) =
BEGIN
 RETURN (SIGMA(DD!Libre()))
END Libre;
...
```

L'espace libre du disque est tout simplement la somme des espaces libres des disques de la séquence.

La classe de chacun des disques de la poule réalise la même vue que la classe ci-avant décrite, «`PouleDisques`». Mais les disques dont est faite cette classe ne sont pas quelconques : ils appartiennent à une classe conçue pour leur permettre de coopérer dans le stockage des informations[8].

La procédure d'écriture de «`PouleDisques`» diffuse toute l'information à stocker. Cependant, nous ne voulons pas mettre en œuvre un mécanisme de réplication. Il faut que les différents disques s'accordent sur les parties de l'information à stocker par chacun. Nous montrons ci-après la procédure d'écriture de ces disques.

[7] Une classe permettant des modifications de la poule de disques aurait à gérer cette relation autrement, par exemple au moyen d'une table d'associations.

[8] Nous pouvons facilement modifier la classe «PouleDisques» pour qu'elle-même puisse faire partie d'une autre poule.

IV.9...

```
CLASS CoDisque IMPLEMENTS Exemple.Disque =
 VAR G: Balance := Adm;
...
PROCEDURE Ecrire: (Inf:SEQ OF DATA)(Clef: KEY) =
 VAR MaDem : Dem; MonAtt: Att;
BEGIN
 MaDem.Cap := Libre(); MaDem.Tail := Inf!LENGTH();
 MonAtt := COCALL G!Distribuer(d[ME]:= MaDem).a[ME];
 Inf!DOMRES(MonAttr);

                ...écrire «Inf» et calculer «Clef»...

 RETURN Clef
END Ecrire;
...
```

Chaque disque commence par calculer l'espace disponible et le nombre d'informations à stocker. L'appel «`G!Distribuer`», préfixé par le mot-clef «`COCALL`», synchronise tous les disques et demande au gestionnaire *un* calcul de quotas. La méthode «`Distribuer`» de la vue «`Balance`» prend pour paramètre une séquence de structures de type «`Dem`». La notation «`d[ME] := MaDem`» indique que cette séquence a la même taille que la séquence de disques appelée, et que la i-ème valeur est fournie par le i-ème exemplaire. Similairement, la notation «`.a[ME]`» indique que le i-ème élément de la séquence résultat est affecté à la variable «`MonAtt`» du i-ème exemplaire. L'information à stocker est alors déterminée. L'écriture de l'information s'en suit et la procédure termine en rendant la clef associée.

Rappelons que ce mode d'appel est toujours à l'étude, et qu'en conséquence, les explications données ici sont incomplètes, voire inconsistantes. Il nous semble tout de même intéressant de fournir l'appel coordonné dans le langage ARCHE car il permet d'exprimer une notion d'atomicité portant sur un ensemble d'objets. Dans l'exemple de la poule de disques, nous pouvons relever deux phases de calcul pour équilibrer la distribution des données. La première est un test qui conduit à déterminer un *état global* de la capacité *courante* des disques. La deuxième est une mise à jour des différents disques avec l'écriture effective des données. Nous n'avons fait aucune hypothèse sur l'environnement de la poule de disques dans la solution présentée. Or, nous pouvons supposer que les disques intégrant la poule sont référencés par d'autres objets dans l'environnement d'exécution. Ces disques peuvent alors être utilisés de façon autonome. Il devient évident que l'exécution des deux phases identifiées ci-avant doit être ininterrompue. Aucun appel à un disque de la séquence ne doit venir s'insérer entre les deux phases.

Le langage GUIDE propose un mécanisme pour exprimer la notion de tran-

saction. La notion de transaction s'applique à un ensemble d'opérations. Elle contraint leur exécution par une propriété plus forte que celle d'atomicité : elle exige un effet *de tout ou rien* sur l'ensemble des opérations. La notion d'atomicité est aussi présente dans le langage GUIDE dans un mécanisme attaché aux objets. L'interférence entre le mécanisme de synchronisation (*cf.* § IV.2, page 53), le mécanisme de transaction et la propriété d'atomicité des objets soulève de sérieux problèmes. Une discussion approfondie de ces conflits est rapportée dans [59].

Chapitre V

Sous-types et héritage

La notion d'héritage regroupe deux facilités. Considérant qu'une classe définit un type, cette notion permet d'exprimer la spécialisation d'une classe. Cette spécialisation fait référence à la relation *est-une* (*is-a*) ou encore relation de *sous-typage*. Du point de vue de la modularité, l'héritage permet d'étendre une réalisation. Cet aspect peut être référencé comme *héritage de réalisation*, ou *sous-classage*. Dans ce qui suit, nous définissons plus avant la relation de sous-typage et le mécanisme de sous-classage du langage ARCHE. Dans ces paragraphes, nous supposons connues les notions de sous-typages, d'héritage mais aussi de polymorphisme et de chaînage dynamique ; nous invitons le lecteur qui ne serait pas familier avec celles-ci à notamment consulter les références [27, 28, 60, 61, 62].

V.1 Sous-typage

Dans ce paragraphe, nous définissons la relation d'ordre de sous-typage, notée <:, du langage ARCHE puis nous examinons plus spécifiquement la déclaration d'une vue, sous-type. Enfin, nous définissons la relation d'affectabilité qui autorise des références polymorphes.

V.1.1 Relation de sous-typage

La relation de sous-typage est définie pour tous les types du langages et ne se limite pas aux types vues. Sa définition est inspirée de celle du langage MODULA-3 [63].

Le sous-typage pour les types ordinaux est tout à fait classique : tout intervalle d'entiers est sous-type des entiers, tout intervalle de valeurs d'un type

énuméré est sous-type de cette énumération, un intervalle est sous-type d'un second si ce dernier le contient complètement. Nous obtenons :

(i) soient deux entiers n et m,
nous avons $[n..m] <:$ INTEGER ;

(ii) soient deux éléments id_1 et id_2 d'une énumération E,
nous avons $[id_1..id_2] <: E$;

(iii) soient n, n', m, et m' des éléments d'un type ordinal T,
si $ord_T(n') \leq ord_T(n)$ et $ord_T(m) \leq ord_T(m')$
alors $[n..m] <: [n'..m']$.

Le constructeur d'ensemble SET OF est monotone pour le sous-typage. Nous obtenons :

(iv) soient deux types ordinaux T et T',
si $T <: T'$ alors SET OF $T <:$ SET OF T'.

Deux séquences ayant le même de type de base sont de même type. Par voie de conséquence, une séquence de type de base T n'est sous-type d'une séquence de type de base T' que si T est identique a T'.

Deux types structures sont en relation de sous-typage si l'un est une extension de l'autre. Cette extension peut être implicite ou explicite. L'extension de structure s'explicite en préfixant la définition de la nouvelle structure avec le nom de la structure étendue. Nous obtenons :

(vii) soient deux structures R et R',
si S est une structure telle que $R' = RS$ alors $R' <: R$.

Deux types vues sont en relation de sous-typage seulement si cela est explicitée au moyen de l'extension. L'extension de type vue s'explicite par préfixage comme pour les types structures. Les types ANY et NULL échappent à cette règle ; ANY est le plus petit type vue et NULL, le plus grand. Soit un type vue T_i, nous définissons :

($viii$) NULL $<: T_i$,

(ix) T_i VIEW .. END $<: T_i$, et

(x) $T_i <:$ ANY.

Enfin, deux types identiques sont sous-types l'un de l'autre :

(xi) soient deux types T et U,
si $T \equiv U$ alors $T <: U$ et $U <: T$.

Deux types *CURRENT* ne sont identiques que s'ils se rapportent à la même vue.

V.1.2 Spécialisation des vues

Nous avons vu dans le paragraphe précédent que deux types vues sont en relation de sous-typage si l'un est défini comme étant une extension de l'autre. Nous étudions plus avant cette facilité de spécialisation. Dans le langage ARCHE, il existe certaines contraintes liées à l'expression de la synchronisation : l'automate de synchronisation du super-type peut être invalidé par l'ajout éventuel de méthodes dans le sous-type. Nous reprenons l'exemple discuté dans [18] afin d'illustrer notre propos.

Exemple V.1

Considérons la vue *tampon* d'une bibliothèque L qui définit l'interface d'un tampon borné d'éléments de type T :

```
tampon  =  VIEW (n : INTEGER) { empty }
             put      : (i: T) () ;
             get      : () (i: T) ;
           STATE
             full     : { get } ;
             empty    : { put } ;
             partial  : { get, put } ;
           POST
             put : { partial, full } ;
             get : { partial, empty } ;
           END
```

Nous voulons ici définir un tampon, appelée *tampQueue*, qui permet le retrait du dernier élément d'un tampon. Ceci se fait naturellement en étendant le type *tampon* de la bibliothèque L avec la méthode *getRear* :

```
tampQueue   =  L.tampon VIEW ()
                 getRear : () (i: T) ;
               POST
                 getRear : get ;
               END
```

Nous pouvons remarquer que la méthode *getRear* ne sera jamais exécutée : aucun état de synchronisation n'y donne accès.

Nous définissons trois règles qui permettent d'établir les contraintes de synchronisation d'un type vue, sous-type, et de les reporter dans le super-type. Ces règles sont fortement liées à la spécialisation inhérente au sous-typage. Un objet dont le type est un sous-type doit pouvoir se comporter comme un objet du super-type. Considérons un objet $\mathcal{O}$, exemplaire d'une classe réalisant un sous-type d'un type T. La synchronisation du sous-type doit permettre d'assurer que $\mathcal{O}$, puisse au moins avoir un comportement commun avec celui que les objets de type T peuvent exhiber. Trois règles sont offertes pour établir les contraintes de synchronisation du type vue, sous-type :

(R_1) les états existants peuvent être redéfinis mais cette redéfinition ne peut se faire que par extension ;

(R_2) de nouveaux états peuvent être définis mais ils sont nécessairement définis par extension d'un état du super-type ;

(R_3) les post-états peuvent être étendus.

La règle R_1 répond aux exigences de la redéfinition d'une méthode dans une sous-classe et les règles R_2 et R_3 à celles de la définition de nouvelles méthodes dans un sous-type. Reprenons l'exemple V.1 pour illustrer l'utilisation de la règle R_1.

Exemple V.2 ———————————————————————

Le type *tampQueue* devient :

```
tampQueue = L.tampon VIEW ()
              getRear : () (i: T) ;
            STATE
```

```
    full     : { getRear } ;
    partial  : { getRear } ;
  POST
    getRear : get
  END
```

Les états *full* et *partial* sont enrichis avec la méthode *getRear*. Le post-état initial de *tampQueue* est celui de son super-type, aucun post-état ne figure dans l'entête de sa déclaration. Lorsqu'un post-état y figure, le post-état initial est l'union de celui-ci avec celui du super-type.

Nous illustrons à présent, au moyen d'un exemple, l'utilisation des règles R_2 et R_3 qui ont trait à la définition de nouveaux états.

Exemple V.3

Nous considérons le type *tampInverseur*, sous type de *L.tampon*, où lorsqu'un objet de type *tampInverseur* comprend au moins deux éléments, il est possible d'intervertir les deux premiers. Pour indiquer que l'objet est dans un tel état, le type *tampInverseur* doit être enrichi d'un nouvel état de synchronisation. Nous définissons :

```
tampInverseur = L.tampon VIEW
                  swap : () () ;
                STATE
                  atLeastTwo : partial : { swap } ;
                POST
                  put  : { atLeastTwo } ;
                  get  : { atLeastTwo } ;
                  swap : { atLeastTwo } ;
                END
```

L'automate correspondant à la synchronisation de cette vue est donnée en figure V.1. L'état *atLeastTwo* est une extension de l'état *partial* avec la méthode *swap* (règle R_2) ; cet état est ajouté aux post-états des méthodes *put* et *get* dans la clause *post* (règle R_3).

Notons que nous ne tenons pas compte des exceptions dans ces définitions. Les modifications à apporter sont présentées dans le chapitre VI qui décrit le traitement d'exceptions dans le langage ARCHE.

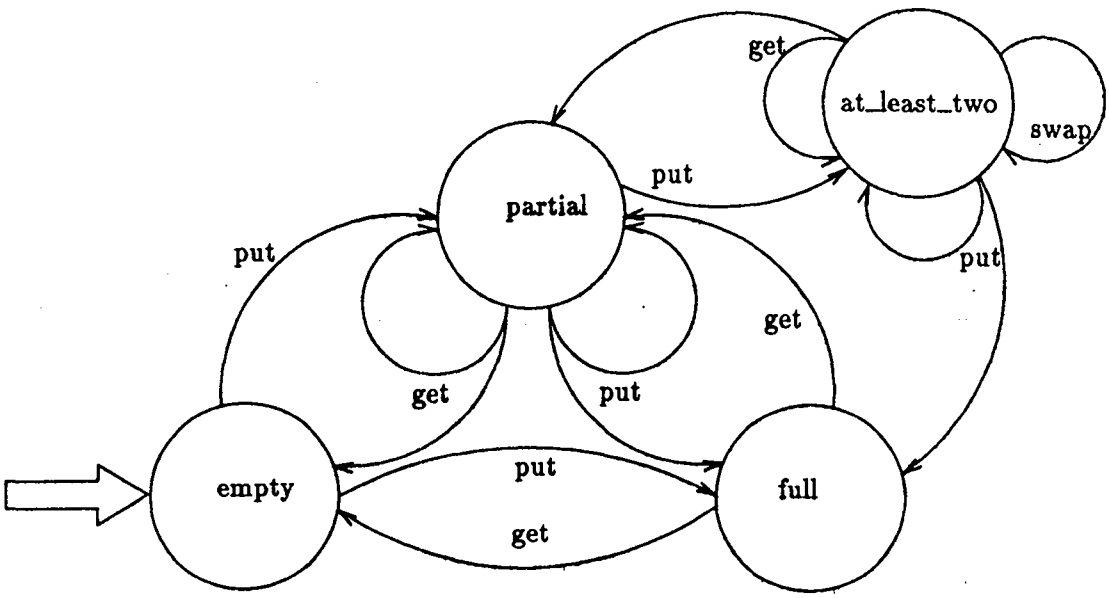


Figure V.1 Synchronisation de la classe tampInverseur.

V.1.3 Relation d'affectabilité

Nous introduisons la relation d'*affectabilité*, notée $\hookrightarrow$, qui autorise la définition de références polymorphes. Cette relation est définie entre les types et entre les expressions et les variables. Nous adoptons une définition similaire à celle décrite dans [63], pour le langage MODULA-3. Nous définissons :

(*i*) soient deux types T et U, si $T <: U$ alors $T \hookrightarrow U$;

(*ii*) soient deux types *ordinaux* T et U, si $T \cap U \neq \emptyset$ alors $T \hookrightarrow U$ et $U \hookrightarrow T$;

(*iii*) soient une expression e de type T et une variable v de type U, si $T \hookrightarrow U$ alors $e \hookrightarrow v$.

La règle (*i*) établit que tout type est affectable à ses super-types. De par notre définition de la relation de sous-typage, nous sommes en mesure de garantir que tout élément d'un type appartient aussi à ses super-types.

La règle (*ii*) pose que deux types ordinaux ayant au moins un élément en commun sont affectables. Cette règle exige une validation par la machine d'exé-

cution. La valeur concernée par la commande d'affectation doit appartenir aux deux types ce qui ne peut être contrôlé qu'à l'exécution.

Enfin, la règle (*iii*), établit qu'une expression e est affectable à une variable v si le type de e est affectable à celui de v.

V.2 Sous-classage

Nous précisons l'expression du *sous-classage* dans le langage ARCHE. L'héritage d'une classe c est exprimée dans l'en-tête de la classe descendante au moyen de la clause :

INHERITS c

L'héritage est *simple*. Concernant l'héritage multiple, nous estimons que cette facilité mérite des études plus approfondies avant d'être introduite dans un langage. Notamment, en ce qui nous concerne, il serait nécessaire de fournir une définition précise du sous-typage multiple.

L'héritage n'est *pas contraint* : une classe a visibilité de l'ensemble des déclarations de son ancêtre. Nous pensons que cette facilité répond mieux au critère d'*extensibilité* du modèle à objets.

Une classe ne peut hériter d'une autre que si son type est sous-type de celui de l'ancêtre. Nous explicitons brièvement les motivations qui ont guidé ce choix. La définition de la synchronisation dans le type ne permet pas de définir simplement un descendant indépendamment de la relation de sous-typage. Considérons une classe D qui hérite d'une classe C mais dont les types vues associés ne sont pas en relation de sous-typage. La classe D n'est pas une spécialisation de C et n'a donc pas à être contrainte par la synchronisation de C. Ceci implique de pouvoir modifier la valeur des états de synchronisation utilisées dans C. Considérons maintenant une méthode r de C. L'utilisation de la réalisation de r comme une observatrice dans D est envisageable. Cependant, cela implique une complète analyse de la classe C et amoindrit la facilité de réutilisation.

La *redéfinition de procédures est explicite*. La liste des procédures redéfinies par un descendant figure dans l'en-tête de la classe dans la clause suivante :

REDEFINES *liste des noms d'opérations redéfinies*

Il est aussi possible de renommer des procédures de l'ancêtre. Ce renommage s'explicite au moyen de la clause suivante :

RENAMES (*ancien nom* AS *nouveau nom* ;)$^+$

Le fait de forcer une redéfinition explicite permet de guider le programmeur. Si nous considérons la réalisation d'un descendant d'une hiérarchie de grande taille, il devient vite facile d'omettre qu'une procédure a déjà été déclarée[1]. Le deuxième point est lié au renommage. Nous pouvons admettre que bien qu'une procédure soit redéfinie, on veuille que sa réalisation d'origine soit accessible par les utilisateurs du descendant. Pour cela, il est nécessaire que la signature de la procédure (renommée) apparaisse dans le type du descendant.

Lorsqu'une procédure est redéfinie, l'ancienne définition est toujours accessible depuis la nouvelle au moyen du mot clé SUPER. Comme pour toute appel d'opération, l'appel à SUPER requiert d'expliciter les paramètres effectifs de l'opération appelée. Cette facilité est aussi fournie pour les méthodes d'initialisation. L'appel à SUPER au sein d'une telle opération se traduit par l'appel à la méthode d'initialisation de la super-classe.

Les procédures peuvent être *localement redéfinies*. Ceci est explicité au moyen des mots clés REDEFINES LOCALLY. La redéfinition locale d'une opération f au sein d'une classe A est ignorée lors de l'exécution d'opérations définies dans les super-classes de A qui appellent f.

Afin d'illustrer l'expression du sous-classage, nous concluons ce paragraphe par la présentation d'une réalisation du type *tampInverseur*, défini dans l'exemple V.3, page 77.

Exemple V.4

Nous proposons tout d'abord une réalisation du type vue *tampon* où nous supposons que le type T est déclaré dans la bibliothèque S

```
CLASS cTampon IMPLEMENTS L.tampon =
   FROM S USES T ;
   CONST
     max = n ;
   VAR
     in, out : INTEGER ;
     tamp : SEQ [0..max-1] OF T ;
   PROCEDURE
     put =
```

[1] A noter que cet aspect de la programmation à objets est d'autant mieux traité qu'un environnement de programmation adéquat est fourni.

```
      BEGIN
        tamp[in] := i ;
        in := (in + 1) MOD max ;
        IF in = (out + 1) MOD max
           THEN
             BECOME full
           ELSE
             BECOME partial
        END
      END put ;
    get =
      BEGIN
        i := tamp[out] ;
        out := (out + 1) MOD max ;
        IF in = out
           THEN
             BECOME empty
           ELSE
             BECOME partial
        END ;
        RETURN(i)
      END get ;
    BEGIN
      in := 0 ;
      out := 0 ;
      BECOME empty
END cTampon
```

Supposons que le type *tampInverseur* soit déclaré dans la bibliothèque *M*. Une réalisation possible de *tampInverseur* est :

```
CLASS cInverseur IMPLEMENTS M.tampInverseur =
  INHERITS
    cTampon  ;
  REDEFINES
    put, get ;
  PROCEDURE
    put =
      BEGIN
        SUPER(i)
        IF ((in-out) > 1) OR ((out-in) > 1)
```

```
            THEN BECOME atLeastTwo
        END
      END put ;
    get =
      BEGIN
        i := SUPER() ;
        IF ((in-out) > 1) OR ((out-in) > 1)
           THEN BECOME atLeastTwo
        END ;
        RETURN(i)
      END get ;
    swap  =
      VAR
        x : T ;
      BEGIN
        x := tamp[in] ;
        tamp[in] := tamp[(in+1) MOD max] ;
        tamp[(in+1) MOD max] := x
      END swap ;
    BEGIN
      SUPER() ;
END cInverseur
```

V.3 Discussion

Dans ce chapitre, nous avons précisé la notion d'héritage dans le langage ARCHE. Nous avons choisi, tout comme dans [61], de distinguer le sous-typage du sous-classage.

Concernant le sous-typage, nous avons repris la relation définie pour le langage MODULA-3 [63]. Nous rappelons plus spécifiquement la relation de sous-typage entre les types vues qui est explicite. Deux types vues sont en relation de sous-typage si l'un est définie comme étant une extension de l'autre. Afin de concilier sous-typage et synchronisation, nous avons été amenés à imposer certaines contraintes sur cette extension. En effet, le mécanisme de synchronisation que nous avons retenu est, comme notamment suggéré dans [18], bien adapté à l'héritage. Cependant, il doit être contraint pour satisfaire la spécialisation, première caractéristique du sous-typage. Notons que cet aspect n'est pas traité dans [18], seul l'héritage comme mécanisme de réutilisation y est considéré.

Nous avons défini trois règles pour établir la synchronisation d'un sous-type à partir de celle de son super-type. Cette définition nous permet d'assurer qu'un objet, exemplaire d'une classe réalisant un sous-type d'un type T, peut avoir un comportement commun avec celui d'un objet, exemplaire d'une classe réalisant T. Cette définition de la relation de sous-typage est complétée par la relation d'affectabilité qui permet d'expliciter l'utilisation d'une entité d'un sous-type pour une entité du super-type.

Dans un deuxième temps, nous avons défini le mécanisme de sous-classage du langage ARCHE. Le peu de travaux formels sur l'héritage multiple nous ont menés à ne retenir qu'un héritage simple. Seules les classes réalisant un sous-type peuvent hériter d'une classe. Cette dernière restriction est due aux interférences avec le mécanisme de synchronisation.

Nous concluons ce chapitre, par l'expression de la généricité. Nous n'offrons pas un tel mécanisme dans ARCHE. Cependant, la présence d'une racine unique, ANY, permet, associée à la règle d'affectabilité, d'inclure une certaine forme de généricité. A ce propos, nous pouvons mentionner l'étude de [28], chapitre 19, qui montre que la généricité et l'héritage sont deux techniques complémentaires. Il y est montré que l'héritage ne peut être exprimé au moyen de la généricité. En revanche, l'héritage permet de simuler la généricité même s'il y est montré que la méthode utilisée est alors parfois trop artificielle voire trop lourde.

Chapitre VI

Traitement d'exceptions

Nous considérons ici le traitement des exceptions dans le langage ARCHE. Nous supposons connue la terminologie de base relative au traitement des exceptions ; pour plus de détail, nous invitons le lecteur à notamment consulter les références [64] et [65]. Ce chapitre se décompose comme suit. Dans un premier temps, nous présentons le modèle retenu pour le traitement des exceptions. Ensuite, nous précisons l'expression de ce modèle. Enfin, nous concluons dans le paragraphe VI.3.

VI.1 Modèle coopération de traitement des exceptions

Dans ce paragraphe, nous décrivons succinctement le modèle retenu pour le traitement des exceptions dans le langage ARCHE. Une définition plus précise de ce modèle, appelé *modèle coopération*, peut notamment être trouvée dans [66]. Le modèle coopération privilégie essentiellement la simplicité. Il est par conséquent fondé sur une extension du *modèle terminaison*, défini pour le traitement des exceptions dans les programmes séquentiels.

Plus précisément, le modèle terminaison retenue est celui exprimé par le mécanisme de traitement d'exceptions (MTE) du langage CLU [67]. Nous n'argumentons pas ici de la simplicité du modèle terminaison comparé aux autres modèles existants, nous renvoyons le lecteur intéressé à la discussion apparaissant dans [67]. Nous définissons brièvement le modèle terminaison qui définit le schéma de contrôle résultant du signal d'une exception dans un programme séquentiel. Dans ce modèle, l'opération au sein de laquelle est levée l'exception termine immédiatement et le contrôle est transféré à l'appelant de l'opération en vue du traitement de l'exception. Nous précisons les extensions apportées au modèle terminaison, qui sont dues à la présence de parallélisme.

L'occurence d'une exception au sein d'un processus peut interférer avec le comportement des autres processus. Considérons deux processus communiquant de manière synchrone. Supposons en outre que l'un d'eux termine exceptionnellement avant d'exécuter cette communication. Il s'ensuit que l'autre processus se trouve bloqué, en attente d'une communication qui ne surviendra pas.

VI.1.1 Traitement des exceptions globales

Dans le modèle coopération, il est possible de signifier la non participation d'un processus à au moins une communication synchrone en signalant une exception dite *globale* qui est *connue* des autre processus. Se fondant sur un modèle terminaison, le signal d'une exception globale entraîne la terminaison du processus qui la signale.

La notion de *capture d'exception globale* est introduite afin d'éviter l'interblocage mentionnée précédemment. De manière générale, un processus capture une exception globale e lorsqu'il communique de manière synchrone avec un processus qui a terminé exceptionnellement suite au signal de l'exception e. La capture d'une exception globale par un processus est comparable à la propagation de l'exception à ce processus. Le traitant de l'exception globale capturée est alors recherché dans le contexte courant de traitement d'exceptions.

Remarquons ici que l'introduction d'exceptions globales a des répercussions sur la déclaration des processus : tout processus déclare les exceptions globales qu'il signale.

VI.1.2 Traitement des exceptions concertées

Nous décrivons ici les caractéristiques du modèle coopération qui sont relatives à la présence de blocs parallèles imbriquées. Cette dernière facilité se retrouve notamment dans le langage ARCHE avec la notion de multi-procédure.

Lorsque les processus sont des composants d'une opération parallèle imbriquée, la terminaison exceptionnelle d'au moins l'un d'eux entraîne la terminaison exceptionnelle de l'opération. L'exception signalée par l'opération doit néanmoins être précisé puisque plusieurs processus, composants de l'opération, peuvent concurremment signaler une exception.

Comme cela a notamment été présenté dans [68], l'occurence de plusieurs exceptions peut être indicatrice d'un état exceptionnel dépendant de la composition de toutes les exceptions signalées. Nous qualifions de *concertée*, l'exception résultante. Le calcul d'une exception concertée ne peut généralement être défini

implicitement : il nécessite une connaissance sémantique des diverses exceptions. Un mécanisme spécifique est par conséquent inclus dans la définition du MTE, nous présentons un tel mécanisme dans le paragraphe VI.2.2.2. Nous introduisons ici une solution implicite. Toutefois, une évaluation sémantique de l'exception n'étant pas possible, la définition proposée n'est applicable que lorsque le programmeur ne désire pas une caractérisation précise des divers domaines exceptionnels.

Définition VI.1 *La valeur par défaut d'une* exception concertée *est :*

(i) *l'exception prédéfinie indicatrice d'un échec, par exemple de nom* FAILURE, *si au moins deux de ses composants ont signalé, soit une exception différente, soit une exception avec des paramètres associés ;*

(ii) *l'exception e si tous les composants signalants, ont signalé l'exception e et que e n'a pas de paramètre associé.*

Dans les programmes séquentiels, un traitant d'exception est associé à une opération séquentielle et a le même contexte. Dans le modèle coopération, cette facilité est étendue à un contexte réparti. Les *traitants associés à une opération parallèle* ont même degré de parallélisme que l'opération et traitent les exceptions concertées que l'opération signale. La déclaration d'un tel traitant est facilement réalisée : il suffit d'associer chaque composant du traitant au composant de l'opération parallèle auquel il est associé. Notons ici que cette définition suppose une déclaration statique de l'opération parallèle. Les composantes d'une multi-procédure n'étant connues, dans le langage ARCHE, qu'à l'exécution, la solution retenue est ici quelque peu différente. Nous précisons cet aspect dans le paragraphe VI.2.1.4.

Le calcul d'exceptions concertées est aussi nécessaire en présence d'un modèle de communication synchrone qui autorise la participation de plus de deux composants à une même communication, par exemple en présence de *rendez-vous généralisés* [69]. Dans le langage ARCHE, nous retrouvons cette facilité avec la notion d'appel coordonné. A titre d'exemple, considérons un appel coordonné de multi-procédure. Si au moins une des composantes de la multi-procédure appelante signale une exception globale alors les composantes participant *effectivement* à l'appel capturent une voire plusieurs exceptions globales. Une exception concertée est ensuite évaluée par chacune des composantes appelantes.

Nous concluons cette présentation par la définition du modèle coopération en présence d'appel asynchrone d'opérations, ce mécanisme se retrouvant dans le langage ARCHE lors de la création d'un exemplaire de classe. Une exception

signalée par une opération, appelée de manière asynchrone, est propagée *par défaut* à l'appelant. Il est toutefois possible de circonvenir cette définition en explicitant les processus devant traiter l'exception lors de l'appel.

VI.2 Expression du traitement des exceptions

Nous définissons ici le MTE du langage ARCHE qui exprime le modèle coopération. Dans un premier temps, nous énumérons les caractéristiques de base de ce MTE.

La propagation des exceptions est *explicite* : lorsqu'une exception n'est pas traitée dans le contexte courant, une exception prédéfinie de nom FAILURE est propagée. Le fait que la propagation des exceptions soit explicite maintient la cohérence du principe d'abstraction. Elle permet en outre de déterminer *statiquement* le traitant de tout exception, autorisant ainsi des vérifications supplémentaires lors de la compilation.

Des *paramètres peuvent être associés aux exceptions.* Cette facilité permet de déterminer les circonstances de l'appel de tout traitant sans recourir à l'utilisation de variables globales.

Enfin, le *traitement des exceptions par défaut* est offert. Un traitant peut alors être déclaré pour toute exception n'ayant pas un traitant spécifiquement prévu. Cette caractéristique est importante car elle permet d'exprimer le traitement des exceptions non anticipées.

Les paragraphes suivant précisent les constructions liguistiques du MTE. Dans le paragraphe VI.2.1, nous étudions la définition des exceptions, l'expression du traitement des exceptions faisant l'objet du paragraphe VI.2.2. Enfin, les opérations dont l'exécution peut entraîner le signal d'une exception sont présentées dans le paragraphe VI.2.3.

VI.2.1 Définition des exceptions

Dans ce paragraphe, nous nous intéressons à la notion d'exception telle qu'elle est offerte dans le langage ARCHE. Nous étudions tout d'abord la déclaration des exceptions. Ensuite, nous examinons l'utilisation des exceptions dans les types vues puis la définition de la relation de sous-typage en présence d'exceptions. Enfin, nous présentons les exceptions prédéfinies du langage.

VI.2.1.1 Représentation et définition des exceptions

Dans un modèle de programmation à objets, la représentation des exceptions comme des classes est essentielle pour maintenir la facilité de réutilisation offerte par l'héritage. Elle permet de spécialiser des exceptions tout en n'ayant pas à systématiquement récrire les opérations concernées. Par exemple, un traitant dont la seule fonction consiste à propager une exception n'a pas à être modifié. Une telle représentation des exceptions est en outre souhaitable du point de vue de l'orthogonalité du langage de programmation.

La définition des exceptions au moyen de classes a notamment été retenue dans le langage à objets fortement typé C++ [70]. Le signal d'une exception se traduit par la création d'un exemplaire de la classe la définissant. L'identification d'une exception repose sur la reconnaissance de la classe dont elle est exemplaire. Les classes étant ici des types, cette détection est comparable à tester le type dynamique d'une référence.

Dans le langage ARCHE, les *exceptions sont des classes particulières.* La notion de *métaclasse* n'étant toutefois pas offerte dans ce langage, nous introduisons une construction linguistique, EXCEPTION, spécifiquement destinée à la déclaration d'une classe caractérisant une exception. Contrairement aux classes ARCHE précédemment introduites, excepté celles réalisant une séquence, une classe définissant une exception ne se décompose pas en la déclaration d'une vue et d'une réalisation. Les seules informations associées à une exception étant ses paramètres, la déclaration d'une réalisation n'est pas nécessaire. La déclaration d'une exception est comparable à celle d'un type structure. Par exemple, une exception e ayant n paramètres a_i de type t_i, $1 \leq i \leq n$ se déclare :

$$\mathrm{e} = \text{EXCEPTION } \mathrm{a_1} : \mathrm{t_1}, \ldots, \mathrm{a_n} : \mathrm{t_n} \text{ END.}$$

Concernant l'équivalence définie sur le type *exception*, deux types, exceptions, sont équivalents si ils sont *égaux.*

Un exemplaire d'une classe *exception* est créé au moyen de la commande de signal d'exception qui est définie ultérieurement dans le paragraphe VI.2.3.1. L'accès au paramètre a_i d'un objet $\mathcal{O}_e$, exemplaire de e, s'exprime au moyen de $\mathcal{O}_e.a_i$. Enfin une exception e' qui n'admet pas de paramètre se déclare : $e' =$ EXCEPTION.

La déclaration d'une exception sous-type s'explicite par extension, comme pour le sous-typage des vues. Par exemple, une exception f, sous-type de e admettant l paramètres supplémentaires b_i de type u_i, $1 \leq i \leq l$ se déclare :

$$\mathrm{f} = \mathrm{e} \text{ EXCEPTION } \mathrm{b_1} : \mathrm{u_1}, \ldots, \mathrm{b_l} : \mathrm{u_l} \text{ END.}$$

VI.2.1.2 Utilisation des exceptions dans les types vues

Un type vue définit des signatures d'opérations. Les types des exceptions signalées par une opération figurent dans la clause SIGNALS de sa signature. Les exceptions nommées doivent toutes être déclarées. Notons ici que lorsqu'une opération s'exécute en tant que composante d'une multi-procédure, ces exceptions sont globales par rapport à l'exécution de la multi-procédure.

La signature de la méthode d'initialisation est donnée dans l'en-tête du type. Les exceptions signalées par cette méthode figurent dans sa signature après le mot clé SIGNALS.

Enfin, les exceptions d'initialisation traitées par tout exemplaire d'une réalisation d'une vue $\mathcal{V}$ sont aussi déclarées dans l'en-tête de la vue après le mot clé HANDLES. Cette dernière déclaration est nécessaire pour statiquement vérifier la correction de l'appel de l'opérateur NEW, qui est un appel asynchrone d'opération (cf. § VI.1.2).

Nous présentons un exemple de déclaration de vue avec exceptions.

Exemple VI.1

Nous considérons à nouveau la vue *tampon* de l'exemple V.1. Les méthodes d'initialisation et *get* peuvent signaler l'exception *vide* et la méthode *put* peut signaler l'exception *plein* :

```
LIBRARY L =
  ... ;
  plein =
    EXCEPTION ;
  vide =
    EXCEPTION ;
  tampon =
    VIEW (n : INTEGER) { empty }  SIGNALS vide ;
      put : (x : T) () SIGNALS plein ;
      get : () (x : T) SIGNALS vide ;
    STATE
      ... ;
    POST
     ... ;
    END ;
END L
```

Considérons à présent une vue *utilTamp* dont toute réalisation crée un exemplaire d'une classe réalisant *L.tampon*. L'exception *vide* est signalée par la méthode d'initialisation si un exemplaire d'une réalisation de ce type a été créé avec une valeur de n inférieure à 1. Il est possible de renforcer le traitement de l'exception d'initialisation *vide* par toute réalisation de cette vue en l'explicitant dans l'entête de *utilTamp*. Nous obtenons :

```
LIBRARY N =
  ... ;
  FROM L USES vide ;
  utilTamp =
    VIEW () HANDLES vide ;
      ... ;
    END ;
END N
```

VI.2.1.3 Relation de sous-typage

Considérons la définition de la relation de sous-typage du langage ARCHE. La signature des méthodes et observatrices d'un super-type est inchangée dans la vue sous-type. Il s'ensuit qu'une opération d'un sous-type peut *au plus* signaler les exceptions de l'opération correspondante dans la définition de son super-type. Nous pensons que cette solution ne répond pas suffisamment à la notion de spécialisation inhérente à la relation de sous-typage. Considérons les deux relations de sous-typage suivantes :

MoyenDeTransportTerrestre <: MoyenDeTransport
MoyenDeTransportAérien <: MoyenDeTransport

Supposons de plus l'existence d'une opération *contrôleMoteur* dans la définition du type *MoyenDeTransport*. Cette opération peut typiquement signaler l'exception *PanneMoteur*. Si l'on étudie à présent le sous-type *MoyenDeTransportAérien*, le domaine exceptionnel associé à *PanneMoteur* peut être divisé, ou spécialisé, en trois sous-domaines : *PanneMoteurGauche*, *PanneMoteurDroit* et *PanneBiMoteur*. Ces trois exceptions sont des spécialisations de *PanneMoteur* ou encore sont en relation de sous-typage avec *PanneMoteur*. Cette relation s'établit aisément lorsque les exceptions sont déclarées comme des classes. Afin d'expliciter la spécialisation des exceptions signalées par une opération m, déclarée par un type U et son sous-type T, nous introduisons la définition suivante

qui établit une relation entre la liste des exceptions signalées par m dans T et celles signalées par m dans U :

Définition VI.2 *Toute exception apparaissant dans la signature de l'opération* m *dans* T *doit être sous-type d'une exception apparaissant dans la signature de l'opération* m *dans* U.

La spécialisation des exceptions prises en compte dans le sous-type peut être explicitée dans la définition de ce dernier. Nous avons toutefois limité cette facilité aux opérations autres que celle d'initialisation. Nous n'avons en effet pas vu l'intérêt d'une telle spécialisation pour ces dernières et avons retenu la solution présentée pour le langage MODULA-3 [63], c'est à dire, la liste des exceptions signalées reste inchangée.

Syntaxiquement, la nouvelle liste des exceptions signalées par une opération spécialisée s'exprime au moyen de la clause EXCEPTION dans toute déclaration d'une vue. Cette clause est de la forme :

$$\text{EXCEPTION } (e_j \text{ OF } op_i : \{ \text{ listes d'exceptions } \} ;)^+$$

où e_j désigne une exception signalée par l'opération op_i du super-type et où toute exception de la liste d'exceptions est une exception signalée par l'opération redéfinissant op_i, chacune de ces exceptions devant être *sous-type* de e_j.

Notons que le mécanisme offert pour exprimer la spécialisation des exceptions garantit la validité de la définition VI.2. Nous illustrons l'expression de la spécialisation d'une exception par un exemple.

Exemple VI.2

Nous considérons à nouveau le type *MoyenDeTransport* et sa spécialisation *MoyenDeTransportAérien*. Nous définissons la bibliothèque P :

```
LIBRARY P =
  ...
  PanneMoteur =
    EXCEPTION ;
  MoyenDeTransport =
    VIEW ()
      controleMoteur () () SIGNALS PanneMoteur ;
      ... ;
    END ;
END P
```

et la bibliothèque Q :

```
LIBRARY Q =
  ... ;
  FROM P USES MoyenDeTransport, PanneMoteur ;
  PanneMoteurGauche =
    PanneMoteur EXCEPTION ;
  PanneMoteurDroit =
    PanneMoteur EXCEPTION ;
  PanneBiMoteur =
    PanneMoteur EXCEPTION ;
  MoyenDeTransportAerien =
    MoyenDeTransport VIEW ()
    ... ;
    EXCEPTION
      PanneMoteur OF controleMoteur :
        { PanneMoteurGauche,
          PanneMoteurDroit,
          PanneBiMoteur
        };
    END ;
END Q
```

Enfin, Il est possible d'étendre la liste des exceptions d'initialisation traitées par un sous-type. Cette extension est précisée après le mot clé HANDLES dans l'en-tête de la vue.

VI.2.1.4 Exceptions d'interface

Nous définissons ici les exceptions prédéfinies du langage ARCHE. Elles sont appelées *exceptions d'interface* et ne sont signalées que par le support d'exécution du langage.

L'exception FAILURE propage une exception qui n'est pas traitée dans le contexte où elle est signalée.

Nous retrouvons les exceptions usuelles sur les types entiers et réels à savoir OVF indiquant un débordement par valeur supérieure et UDF, un débordement par valeur inférieure.

L'exception TIMEOUT est signalée lorsqu'une communication n'est pas survenue à l'expiration d'un intervalle de temps Δt fixé par le support d'exécution du langage. Cette exception peut être signalée lors de l'appel à une opération ou à une multi-procédure. L'occurence de cette exception peut indiquer : la panne du medium de communication ; la panne des processeurs où s'exécutent les processus correspondant aux objets à qui sont émis les messages ; ou encore un interblocage. Cette exception peut aussi être signalée lors d'un appel *coordonné* d'opération. Hormis les cas cités précédemment, l'exception peut alors être indicatrice d'une erreur de conception, notamment l'absence de participation à l'appel coordonné d'une, voire plusieurs composantes de la multi-procédure appelante.

L'exception UNCREATE est signalée lorsque la création d'un exemplaire d'une classe n'a pu être assurée par le support d'exécution.

Le modèle coopération permet d'associer des traitants d'exceptions à une opération parallèle (cf. § VI.1.2). Le rôle de ces traitants est de permettre la restauration d'état cohérent et éventuellement de masquer une exception concertée. Le caractère dynamique des multi-procédures interdit le masquage des exceptions concertées par l'appelée. La restauration d'un état cohérent par les divers composantes d'une multi-procédure est cependant toujours souhaitable. Nous introduisons pour cela la possibilité d'associer le traitant d'une exception particulière, appelée RESTORE, à une *méthode*[1]. Ce traitant a pour fonction de restaurer un état cohérent lorsqu'une méthode s'exécute en tant que composante d'une multi-procédure et que la multi-procédure termine exceptionnellement. Les diverses composantes déclarant un tel traitant sont synchronisées préalablement à son exécution. Enfin, les exceptions transmises à l'appelante sont celles signalées par les diverses composantes. Toutefois, une exception signalée lors de l'exécution du traitant de RESTORE, par une composante, se substitue à l'exception que la composante a pu signaler auparavant.

Nous définissons enfin deux *exceptions particulières*, TERMINATED et PRESENT, utilisées lors de l'exécution d'une multi-procédure pour le calcul d'une exception concertée. La première correspond à la terminaison standard d'une composante de la multi-procédure. La seconde est utilisée lors d'un appel coordonné où au moins une des composantes signale une exception globale, elle indique que la composante *signalant l'exception* PRESENT participe effectivement à l'appel. Ces deux exceptions sont systématiquement ajoutées dans le contexte de traitement d'exceptions courant et ne sont jamais traduites en l'exception FAILURE. Précisons enfin que ces deux exceptions sont écartées pour le calcul par défaut d'une exception concertée, qui est donné par la définition VI.1,

[1] L'association d'un tel traitant à une observatrice n'a pas de sens puisqu'une telle opération ne modifie pas, par définition, l'état de l'objet.

page 87.

VI.2.2 Réalisation du traitement des exceptions

Nous nous intéressons ici à l'expression de la réalisation du traitement des exceptions.

VI.2.2.1 Signature de procédure

Une signature de procédure fait état des exceptions que la procédure signale dans la clause SIGNALS. Lors de la redéfinition d'une procédure p, il est possible de modifier la liste des exceptions explicitées dans la signature de p tant que la définition VI.2 est respectée.

VI.2.2.2 Calcul explicite des exceptions concertées

Le mécanisme, fournit dans le langage ARCHE, pour le calcul explicite des exceptions concertées introduit la notion de *fonction de résolution.* Les déclarations de fonctions de résolution figurent dans la clause RESOLUTION des classes. La forme générale d'une fonction de résolution est :

$$r \text{ HANDLES } l_h \text{ SIGNALS } l_s = C \text{ END } r$$

où :

- r est le nom de la fonction de résolution ;
- C est une commande ;
- l_h détermine la liste des exceptions considérées dans C, ces exceptions peuvent donc être signalées par les composantes d'une multi-procédure pour laquelle la fonction de résolution utilisée est r :
- l_s détermine la liste à laquelle appartient nécessairement l'exception (concertée) signalée par r.

Le paramètre formel de toute fonction de résolution est implicite. Il s'agit d'une séquence d'exceptions de nom SEQEXC. Lors de l'exécution d'une telle fonction, son paramètre effectif a autant d'éléments que la multi-procédure signalante a de composantes et la valeur du $i^{ème}$ élément détermine l'exception

signalée par la $i^{ème}$ composante. Lorsqu'une composante termine normalement, c'est à dire ne signale pas d'exception, l'élément qui lui correspond dans SEQEXC a pour valeur TERMINATED. La prise en compte de la terminaison standard des composantes dans la fonction de résolution nous est en effet apparue utile lors de la programmation d'exemples.

Il est possible de redéfinir une fonction de résolution d'une classe dans ses sous-classes ; soit la redéfinition de r suivante :

$$r \text{ HANDLES } l'_h \text{ SIGNALS } l'_s = C' \text{ END } r ;$$

la condition de validité de cette redéfinition est inspirée de la règle de sous-typage des fonctions de [27]. L'en-tête de cette fonction doit satisfaire les deux conditions suivantes :

(i) $\forall$ e $\in l_h$, $\exists$ f $\in l'_h$ tel que e <: f (*contravariance*) ;

(ii) $\forall$ e $\in l'_s$, $\exists$ f $\in l_s$ tel que e <: f (*covariance*).

Indiquons qu'une fonction de résolution a accès, en lecture, aux variables d'état de la classe qui la déclare. Cette facilité autorise la prise en compte de l'état de l'objet pour la résolution des exceptions multiples.

Une commande spécifique permet d'identifier la valeur de l'exception transportée par tout élément d'une séquence d'exceptions. La forme générale de cette commande est :

EXCEPTION CASE expression OF $(e_i\ (v_i) : C_i)^+$ ELSE : C END

où :

- C_i et C sont des commandes du langage,
- e_i est un type exception, et
- v_i est une variable de type e_i, locale à C_i.

Nous illustrons la déclaration d'une fonction de résolution au moyen d'un exemple.

Exemple VI.3

Considérons la gestion d'une séquence de tampons bornés, éventuellement de taille différente, de type *L.tampon* (cf. exemple VI.1, page 90), telle que les ajouts et retraits d'éléments soient effectués sur tous les tampons, accessibles, de la séquence. Appelons cette séquence *seqTamp*.

La méthode *put* signale l'exception *plein* lorsqu'il n'y a plus de place dans le tampon une fois l'ajout de l'élément effectué. L'appel à la multi-procédure *put* de la séquence *seqTamp* peut conduire à des signaux concurrents de l'exception *plein*. Un type possible pour l'exception concertée résultante est *seqPlein* qui indique la séquence des tampons pleins de *seqTamp*. La déclaration du type *seqPlein* est :

```
seqPlein = EXCEPTION tamp : SEQ OF tampon END
```

Nous définissons la fonction de résolution *resolPut* qui retourne *seqPlein* si toutes les exceptions signalées par les divers composantes de la multiprocédure sont de type *plein* et qui retourne l'exception FAILURE sinon. Nous supposons que cette fonction est déclarée dans une classe dont la variable d'état *tampons*, de type SEQ OF *tampon*, désigne la séquence de tampons à laquelle est ajouté l'élément. Nous précisons la déclaration de la fonction *resolPut* où nous utilisons la commande RAISE, bien que non encore introduite (cf. § VI.2.3.1) ; cette dernière commande explicite la levée d'une exception. Nous obtenons :

```
resolPut HANDLES plein SIGNALS seqPlein =
  VAR
    i : INTEGER := 0 ;
    suite : BOOLEAN := TRUE ;
    tampPlein : SEQ OF tampon := <> ;
  BEGIN
    WHILE (i < SEQEXC ! Length()) AND (suite) DO
      i := i + 1 ;
      EXCEPTION CASE SEQEXC[i] OF
        plein      : tampPlein ! Append(tampons[i]) ;
        TERMINATED : SKIP ;
        ELSE       : suite := FALSE ;
      END
    END ;
    IF suite
       THEN RAISE seqPlein(tampPlein)
```

```
      ELSE RAISE FAILURE
    END ;
  END resolPut ;
```

VI.2.2.3 Commande de traitement d'exceptions

La commande de traitement d'exceptions permet de déclarer un contexte de traitement d'exceptions : elle associe une liste de traitants d'exceptions à une commande du langage. La forme générale de cette commande est :

$$\text{TRY USING } r\ C\ \text{EXCEPT } (e_i\ (v_i) : C_i\ ;)^*\ \text{ELSE } C\ \text{END } ;$$

où :

- r est le nom d'une fonction de résolution ;
- C est une commande ;
- v_i est un identificateur (optionnel) servant à nommer l'exception,
- e_i est un type exception ;
- C_i est une commande qui définit le traitant de l'exception e_i ;
- la clause ELSE permet de déclarer un traitant d'exceptions par défaut qui est défini par la commande C.

Lors de la définition de notre modèle de traitement d'exceptions, nous avons indiqué qu'une exception concertée était calculée par l'opération parallèle *appelée*. Cependant, les opérations parallèles citées étaient supposées *statiquement* déclarées. La définition d'une multi-procédure est, dans le langage ARCHE, dynamique et établie par l'appelante. Par conséquent, le calcul de l'exception concertée revient ici à l'appelante de la multi-procédure. La clause USING de la commande de traitement d'exceptions, qui est optionnelle, précise la fonction de résolution à appeler pour le calcul d'une exception concertée lors d'un appel à une multi-procédure au sein de C. Si cette clause n'est pas mentionnée, la valeur de toute exception concertée est donnée par la définition VI.1, page 87.

Remarquons que deux exceptions pour lesquelles un traitant est déclaré dans la commande de traitement d'exceptions peuvent être en relation de sous-typage.

Par conséquent, plusieurs traitants peuvent être déclarés pour une même exception. Le traitant exécuté sera alors celui qui apparaît en premier dans la déclaration des traitants.

La signification de la commande de traitement d'exceptions diffère suivant que *C* définisse ou non le comportement d'une méthode d'initialisation.

Dans l'affirmative, la commande déclare des traitants pour des exceptions signalées par des méthodes d'initialisation. Ces traitants sont des points d'entrée au même titre que les opérations. Remarquons que toute exception figurant dans la clause HANDLES de la vue réalisée doit nécessairement avoir un traitant qui lui correspond dans la commande de traitement d'exceptions. Nous reprenons l'exemple VI.1, page 90, pour illustrer notre propos.

Exemple VI.4

Considérons le type *utilTamp* de la bibliothèque *N* qui est donnée dans l'exemple VI.1. Une réalisation de ce type doit nécessairement déclaré un traitant pour l'exception *vide*. Nous obtenons :

```
CLASS cUtilTamp IMPLEMENTS N.utilTamp =
  FROM L USES tampon, vide ;
  VAR
    tamp : tampon ;
  PROCEDURE
    ... ;
  BEGIN
    TRY
      (* Corps de la methode d'initialisation *) ;
    EXCEPT
      vide (e) : (* traitant de l'exception
                    d'initialisation vide
                 *)
    END
END cUtilTamp
```

Nous illustrons à présent l'utilisation de la commande de traitement d'exceptions lorsque celle-ci ne définit pas le comportement d'une méthode d'initialisation.

Exemple VI.5

Reprenons l'exemple VI.3, page 97, et considérons l'expression de l'appel à la multi-procédure *put* de la séquence de nom *tampons*. La commande de traitement d'exceptions englobant cet appel explicite la fonction *resolPut* pour le calcul de toute exception concertée. Un traitement possible de l'exception *seqPlein* consiste à retirer tous les éléments, appartenant à *tamp*, de la séquence *tampons*. Supposons déclarée la variable *elem* qui désigne l'élément à ajouter, nous obtenons :

```
TRY USING resolPut
  tampons ! put (elem) ;
EXCEPT
  seqPlein (e) : tampons ! Filter(e.seqPlein) ;
  ELSE         : RAISE FAILURE ;
END
```

VI.2.2.4 Création d'objet

Considérons la création d'un exemplaire d'une classe $\mathcal{R}$, réalisant $\mathcal{V}$, de la forme :

$$\mathcal{O} := \text{NEW}.\mathcal{R}\ \textit{(paramètres effectifs)}$$

et une exception e pouvant être signalée par la méthode d'initialisation de $\mathcal{V}$.

Il est possible de préciser les processus traitant e en explicitant "*e : traitants*" dans la liste des paramètres effectifs, *traitants* étant de type SEQ OF T où T est une vue. Pour que le passage de ce paramètre soit valide, il est nécessaire que la vue T traite l'exception e. En revanche, si l'ensemble des processus traitant une exception signalée par la méthode d'initialisation n'est pas explicité dans l'appel à l'opérateur de création, la classe qui effectue la création de l'objet doit déclarer un traitant pour cette exception qui, par défaut, la traite.

VI.2.3 Signalement des exceptions

Nous examinons à présent les commandes et expressions qui peuvent signaler une exception. Il s'agit de la commande de signal explicite et de tout appel

d'opération. Nous introduisons de plus une nouvelle commande qui permet à une composante de multi-procédure de capturer les exceptions globales signalées les autres composantes de la multi-procédure. Nous explicitons l'intérêt d'une telle commande lors de sa présentation.

VI.2.3.1 Signal explicite d'exception

La forme générale d'une commande de signal d'une exception est :

RAISE e $(x_1, ..., x_n)$.

L'exécution de cette commande conduit à la création d'un exemplaire de la classe *e* puis au signal classique de l'*exception*. Nous employons le terme *exception* pour désigner à la fois les classes d'exception et les objets qui en sont des exemplaires, la référence aux unes ou aux autres pouvant se déduire du contexte.

Lorsqu'une exception est signalée par une méthode d'initialisation, elle est signalée à tous les processus la traitant. Ces processus sont, soit ceux explicités lors de la création de l'objet, soit celui défini par défaut et qui est l'émetteur du message de création.

Une exception traitée peut être propagée en explicitant : RAISE *v* au sein d'un traitant, où *v* est le nom de l'exception traitée. Cette facilité n'est permise qu'au sein d'un traitant où l'exception traitée est explicitement nommée. Dans la définition actuelle du MTE, nous n'offrons pas la possibilité de déclarer des variables de type exception hormis dans la déclaration des traitants d'une commande de traitement d'exceptions. Nous envisageons cette extension mais nous pensons qu'elle mérite encore d'être étudiée avant d'être incluse dans la définition du MTE.

VI.2.3.2 Appel simple d'opération

Tout appel d'une opération simple, qu'il s'agisse d'une procédure locale à l'objet ou d'une opération, conduit à la traduction de l'exception signalée en l'exception FAILURE si l'exception signalée n'est pas traitée dans le contexte courant de traitement d'exceptions.

VI.2.3.3 Appel coordonné d'opération

Une composante de multi-procédure peut capturer une voire plusieurs exceptions globales lorsqu'elle appelle une opération, ou une multi-procédure, de

manière coordonnée. Une telle capture survient lorsqu'une des composantes de la multi-procédure appelée signale une exception globale. Dans ce cas, une exception concertée est évaluée et son traitant est recherché dans le contexte courant de traitement d'exceptions.

VI.2.3.4 Capture explicite d'exceptions globales

Jusqu'à présent, nous avons essentiellement considéré le traitement des exceptions en présence de processus coopérants. Une multi-procédure peut toutefois définir un calcul parallèle avec des processus compétitifs.

Considérons un tableau réparti sur divers objets. La recherche d'un élément dans un tel tableau peut être effectuée en parallèle au moyen d'une multi-procédure. Pour des raisons d'efficacité, il peut être pratique d'expliciter la terminaison anticipée des composantes de la multi-procédure dès que l'une d'elles signale une exception. La seule manière de l'expliciter, étant donnée la définition de notre modèle est de recourir à une communication artificielle. Afin d'éviter une synchronisation inutile, nous introduisons la commande non bloquante ACCEPT.

Un appel à la commande ACCEPT se traduit par la capture d'exceptions globales et la recherche du traitant de l'exception concertée correpondante, si l'opération l'exécutant est composante d'une multi-procédure et a connaissance du signal d'une exception (globale) par une voire plusieurs autres composantes ; la commande ACCEPT est sans effet sinon.

Nous exemplifions l'utilisation de la commande ACCEPT dans ce qui suit.

Exemple VI.6

Considérons la gestion, précédemment citée, d'un tableau réparti. Nous nous attardons plus spécifiquement sur la programmation de l'opération d'un élément dans une sous-partie d'un tableau d'éléments de type T. Nous supposons effectuées les déclarations suivantes :

```
TYPE
  trouve : EXCEPTION place : INTEGER END ;
  absent : EXCEPTION ;
VAR
  etape   : INTEGER ;
  tableau : SEQ OF T ;
```

Concernant ces déclarations, l'*exception trouve* signifie que l'élément recherché est présent et qu'il se trouve à la place, *place*, du sous-tableau traité. L'exception *absent* signifie que l'élément ne figure pas dans le sous tableau. Par ailleurs, la variable *etape* détermine le nombre d'éléments consultés entre chaque acceptation d'exceptions (distantes) et la variable *tableau* détermine la sous-partie du tableau localement consultée. Une déclaration possible de la procédure de recherche est :

```
recherche : (e : T) () SIGNALS trouve, absent =
  VAR
   i : INTEGER := 1 ;
  BEGIN
    WHILE i < tableau ! Length() DO
      WHILE (i MOD etape != 0 ) DO
        IF tableau[i] = e
           THEN RAISE trouve(i)
        END ;
        i := i + 1 ;
      END ;
      ACCEPT ;
    END ;
    RAISE absent ;
  END recherche ;
```

Notons qu'aucune fonction de résolution n'est explicitée pour le calcul des exceptions concertées au sein de cette opération ; la définition par défaut est retenue. Par conséquent, si plus de deux composantes signalent concurremment l'exception *trouve* ou si une des composantes signale l'exception *absent* concurremment avec une composante signalant *trouve*, la composante qui capture ces exceptions signale l'exception FAILURE. La reconnaissance de l'exception concertée est laissée à la charge de l'appelante de la multiprocédure *recherche*. Il serait toutefois préférable d'effectuer une première évaluation d'exception concertée lors de la capture du fait du caractère imnprécis de l'exception FAILURE.

VI.3 Discussion

Les premières expérimentations que nous avons pu mener avec le langage ARCHE pour l'écriture de programmes robustes, c'est à dire qui exhibent un comportement satisfaisant même dans les cas ne survenant que très rarement,

sont encourageantes. L'écriture d'un protocole de validation à deux phases a montré le caractère réutilisable des logiciels. Une application fondée sur la technique de programmation en N versions a aussi été écrite [71]. Ces deux exemples recourrent essentiellement à la facilité de traitement d'exceptions et à la notion de multi-procédure. Un système de gestion de données répliquées qui a été conçu dans le langage POLYGOTH [72] devrait aussi s'écrire facilement dans le langage ARCHE tout en n'ayant pas les inconvénients de la première solution citée. Le caractère statique des multiprocédures et l'absence d'un MTE avaient alors été perçus comme des limites du langage POLYGOTH pour l'écriture d'une telle application.

Chapitre VII

Conclusion

Dans ce document, nous avons présenté les principales caractéristiques du langage à objets parallèles ARCHE. Dans ce qui suit, nous rappelons succinctement les principaux point développés puis nous donnons les principes de mise en œuvre du compilateur du langage.

La première partie a introduit les travaux à la base de la conception du langage ARCHE ; il s'agit de la définition de la notion de multiprocédure et du langage POLYGOTH. La seconde partie a consisté en la définition informelle du langage ARCHE. La notion d'objet ainsi que le système de type du langage ont tout d'abord été traités. Les aspects du langage relatifs à la présence de parallélisme ont ensuite été introduits ; nous trouvons notamment la notion de multiprocédure et un mécanisme de synchronisation conditionnelle compatible avec celui d'héritage. Ce dernier mécanisme qui a fait l'objet d'un chapitre se subdivise en la facilité de sous-typage et celle de sous-classage. Enfin, le traitement des exceptions dans le langage ARCHE a été présenté.

Concernant le compilateur du langage ACHE, il repose sur la définition d'une *machine objet* dans laquelle l'objet est la seule entité manipulée à l'exécution. La description d'une réalisation centralisée de cette machine, au dessus du noyau MACH, peut être trouvée dans [73], une réalisation distribuée étant actuellement à l'étude [74]. La phase de compilation d'une unité *bibliothèque* (*library*) ou *classe* (*class*) repose sur l'utilisation des outils présentés dans [75]. Elle s'effectue en trois étapes. Lors de la première étape, l'analyse lexicale et syntaxique de l'unité est d'abord effectuée. Un arbre d'analyse qui sera exploité lors des phases de compilation suivantes est aussi généré. La deuxième étape est celle d'analyse sémantique. Des descriptions de types ou classes sont générées suivant l'unité compilée. Un contrôle sur les types utilisés est réalisé. D'autres vérifications, comme la validité de la réalisation d'un type lors de la compilation d'une classe, sont aussi effectuées. La troisième et dernière étape qui concerne les unités classes

est celle de génération de code. Le code produit est du code C. Ce code fait aussi appel à des opérations plus complexe offertes par la *machine objet*, par exemple, les opérations d'appels de méthodes.

Bibliographie

[1] J.P. Banâtre and M. Banâtre, *Les systèmes distribués : l'expérience du système* GOTHIC. InterEditions, 1991.

[2] N. Wirth, *Programming in Modula-2.* Springer-Verlag, 1982.

[3] P. Lecler, "Une approche de la programmation des systèmes distribués fondée sur la fragmentation des données et des calculs, et sa mise en œuvre dans le système GOTHIC," thèse, Université de Rennes-I, Sept. 1989.

[4] M. Benveniste, "Operational semantics of a distributed object-oriented language and its Z formal specification," Rapport de recherche 1230, INRIA, May 1990.

[5] P. Le Certen and P. Lecler, "Le langage POLYGOTH : présentation et exemples," in *[1]*, pp. 81–109, 1991.

[6] J.P. Banâtre, "Contribution à l'étude de méthodes et d'outils de construction de programmes parallèles et fiables," thèse d'Etat, Université de Rennes-I, Dec. 1980.

[7] P. Henderson, "Generalized blocks," Tech. Rep. RC 6240, IBM Research Division, T.J. Watson Research center, Oct. 1976.

[8] J.P. Banâtre, M. Banâtre, and F. Ployette, "The concept of multi-function: A general structuring tool for distributed operating systems," in *6th Conference on Distributed Computing Systems*, (Cambridge, Mass.), pp. 478–485, May 1986.

[9] M. Benveniste, "Procédure, parallélisme et objets : une approche par la généralisation," thèse, Université de Rennes-I, 1992. In preparation.

[10] M. Atkinson and R. Morrison, "Procedures as persistent data objects," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 539–559, Oct. 1985.

[11] F. Ployette, P. Le Certen, and P. Lecler, "POLYGOTH: le langage de GOTHIC, Manuel d'Utilisation." LSP project internal research report, IRISA, Sept. 1988.

[12] D. Callahan and K. Kennedy, "Compiling programs for distributed-memory multiprocessors," *The Journal of Supercomputing*, no. 2, pp. 151–169, 1988.

[13] M. Rosing, R. Schnabel, and R. Weaver, "Dino: Summary and examples," in *ACM 3rd Conference on Hypercubes Concurrent Computers and Applications*, pp. 472–481, 1988.

[14] F. André, J.-L. Pazat, and H. Thomas, "Pandore : A system to manage data distribution," in *International Conference on Supercomputing*, ACM, June 1990.

[15] B. Michel, "Conception et réalisation de la mémoire virtuelle de GOTHIC," thèse, Université de Rennes-I, Sept. 1989.

[16] E. Dijkstra, "The structure of the 'THE' multiprogramming system," *Communications of the ACM*, vol. 11, pp. 341–346, May 1968.

[17] P. Brinch Hansen, "Structured multiprogramming," *Communications of the ACM*, vol. 15, pp. 574–578, July 1972.

[18] D. G. Kafura and K. H. Lee, "Inheritance in actor based concurrent object-oriented languages," *The Computer Journal*, vol. 32, no. 4, pp. 297–303, 1989.

[19] J. Guttag and J. Horning, "The algebraic specification of abstract data types," *Acta Informatica*, vol. 10, no. 1, pp. 27–52, 1978.

[20] J. Guttag, J. Horning, and J. Wing, "LARCH in five easy pieces," Tech. Rep. 5, Digital, Systems Research Center, July 1985.

[21] C. Jones, *Systematic Software Development using VDM.* Prentice-Hall International, 1986.

[22] J. Spivey, *The Z Notation: A Reference Manual.* Prentice-Hall International, 1989.

[23] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," Tech. Rep. SRI-CSL-91-05, SRI International, Menlo Park, Feb. 1991.

[24] K. Chandy and J. Misra, *Parallel program design: A foundation.* Addison-Wesley Publishing Company, Inc., Readings, Mass., 1988.

[25] C. Morgan, *Programming from Specifications.* Prentice-Hall International, 1990.

[26] M. Bidoit, H.-J. Kreowski, P.Lescanne, F.Orejas, and D.Sannella, eds., *Algebraic system specification and development: A survey and annotated bibliography.* No. 501 in Lecture Notes in Computer Science, Springer-Verlag, 1991.

[27] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM Computing Surveys*, vol. 17, pp. 471–522, Dec. 1985.

[28] B. Meyer, *Object-oriented software construction.* Prentice-Hall International, 1988.

[29] W. Cook, "A proposal for making Eiffel type-safe," in *[76]*, pp. 57–70, 1989.

[30] J. Hur and K. Chon, "Self and selftype," *Information Processing Letters*, vol. 36, pp. 225–230, Dec. 1990.

[31] M. Sakkinen, "Selftype is a special case," *Information Processing Letters*, vol. 38, pp. 221–224, May 1991.

[32] B. Meyer, "Eiffel : A language and environment for software engineering," *The Journal of Systems and Software*, vol. 8, pp. 199–246, 1988.

[33] P. Buhr and C. Zarnke, "Nesting in an object oriented language is NOT for the birds," in *[77]* (S. Cook, ed.), pp. 128–145, 1988.

[34] W. Cook, *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[35] S. Krakowiak, *Principes des systèmes d'exploitation des ordinateurs.* Dunod informatique, 1985.

[36] F. André, D. Herman, and J.-P. Verjus, *Synchronisation de programmes parallèles.* Dunod informatique, 1983.

[37] G. Andrews and F. Schneider, "Concepts and notations for concurrent programming," *ACM Computing Surveys*, vol. 15, pp. 3–43, Mar. 1983.

[38] R. Filman and D. Friedman, *Coordinated Computing: Tools and Techniques for Distributed Software.* Computer Science, McGraw-Hill Book Company, 1984.

[39] C. Hoare, *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[40] B. Nelson, *Remote Procedure Call.* PhD thesis, Carnegie-Mellon University, Pittsburgh, 1981.

[41] G. Agha, *Actors: a Model of Concurrent Computation in Distributed Systems.* Artificial Intelligence, MIT Press, 1986.

[42] G. Agha and C. Hewitt, "Concurrent programming using actors," in *[78]*, pp. 37–54, 1987.

[43] C. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM*, vol. 17, pp. 549–557, Oct. 1974.

[44] R. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE transactions on software engineering*, vol. SE-12, pp. 157–171, Jan. 1986.

[45] C. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[46] O. Nierstrasz, "Active objects in Hybrid," in *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, (Orlando, Florida), pp. 243–253, 1987.

[47] S. Krakowiak, M. Meysembourg, H. Van Nguyen, M. Riveill, C. Roisin, and X. Rousset de Pina, "Design and implementation of an object-oriented strongly typed language for distributed applications," *Journal of Object-Oriented Programming*, pp. 11–22, Sept. 1990.

[48] P. Robert and J. Verjus, "Towards autonomous descriptions of synchronization modules," in *Proceedings of IFIP congress (B. Gilchrist, ed.)*, pp. 981–986, 1977.

[49] P. America, "POOL-T: a parallel object-oriented language," in *[78]*, pp. 199–220, 1987.

[50] C. Morgan and J. Woodcock, "Refinement of state-based concurrent systems," in *VDM'90: VDM and Z, formal methods in Software Development*, no. 428 in Lecture Notes in Computer Science, Springer-Verlag, Apr. 1990.

[51] O. Nierstrasz and M. Papathomas, "Viewing objects as patterns of communicating agents," in *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 38–43, 1990.

[52] O. Nierstrasz and M. Papathomas, "Towards a type theory for active objects," *ACM OOPS Messenger*, vol. 2, pp. 89–93, Apr. 1991.

[53] T. Leconte, "Compilation des synchronisations pour un langage parallèle à actions atomiques," thèse, Université de Rennes-I, 1992. In preparation.

[54] S. Keene, *Object-oriented programming in Common Lisp: A programmer's guide to* CLOS. Addison-Wesley Publishing Company, Inc., Readings, Mass., 1989.

[55] N. Francez, *Fairness.* Springer-Verlag, 1991.

[56] M. Raynal and J. Helary, *Synchronisation et contrôle des systèmes et des programmes répartis.* Eyrolles, 1987.

[57] R. Back, E. Hartikainen, and R. Kurki-Suonio, "Multi-process handshaking on broadcasting networks," Tech. Rep. Reports in Computer Science, 42, Abo Akademi, 1985.

[58] C. Morin, "Protocole d'appel de multiprocédure à distance," in *[1]*, pp. 217–249, 1991.

[59] D. Le Métayer, M. Jégado, M. Benveniste, and V. Issarny, "Evaluation report of the COMMANDOS platform: Language, object model and reliability." Rapport interne, convention BULL-INRIA, IRISA, Rennes, Apr. 1991.

[60] P. America, "A behavioural approach to subtyping in object-oriented programming languages," tech. rep., Philips research laboratory, January 1989.

[61] P. America and F. van der Linden, "A parallel object-oriented language with inheritance and subtyping," tech. rep., Philips research laboratory, December 1989.

[62] T. Korson and J. D. McGregor, "Understanding object-oriented: A unifying paradigm," *Communications of the ACM*, vol. 33, pp. 40–60, September 1990.

[63] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson, "The Modula-3 type system," in *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 202–212, January 1989.

[64] F. Cristian, "Exception handling," Tech. Rep. RJ 5724, IBM Research Division, T.J. Watson Research center, July 1987.

[65] V. Issarny, "Le traitement d'exceptions: Aspects théoriques et pratiques," Rapport de recherche 1118, INRIA, Nov. 1989.

[66] V. Issarny, "An exception handling model for parallel programming and its verification," in *Proceedings of the ACM SIGSOFT'91 Conference on Software for Critical Systems*, (New Orleans, Louisiana), pp. 92–100, December 1991.

[67] B. H. Liskov and A. Snyder, "Exception handling in CLU," *IEEE transactions on software engineering*, vol. SE-5, pp. 546–558, November 1979.

[68] R. H. Campbell and B. Randell, "Error recovery in asynchronous systems," *IEEE transactions on software engineering*, vol. SE-12, pp. 811–826, August 1986.

[69] J. Franco and D. Friedman, "Creating efficient programs by exchanging data for procedures," *Computer Languages*, vol. 14, pp. 11–23, Jan. 1989.

[70] A. Koenig and B. Stroustrup, "Exception handling for C++ (revised)," in *Usenix C++ Conference*, pp. 149–176, 1990.

[71] M. Jegado, "A case study in N-version programming," note interne LSP, IRISA, November 1990.

[72] V. Issarny, "Construction d'un système de gestion de fichiers répartis à l'aide du concept d'objet fragmenté," in *Actes des conférences techniques. Convention UNIX 89*, pp. 43–54, Mars 1989.

[73] S. Olhagaray, "Machine d'exécution pour le langage ARCHE," rapport de fin d'études, IRISA, Juin 1991.

[74] I. Puaut and J.-P. Routeau, "Supporting an object-based system: Experience with the MACH micro-kernel," in *ERCIM workshop on Distributed Systems*, (Lisbon, Portugal), Nov. 1991.

[75] J. Grosh and H. Emmelmann, "A Tool Box for Compiler Construction," Compiler Generation Report 20, GMD Forschungsstelle an der Universität Karlsruhe, January 1990.

[76] S. Cook, ed., *European Conference on Object-Oriented Programming*. Nottingham, G.B.: Cambridge University Press, July 1989.

[77] S. Gjessing and K. Nygaard, eds., *European Conference on Object-Oriented Programming*. No. 322 in Lecture Notes in Computer Science, Springer-Verlag, Aug. 1988.

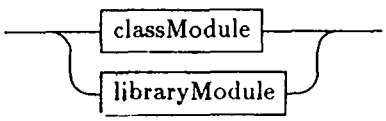
[78] A. Yonezawa and M. Tokoro, eds., *Object-Oriented Concurrent Programming*. Computer Systems, MIT Press, 1987.

Annexe A:
Syntaxe du langage ARCHE

Marc Benveniste
Valérie Issarny
IRISA/INRIA-Rennes
Campus de Beaulieu, 35042 Rennes Cedex, FRANCE
e-mail: mbenveni/issarny@irisa.fr

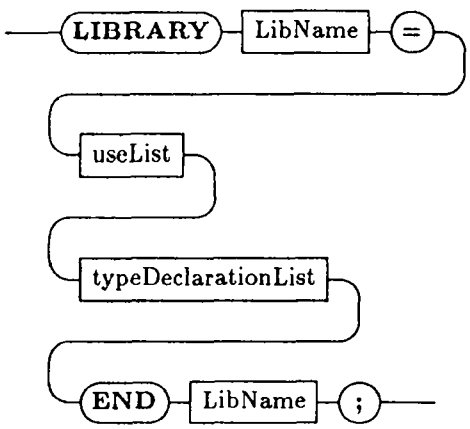
Les symboles terminaux suivants sont lexicalement identiques : LibName, TypeName, FieldName, EnumItem, StateName, MethName, ClassName, ViewName, ParName, ResolName, ConstName et VarName. Ils sont tous formés comme les identificateurs de ARCHE, Ident, à savoir : suite de lettres et de chiffres commençant par une lettre.

compilationUnit

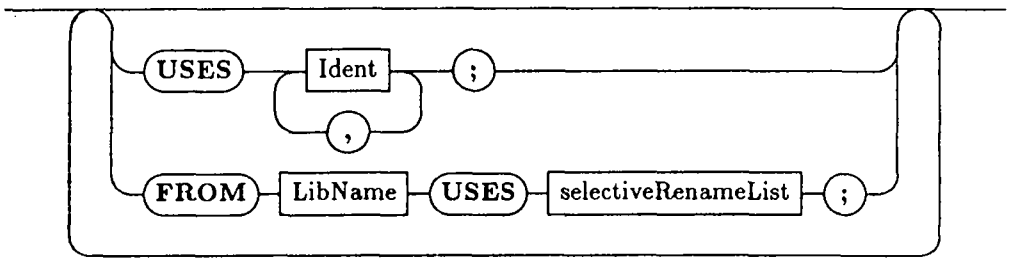


1 L'unité bibliothèque

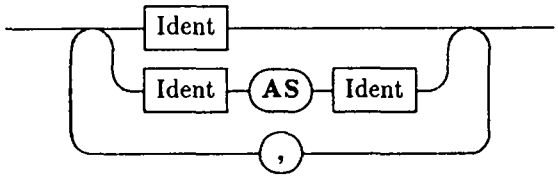
libraryModule



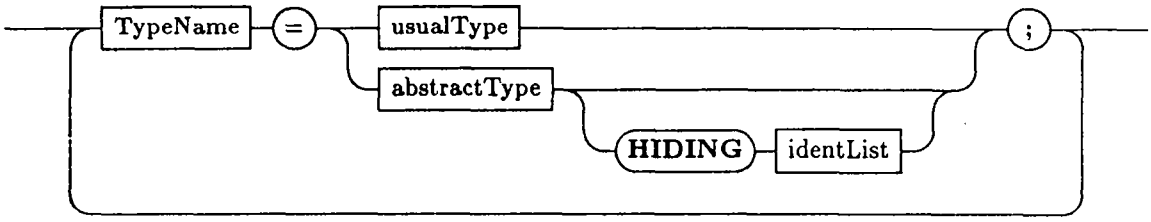
useList



selectiveRenameList

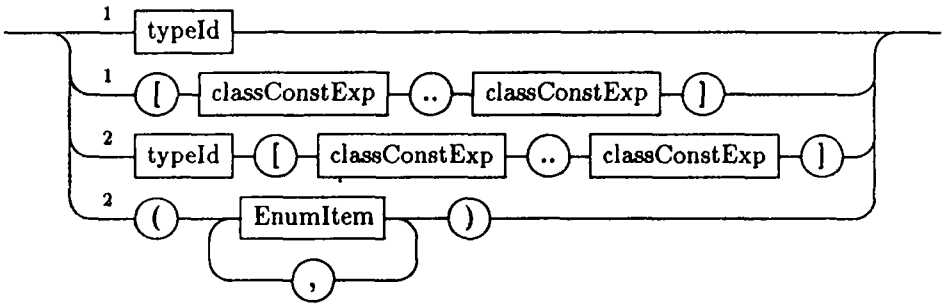


typeDeclarationList

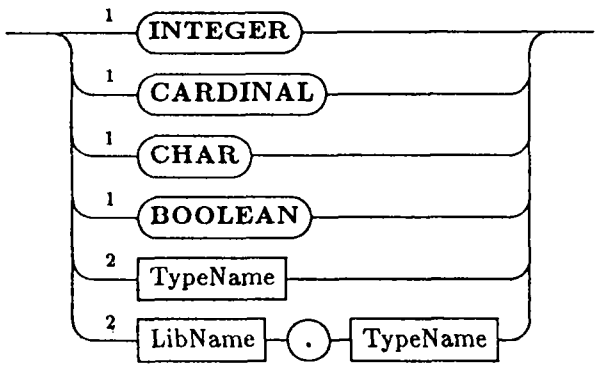


1.1 Les types concrets

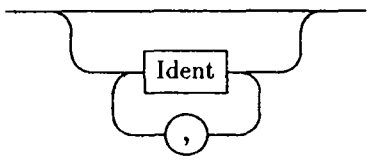
simpleType



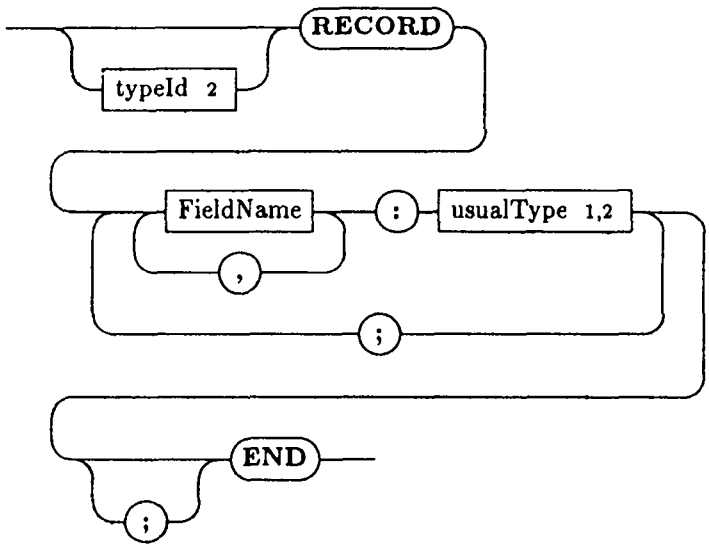
typeId



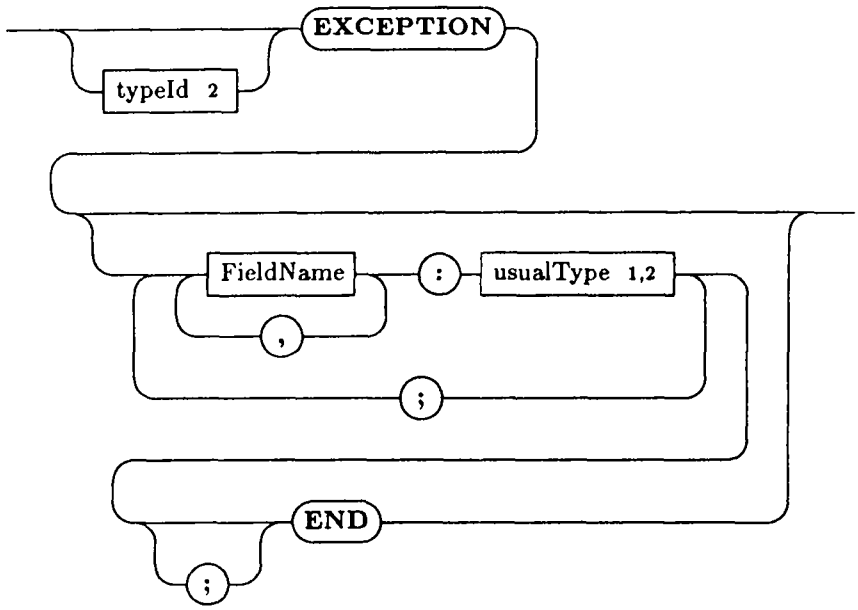
identList



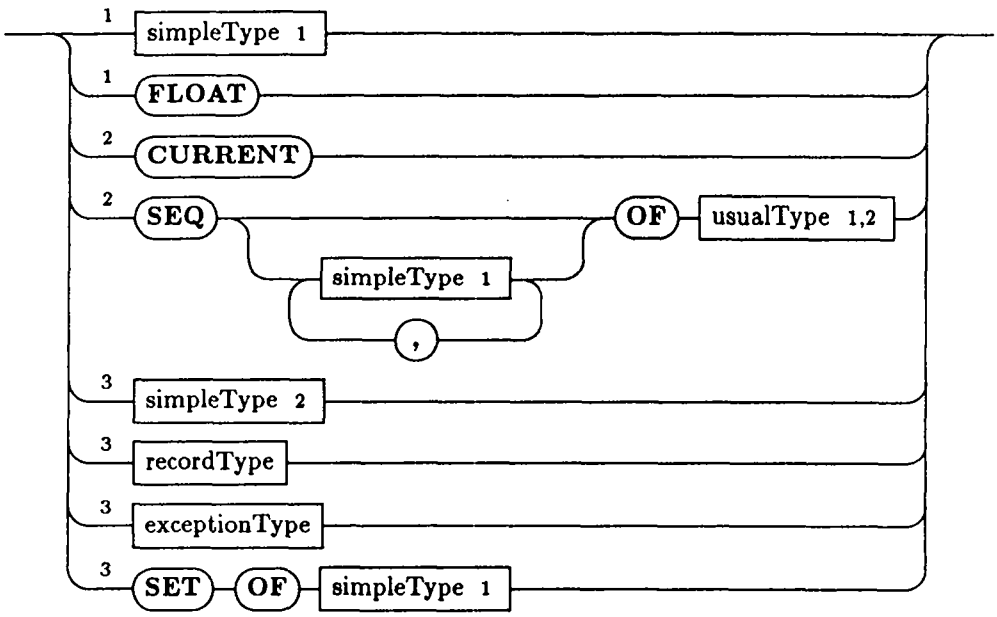
recordType



exceptionType

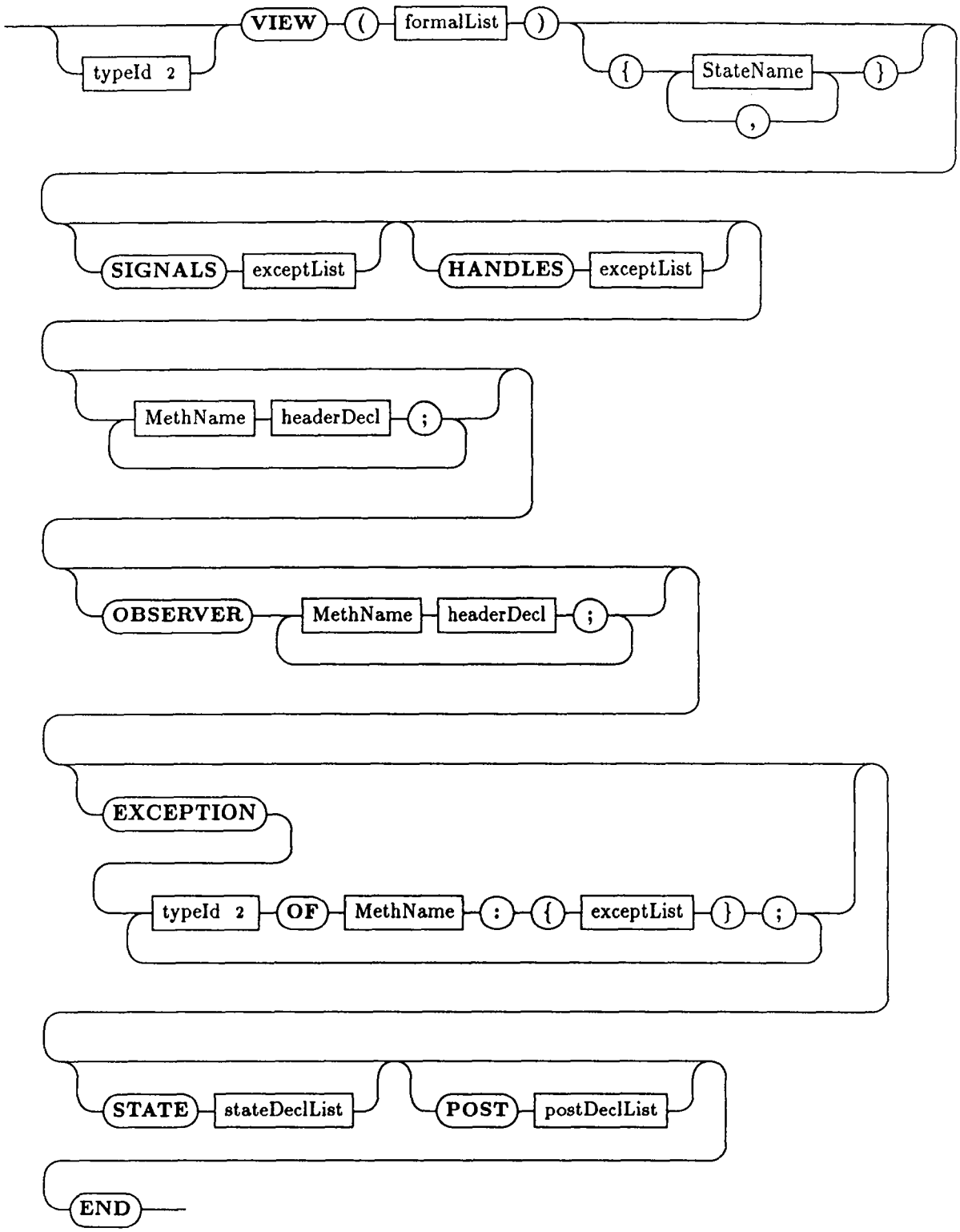


usualType

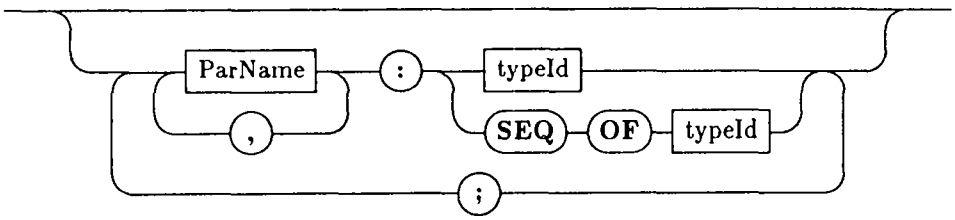


1.2 Les types abstraits

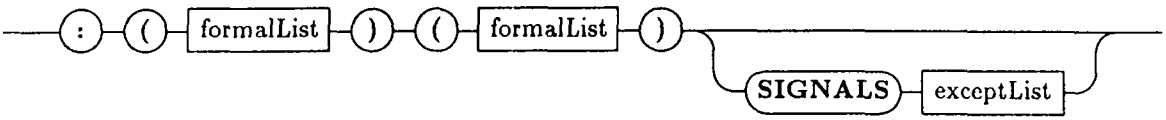
abstractType



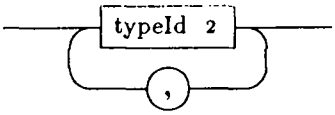
formalList



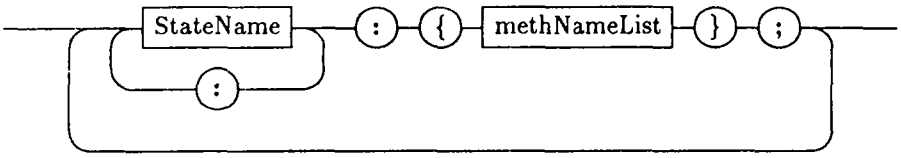
headerDecl



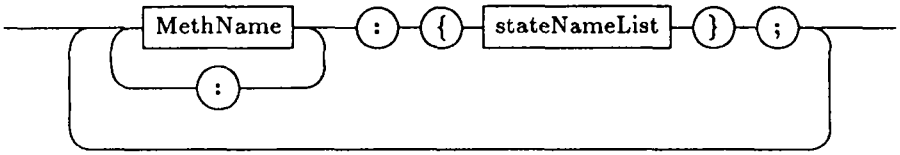
exceptList



stateDeclList

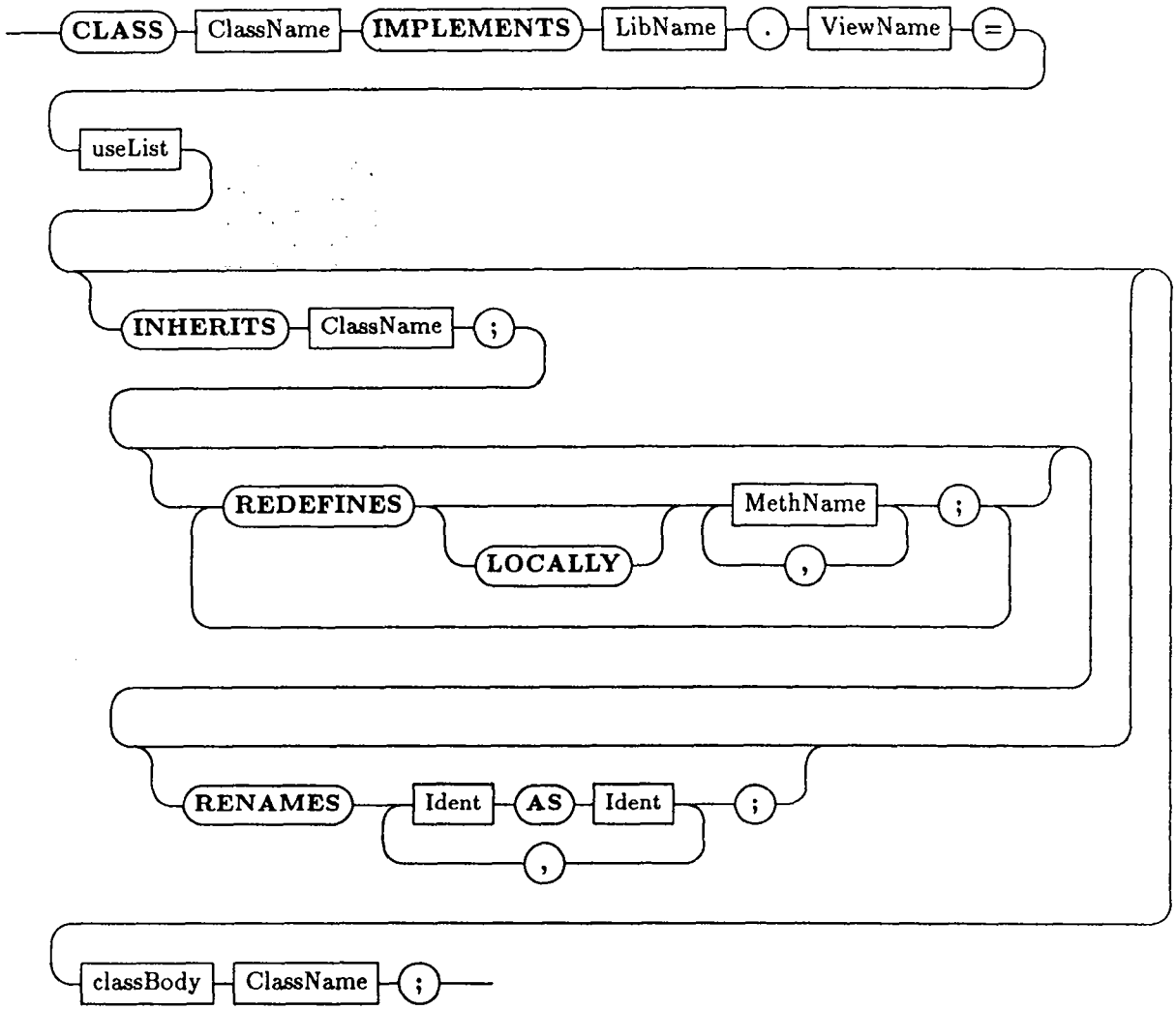


postDeclList

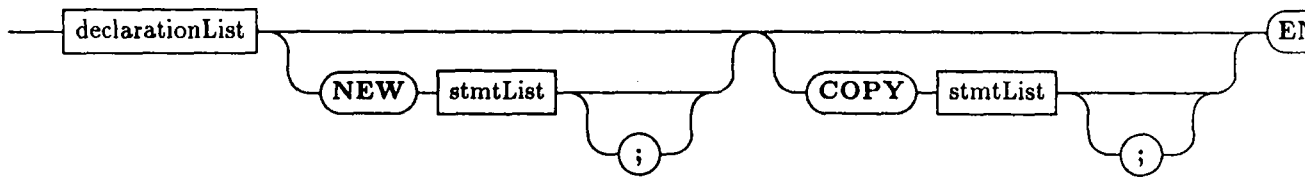


2 L'unité classe

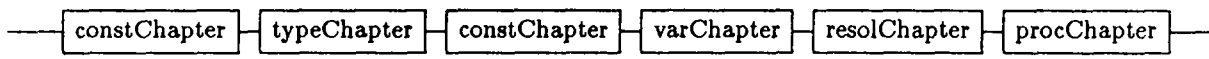
classModule



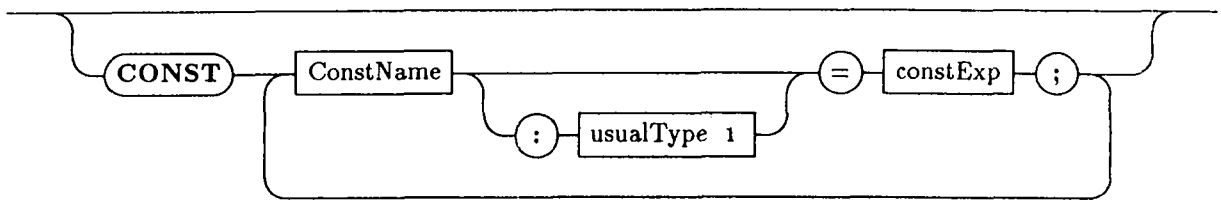
classBody



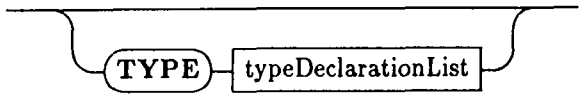
declarationList



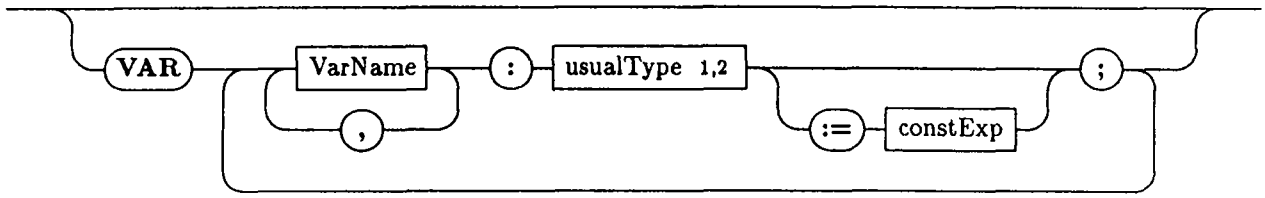
constChapter



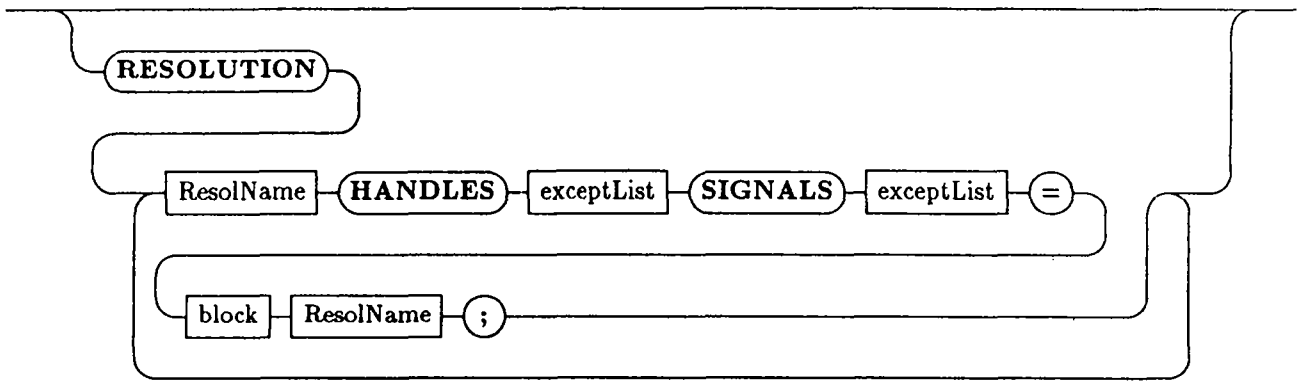
typeChapter



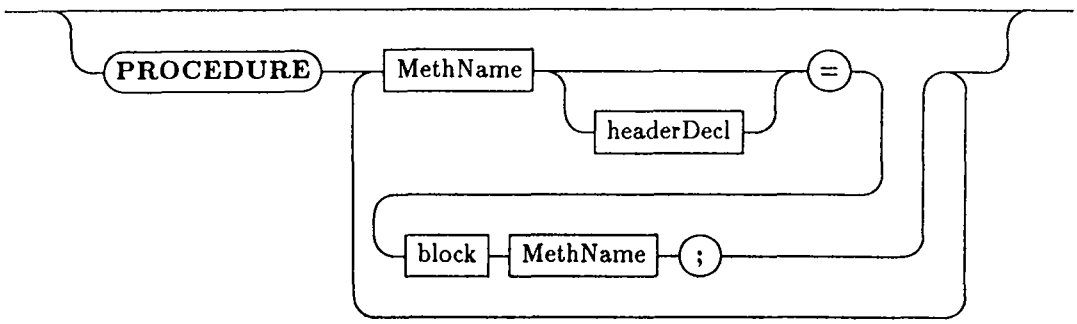
varChapter



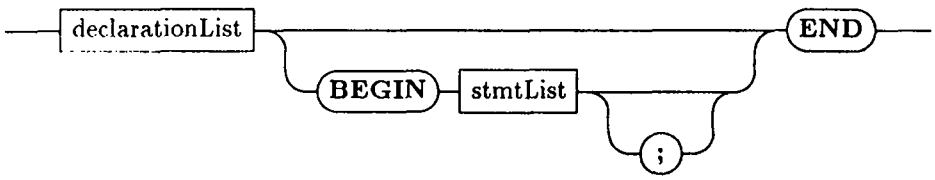
resolChapter



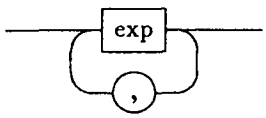
procChapter



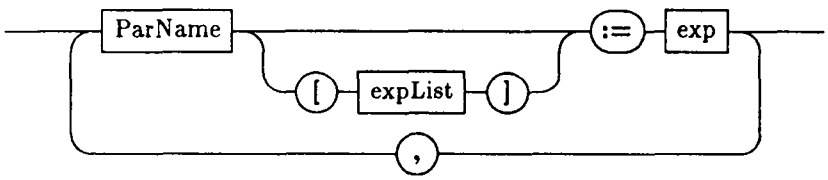
block



expList



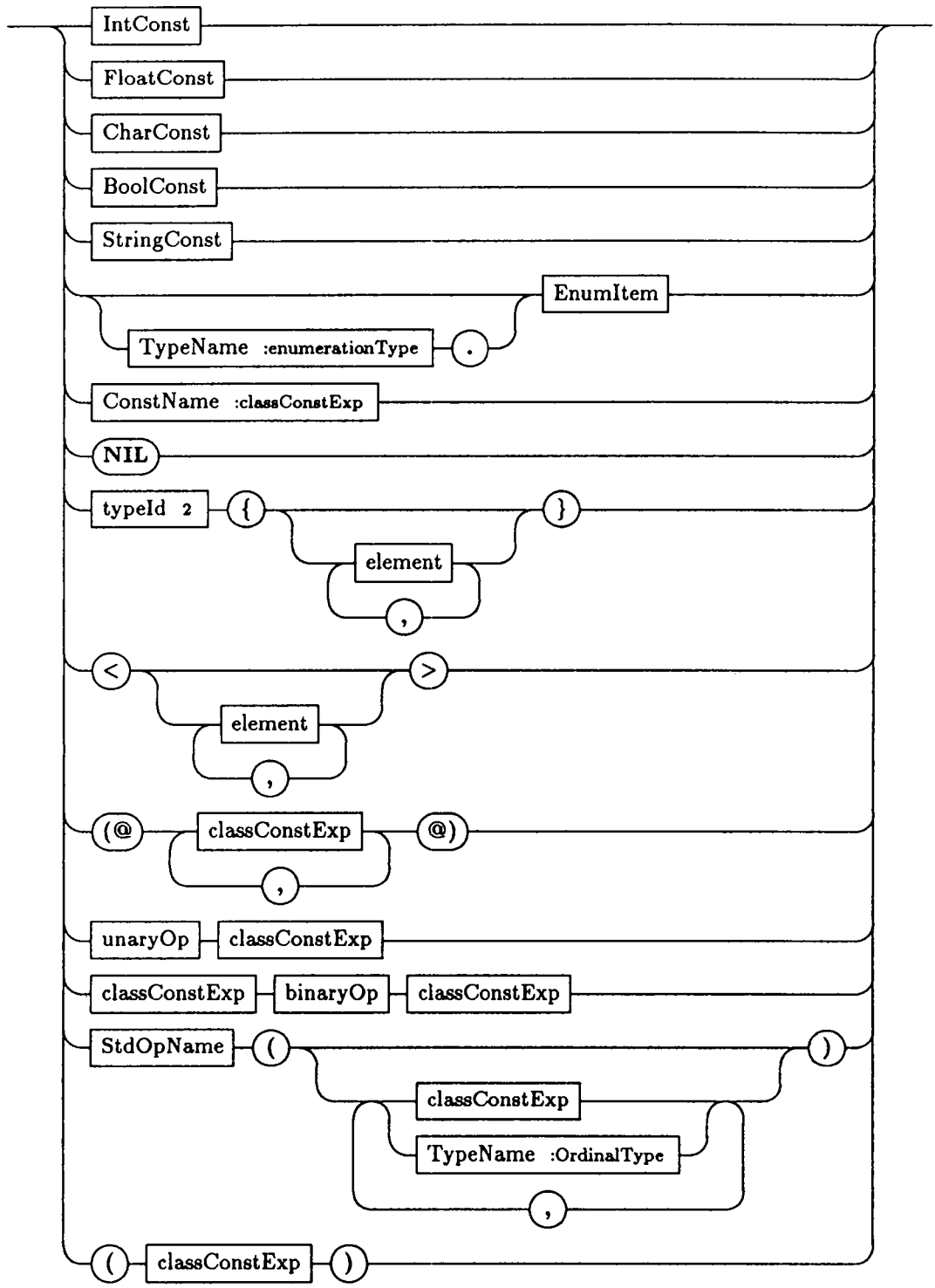
assList



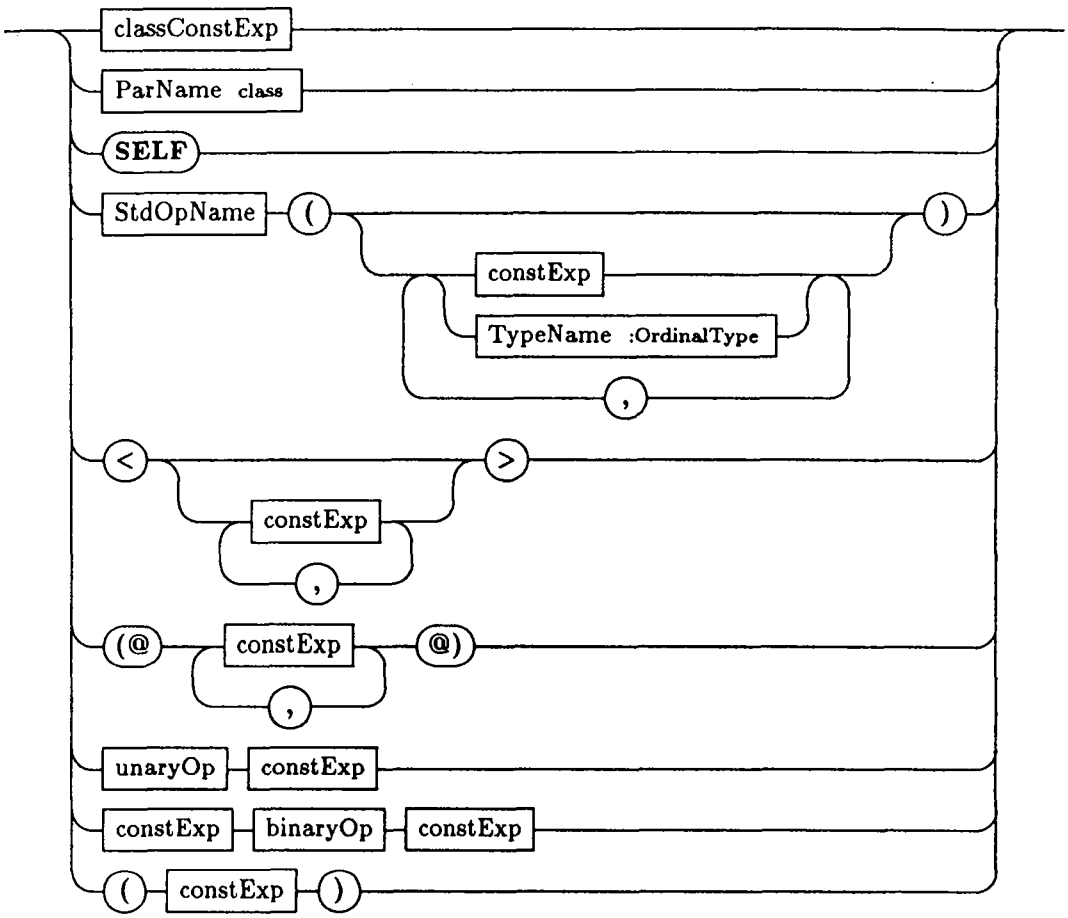
3 Les expressions

3.1 Les expressions constantes

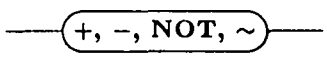
classConstExp



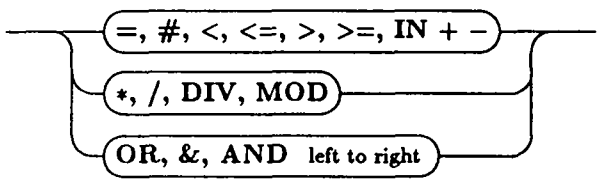
constExp



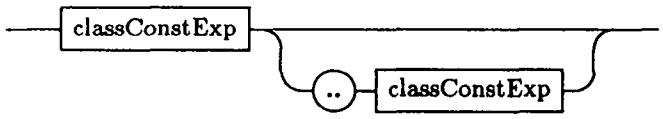
unaryOp



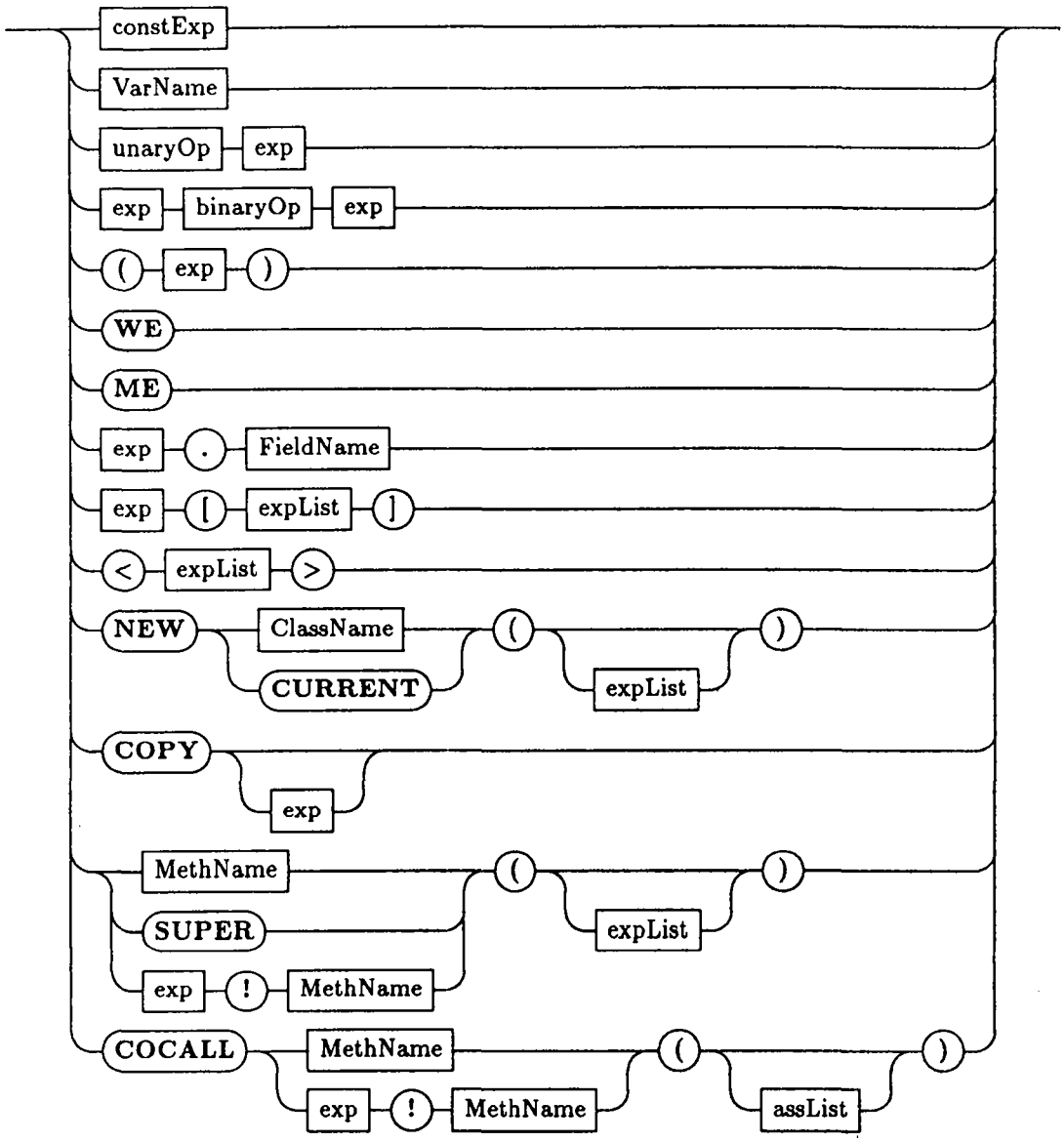
binaryOp



element

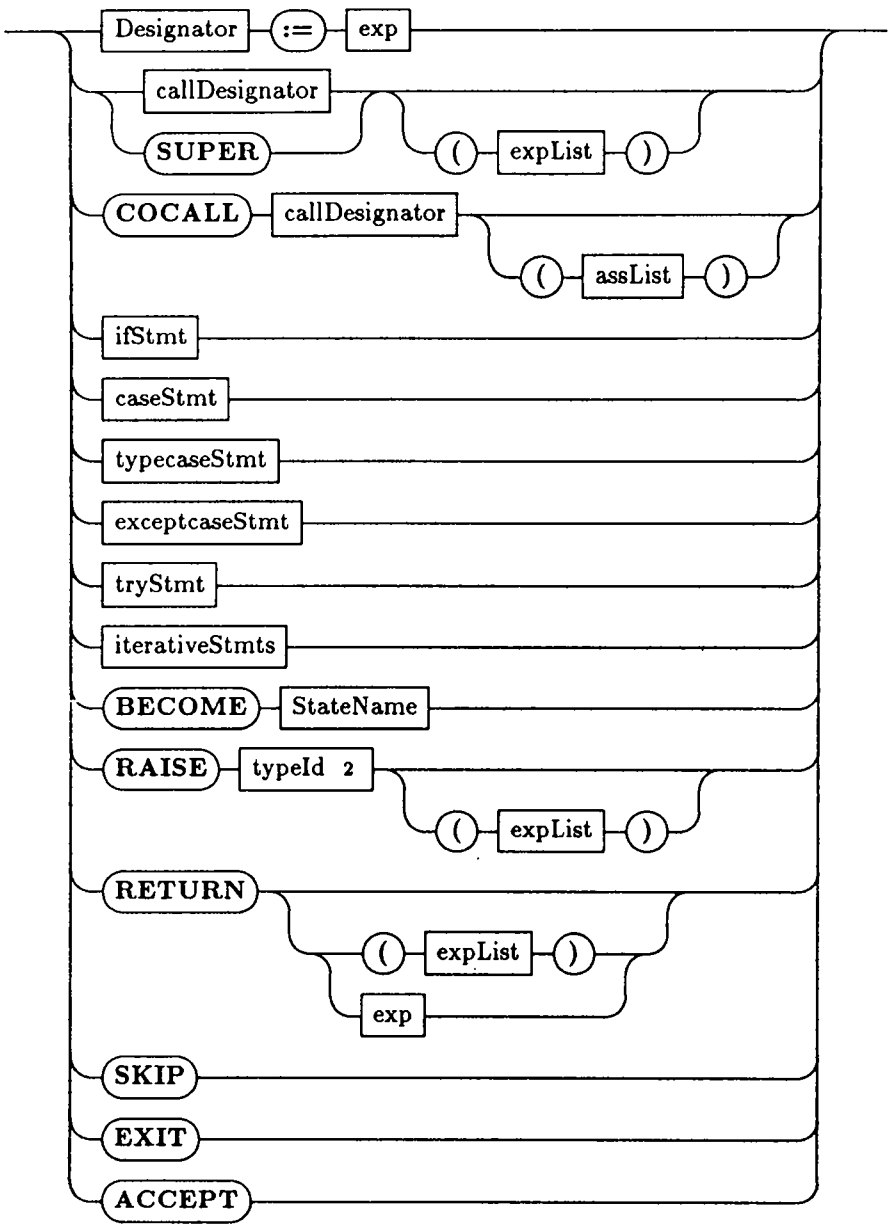


exp

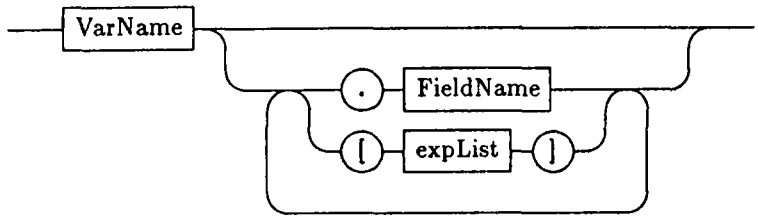

constExp
VarName
unaryOp
exp
exp
binaryOp
exp
(
exp
)
WE
ME
exp
.
FieldName
exp
[
expList
]
<
expList
>
NEW
ClassName
CURRENT
(
expList
)
COPY
exp
MethName
SUPER
exp
!
MethName
(
expList
)
COCALL
MethName
exp
!
MethName
(
assList
)

4 Les commandes

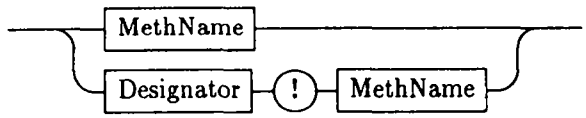
stmt



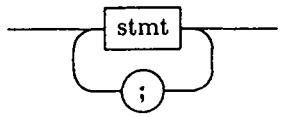
Designator



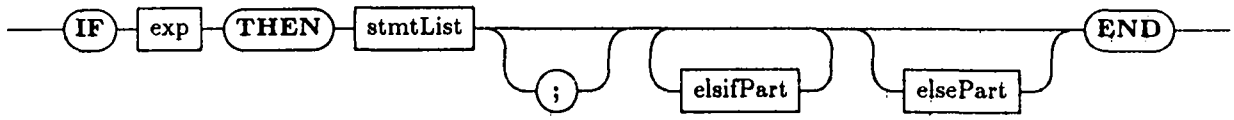
callDesignator



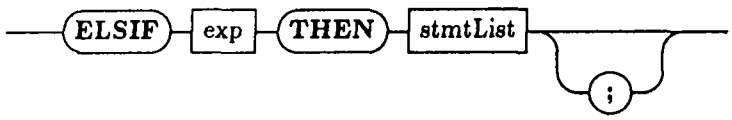
stmtList



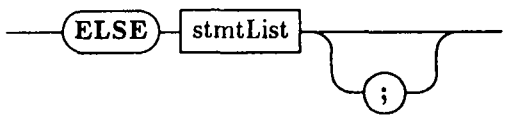
ifStmt



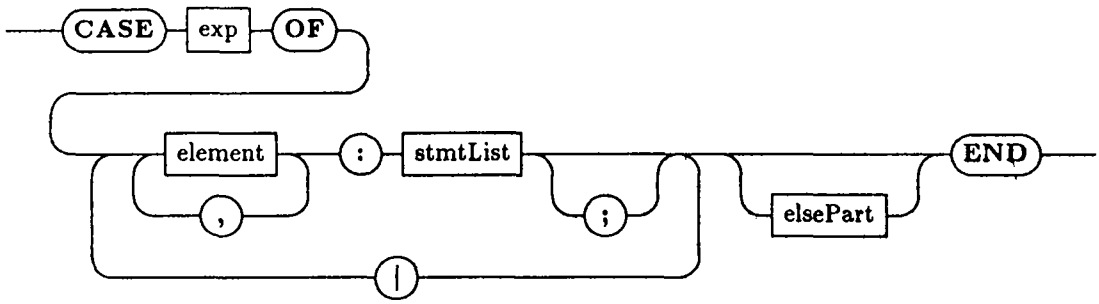
elsifPart



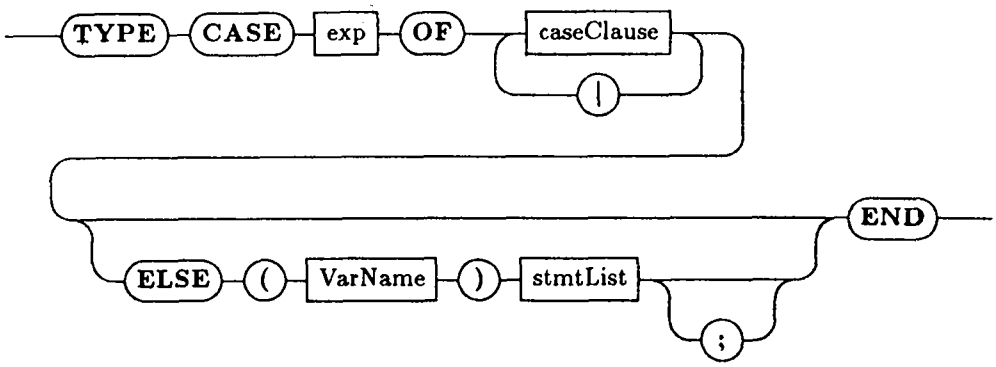
elsePart



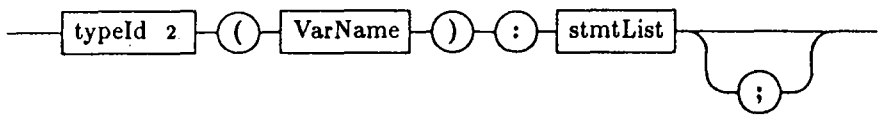
caseStmt



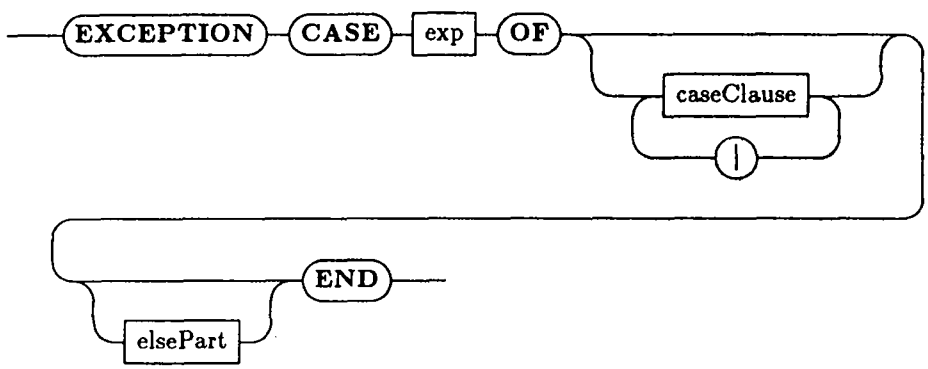
typecaseStmt



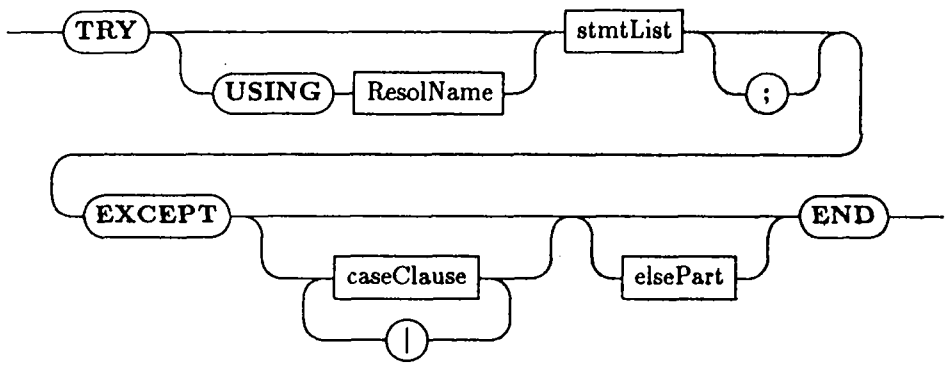
caseClause



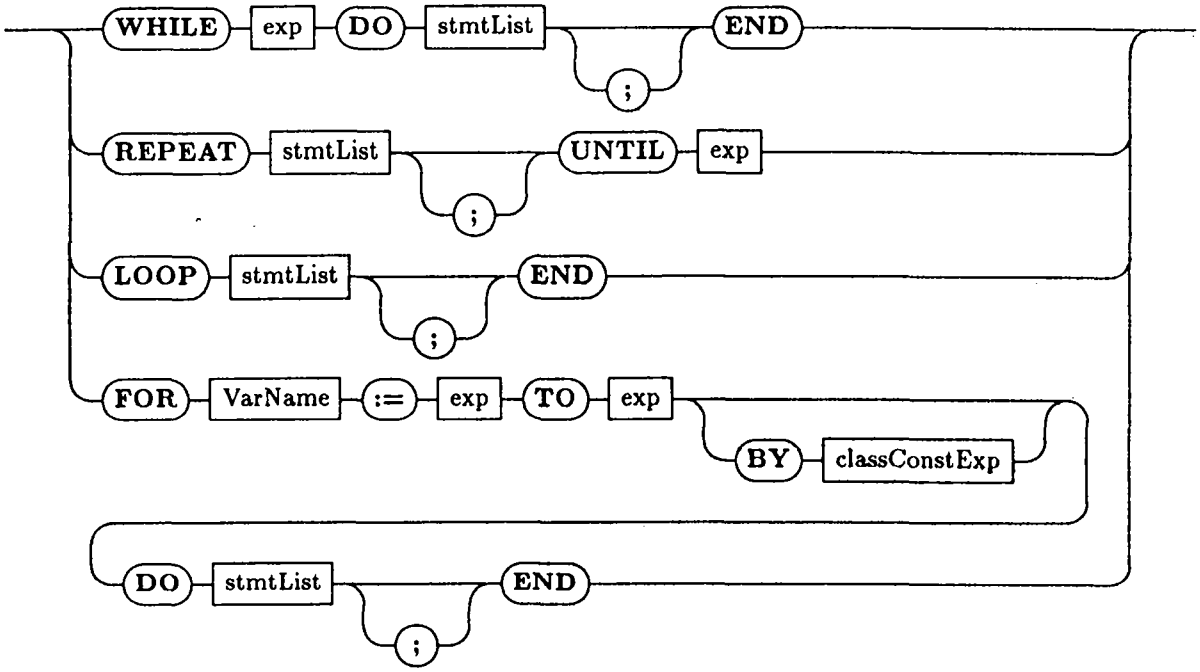
exceptcaseStmt



tryStmt

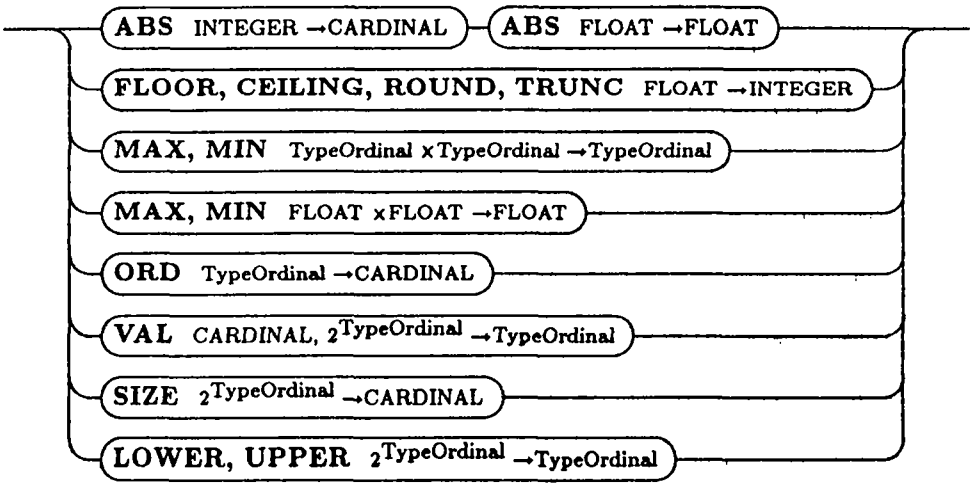


iterativeStmts



5 Opérations primitives

StdOpName



LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 632 L-STABLE PARALLEL ONE-BLOCK METHODS FOR ORDINARY DIFFERENTIAL EQUATIONS
Philippe CHARTIER, Bernard PHILIPPE
Janvier 1992, 28 pages.

PI 633 ON EFFICIENT CHARACTERIZING SOLUTIONS OF LINEAR DIOPHANTINE EQUATIONS AND ITS APPLICATION TO DATA DEPENDENCE ANALYSIS
Christine EISENBEIS, Olivier TEMAM, Harry WIJSHOFF
Janvier 1992, 22 pages.

PI 634 UN NOYAU DE SYSTEME REPARTI POUR LES APPLICATIONS GEREES PAR UN TEMPS VIRTUEL
Philippe INGELS, Carlos MAZIERO, Michel RAYNAL
Janvier 1992, 20 pages.

PI 635 A NOTE ON CHERNIKOVA'S ALGORITHM
Hervé LE VERGE
Février 1992, 28 pages.

PI 636 ENSEIGNER LA TYPOGRAPHIE NUMERIQUE
Jacques ANDRE, Roger D. HERSCH
Février 1992, 26 pages.

PI 637 TRADE-OFFS BETWEEN SHARED VIRTUAL MEMORY AND MESSAGE PASSING ON AN IPSC/2 HYPERCUBE
Thierry PRIOL, Zakaria LAHJOMRI
Février 1992, 26 pages.

PI 638 RUPTURES ET CONTINUITES DANS UN CHANGEMENT DE SYSTEME TECHNIQUE
Alan MARSHALL
Mars 1992, 510 pages.

PI 639 EFFICIENT LINEAR SYSTOLIC ARRAY FOR THE KNAPSACK PROBLEM
Rumen ANDONOV, Patrice QUINTON
Mars 1992, 18 pages.

PI 640 TOWARDS THE RECONSTRUCTION OF POSET
Dieter KRATSCH, Jean-Xavier RAMPON
Mars 1992, 22 pages.

PI 641 MADMACS : A TOOL FOR THE LAYOUT OF REGULAR ARRAYS
Eric GAUTRIN, Laurent PERRAUDEAU
Mars 1992, 12 pages.

PI 642 ARCHE : UN LANGAGE PARALLELE A OBJETS FORTEMENT TYPES
Marc BENVENISTE, Valérie ISSARNY
Mars 1992, 132 pages.

ISSN 0249-6399