Adjonction de la compilation separee et de la modularite au langage Pascal a l'aide de Mentor

V. Migot, M. Loyer

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports Techniques

N° 24

ADJONCTION DE LA COMPILATION SÉPARÉE ET DE LA MODULARITÉ AU LANGAGE PASCAL À L'AIDE DE MENTOR

Valérie MIGOT
Michel LOYER

Mai 1983

ADJONCTION DE LA COMPILATION SEPAREE ET DE LA MODULARITE AU LANGAGE PASCAL A L'AIDE DE MENTOR

Valérie MIGOT, Michel LOYER
INRIA
Domaine de Voluceau 78153 Le Chesnay

RESUME

Dans la premiere partie de ce rapport, nous présentons des extensions au langage Pascal standard ISO pour introduire la compilation séparée et des mécanismes de modularité.

Dans la suite, nous montrons comment ces extensions peuvent être traitées sous Mentor, a l'aide d'un pré-processeur qui transforme des programmes écrits en Pascal modulaire en des programmes acceptables par un compilateur Pascal standard.

ABSTRACT

First, we describe some extensions to the Pascal Language (standard ISO) which give facilities for separate compilation and modularity.

Then, we present an implementation of these extensions, using Mentor. A pre-processor transforms programs written in extended Pascal into programs written in standard Pascal.

Ce rapport a été écrit a l'aide de
l'éditeur Mentor_Rapport [MEL 83].

Table des matières

Introduction

Le choix des extensions que nous proposons de faire a Pascal a été guidé par les objectifs suivants:

- permettre la réalisation d'une application Pascal au moyen de plusieurs unités de compilation en assurant la cohérence globale, c'est-à-dire en garantissant que les objets définis dans une unité seront manipulés de façon correcte dans les autres unités de compilation de l'application,

- offrir des mécanismes permettant d'expliciter les liens entre les différentes unités de compilation,

- limiter au minimum les ajouts au Pascal standard ([ISO 82] [AFN 82]) et ne pas compliquer par ces ajouts les mécanismes nécessaires a la compilation d'un programme Pascal.

Ces extensions sont dérivées des mécanismes que nous avons définis pour le langage Legos ([BOU 81] [LOY 81]).

Elles peuvent être traitées de deux façons:

- Soit directement, par modification d'un compilateur Pascal standard ISO. Actuellement, un autre membre de l'équipe (Francis Prusker) a terminé l'introduction de ces extensions dans un compilateur Pascal sur SM90. Ce travail fera l'objet d'un rapport technique a paraître.

- Soit par un pré-processeur qui transforme un programme écrit en Pascal modulaire en un programme acceptable par un compilateur standard. C'est ce que nous avons fait dans le pré-processeur, utilisable sous Mentor ([DON 80] [DON 81] [MEL 80] [MEL 81]), que nous décrivons dans la troisième partie.

En respectant ces objectifs, nous ne prétendons ni définir de nouveaux mécanismes de modularité ni, a fortiori, un nouveau langage, mais nous nous situons plutôt dans le cadre des propositions d'extension de Pascal demandées par des organismes comme l'ISO, l'AFNOR ou l'ANSI.

1– Présentation des extensions proposées

1.1- Définitions

L' *unité de compilation* est le programme Pascal tel qu'il est défini dans la norme ISO.

Une *application* est constituée d'une ou plusieurs unités de compilation.

Nous appelons *interface* d'un programme l'ensemble des objets déclarés au plus haut niveau dans un programme.

Enfin le terme *composant* nous sert a désigner interface ou programme lorsqu'il n'y a pas lieu de distinguer ces deux catégories d'objets.

Dans un programme P, il est possible de référencer des objets déclarés dans d'autres programmes. Ces programmes -ou plus exactement les interfaces de ces programmes- constituent le *contexte* du programme P.

Ces programmes sont indiqués dans une liste *USE* placée en tête du programme P.

On a alors la regle suivante:

> Tout objet référencé dans un programme P doit avoir été déclaré dans le programme P ou dans une des interfaces des programmes qui appartiennent a son contexte.

Exemple de programmes écrits en Pascal modulaire:

```
program DEFINITION;
     const
          MAX     =20;
     type PILE    =
               record
                    P:array[1..MAX]of INTEGER;
                    TOP:INTEGER
               end;

     procedure EMPILER(var PIL:PILE;ITEM:INTEGER);external;

     procedure DEPILER(var PIL:PILE);external;

     function SOMMET(var PIL:PILE):INTEGER;external;

     function PILENONVIDE(var PIL:PILE):BOOLEAN;external;

     procedure INIT(var PIL:PILE);external;
```

Le corps de chaque sous-programme, dont l'en-tête est déclarée dans le programme DEFINITION, est défini dans le programme suivant:

```
use DEFINITION;
program REALISATION;

     procedure EMPILER;
          begin
          if PIL.TOP<MAX then
               begin
               PIL.TOP:=PIL.TOP+1;
               PIL.P[PIL.TOP]:=ITEM
               end
          end;

     procedure DEPILER;
          begin
          if PIL.TOP#0 then PIL.TOP:=PIL.TOP-1
          end;

     function SOMMET;
          begin
          if PIL.TOP#0 then SOMMET:=PIL.P[PIL.TOP]
          else SOMMET:=-1234567890
          end;

     function PILENONVIDE;
          begin
          if PIL.TOP=0 then PILENONVIDE:=FALSE
          else PILENONVIDE:=TRUE
          end;

     procedure INIT;
          begin
          PIL.TOP:=0
          end;
```

Ce troisieme programme référence le programme DEFINITION:

```
use DEFINITION;
program UTILISATION(OUTPUT);
     var  PILE1,PILE2:PILE;
          I:INTEGER;
     begin
     INIT(PILE1);
     INIT(PILE2);
     for I:=1 to 5 do EMPILER(PILE1,I);
     while PILENONVIDE(PILE1)do
          begin
          EMPILER(PILE2,-SOMMET(PILE1));
          DEPILER(PILE1)
          end;
     while PILENONVIDE(PILE2)do
          begin
          WRITELN(SOMMET(PILE2));
          DEPILER(PILE2)
```

```
        end
    end.
```

1.2- Les ajouts au Pascal standard

1.2.1- Les ajouts

Les ajouts proposés sont minimes. Ils consistent en deux points:

1- Ajouter en tête d'un programme une liste *USE* énumérant les autres unités de compilation dont des objets sont référencés dans le programme P.

2- Autoriser que l'en-tête et le corps d'un sous-programme de plus haut niveau soient dans deux unités de compilation distinctes.
Dans l'unité de compilation contenant la définition de l'en-tête, le corps est alors remplacé par la directive EXTERNAL. Cette directive EXTERNAL est semblable à la directive FORWARD, la seule différence étant que le corps associé n'est pas défini ultérieurement dans la même unité de compilation mais dans une autre unité.
Quant à l'unité contenant la définition du corps associé, elle doit contenir dans sa liste *USE* le nom de l'unité où a été définie l'en-tête.

Nous avons refusé de faire d'autres ajouts comme d'introduire un mécanisme pour cacher la structure d'un type à l'extérieur de l'unité de compilation où il a été défini ou comme d'étendre la notation pointée sous la forme: nom_de_programme.nom_d'objet .
Ces ajouts supplémentaires, bien que fort utiles, nous auraient trop éloignés du Pascal standard et auraient rendu plus compliquée l'adaptation de compilateurs existants.

C'est ce même souci de simplicité qui nous a conduit à ne pas autoriser la compilation séparée des corps des sous-programmes imbriqués comme c'est le cas en Ada [ADA 80] ou comme dans la proposition d'extension à Pascal, de D.C.Robbins et L.R.Carter [CAR 82].

1.2.2- Les avantages

Ces ajouts offrent les avantages suivants:

- permettre la vérification de cohérence (en particulier la cohérence des types) entre les différents composants d'une application,

- éviter la redescription des objets externes dans les composants qui les importent, ce qui est en général fastidieux et source d'erreurs,

- connaître facilement, grâce aux listes *USE,* l'ensemble des dépendances entre les composants d'une application et ainsi mesurer les conséquences entraînées par la modification d'un composant sur le reste de l'application (en particulier, déduire les recompilations).

1.2.3- Les répercussions

Un programme ne peut être compilé que si l'ensemble des programmes constituant son contexte a été compilé au préalable. Cela définit donc un ordre partiel de compilation entre les unités d'une application. De même la modification d'une unité de compilation P implique normalement la recompilation des unités référençant P.

La gestion des divers composants d'une application et le maintien de la cohérence de l'ensemble pourront être utilement faits à l'aide d'un catalogue réunissant les informations contenues dans les listes *USE* et d'un outil reconfigurateur comme Make [FEL 77] par exemple.

Pour avoir un compilateur traitant les mécanismes de compilation séparée proposés, les ajouts a faire consistent:

- en fin de compilation, à conserver une table de description des objets déclarés au plus haut niveau dans le programme P, c'est-a-dire a conserver sous forme interne l'interface de P;

- en début de compilation, a fusionner les tables de description des programmes figurant dans la liste *USE* de P, c'est-a-dire a bâtir le contexte de P.

Ces deux opérations se situant en amont et en aval de la compilation proprement dite peuvent être assez facilement ajoutées a un compilateur Pascal standard.

Une autre solution consiste a transformer, à l'aide d'un pré-processeur, les textes des programmes utilisant ces mécanismes de compilation séparée en programmes Pascal standard utilisant des mécanismes de compilation séparée plus élémentaires (def, ref). Cette solution sera développée dans la troisième partie de ce rapport.

Remarque:

Pour des raisons d'efficacité, il est important que le contexte de compilation d'un programme soit réduit au minimum. Pour cela nous n'avons pas autorisé l'héritage de listes *USE* entre unités de compilation. Ainsi, si un programme Q est mentionné dans la liste *USE* d'un programme R et si un programme P est lui-même mentionné dans celle de Q, alors P ne fait pas implicitement partie du contexte de R. Et P n'aura a être mis dans la liste *USE* de R que si R référence effectivement des objets de P.

Ceci permet d'une part de ne garder trace, au moyen des listes *USE,* que des liaisons effectives entre programmes et d'autre part de minimiser la taille de la table de contexte résultant de la fusion.

1.3- Un mécanisme complémentaire: les interfaces réduites

1.3.1- Interfaces réduites

Une autre opération permettant de limiter les contextes de compilation consiste à pouvoir définir une ou plusieurs *interfaces réduites* à partir de ce que nous avons appelé jusqu'ici interface d'un programme et que nous appellerons désormais *interface globale* d'un programme (i.e. l'ensemble des objets déclarés au plus haut niveau).

Interface réduite sur un composant

Une interface réduite est obtenue en ôtant certaines déclarations de l'interface globale d'un programme (ou d'une interface déja réduite). Cette réduction doit se faire en respectant la regle suivante:

> Si, lors de la création d'une interface réduite I sur un composant C, une déclaration D est conservée, alors tout objet utilisé pour la définition de D et déclaré dans C doit lui aussi être conservé dans I.

Cette regle garantit que les objets conservés dans l'interface forment un ensemble cohérent. Ainsi, si une déclaration de sous-programme est conservée, on devra conserver également les déclarations des types des parametres formels.

A chaque interface réduite est associé un nom qui peut être mentionné dans la liste *USE* d'un autre programme.

Pour un même programme, comme dans Protel [CAS 81], on peut définir plusieurs interfaces réduites: chacune est alors un "cache" particulier donnant un certain "angle de vue" sur le programme.

Interface sur un ensemble de composants

Une interface réduite peut être construite, de la même façon, non plus sur un composant, mais sur un ensemble de composants. Ce mécanisme permet de définir une interface plus synthétique pour un ensemble de composants constituant une sous-partie d'une application.

Pour l'utilisateur de cette interface, la façon dont sont agencés les composants constituant cette sous-partie est cachée. Pour lui, tout se passe comme si cette interface ne portait que sur un seul composant.

Ceci permet une description hiérarchique d'une application. L'application ou systeme se décompose en plusieurs sous-systèmes, caractérisés par leurs interfaces. Chaque sous-systeme se décompose a son tour en composants plus élémentaires.

Ce mécanisme s'inspire des configurations définies dans le langage Mesa [MIT 79].

1.3.2- Le constructeur d'interfaces

D'un point de vue pratique, la réduction d'une interface globale consiste à produire une table réduite à partir des tables de descriptions des objets déclarés dans des programmes déjà compilés.

Cette réduction est réalisée par un processeur distinct du compilateur, le constructeur d'interfaces.

Ce constructeur peut être conçu pour travailler à partir d'une description de l'interface réduite fournie par l'utilisateur sous une forme syntaxique proche de Pascal. Le constructeur, tout en produisant les tables réduites, vérifie la cohérence de cette description. C'est l'approche que nous avons choisie pour Legos [BOU 81].

Une autre solution consiste à définir un constructeur d'interfaces interactif. Le programmeur se contente d'indiquer les noms des objets qu'il souhaite conserver dans l'interface réduite et est guidé dans ces choix par le constructeur. Celui-ci contrôle à chaque instant la correction de l'interface en construction et interdit les opérations illicites. Outre les tables réduites, le constucteur produit le texte de l'interface construite sous forme Pascal.

Cette solution a été retenue pour la réalisation du pré-processeur que nous allons maintenant présenter.

2– Contexte de réalisation

Pour implémenter, sur le système Multics de l'INRIA, les mécanismes présentés dans la premiere partie, il était possible d'écrire de façon classique un pré-processeur au compilateur. Nous avons préféré utiliser Mentor pour programmer ce pré-processeur et les commandes de manipulation des programmes écrits dans notre Pascal-modulaire.

2.1- Le compilateur Pascal-Multics

Le compilateur Pascal disponible sur Multics, dans la version 6.02, est muni de mécanismes de compilation séparée élémentaires.

Il permet de partager des variables et des sous-programmes entre des programmes compilés séparément.

Ces mécanismes se réduisent à l'importation et a l'exportation de variables et de sous-programmes en respectant les regles suivantes [PAS 81]:

- Dans le programme qui exporte,

les variables et les sous-programmes exportés doivent être déclarés au plus haut niveau avec l'attribut 'def';

d'autre part, si le corps du programme est vide, dans l'en-tête, le nom du programme doit être précédé de 'def'.

- Dans le programme qui importe,

les noms des variables et sous-programmes importés doivent être cités dans une liste, appelée liste d'externes, située dans l'en-tête du programme;

de plus, les objets importés doivent être redéclarés dans le programme qui les importe (pour les sous-programmes, le corps est remplacé par la directive 'external').

2.2- Présentation rapide de Mentor

Mentor est un système de manipulation de programmes représentés par des arbres de syntaxe abstraite ([DON 79] [DON 80] [DON 81] [MEL 80] [MEL 81]). Les données manipulées sont des fragments de programmes et des informations associées à ces programmes, comme par exemple les commentaires, ces informations portant le nom d'annotations structurées. Les données ont une représentation abstraite unique sous forme d'arbres étiquetés.

Le langage de commande du système est un langage interactif appelé Mentol. Il permet de mettre en oeuvre divers outils de manipulation des représentations abstraites et de développer ainsi un environnement de programmation.

Les outils de base incluent les primitives de repérage et de manipulation (destruction ou insertion de sous-arbres par exemple) et des processeurs spécialisés (en particulier analyseurs et décompilateurs permettant la traduction entre les arbres de la représentation abstraite et le texte source). Les variables de Mentor sont des repères dans les arborescences de programme. Leur nom est de la forme: @<identificateur>. Un repere standard, dit repère courant, est utilisé implicitement dans la plupart des manipulations. Le nom explicite de ce repere est @k. Il est également appelé programme courant ou variable courante.

L'impression d'un sous-arbre est obtenue par la commande P. @1 P N imprime le sous-arbre désigné par la variable @1 avec un niveau de détail indiqué par l'entier N. Le texte du programme est éventuellement abrégé a l'aide des symboles '#' pour les sous-arbres dont l'opérateur de tête est d'arité fixe et '...' pour les listes.

Le langage Mentol permet d'écrire des procédures de manipulation et de transformation d'arbres. L'appel d'une procédure se fait en indiquant son nom précédé d'un point et suivi de ses arguments éventuels entre les symboles '<' et '>'.

Sous Mentor, la sauvegarde des programmes utilise deux types de fichiers. Ceux dont le nom est suffixé par *pascal* sont des fichiers texte traditionnels. Ceux dont le nom est suffixé par *polish* contiennent la forme arborescente du programme.

Nous avons utilisé Mentor dans la version 4.01 appliquée au langage Pascal. Pour utiliser le pré-processeur, il faut se placer au début de la session dans l'environnement pascal-modulaire; celui-ci se compose d'un ensemble de procédures Mentol permettant les transformations de programmes et la gestion d'un catalogue.

2.3- Adaptation des mécanismes

Avant de présenter le pré-processeur, nous allons donner quelques définitions, justifier les adaptations que nous avons dû faire et expliquer le mécanisme de base utilisé dans la plupart des manipulations effectuées sur les programmes.

2.3.1- Définitions

Nous appelons *programme pascal–modulaire* ou *programme modulaire* un programme écrit dans le langage Pascal complété par les mécanismes de modularité que nous avons précédemment décrits.

Un *programme pascal–compilable* ou *programme compilable* est un programme acceptable par le compilateur Pascal disponible sur Multics.

Une *interface cohérente* est une interface dont tous les objets utilisés dans les déclarations sont décrits dans l'interface ou appartiennent à son contexte.

2.3.2- Adaptation

Les objets manipulés sous Mentor, dans la version utilisée, devant être des "morceaux" de programme Pascal standard, nous avons dû adapter les mécanismes présentés dans la première partie de la façon suivante:

La liste *USE* est introduite sous forme d'une annotation structurée, située en tête du programme.

Les interfaces devant avoir la même structure syntaxique qu'un programme pour être acceptables par Mentor, les en-têtes de sous-programmes apparaissant dans les interfaces sont suivies de 'external'.

D'autre part, le compilateur interdit la séparation dans deux programmes distincts de l'en-tête et du corps d'un sous-programme. Il ne permet donc pas la décomposition d'une unité fonctionnelle en deux unités de compilation correspondant à la spécification et la réalisation. Pour pallier cette difficulté, nous avons défini la notion d'interface virtuelle: une interface virtuelle contient des descriptions d'objets (constantes, types, variables) et des en-têtes de sous-programmes dont les corps sont définis ultérieurement dans un autre programme. Une interface virtuelle se présente comme une interface classique. Elle permet de définir une spécification pour une partie d'une application non encore écrite; elle sert ainsi de cahier des charges pour ceux qui doivent réaliser cette spécification et peut également être utilisée pour la compilation d'autres programmes,sans nécessiter que la réalisation soit déjà écrite. L'interface virtuelle et le programme la réalisant sont transformés par le pré-processeur en un seul programme compilable.

2.3.3- La fusion structurelle

L'opération de base du pré-processeur consiste à fusionner deux programmes Pascal. Cette opération est utilisée lors de la création du programme compilable (il faut fusionner les interfaces utilisées dans le programme modulaire). Lorsque l'on veut obtenir une interface sur un ensemble d'interfaces, il faut également fusionner les interfaces considérées. Dans les deux cas, le mécanisme élémentaire de fusion est itéré pour chaque interface.

Pour obtenir un programme syntaxiquement correct par fusion de deux programmes, il faut faire une inclusion structurée de texte. Sous Mentor, cela revient à insérer des sous-arbres. Pour réaliser cette opération, on suppose que l'un des deux programmes a un corps vide, ce qui est légitime car l'un des deux programmes est toujours une interface pour ce qui nous concerne.

Soient P1 et P2 les deux programmes que l'on veut fusionner. On suppose que P1 a un corps vide. Le programme P3 résultant de la fusion de P1 avec P2 est obtenu en dupliquant P2 dans P3, puis en insérant les déclarations de constantes de P1, s'il y en a, au début de la zone de constantes de P3 si elle existe sinon en créant une zone de constantes dans P3 identique a celle de P1. On opère de même avec les déclarations de types, puis celles de variables et finalement avec les sous-programmes. Cette insertion se fait sans modifier l'ordre dans lequel sont déclarés les objets dans chacun des programmes.

Dans le programme P3, les déclarations contenues dans P1 apparaissent toujours avant celles contenues dans P2 dans chacune des quatre zones de constantes, types, variables et sous-programmes.

3– Utilisation du pré–processeur sous Mentor

Pour utiliser le pré-processeur, il faut charger l'environnement pascal-modulaire au début de la session Mentor. On dispose alors des commandes suivantes:

pcat
: imprime le catalogue courant

chcat
: change le catalogue courant

use
: positionne la liste *USE* du programme courant

mcompile
: compile un programme modulaire

crintf
: crée une interface a partir d'un programme ou d'une interface

fusionintf
: crée une interface a partir de plusieurs interfaces

saveintf
: sauvegarde une interface et range son nom dans le catalogue

realize
: permet d'indiquer qu'un programme réalise une interface virtuelle et crée le programme modulaire correspondant

3.1- Le catalogue

A chaque application correspond un catalogue qui contient tous les renseignements sur sa structure, c'est-à-dire les noms des programmes modulaires et des interfaces constituant l'application.

Ces éléments sont rangés automatiquement dans le catalogue lors de l'exécution des commandes de compilation et de création d'interfaces, chaque fois qu'un composant a été

créé ou modifié. Les renseignements contenus dans le catalogue sont utilisés par les commandes constituant le pré-processeur.

A chaque nom d'interface sont associés:
- la liste *USE* de l'interface
- le nom du programme dont elle est l'interface, s'il existe
- éventuellement, le nom des interfaces qu'elle englobe
- la date de la derniere modification.

A chaque nom de programme sont associées:
- la liste *USE* du programme
- la date de la derniere création du programme compilable associé.

A partir des noms que le catalogue contient et des listes *USE,* on peut constituer le graphe des dépendances entre les unités de l'application. Celui-ci est orienté et sans cycle. Cette derniere propriété est garantie par le système.

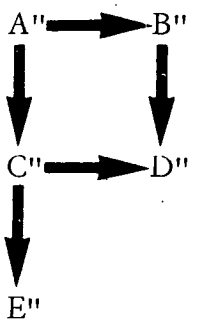
Le nom du catalogue est demandé à l'utilisateur lorsqu'il débute une session.
Pour en changer au cours de la session, il faut utiliser la commande *chcat*.
La commande *pcat* permet d'obtenir la liste des objets contenus dans le catalogue courant. Ces deux commandes sont des appels de procédures écrites en Mentol, ainsi d'ailleurs que toutes les autres commandes que nous décrivons par la suite.

Exemple:

Supposons que le catalogue contienne les programmes A, B, C, D, E et les interfaces A", B", C", D", E" telles que A" (respectivement B", C", D", E") soit une interface sur le programme A (respectivement B, C, D, E).
Supposons que l'on ait les relations de dépendance suivantes entre les interfaces:

B" utilise A"
C" utilise A"
D" utilise C" et B"
E" utilise C"

Alors le graphe représentatif de l'application sera:



Dans cet exemple, une relation telle que A" utilise D" est interdite car elle entraînerait la création d'un cycle dans le graphe des dépendances de l'application. Le système vérifie qu'une telle relation n'apparaît pas.

3.2- Définition du contexte de compilation

La commande *use* permet à l'utilisateur d'indiquer la liste des interfaces INTF1, ..., INTFn qui constituent le contexte du programme courant. Cette liste est indispensable pour compiler un programme modulaire.

La commande *use* vérifie l'existence des interfaces citées en examinant le catalogue et ordonne cette liste de la manière suivante:

> Quelle que soit l'interface INTFi apparaissant dans la liste INTF1, ..., INTFi, ..., INTFn, l'interface INTFi ne référence aucune des interfaces INTF1, ..., INTF(i-1).

Elle vérifie également qu'aucun cycle n'est créé dans le graphe de l'application par cette nouvelle relation de dépendance entre les unités de compilation.

Elle insere finalement la liste ordonnée USE(INTF1,..,INTFn) sous forme d'une annotation structurée en tête du programme courant. Cette annotation apparait comme un commentaire dans le texte du programme décompilé.

3.3- Compilation d'un programme modulaire

La commande *mcompile* permet de compiler un programme modulaire. Pour cela, elle se décompose en trois étapes:

- production d'un programme compilable a partir d'un programme modulaire
- compilation de ce programme
- mise a jour du catalogue.

Un exemple est donné en annexe.

Pour générer le programme compilable, il faut connaître tous les objets importés et établir la liste d'externes correspondant dans le programme compilable. Cela se fait a partir de la

liste *USE* du programme modulaire. Il faut aussi connaître les objets exportés afin de pouvoir précéder leur déclaration de l'attribut 'def' dans le programme compilable. Ces objets exportés sont ceux de l'interface associée au programme.

Il est donc nécessaire de construire l'interface sur le programme avant de créer le programme compilable associé. Pour ce faire, deux solutions sont envisageables.

On peut exporter systématiquement tous les objets déclarés dans le programme, en particulier toutes les variables et tous les sous-programmes. Cette solution ne tient pas compte des contraintes apportées par le compilateur. En effet, toutes les variables ne sont pas exportables (par exemple les indices de boucles).

Dans la solution retenue, on exporte sélectivement certains objets. Cette opération se fait à l'aide d'une interface réduite sur le programme.

La phase préliminaire de la commande *mcompile* consiste à créer une interface sur le programme. Elle fait appel a la commande *crintf* qui permet de construire une interface réduite. Cette opération est décrite en détails dans la section suivante.

A partir du programme P, de la liste des interfaces qu'il utilise et de l'interface I qui vient d'être construite, le programme compilable IC correspondant peut maintenant être créé. Pour cela, il faut fusionner le programme P et les interfaces IN1F1, ..., IN1Fn (qu'il référence dans la liste *USE* préalablement ordonnée). Cette opération se fait en insérant l'interface IN1F1 dans une copie du programme P, puis IN1F2 dans le programme résultant de la précédente fusion, puis IN1Fn. L'insertion se fait zone par zone et elle produit un programme dont les déclarations sont dans un ordre cohérent.

Afin de respecter les mécanismes de compilation séparée, deux modifications doivent être faites sur le programme obtenu. D'une part, on indique que les déclarations que l'on a insérées correspondent à des objets importés en mettant leurs noms dans la liste d'externes du programme créé. D'autre part, les objets laissés dans l'interface I sont exportés, ce qui se fait en ajoutant l'attribut 'def' devant leur déclaration. On obtient ainsi le programme IC qui est alors compilé.

Le nom du programme P et celui de l'interface I sont rangés dans le catalogue, ainsi que la liste *USE* de chacun d'eux et la date à laquelle la compilation a eu lieu.

Si le programme P a déjà été compilé une premiere fois, la commande *mcompile* fonctionne légerement différemment. L'utilisateur doit indiquer s'il veut créer une nouvelle interface (dans ce cas, tout ce passe comme précédemment) ou bien réutiliser une ancienne interface sur ce programme. Dans ce dernier cas, il doit donner le nom de cette interface. S'il ne la modifie pas, le systeme vérifie la validité de l'ancienne interface sur le programme. Sinon, le systeme fournit une liste des programmes qui utilisent cette interface et qu'il faut donc normalement recompiler si elle a été modifiée.

3.4- Opérations sur les interfaces: réduction et fusion

3.4.1- Réduction d'une interface

Nous décrivons ici la construction d'une interface globale à partir d'un programme modulaire et sa réduction.

L'interface globale d'un programme modulaire est construite de la manière suivante: le corps du programme est supprimé, les zones "label" et "value" sont enlevées, les corps des procédures et des fonctions déclarées au plus haut niveau sont remplacés par 'external'. L'interface globale contient la description de tous les objets déclarés dans le programme.

La cohérence de l'interface globale est alors vérifiée. Cette opération revient à vérifier que tous les objets utilisés dans l'interface sont soit entièrement définis dans celle-ci, soit définis dans le contexte de compilation de l'interface. Il suffit en fait de le vérifier pour toutes les constantes et tous les types utilisés dans l'interface.

L'interface globale du programme peut ensuite être réduite. Cette opération se fait interactivement, en conservant la cohérence de l'interface, de la manière suivante.

Le système demande à l'utilisateur s'il veut laisser toutes les déclarations de sous-programmes, sinon il examine chaque déclaration et l'enlève si l'utilisateur le désire. Le processus se répète pour les déclarations de variables, puis celles de types et de constantes. Pour les types et les constantes, le systeme interdit la suppression d'un objet utilisé dans des déclarations déjà conservées dans l'interface.

Une telle réduction est possible soit au cours de la commande *mcompile,* soit à partir d'une interface, par la commande *crintf.*

La commande *crintf* crée une interface à partir d'un programme modulaire ou d'une interface.

Si elle est appliquée à un programme (ce qui est le cas lors de l'appel implicite par la commande *mcompile),* elle crée l'interface globale et la réduit interactivement. Appliquée à une interface qui figure déjà dans le catalogue, elle crée une autre interface par réduction. Cette interface peut alors être utilisée par un autre programme, sans avoir à recompiler le programme dont elle est extraite.

On peut ainsi obtenir un ensemble d'interfaces sur un même programme, par réductions successives, à partir d'une interface-mère créée lors de la commande *mcompile.* Chaque interface dérivée est un sous-ensemble cohérent formé d'objets de l'interface-mère. Toute modification de l'interface-mère invalide les autres interfaces.

La liste *USE* de l'interface réduite est calculée a partir de celle de l'interface dont elle est issue. Si une interface n'est plus effectivement utilisée par la nouvelle interface, alors son nom est ôté de la liste.

Finalement, l'utilisateur doit donner un nom à l'interface ainsi créée.

Exemple de création d'interface, extrait d'une session Mentor:

L'interface que l'on veut réduire est la suivante:

```
program PPILE_INTF;
     const
          MAX      =20;
     type PILE     =
                record
                     P:array[1..MAX]of INTEGER;
                     TOP:INTEGER
                end;

          procedure EMPILER(var PIL:PILE;ITEM:INTEGER);external;

          procedure DEPILER(var PIL:PILE);external;

          function SOMMET(var PIL:PILE):INTEGER;external;

          function PILENONVIDE(var PIL:PILE):BOOLEAN;external;

          procedure INIT(var PIL:PILE);external;
```

?.crintf
Voulez-vous laisser tous les sous-programmes? [line]:non

Voulez-vous enlever tous les sous-programmes? [line]:non

Voulez-vous laisser EMPILER ? [line]:oui

Voulez-vous laisser DEPILER ? [line]:oui

Voulez-vous laisser SOMMET ? [line]:non

Voulez-vous laisser PILENONVIDE ? [line]:oui

Voulez-vous laisser INIT ? [line]:non

Voulez-vous laisser tous les types? [line]:non

Voulez-vous enlever tous les types? [line]:oui

Le type PILE est automatiquement conserve

Voulez-vous laisser toutes les constantes? [line]:non

Voulez-vous enlever toutes les constantes? [line]:oui

La constante MAX est automatiquement conservee

Quel nom voulez-vous donner a cette interface?
[ident]:ppile_intf2

L'interface réduite est alors:

```
program PPILE_INTF2;
     const
          MAX      =20;
     type PILE     =
                record
                     P:array[1..MAX]of INTEGER;
                     TOP:INTEGER
                end;

     procedure EMPILER(var PIL:PILE;ITEM:INTEGER);external;

     procedure DEPILER(var PIL:PILE);external;

     function PILENONVIDE(var PIL:PILE):BOOLEAN;external;
```

Lors de la réduction de l'interface, la déclaration du type PILE est automatiquement gardée car il est utilisé dans les déclarations des sous-programmes qui sont conservés, de même pour la constante MAX.

La commande *crintf* est ici présentée dans la version destinée à l'apprentissage. Dans la version usuelle, il suffit d'indiquer si l'on veut garder toutes les déclarations ou au contraire toutes les enlever, ou bien de donner une liste des objets que l'on veut conserver.

3.4.2- Fusion de plusieurs interfaces

Une autre méthode pour définir une interface est de créer une interface sur un ensemble d'interfaces. Cette interface peut être soit une réunion de toutes les interfaces considérées, soit une sélection cohérente d'objets de ces interfaces. Elle est obtenue par une fusion des interfaces considérées, suivie éventuellement d'une réduction.

La commande *fusioninf* demande la liste des interfaces que l'on veut englober et le nom de l'interface que l'on va construire. Cette liste est alors ordonnée de la même façon que dans la commande *use* afin d'assurer la cohérence globale de l'interface résultante. Cette interface est obtenue en fusionnant une à une les interfaces de la liste ordonnée, suivant le mécanisme précédemment décrit. La liste *USE* de l'interface résultante est calculée à l'aide des renseignements contenus dans le catalogue. L'interface peut ensuite être réduite interactivement si l'utilisateur ne veut garder que certains objets de chaque interface englobée.

3.5- Interface virtuelle

Dans ce qui précede, les interfaces étaient construites à partir de programmes pré-existants. On peut aussi faire l'inverse en commençant par spécifier l'interface.

Une *interface virtuelle* contient des descriptions d'objets qui seront déclarés dans un programme écrit ultérieurement. Les objets décrits dans l'interface sont considérés comme des déclarations du programme associé: il est ainsi inutile de les redéclarer dans le programme. Avec ce mécanisme, il est maintenant possible, en Pascal-Modulaire, de séparer l'en-tête et le corps d'un sous-programme, ce qu'interdit le compilateur Pascal. La séparation entre spécification et réalisation peut donc être effectuée.

Pour réaliser cette opération, il faut conserver l'interface virtuelle IV que l'on vient d'écrire et rentrer son nom dans le catalogue a l'aide de la commande *saveintf*.

Si le programme PR réalise l'interface IV, la commande *realize* appliquée à ce programme vérifie qu'a chaque description de sous-programme dans l'interface virtuelle correspond bien la déclaration complete dans le programme PR. Elle permet alors d'obtenir un programme modulaire PM dont l'interface effective est identique a l'interface virtuelle IV.

Exemple:
Soit PILEDEFINITION une interface virtuelle:

```
program PILEDEFINITION;
     const
          N       =28;
     type PILE    =
               record
                    P:array[1..N]of INTEGER;
                    TOP:INTEGER
               end;

     procedure EMPILER(var PIL:PILE;ITEM:INTEGER);external;

     procedure DEPILER(var PIL:PILE);external;

     function SOMMET(var PIL:PILE):INTEGER;external;

     function PILENONVIDE(var PIL:PILE):BOOLEAN;external;

     procedure INIT(var PIL:PILE);external;
```

Voici un programme réalisant cette interface virtuelle:

```
program PILEREALISATION;
```

```
procedure EMPILER;
     begin
     #
     end;

procedure DEPILER;
     begin
     #
     end;

function SOMMET;
     begin
     #
     end;

function PILENONVIDE;
     begin
     #
     end;

procedure INIT;
     begin
     #
     end;
```

A partir de l'interface virtuelle et du programme la réalisant, on obtient le résultat suivant par fusion de l'interface avec le programme et en ayant substitué dans chaque déclaration de sous-programmes contenues dans l'interface, 'external' par 'forward'.

```
program PILE_R;
     const
          N       =28;
     type PILE    =...;

     procedure EMPILER(#;#);forward;

     procedure DEPILER(#);forward;

     function SOMMET(#):INTEGER;forward;

     function PILENONVIDE(#):BOOLEAN;forward;

     procedure INIT(#);forward;

     procedure EMPILER;
          begin
          #
          end;

     procedure DEPILER;
          begin
          #
          end;
```

```
function SOMMET;
     begin
     #
     end;

function PILENONVIDE;
     begin
     #
     end;

procedure INIT;
     begin
     #
     end;
```

Dans le programme PILEREALISATION réalisant les spécifications décrites dans l'interface PILEDEFINITION, nous ne faisons aucune hypothèse quant à l'ordre dans lequel apparaissent les sous-programmes. Il peut y avoir des références croisées, sans que celles-ci soient indiquées à l'aide de 'forward'. Ce choix a une répercussion sur le programme résultant de la fusion: tous les sous-programmes sont déclarés en deux parties, la première correspond à la spécification, l'autre à la réalisation. La réorganisation des déclarations, qui ne laisserait que les 'forward' indispensables, est une opération coûteuse que nous n'effectuons pas.

Deux méthodologies de programmation sont laissées au choix de l'utilisateur: soit conserver le programme qu'il a écrit, réalisant la spécification, soit le remplacer par celui résultant de la fusion.

Dans le premier cas, il faut indiquer que le programme réalise une interface et mettre le nom de l'interface dans la liste *USE* du programme. La commande *mcompile* appliquée à un tel programme appelle préalablement la commande *realize*.

L'autre solution revient à oublier le passage par une interface virtuelle et à faire comme si l'interface avait été extraite a posteriori du programme, dès lors que l'on a effectué la commande *realize*. Dans ce cas, on s'éloigne du texte de l'utilisateur, car on le remplace par le programme résultant de la fusion.

Conclusion

Les mécanismes présentés respectent la structure du Pascal standard auquel ils apportent très peu de modifications syntaxiques ou sémantiques. Ils permettent de garantir la vérification de cohérence pour toute l'application (sans nécessiter une description multiple des objets partagés, source d'erreurs) et de mieux maîtriser sa structure par la connaissance des dépendances entre composants.

Ces mécanismes peuvent être facilement inclus dans un compilateur ou traité par un pré-processeur. C'est ce que nous avons nous-même fait en utilisant des outils (compilateur, Mentor) appartenant à l'environnement Pascal standard sous Multics.

Par ailleurs, ces mécanismes ne sont pas coercitifs et aucune méthodologie de programmation n'est, a priori, sous-jacente. Ainsi, rien n'oblige à structurer une application de manière arborescente ou à effectuer une décomposition systématique entre spécification et réalisation même si ces méthodologies sont les plus utilisées. Ils offrent donc une grande souplesse au programmeur pour le découpage de son application et la construction des programmes.

Annexe

Exemple de session Mentor

Les commentaires sur le déroulement de la session sont rajoutés en italiques dans le texte.

```
mentor
- MENTOR VERSION 4.1 / JAN-82 ON MULTICS (IREP)
User name:valerie
VALERIE.prelude FILE LOADING
Donner le nom du catalogue dans lequel vous voulez travailler:
[ident]:catalog
Le catalogue est vide.
?
```

Nous ne décrivons pas l'étape d'édition du programme sous Mentor.

Voici le programme modulaire que l'on veut compiler, repéré par la variable courante

```
?p*

program PPILE;
    const
        MAX=20;
    type PILE=
            record
                P:array[1..MAX]of INTEGER;
                TOP:INTEGER
            end;

    procedure EMPILER(var PIL:PILE;ITEM:INTEGER);
        begin
        if PIL.TOP<MAX then
            begin
            PIL.TOP:=PIL.TOP+1;
            PIL.P[PIL.TOP]:=ITEM
            end
        end;

    procedure DEPILER(var PIL:PILE);
        begin
        if PIL.TOP#0 then PIL.TOP:=PIL.TOP-1
        end;
```

```
function SOMMET(var PIL:PILE):INTEGER;
    begin
    if PIL.TOP#0 then SOMMET:=PIL.P[PIL.TOP]
    else SOMMET:=-1234567890
    end;

function PILENONVIDE(var PIL:PILE):BOOLEAN;
    begin
    if PIL.TOP=0 then PILENONVIDE:=FALSE
    else PILENONVIDE:=TRUE
    end;

procedure INIT(var PIL:PILE);
    begin
    PIL.TOP:=0
    end;
```

Il faut maintenant indiquer qu'il n'utilise aucune interface.

```
?.use
[lexp]:;
```

L'en-tête du programme devient:

```
?1p

(*USE*)
program PPILE;
```

On peut alors compiler ce programme.

```
?.mcompile
PPILE.polish   FILE CREATED
PPILE.pascal   FILE CREATED
Voulez-vous reduire l'interface globale? ...oui ou non?
[line]:non
Quel nom voulez-vous donner a l'interface? [ident]:ppile_i
lline
PPILE_I.polish   FILE CREATED
PPILE_I.pascal   FILE CREATED
CATALOG.polish   FILE CREATED
lline
PPILE_IC.polish   FILE CREATED
PPILE_IC.pascal   FILE CREATED
PASCAL 6.02
```

L'interface du programme est repérée par la variable Mentor @interface

```
?@interface p*

program PPILE_I;
      const
            MAX=20;
      type PILE=
                  record
                        P:array[1..MAX]of INTEGER;
                        TOP:INTEGER
                  end;

      procedure EMPILER(var PIL:PILE;ITEM:INTEGER);external;

      procedure DEPILER(var PIL:PILE);external;

      function SOMMET(var PIL:PILE):INTEGER;external;

      function PILENONVIDE(var PIL:PILE):BOOLEAN;external;

      procedure INIT(var PIL:PILE);external;

?
```

Supposons que l'utilisateur ait écrit le programme suivant:

```
?p*

program MAIN(OUTPUT);
      var   PP1,PP2:PILE;

      procedure PILLE(var PILE1,PILE2:PILE);
            var   I:INTEGER;
            begin
            INIT(PILE1);
            INIT(PILE2);
            for I:=1 to 5 do EMPILER(PILE1,I);
            while PILENONVIDE(PILE1)do
                  begin
                  EMPILER(PILE2,-SOMMET(PILE1));
                  DEPILER(PILE1)
                  end;
            while PILENONVIDE(PILE2)do
```

```
            begin
            WRITELN(SOMMET(PILE2));
            DEPILER(PILE2)
            end
        end;

    begin
    PILLE(PP1,PP2)
    end.
```

Ce programme utilise des objets définis dans le programme PPILE et contenus dans l'interface PPILE_I

```
?.use
[lexp]:(ppile_i);

?lp
(*USE(PPILE_I)*)
program MAIN(OUTPUT);

?.mcompile
MAIN.polish   FILE CREATED
MAIN.pascal   FILE CREATED
Voulez-vous reduire l'interface complete? ...oui ou non?
[line]:non
Quel nom voulez-vous donner a l'interface? [ident]:main_i
lline
MAIN_I.polish   FILE CREATED
MAIN_I.pascal   FILE CREATED
CATALOG.polish   FILE OVERWRITTEN
lline
MAIN_IC.polish   FILE CREATED
MAIN_IC.pascal   FILE CREATED
PASCAL 6.02
```

Le programme compilable obtenu est le suivant:

```
?@compilable p*

program MAIN_IC(OUTPUT,EMPILER,DEPILER,
                SOMMET,PILENONVIDE,INIT);
    const
        MAX=20;
```

```
type PILE=
        record
            P:array[1..MAX]of INTEGER;
            TOP:INTEGER
        end;
var  def PP1,def PP2:PILE;

procedure EMPILER(var PIL:PILE;ITEM:INTEGER);external;

procedure DEPILER(var PIL:PILE);external;

function SOMME1(var PIL:PILE):INTEGER;external;

function PILENONVIDE(var PIL:PILE):BOOLEAN;exernal;

procedure INIT(var PIL:PILE);external;

procedure def PILLE(var PILE1,PILE2:PILE);
    var  I:INTEGER;
    begin
    INIT(PILE1);
    INIT(PILE2);
    for I:=1 to 5 do EMPILER(PILE1,I);
    while PILENONVIDE(PILE1)do
        begin
        EMPILER(PILE2,-SOMME1(PILE1));
        DEPILER(PILE1)
        end;
    while PILENONVIDE(PILE2)do
        begin
        WRITELN(SOMME1(PILE2));
        DEPILER(PILE2)
        end
    end;

begin
PILLE(PP1,PP2)
end.
```

Le contenu du catalogue est le suivant:

```
?.pcat
PPILE_I
    03/03/83   16:00
    USE
    interface sur le programme: PPILE
```

MAIN_I
03/03/83 16:10
USE (PPILE_I)
interface sur le programme: MAIN

Bibliographie

Ada 80

Reference Manual for the Ada Programming Language, *Proposed Standard Document* Honeywell, Inc & CII-Honeywell Bull - Juillet 80

Afn 82

Le langage de programmation Pascal, *Spécifications et mise en oeuvre* Normes et techniques - AFNOR - SOL - 1982

Bou 81

J-L.Bouchenez, M.Loyer, F.Prusker, J-C.Sogno, *Legos: un langage modulaire* INRIA - Rapport technique no 1 - Avril 1981

Car 82

L.R.Carter, D.C.Robbins, *An extension to Pascal to provide for separate compilation and encapsulation* Proposition ANSI X3J9/82-073

Cas 81

P.M.Cashin, M.L.Joliat, R.F.Kamel, D.M.Lasker, *Experience with a Modular Typed Language: Protel* 5th Intl.Conf. on Software Engineering - Mars 1981

Don 79

V.Donzeau-Gouge, G.Huet, G.Kahn, B.Lang, *Introduction au système Mentor et a ses applications* Actes des journées Francophones sur la Certification du Logiciel - Geneve - Janvier 1979 -

Don 80

V.Donzeau-Gouge, G.Huet, G.Kahn, B.Lang, *Programming Environnements based on Structured Editors: The Mentor Experience* INRIA - Rapport de Recherche no 26 - Juillet 1980

Don 81

V.Donzeau-Gouge, G.Huet, G.Kahn, B.Lang, *The Mentor Program Manipulation System* Documentation Multics de l'INRIA - 1981

Fel 77

S.I.Feldman, *Make – A Program for Maintaining Computer Programs* Bell Laboratories CSTR 57 - 1977

Iso 82

ISO Norme Pascal, *Specification for Computer Programming Language Pascal* ISO/DIS 7185

Jen 78

K.Jensen, N.Wirth, *Pascal User Manual and Report* Springer Verlag - 1978

Loy 81

M.Loyer, *Modularité et compilation séparée: Les choix du langage Legos* Thèse de docteur-ingénieur, Université P.Sabatier Toulouse 1981

Mel 80

B.Mélese, *Manipulation de programmes Pascal au niveau des concepts du langage* These de 3eme cycle, Université Paris 11, juin 1980

Mel 81

B.Mélese, *Mentor: L'environnement Pascal* INRIA - Rapport Technique no 5 - Octobre 1981

Mel 83

B.Mélese, *Mentor Rapport – Manipulation de textes structurés sous Mentor* INRIA - Janvier 1983

Mit 79

J.G.Mitchell, W.Maybury, R.Sweet, *Mesa Language Manual – Version 5.0* Xerox Palo Alto Research Center - April 1979

Pas 81

>am>PASCAL>info>pascal.gi.info, *Grenoble University Multics PASCAL Compiler* Documentation Multics de l'INRIA - 1981

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique