Ada 95 pour le temps réel et les systèmes distribués

J-P. Rosen
ADALOG
19-21 rue du 8 mai 1945
94110 ARCUEIL

Tel: 01 41 24 31 40
Fax: 01 41 24 07 36
E-m: rosen@adalog.fr

Ce support de cours présente les fonctionnalités qu'offre Ada dans sa version "95" pour les applications temps-réel et distribuées. Un bon nombre de celles-ci étaient déjà présentes dans la version 83, et ont fait l'objet de nombreuses publications; nous les rappelons brièvement ici, et insistons plus sur les nouveautés fournies par la nouvelle version.

I Généralités sur le langage Ada

I.1 Historique

En janvier 1975, le Ministère Américain de la Défense (DOD) a constitué un comité d'experts, le High Order Language Working Group (HOLWG), avec pour mission de trouver une approche systématique aux problèmes de qualité et de coût des logiciels militaires. La plus grosse part de ces logiciels, ou tout du moins là où résidaient les plus gros frais de maintenance, était dans le domaine des systèmes temps-réels embarqués. Développer un langage de programmation universel apparut alors être une solution à beaucoup de ces problèmes, et le HOLWG a produit une succession de cahiers de charges précisant les caractéristiques souhaitables d'un tel langage.

Au printemps de 1977, dix-sept organismes ont répondu à l'appel d'offres du DOD, et après quelques mois, quatre d'entre eux ont été retenus pour une pré-étude. Les propositions furent évaluées de façon anonyme, et ce fut finalement une équipe francaise, dirigée par Jean Ichbiah, qui remporta l'appel d'offre. Le langage fut alors baptisé Ada, du nom d'Ada Augusta Byron, fille de Lord Byron et disciple de Charles Babbage.

Le document proposé en 1979 était en fait beaucoup trop vague pour permettre l'établissement d'un standard rigoureux. Une première version révisée fut produite en Juillet 1980, dont la qualité était au moins aussi bonne que celle de la plupart des normes des autres langages de programmation. Mais pour Ada, il fallait faire beaucoup mieux! L'exigence de portabilité nécessitait une définition beaucoup plus précise, et il fallut encore deux ans de discussion entre experts venus du monde entier pour parvenir à un nouveau document en Juillet 1982. Après quelques modifications de moindre importance, le langage fut standardisé par l'ANSI[1] en 1983.

La standardisation internationale exigeait que la norme soit déposée simultanément en deux des trois langues officielles de l'ISO[2] qui sont l'anglais, le français, et le russe.La normalisation internationale était donc suspendue à la parution de la norme française. Il fallut attendre 1987 pour obtenir une norme française satisfaisante [Afn87] et la standardisation ISO. Ces documents constituent la seule définition

[1] American National Standard Institute

[2] Organisation Internationale de Normalisation / International Organisation for Standardization / МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ. Noter que le sigle "ISO" n'est l'acronyme de son nom dans aucune de ses langues officielles!

officielle du langage. Ada est également une norme européenne (CEN 28652), allemande (DIN 66268), suédoise (SS 63-6115), tchèque, japonaise...

Il était important de définir rigoureusement le langage; encore fallait-il être sûr que les compilateurs respectent la norme! pour cela, le DOD a déposé le nom "Ada", et l'autorisation d'utiliser ce nom pour un compilateur fut subordonnée à la confrontation avec succès du compilateur à une suite de validation. Le premier compilateur Ada ainsi validé fut Ada/ED, réalisé par l'équipe de New York University. De nombreux compilateurs ont été validés depuis[3]. Plus récemment, le DoD a abandonné ses droits sur la marque, considérant que cette contrainte n'était plus nécessaire. Il a, en revanche, déposé une marque de certification utilisable uniquement par les compilateurs validés. La notion de validation est maintenant tellement liée au langage Ada qu'il n'existe plus de compilateur non validé.

Il est de règle, 5 ans après la parution d'une norme, de soit confirmer le standard tel quel, soit de décider de mettre en route une révision. La décision fut donc prise en 1988 de démarrer le processus de révision de la norme. Le ministère américain de la Défense (DoD) a une fois encore fourni le support financier nécessaire, et la révision a été conduite par l'*Ada 9X Project Office*, dirigé par Chris Anderson.

Dans la grande tradition d'ouverture d'Ada, un appel fut lancé aux utilisateurs pour connaître leurs desiderata. La réponse fut très importante, et une synthèse en fut tirée, regroupant par thème les principales tendances des demandes, et leur attribuant un degré de priorité. De cette synthèse, une équipe (le MRT, *Mapping Revision Team*) a tiré un document destiné à décrire les transformations nécessaires pour satisfaire les points identifiés à l'étape précédente. Ce document était encore très touffu et comportait quasiment tout ce qu'il était *envisageable* de modifier, c'est à dire nettement plus que ce qu'il était *souhaitable* de modifier. Sur la base de ce document, une première réunion du groupe de travail de l'ISO s'est tenue au printemps 1992. Cette réunion marathon (50 participants de 12 pays[4] travaillèrent ensemble de façon quasi-ininterrompue, pendant une semaine, de 9h du matin à 11h du soir) permit d'aboutir à un accord unanime sur les éléments qu'il convenait de changer dans le langage. D'autres réunions internationales, à Salem puis à Boston (USA), ont permis de préciser les détails de la révision. Une dernière réunion au mois de mars 1994 à Villars (Suisse) a permis de figer définitivement le langage. Les documents ne subirent alors plus que des changements éditoriaux pendant que le processus bureaucratique suivait son chemin à l'ISO, pour aboutir à l'approbation simultanée par l'ANSI et l'ISO au tout début de 1995.

I.2 Objectifs du langage

Contrairement à la plupart des langages de programmation, Ada n'est pas le fruit des idées personnelles de quelque grand nom de l'informatique, mais a été conçu pour répondre à un *cahier des charges* précis, dont l'idée directrice était de diminuer le coût des logiciels, en tenant compte de tous les aspects du cycle de vie. Le langage est donc bâti autour de quelques idées-force:

- *Privilégier la facilité de maintenance sur la facilité d'écriture:* le coût de codage représente environ 6% de l'effort de développement d'un logiciel; la maintenance représente plus de 60%.
- *Fournir un contrôle de type extrêmement rigoureux:* Plus une erreur est diagnostiquée tôt, moins elle est coûteuse à corriger. Le langage fournit donc des outils permettant de diagnostiquer beaucoup d'erreurs de cohérence dès la compilation. Des contrôles nombreux à l'exécution permettront de diagnostiquer les erreurs dynamiques.
- *Offrir un support aux méthodologies de programmation:* le langage doit faciliter la mise en oeuvre des méthodes du génie logiciel, sans exclusivité et sans omission.
- *Offrir un support à une industrie du composant logiciel:* en fournissant des interfaces standard et des garanties de portabilité des applications indépendamment des machines, Ada permet la création d'entreprises spécialisées en composants logiciels; d'autres entreprises réalisant alors des produits finis (applications) par assemblage de composants logiciels.

[3] En décembre 1993, on recensait 693 compilateurs validés par 53 companies.

[4] Allemagne, Belgique, Canada, Grande-Bretagne, Espagne, France, Japon, Pays-Bas, Russie, Suède, Suisse, USA

■ *Permettre une programmation intrinsèquement sûre:* ceci signifie qu'un programme doit être capable de traiter *toutes* les situations anormales, y compris celles résultant d'erreurs du programme (auto-vérifications, fonctionnement en mode dégradé).
■ *Etre portable entre machines d'architecture différentes:* Les programmes doivent donner des résultats identiques, ou au moins équivalents, sur des machines différentes, y compris en ce qui concerne la précision des calculs numériques.
■ *Permettre des implémentations efficaces et donner accès à des interfaces de bas niveau:* exigence indispensable à la réalisation notamment de systèmes "temps réel".

I.3 Syntaxe de base

Ada est un langage algorithmique d'une puissance d'expression considérable, dérivé de Pascal dont il a retenu les structures de contrôle et certains types de données, mais avec en plus des possiblités d'encapsulation de données, de modularité, de modélisation de tâches parallèles, et de traitement des situations exceptionnelles.

Le programme Ada ci-dessous est un premier exemple qui imprime "Bonjour", "Bon après-midi", ou "Bonsoir" en fonction de l'heure de la journée.

```
with TEXT_IO; use TEXT_IO;
with CALENDAR; use CALENDAR;
procedure BONJOUR is
   Heure : DAY_DURATION;
begin
   Heure := SECONDS(CLOCK)/3600;
   if Heure < 12.00 then
      PUT_LINE("Bonjour");
   elsif Heure < 19.00 then
      PUT_LINE("Bon après-midi");
   else
      PUT_LINE("Bonsoir");
   end if;
end BONJOUR;
```

Les clauses **with** et **use** en tête de la procédure permettent d'utiliser deux paquetages (**package**) qui fournissent respectivement l'accès aux fonctionnalités d'entrées-sorties et à l'utilisation du temps.

Sous-programmes et paquetages constituent des unités de compilation, c'est à dire des structures pouvant être compilées séparément. Un programme complet est donc constitué d'un ensemble d'unités de compilation, fournissant ou utilisant des fonctionnalités aux autres unités. D'une façon générale, toute unité de compilation doit citer au début les autres unités qu'elle utilise au moyen d'une clause **with**.

Les instructions de base sont inspirées de Pascal, tout en tenant compte des imperfections reconnues de celui-ci: toutes les instructions se terminent par un point-virgule; les instructions structurées se terminent par un mot clé, éliminant ainsi le besoin de blocs **begin**..**end**.

```
if condition then
   instructions
elsif condition then
   instructions
else
   instructions
end if;
```

```
case expression is
   when choix {|choix} =>
   instructions
   when choix {|choix} =>
         instructions
   when others =>
         instructions
end case;
```

```
[Etiquette_de_boucle:]
   [while condition]
   [for constante in [reverse] intervalle]
   loop
      instructions
   end loop;
```

Figure 1: Instructions de base

Les procédures et fonctions (que l'on appelle collectivement des sous-programmes) se présentent de façon similaire à Pascal: une en-tête, suivie de déclarations, puis d'instructions.

Procédure
```
procedure Identificateur [(paramètres)] is
   Déclarations
begin
   Instructions
exception
   Traite-exceptions
end Identificateur;
```

Fonction
```
function Identificateur[(paramètres)]return Identificateur_de_type is
   Déclarations
begin
   Instructions
exception
   Traite-exceptions
end Identificateur;
```

Paramètres

Identificateur : [**in** | **out** | **in out**] *Identificateur_de type*

Figure 2: Sous-programmes

Le mode de passage des paramètres fait référence à l'usage qui en est fait, et non à la méthode de passage: les paramètres peuvent ainsi être déclarés **in** (lecture seulement), **out** (écriture seulement), ou **in out** (lecture et écriture). Ceci remplace avantageusement la notion de "passage par valeur" ou de "passage par variable" utilisée en Pascal. Bien sûr, tous les sous-programmes sont récursifs (et réentrants, puisque Ada permet le parallélisme).

I.4 Le modèle du typage fort

Comme en Pascal, toutes les variables doivent être déclarées et munies d'un type. Le typage est extrèmement strict, (beaucoup plus qu'en Pascal - ne parlons pas de C ou de C++), et ceci constitue un des atouts majeurs du langage.

Un type définit un ensemble de *valeurs* muni d'un ensemble d'*opérations* portant sur ces valeurs. En fait, un type Ada représente des entités de plus haut niveau que les types d'autres langages: il représente en effet une entité *du domaine de problème*, et non une entité *machine*. Le programmeur exprime les exigences de son problème; si par exemple il doit représenter des temps avec une précision absolue de 1 ms et des longueurs avec une précision relative de 10^{-5}, il déclarera:

```
type TEMPS    is delta 0.001  range 0.0 .. 86400.0*366;
type LONGUEUR is digits 5     range 1.0E-16..1.0E25;
```

C'est le compilateur qui choisira, parmi les types machine disponibles, celui satisfaisant au moins les exigences ainsi exprimées. Le langage offre une vaste palette de types numériques: types entiers signés ou non (les types non signés sont munis d'une arithmétique modulaire), types flottants, fixes et décimaux. Par exemple, la première déclaration ci-dessus correspond à une déclaration d'un nombre *point fixe*, qui approxime les nombres réels avec une erreur *absolue* constante, alors que la seconde correspond à une déclaration de nombre *point flottant* qui approxime les nombres réels avec une erreur *relative* constante. Ces derniers correspondent aux nombres "réels" des autres langages. Noter la possibilité de limiter l'intervalle de valeurs en plus du type et de l'étendue de la précision[5].

[5] 10^{-16} m. représente le diamètre du quark, 10^{25} m. est le diamètre de l'univers (cf. [Mor82]).

Puisqu'il s'agit d'entités de nature différente, deux variables de types différents sont absolument incompatibles entre elles (on ne peut additionner une longueur et un temps!), *même lorsqu'il s'agit de types numériques*. Ceci s'applique bien entendu également aux entiers. Etant donné:

```
type Age    is range  0..120;  -- Soyons optimistes
type Etage  is range -3..10;
```

on peut écrire:

```
declare
   A: Age;
   F: Etage;
begin
   A := 5;
   F := 5;
   ...
```

mais l'affectation suivant est interdite par le compilateur:

```
A := F;     -- ERREUR: incompatibilité de types
```

Autrement dit, en Ada, on ne peut pas "mélanger des choux et des carottes". Ada est le seul langage à offrir cette propriété, qui paraît tout à fait naturelle aux débutants en informatique... et pose parfois des problèmes aux programmeurs expérimentés.

Ada fait une distinction très nette entre la notion de *type* et la notion de *sous-type*. Un sous-type résulte de l'application d'une *contrainte* sur un type. Une contrainte peut être par exemple un intervalle numérique, ou les valeurs des bornes d'un type tableau. L'affectation, le passage de paramètre sont autorisés si les *types* correspondent; une exception se produira si les *sous-types* diffèrent.

Une contrainte de sous-type peut toujours être dynamique: ceci permet notamment de choisir les tailles des tableaux lors de l'exécution, sans pour autant recourir à l'utilisation de types "pointeurs" (qui existent par ailleurs dans le langage, où on les appelle des types *accès*).

Ce mécanisme permet de définir des sous-programmes travaillant sur des tableaux de bornes quelconques. Des *attributs* permettent de connaître les valeurs effectives. Un exemple de sous-programme effectuant le produit de deux matrices est donné ci-dessous:

```
type MATRICE is array (POSITIVE range <>, POSITIVE range <>) of FLOAT;
MATRICE_ERROR : exception;

function "*"(X,Y : MATRICE) return MATRICE is
   Produit: MATRICE(X'RANGE(1), Y'RANGE(2));
begin
   if X'FIRST(2) /= Y'FIRST(1) or
      X'LAST (2) /= Y'LAST (1)
   then
      raise MATRICE_ERROR;
   end if;

   for I in X'RANGE(1) loop
      for j in Y'RANGE(2) loop
         Produit(I,J) := 0.0;
         for K in X'RANGE(2) loop
            Produit(I,J):= Produit(I,J) + X(I,K)*Y(K,J);
         end loop;
      end loop;
   end loop;

   return Produit;
end "*";
```

Ce sous-programme permet de faire le produit de deux objets de type MATRICE dès lors que les bornes de la deuxième dimension de la première matrice correspondent aux bornes de la première dimension de la deuxième matrice. Remarquer la possibilité de définir des opérateurs arithmétiques entre

types quelconques (y compris tableaux), ainsi que de définir des fonctions renvoyant des types structurés. Noter également que grâce aux attributs, le compilateur est capable de déterminer qu'aucun débordement d'indice n'est possible. L'optimiseur supprimera donc tous les contrôles inutiles, et le gain de sécurité apporté par Ada ne se traduira par aucune perte de performance.

Il est enfin possible de définir un nouveau type à partir d'un type existant, dont il héritera des propriétés (domaine de définition et opérations). Un tel type est appelé type *dérivé*. Bien sûr, il s'agit d'un type différent, et donc *incompatible* avec le type d'origine.

```
   type Mètres is digits 5 range 0.0 .. 1.0E15;
   type Yards is new Mètres;
   M : Mètres;
   Y : Yards;
begin
   Y := Y + M;     -- Interdit! On ne peut additionner
                   -- des Yards et des Mètres.
```

Ce mécanisme a été enrichi en Ada 95 pour fournir le mécanisme de base de la Programmation Orientée Objet (cf. ci-dessous).

I.5 Organisation de projet et bibliothèques hiérarchiques

Une notion centrale en Ada est celle de paquetage (**package**). Le paquetage permet de regrouper dans une seule entité des types de données, des objets (variables ou constantes), et des sous-programmes (procédures et fonctions) manipulant ces objets. Le paquetage comporte une partie *visible* comportant les informations utilisables à l'extérieur du paquetage, et une partie *cachée* qui regroupe les détails d'implémentation auxquels les utilisateurs du paquetage n'ont pas accès.

Les paquetages peuvent être compilés séparément, et seule leur spécification est nécessaire pour permettre l'utilisation du paquetage. On réalise ainsi une séparation complète entre les spécifications d'une unité fonctionnelle et son implémentation. De plus, les règles très rigoureuses de recompilation garantissent que, d'une part, il n'est pas possible d'utiliser une information cachée d'un paquetage, et d'autre part qu'en cas de modification de la partie "spécification", la recompilation des unités qui utilisaient cette spécification est obligatoire: le langage garantit donc la cohérence globale des différentes unités constituant un programme.

L'exemple ci-dessous montre la spécification d'un paquetage de gestion de nombres complexes.

```
package Nombres_complexes is
   type COMPLEX is private;
   I : constant COMPLEX;

   function "+"(X, Y: COMPLEX) return COMPLEX;
   function "-"(X, Y: COMPLEX) return COMPLEX;
   function "*"(X, Y: COMPLEX) return COMPLEX;
   function "/"(X, Y: COMPLEX) return COMPLEX;

   function CMPLX(X, Y : FLOAT) return COMPLEX;
   function POLAR(X, Y : FLOAT) return COMPLEX;

private
   type COMPLEX is
      record
         REEL : FLOAT;
         IMAG : FLOAT;
      end record;
   I : constant COMPLEX := (0.0, 1.0);
end Nombres_complexes;
```

On notera dans cet exemple que l'utilisateur n'a pas accès aux propriétés déclarées dans la partie **private**. Si, pour des raisons internes au projet, on décide de représenter les nombres complexes sous forme polaire au lieu de cartésienne, les règles du langage garantissent qu'aucune application utilisant ce paquetage n'aura à être modifiée. Un type déclaré privé (**private**) n'est manipulable que par les

fonctionnalités déclarées dans la spécification du paquetage, sauf qu'il dispose tout de même de l'affectation et de la comparaison d'égalité. Si l'on souhaite également interdire ces opérations, on peut le déclarer limité (**limited private**).

Ada 95 a enrichi les fonctionnalités des paquetages avec la notion de paquetages hiérarchiques. Il est possible de créer des paquetages "enfant" qui rajoutent des fonctionnalités à des paquetages existant sans avoir à toucher à leurs "parents", donc sans recompilation des utilisateurs de ces parents. Ceci permet une meilleure séparation en fonctionnalités "principales" et fonctionnalités "annexes", facilitant l'utilisation aussi bien que la maintenance. Par exemple, lorsque l'on fait un type de donnée abstrait, on ne souhaite généralement pas mélanger les propriétés de base avec les entrées-sorties par exemple. Ceci peut s'obtenir de la façon suivante:

```
package Nombres_Complexes is -- encore lui!
   type Complex is private;

   -- Opérations sur les nombres complexes

private
   type Complex is ...
end Nombres_Complexes;

package Nombres_Complexes.IO is -- Un enfant

   -- Entrées-sorties sur les complexes

end Nombres_Complexes.IO;
```

Le corps du paquetage enfant a accès à la partie privée du parent, autorisant donc des implémentations efficaces fondées directement sur la représentation interne du type abstrait.

De plus, il est possible de définir des enfants "privés", non accessibles depuis les unités extérieures à la famille. Ceci permet de rajouter des contrôles supplémentaires, puisque seul le parent est visible de l'extérieur et constitue le point d'entrée d'un sous-système dont les autres éléments sont cachés.

I.6 Génériques et réutilisation

Une autre propriété très importante du langage Ada est la notion d'unités génériques. Il s'agit d'unités paramétrisables, qui permettent de définir un algorithme indépendamment des types d'objet manipulés. Par exemple, une procédure qui intervertirait ses deux arguments est donnée ci-dessous. La déclaration du type T comme **private** signifie que n'importe quel type pour lequel l'affectation (et la comparaison d'égalité) sont définis peut faire l'affaire. L'unité générique n'est pas utilisable par elle-même: on doit en faire une *instanciation* qui précise les valeurs des paramètres génériques. Ainsi, la déclaration de générique de l'exemple suivant est suivie d'une instanciation qui produit une procédure permettant d'échanger deux variables de type COULEUR:

```
generic
   type ELEM is private;
procedure SWAP(X,Y : in out ELEM);

procedure SWAP(X,Y : in out ELEM) is
   TEMP : ELEM;
begin
   TEMP  := X;
   X     := Y;
   Y     := TEMP;
end SWAP;

procedure SWAP_COULEUR is new SWAP(COULEUR);

      ...
   SWAP_COULEUR(C1,C2);
      ...
```

L'exemple suivant donne la spécification d'une unité générique de tri de tableau. Les paramètres génériques sont le type d'indice du tableau, le type de composant, le type du tableau lui-même, et une fonction de comparaison. Cette fonction de comparaison est munie d'une valeur par défaut (clause **is <>**), signifiant que si l'utilisateur ne fournit pas de valeur, la comparaison prédéfinie (si elle existe) doit être utilisée. Les avantages de cette démarche sont évidents: à partir d'un algorithme écrit une fois pour toute, et que l'on pourra se donner le mal d'optimiser soigneusement, il est possible d'obtenir instantanément les fonctions de tri sur n'importe quel type de données, avec n'importe quel critère de comparaison.

```
generic
   type INDEX     is (<>);
   type COMPOSANT is private;
   type TABLEAU   is array(INDEX) of COMPOSANT;
   with function "<"(X,Y : COMPOSANT) return BOOLEAN is <>;
procedure TRI(TAB : in out TABLEAU);

type  INT is range 1..10;
type  ARR is array(INT) of FLOAT;
procedure TRI_ASCENDANT  is new TRI (INT, FLOAT, ARR);
procedure TRI_DESCENDANT is new TRI (INDEX     => INF,
                                     COMPOSANT => FLOAT,
                                     TABLEAU   => ARR,
                                     "<"       => ">");
```

Remarquer au passage que lors de la deuxième instanciation, nous avons utilisé une association *nommée* de paramètres: la correspondance entre paramètres formels et réels se fait par correspondance de noms, et non par la position. Ceci améliore considérablement la lisibilité et évite bien des erreurs. L'association nommée est également possible (et même recommandée) pour les appels de sous-programmes.

Les génériques constituent un outil très puissant pour la réutiilisation; comme dans l'exemple ci-dessus, ils permettent de définir des algorithmes généraux applicables à *tout* type vérifiant *au moins* certaines propriétés de base. Remarquons que la faculté de paramétrer les génériques Ada non seulement par un type de données, mais aussi au moyen des opérations que ce type doit posséder, les rend beaucoup plus puissants que les notions équivalentes d'autres langages (génériques d'Eiffel ou templates de C++).

I.7 Exceptions

La notion d'exception fournit un moyen commode de traiter tout ce qui peut être considéré comme "anormal" ou "exceptionnel" dans le déroulement d'un programme.

Une exception se comporte comme une sorte de déroutement déclenché par programme depuis une séquence d'exécution "normale" vers une séquence chargée de traiter les cas exceptionnels. Une exception peut être déclarée par l'utilisateur; certaines exceptions sont prédéfinies, comme STORAGE_ERROR en cas de mémoire insuffisante, ou CONSTRAINT_ERROR en cas de valeur incorrecte affectée à une variable.

Une exception est déclenchée soit implicitement (comme dans les cas précédants), soit explicitement au moyen de l'instruction **raise**. Un bloc de programme peut déclarer un traite-exception (*handler*) auquel le contrôle sera donné si l'exception citée se produit dans le bloc considéré. Voici un bloc dans lequel un traite-exception permet de renvoyer une valeur par défaut dans le cas où il se produit une division par 0:

```
   ...
begin
   Resultat := A/B;
exception
   when CONSTRAINT_ERROR =>
      Resultat := FLOAT'LARGE;
end;
   ...
```

Si une exception n'est pas traitée localement, elle est *propagée* aux unités appelantes, jusqu'à ce que l'on trouve un sous-traitant adapté ou que l'exception sorte du programme principal, ce qui arrête l'exécution du programme. Une clause spéciale, **when others,** permet de traiter toutes les exceptions. Voici un exemple de ce que pourrait être un programme principal qui garantirait un arrêt propre, quoi qu'il arrive dans le programme, sans connaissance à priori de la fonctionnalité des éléments appelés:

```
procedure MAIN is
begin
   FAIRE_LE_TRAVAIL;
exception
   when others =>
      NETTOYAGE;
end MAIN;
```

Il est important de noter que des événements "catastrophiques", comme la saturation de l'espace mémoire, provoquent simplement en Ada la levée d'exceptions prédéfinies, et sont donc traitables par le programme utilisateur.

Le mécanisme des exceptions permet également de simplifier l'écriture des programmes et d'améliorer la lisibilité, car il n'est plus nécessaire de prévoir dans les appels de procédure de variable en sortie pour spécifier si un appel s'est passé correctement ou non. En cas de problème, une exception est levée que l'utilisateur est libre de traiter ou de laisser propager s'il ne s'estime pas compétent pour la traiter. Remarquer que l'oubli par un programmeur du traitement d'une situation exceptionnelle provoquera la propagation de l'exception; l'erreur deviendra alors apparente, alors que l'oubli du test d'une condition de sortie entraînera la continuation du programme comme si les résultats étaient valables, provoquant donc des résultats erronnés non visibles. La solution du traitement par exception est donc intrinsèquement plus fiable que la technique du code de retour.

I.8 Le concept d'annexes

Ada est un langage à spectre très large; ses plus beaux succès se trouvent aussi bien dans le contrôle de simulateurs ou l'aéronautique qu'en gestion financière ou en CAO. C'est la raison pour laquelle le langage laisse certains points comme "dépendant de l'implémentation", afin de ne pas privilégier un type d'application aux dépens d'un autre. D'autre part, Ada est un langage extensible: son mécanisme de paquetages permet de rajouter toutes les fonctionnalités voulues sans avoir à toucher au langage lui-même. Or il est évident qu'un utilisateur "temps réel" n'aura pas besoin des mêmes fonctionnalités qu'un utilisateur "gestion".

Ada 95 est muni d'un certain nombre d'annexes correspondant à divers domaines d'utilisation. Elles précisent des éléments du langage autrement laissés libre, et définissent des paquetages spécifiques aux différents domaines. Les annexes ne contiennent ni nouvelle syntaxe, ni modification des règles du langage telles que définies dans la norme principale. Les fournisseurs de compilateurs seront libres de passer une, plusieurs, ou même aucune des suites de tests correspondant aux différentes annexes. Bien entendu, le certificat de validation mentionnera quelles annexes ont été passées.

Si la validation n'est accordée pour une annexe que si elle entièrement implémentée, il n'est pas interdit de n'en fournir qu'une partie des spécificités, *à condition que les fonctionnalités offertes soient conformes à la définition de la norme*. Il sera donc possible (et nous attendons des fournisseurs qu'ils le fassent) d'offrir, par exemple, le paquetage de gestion des nombres complexes même si des contraintes matérielles empêchent de valider totalement l'annexe (précision de la machine insuffisante par exemple).

Il est important de comprendre que ce mécanisme n'est pas, comme on pourrait le croire à première vue, une rupture avec le dogme d'unicité du langage; au contraire, il augmente la portabilité des applications d'un certain domaine en poussant la standardisation spécifique au delà de ce qu'il était possible de faire dans le cadre d'un langage général.

On trouve ainsi:

■ L'annexe sur la *programmation système* qui décrit des paquetages et des pragmas permettant une gestion plus directe des interruptions, la pré-élaboration de paquetages (mécanisme facilitant la mise en mémoire morte de parties de systèmes), et des manipulations supplémentaires de bas niveau sur les tâches. La documentation devra comporter obligatoirement des précisions sur l'efficacité en temps d'exécution de certains mécanismes.
■ L'annexe sur les *systèmes temps réel* qui rajoute des fonctionnalités de gestion des priorités (priorités dynamiques notamment), qui permet de choisir entre plusieurs politiques d'ordonnancement des tâches et de gestion des files d'attente (files prioritaires au lieu du modèle premier arrivé-premier servi par défaut), qui comporte des exigences sur les délais avant avortement et la précision de l'instruction **delay**, qui fournit un modèle de tâches simplifié permettant une implémentation extrêmement efficace, et de nouvelles fonctionnalités de gestion du temps. Noter qu'un compilateur enregistré comme supportant cette annexe devra obligatoirement supporter également l'annexe sur la programmation système.
■ L'annexe sur les *systèmes distribués* qui permet de définir un programme comme un ensemble de partitions s'exécutant sur des processeurs différents. Le programmeur peut contrôler le format des messages échangés entre les différentes partitions, et il est possible de remplacer dynamiquement une partition sans relancer tout le système. Cette annexe définit également un modèle d'appel de sous-programme distant (*remote procedure call*).
■ L'annexe sur les *systèmes d'information* qui fournit de nouvelles fonctionnalités de présentation des données (comme les clauses *picture* de COBOL) et des paquetages d'arithmétique complémentaire sur les nombres décimaux. Cette annexe s'est réduite au fil du temps, car une grande partie de ce qu'elle comprenait a été jugé tellement important qu'elle a été remise dans le corps du langage!
■ L'annexe *sûreté et sécurité* qui impose des contraintes supplémentaires au compilateur sur le traitement des cas définis comme erronés dans le langage, sur un certain nombre de points laissés à la discrétion de l'implémentation, et sur la certification du compilateur aussi bien que du code produit (possibilité de contrôler manuellement le code généré notamment).
■ L'annexe sur les *numériques* qui fournit un modèle encore plus précis des calculs réels, notamment pour les machines utilisant le modèle IEEE des nombres flottants, et inclue la définition des paquetages de nombres complexes, ainsi que les fonctions élémentaires et les entrées-sorties sur ces nombres complexes.

II Le modèle de programmation orientée objet.

De nombreuses demandes d'utilisateurs critiquaient le manque de support d'Ada 83 pour le paradigme d'héritage. Ce point était cependant délicat, car si l'héritage facilite le développement rapide d'applications, son efficacité pour améliorer la maintenance à long terme reste à démontrer; de plus, il peut conduire à un affaiblissement du typage, puisque le polymorphisme implique la vérification de certaines cohérences de type à l'exécution, alors qu'Ada assure traditionnellement tous ces contrôles lors de la compilation. Enfin, la programmation orientée objet conduit souvent à une multiplication de pointeurs et d'allocations dynamiques cachées. Or le monde du temps réel, particulièrement pour les applications devant fonctionner 24 heures sur 24, est très hostile à l'allocation dynamique à cause des risques de fragmentation mémoire et de perte d'espace qu'elle provoque. Il fallait donc introduire un mécanisme permettant d'ouvrir Ada à de nouvelles applications, sans lui faire perdre ses principes fondamentaux: primauté de la facilité de maintenance sur la facilité de conception, contrôle précis de l'allocation et vérifications strictes lors de la compilation.

La solution retenue est fondée sur une extension naturelle du mécanisme des types dérivés: les types étiquetés (*tagged*). Il s'agit de types munis d'un attribut supplémentaire caché permettant d'identifier le type d'un objet à l'intérieur de sa *classe* (ensemble des types dérivés directement ou indirectement d'un même ancêtre).

Ces types offrent de nouvelles possibilités:
■ l'extension de types. Il est possible, lorsque le type parent d'un type dérivé est un type étiqueté, d'étendre le type de donnée en rajoutant des champs supplémentaires lors de la dérivation. Les opérations

du parent restent disponibles, mais n'opèrent bien sûr que sur la partie commune. Il s'agit donc bien, cette fois-ci, d'un véritable mécanisme d'héritage.

- Le polymorphisme. Des sous-programmes peuvent posséder des paramètres se référant à la *classe* d'un type étiquetté: ils sont applicables alors non seulement au type d'origine, mais également à tous les types qui en sont dérivés. Il est également possible de définir des pointeurs sur classe, pouvant désigner n'importe quel objet appartenant à la classe. Cette dernière forme de polymorphisme permet l'implémentation des mécanismes habituels de la Programmation Orientée Objet, tout en conservant le contrôle explicite des pointeurs.
- Les liaisons dynamiques. Si on appelle un sous-programme en lui fournissant un paramètre réel de type *classe,* le sous-programme effectivement appelé n'est pas connu à la compilation. C'est l'étiquette *(tag)* de l'objet réel qui détermine à l'exécution le sous-programme effectivement appelé. Remarquons que ce mécanisme n'exige *pas* la présence de pointeurs: contrairement à nombre de langages orientés objets[6], il est donc possible d'effectuer des liaisons dynamiques sans imposer de pointeurs, cachés ou non.
- Les types abstraits. Il est possible de définir des types munis de l'attribut **abstract**, qui ne peuvent servir à définir des objets, mais seulement de racine d'arbres de dérivation pour des types concrets définis plus tard.

Généralement, les langages orientés objet fournissent toutes ces possibilités "en bloc". En Ada 95, l'utilisateur ne demandera que les fonctionnalités dont il a besoin, et ne paiera le prix que de ce qu'il utilise. Là où l'extensibilité et la dynamicité sont plus importants que la sécurité, Ada offrira toutes les fonctionnalités nécessaires. En revanche, un concepteur qui ne souhaiterait *pas* utiliser ces nouvelles fonctionnalités conservera le même niveau de contrôle de type et la même efficacité du code généré qu'avec Ada 83.

On notera qu'en Ada 95 l'héritage *multiple* sera obtenu au moyen des autres blocs de base du langage plutôt que par une construction syntaxique spéciale. Ceci n'est pas une omission, mais un choix délibéré: le risque de complexité lié à ce mécanisme aurait été trop important, et les autres perfectionnements, notamment concernant les génériques, ont permis de construire tous les cas d'héritage multiple que l'on peut rencontrer dans la littérature.

Pour plus de détails, on trouvera dans [Ros94]ou [Bar94] une discussion complète des mécanismes de la POO en Ada 95.

III Le Parallélisme

III.1 Tâches

Le langage Ada permet de définir des tâches, qui sont des unités de programme s'exécutant en parallèle. Les tâches sont des objets appartenant à des *types tâche*, et peuvent donc être membres de structures de données: on peut ainsi définir des tableaux de tâches, des pointeurs sur des tâches, etc.

Chaque tâche peut être munie d'une *priorité d'exécution* (le support des priorités est obligatoire pour les implémentations fournissant l'annexe "temps réel"). La priorité initiale est définie par une déclaration de la tâche, mais un paquetage prédéfini fournit les services permettant de changer dynamiquement cette priorité.

Les types tâches peuvent être munis de discriminants, qui servent à paramétriser leur comportement. On peut transmettre ainsi à la täche des valeurs simples (et notamment l'espace mémoire requis ou la priorité d'exécution), mais également des pointeurs sur des blocs de paramètres, sur d'autres tâches... En particulier, ceci permet de construire une tâche qui fait partie d'une structure de donnée, et comportant un pointeur sur la structure englobante. Il est ainsi possible de réaliser de véritables *objets actifs*.

[6] dont Eiffel et C++

Deux moyens permettent aux tâches de se synchroniser: l'un est orienté traitement, c'est le *rendez-vous*; l'autre est orienté données, ce sont les *objets protégés*.

III.2 Synchronisation par rendez-vous

Dans le premier cas, une tâche serveuse déclare des points d'entrée, dont la spécification est identique à une spécification de procédure. Une instruction spéciale, **accept**, lui permet d'accepter un appel du point d'entrée. Une autre tâche peut appeler le point d'entrée à tout moment. Une tâche acceptant une entrée, ou une tâche appelant un point d'entrée est suspendue jusqu'à ce que son partenaire ait effectué l'instruction complémentaire. Le rendez-vous a alors lieu, la communication s'établissant par échange de paramètres comme pour un appel de procédure. Une fois le rendez-vous terminé, les deux tâches repartent en parallèle. D'autres formes syntaxiques permettent de raffiner ce mécanisme, en permettant l'attente multiple d'un serveur sur plusieurs points d'entrée à la fois, l'attente avec temporisation (*time out*), l'acceptation conditionnelle, ou la possibilité de terminer automatiquement la tâche si le serveur n'est plus nécessaire.

Si la tâche qui exécute le rendez-vous est de priorité inférieure à son client, elle hérite de la priorité de ce client pour la durée du rendez-vous.

Voici un exemple de tâche simple permettant de tamponner la transmission d'un caractère entre une tâche productrice et une tâche consommatrice:

```
task TAMPON is
   entry LIRE  (C : out CHARACTER);
   entry ECRIRE(C : in CHARACTER);
end TAMPON;

task body TAMPON is
   Occupe : BOOLEAN := False;
   Valeur : CHARACTER;
begin
   loop
      select
      when not Occupe =>
         accept ECRIRE(C : in CHARACTER) do
            Valeur := C;
         end ECRIRE;
         Occupe := True;
      or when Occupe =>
         accept LIRE(C: out CHARACTER) do
            C := Valeur;
         end LIRE;
         Occupe := False;
      or terminate;
      end select;
   end loop;
end TAMPON;
```

III.3 Synchronisation par objets protégés

Dans le second cas, on définit des objets (ou de types d'objets) auxquels sont associés des sous-programmes spéciaux, dont on garantit l'exclusivité d'accès. Ils s'apparentent donc aux moniteurs de Hoare, mais en plus perfectionné (possibilité de mettre des barrières notamment).

Voici un tel objet protégé:

```
protected Barrière is --
   entry Passer;
   procedure Ouvrir;
   function En_Attente return Natural;
private
   Ouverte : Boolean := False;
end Barrière;
```

```
protected body Barrière is
   entry Passer is
   begin
      if Passer'Count = 0 then   -- Le dernier referme
         Ouverte := False;       -- la barrière.
      end if;
   end Passer;

   procedure Ouvrir is
   begin
      if En_attente > 0 then
         Ouverte := True;
      end if;
   end Ouvrir;

   function En_Attente return Natural is
   begin
      return Passer'Count;
   end En_Attente;
end Barrière;
```

Une tâche appelant l'entrée Passer est bloquée, jusqu'à ce qu'une autre tâche appelle la procédure Ouvrir; toutes les tâches en attente sont alors libérées. La fonction En_Attente permet de connaître le nombre de tâches en attente.

Pour bien comprendre cet exemple, il faut savoir que:

- Plusieurs tâches peuvent appeler simultanément une fonction de l'objet protégé, ou (exclusivement) une seule tâche peut exécuter une procédure ou une entrée (algorithme des lecteurs/écrivains).
- Des tâches susceptibles de s'exécuter parce que la condition associée à une entrée est devenue vraie s'exécutent toujours *avant* toute autre tâche appelant une opération de l'objet protégé.

Par conséquent, les tâches libérées lorsqu'un appel à Ouvir a remis la variable Ouverte à True reprendront effectivement la main avant toute tâche non bloquée sur l'entrée. Une tâche qui appellerait l'entrée Passer juste après qu'une autre ait appelé Ouvrir se bloquera donc bien (après que toutes les tâches en attente aient été libérée); aucune condition de course n'est possible.

Il est possible (avec les implémentations supportant l'annexe "temps réel") d'associer une priorité à un objet protégé, qui est une priorité "plafond": toute opération protégée s'effectuera à ce niveau de priorité, et il est interdit à une tâche de niveau de priorité supérieure d'appeler ces opérations protégées. Ce mécanisme permet de garantir l'absence de phénomènes d'inversion de priorité; de plus, sur un système mono-processeur, il est suffisant pour garantir l'exclusivité d'accès aux objets protégés, sans nécessiter de sémaphore supplémentaire, ce qui rend les objets protégés particulièrement efficaces.

III.4 Autres fonctionnalités

Aux mécanismes de rendez-vous que nous avons cités et qui existaient déjà en Ada 83, Ada 95 a rajouté le *select asynchrone*, qui permet à un client de continuer à travailler pendant qu'une demande de rendez-vous est active, ou après avoir armé un délai. Si la demande est servie, ou si le délai expire, avant la fin du traitement associé, celui-ci est avorté; sinon, c'est la demande qui est annulée. Les deux formes de select asynchrone sont montrées ci-dessous:

```
select                          select
   appel d'entrée                  delay [until] ...;
then abort                      then abort
   instructions                    instructions
end select;                     end select;
```

Une autre fonctionnalité importante est l'instruction **requeue** qui permet pendant le traitement d'un **accept** de tâche ou d'un appel d'une entrée d'objet protégé de renvoyer le client en attente sur une autre queue. Par exemple:

```
type Urgence is (Lettre, Télégramme);
   ...
accept Envoyer (Quoi : Urgence; Message : String) do
   if Quoi = Télégramme then
      -- On s'en occupe tout de suite
   else
      requeue Voir_Plus_Tard(Quoi, Message);
   end if;
end Envoyer;
```

Ici, une tâche serveuse de messages dispose d'un point d'entrée unique pour des messages urgents (les télégrammes) ou moins urgents (les lettres). Elle traîte les télégrammes immédiatement, mais renvoie les lettres sur son entrée Voir_Plus_Tard, qu'elle ira traiter lorsqu'il n'y aura plus de messages en atttente sur Envoyer.

Enfin, l'annexe "programmation système" définit les pragmas ATOMIC, garantissant l'indivisibilité physique des accès mémoire aux objets concernés, et VOLATILE qui informe le compilateur que la valeur d'une variable est susceptible de changer par des moyens extérieurs au langage (comme dans le cas d'un périphérique en mapping d'adresse), et que l'optimiseur doit prendre des précautions en conséquence.

IV La gestion du bas niveau

Ada étant un langage également destiné à la programmation des systèmes, il était nécessaire de fournir des interfaces de bas niveau. Il est ainsi possible de forcer la représentation physique des structures abstraites, de spécifier des traitements d'interruptions physiques, d'inhiber des vérifications de type, et même d'inclure du code machine. Toutefois, des précautions ont été prises pour que même ce genre de manipulations ne puissent se faire sans un minimum de précaution.

IV.1 Clauses de représentation

Une possibilité très intéressante est la possibilité de spécifier au bit près la représentation des types de données. Normalement, la représentation interne est choisie par le compilateur, et le langage garantit la totale indépendance de l'utilisation des types vis à vis de leur représentation. Cependant, il peut être nécessaire d'imposer une représentation, notamment lors d'interfaçages avec le matériel ou avec des sous-programmes écrits dans d'autres langages. On peut alors donner au programmeur une vision de haut niveau, tout en imposant la représentation de bas niveau. On peut ainsi décrire une cellule de mémoire-écran d'IBM-PC:

```
type T_COULEUR is
 (Noir,         Bleu,          Vert,       Cyan,
  Rouge,        Magenta,       Marron,     Gris,
  Gris_sombre,  Bleu_clair,    Vert_clair, Cyan_clair,
  Rouge_clair,  Magenta_clair, Jaune,      Blanc);
subtype T_COULEUR_FOND is T_COULEUR range Noir..Gris;

-- Représentation des énumératifs:
for T_COULEUR use (0, 1,  2,  3,  4,  5,  6,  7,
                   8, 9, 10, 11, 12, 13, 14, 15);

type T_CLIGNOTEMENT is (Fixe, Clignotant);
for T_CLIGNOTEMENT use (0, 1)

type T_ATTRIBUT is
record
   Clignotement : T_CLIGNOTEMENT;
   Fond         : T_COULEUR_FOND;
   Devant       : T_COULEUR;
end record
```

```
-- Représentation du type article: at <mot>
--                                 range <bits>
for T_ATTRIBUT use
   record
      Clignotement   at 0 range 0..0;
      Fond           at 0 range 1..3;
      Devant         at 0 range 4..7;
   end record;
```

Si l'on veut changer la valeur du bit "Clignotement" d'un attribut, on écrira simplement:

```
A.Clignotement := Fixe;
```

ce qui aura effectivement pour effet de changer la valeur du bit considéré. On voit qu'ainsi, il n'est plus jamais besoin de calculer des masques, décalages, etc. pour intervenir à bas niveau. En cas de modification de la représentation interne des données, aucune modification du code n'est nécessaire.

On voit que les règles du langage ne sont nullement changées par la présence de clauses de représentation, ce qui constitue un outil très puissant. Supposons que nous ayons besoin, par exemple lors de l'écriture d'une interface, de deux représentations physiques différentes d'un même type de données. Il suffit de faire un type dérivé muni de clauses différentes de celles d'origine. Les types dérivés étant convertibles entre eux, le changement de représentation sera pris en charge par le compilateur, comme dans l'exemple suivant:

```
   type Format_1 is ...
   for  Format_1 use ...   -- Représentation de Format_1

   type Format_2 is new Format_1;
   for  Format_2 use... --  Représentation de Format_2

   V1 : Format_1;
   V2 : Format_2;
begin
   ...
   V1 := Format_1(V2);  -- Conversion prise en charge par le compilateur
   V2 := Format_2(V1);  -- Conversion prise en charge par le compilateur
```

Ces conversions sont des conversions de haut niveau, respectant la sémantique des types. Il est parfois nécessaire d'effectuer des conversions plus directes (*type cast*), notamment lorsqu'une même donnée doit être vue selon deux niveaux d'abstraction différents. Par exemple, des caractères sont des entités propres, qui ne sont pas considérées comme des valeurs numériques. On ne peut donc additionner des caractères, ce qui n'aurait aucun sens... sauf si l'on veut calculer un CRC. On peut alors forcer une vue d'un caractère sous forme numérique grâce à l'instantiation de la fonction UNCHECKED_CONVERSION:

```
   type Octet is mod 256;  -- type modulaire
   for  Octet'Size use 8;  -- représenté sur 8 bits

   function Char_to_Octet is new Unchecked_Conversion(Character, Octet);

   CRC : Octet;
   C   : Character;
begin
   ...
   CRC := CRC * 2 + Char__to_Octet(C);
```

La sécurité des types n'est pas mise en cause, car les règles du langage imposent dans ce cas de mettre en tête de module une clause "**with** UNCHECKED_CONVERSION;", dont la présence est tracée par le compilateur. Les zônes de code effectuant ce genre d'opérations, potentiellement dangereuses ou non portables, sont donc instantanément identifiables.

IV.2 Flots de données

Un flot de données est un type descendant du type (étiqueté) Root_Stream_Type défini dans le paquetage Ada.Streams. Conceptuellement, c'est une suite d'octets, muni de procédures Read et Write permettant de lire et d'écrire des tableaux d'octets dans le flot.

En plus de cette utilisation de premier niveau, le grand intérêt des flots de données vient du fait que le langage fournit automatiquement (sous forme d'attributs) des procédures permettant de lire et d'écrire tout type de donnée (non limité) depuis ou vers un flot de données. De plus, l'utilisateur peut redéfinir lui-même ces procédures s'il souhaite une représentation externe différente. Il peut aussi définir ces procédures pour des types limités.

Il est ainsi possible de passer de façon sûre et portable d'une vue abstraite d'un type de donnée vers un simple ensemble d'octets. C'est en particulier indispensable pour permettre de transmettre des types de haut niveau sur un réseau, ou si l'on veut écrire des méthodes d'accès pour des fichiers indexés ou autre.

Physiquement, un flot peut être une structure de donnée en mémoire, un fichier, une interface réseau... Le langage fournit une implémentation des flots de données dans un fichier binaire, ainsi que dans un fichier texte. Comme le contenu physique est un ensemble d'octets, mais que les fonctionnalités des flots permettent de les convertir depuis/vers n'importe quel type, cela permet de créer des fichiers de données hétérogènes sans perdre les avantages du typage fort.

IV.3 Gestion des interruptions

Ada 83 faisait gérer les interruptions par des tâches, qui associaient des entrées à des interruptions physiques. A l'usage, ce mécanisme s'est révélé peu conforme à l'attente des utilisateurs temps-réel. Il existe toujours en Ada 95 (ne serait-ce que par compatibilité ascendante), mais est classé "obsolète", ce qui signifie que l'on est censé lui préférer le nouveau mécanisme, l'attachement d'interruptions à des procédures protégées. Ce mécanisme fait partie de l'annexe "programmation système".

Il existe deux moyens d'attacher une procédure protégée à une interruption: statiquement, au moyen de pragmas, ou dynamiquement au moyen de sous-programmes fournis dans le paquetage Ada.Interrupts. Ce paquetage fournit également des fonctionnalités annexes permettant de savoir si une procédure est actuellement attachée à une interruption, d'attacher une interruption à une procédure protégée en récupérant un pointeur sur le traitement d'interruption précédemment attaché (pour le rétablir plus tard), etc.

Le choix de représenter les traitements d'interruption par des procédures protégées plutôt que par des sous-programmes normaux permet d'exprimer de façon commode la nature non ré-entrante des traitements d'interruptions. Il permet également d'interdire l'utilisation des instructions de synchronisation bloquantes, telles que **delay** ou appel d'entrée, depuis les gestionnaires d'interruption.

IV.4 Contrôle de l'ordonnancement et gestion des queues

Normalement, l'ordonnancement des tâches est géré par l'exécutif fourni avec le compilateur. Ada 95 autorise plusieurs politiques d'ordonnancement, sélectionnables au moyen d'un pragma. Une seule est standardisée: FIFO_Within_Priorities, qui exprime que les tâches sont ordonnancées dans l'ordre où elles deviennent candidates, les tâches moins prioritaires ne s'exécutant que si aucune tâche plus prioritaire n'est susceptible de prendre le contrôle d'un processeur. Si cette politique est choisie, alors il est obligatoire de spécifier également le plafond de priorité pour les types protégés. Une implémentation est autorisée à fournir d'autres politiques d'ordonnancement.

Cependant, de nombreuses applications temps réel souhaitent un contrôle plus précis de l'ordonnancement. Dans ce but, l'annexe "programmation système" fournit des attributs et un paquetage permettant de récupérer le "Task_ID" interne d'une tâche (en particulier, de l'appelant dans un rendez-vous) et d'effectuer certaines opérations dessus. Un paquetage permet de créer des attributs de tâches, sortes de variables associées à chaque tâche (concrètement, cela signifie que l'on rajoute des variables

utilisateur dans le bloc de contrôle de tâche). A ces fonctionnalités, l'annexe "temps réel" rajoute un paquetage de contrôle synchrone fournissant une sorte de sémaphore sur lequel les tâches peuvent venir se bloquer, et un paquetage de contrôle asynchrone permettant de suspendre et de relancer n'importe quelle tâche dont on connaît le "Task_ID". L'ensemble de ces fonctionnalités permet à l'utilisateur de contrôler entièrement l'ordonnancement des tâches; en particulier, les attributs permettent de stocker dans chaque tâche les données nécessaires à la détermination de la prochaine tâche à ordonnancer.

En ce qui concerne la gestion des queues, les requêtes de rendez-vous et les files d'attente des entrées protégées sont traitées par défaut dans l'ordre premier arrivé-premier servi. L'annexe "temps-réel" fournit un pragma demandant de traiter les requêtes par priorité (les tâches plus prioritaires "doublent" les tâches moins prioritaires dans la queue); toute implémentation est autorisée à fournir des pragmas définissant d'autres algorithmes de mise en queue.

IV.5 Contrôle de l'allocation mémoire

Le paquetage Storage_Elements fournit la notion de tableau d'octets, un type adresse sous forme de nombre entier, et des fonctionnalités de conversion (propres!) entre types accès et adresses. Il est donc possible d'accéder directement à la mémoire sans recourir à des utilisations douteuses (et souvent non portables) de UNCHECKED_CONVERSION.

Enfin le paquetage Storage_Pools fournit un type de données permettant à l'utilisateur de définir des "objets pools", et de les associer à des types accès. L'allocation et la désallocation dynamique pour ces objets (instruction **new**) doit obligatoirement utiliser ces objets. L'utilisateur contrôle donc entièrement les algorithmes d'allocation/désallocation de mémoire dynamique, et peut donc garantir l'absence de fragmentation, le temps maximum d'une allocation dynamique, etc. indépendemment de la bibliothèque d'exécution (*run-time*) du compilateur.

IV.6 Gestion du temps

Ada étant un langage "temps réel", il est bien entendu possible de suspendre une tâche pendant un certain intervalle de temps et d'accéder à l'heure absolue. Le paquetage CALENDAR fournit la définition d'un type "temps" (TIME), et d'une fonction renvoyant l'heure courante. Ce temps est lié à l'horloge "normale" du système. S'il existe plusieurs bases de temps, l'implémentation a le droit de fournir d'autres types "temps" (documentés comme tels) correspondant à ces différentes horloges. L'annexe "temps réel" spécifie une telle horloge (dans le paquetage Ada.Real_Time), obligatoire pour les implémentations supportant cette annexe, et qui fournit un temps "informatique", garanti monotone croîssant[7].

Il est possible de mettre une tâche en attente au moyen de l'instruction **delay**:

```
delay 3.0;        -- Attente relative, 3s.
delay until Time_Of((1995, 01, 11, 15*Heure+30*Minute)); -- Attente absolue
```

Le premier cas correspond à une attente relative, la durée étant exprimée en secondes. Le second correspond à une attente absolue jusqu'à une certaine heure, exprimée dans un quelconque "type temps". Si l'implémentation en fournit plusieurs, c'est le type de l'argument qui déterminera l'horloge utilisée.

Ces mêmes définitions de la sémantique liée au temps sont utilisées dans toutes les instructions utilisant une forme "delay", comme les appels d'entrée et acceptations temporisés (avec *time-out*) et les transferts de contrôle asynchrone.

Ce mécanisme permet donc de disposer d'instructions liées au temps portables, tout en bénéficiant de l'accès à des horloges spécifiques si la précision est plus importante que la portabilité.

[7] Le temps par défaut peut correspondre à l'heure légale, et donc subir des sauts vers l'avant ou vers l'arrière lors des passages heure d'été <=> heure d'hiver.

V Interfaçages

Un langage moderne se doit de permettre l'accès aux grandes bibliothèques disponibles sur le marché; or souvent ces bibliothèques ne sont fournies qu'avec des interfaces pour d'autres langages. Aussi un grand effort a-t-il été apporté pour permettre d'assurer l'interopérabilité d'Ada avec les autres langages.

Le pragma Import permet d'utiliser en Ada des éléments en provenance d'autres langages: sous-programmes, variables bien entendu, mais aussi exceptions et même classes. C'est ainsi que le compilateur GNAT (v. à la fin de ce papier) permet de réutiliser directement en Ada des classes définies en C++. Inversement, le pragma Export permet d'écrire des éléments en Ada et de les rendre utilisables par d'autres langages. La cohérence avec les types de données d'autres langages est apportée par le paquetage Interfaces qui regroupe des paquetages enfants pour les différents langages dont l'interfaçage est supporté. Les spécifications de Interfaces.C, Interfaces.FORTRAN et Interfaces.COBOL sont données dans la norme. On y trouve les déclarations Ada des types de ces langages, ainsi que des fonctionnalités nécessaires à une bonne liaison. C'est ainsi que, pour C, le paquetage enfant Interfaces.C.Strings gère des chaînes de caractères "à la C" (pointeur sur un flot d'octets terminé par un caractère NUL), et que Interfaces.C.Pointers gère une arithmétique sur pointeurs.

Nous ne pouvons qu'insister sur la facilité d'interfaçage apportée par ces paquetages; dans le cadre du projet GNAT, nous avons implémenté une interface complète avec le système d'exploitation OS/2 (en utilisant la bibliothèque C); grâce à ces paquetages, tout a fonctionné sans problème dès le premier essai !

VI Distribution

L'annexe "systèmes distribués" définit certains éléments supplémentaires permettant d'écrire des systèmes distribués en Ada. Ada 95 est le premier langage à inclure un tel modèle au niveau de la définition du langage, donc de façon portable et indépendante des systèmes d'exploitation ou des exécutifs sous-jacents.

VI.1 Partitions

Un programme Ada est constitué d'un ensemble de "partitions". Les partitions peuvent être actives ou passives; dans ce dernier cas, elles ne peuvent posséder de flôt de contrôle en propre. La façon de construire des partitions à partir du programme est laissée à la discrétion de l'implémentation, mais les contrôles à la compilation sont faits au niveau "programme": on garantit donc la cohérence de toutes les partitions figurant dans une même bibliothèque.

La définition de la notion de partition est volontairement lâche, afin de lui permettre de correspondre à différentes utilisations, comme résumées dans le tableau ci-dessous:

	Partition active	**Partition passive**
Cartes autonomes	Carte processeur	Carte mémoire
Réseau local	Ordinateur	-
Système d'exploitation	Processus	DLL

Figure 3: Utilisation des partitions

VI.2 Classification des paquetages

L'écriture d'un système distribué nécessite certaines précautions. Par exemple, si plusieurs partitions utilisent un paquetage "normal", celui-ci sera dupliqué dans chacune d'elles: s'il possède des états cachés, ils évolueront donc indépendemment dans chaque partition.

Lorsque ceci n'est pas le comportement souhaité, Ada permet de fournir des pragmas de catégorisation. Ceux-ci permettront de définir des sémantiques différentes en cas de distribution, et apporteront leur lot

de restrictions (vérifiées par le compilateur) nécessaires au bon fonctionnement du système. On trouve ainsi:

- Les paquetages "purs" (*pure*). Ils sont dupliqués dans chaque partition, mais les restrictions imposées garantissent que leur comportement est indépendant de leur duplication (pas d'état local).
- Les paquetages "partagés passifs" (*shared_passive*). Ils ne peuvent contenir aucune entité active (ni servir à accéder une entité active de façon indirecte), et sont présents dans une seule partition passive du système. Ils peuvent contenir des sous-programmes et des états rémanents, qui sont partagés par tous les utilisateurs du système distribué.
- Les paquetages "types distants" (*remote_types*). Ils servent à définir les types de données susceptibles d'être échangés entre partitions. Les restrictions associées garantissent que ces types ont les propriétés nécessaires pour permettre leur transmission sur un réseau (pas de pointeurs locaux par exemple, sauf si l'utilisateur a défini une procédure permettant la transmission).
- Les paquetages "appel distants" (*remote_call_interface*, en abrégé RCI). La spécification de ces paquetages est dupliquée dans chaque partition, mais le corps n'est physiquement présent que dans une seule; dans les autres, un corps "bouchon" route les appels vers la partition qui possède le corps vrai.

Lorsqu'un appel est transmis à travers le réseau, le compilateur doit utiliser les services d'un paquetage prédéfini (RPC), dont le corps peut être fourni par l'utilisateur, le fournisseur du réseau, etc. Le compilateur est ainsi indépendant de la couche "transport", et il est possible d'implémenter la distribution par-dessus n'importe quelle couche réseau sans avoir à toucher au compilateur.

VI.3 Distribution par RPC "classique"

Lorsqu'un sous-programme est défini dans un paquetage RCI, tous les appels sont naturellement routés vers la partition sur laquelle il se trouve. Noter que la propriété "appel à distance" est héritée par tous les sous-programmes dérivés, même si cette dérivation n'a *pas* lieu dans un paquetage déclaré RCI. On peut donc avoir des appels à distance vers des sous-programmes situés n'importe où, mais dont la définition initiale se trouvait dans un paquetage RCI, ce qui permet de garantir qu'ils vérifient les restrictions nécessaires à la transmission des arguments sur un réseau.

De même, un pointeur sur sous-programme défini dans un paquetage RCI ou "types distants" est un type accès à distance (*remote access type*). Ceci signifie que le pointeur contient l'information nécessaire pour localiser la partition où se trouve physiquement le sous-programme. Tout appel à travers un tel pointeur sera un appel distant, le sous-programme appelé pouvant être situé n'importe où. Les règles sur les pointeurs garantissent cependant que les paramètres de tels sous-programmes sont toujours transmissibles sur réseau.

Dans l'exemple ci-dessous, on définit des serveurs génériques. Chaque serveur traite les requêtes qui lui sont adressées, ou s'il est trop occupé transmet la requête au serveur suivant. La remontée des résultats a lieu en sens inverse, chaque serveur appelant une procédure fournie par pointeur lors de la demande de service afin de les traiter. Les serveurs ne se connaissent pas entre eux: au démarrage ils s'enregistrent auprès d'un gestionnaire centralisé qui leur fournit simplement un pointeur (distant) sur le prochain serveur.

```
package Controller is
   pragma Remote_Call_Interface;

   type Job_Type is ...;
   type Result_Type is ...;

   type Return_Address is access procedure (Result : Result_Type);

   type Worker_Ptr     is access procedure (Job: Job_Type; Ret : Return_Address);

   procedure Register_Worker (Ptr  : in  Worker_Ptr;
                              Next : out Worker_Ptr);
   procedure Give_Results (Result : Result_Type);
end Controller;
```

```
with Controller; use Controller;
generic
package Worker is
   pragma Remote_Call_Interface;

   procedure Do_Job (Job : Job_Type; Ret : Return_Address);
   procedure Pass_Back_Results (Result : Result_Type);
end Worker;

package body Worker is
   Next_Worker     : Worker_Ptr;
   Previous_Worker : Return_Address;

   procedure Do_Job (Job : Job_Type; Ret : Return_Address) is
   begin
      Previous_Worker := Ret;
      if This_Worker_Too_Busy and then Next_Worker /= null then
         Next_Worker(Job, Pass_Back_Result'Access);
      else
         declare
            Result : Result_Type;
         begin
            Do_The_Work(Job, Result);
            Previous_Worker(Result);
         end;
      end if;
   end Do_Job;

   procedure Pass_Back_Results (Result : Result_Type) is
   begin
      Previous_Worker(Result);
   end Pass_Back_Results;

begin
   Controller.Register_Worker(Do_Job'Access, Next_Worker);
end Worker;

package Worker_1 is new Worker;
package Worker_2 is new Worker;
   ...
package Worker_9 is new Worker;
```

Remarquer que l'on peut alors créer autant de serveurs que l'on souhaite par une simple instantiation de générique, et les placer ensuite dans des partitions différentes.

VI.4 Modèle objet distribué

Le même principe que nous avons vu pour les sous-programmes normaux et les pointeurs s'applique aux pointeurs sur classe. Dans ce cas, les appels seront non seulement distants, mais aussi dynamiques. Autrement dit, la liaison dynamique pourra s'effectuer de façon transparente à travers le réseau sans que l'utilisateur ait à s'en préoccuper.

Dans l'exemple ci-dessous, on définit d'abord une classe de "dérouleur de bande". Différentes implémentations peuvent en être dérivées, que l'on attribuera éventuellement à différents noeuds d'un réseau. Tout serveur de bande s'enregistre auprès d'un serveur de nom. L'utilisateur récupère *via* le serveur de nom un pointeur (distant) sur l'objet désiré. Tous les appels aux méthodes de cet objet seront donc routés à travers le réseau vers l'objet considéré.

```
package Tapes is
   pragma Pure(Tapes);

   type Data is ...;
   type Tape is abstract tagged limited private;

   procedure Rewind (T :  access Tape) is abstract;
   procedure Read   (T :  access Tape; Value : out Data) is abstract;
   procedure Write  (T :  access Tape; Value : in  Data) is abstract;

private
   type Tape is abstract tagged limited null record;
end Tapes;

with Tapes;
package Name_Server is
   pragma Remote_Call_Interface;

   type Tape_Ptr is access all Tapes.Tape'Class;

   function  Find     (Name : String)      return Tape_Ptr;
   procedure Register (Name : String; T : Tape_Ptr);
   procedure Remove   (T    : Tape_Ptr);
end Name_Server;

package Tape_Driver is
   pragma Elaborate_Body(Tape_Driver);
end Tape_Driver;

with Tapes, Name_Server;
package body Tape_Driver is

   type New_Tape is new Tapes.Tape with ...
   procedure Rewind (T : access Tape) is ...
   -- Redéfinition de Rewind
   procedure Read (T : access Tape; Value : out Data) is ...
   -- Redéfinition de Read
   procedure Write (T : access Tape; Value : in Data) is ...
   -- Redéfinition de Write

   Tape1, Tape2 : aliased New_Tape;
begin
   Name_Server.Register ("Tape 1", Tape1'Access);
   Name_Server.Register ("Tape 2", Tape2'Access);
end Tape_Driver;

with Tapes, Name_Server;
procedure Client is
   T1, T2 : Name_Server.Tape_Ptr;
   Buffer : Data;
begin
   T1 := Name_Server.Find ("Tape 1");
   T2 := Name_Server.Find ("Tape 2");
   Tapes.Rewind(T1); Tapes.Rewind(T2);
   Tapes.Read(T1, Buffer);
   Tapes.Write(T2, Buffer);
end Client;
```

On notera que ce modèle est parfaitement compatible avec le modèle CORBA. Les schémas d'implémentation de CORBA avec Ada 95 sont en cours de standardisation, et plusieurs fournisseurs ont annoncé des outils de traduction d'IDL vers Ada.

VI.5 Cohérence d'un système distribué

Le problème de la cohérence d'un système distribué est délicat, car il doit répondre à deux exigences contradictoires:

- d'une part, il faut garantir que toutes les partitions du système "voient" les données communes de la même façon;
- d'autre part, il faut pouvoir arrêter et relancer une partition, éventuellement pour corriger une erreur, sans être obligé d'arrêter tout le système.

Un programme Ada est dit "cohérent" si toutes les partitions qui le constituent utilisent la même version de toutes les unités de compilation. On autorise un programme Ada à être incohérent, c'est à dire qu'il est possible de corriger une erreur qui n'affecte qu'une seule partition, d'arrêter la partition et de la relancer avec la nouvelle version. Toutefois, si la correction implique un paquetage "partagé passif" ou RCI (les seuls qui ne soient pas dupliqués), alors toute communication est coupée entre la nouvelle partition et le reste du système. On autorise donc un système *localement incohérent*, tant que cela ne sort pas de la partition en cause; la cohérence est exigée pour les seuls éléments qui sont physiquement répartis sur le réseau.

De plus, il est attaché à chaque unité de compilation un attribut "VERSION" (pour la spécification) et "BODY_VERSION" (pour le corps), dont la valeur change à chaque modification. Il est donc possible de programmer des contrôles de cohérence plus fins si cela est nécessaire.

VII L'état de l'art

La standardisation est acquise à l'heure actuelle: le document définitif a passé toutes les étapes de vote et est en cours de publication à l'ISO. La première suite de validation comportant les tests des nouveautés Ada 95 sera mise en service au mois de mars 1995.

A titre transitoire pour permettre aux constructeurs de répondre plus rapidement aux besoins spécifiques de leur clientèle, on autorisera des implémentations partielles des nouveautés Ada 95, la totalité des fonctionnalités d'Ada 83 restant obligatoire. Les fabricants de compilateurs ont annoncé leurs plans de transition, certains mettant l'accent d'abord sur les aspects POO, d'autres sur les fonctionnalités temps-réel. Certains constructeurs proposent déjà des compilateurs fournissant au moins une partie des nouvelles possibilités.

Un des obstacles importants à la diffusion d'Ada a longtemps été le prix des compilateurs, notamment pour les institutions universitaires. Il est vrai que les fournisseurs ont consenti un important effort financier pour les organismes d'enseignement, mais le meilleur moyen de convaincre les gens d'utiliser Ada est de leur faire essayer le langage[8]. Or, peu d'enseignants (aux crédits limités!) sont prêts à acheter un compilateur, même à prix réduit, juste pour voir...

Conscient de cet état de fait, le DoD a financé le développement d'un compilateur Ada 95 dont la diffusion est entièrement libre et gratuite. Il s'agit en fait d'un frontal du célèbre compilateur multi-langage GCC, faisant partie de l'environnement GNU de la Free Software Foundation, connu sous le nom de GNAT (GNU Ada Translator). La réalisation en a été confiée à la fameuse équipe de New York University, qui s'était déjà illustrée en réalisant le premier compilateur Ada (83) validé.

Comme tous les logiciels diffusés par la Free Software Foundation, le GNAT est disponible non seulement sous forme exécutable, mais aussi sous forme de sources. Tous les enseignants des cours de compilation peuvent ainsi offrir à leurs élèves d'intervenir dans un compilateur Ada, c'est à dire dans ce qui se fait de mieux en matière de compilation.

Le GNAT est disponible par les canaux de diffusion habituels du GNU, en particulier on peut le charger par FTP anonyme depuis tous les bons serveurs de logiciels du domaine public (en France, sur le serveur de l'université Paris VI: ftp.ibp.fr, répertoire pub/gnat).

[8] Nous avons souvent constaté que les plus farouches opposants à Ada ne l'avaient jamais essayé...

VIII Conclusion

Le langage Ada, dans sa nouvelle version, propose la synthèse des qualités démontrées d'Ada 83 et des nouvelles tendances de l'informatique. Il est le produit d'une longue réflexion sur la base des besoins des utilisateurs pour fournir un langage lisible, fiable et efficace répondant aux besoins de l'informatique moderne.

Ada offre des outils spécifiques, et un réel support aux besoins de la programmation en temps-réel. C'est le premier langage orienté objet standardisé et le premier langage à intégrer un modèle d'exécution distribuée indépendant de tout système d'exploitation, tout en conservant les atouts d'Ada 83: contrôles stricts à la compilation, cohérence des types, facilités de traitement des situations exceptionnelles et parallélisme intégré au langage. Tout développeur temps-réel se doit d'évaluer l'opportunité de l'utiliser lors de la phase des choix technologiques de son projet.

IX Bibliographie

[Afn87] *AFNOR: Langages de programmation-Ada, norme NF EN 28652. Eyrolle, Paris 1987.*

[Bar88] *G.J.P. Barnes, Programmer en Ada, InterEditions, Paris, 1988.*

[Bar94] *S. Barbey, M. Kempe et A. Strohmeier, "La programmation par objets avec Ada 9X", TSI Vol 13 n°5, HERMES Paris, 1994.*

[Ber89] *J.-M. Bergé, L.-O. Donzelle, V. Olive, J. Rouillard, Ada avec le sourire, Presses Polytechniques Romandes - Collection CNET-ENST, Lausanne 1989.*

[Bek94] *D. Bekele et J-M. Rigaud, "Ada et les systèmes répartis", TSI Vol 13 n°5, HERMES Paris, 1994.*

[Boo88] *G. Booch, Ingénierie du logiciel avec Ada, InterEditions, Paris, 1988 (traduction française de J.P. Rosen).*

[Bur 85] *A. Burns, Concurrent Programming in Ada (Ada Companion Series), Cambridge University Press, Cambridge, TBSL*

[Gau89] *M. Gauthier, Ada: un apprentissage, Dunod, Paris 1989.*

[Ket93] *N. Kettani, "Ada 9X: Object Oriented Programming with Inherant Qualities of Ada", Actes des journées "Génie Logiciel et ses Applications", EC2, Nanterre 1993.*

[Mor82] *Morrison, Morrison, Eames: Powers of Ten, Scientific American Library, New York 1982.*

[Ros92] *"Ada 9X : une évolution, pas une révolution", Techniques et science informatiques, Volume 11 n°5/1992, HERMES, Paris.*

[Ros94] *J-P. Rosen, N. Kettani, "Apport d'Ada 9X aux paradigmes orientés objet", Acte des Journées Internationales sur les Nouveautés en Génie Logiciel, C3ST, Courbevoie 1994*