A D A

9X

Version 2.2 (10/93)

S O M M A I R E

I. INTRODUCTION

A. HISTORIQUE

Vers 1970 le coût du logiciel commença à dépasser celui du matériel avec la part du lion pour la maintenance. Le DoD, département de la défense américaine dépensait 3 milliards de dollars en logiciel et recensa parmi les applications livrées 450 langages et de 500 à 1500 outils de génération différents.

A partir de 1975 le DoD fit circuler le cahier des charges du langage nécessaire à tous ses besoins, et particulièrement les applications embarquées. En 1979 le langage "vert" de CII-Honeywell-Bull était vainqueur et se voyait attribuer le nom de la comtesse Ada Byron. En 1983 le langage était normalisé ANSI, puis par ISO en 1987.

Actuellement, une nouvelle version du langage ADA 9X est en préparation. Elle apporterait des extensions dans de nombreux domaines :

- orienté objet
- Temps réel
- Mathématique
- Gestion
- ....

En fait chaque domaine constituerait une option, que les compilateurs respecteraient ou non suivant la clientèle visée. Avant même la sortie officielle de la normalisation, ALSYS annonce sortir en 1994 une première version avec les extensions orientées objets telles qu'elles sont définies dans les draft.

B. PRESENTATION GENERALE

ADA 9X incorpore les trois améliorations suivantes :

- Orientation objets
- Hiérarchie des bibliothèques pour les gros projets.
- Amélioration du multitâches.

D'autres améliorations sont prévues, mais elles apparaîtront sous la forme de paquetages optionnels, qu'un fournisseur est libre de traiter ou non suivant le type de clientèle qu'il vise :

- Programmation système (assembleur inclus, interruptions, variables partagées, identification des tâches)
- Temps réel (priorités dynamiques, séquencements, files, nécessite la programmation système)
- Systèmes distribués (partition dynamique ou statique entre plusieurs processeurs/systèmes)
- Interface aux autres langages (C ou COBOL) Nécessaire pour les informations systèmes
- Informations systèmes (valeurs décimales et conversions en chaînes de représentation)
- Sécurité (Restrictions à apporter en environnement critiques)
- Mathématique (Modèles numériques)

II. AMELIORATIONS DES ANCIENS CONCEPTS

A. NOUVEAUX TYPES

1. CARACTERE

- Le type CHARACTER utilise à présent le codage 8 Bits ASCII.
- Le nouveau type WIDE_CHARACTER utilise un codage 16bits.

Les types STRING et WIDE_STRING en dérivent directement.
Des paquetages apportent des manipulations de chaînes :

- De taille fixe.
- De taille logique dynamique mais majorée à une taille physique fixée d'avance.
- De taille physique et logique dynamique

2. ENTIER

Les entiers (signés) sont enrichis par un nouveau type **MOD** non signé.

Type BYTE is MOD 256**;** -- 0..255

Un paquetage de génération de nombres aléatoires est défini.

3. REELS FIXES

- Notation traditionnelle : **TYPE** Volt **is delta** 0..125 **range** 0.0..255.0;
- Nouvelle notation : **TYPE** Money **is delta** 0.01 **digits** 15;

4. REELS FLOTTANTS

Les paquetages mathématiques sont définis (Generic_Elementary_Functions)

B. DEFINITION INCOMPLETE DE TYPE

De façon à permettre les références récursives dans les **record** (pointeur vers le type), la déclaration
type Person(<>);
permet une référence avant la déclaration du pointeur.

C. PARAMETRES OUT

Un paramètre de type out peut être lu.

```
PROCEDURE COMPTAGE( RESULTAT OUT INTEGER)
BEGIN
    RESULTAT :=0;
    LOOP
        IF condition THEN
            RESULTAT := RESULTAT + 1;
        END IF;
    END LOOP;
END COMPTAGE;
```

D. ENTREES SORTIES

Les entrées sorties comporte un nouveau type, le fichier interne en mémoire :

```
with Ada.IO_EXCEPTIONS;
with System.Storage_Elements;
generic
```

```
        type Element_Type is private;
package Ada.Storage_IO is
        Buffer_Size : constant System.Storage_Elements.Storage_Count := implementation-defined
        subtype Buffer_Type is System.Storage_Elements.Storage_Array[1..Buffer_Size);
        procedure Read(Buffer : in Buffer_Type; Item : out Element-Type);
        procedure Write(Buffer : out Buffer_Type; Item : in  Element_Type);
        Data_Error : exception renames IO_Exceptions.Data_Error;
private
end Ada.Direct_IO;
```

et également un type stream

```
with Ada.EXCEPTIONS;
with System.Storage_Elements;
with Ada.Tags
package Ada.Stream is
        pragma Pure(Streams);
        type Root_Stream_Type is abstract tagged limited private;
        procedure Read(
                Stream :  in out  Root_Stream_type;
                Item : out System.Storage_Elements.Storage_Array;
                Last : out System.Storage_Elements.Storage_offset) is abstract;

        procedure Write(
                Stream :  in out  Root_Stream_type;
                Item : in System.Storage_Elements.Storage_Array;
                ) is abstract;
private
end Ada.Stream;
```

Des sous-paquetages apportent des accès fichiers séquentiels, à accès direct.
En fait, ce nouveau type (stream) est applicable pour toute communication (réseau, ...)

Un type comporte des fonctions attributs, *matrice'***read** et *matrice'***write** sont les fonctions de lecture et d'écriture pour un type donné sur un nouveau type de fichier (stream). Il est possible de les surcharger par ses propres procédures par les déclarations :
for matrice'**read use** my_matrice_read;
for matrice'**write use** my_matrice_write;

E. PARAMETRES FONCTIONS

On reproche à ADA 83 par rapport au langage PASCAL de ne pas permettre le passage en paramètre de fonctions. Le concept est supporté en ADA 9X via les pointeurs de fonction :

```
Type Integree is access function(X:Float) return Float;
```

Un tel type peut alors servir de paramètre à une fonction :

```
Function Integrer( F : Integree; Debut,Fin : Float;
                precision : Float := 1.0e-7);
```

L'attribut **Access** permet à l'appel d'indiquer la fonction paramètre :

```
Integrer( Log'Access, 1.0, 2.0)
```

Dans le corps de la fonction *Integrer*, la fonction *F* sera appelée par la syntaxe habituelle pour les pointeurs en ADA :

```
F.all(X);
```

F. POINTEURS DE VARIABLES STATIQUES

Un pointeur de type pointeur qualifié de **all** peut être initialisé à l'adresse de toute variable statique qualifiée de **aliased**;

```
type Pointeur_entier  is access all integer;
Pointeur : Pointeur_entier;
I : aliased Integer;
...
Pointeur := I'access;
```

L'accès peut être limité en lecture par la déclaration :

```
type Pointeur_entier  is access constant integer;
Pointeur : Pointeur_entier;
I : constant aliased Integer;
...
Pointeur := I'access;
```

G. PARAMETRES GENERIQUES

Les types de paramètres génériques ont été étendus, aux types déjà existants :

- **type** NOM **is range** <>; -- type entier
- **type** NOM **is digits** <>; -- type réel
- **type** NOM **is private;** -- type privé
- **type** NOM **is limited private**;-- type limité privé

ont été ajoutés :

- **type** NOM **is** (<>) ; type quelconque, les possibilités sont limitées aux références
- **type** NOM **is delta** <> -- type réel fixe
- **type** NOM **is array**( ) of -- type tableau
- **type** NOM **is mod** <> -- type entier non signé
- **type** NOM **is access** <> -- type pointeur
- **type** NOM **is [[abstract] [tagged]][limited] private** -- type privé taggé ou non

H. EXCEPTION

La branche

```
when error : others =>
```

d'une zone exception permet de récupérer dans la variable *error* implicitement définie le type d'erreur, dont les attributs peuvent être récupérés, ex :

```
PRINT("Exception " &
        SYSTEM.EXCEPTION_NAME(ERROR)
        SYSTEM.EXCEPTION_INFORMATION(ERROR));
```

III. LIBRAIRIES HIERARCHIQUES

Au paquetage de base :

```
package Nombre_complexe is
        type Complexe is private;
        function "+" (X,Y : Complexe) return Complexe;
private
...
end Nombre_complexe;
```

Il est ultérieurement possible d'ajouter un sous-paquetage :

```
package Nombre_complexe.Polaire is

        procedure Polaire_to_complexe(R,theta:Float) return Complexe;
private
...
end Nombre_complexe;
```

Pour accéder au sous-paquetage *Nombre_complex.Polaire*, le paquetage client devra utiliser la clause :

```
with Nombre_complexe.Polaire;
```

Cette clause lui donne implicitement accès à *Nombre_complexe* sans nécessiter d'autres clauses with alors que

```
with Nombre_complexe;
```

n'aurait pas donné accès à *Nombre_complexe.Polaire;*
La procédure *Polaire_to_complexe* peut ensuite être appelée par la notation préfixée :

```
C := Nombre_complexe. Polaire_to_complexe(4, 2)
```

sans nécessiter l'introduction du nom du sous-paquetage *Polaire*. Pour un acces direct, une clause

```
use Nombre_complexe;
```

suffit. On peut bien sûr utiliser

```
use Nombre_complexe.polaire;
```

pour limiter l'accès direct aux procédures du sous-paquetage.
Les parties privées et le corps des paquetages enfants ont accès aux parties privées du paquetage parent.

IV. PROGRAMMATION ORIENTEE OBJETS

A. TYPES EXTENSIBLES

1. SELECTION STATIQUE

Un type extensible est signalé par le mot-clef **TAGGED.** Seuls les types **record** et **private** (**record**) peuvent être des types extensibles.

```
Type Forme is tagged
        record
                Centre : Point;
                Couleur : Tcouleur;
        end record;
procedure Affiche ( F:  in Forme) ;
procedure Deplace ( F: in Forme; Vers : Point) is
begin
        Centre := Vers;
        Affiche(F);
end deplace;
```

Il est alors possible de déclarer dans un autre paquetage le type dérivé Carré :

```
Type Carre is new Forme with
        record
                Cote : Float;
        end record;
```

Le type Carre hérite des champs Centre et Couleur du type Carre possède en propre le champ Cote.
Toute fonction ou procédure déclarée dans le même paquetage où est déclaré le type **tagged**, comportant un paramètre ou un résultat de fonction du type **tagged** est utilisable avec le type dérivé. Le type dérivé hérite de toutes les fonctions et procédures du type parent (ex Affiche). Toutefois, si la fonction héritée ne convient pas, il est possible de la surcharger à l'aide de sa propre fonction :

```
procedure Affiche ( R:  in Carre);
```

Un type peu dériver d'un autre type sans contenir de champ propre :

```
Type Forme2 is new Forme with null record;
```

Enfin un type peut également être dérivé à partir d'un type dérivé.

```
Type Rectangle is new Carre with
        record
                Largeur : Float;
        end record;
```

Ce mécanisme correspond à un héritage simple avec sélection statique des méthodes. En effet tout est résolu à la compilation.

2. SELECTION DYNAMIQUE

Dans l'exemple précédant, si la procédure *Deplace* est appelée sur une variable de type *Carre*, la procédure *Deplace* héritée de *forme* sera exécutée. Celle-ci appellera la version *Forme* de la procédure *Affiche* bien que *Carre* ait sa propre version. Le type *carre* du paramètre est oublié dès l'entrée dans la procédure *Deplace*.

L'attribut **Class** permet de déclarer des paramètres dont le type ne sera pas oublié.

```
procedure Deplace ( F: in Forme'Class; Vers : Point) is
begin
        Centre := Vers;
        Affiche(F);
end deplace;
```

Dans cet exemple, Cette procédure est également héritée par tous les types dérivés de *Forme*, mais appliquée à une variable de type *Carre*, la version *Carre* de *Affiche* sera appelée.

Une liste de données hétérogènes peut être gérée à travers un pointeur de classe :

```
type FormePtr is access Forme'Class;
```

B. TYPES ABSTRAITS

Un type peut être déclaré sans champ afin de déclarer un ensemble de procédures et fonctions héritables.

```
Type Forme is abstract tagged null record;
```

Il s'agit d'un type abstrait sur lequel il n'est pas possible de déclarer des variables.
Il est aussi possible de déclarer des procédures et fonctions

```
procedure Affiche ( F:  in Forme) is abstract
```

au niveau du type de base sans définir de code. Le type dérivé devra alors définir le code de la fonction déclarée.

C. PAQUETAGE ADA.TAGS

Le paquetage

```
package Ada.Tags is
        type Tag is dépend du fournisseur;
        function Expanded_Name( T: Tag) return String;
        function External_Tag(T: Tag) return String;
        function Internal_Tag(External : string) return Tag;
end Ada.Tags;
```

permet d'identifier un type.

D. TYPES CONTROLES

La déclaration d'un type contrôlé permet de spécifier des fonctions d'initialisation et des fonctions de destructions appelées automatiquement à la création d'une variable et à sa destruction (ex sortie de procédure pour une variable locale).
Il est possible de redéfinir l'affectation sur ces types contrôlés.

Un type contrôlé s'obtient par héritage du type abstrait système :

```
with System.Finalization_Implementation; -- version 4.0
use System;
package Ada.Finalization is
        type Controlled is abstract
                new Finalization_Implementation.Root_Controlled with null record;
        procedure Initialize(Object : in out Controlled);
        procedure Split(Object : in out Controlled) is abstract;
        procedure Finalize(Object :  in out Controlled) is abstract;
        Root_Part : Finalization_Implementation.Root_Controlled renames Filalization_Implementation.Root_Part;
        type Limited_Controlled is abstract
                new Finalization_Implementation.Root_Limited_Controlled with null record;
        procedure Initialize(Object : in out Limited_Controlled);
        procedure Finalize(Object :  in out Limited_Controlled) is abstract;
end Ada.Finalization;
```

La procédure d'initialisation par défaut ne réalise aucun traitement mais peut être surchargée. Les procédures Finalize et Split sont abstraites et doivent être définies. L'affectation est réalisée par copie binaire puis appel de Split. Le type limité contrôlé ne comporte pas d'affectation.

```
with System.Finalization_Implementation; -- version mai 94
use System;
package Ada.Finalization is
        type Controlled is abstract
                new Finalization_Implementation.Root_Controlled with null record;
```

```
        procedure Initialize(Object : in out Controlled);
        procedure Adjust(Object : in out Controlled);
        procedure Finalize(Object :  in out Controlled);
        Root_Part : Finalization_Implementation.Root_Controlled renames Filalization_Implementation.Root_Part;
        type Limited_Controlled is abstract
                new Finalization_Implementation.Root_Limited_Controlled with null record;
        procedure Initialize(Object : in out Limited_Controlled);
        procedure Finalize(Object :  in out Limited_Controlled);
end Ada.Finalization;
```

Une modification de dernière minute remplacerait la fonction *split* par la fonction *adjust*, et ne proposerait des fonctions par défaut de Adjust et finalyze.

V. MULTI-TACHES

A. DISCRIMINANT DE TACHE

```
task type Keyboad_Driver(ID : Keyboard_ID := New_ID) is
        entry Read(C : out Character);
        entry Write(C : in Character);
end Keyboard_Driver;
Teletype : Keyboard_Driver(TTY_ID);
```

Les paramètres discriminant du type tâche permettront d'indiquer des identifications pour distinguer les tâches entre elles.

B. VARIABLES PROTEGES

```
protected variable is
        function lire return item;
        procedure ecrire(valeur : item);
private
        donnee : Item;
end variable;
```

Seuls les sous-programmes déclarés dans la zone protégée *variables* peuvent accéder à la zone privée en lecture pour les fonctions, en lecture/écriture pour les procédures. Il n'y a pas de tâche associée à *variable* mais garantie d'exclusion mutuelle. Une seule procédure ou fonction peut s'exécuter à un instant donné.

Les sous-programmes s'utilisent à l'aide de la notation préfixée habituelle :

```
variable.ecrire(4);
i:= variable.lire();
```

C. TYPES PROTEGES

On peut aussi déclarer des types protégés et utiliser des barrières analogues aux gardes des tâches.

```
protected type Tampon_Tournant is
        entry Ecrire(X : in Item);
        entry Lire(X : out Item);
private
        A: Item_Array(1..Max);
        I,J : Integer range 1..Max :=1;
        Count : Integer range 0..Max :=0;
end Bounded_Buffer;
```

```
protected body Tampon_Tournant is
        entry Ecrire(X : in Item) when Count<Max is
        begin
                A(I) := X;
                I := I mod Max + 1; Count := Count +1;
        end Ecrire;
        entry Lire(X : out Item) when Count>0 is
        begin
                X := A(J);
                J := J  mod Max + 1; Count := Count-1;
        end Lire;
end Tampon_Tournant;
```

Il n'y a pas de tâche associée au type ou aux variables, mais les demandes sont misent en attente de la même façon que pour les rendez-vous si la condition d'appel n'est pas vraie. Toutefois les conditions ne sont ré-évaluée que lors d'un nouvel appel à une autre entrée du type protégé. Ainsi, dans cet exemple, au début tout appel à *Lire* sera mis en attente. Seuls des appels à Ecrire permettront de ré-évaluer le déblocage des attentes de *Lire*.

Autre exemple avec un discriminant :

```
protected type Semaphore(Nombre_de_ressources : Integer : 1) is
        entry Reserve;
        procedure Libere;
        function Ressources_Libre return Integer;
private
        Compteur : Integer := Nombre_de_ressources;
end Semaphore;
```

```
protected body Semaphore is
        entry Reserve when Compteur >0 is
        begin
                Compteur := Compteur - 1;
        end Reserve;
        procedure Libere is
        begin
                Compteur := Compteur + 1;
        function Ressources_Libres is
        begin
                return  Compteur;
        end Nombre;
end Semaphore;
```

Le discriminant peut aussi être utilisé dans les tâches afin de paramétrer les différentes tâches de ce type.

D. DECLENCHEMENT A HEURE FIXE

```
delay until heure;
```

Ceci permet d'attendre une date indiquée en valeur absolue alors qu'en ADA 83 il n'était possible d'indiquer qu'un délai relatif.

E. TRAITEMENT ABORTABLE

```
select
        delay 5.0;
        Put_line("Temps de calcul trop long");
then abort
        Inversion_de_matrice(X);
end select;
```

Si le traitement de l'inversion de matrice prend plus de 5 secondes, alors ce traitement sera interrompu, et le message d'erreur sera affiché.

```
loop
    select
        Terminal.Wait_For_Interrupt;
        Put_line("Fin du terminal");
    then abort
        Put_line("-> ");
        Get_Line(Command, Last);
        Process_Command(Command(1..Last));
    end select;
end loop
```

L'interprétation des commandes sera arrêtée dès le rendez-vous Wait_For_Interrupt sera traité.

Il est également possible d'indiquer une entrée à la place du délai pour interrompre le traitement sur **entry (**rendez-vous /tâche ou type protégé.

F. INTERRUPTIONS

Les pragmas Interrupt_handler et Attach_Handler permettent d'attacher une procédure (protégée) à une interruption. (Annexe extension système)

G. PRIORITE

Les priorités peuvent être statiquement ou dynamiquement définies via pragma.
(annexe extension temps réel)

H. REQUEUE

La nouvelle instruction **requeue** permet de mettre en attente une demande que l'on ne peut satisfaire ou la rediriger ailleurs.

requeue *entry_name* [**with abort**];

Si l'entrée spécifiée dans l'instruction **requeue** comporte des paramètres, ils doivent être les mêmes que ceux de l'entrée en cours et sont utilisés implicitement sans nécessiter d'être respécifiés.

I. SELECT ET BOUCLES

Il est possible de spécifier par une boucle une liste d'entrées sur un select.

VI. ANNEXES

Ces annexes ne seront pas forcement réalisés par tous les compilateurs. Elles touchent des domaines spécifiques qui n'intéressent pas forcément tous les marchés.

A. PROGRAMMATION SYSTEME

- Accès aux opérations machines (assembleur, accès au système)
- Clauses de représentation
- Support des interruptions
- Définition des pré-élaborations
- Pragmas atomic (indivisible), volatil, atomic_component, volatil_component
- Tâches (identification, attributs,

B. TEMPS REEL

- Définition de priorité
- Ordonnancement des tâches
- Ordonnancement des files
- Priorités dynamiques
- Horloge
- Suspension de tâche

C. SYSTEMES DISTRIBUES

- Partition entre plusieurs processeurs ou tâches
- Catégories de bibliothèques (partagées, distantes)
- Appel de procédures distantes

D. INFORMATIONS

- Représentation machine
- Représentation compacte décimale
- Affichage de nombre

E. NOMBRES

- Arithmétique, mathématiques et entrées/sorties de complexes
- Modèles de flottants
- Modèles de réels fixes

F. SECURITE

- Exécution prévisible (initialisation de toutes les variables, documentation sur la génération de code)
- Aide à la validation du code (documentation dans le code, suivi des données)
- Points d'inspection de données (donne l'association variable / ressources machine)
- Restrictions pour certification (pragmas pour interdire l'usage de telle ou telle construction ADA)

G. INTERFACE AUX AUTRES LANGAGES

- Importation de donnée externe.
- Interfaçage de bibliothèque externe.
- Interfaçage au langage C (y compris chaînes et pointeurs)
- Interfaçage au langage COBOL
- Interfaçage au langage FORTRAN

H. FONCTIONS OBSOLETES

Certaines anciennes syntaxes sont conservées par compatibilité avec ADA 83

VII.ANNEXE : BIBLIOGRAPHIE

Introducing ADA 9X, Ada 9X Project Report, février 93 (rapport)
ADA 9X : A Language adapted to hard real time programm, Marc RICHARD-FOY ALSYS (article)
ADA 9X : Mapping Document Volume 1 Mapping rationnal(mars 92)
ADA 9X : Reference Manual 4.0 15 septembre 93