ECOLE NATIONALE SUPERIEURE DES TELECOMMUNICATIONS

THESE

présentée par

Jean-Pierre ROSEN

pour obtenir

le grade de **DOCTEUR DE L'ENST**

Spécialité: Systèmes Informatiques

Sujet de la thèse:

Une machine virtuelle pour Ada: Le système d'exploitation

Soutenue le 2 Octobre 1986 devant la commission composée de:

Mr. G. Pujolle	Président
Mr. R.B.K. Dewar	Directeur de thèse
Mme. V. Donzeau-Gouge **Mr. E. Morel**	Rapporteurs
Mr. D. Chandesris **Mr. M. Gauthier** **Mme C. Nora** **Mr. O. Roubine**	Examinateurs

Je tiens à exprimer toute ma reconnaissance à Monsieur Guy PUJOLLE qui me fait l'honneur de présider cette thèse.

Que les membres du Jury, Mme Véronique DONZEAU-GOUGE, Mr. Etienne MOREL, Mr. Dominique CHANDESRIS, Mr. Michel GAUTHIER, Mr. Olivier ROUBINE, et Mme Christine NORA reçoivent ici tous mes remerciements pour avoir bien voulu consacrer une part de leur temps précieux à l'étude de ce travail.

Je tiens à exprimer toute ma gratitude à l'équipe du projet NYUADA à New York University, et plus spécialement Mrs Robert DEWAR, Edmond SCHONBERG, et Dave SHIELDS dont les compétences et le contact stimulant m'ont beaucoup aidé dans la réalisation de ce projet.

Que l'ENST, et plus particulièrement son directeur de l'époque Mr Guy MALLEUS, trouvent ici l'expression de ma reconnaissance pour m'avoir permis de prendre le congé sabbatique grâce auquel ce travail a pu être réalisé.

Ma reconnaissance toute particulière va à Philippe KRUCHTEN qui m'a invité à participer à ce projet et sans lequel cette thèse n'aurait jamais vu le jour.

Merci enfin à Gerry FISHER, dont les fructueuses discussions m'ont permis de mettre au point nombre d'algorithmes rapportés ici, ainsi qu'à Jérome CHIABAUT qui m'a aidé dans la réalisation de ce projet.

A mes parents

A Catherine

A Jérémy et Cécilia

A mes amis

Plan de la thèse

III. Entrées-sorties

IV. Modèle de système de gestion des tâches

V. Implémentation du système de gestion des tâches

Appendices

Bibliographie

Introduction

1. Le langage de programmation Ada (R)

1.1 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 coûts des logiciels militaires. Le DOD est en effet le plus grand consommateur de logiciel du globe, et il utilisait à ce moment-là environ 400 langages de programmation et dialectes differents. 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 (*embedded*), c'est-à-dire des systèmes informatiques qui ne sont qu'un composant d'un ensemble électro-mécanique plus vaste: systèmes d'armes, de radar, de commutation... 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. Chacun de ces rapports, connus sous les noms de Strawman, Woodenman, Tinman, Ironman, Revised Ironman, a suscité un intérêt considérable tant au sein du DOD qu'à l'extérieur, et a été pensé et repensé et modifié jusqu'à aboutir au cahier des charges Steelman [DoD 78].

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: Softech, Intermetrics, SRI et Cii-Honeywell-Bull. En mars 1978, il ne restait plus en lice

---

(R) Ada est une marque déposée du Département Américain de la Défense (AJPO)

qu'Intermetrics et Honeywell, l'équipe francaise dirigée par Jean Ichbiah remportant finalement l'appel d'offre un an plus tard, le 2 Mai 1979. Le "Langage Vert" est alors rebaptisé *Ada*, du nom de Ada Byron, fille du poète anglais, considérée comme le premier programmeur de l'histoire pour ses travaux sur la *Machine Analytique Mécanique* de Charles Babbage. (Sources: [Barnes 84])

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, et l'on pensait à l'époque que le langage pourrait être normalisé avant la fin de cette même année. En fait, de nombreux problèmes apparurent dans ce document, et il fallut deux ans de discussions entre experts venus du monde entier pour parvenir à un nouveau document en Juillet 1982. Après encore quelques modifications de moindre importance, le langage fut standardisé par l'ANSI[1]. Ce dernier document, gros d'environ 300 pages, qui constitue la seule définition officielle du langage, est connu sous le nom de "Reference Manual for the Ada Programming Language" [DoD 83]. Nous y ferons abondamment référence ici sous la forme désormais consacrée par l'usage: [LRM c.s.p(a)]. LRM est l'abréviation de *Language Reference Manual,* et c, s, p et a désignent respectivement les numéros de chapitre, section, paragraphe et alinéa.

Il était important de définir rigoureusement le langage; encore fallait-il être sûr que les compilateurs respectaient la norme! pour cela, le DOD a déposé le nom "Ada", et l'autorisation d'utiliser ce nom pour un compilateur est subordonnée à la confrontation avec succès du compilateur à une suite de validation, développée par J. Goodenough et son équipe à Softech [Goodenough 81]. Le premier compilateur Ada ainsi validé fut Ada/ED, réalisé par l'équipe de New York University. De nombreux compilateurs ont été validés depuis.

1.2 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:

---

1 American National Standard Institute

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'execution permettront de diagnostiquer les erreurs dynamiques.

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 de systèmes "temps réel".

Le but ultime du langage est de permettre l'émergence d'une industrie du *composant logiciel*. De quoi s'agit-il? L'électronicien qui souhaite construire un circuit logique ne reconçoit pas à chaque fois les portes dont il a besoin: il achète des *composants* qui lui fournissent les fonctions logiques nécessaires. C'est d'ailleurs grâce à cette notion de composant réutilisable, à faible coût car produit à un grand nombre d'exemplaires, que l'industrie électronique a pu prendre le développement que nous lui connaissons aujourd'hui. Or il faut bien reconnaître que cette démarche est actuellement inexistante dans le monde du logiciel. Combien de programmes de tri sont-ils réécrits chaque année? Ada, en fournissant des interfaces standard et des garanties de portabilité des applications indépendamment des machines, permettra la création d'entreprises spécialisées en composants logiciels; d'autres entreprises réaliseront alors des produits finis (applications) par assemblage de composants logiciels. L'industrie du logiciel ne fait que redécouvrir les méthodes qui ont fait le succès de bien d'autres branches industrielles, avec quelques dizaines d'années de retard...

1.3 Au delà du langage

La définition d'un nouveau langage est une condition nécessaire, mais non suffisante, pour entrer dans une ère de production industrielle de logiciel. Aussi ne peut il exister de compilateur Ada sans un environnement de programmation associé; en particulier, le

langage définit des règles très strictes concernant les dépendances entre unités de compilation, pour garantir la cohérence globale des différentes unités constituant un programme Ada. Ceci nécessite au moins un outil de gestion de projet associé. En fait, parallèlement à la définition du langage, se poursuivait un effort de définition de l'APSE (*Ada Programming Support Environment*) qui a abouti au rapport STONEMAN. Celui-ci définit les outils logiciels qui doivent faire partie d'un environnement de programmation Ada.

Aujourd'hui que des compilateurs de qualité industrielle existent sur le marché, un grand effort est entrepris en ce qui concerne les outils d'environnement. Les objectifs d'Ada sont en fait ceux de tout le mouvement du génie logiciel, et le langage n'est qu'un élément qui s'intègre dans un ensemble, notamment méthodologique, plus vaste.

1.4 Présentation sommaire du langage.

Ada est un langage algorithmique d'un puissance d'expression considérable, dérivé de Pascal dont il a retenu les structures de contrôle et certains types de données. On trouvera dans [Rosen 86] par exemple une présentation générale du langage. Disons simplement ici que ses points originaux sont la définition précise des règles de l'arithmétique utilisée sur les valeurs approchées; la notion de *paquetage* permettant de regrouper dans une enveloppe commune des types abstraits et les opérations entre objets de ces types; la définition d'un mécanisme de traitement des situations exceptionnelles; la possibilité de définir des traitements parallèles avec gestion du temps réel; et la possibilité de définir des unités génériques permettant d'établir des algorithmes réutilisables indépendamment des types de données manipulées.

Ada n'est pas un langage de *très* haut niveau, pas du niveau de LISP, SNOBOL, SETL, ou PROLOG en tous cas, mais c'est un langage qui a un spectre sémantique assez vaste, et qui, plutôt que d'être un langage révolutionnaire, représente une synthèse de l'expérience acquise avec les langages procéduraux "classiques" qui l'ont précédé.

2. Le projet NYUADA

Le projet NYUADA[3] a démarré en 1978 au sein du département d'informatique du Courant Institute of Mathematical Sciences de New York University, sous l'égide de Robert Dewar, Gerald Fisher et Edmond Schonberg.

L'objectif initial du projet était de faire une petite étude des problèmes d'optimisation posés par le langage Ada. Il s'est très vite avéré à l'époque que le langage était assez mal défini [Ichbiah 79a, Ichbiah 79b] et qu'une spécification plus rigoureuse était indispensable si l'on voulait aborder les problèmes d'optimisation. Ceci a conduit à un premier *modèle sémantique exécutable d'Ada* consistant en approximativement 2000 lignes de SETL, décrivant l'aspect sémantique dynamique sous forme d'un interprète exécutant une représentation intermédiaire appelée AIS (*Ada Intermediate Source*). Deux mois plus tard, un analyseur lexical et syntaxique y était ajouté, permettant de générer de l'AIS pour des programmes Ada corrects seulement. Puis au cours des 6 mois qui suivirent, l'analyse de la sémantique statique du langage a été progressivement ajoutée, et lorsque la définition d'Ada 1980 a été publiée [DoD 80], le traducteur Ada/Ed ("Ed" pour "*Educational*") était déjà un modèle exécutable d'une majeure partie de sa sémantique statique et dynamique. Plusieurs des aspects du langage qui n'étaient pas implémentables au vu de leur définition en "Preliminary Ada" (comme les discriminants et la dérivation des sous-programmes) avaient été laissés de côté, et ajoutés lors de la parution de la proposition de norme Ada en 1980. Par contre, la sémantique des processus parallèles: activation de tâches, rendez-vous, instructions **abort** et **select** avait été réalisée dès le début [Dewar 80].

A ce point, il était devenu clair aux yeux de tous qu'Ada/Ed pouvait remplir deux fonctions: celle de traducteur type, et celle de définition rigoureuse du langage, cette dernière étant une conséquence directe:

a) de la concision du système (25000 lignes de SETL, documentation comprise) [NYU 83a, 83b]
b) du modèle très abstrait choisi pour représenter l'environnement d'exécution [Kruchten 84b]
c) de la démonstration de la conformité d'Ada/Ed à la suite de validation, qui constitue *de facto* une définition supplémentaire d'Ada.

---

[3] Ce projet a été financé par l'armée de terre des Etats-Unis, sous contrat numéro DAAB027-82-K-J196 du CORADCOM à Fort Monmouth, N.J., et par l'Ada Joint Program Office.

L'effort de NYUADA s'est donc poursuivi dans ces deux directions. En à peu près 6 mois, Ada/Ed a été modifié pour s'adapter au modifications apportés par le nouveau Manuel de Référence publié en Juillet 1982 [DoD 82], puis à nouveau début 1983 pour refléter les dernières modifications contenues dans la norme ANSI de janvier 1983 [DoD 83]. Finalement Ada/Ed a été validé officiellement le 11 Avril 1983 par l'AJPO (*Ada Joint Program Office*). De son démarrage en 79 jusqu'à sa validation, ANSI-Ada/Ed n'aura coûté que 100 hommes-mois.

La preuve ultime de l'utilité d'un prototype est bien entendu la construction d'un système de production *grandeur nature* qui utilise ce prototype comme modèle. Depuis la première validation, le projet NYUADA s'est engagé dans la voie de la construction d'un traducteur considérablement plus performant, écrit d'abord en SETL, puis finalement en C, utilisant le premier système constamment mis à jour comme spécification de réalisation.

Ce système devait pouvoir être utilisé pour l'enseignement d'Ada et l'expérimentation à petite échelle. Il devait être simple d'emploi, robuste, et bon marché, pour lui assurer une large diffusion. Il devait dans une deuxième étape servir de prototype à une implémentation plus efficace destinée à des micro-ordinateurs personnels, genre IBM PC.

Il n'était pas dans les objectifs du projet de réaliser un système *performant*, le seul critère de performance fixé était de gagner deux ordre de grandeur (un facteur 100) en temps d'exécution par rapport au système Ada/Ed originel, celui validé en avril 1983, pour le prototype intermédiaire en SETL, puis un nouvel ordre de grandeur par la réécriture en C. Compte tenu des performances extrèmement faibles du système d'origine, ces chiffres n'étaient pas suffisants pour un compilateur industriel, mais étaient acceptables pour un système à but pédagogique.

Ces objectifs à la fois méritent quelques explications et imposent d'emblée quelques choix de conception:

- Une réalisation rapide et économique: le projet NYUADA n'était constitué que d'une demi-douzaine de personnes, dont la moitié ne travaillaient pas à plein temps sur le projet; d'autre part une partie de la force de travail devait être consacrée à la maintenance du système Ada/Ed initial.
- Un système destiné à l'enseignement et à l'expérimentation: c'était à la fois la vocation du Courant Institute (qui n'est pas une société commerciale de logiciel, mais un institut de recherche au sein d'une université) et l'objectif fixé par

l'organisme qui a supporté financièrement ce projet: l'armée de terre des Etats-Unis (le CORADCOM, à Fort Monmouth, dans le New Jersey).

- Portabilité sur des petites machines: afin d'assurer une large diffusion d'un traducteur Ada complet dans les universités, collèges, entreprises, et par là étendre rapidement la connaissance de ce langage, clé de son succès. Il existait un précédent dans ce sens: le compilateur Pascal UCSD (Université de Californie à San Diego).

La figure ci-dessous montre comment le système a évolué du prototype initial en SETL au système opérationnel final en C, en ne faisant évoluer que certains éléments à la fois, avec des interfaces fixes, bien définies, de façon à avoir constamment un système exécutable et vérifiable [Kruchten 85], [Schonberg 86].

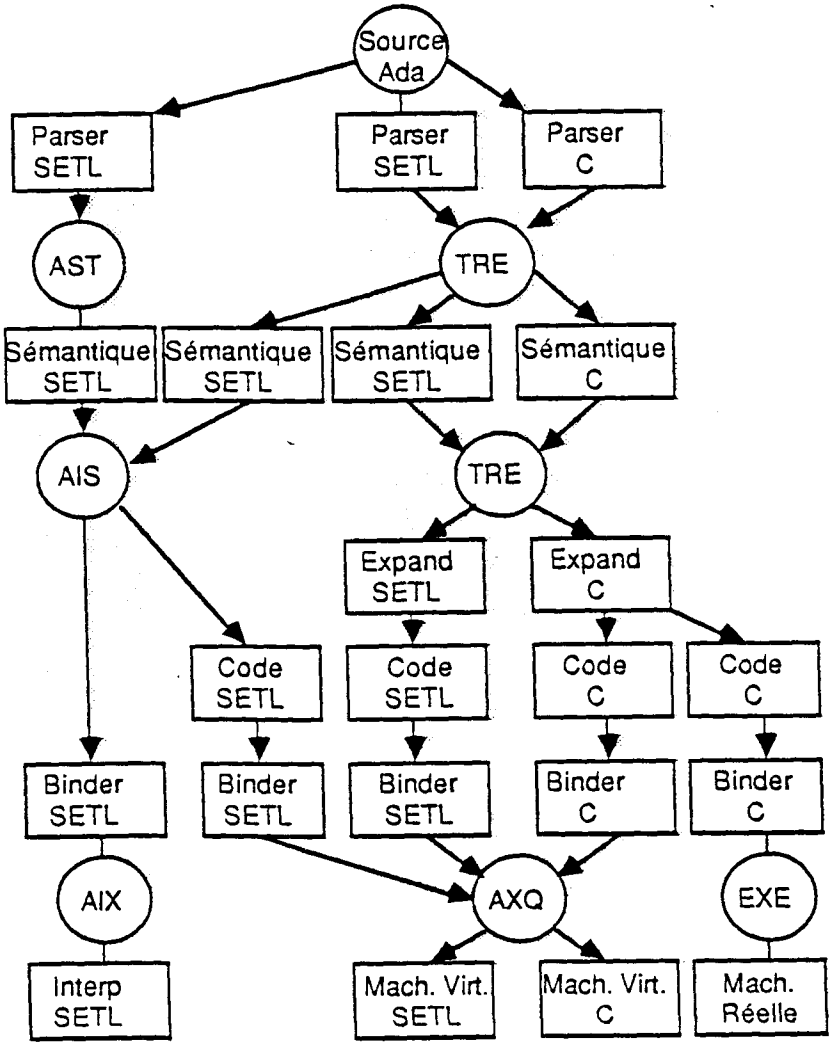


Figure 1: évolution du système Ada/Ed

Cette figure montre aussi bien les différentes transformations subies par un programme Ada (dans le sens vertical) que l'évolution du système Ada/ED (dans le sens horizontal). Le système de "haut niveau" constitue la partie la plus à gauche: il est constitué d'une phase d'analyse syntaxique et sémantique produisant une représentation intermédiaire arborescente (AIS, Abstract Intermediate Source), et d'un interprète de haut niveau exécutant directement l'arbre. Cet interprète travaillait en fait en produisant lui même du code "à la volée" ([NYU 83b], [Kruchten 85]). Il a donné naissance à un générateur de code pour une machine virtuelle de bas niveau (la Machine Ada), et à l'interprète correspondant. Noter que ceci a nécessité l'ajout d'une nouvelle phase intermédiaire au compilateur. La conservation de la compatibilité AIS à ce niveau a permis de développer simultanément le générateur de code en utilisant l'ancien frontal, et un nouveau frontal de bas niveau utilisant l'interprète de haut niveau pour la mise au point.

Par la suite, la phase de génération de code fut séparée en deux passes, travaillant directement sur les structures internes du frontal de bas niveau (TRE). Le maintien de la compatibilité au niveau des structures a de la même façon permis le développement en parallèle d'un analyseur syntaxique, puis d'un analyseur sémantique, écrits dans le langage C. A tout moment, les développeurs d'une phase du système pouvaient utiliser les versions précédentes et éprouvées du système pour les autres phases.

Finalement, fin Janvier 1985, lorsqu'il apparut que les problèmes étaient suffisamment bien maîtrisés, le modèle SETL fut figé, et tout le système fut traduit en langage C. Cette dernière version, qui constitue le produit final Ada/Ed-C dont les versions précédentes n'étaient que des prototypes intermédiaires, a été validée officiellement par l'AJPO au printemps de 1986. Elle est disponible sur Vax Unix ou VMS, SUN et ELXSI. Sa validation sur IBM/PC est imminente à la date où nous écrivons ces lignes.

La partie la plus à droite de la figure montre enfin comment un nouveau générateur de code pourrait être "greffé" sur le système pour obtenir un compilateur générant du code natif pour une machine cible réelle.

3. Motivations et objectifs de cette étude

L'étude que nous avons menée en collaboration avec P. Kruchten consistait en la réalisation du prototype SETL de l'interprète de bas niveau. Notre objectif principal était de réaliser rapidement et de facon la plus économique possible un système de traduction portable et complet, s'appuyant sur le modèle constitué par le système déjà validé, mais de plus bas niveau sémantique, de façon à réduire le "saut sémantique" entre l'interprète de

haut niveau et le produit final en C. En particulier, nous devions définir toute la structure de la machine et tous les algorithmes à mettre en oeuvre pour réaliser les exigences de [DoD 83] en bénéficiant des avantages d'un langage de très haut niveau, afin de réduire l'effort du codage final à une simple traduction.

Le sujet de la présente thèse est le système d'exploitation qui permet de gérer la machine virtuelle. Nous avons regroupé sous le terme "système d'exploitation" tout l'ensemble des services qui font partie de l'Ada Machine, mais qui seraient implémentés sous forme logicielle sur une machine réelle. Ceci couvre principalement le système de gestion des tâches, les entrées-sorties, ainsi que l'arithmétique réelle, et plus spécialement l'arithmétique point fixe.

La structure "matérielle" de la Machine Ada est le sujet de la thèse de P. Kruchten [Kruchten86]. Nous nous contenterons de présenter dans le premier chapitre les grandes lignes de la conception de la Machine Ada, et précisant uniquement les notions qui sont utiles à la compréhension du fonctionnement du système d'exploitation. Nous ne saurions trop recommander au lecteur intéressé par la structure de cette machine de se reporter à [Kruchten 86], et nous y ferons nous mêmes de nombreux renvois dans le texte de cette thèse.

4. Organisation du document

Une part importante de ce document est la traduction en français de la documentation que nous avons laissée aux membres du projet NYUADA pour leur permettre de compléter et de maintenir le prototype existant, et d'en comprendre les mécanismes pour pouvoir les transporter dans un langage de plus bas niveau. Certains textes sont repris de diverses publications en anglais ou en français, cf. [Rosen 83,84a,84b,85].

Le chapitre 1 rappelle brièvement les caractéristiques de la Machine Ada qui sont nécessaires à la compréhension du fonctionnement du système d'exploitation; le chapitre 2 présente l'implémentation de l'arithmétique, et plus particulièrement de l'aspect le plus nouveau, l'arithmétique point fixe. Le chapitre 3 présente les mécanismes d'entrées/sorties. Le chapitre 4 développe le modèle de système de gestion des tâches que nous avons élaboré, et le chapitre 5 décrit son implémentation effective dans la Machine Ada.

En annexe, on trouvera l'"appendice F" de notre implémentation décrivant ses particularités selon le format exigé par la norme, une description de notre outil de travail

principal, déjà mentionné à plusieurs reprises: le langage SETL, ainsi qu'une rapide biographie de celle qui donna son nom au langage: Lady Ada Augusta Byron, comtesse de Lovelace.

Chapitre I.
Présentation de la Machine Ada

Ce chapitre présente de façon très succincte la Machine Ada et quelques mécanismes spécifiques auxquels il est fait allusion dans le reste de la thèse. Pour une description plus détaillée, nous invitons le lecteur à se reporter à [Kruchten 86].

1. Architecture générale

La Machine Ada est un ordinateur relativement "classique", à architecture de type Von Neumann, composée d'une unité centrale, munie de registres spécifiques, et d'une mémoire. Elle exécute des instructions réparties en plusieurs classes: instructions de contrôle, instructions arithmétiques, appels système. Des instructions de plus haut niveau sémantiques sont plus directement liées au langage Ada: élaboration de types, évaluation d'attributs, allocation mémoire et création d'objets, gestion des tâches.

On notera l'absence de la notion de périphérique et de toute instruction directement liée aux entrées-sorties: s'agissant d'une machine simulée, les entrées-sorties sont directement prises en compte par des appels "système", et n'apparaissent pas directement. La machine se présente donc comme une machine ayant deux modes de fonctionnement, normal et privilégié, le mode privilégié étant réservé aux appels systèmes, et donc totalement caché du point de vue du jeu d'instructions normal.

La Machine Ada est une machine "pile": elle ne dispose pas de registres banalisés, toutes les opérations s'effectuant à partir ou à destination de la pile d'exécution. Mise à part sa fonction de stockage intermédiaire pour l'évaluation d'expressions arithmétiques, la pile contient, dans sa partie basse, le *bloc de contrôle de tâche* qui définit le contexte logiciel de la tâche qui s'exécute, et des *environnements d'appel* et *environnements de*

bloc qui décrivent le contexte procédural dynamique. Le bloc de contrôle de tâche est décrit en détails dans le chapitre sur le système de gestion des tâches de la présente thèse. Nous donnerons une description succincte des environnements de bloc dans la suite de ce chapitre.

La mémoire de la Machine Ada est segmentée, et il existe quatre types de segments: les segments de pile (cf. ci-dessus), les segments de données qui contiennent les objets alloués statiquement, les segments de code contenant les instructions machine correspondant aux différentes unités de programme, et le tas, qui contient tous les autres objets: variables dynamiques (types **access** de Ada) et variables locales aux procédures.

La principale originalité de la Machine Ada tient à son mécanisme d'allocation et d'adressage des variables locales aux procédures. Contrairement à ce qui se fait habituellement dans les machines adaptées à des langages à structure de blocs, la pile ne contient pas directement les variables locales, mais uniquement leurs adresses, les variables étant toujours allouées dans le tas. Une règle fondamentale est que la pile contient des *valeurs* ou des *adresses*; elle ne contient jamais d'*objets*. Tout accès à une variable locale passe donc obligatoirement par une déréférence. La justification complète de ce mécanisme se trouve dans [Kruchten 86]. Notons simplement ici qu'il simplifie énormément l'accès aux objets partagés entre tâches en évitant d'avoir à gérer une pile "cactus" [Ibsen 83,84], et permet l'implémentation d'un mécanisme totalement original pour l'accès aux variables non globales, non locales [Kruchten 86].

2. Notion de patron de type

Une des particularités du langage Ada est de permettre une grande dynamicité dans la définition des (sous-)types, tout en gardant des contrôles très stricts. Du point de vue du compilateur, ceci signifie qu'une grande partie des éléments nécessaires à la génération de contrôles, et même parfois à l'adressage des éléments d'un objet composite, n'est pas connue au moment de la compilation. Il est donc nécessaire de définir des descripteurs de type qui seront utilisés pour ces vérifications ainsi que pour l'adressage d'objets composites. Ces descripteurs sont appelés des *patrons de type* dans la Machine Ada. Les patrons de type générés par le compilateur sont incomplets; c'est au moment de l'*élaboration* du type qu'ils seront garnis avec les informations dynamiques nécessaires, et pourront dès lors être utilisés.

Il existe différentes sortes de patrons de types correspondant aux différentes grandes classes de types définies par le langage: entiers, point fixe, point flottant, énumération,

tableau, enregistrement, accès, tâches. Tous les patrons de type ont au moins deux champs en commun: *kind*, qui est le discriminant entre les différentes sortes de patrons, et *size*, taille des objets appartenant au type.

Les sous-programmes et tâches sont également décrits par des patrons de type. Ceux-ci sont garnis partiellement au moment de l'élaboration de la spécification de sous-programme ou de tâche, mais les informations nécessaires à l'accès aux objets non locaux non globaux ne peuvent être obtenues qu'au moment de l'élaboration du corps correspondant. Nous avons utilisé pour celà le fait qu'il n'est pas possible d'appeler un sous-programme ou d'activer une tâche avant élaboration de son corps [LRM 3.9(5-6)].

3. Environnements d'appel et environnements de bloc

Un environnement d'appel est une structure particulière qui est créée sur la pile lors d'un appel de sous-programme. Il sert à passer les paramètres au sous-programme appelé, ainsi que les variables non locales, non globales, qui sont traitées comme des paramètres [Kruchten 86]. Il réserve également l'espace dans la pile correspondant aux adresses de toutes les variables locales du sous-programme, y compris les variables déclarées dans des *instructions bloc* internes.

Un environnement de bloc est également une structure allouée sur la pile. Il est créé lors de l'élaboration d'une partie déclarative. Il permet de mettre en place le traite-exception courant [Kruchten 86], et contient des têtes de chaînes nécessaires à l'activation des tâches créées par l'élaboration de cette partie déclarative, ainsi que celles nécessaires à la gestion de la terminaison des tâches dont cette partie déclarative est la *construction maîtresse*. Ces chaînages sont décrits en détail dans la partie de cette thèse consacrée au système de gestion des tâches.

4. Particularités de réalisation dues à l'usage de SETL

Le langage SETL est un langage de très haut niveau, qui essaie d'affranchir autant que possible le programmeur des contraintes liées à la machine. Dans cet esprit, les objets de type entier ne sont pas limités; SETL fournit automatiquement une arithmétique étendue. Cette facilité n'est pas utilisable, et est même parfois gènante, lorsqu'il s'agit de simuler une machine dont le nombre de bits par mot mémoire est fini. Nous avons donc du définir nous-mêmes des notions telles que le mot mémoire ou l'octet, en vérifiant éventuellement

que les valeurs mises dans les structures correspondantes ne dépassaient pas les valeurs autorisées. Mots et octets sont représentés par des entiers SETL.

Dans l'élaboration de notre modèle arithmétique, nous avons été amenés à définir des objets plus longs que des entiers standard. Ils ont été modèlisés par des paires d'entiers (type *xleng*), et même dans certains cas par des doubles paires (type *double_xleng*), ceci de façon à préparer la version future en C où seul le type INTEGER limité serait disponible.

Ainsi, malgré les possibilités d'arithmétique étendue de SETL, nous avons ramené la mémoire de la machine à une mémoire d'octets ou de mots conventionnelle. Les facilités de SETL resteront cependant apparentes dans un cas: lorsque nous parlerons de l'encombrement d'objets, nous donnerons le nombre de positions de tuple occupés par l'objet, étant entendu que chaque position de tuple peut être occupée soit par un mot, soit par un octet.

Chapitre II.
Arithmétique réelle[1]

Cette section décrit les propriétés des nombres appelés "réels" en Ada, et la façon dont ils ont été implémentés dans l'Ada Machine. Les paragraphes du manuel de référence concernés vont de 3.5.6 à 3.5.9.

1. Nombres réels et analyse numérique.

Pour le mathématicien, le terme "nombre réel" désigne un élément de l'ensemble infini R. Il n'est cependant bien entendu possible que de représenter un nombre *fini* de valeurs sur une machine. Il est donc nécessaire d'extraire un sous-ensemble fini de valeurs de R, sur lequel des calculs pourront être effectués, et un des rôles de l'analyse numérique est précisément de déterminer l'influence de cette restriction sur la validité des résultats obtenus.

Peu de langages définissent le sous-ensemble de R utilisé effectivement pour faire des calculs, car une telle spécification risquerait d'être trop contraignante pour certains matériels, et donc de conduire à des implémentations inefficaces. Mais le prix à payer pour cette absence de contraintes est l'absence totale de garantie quant à la précision des résultats obtenus en exécutant un même programme sur des matériels différents.

Le langage Ada se devait d'être à la fois portable *et* efficace. Il fallait donc définir un moyen terme entre ne rien définir du tout (comme en FORTRAN) ou définir exactement les

---

[1] Cette partie de la thèse a fait l'objet d'une présentation lors des journées Ada de l'AFCET/ENST, 11-12 Décembre 1984 [Rosen 84b], ainsi que d'un exposé devant l'*Ada Europe Numerics Working Group.*

nombres représentés (comme cela a été tenté, sans grand succès, en PL/1). La solution a consisté à définir un ensemble minimum de nombres obligatoirement représentés, tout en laissant une certaine souplesse à l'implémentation sous réserve que certaines propriétés *minimales* soient respectées.

2. La modélisation des nombres réels en Ada

En général, les nombres que l'utilisateur souhaite utiliser ne correspondent pas aux nombres qui peuvent être manipulés par la machine. L'approche choisie en Ada utilise une notion familière aux physiciens: celle d'incertitude. Les valeurs réelles sont entachées d'une certaine incertitude. L'utilisateur est libre de choisir une incertitude absolue (représentation en *point fixe*) ou une incertitude relative (représentation en *point flottant*), et peut spécifier dans chaque cas l'incertitude maximale souhaitée.

2.1 Nombres et intervalles modèles

A chaque type réel est associé un ensemble (fini) de nombres, appelés *nombres modèles*, dont on garantit qu'ils seront représentés *exactement* dans n'importe quelle implémentation. Tout nombre réel qui n'est pas un nombre modèle appartient à un certain *intervalle modèle*, défini comme le plus petit intervalle entre nombres modèles contenant le nombre donné. L'intervalle modèle correspondant à un nombre modèle ne contient que ce nombre.

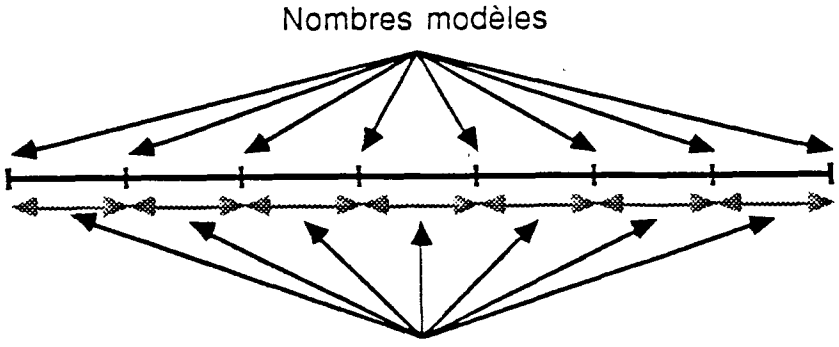


Figure 1: Nombres et intervalles modèles

Le langage spécifie que le résultat d'une opération entre deux nombres doit appartenir à l'intervalle définit comme l'ensemble des résultats de l'opération appliquée à toutes les valeurs de chacun des deux intervalles modèles d'origine. Ceci n'est rien d'autre que

l'application des règles de calcul habituel sur des valeurs entachées d'incertitude, mais a des implications moins évidentes qu'il n'y paraît à première vue. Par exemple, l'élévation d'un nombre point flottant à une puissance entière est définie comme équivalente à une suite de multiplications [LRM 4.5.6(6)]. Or, pour certaines valeurs de X, l'intervalle modèle de ((X.X).X).X est inférieur à celui de (X.X).(X.X). Certains algorithmes d'élévation à une puissance entière peuvent donc ne pas être applicables[2].

Si les règles du langage imposent de représenter exactement les nombres modèles, elles n'interdisent pas à une implémentation de représenter exactement d'autres nombres. En particulier, une implémentation peut fournir une représentation pour des nombres situés à l'extérieur de l'intervalle de définition des nombres modèles. Ces nombres supplémentaires sont appelés *nombres sûrs*, car aucun dépassement de capacité ne peut se produire tant que les calculs restent dans l'intervalle des nombres sûrs.

Ainsi, les nombres modèles définissent un ensemble minimum de nombres dont on garantit les propriétés quelle que soit l'implémentation; les nombres sûrs au contraire définissent l'ensemble de tous les nombres disponibles sur une implémentation donnée. Bien entendu, ces derniers sont toujours un sur-ensemble des nombres modèles.

Il est possible qu'une implémentation ne dispose pas de types de données permettant de satisfaire une déclaration de type de l'utilisateur; dans ce cas, le programme doit être refusé par le compilateur. Ainsi, l'acceptation par un compilateur d'un programme constitue un contrat, garantissant la précision des résultats.

2.2 Définition des nombres point flottant

Un type point flottant est défini (dans sa forme la plus générale) par une déclaration de la forme:

type <nom> **is digits** <nb. de chiffres> **range** <val. inf.> .. <val. sup.>;

Ici, <nb. de chiffres> est le nombre de chiffres significatifs (en base dix) souhaité par l'utilisateur. A partir de ce nombre, le compilateur va déterminer le nombre d'éléments binaires nécessaires à une représentation garantissant *au moins* cette précision (ce nombre est appelé B dans le manuel de référence), puis, s'il dispose de plusieurs types

---

2 Ceci ne résulte pas d'une erreur dans la définition du langage, mais d'une volonté de garantir la portabilité des résultats d'un programme Ada. "Le problème était connu, le choix délibéré" (J. Ichbiah, commentaire sur le manuel de référence 83-00312).

point flottant manipulables par le matériel, choisir la représentation adéquate [LRM 3.5.7(6)]. Ainsi, une même spécification pourra être implémentée en "simple précision" ou en "double précision" suivant les machines; seul compte le nombre d'éléments binaires nécessaires à la représentation souhaitée par l'utilisateur.

Les nombres modèles sont définis à partir de la représentation binaire. Ce sont les nombres dont la mantisse s'écrit exactement sur B bits, avec un exposant dans l'intervalle +-4.B. Ce choix de l'intervalle d'exposant requis est arbitraire, mais correspond assez bien aux matériels existant: pour une mantisse de 24 bits, cela signifie que l'exposant doit être dans l'intervalle +-96, soit au moins 7 bits (plus le signe). Pour une mantisse de 48 bits, l'exposant doit être dans l'intervalle +-192, soit au moins 8 bits (plus le signe). Noter qu'en général, une implémentation autorisera des valeurs de l'exposant extérieures à l'intervalle requis de +-4.B. Ces valeurs supplémentaires seront des *nombres sûrs*, mais non des *nombres modèles*.

2.3 Définition des nombres point fixe

Un type point fixe est défini (dans sa forme la plus générale) par une déclaration de la forme:

type <nom> **is delta** <incrément> **range** <val. inf.> .. <val. sup.>;

Comme pour un type point flottant, l'incrément donné par l'utilisateur ne représente qu'une précision *minimale* souhaitée. La précision effective (en l'absence d'une clause explicite de représentation) est choisie par le compilateur. Dans la terminologie Ada, la précision souhaitée par l'utilisateur est appelée le *delta*, alors que celle choisie effectivement est appelée le *small*. Le *small* est choisi comme la plus grande puissance de deux immédiatement inférieure au *delta* [LRM 3.5.9(5)].

Pour voir l'influence de *small* sur les résultats, considérons l'exemple suivant:

```
procedure TEST_FIX is
    type FIX is delta 0.1 range -10.0 .. 10.0;
    X : FIX := 0.4;
    Y : FIX;
begin
    Y := X + X + X;
end TEST_FIX;
```

La valeur du *delta* est 0.1, la valeur du *small* est donc 0.0625. Le nombre 0.4 n'est pas un nombre modèle, sa représentation peut donc être n'importe quelle valeur entre les nombres modèles adjacents, 0.375 et 0.4375. La valeur de Y sera donc comprise entre 1.125 et 1.3125. En particulier, il est très vraisemblable que le résultat effectivement obtenu sera une des valeurs extrêmes. Si l'on imprime le résultat, la valeur imprimée sera arrondie au nombre de chiffres après la virgule[3] correspondant à la définition de *delta*. La valeur imprimée sera donc 1.1 ou 1.3. Ceci ne surprendra pas le numéricien, puisque le résultat est largement à l'intérieur de la précision attendue de la définition, soit 1.2+-0.3. Mais pour beaucoup d'utilisateurs, la définition du type FIX semble impliquer une définition de nombres au pas de 0.1, et donc un résultat d'exactement 1.2. Si on utilise un type point fixe pour représenter des valeurs exactes plutôt que des valeurs approchées (ce qui ne correspond pas à la philosophie des types point fixe), il faut spécifier un *small* égal au *delta* au moyen d'une clause de représentation, sous peine de résultats surprenants, au moins en apparence. C'est le cas par exemple si l'on souhaite représenter sous forme de points fixes des francs et des centimes:

```
...
type Francs is delta 0.01 range 0.0 .. 1.0E10;
for Francs'small use 0.01;
...
```

Noter qu'ici aussi, le compilateur est libre d'accepter ou de refuser la clause de représentation. S'il l'accepte, cela constitue une garantie que les nombres modèles seront des multiples du *small* spécifié.

3. Implémentation de l'arithmétique point flottant

L'implémentation de l'arithmétique point flottant ne pose pas de problème spécial, car le modèle Ada des nombres point flottant correspond bien aux facilités offertes par le matériel. Dans le cas de notre implémentation sur VAX, les points flottants standard sont utilisés. La seule difficulté est due à la structure d'interprète de SETL, qui ne permet pas à un programme utilisateur de rattraper les déroutements: il est donc nécessaire de garantir qu'aucune opération ne provoquera de débordement matériel.

Pour cela, l'intervalle des nombres point flottant offerts à l'utilisateur a été réduit à la moitié de l'intervalle offert par le matériel. Après chaque addition ou soustraction, une vérification est faite qu'il n'y a pas eu de débordement de la capacité, ou sinon l'exception

---

[3] Ou plutôt le point, le langage spécifiant un format d'impression à la norme anglo-saxonne!

NUMERIC_ERROR est levée. Le problème est plus compliqué pour tester les débordements des multiplications et divisions. La solution la plus simple consiste à additionner les logarithmes des deux opérandes, et à comparer le résultat au logarithme de la plus grande valeur point flottant autorisée. On peut toutefois éviter le double appel à la fonction logarithme en prenant (formellement) le logarithme en base deux des nombres, qui n'est autre que l'exposant de la représentation machine normalisée.

4. Implémentation de l'arithmétique point fixe

Note: Pour simplifier, toutes les explications qui suivent sont données en supposant que tous les opérandes sont positifs. Dans les autres cas, le signe est traité séparément, et les opérations sont effectuées sur les valeurs absolues des opérandes.

4.1 Résumé des difficultés d'implémentation de l'arithmétique point fixe

L'idée que l'on se fait généralement des points fixes est celle d'entiers munis d'un facteur d'échelle [Froggatt 86]. Cependant, le langage autorise (3.5.10(14), 4.5.5(11)) la multiplication et la division de n'importe quel valeurs points fixes, même de types différents, dont le résultat est de type *universal_fixed*. Ce résultat doit alors être converti explicitement par l'utilisateur dans un autre type point fixe. Noter que cette conversion obligatoire attire l'attention du programmeur sur le fait que ces opérations peuvent introduire une perte de précision sur le résultat. Pour la même raison, un des opérateurs de la multiplication ou de la division ne peut être un littéral numérique. Mais la notion de valeurs *universelles* n'a pas de signification à l'exécution, il est nécessaire de déterminer un type valide pour le résultat de telles opérations, qui permettra de garantir la précision exigée par le manuel de référence. De plus, les types point fixes peuvent interférer avec les points flottants par le biais des conversions, et il est nécessaire d'étudier soigneusement les procédures correspondantes afin de satisfaire aux exigences du langage. En particulier, il est nécessaire de définir une arithmétique étendue pour traiter correctement ces problèmes; une des difficultés consiste à trouver des algorithmes autorisant une arithmétique de taille limitée, et la plus petite possible.

Une autre difficulté vient de la définition des attributs: certains d'entre eux sont liés aux *sous_types*, ce qui signifie qu'ils peuvent être dynamiques. Par exemple, bien que la représentation d'un objet ne dépende que du type de base, et soit par conséquent statique, les nombres modèles sont définis à partir du sous-type, et peuvent donc être dynamiques.

4.2 Représentation des points fixes

Les nombres point fixe sont définis comme étant de la forme *signe.mantisse.small* [LRM 3.5.9(4)]. Mais le *small* est une caractéristique du type point fixe, il n'est pas nécessaire de le conserver dans chaque valeur; en fait il suffit de le garder dans le patron de type. De fait ne reste donc à conserver dans une variable d'un type point fixe que *signe.mantisse*, c'est à dire un simple entier, que nous appellerons la *représentation* de la valeur point fixe.

REPRESENTATION DE SMALL

Le langage autorise une clause de représentation pour *small*, chaque implémentation étant libre de définir les valeurs autorisées pour une telle clause. Il était spécifié initialement que le nouvel interprète d'Ada/ED devait admettre des clauses de représentation pour *small* qui soient des puissances de deux ou des puissances de dix. Les puissances de deux sont obligatoires (la valeur implicite pour *small* si aucune clause de représentation n'est spécifiée est toujours une puissance de deux [LRM 3.5.9(5)]), et il est vraisemblable que les puissances de dix seront très souvent utilisées, spécialement par les programmes désirant utiliser les points fixes pour représenter des "francs et centimes" par exemple.

Le premier modèle de points fixes utilisait deux positions dans le patron, une pour stocker la base (2 ou 10), et l'autre pour la puissance de la base correspondant au *small*. Toutefois, manipuler deux espèces différentes de types point fixe posait de nombreux problèmes en cas de conversion, et surtout pour la multiplication et la division de deux valeurs dont l'une avait un *small* en 2^n et l'autre un *small* en 10^n: dans ce cas, le résultat était d'un type dont le *small* n'était ni une puissance de 2, ni une puissance de 10; il en résultait la nécessité d'une conversion qui pouvait faire perdre de la précision. En quelque sorte, la multiplication et la division étaient des opérations externes sur l'ensemble des nombres point fixe ainsi définis. La solution a consisté à clore l'ensemble des nombres point fixe, en autorisant pour *small* n'importe quel nombre de la forme $2^p.5^q$, incluant donc les puissances de 2 et de 10 comme cas particulier. Cette solution présente les avantages suivants:

- Plus grande simplicité et meilleure efficacité dans les manipulations d'opérations points fixes;
- Unification des types points fixes;

- Plus grande gamme de valeurs autorisées pour les clauses de représentation de *small*.
- Facilité de formatage pour l'impression (cf. chapitre IV).

Remarquons au passage qu'une implémentation désirant autoriser diverses valeurs de *small* doit pouvoir conserver ce *small* à l'exécution. Or *small* est toujours réel, ou plus exactement rationnel, et de précision arbitraire. Si l'implémentation souhaite autoriser n'importe quelle valeur de *small*, il lui faut le conserver sous forme de rationnel à l'exécution, ce qui semble très coûteux. Le conserver sous forme de point flottant revient à limiter les valeurs de *small* aux nombres modèles du type point flottant. Remarquer que notre implémentation permet de revenir à la forme rationnelle si besoin est, tout en ne nécessitant que deux positions pour le stockage de *small* (les valeurs de p et q), comme dans la solution précédente.

REPRESENTATION DE LA MANTISSE

La mantisse n'est en fait qu'un simple entier. Mais dans le but de préparer la future version C des entiers longs, nous avons décidé d'implémenter les valeurs points fixes sur une ou deux positions de tuple, c'est à dire comme des entiers standard ou longs, suivant l'intervalle nécessaire. Le module d'arithmétique point fixe contient donc également un module d'arithmétique étendue.

A part les entiers standard et longs, l'arithmétique point fixe utilise un type intermédiaire qui est un quadruple entier, appelé *double_xleng*. Ce type n'est utilisé que de façon interne par la multiplication et la division.

Les valeurs point fixes sont toujours rangées dans un tuple. Les valeurs entières standard sont rangées dans le premier élément du tuple, et les entiers longs dans les deux premiers éléments du tuple, de façon à modéliser un paquetage de multi-précision dans lequel un nombre est représenté comme un tableau de tranches. La représentation est en découpage binaire vrai: s'il y a B bits dans un mot machine, la première tranche est un nombre signé entre -2^{B-1} et $2^{B-1}-1$, et les tranches suivantes sont des nombres non signés entre 0 et 2^{B-1}.

Les opérations entre valeurs étendues ont été implémentées de façon très simples, puisque SETL fournit déjà une arithmétique étendue: Les tranches sont regroupées pour

former un entier SETL, l'opération est effectuée entre entiers SETL, puis les tranches sont reconstituées à partir du résultat.

4.3 Opérations sur les nombres points fixes

Les opérations concernant les nombres points fixes comprennent l'addition, la soustraction, la multiplication, et la division.

4.3.1 Opérateurs additifs

L'addition et la soustraction ne sont autorisées qu'entre opérandes du même type point fixe. Elles sont traitées comme des opérations entières entre les mantisses, et ne posent pas de problème particulier.

4.3.2 Opérateurs multiplicatifs à résultat point fixe ou entier

La multiplication et la division sont autorisées entre n'importe quels types points fixes. Elles fournissent un résultat du type *universal_fixed*. La multiplication entre une valeur point fixe et une valeur du type INTEGER, ainsi que la division d'une valeur point fixe par une valeur du type INTEGER sont autorisées. Elles fournissent un résultat du même type que l'opérande point fixe, et ne nécessitent donc pas de conversion. On remarquera que ces dernières opérations ne sont possibles qu'avec le type prédéfini INTEGER, et non pas avec n'importe quel type entier. Dans le but d'unifier l'interface avec les procédures d'arithmétique point fixe, le système comporte un type appelé INTEGER_FIXED, défini ainsi:

```
Min  : constant FLOAT  :=     -- valeur point flottant de INTEGER'FIRST[4]
Max  : constant FLOAT  :=     -- valeur point flottant de INTEGER'LAST
type INTEGER_FIXED is delta 1.0 range Min .. Max;
```

INTEGER_FIXED est isomorphe à INTEGER (les valeurs des deux types ont la même représentation), aussi les opérations entre points fixes et INTEGER sont générées comme des opérations avec le type INTEGER_FIXED.

---

[4] Les bornes d'un type fixe devant être statiques, et une conversion de type ne l'étant pas, il n'est pas autorisé d'écrire cette définition dans le langage sous la forme FLOAT(INTEGER'FIRST).

La multiplication de deux valeurs points fixes fournit un résultat dont le *small* est égal au produit des *small* des deux opérandes. Pour effectuer la multiplication, on construit d'abord un descripteur temporaire pour le résultat. Celui-ci est de type *double_xleng* (puisque chaque opérande peut être de type *xleng*), avec des exposants de puissance de 2 et de 5 qui sont la somme de ceux des opérandes; les mantisses sont alors multipliées, et le résultat converti dans le type du résultat en utilisant les procédures normales de conversions entre points fixes avec le descripteur temporaire.

La division est sensiblement plus compliquée, car le quotient des mantisses n'est généralement pas entier, et il nous faut la valeur exacte du quotient pour garantir la précision du résultat. Nous allons maintenant démontrer qu'il existe un type point fixe dans lequel on peut convertir le dividende *avant* d'effectuer la division qui fournira une précision suffisante sur le résultat. Ce type est celui dont le *small* est égal au produit du *small* du diviseur par celui du résultat (toujours connu, d'après les règles du langage [LRM 4.5.5(11)]).

Posons S_1 le *small* de l'opérande 1, S_2 le *small* de l'opérande 2, et S_r le *small* du résultat. De la même façon, appelons R_1, R_2, et R_r les représentations de l'opérande 1, de l'opérande 2, et du résultat, respectivement. R_r est par définition tel que:

$$R_r.S_r <= \frac{R_1.S_1}{R_2.S_2} < (R_r+1).S_r$$

Comme le *small* du type intermédiaire, S_i, est égal à $S_2.S_r$, la conversion de R_1 dans le type intermédiaire fournit une représentation R_i telle que:

$$R_i.S_2.S_r = R_1.S_1 \qquad (1)$$

Comme R_i a été trouvé au moyen d'une division entière, par définition:

$$R_i <= \frac{R_1.S_1}{S_2.S_r} < R_i+1$$

En divisant par R_2, nous obtenons:

$$\frac{R_i}{R_2} <= \frac{R_1.S_1}{R_2.S_2.S_r} < \frac{R_i+1}{R_2} \qquad (2)$$

appelons q le quotient de R_i par R_2:

$$R_i = q.R_2+r$$

L'équation (2) devient:

$$q + \frac{r}{R_2} <= \frac{R_1.S_1}{R_2.S_2.S_r} < q + \frac{r+1}{R_2}$$

Comme:

$$q <= q + \frac{r}{R_2}$$

$$q + \frac{r+1}{R_2} <= q + 1$$

en multipliant par S_r, il vient:

$$q.S_r <= \frac{R_1.S_1}{R_2.S_2} < (q+1).S_r$$

Par conséquent, q, obtenu par une division entière de la représentation intermédiaire par la représentation du diviseur, est une valeur acceptable pour R_r[5].

La division opère entre une valeur de type *double_xleng* et une valeur de type *xleng*, fournissant une valeur de type *xleng*; par conséquent, un descripteur est d'abord construit avec des puissances de 2 et de 5 qui sont la somme de celles du deuxième opérande et du résultat; ensuite, le premier opérande est converti dans ce type, puis divisé par le second opérande. Ceci garantit que le résultat a automatiquement le type et la précision désirés. Une vérification est ensuite faite qu'il n'y a pas débordement. L'exception NUMERIC_ERROR est levée sinon.

---

[5]Mes remerciements à G. Fisher pour son aide dans cette démonstration.

4.3.3 Opérateurs multiplicatifs à résultat point flottant

[LRM 4.5.5(11)] précise que le résultat d'un produit ou d'un quotient d'expressions point fixe doit etre converti dans un type numérique, mais aucune restriction ne s'applique à ce type; il peut donc parfaitement s'agir d'un type point flottant, auquel cas l'algorithme précédant ne peut s'appliquer.

Dans le cas d'une multiplication ou d'une division à résultat point flottant, nous reconstruisons les valeurs rationnelles des opérandes, nous effectuons le produit des numérateurs et des dénominateurs (en les croisant pour la division), puis nous effectuons la division *flottante* du numérateur par le dénominateur du résultat. Ceci ne pose aucun problème de précision.

4.4 Conversions mettant en jeu des points fixes

On peut distinguer trois types de conversions: entre valeurs points fixes, à partir ou à destination d'entiers, et à partir ou à destination de points flottants.

4.4.1 Conversions entre points fixes

Le seul paramètre important en matière de conversion entre types points fixes est le *small* des types. Nous considérerons donc que nous effectuons une conversion entre un type *source* dont le *small* est S_s et un type cible dont le *small* est est S_c. Nous rappelons que la façon dont le *small* est gardé dans le descripteur de type (les puissances de 2 et de 5) n'est qu'une façon simplifiée de les garder sous forme rationnelle.Les nombres modèles d'un type point fixe étant les multiples entiers du *small*, convertir une valeur V depuis un type source vers un type cible consiste fondamentalement à trouver une certaine représentation R_c ([LRM 4.5.7]) telle que, pour une représentation dans le type source R_s:

$$R_c . S_c <= R_s . S_s <= (R_c+1) . S_c$$

Chaque *small* est entièrement caractérisé par p et q, ses exposants de puissances de 2 et de 5 . Pour simplifier l'énoncé mathématique de la démonstration, nous appellerons p+ la valeur de p si celui-ci est positif, 0 sinon; et de même pour q. Le facteur de conversion est un nombre rationnel, défini comme:

$$\frac{N}{D} = \frac{S_s}{S_c} = \frac{2^{ps+} . 2^{pc-} . 5^{qs+} . 5^{qc-}}{2^{ps-} . 2^{pc+} . 5^{qs-} . 5^{qc+}}$$

La conversion est obtenue en appliquant:

$$R_c = R_s \cdot N / D$$

Aucune erreur d'arrondi ne peut se produire du moment que la multiplication est effectuée avant la division.

Un des buts du projet, compte tenu de la prochaine version du système écrite en C, était d'éviter une arithmétique de taille arbitraire, étant entendu qu'une arithmétique étendue était requise de toute façon. Si l'implémentation de l'arithmétique point fixe ne pose guère de problème tant que les *small* sont des puissances de 2, le fait d'introduire des *small* quelconques provoque l'apparition de facteurs de conversions non réductibles simplement, qui semblent à priori nécessiter une arithmétique de taille arbitraire. C'est pour cette raison qu'aucun compilateur Ada validé actuellement (mis à part Ada/Ed bien évidemment) ne supporte de clauses de représentation autres qu'en puissances de 2. Nous n'avons pu trouver aucune bibliographie sur le sujet à l'époque où nous avons étudié ce problème.

Depuis, [Froggatt 86] a démontré qu'une arithmétique triple longueur était suffisante pour implémenter la multiplication, la division, et la conversion entre points fixes avec des *small* quelconques. Il n'a pu cependant étendre sa méthode (par utilisation de fractions continues) aux conversions mettant en jeu des nombres points flottant, et la question de savoir la taille de l'arithmétique à mettre en jeu pour implémenter les points fixes dans le cas le plus général reste ouverte.

Le contexte dans lequel nous nous situions était cependant plus favorable, dans la mesure où nous n'avions pas à autoriser des *small* quelconques. Nous avons donc tenté, compte tenu de cette limitation, à nous restreindre à une arithmétique double précision. Celle-ci est en général la plus facile à implémenter sur des machines réelles, le passage à la triple précision nécessitant par contre en général le codage des opérations sous forme de boucles, et n'étant plus alors très éloigné d'une arithmétique arbitraire, aussi bien du point de vue de la complexité que des performances. Il nous fallait donc définir des limitations aux valeurs des *small* autorisées (ou plus précisément aux valeurs de p et q) permettant de respecter cette contrainte de double précision au maximum dans tous les calculs intermédiaires. La taille maximum d'une valeur point fixe étant de 64 bits, il nous fallait une arithmétique autorisant des valeurs temporaires jusqu'à 128 bits, dans le cas du produit de deux valeurs de 64 bits. Pour ne pas accroitre cette limite, il nous fallait limiter les valeurs de N et D à 64 bits, donc limiter les valeurs autorisées du numérateur et du

dénominateur de la représentation rationnelle de *small* à 32 bits. Ceci nécessite la limitation de p à l'intervalle +-31, et de q à l'intervalle +-9, avec la contrainte supplémentaire que si p et q sont de même signe, alors $2^p.5^q$ doit lui-même tenir sur 32 bits. Par rapport aux spécifications initiales, qui étaient de supporter des *small* en puissance de 2 ou de 10, ceci autorise des *small* jusqu'à 2^{+-31} et 10^{+-9}, ce qui paraît raisonnable compte tenu des buts du système Ada/Ed.

4.4.2 Conversions mettant en jeu des entiers

Pour les conversions mettant en jeu des entiers, il existe un type dans l'interprète appelé INTEGER_FIXED, correspondant à un type point fixe ayant le même intervalle de valeurs que INTEGER, et un *delta* de 1.0. Les représentations des valeurs de INTEGER_FIXED sont les mêmes que celles des valeurs INTEGER correspondantes. Par conséquent, les conversions mettant en jeu des valeurs entières sont générées comme des conversions normales entre points fixes, en utilisant le descripteur d'INTEGER_FIXED pour les valeurs entières.

4.4.3 Conversions mettant en jeu des points flottants

Pour la conversion à destination d'un type point flottant, nous utilisons le même principe que pour la division à résultat flottant, puisque cette opération peut être vue comme une division de la valeur point fixe par 1.0, avec un résultat du type point flottant considéré. Autrement dit, nous calculons la représentation rationnelle du nombre point fixe, puis nous effectuons la division *flottante* du numérateur par le dénominateur.

La conversion de point flottant en point fixe demande certaines précautions pour garantir la précision nécessaire, spécialement lorsque la valeur point fixe est égale à, ou proche d'un nombre modèle du type point flottant. Tout d'abord, remarquons qu'une valeur point flottant est définie ([LRM 3.5.7]) par *signe.mantisse*.$2^{exposant}$. Ceci peut être interprété comme une valeur point fixe dont le *small* serait $2^{exposant-21}$, puisque la *mantisse* représente un nombre compris entre 0.5 et 1.0, avec une représentation sur 21 bits (sur Vax). Il n'est cependant pas possible d'appliquer le même algorithme que pour la conversion entre points fixes, car l'exposant peut varier dans un intervalle de +-84, plus large donc que l'intervalle de +-31 dont nous avons besoin pour garantir l'absence de débordement lors des calculs intermédiaires. Soit $M.2^e$ la valeur point flottant à convertir. Soit:

$$M' = M.2^{21}$$

En appelant p et q les exposants de 2 et de 5 du type (point fixe) du résultat, la conversion de la valeur consiste simplement à trouver un nombre R tel que:

$$R = M' . 2^{e-21} . \frac{2^{p-} . 5^{q-}}{2^{p+} . 5^{q+}}$$

Posons:

$$x = e-21-p$$

Il vient:

$$R = M' \frac{5^{q-}}{5^{q+}} . 2^{x}$$

Etant donné que e appartient à l'intervalle +-84 et p à l'intervalle +-31, x appartient à l'intervalle -136,+94. Si q est positif et x négatif, le numérateur ne nécessite que les 21 bits de M'. Si q est positif et x positif, le numérateur tient sur 21+94 = 115 bits. Si q est négatif et x négatif, le numérateur tient sur 21+31 = 52 bits. Le numérateur, pour lequel 128 bits sont autorisés, ne peut donc déborder que si q est négatif et x positif (21 + 31 + 94 = 146 bits). Mais dans ce cas, la conversion ne met en jeu aucune division. Il est donc possible de lever NUMERIC_ERROR en toute sécurité si le résultat ne tient pas sur 128 bits. Si x est positif, on ne perd aucune précision en calculant d'abord $M'x5^{q-}/5^{q+}$, et en divisant ensuite par 2^x. Ce dernier peut ne pas tenir sur 64 bits, mais la division peut être effectuée par des décalages répétés, et il n'est donc pas nécessaire de calculer effectivement 2^x.

Il est important de noter que notre algorithme fonctionne grâce au fait que les nombres point flottant n'ont que 21 bits de mantisse; si nous devions supporter des nombres en flottant double précision, nous ne pourrions plus nous contenter de notre arithmétique double précision... pour l'implémentation des points fixes! Autrement dit, il n'est pas possible de proposer de méthode d'implémentation des points fixes qui omettraient leurs relations avec les points flottants, car c'est bien là que peuvent se trouver les plus grandes exigences en matière de précision.

4.5 Attributs point fixe

Les différents attributs point fixe sont définis par [LRM 3.5.10]. On peut les répartir en plusieurs classes, selon le moment où il est possible de les évaluer.

La première classe comprend les attributs qui sont statiques et indépendants de l'implémentation. Ils sont évalués par le frontal, et n'apparaissent pas au générateur de code. Ces attributs sont DELTA, AFT, SMALL, et SAFE_SMALL.

La seconde classe comprend les attributs statiques qui dépendent de l'implémentation. Ils sont donc évalués par le générateur de code. Ces attributs sont MACHINE_ROUNDS et MACHINE_OVERFLOWS.

La troisième classe comprend les attributs qui sont (potentiellement) non statiques, c'est à dire les attributs qui dépendent de l'intervalle de la contrainte de point fixe (le *delta* étant lui toujours statique). Ils sont évalués par le générateur de code s'ils sont statiques, ou à l'exécution sinon. Ces attributs sont FORE, FIRST, LAST, MANTISSA, et LARGE. L'attribut SAFE_LARGE est transformé par le frontal en BASE'LARGE, et est donc traité comme l'attribut LARGE.

Les valeurs de FIRST et LAST (ou plutôt leurs représentations) sont contenues, comme pour tout type numérique ou énumération, dans le patron de type.

LARGE est supposé appartenir au type *universal_real*. Mais comme c'est (par définition) un nombre modèle, il est calculé comme une valeur appartenant au type de base. En cas de besoin, une conversion est générée par le frontal. MANTISSA et FORE sont supposés appartenir au type *universal_integer*. Ils sont calculés comme des INTEGER.

Un problème particulier se pose pour les attributs MANTISSA et LARGE d'un sous-type dynamique: ils sont définis à partir du *small* du sous-type point fixe, qui peut être plus grand que le *small* du type de base. Cependant, la représentation réelle des valeurs du sous-type utilise le *small* du type de base, pour éviter des conversions entre valeurs appartenant à des sous-types différents du même type de base. Pour cette raison, le rapport entre le *small* du sous-type et celui du type de base est empilé avant d'exécuter l'instruction *attribute* (ce rapport est toujours entier, cf. [LRM 3.5.9(16)]).

Chapitre III.
Entrées-sorties

En Ada, toutes les entrées-sorties sont définies de façon *externe* au langage, au moyen de deux paquetages génériques (SEQUENTIAL_IO: entrées-sorties séquentielles, et DIRECT_IO: entrées-sorties directes), et d'un paquetage normal, mais contenant des sous-paquetages génériques (TEXT_IO: entrées-sorties texte). Un paquetage annexe, IO_EXCEPTIONS, contient uniquement les définitions des exceptions liées aux entrées-sorties de façon à ce qu'elles soient communes à tous les paquetages. La définition précise de ces paquetages forme le chapitre 14 de [DoD 83].

L'exigence de séparer les entrées-sorties du langage était d'ailleurs une exigence du rapport Steelman [DoD 78]. On peut même considérer que l'une des preuves de la souplesse d'Ada est justement d'être un langage dans lequel même les entrées-sorties peuvent être définies au moyen des structures standard du langage.

Les entrées-sorties d'Ada sont très conventionnelles, au sens que l'on retrouve la notion de fichier externe, lié à une représentation interne par une opération d'ouverture, et des opérations de lecture et d'écriture classiques. Les entrées-sorties séquentielles offrent quasiment les mêmes possibilités que les fichiers séquentiels de Pascal. Les entrées-sorties directes offrent un mode d'accès équivalent aux entrées-sorties séquentielles, mais permettent de plus un accès en ordre aléatoire en précisant la position ordinale dans le fichier de l'élément à lire. Contrairement aux fichiers séquentiels qui ne peuvent être que ralongés, les fichier directs peuvent être directement modifiés.

Les fichiers texte permettent de sortir des résultats sous forme imprimable. Ils se comportent comme des fichiers séquentiels de caractères, mais offrent une structuration supplémentaire: les caractères sont regroupés en lignes, séparées par une *marque de fin*

de ligne; les lignes sont regroupées en pages, séparées par une *marque de fin de page*. Enfin, le fichier est terminé par une *marque de fin de fichier*. Cette notion de marque n'est en fait utile que pour la définition du fonctionnement des entrées-sorties, car elles ne sont en aucun cas transmises au programme utilisateur.

1. Principes généraux

Les principes généraux de notre implémentation du mécansime d'entrées-sorties que nous allons énoncer ici s'appliquent à tous les types de fichiers, qu'ils soient séquentiels, directs, ou texte.

La plus grande partie du système d'entrées-sorties de l'interprète de bas niveau est dérivé directement de celui de haut niveau. En particulier, la gestion des fichiers, qui fait appel à des fonctions SETL de haut niveau, n'a pas été changée, contrairement à la politique du reste de l'interprète qui était de rabaisser le niveau sémantique du système. Ceci est dû au fait que les modalités d'entrées-sorties sont tellement différentes d'un langage à l'autre qu'il n'est pas apparu rentable d'investir du temps à redéfinir des structures qui, de toutes façons, devraient être redéfinies lors de la traduction dans un langage de plus bas niveau. D'autre part, les fonctionnalités concernées s'exécutant hors du contexte de la machine virtuelle (c'est à dire que la fonctionnalité est directement réalisée en SETL et non par interprétation d'une description de plus haut niveau), il n'y avait rien à gagner du point de vue de l'efficacité. Nous avons donc adopté la politique nous permettant de réaliser le plus rapidement possible un système fonctionnant correctement.

A part quelques différences dans les normes d'appel des procédures, l'essentiel de notre travail a porté sur l'amélioration du système de lecture anticipée de façon à permettre un plus grand confort d'utilisation de TEXT_IO pour des échanges avec un terminal interactif[1], et sur le formattage des points fixes compte tenu de la nouvelle représentation de ceux ci.

---

[1]Cette partie de la thèse a donné lieu à une communication lors de l'*IEEE conference on Ada applications and environments* [Rosen 84a].

1.1 Appels au système d'entrées-sorties

Les appels au système d'entrées-sorties, ainsi d'ailleurs qu'aux fonctionnalités du paquetage CALENDAR et au paquetage mathématique réalisé par J. Chiabaut [Chiabaut 85],[Kok 84], sont réalisés au moyen d'une instruction machine spéciale: *call_predef*. Cette instruction admet un argument qui est un discriminant servant à déterminer la fonction effectivement appelée.

Les paramètres de la fonction se trouvent sur la pile d'exécution, exactement comme pour un appel de procédure normal. En fait, les paquetages d'entrée-sortie, CALENDAR, et le paquetage mathématique sont définis comme des paquetages standard Ada, dont les corps de procédure contiennent un pragma spécial générant l'instruction *call_predef*. L'interface utilisateur est ainsi parfaitement standard.

1.2 Structures générales

La représentation interne d'une valeur fichier est simplement un entier. Les différentes variables dénotant l'état des fichiers sont implémentées sous forme de variables internes à l'interprète qui sont des correspondances entre les numéros de fichiers et les valeurs correspondantes, dont le détail est donné ci-dessous. Cette méthode est bien adaptée à la philosophie de SETL; dans une implémentation de plus bas niveau, la structure serait inversée, c'est à dire qu'une variable d'un type fichier serait un enregistrement qui contiendrait toute l'information relative à l'état du fichier. Une telle variable est généralement appelée FCB (pour *File Control Block*, bloc de contrôle de fichier) dans les systèmes d'exploitation.

Les tables communes à tous les types de fichiers sont les suivantes:

OPENFILES: un ensemble contenant les valeurs des entiers associés avec des fichiers actuellement ouverts.

IOTEMPS: un ensemble contenant les noms des fichiers temporaires. Ceux-ci sont en effet implémentés sous forme de fichiers permanents dont les noms sont générés automatiquement à partir de leur heure de création, et qui sont automatiquement effacés à la terminaison du programme.

IOFNAMES: une correspondance entre les numéros de fichiers et les noms de fichiers externes associés.

IOFORMS: une correspondance entre les numéros de fichiers et les paramètres FORM associés.

IOMODES: une correspondance entre les numéros de fichiers et et leurs modes d'ouverture (séquentiel, direct, ou texte en mode IN_FILE, OUT_FILE, ou INOUT_FILE).

A chaque genre de fichier (séquentiel, direct, texte) est associé de plus un ensemble (au sens de SETL) des numéros de fichiers ouverts selon la méthode correspondante. Ceci permet de tester aisément la nature du fichier. Par exemple, pour déterminer si un fichier F est un fichier de texte, on écrira:

if F **in** TIO_FILES **then** ...

Dans une implémentation de plus bas niveau, un discriminant du FCB serait utilisé pour déterminer la méthode d'accès.

2. Entrées-sorties séquentielles

Les entrées-sorties séquentielles utilisent directement le système d'entrées-sorties binaires séquentielles de SETL. Un tampon mémoire contient l'élément courant du fichier, ou une marque spéciale si la fin de fichier a été atteinte. La gestion des entrées-sorties séquentielles nécessite les structures suivantes:

SIO_BUFFERS: Une correspondance entre les numéros de fichiers et les tampons correspondant. Les tampons sont des tuples d'octets contenant l'image mémoire de l'objet.

SIO_FILES: Un ensemble des numéros de fichiers associés à des fichiers séquentiels.

3. Entrées-sorties directes

Les fichiers directs sont modélisés par des tuples en mémoire, chaque élément du tuple correspondant à un enregistrement. A l'ouverture du fichier, le fichier externe est entièrement lu et recopié en mémoire. Il est de nouveau recopié sur le fichier externe lors de la fermeture du fichier. La structure de tuple est bien adaptée à modéliser un fichier direct, puisqu'elle est adressée par un indice entier, qu'elle est virtuellement infinie, et qu'elle peut contenir des éléments indéfinis. D'autre part, SETL ne possèdant pas une notion de fichier direct qui aurait pu servir de support immédiat aux fichiers directs d'Ada, une implémentation avec des fichiers séquentiels aurait été extrèmement inefficace.

La gestion des entrées-sorties directes nécessite les structures suivantes:

DFILES: une correspondance entre les numéros de fichiers et les tuples qui les représentent en mémoire.
DPOSS: une correspondance entre les numéros de fichiers et la position de l'élément courant dans le fichier.
DSIZES: une correspondance entre les numéros de fichiers et le nombre courant d'éléments dans le fichier.
DIO_FILES: Un ensemble des numéros de fichiers associés à des fichiers directs.

4. Entrées-sorties mode texte

4.1 Généralités

Les entrées-sorties de texte utilisent le système normal d'entrées-sorties texte de SETL. La gestion des entrées-sorties séquentielles nécessite les structures suivantes:

PAGE_NUMBER: une correspondance entre les numéros de fichiers et le numéro de page courante.
LINE_NUMBER: une correspondance entre les numéros de fichiers et le numéro de ligne courante.
COLUMN_NUMBER: une correspondance entre les numéros de fichiers et la colonne courante.
TEXT_BUFFERS: une correspondance entre les numéros de fichiers et le tampon de ligne courant (cf. ci-dessous).
NEXT_LINES: une correspondance entre les numéros de fichiers et le pré-tampon 1 correspondant (cf. ci-dessous).
NEXT_LINE2S: une correspondance entre les numéros de fichiers et le pré-tampon 2 correspondant (cf. ci-dessous).
TIO_FILES: Un ensemble des numéros de fichiers associés à des fichiers textes.
STANDARD_IN_FILE: le numéro de fichier associé au fichier STANDARD_INPUT [LRM 14.3.2].
STANDARD_OUT_FILE: le numéro de fichier associé au fichier STANDARD_OUTPUT [LRM 14.3.2].
CURRENT_IN_FILE: le numéro de fichier associé au fichier CURRENT_INPUT [LRM 14.3.2].

STANDARD_IN_FILE: le numéro de fichier associé au fichier CURRENT_OUTPUT [LRM 14.3.2].

4.2 Marques de fin de ligne, de fin de page et de fin de fichier

Les fichiers de texte vus par SETL se composent naturellement d'une suite de lignes. Les lignes sont lues entièrement lorsque nécessaire (pas de lecture caractère par caractère), et stockées dans un tampon (cf. ci-dessous). Il n'y a donc pas de caractère de fin de ligne, la fin de ligne est considérée comme atteinte lorsque le dernier caractère du tampon a été lu.

SETL ne dispose pas de structure en page. Nous avons donc décidé de représenter la fin de page par une ligne ne comportant que le caractère ASCII Form-Feed (FF)[2].

La fin de fichier est signalée par SETL en renvoyant OM à la place de la ligne dont la lecture avait été demandée; dans ce cas, une marque spéciale (un atome) est mise dans le tampon.

4.3 Formattage et décodage des données

4.3.1 Position du problème

Les fichiers de texte sont composés de caractères. Le formattage est l'opération consistant à représenter une valeur d'un type numérique ou énumération sous forme d'une suite de caractères. Le décodage est l'opération inverse, consistant à passer de la représentation externe imprimable d'une valeur à sa représentation interne.

Le formattage et le décodage ne sont pas nécessairement lié à une entrée-sortie, car Ada fournit des procédures permettant de formatter ou de décoder des valeurs à destination ou à partir d'une chaîne de caractères; cette distinction n'est cependant pas importante du point de vue de l'implémentation, car les mêmes services sont bien entendu utilisés dans les deux cas.

---

[2]Ceci est autorisé, car le manuel de référence précise que "l'effet de la lecture ou de l'écriture de caractères de contrôle (autres que la tabulation horizontale) n'est pas défini par le langage". [LRM14.3(7)]

Les caractères et chaînes de caractères peuvent bien évidemment être transmis tels quels dans les fichiers texte, et ne nécessitent aucune opération particulière pour le formattage ou le décodage.

L'impression d'un nombre entier est obtenue aux moyens des procédures (surchargées) PUT, définies dans le paquetage INTEGER_IO. La seule particularité est qu'il est possible d'imprimer le nombre dans n'importe quelle base comprise entre 2 et 16 au moyen d'un paramètre facultatif (BASE). Un paramètre, également facultatif, WIDTH, donne le nombre de positions à utiliser pour imprimer le nombre. En cas de besoin, la valeur est ajustée à droite [LRM 14.3.7].

L'impression d'un nombre réel est obtenue au moyen des procédures (surchargées) PUT, définies dans le paquetage FIXED_IO pour les valeurs point fixe et dans le paquetage FLOAT_IO pour les valeurs point flottant. Ces deux procédures se comportent de la même façon [LRM 14.3.8].

Un nombre réel peut s'écrire sous l'une des deux formes suivantes:

FORE . AFT

ou

FORE . AFT E EXP

FORE étant la partie avant le point décimal, AFT la partie après le point décimal, et EXP l'exposant facultatif. Les procédures PUT admettent des paramètres (avec valeurs par défaut) permettant de spécifier les valeurs des longueurs des champs FORE, AFT, et EXP. Si FORE est supérieur à la longueur nécessaire, le nombre est complété à gauche par des espaces; si AFT est supérieur à la longueur nécessaire (compte tenu du *small* du type), le nombre est complété à droite par des zéros non significatifs; une valeur nulle de EXP a la signification conventionnelle que le champ EXP est omis, autrement si EXP est supérieur à la longueur nécessaire, l'exposant est complété à gauche avec des zéros non significatifs.

Si le champ AFT est inférieur à ce qui est nécessaire pour imprimer tous les chiffres significatifs, la valeur est arrondie à AFT chiffre significatifs après la virgule.

Le problème de l'impression des nombres réels est donc de trouver un algorithme d'encodage permettant de garantir la précision souhaitée, sans nécessiter dans le cas des points fixes une arithmétique de précision arbitraire.

L'impression des valeurs de type énumération est obtenue aux moyens des procédures (surchargées) PUT, définies dans le paquetage ENUMERATION_IO. Les valeurs sont représentées par la suite de caractères correspondant à leur définition dans la déclaration de leur type. Un paramètre permet de les imprimer au choix en majuscules ou en minuscules [LRM 14.3.9].

Le décodage des valeurs numériques suit la syntaxe des nombres Ada [LRM 14.3.5(6)]. Le décodage des valeurs énumération suit la syntaxe des litéraux énumération [LRM 14.3.5(5)].

4.3.2 Formattage des valeurs entières

Le formattage des valeurs entières ne pose pas de problème spécial. Dans le cas d'un formattage en base 10, SETL nous fournit une fonction prédéfinie qui l'effectue; pour les autres bases, nous utilisons l'algorithme classique par divisions successives par la valeur de la base.

4.3.3 Formattage des valeurs point flottant

SETL fournit une fonction prédéfinie pour le formattage des valeurs point flottant, dont la précision est suffisante compte tenu des limitations sur DIGITS de notre implémentation. Nous utilisons donc cette fonction pour effectuer le formattage, suivie de quelques manipulations de chaînes de caractères pour respecter les normes de présentation Ada.

4.3.4 Formattage des valeurs point fixe

La première idée qui vient à l'esprit pour formatter des valeurs point fixes serait de les convertir en point flottant, puis d'utiliser les routines d'impression de point flottant, puisque la représentation externe des points fixes est la même que celle des points flottants. Malheureusement, cette mèthode est inapplicable, car la précision serait insuffisante.

Une conséquence du choix que nous avons fait de supporter des clauses de représentation pour *small* qui sont le produit d'une puissance de 2 par une puissance de 5 est que le nombre de chiffres significatifs nécessaires pour représenter *exactement* n'importe quelle valeur d'un type point fixe est fini, et que le nombre de chiffres significatifs après le point décimal peut être déterminé *a priori*. En effet:

Appelons V la valeur à imprimer, R sa représentation, p et q les puissances de 2 et de 5 du type auquel appartient la valeur (cf. Chapitre 1 pour la définition de ces termes).

$$V = R.2^p.5^q$$

Si p et q sont positifs, V est entier (pas de chiffres après la virgule). Si seul p est négatif, alors:

$$\begin{aligned} V &= R.2^p.5^q \\ &= R.5^{-p}.10^p.5^q \\ &= R.5^{q-p}.10^p \end{aligned}$$

Comme $R.5^{q-p}$ est entier, ceci démontre que V possède exactement -p chiffres après la virgule. On démontre de même que si seul q est négatif, V s'écrit avec exactement -q chiffres après la virgule.

Si p et q sont tous deux négatifs: supposons que p > q. Le raisonnement serait le même dans le cas où p <= q.

$$\begin{aligned} V &= R.2^p.5^q \\ &= R.2^p.2^{-q}.10^q \\ &= R.2^{p-q}.10^q \end{aligned}$$

Comme p est supérieur à q par hypothèse, $R.2^{p-q}$ est entier, et le nombre peut s'écrire avec q chiffres après la virgule. Nous avons donc démontré que tout nombre de la forme $R.2^p.5^q$ s'écrit avec exactement Max(|p|, |q|) chiffres après la virgule.

Soit r=Max(|p|, |q|), ou 0 si p et q sont positifs. Pour formatter correctement un nombre point fixe, il suffit de multiplier R par $10^r.2^p.5^q$ (le résultat est forcément entier), traduire ensuite le résultat au moyen de la procédure normale de formattage de nombre entier, puis rajouter un point décimal avant les r derniers chiffres. Si la valeur spécifiée par l'utilisateur pour AFT est inférieure au minimum requis par le type point fixe et qu'un arrondi est nécessaire, il suffit d'ajouter 5.10^{r-1} à $R.10^r$, puis de mettre les r derniers chiffres du résultat à 0.

Remarquons pour terminer que notre méthode profite de ce que, contrairement aux valeurs entières [LRM 14.3.7(2)], les valeurs réelles sont toujours écrites en décimal [LRM 14.3.8(2)].

4.3.5 Décodage des valeurs numériques

Un mini-analyseur syntaxique unique est utilisé pour décoder toutes les valeurs numériques. Il retourne directement les valeurs entières ou flottantes dans le cas de décodages de nombres entiers ou flottants.

Dans le cas du décodage d'un nombre point fixe, il retourne un tuple donnant la valeur entière que représenterait le nombre s'il n'avait pas de point, la valeur de la base, et le nombre de chiffres après le point. Soit V la valeur exacte, b la base, et a (pour AFT) le nombre de chiffres après le point. La valeur retournée est donc une représentation R_b telle que:

$$R_b = V * b^a$$

La représentation cherchée dans le type point fixe, R, est telle que:

$$V = R.2^p.5^q = R_b.b^{-a}$$

$$R = R_b \frac{N}{D} = R_b \frac{2^{p-}.5^{q-}}{2^{p+}.5^{q+}.b^a}$$

R est obtenu en effectuant R_b x N / D.

4.3.6 Formattage et décodage des valeurs de types énumération

Pour permettre le formattage des valeurs de type énumération, un patron de type pour un type énumération contient les chaînes de caractères correspondant aux différentes valeurs. Il suffit donc d'inspecter cette table pour retrouver la représentation d'une valeur.

Cette table pouvant être assez longue, elle est remplacée dans le cas d'un sous-type énumération par un pointeur sur le patron de type du type de base, de façon à éviter de la dupliquer lors de chaque déclaration de sous-type.

La même table est utilisée en sens inverse pour le décodage des valeurs de types énumération.

Cette table n'est pas présente dans le cas des types entiers; une instanciation de ENUMERATION_IO avec un type entier donnera donc des résultats imprévisibles, mais ceci est autorisé par le langage [LRM 14.3.9(15)].

5. Entrées-sorties mode texte et terminaux interactifs

Le manuel de référence ne comporte (intentionnellement) aucune définition précise de ce qu'est un fichier *externe* [LRM 14.1(1)]. Cependant, les opérations définies sur le type abstrait "fichier" sont plus proches de celles applicables à un fichier "disque" qu'à un fichier "exotique", comme un terminal interactif. Dans l'interprète de haut niveau d'Ada/Ed, il était extrêmement pénible d'utiliser TEXT_IO pour effectuer des opérations d'entrées-sorties sur des terminaux interactifs, à cause de la nécessité d'effectuer des opérations "en avance" dans certains cas, et de l'incapacité du système à corriger des situations où l'utilisateur entrait des données qui ne se conformaient pas au modèle décrit par le standard.

Cependant, étant donné le but pédagogique d'Ada/Ed, il était vraisemblable que la plupart des utilisateurs utiliseraient TEXT_IO pour effectuer des opérations interactives à partir d'un terminal de temps partagé, et il nous est apparu nécessaire de fournir une interface plus ergonomique.

5.1 Problèmes relatifs à l'usage de TEXT_IO pour un terminal interactif.

Pour décrire les problèmes susceptibles de se poser si l'implémenteur ne porte pas d'attention spéciale aux entrées-sorties interactives, nous allons tout d'abord supposer une implémentation très naïve: les lignes sont lues ou écrites sous forme d'"enregistrements" (ce terme n'ayant bien sûr rien à voir avec les **record** Ada), la fin de l'enregistrement servant donc de marque de fin de ligne. La fin de fichier est fournie au moyen d'un drapeau quelconque retourné par le système d'exploitation lors d'une tentative de lecture au delà de la fin du fichier. La fin de page est notée par un enregistrement ne contenant que le caractère ASCII Form-Feed (FF). Dans une telle implémentation, le saut d'une marque de fin de ligne consistera à lire effectivement l'enregistrement du fichier d'entrée -dans le cas d'un terminal interactif, une nouvelle ligne est demandée à l'utilisateur.

Supposons qu'un utilisateur naïf, utilisant cette implémentation naïve, écrive le programme suivant:

```
with TEXT_IO; use TEXT_IO;
procedure PERROQUET is
    Ligne       : STRING(1..132);
    Longueur  : NATURAL;
begin
    while not END_OF_FILE loop
        PUT_LINE("Entrez une ligne:");
        GET_LINE(Ligne, Longueur);
        PUT_LINE("Vous avez dit: " & Ligne(1..Longueur));
    end loop;
end PERROQUET;
```

Si nous essayons d'exécuter ce programme interactivement, nous obtiendrons quelque chose comme ceci:

```
?                                   (1)
Entrez une ligne:
?ABCD                               (2)
Vous avez dit:
Entrez une ligne:
?XYZ                                (3)
Vous avez dit: ABCD
Entrez une ligne:
?^Z                                 (4)
Vous avez dit: XYZ
```

Que s'est-il passé? Nous devons d'abord vérifier la présence d'une marque de fin de fichier. Pour celà, le système doit demander une ligne, d'où le point d'interrogation qui apparaît en (1). L'utilisateur voyant apparaître cette demande inattendue, frappe simplement Retour-chariot. Mais maintenant, nous avons une ligne lue. Lors du GET_LINE suivant, le contenu (vide) de cette ligne sera mis dans la variable *Ligne*. Mais comme le terminateur de ligne est sauté, le système force la lecture d'une nouvelle ligne, d'où le point d'interrogation en (2). L'utilisateur croit répondre à la question "Entrez une ligne", mais en fait la réponse à cette question est la ligne vide entrée en (1). Par conséquen, une ligne vide est imprimée, et le processus va se poursuivre. La ligne que l'utilisateur a entrée en (2) sera utilisée par le GET_LINE suivant, mais le fait de sauter le terminateur de ligne déclanchera la lecture d'une nouvelle ligne, ce qui continuera à tromper l'utilisateur. L'ordinateur lit toujours une ligne en avance de ce que croit l'utilisateur. Entrées et sorties sont *désynchronisées*.

Remarquer que nous avons supposé dans l'exemple précédent que l'utilisateur avait été capable de terminer le programme en tapant une marque de fin de fichier (Ici, le caractère CTRL-Z, noté ^Z). Mais ceci peut très bien ne pas fonctionner. Le langage précise que la fin de fichier doit être précédée d'une fin de page. Ici, l'utilisateur n'a pas tapé cette marque de fin de page, et par conséquent le "fichier" qu'il a tapé n'est pas un fichier de texte valide pour Ada, et le système peut ne pas être capable de reconnaitre une fin de fichier valide! Si ceci n'est pas un problème pour les fichiers générés par des programmes Ada (ils contiendront toujours une marque de fin de page correcte), il n'est pas possible d'exiger de l'utilisateur de rentrer une fin de page auparavant chaque fois qu'il souhaite entrer une fin de fichier!

5.2 Lecture différée et gestion des pages

Nous avons montré sur l'exemple précédent que la désynchronisation vient de ce qu'une nouvelle ligne est lue sur le terminal dès que la précédent est épuisée. Si nous voulons éviter la désynchronisation, nous devons *différer* la lecture effective jusqu'au moment où il est effectivement nécessaire d'accéder au contenu de la ligne. Cette méthode de lecture différée n'est pas nouvelle, et a été utilisée dans de nombreuses implémentations de Pascal. Mais en Ada, le problème d'éviter des lectures inutiles est compliqué par les propriétés suivantes:

- Il est possible d'interroger le statut de la fin de page en étant positionné avant une marque de fin de ligne: ceci nécessite un tampon de lecture anticipée d'une ligne;
- Il est possible d'interroger le statut de la fin de fichier en étant positionné avant une marque de fin de ligne, qui peut elle-même être suivie d'une marque de fin de page: ceci nécessite un tampon de lecture anticipée de deux lignes.

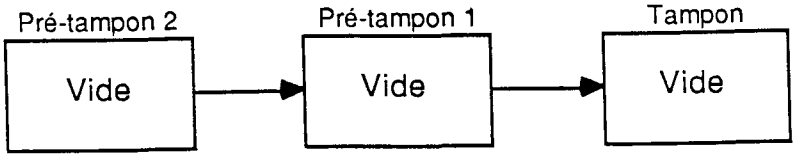
La solution la plus simple pour l'implémenteur est de toujours garder un tampon de deux lignes d'avance; cette solution n'est évidemment pas compatible avec un usage interactif. A la place, nous proposons un mécanisme pour gérer les lectures, avec les spécifications suivantes:

- Accepter tous les fichiers ayant une structure correcte, c'est à dire être *compatible* si l'organe d'entrée standard est redirigé vers un fichier réel, chose qui est souvent autorisée par de nombreux systèmes d'exploitation, mais peut ne pas être connue de l'environnement Ada.

- Etre tolérant aux fichiers n'ayant pas la structure correcte attendue.
- Différer la lecture autant que possible, et n'effectuer de lecture en avance que lorsque celle-ci est absolument nécessaire.

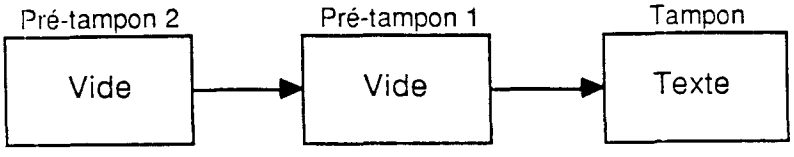
5.2.1 Gestion du texte normal

Notre implémentation utilise un pipe-line de trois tampons. Chaque tampon peut contenir du texte (obtenu depuis le fichier), une marque de fin de page (MFP), une marque de fin de fichier (MFF), ou être vide. Immédiatement après l'ouverture d'un fichier, les trois tampons sont vides:



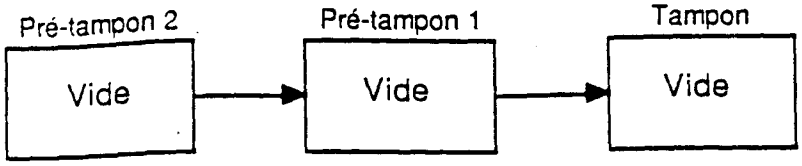
Le vrai tampon de lecture est, bien sûr, celui situé le plus à droite (marqué Tampon). Les deux autres, marqués Pré-tampon, constituent le pipe-line de lecture en avance. Lorsqu'une tentative d'accès au tampon est effectuée alors que celui-ci est vide, une ligne est lue au terminal et mise dans le tampon de lecture. Cette vérification doit impérativement être effectuée avant chaque opération de lecture. Mais une importante propriété du langage est que *toute opération de lecture doit commencer par vérifier l'état de la fin de fichier avant de tenter l'opération.* Par conséquent, la vérification que le tampon n'est pas vide peut être effectuée à l'intérieur de la procédure qui vérifie que le premier tampon ne contient pas une MFF. Ainsi, le mécanisme de lecture différée est complètement caché à presque toutes les procédures d'entrées-sorties, à l'exception de celles qui sont explicitement concernées (SKIP_LINE, fonctions END_OF_...).

Après l'entrée d'une ligne, la structure du pipe-line sera:



Lorsqu'une opération SKIP_LINE est effectuée, qu'elle soit due à un appel explicite à la procédure SKIP_LINE ou implicite lors de l'entrée de caractères qui sautent la marque de

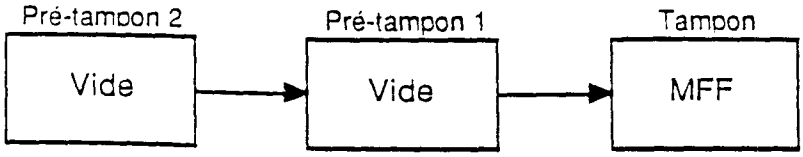
fin de ligne, chaque tampon est transféré dans son voisin de droite, le tampon le plus à gauche étant mis à l'état *vide*. Le pipe-line deviendra:



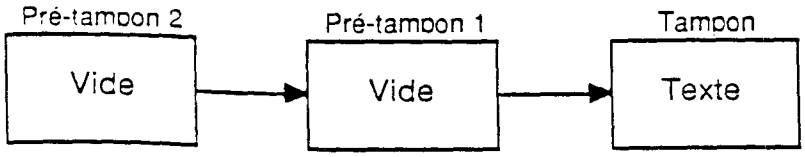
Remarquer que SKIP_LINE n'effectue pas la lecture effective: celle-ci sera différée jusqu'à la prochaine opération de lecture qui l'effectuera lors du test du tampon d'entrée. Noter également que SKIP_LINE soi-même, comme toutes les autres opérations de lecture, commence par vérifier la présence de la fin de fichier, et peut par conséquent provoquer la lecture d'une ligne sur le terminal si le tampon est intialement vide. Mais ceci ne peut se produire que si deux SKIP_LINE sont effectués d'affilée, et il est alors effectivement nécessaire dans ce cas de lire une ligne sur le terminal, pour la purger immédiatement.

5.2.2 Vérification des conditions END_OF_...

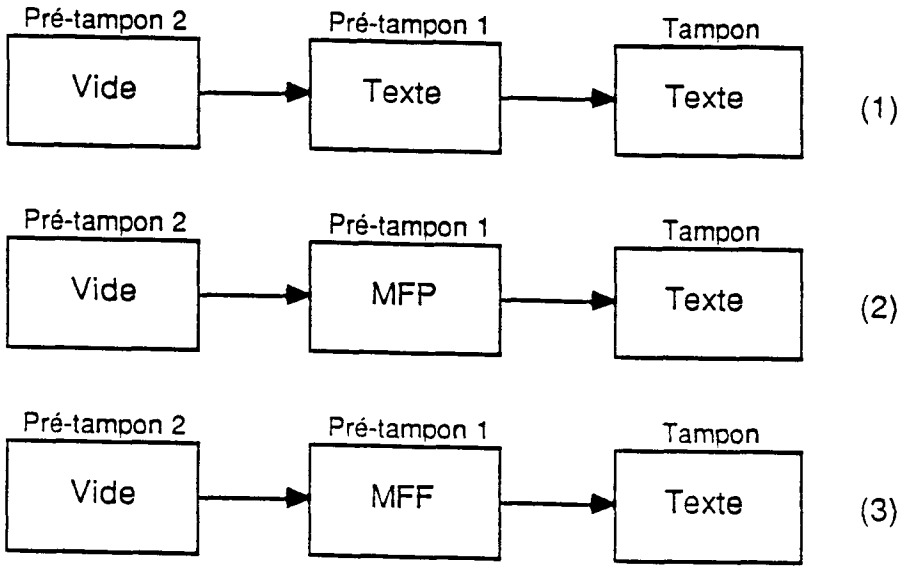
Comme toutes les autres procédures de lecture, les procédures de test de condition de fin (END_OF_LINE, END_OF_PAGE, END_OF_FILE) vont commencer par tester la présence d'une fin de fichier, ce qui peut provoquer une demande de ligne à l'utilisateur. Si le tampon de lecture contient une marque de fin de fichier, les trois conditions sont vraies:



Si le tampon de lecture contient du texte qui n'a pas été complètement épuisé, les trois conditions sont fausses:

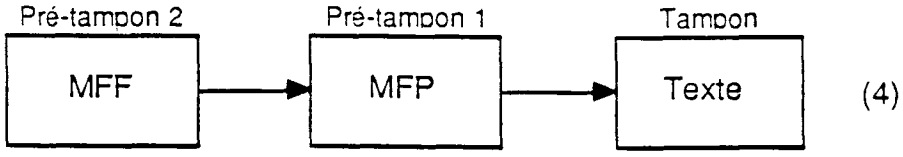


Considérons maintenant le cas où nous nous trouvons à la fin de ligne. La condition END_OF_LINE se contente de renvoyer TRUE dans ce cas. Les autres doivent lire une nouvelle ligne pour tester la fin de page. L'utilisateur pourra alors entrer (1) une ligne, (2) une marque de fin de page, ou (3) une marque de fin de fichier. Noter que cette dernière circonstance n'est normalement pas autorisée, puisqu'une marque de fin de page doit toujours précéder la marque de fin de fichier dans un fichier de données Ada valide, mais qui sait ce que peut taper un usager!



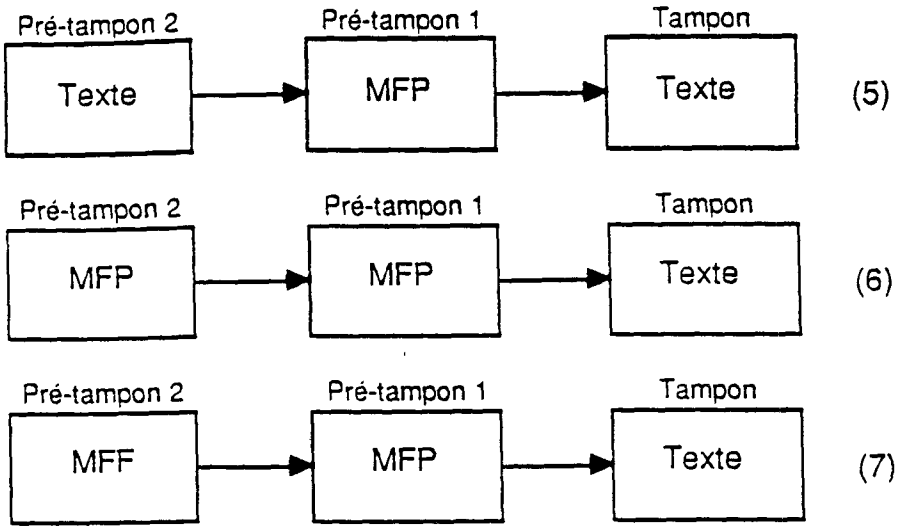
Dans le cas (1), les deux conditions restantes renvoient FALSE. Le prochain SKIP_LINE transférera le contenu du pré-tampon 1 dans le tampon, qui ne sera donc pas vide, et aucune ligne supplémentaire ne sera demandée au terminal.

Dans le cas (3), la réponse incorrecte de l'utilisateur sera *corrigée* en la transformant comme suit:



Maintenant, la condition END_OF_PAGE n'a qu'à tester le second tampon, et renvoyer une valeur en conséquence. Il ne reste plus que le cas de la fin de fichier, et le pipe-line ne peut être que dans l'un des états (2) ou (4). Si le tampon 1 contient MFF (cas (4)), END_OF_FILE renvoie TRUE. Dans le cas (2), il nous faut lire une ligne supplémentaire,

dans le pré-tampon 2. Une fois encore, l'utilisateur peut entrer du texte, une marque de fin de page, ou une marque de fin de fichier:

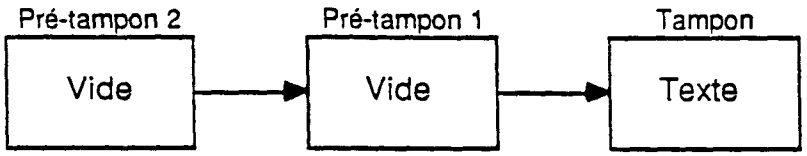


Dans les cas (5) et (6), END_OF_FILE renvoie FALSE. Dans le cas (7), END_OF_FILE renvoie TRUE. Remarquer que le cas (6) est un autre cas de fichier de données Ada incorrect: deux marques de fin de page devraient être séparées par au moins une marque de fin de ligne (une page vide contient au moins une ligne vide). Nous verrons dans la prochaine section comment il est possible de corriger également ce cas de figure.

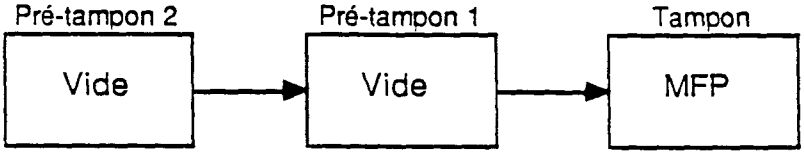
5.2.3 Traitement des fins de pages

Chaque fois qu'une marque de fin de page est mise dans le tampon de lecture, aussi bien suite à la lecture directe d'une marque de fin de page que suite à un transfert depuis le tampon précédent, le compte de page est augmenté d'une unité, et le numéro de ligne est remis à 1. Ensuite, l'opération requise est effectuée à nouveau (lecture effective ou transfert des éléments du pipe-line).

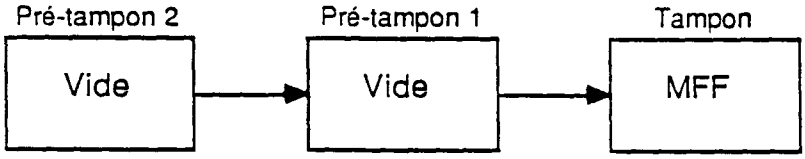
Le pipe-line peut alors se trouver dans l'un des états suivants:



(8)

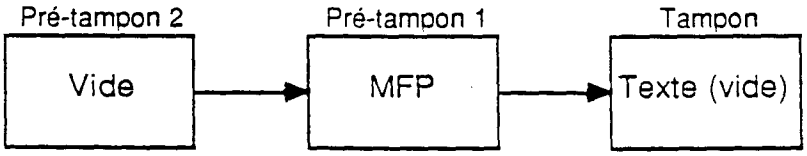


(9)



(10)

Le cas (8) ne pose pas de problème spécial. Le cas (9) peut provenir du cas (6) ci-dessus, ou d'un utilisateur qui tape (de façon illégale) deux marques de fin de page d'affilée. Dans ce cas, nous ajoutons la ligne vide manquante:



(11)

Le cas (10) est correct: il correspond à une fin de page suivie d'une fin de fichier.

En résumé, ce mécanisme nous permet de traiter correctement les fichiers qui suivent le modèle Ada, tout en permettant de corriger les cas les plus fréquents de fichiers incorrects: fins de fichiers non précédées de fin de page, ou deux fins de page consécutives sans ligne vide entre les deux.

Que se passe-t-il du point de vue de l'utilisateur? Aucune lecture n'est demandée par le test de fin de ligne, une ligne est entrée lors du test de fin de page, une ligne est entrée également lors du test de fin de fichier, sauf si l'utilisateur répond par une marque de fin de page, ce qui provoque la lecture d'une ligne supplémentaire. Bien sûr, une ligne supplémentaire est lue si les opérations précédentes suivent immédiatement un appel à SKIP_LINE, mais ce cas est très peu probable, car il est préférable de tester les conditions

END_OF_... lorsque l'on se trouve en fin de ligne plutôt qu'en début. Dans tous les cas, l'utilisateur est libre de taper des marques de fin de page ou de fichier à volonté.

5.3 Effet d'une écriture de marque de fin de page

Comme nous avons choisi de maintenir la compatibilité entre terminaux et fichiers réels (et en fait notre implémentation ne connait même pas avec quelle sorte de fichier elle travaille), l'effet d'un appel à NEW_PAGE ne peut être que d'envoyer une ligne contenant le seul caractère FF. Sur la plupart des terminaux modernes, ceci provoquera l'effet escompté (effacement d'écran ou éjection d'une feuille de papier), et certains systèmes d'exploitation (VMS par exemple) sont capables de transformer le FF en une suite de sauts de lignes si le type déclaré du terminal indique qu'il ne reconnait pas le caractère FF. Cette solution n'est cependant pas parfaite, mais il paraît difficile de faire mieux sans connaissance du type de terminal.

Noter en particulier qu'à la fermeture du fichier, une marque de fin de page doit être écrite, car une marque de fin de page précède toujours la marque de fin de fichier [LRM 14.3(6)].. Comme le fichier STANDARD_OUTPUT, correspondant au terminal généralement, est automatiquement fermé à la fin du programme, ceci signifie qu'un caractère FF lui est envoyé. Autrement dit, à la fin du programme, l'écran est toujours effacé. Ceci peut paraître fâcheux, mais il n'est pas possible de l'éviter sauf à ne pas respecter la structure "officielle" des fichiers Ada et à faire confiance au mécanisme décrit ci-dessus pour rétablir la marque de fin de page manquante lors de la relecture du fichier dans le cas des "vrais" fichiers disque.

Chapitre IV.
Modèle de système de gestion des tâches

Cette partie de la thèse décrit le système de gestion des tâches qui sert de noyau d'exécutif à la Machine Ada. Nous pensons toutefois que le modèle que nous avons été amenés à développer pour les besoins de la Machine Ada peut être utilisé comme base du développement de noyaux temps réel pour machine nue. Nous avons donc séparé cette partie de la thèse en plusieurs parties: après un rappel des principales propriétés du modèle de tâches du langage Ada, nous décrivons notre modèle d'implémentation, indépendamment de toute référence à la Machine Ada. Ceci couvre les différents aspects liés directement à la notion de tâche en Ada, et plus particulièrement les problèmes liés à l'activation et à la terminaison des tâches, aux communications et à la synchronisations entre tâches. Nous décrirons dans le chapitre suivant l'implémentation qui a été faite de ce modèle dans le cadre de la Machine Ada. Le lecteur désirant reprendre notre modèle pour réaliser un noyau temps réel sur machine nue pourra utiliser cette deuxième partie comme un exemple d'une façon dont on peut réaliser le modèle théorique en pratique.

Le système de gestion des tâches est basé principalement sur le modèle décrit dans [Rosen 83]. Toutefois, celui-ci était un modèle purement théorique qui n'avait pas été réalisé effectivement. Si les principes de base sont restés les mêmes (notion d'*équivalences*, utilisation des primitives SIGNAL et WAIT), un certain nombre d'aménagements ont été nécessaires, soit pour corriger certaines erreurs, soit pour adapter le modèle à la norme 1983 ([Rosen 83] était basé sur le document de Juillet 82, [DoD 82]), soit pour tenir compte de la structure d'interprète de la Machine Ada.

Soulignons que le modèle définitif décrit ici a été validé sur la suite 1.7 de l'ACVC, et est utilisé par les compilateurs aujourd'hui commercialisés par NYU.

1. Le modèle Ada de la gestion des tâches

Le modèle de gestion des tâches au sens du langage Ada est décrit dans le chapitre 9 du manuel de référence, et nous supposerons que le lecteur a une certaine connaissance de ce chapitre. Une revue critique du modèle Ada de gestion des tâches peut être trouvé dans [Burns 86]. Nous allons simplement souligner ici certains points délicats qui ont une influence déterminante sur la structure du système, afin d'éviter des confusions par la suite.

Une *tâche* est une entité décrite par une suite d'actions séquentielles, dont l'exécution se produit en parallèle avec d'autres tâches. Ce parallélisme peut être apparent, sur une implémentation mono-processeur, ou réel, sur une implémentation multi-processeur [LRM 9(1..5)].

Les tâches communiquent et se synchronisent explicitement entre elles au moyen d'un mécanisme unique appelé *rendez-vous* [LRM 9.5]. Un rendez-vous a lieu lorsqu'une tâche qui a déclaré un point d'entrée *accepte* cette entrée, et qu'une autre tâche *appelle* cette entrée. Le rendez-vous est une opération fondamentalement dissymétrique. Nous appellerons *propriétaire* la tâche qui a déclaré le point d'entrée, et *client* la tâche qui appelle le point d'entrée. Lorsque les deux tâches sont ainsi arrivées au rendez-vous, celui-ci a lieu: le propriétaire exécute en quelque sorte une procédure sur des paramètres fournis par le client. Ce mécanisme est particulièrement bien adapté à une structure ou les tâches déclarant des entrées se comportent en serveurs pour d'autres tâches. Il existe de plus des points de synchronisation implicites lors de la création, de l'activation, et de la terminaison de tâches [LRM 9.11(2)].

Toutes les explications données ici le sont en terme d'interactions entre tâches. Nous considérons donc que le programme principal est exécuté par une tâche anonyme, comme indiqué dans le manuel de référence [LRM10.1(8)]. Comme nous le verrons dans le chapitre suivant, notre implémentation va au delà de la simple analogie supposée par le manuel de référence, car le programme principal est effectivement appelé par une tâche anonyme, gérée exactement comme les autres tâches du système.

Le *parent* d'une tâche est la tâche qui élabore la partie déclarative ou exécute l'allocateur qui provoque la création de la tâche. Tous les événements en rapport avec la création et l'activation des tâches ont lieu entre la tâche et son parent. Les filles d'une tâche sont les tâches dont celle-ci est le parent.

Le *maître* d'une tâche est défini par [LRM 9.4] comme une *construction*, mais en fait les dépendances sont liées à l'élaboration de la construction donnée par une certain tâche. Pour éviter toute confusion, le terme *maître* tout seul sera réservé au maître dans le sens du LRM; la tâche qui a élaborée la construction sera appelée la *tâche maîtresse*, et l'occurrence (dynamique) de la construction sera appelée la *construction maîtresse*. Tous les événements en rapport avec la terminaison des tâches ont lieu entre la tâche et sa tâche maîtresse. Remarquer que le parent d'une tâche est généralement différent de sa tâche maîtresse. Noter également que la notion de maître est transitive; nous parlerons ainsi *des* maîtres d'une tâche, ce qui correspondra à l'ensemble des maîtres directs et indirects de la tâche. Nous dirons qu'une tâche *dépend* de ses maîtres, et les tâches qui dépendent d'une certaine tâche maîtresse seront appelées ses tâches dépendantes.

Un autre point important à noter est la différence entre les états *achevés* et *terminés*. Une tâche est *achevée* lorsqu'elle a terminé son exécution. Elle devient *terminée* lorsqu'elle est achevée, et que toutes les sous-tâches qui dépendent d'elle sont terminées [LRM 9.4]. Il est important de remarquer qu'une tâche ne peut jamais passer à l'état terminé sans passer auparavant par l'état achevé, même dans le cas de l'instruction **abort** ou d'exception lors de l'activation, et qu'il est possible de le vérifier par programme[1].

2. Conditions de réalisation d'un noyau de gestion de tâches

Un noyau de gestion de tâches pour Ada doit définir ce qu'est l'objet tâche en tant qu'entité Ada, et doit permettre d'implémenter les relations entre tâches, les conditions d'activation, de terminaison et d'avortement définis par le manuel de référence.

Dans le strict cadre du projet NYUADA, nous n'avions pas de contraintes particulières de réalisation: l'espace mémoire était important, et la vitesse d'exécution était loin d'être critique. Nous nous sommes donc attachés à tenter de réaliser un modèle conceptuellement simple, afin de permettre une évolution plus facile vers le produit définitif et à faciliter la maintenance future.

A cette contrainte naturelle, nous avons rajouté une contrainte de réalisation, allant dans le même sens, mais plus spécifiquement destinée à l'implémentation de notre modèle

---

1 La suite de validation 1.4 active lors du développement du système ne comportait pas de tests portant sur ces conditions de terminaison. Nous avons donc été amenés à développer notre propre jeu de tests, qui furent soumis à l'Ada Validation Office, et incorporés dans les versions ultérieures de la suite de validation.

en tant qu'exécutif temps réel: limiter les chaînages au maximum. Les manipulations de chaînes, souvent nombreuses dans un noyau de gestion de tâches, sont généralement des séquences ininterruptibles, et qui peuvent être relativement longues. Elles constituent donc un obstacle majeur à un bon temps de réponse du système. D'autre part, nous avions pu constater de notre expérience professionnelle précédente que la fiabilité d'un système est souvent inversement proportionnelle au nombre de chaînages qu'il contient...

3. Structures et primitives de base

3.1 Bloc de contrôle de tâche

Chaque tâche possède un *bloc de contrôle de tâche* (*Task Control Block*, ou en abrégé TCB) qui contient toutes les informations nécessaires à la gestion de la tâche et à ses relations avec les autres tâches du système. Une tâche étant entièrement caractérisée par son TCB, la *valeur* d'un objet tâche consiste simplement en un pointeur sur son TCB. Par la suite, nous assimilerons complètement, du point de vue de la gestion des tâches, une tâche à son TCB, et des expressions telles que "pointeur sur la tâche" seront utilisées pour "pointeur sur le TCB décrivant la tâche". Nous décrirons dans la suite de ce chapitre les différents éléments du TCB qui participent directement au modèle de gestion des tâches; dans une implémentation pratique du modèle, certains éléments devront y être ajoutés, comme le contexte matériel de la tâche par exemple.

Le TCB d'une tâche est initialisé à la création de la tâche, ou plus exactement la création d'une tâche consiste à créer le TCB qui la décrit.

3.2 Statuts et événements

Les notions de *statut* et d'*événements* sont fondamentales à la compréhension du noyau de gestion des tâches; nous allons donc les présenter ici. La justification des différentes valeurs de statuts et d'événements se fera au fur et à mesure de la description de leur utilisation dans les autres parties de ce chapitre.

A tout moment, une tâche se trouve dans un certain état (en train de calculer, en attente sur délai, etc.). Le *statut* d'une tâche définit, vis à vis du monde extérieur (c'est à dire des autres tâches), ce que la tâche est actuellement en train de faire. Lorsqu'une tâche n'est pas *active* (c'est à dire en train de calculer effectivement, ou prête à prendre le

contrôle de l'UC), elle est *en attente*, et son statut indique la raison précise pour laquelle elle se trouve en attente.

Si une tâche est réveillée d'un état d'attente, c'est que *quelque chose* s'est produit qui a déclenché le réveil de la tâche. Ce *quelque chose* est appelé un *événement*. Le plus récent événement ayant réveillé une tâche fait partie de son TCB; en particulier, lorsqu'une tâche est réveillée, elle peut toujours connaître la raison de son réveil en inspectant le champ *événement* de son TCB.

Le point fondamental est que le *statut* d'une tâche est toujours positionné par la tâche elle-même pour indiquer aux autres tâches la raison de sa mise en attente; l'*événement* est positionné par une autre tâche pour indiquer à sa propriétaire la raison de son réveil.

3.2.1 Valeurs du statut

La notion de statut de tâche est courante dans les systèmes d'exploitation. Là encore, nous nous sommes attachés à limiter le nombre d'états possibles au maximum pour diminuer la complexité globale du système. Nous avons pu nous limiter aux états suivants, dont les relations seront détaillées par la suite:

ACTIVE: La tâche n'est en train d'exécuter aucune action spéciale du point de vue du système de gestion des tâches. Elle peut être effectivement en train de calculer, ou candidate pour obtenir l'UC, auquel cas elle est chaînée dans la table d'ordonnancement. Une telle tâche est appelée *active ordonnançable*. Il peut également s'agir d'une tâche qui a été créée, mais pas encore activée, ou d'une tâche exécutant une instruction **delay** simple. Une telle tâche est appelée *active non ordonnançable*. Une tâche dans l'état ACTIVE ne peut se trouver sur aucune chaîne, à l'exception bien entendu de la chaîne d'ordonnancement ou de la chaîne d'horloge.

ACTIVATING: La tâche est en attente que des tâches filles terminent leur activation.

CALLING_RDV: La tâche est en attente sur un appel d'entrée non temporisé.

TIMED_RDV: La tâche est en attente sur un appel d'entrée temporisé.

SELECTING_NOTERM: La tâche est en attente sur un **select** (ou un **accept** simple) sans alternative **terminate**.

SELECTING_TERM: La tâche est en attente sur un **select** avec alternative **terminate**, mais il existe au moins une tâche dépendant de celle-ci qui n'est pas elle même terminée ou TERMINATABLE.

TERMINATABLE: La tâche est en attente sur un **select** avec alternative **terminate**, et toutes les tâches qui dépendent d'elle sont soit terminées, soit elles-mêmes TERMINATABLE. Le maître dont dépend la tâche ne peut être achevé, car sinon toute la famille serait terminée collectivement [LRM 9.4(6-10)].

COMPLETED: La tâche est en train de quitter un construction maîtresse, et est en attente de terminaison de sous-tâches.

TERMINATED: La tâche est dans l'état terminé au sens du LRM. Elle ne peut plus prendre le contrôle de l'UC. Toute tentative d'appeler un de ses entrées provoquera la levée de l'exception TASKING_ERROR chez l'appelant.

ABNORMAL: La tâche a été avortée en cours de rendez-vous (en tant que client), ou bien elle a été avortée et certaines de ses sous-tâches sont elles-mêmes ABNORMAL; ceci signifie qu'il existe au moins une sous-tâche directe ou indirecte qui est engagée dans un rendez-vous en tant que client.

Les tableaux suivants résument les différents événements pouvant provoquer des transitions entre ces états. Nous reviendrons plus en détails sur le fonctionnement précis de ces changements d'état par la suite.

Etat initial:	ACTIVE (non ordonnançable)
Etat final:	Evénement provoquant la transition:
ACTIVE (ordonnançable)	Activation Expiration de délai
ABNORMAL	Avortement avec des sous-tâches en cours de rendez-vous
TERMINATED	Avortement sans sous-tâches en cours de rendez-vous

Etat initial:	ACTIVE (ordonnançable)
Etat final:	Evénement provoquant la transition:
ACTIVE (non ordonnançable)	Exécution d'un ordre **delay**
ACTIVATING	Activation de sous-tâches (au début du bloc qui a déclaré les tâches)
CALLING_RDV	Appel d'entrée simple
TIMED_RDV	Appel d'entrée conditionnel ou temporisé
SELECTING_TERM	Exécution d'une instruction **select** avec alternative **terminate**, avec des sous-tâches actives ou non TERMINATABLE
SELECTING_NOTERM	Exécution d'une instruction **select** sans alternative **terminate**
TERMINATABLE	Exécution d'une instruction **select** sans alternative **terminate**, toutes les sous-tâches sont TERMINATED ou TERMINATABLE
COMPLETED	Atteinte de la fin d'une construction maîtresse (y compris le corps de la tâche) avec des sous-tâches actives
ABNORMAL	Avortement direct ou indirect, en étant engagé dans un rendez-vous, ou en ayant des sous-tâches (directes ou indirectes) engagées dans un rendez-vous
TERMINATED	Atteinte de la fin du corps de tâche, sans sous-tâche active Avortement sans sous-tâche directe ou indirecte en cours de rendez-vous

---

Etat initial:	ACTIVATING
Etat final:	Evénement provoquant la transition:
ACTIVE (ordonnançable)	Fin d'activation des sous-tâches
ABNORMAL	Avortement direct ou indirect, en ayant des sous-tâches (directes ou indirectes) engagées dans un rendez-vous
TERMINATED	Avortement sans sous-tâche directe ou indirecte en cours de rendez-vous

---

Etat initial:	COMPLETED
Etat final:	Evénement provoquant la transition:
ACTIVE (ordonnançable)	Terminaison des sous-tâches
ABNORMAL	Avortement direct ou indirect, en ayant des sous-tâches (directes ou indirectes) engagées dans un rendez-vous
TERMINATED	Avortement sans sous-tâche directe ou indirecte en cours de rendez-vous

---

Etat initial:	CALLING_RDV
Etat final:	Evénement provoquant la transition:
ACTIVE (ordonnançable)	Fin du rendez-vous
ABNORMAL	Avortement pendant le rendez-vous, ou avant que le rendez-vous ne soit accepté mais avec des sous-tâches en cours de rendez-vous
TERMINATED	Avortement avant que le rendez-vous ne soit accepté sans sous-tâches en cours de rendez-vous

---

Etat initial:	TIMED_RDV
Etat final:	Evénement provoquant la transition:
ACTIVE (ordonnançable)	Fin du rendez-vous
ABNORMAL	Avortement pendant le rendez-vous, ou avant que le rendez-vous ne soit accepté mais avec des sous-tâches en cours de rendez-vous
TERMINATED	Avortement avant que le rendez-vous ne soit accepté sans sous-tâches en cours de rendez-vous

---

Etat initial:	ABNORMAL
Etat final:	Evénement provoquant la transition:
TERMINATED	Fin du rendez-vous de la tâche ou de la dernière de ses sous-tâches ABNORMAL

Etat initial:	SELECTING_NOTERM
Etat final:	Evénement provoquant la transition:
ACTIVE (ordonnançable)	Appel d'entrée, ou expiration du délai
ABNORMAL	Avortement direct ou indirect, en ayant des sous-tâches (directes ou indirectes) engagées dans un rendez-vous
TERMINATED	Avortement sans sous-tâche directe ou indirecte en cours de rendez-vous

---

Etat initial:	SELECTING_TERM
Etat final:	Evénement provoquant la transition:
ACTIVE (ordonnançable)	Appel d'entrée, expiration du délai, ou vérification des conditions de terminaison
ABNORMAL	Avortement direct ou indirect, en ayant des sous-tâches (directes ou indirectes) engagées dans un rendez-vous
TERMINATED	Avortement sans sous-tâche directe ou indirecte en cours de rendez-vous

---

Etat initial:	TERMINATABLE
Etat final:	Evénement provoquant la transition:
ACTIVE (ordonnançable)	Appel d'entrée, expiration du délai, ou vérification des conditions de terminaison
ABNORMAL	Avortement direct ou indirect, en ayant des sous-tâches (directes ou indirectes) engagées dans un rendez-vous
TERMINATED	Avortement sans sous-tâche directe ou indirecte en cours de rendez-vous

---

Etat initial:	TERMINATED
Etat final:	Evénement provoquant la transition:
(détruite)	Tâche maîtresse quittant la construction maîtresse

3.2.2 Valeurs des événements

Nous allons énumérer rapidement ci-dessous les différents événements susceptibles de réveiller une tâche qui s'est mise en attente. On remarquera qu'un identificateur d'entrée est un événement. Lors d'une implémentation de notre modèle de gestion de

tâches, on choisira une représentation des valeurs d'événements permettant commodément d'associer un événement à chaque entrée d'une tâche, par exemple en repérant les identificateurs d'entrées par des entier positifs, et les autres événements par des entiers négatifs.

N° d'entrée
: Un numéro d'entrée est un événement utilisé pour signaler à une tache serveuse en attente de rendez-vous qu'une de ses entrées a été appelée.

TIMER_EVENT
: Evénement utilisé pour réveiller une tâche suite à l'expiration d'un délai.

TERMINATE_EVENT
: Evénement utilisé pour signaler à une tâche en attente sur une instruction **select** avec alternative **terminate** qu'elle peut se terminer.

ENDRDV_EVENT
: Evénement utilisé pour réveiller une tâche appelante lors de la terminaison normale d'un appel de rendez-vous.

TASKERR_EVENT
: Evénement utilisé pour réveiller une tâche suite à une terminaison anormale de rendez-vous. La tâche réveillée doit lever l'exception TASKING_ERROR.

NO_EVENT
: "faux" événement, utilisé pour réveiller une tâche sans modifier son événement courant.

3.3 Equivalences

L'acceptation de rendez-vous utilise les instructions **accept** et **select**, dont il existe différentes formes [LRM 9.5, 9.7.1]:

- instruction **accept** simple
- instruction **select** simple
- instruction **select** avec partie **else**
- instruction **select** avec alternatives **delay**
- instruction **select** avec alternative **terminate.**

De même, l'appel d'entrée peut prendre plusieurs formes:

- appel d'entrée simple
- appel d'entrée conditionnel
- appel d'entrée temporisé

Dans le but de simplifier l'interface avec le système de gestion des tâches, nous avons pu réduire ces différentes sortes d'instructions au moyen *d'équivalences*.

On peut tout d'abord remarquer qu'un **accept** simple est équivalent à un **select** à une seule branche:

```
select
    accept ...
end select;
```

On peut remarquer également qu'un délai de longueur nulle est équivalent à une partie **else**. Enfin, l'absence d'alternative **delay** est équivalente à une alternative **delay** "fantôme", dont la durée serait infinie, et qui ne serait donc jamais sélectionnée.

Une remarque s'impose à propos de l'équivalence entre partie **else** et branche **delay** de délai 0: cette équivalence est autorisée, et ne pose pas de problème dans le cas d'un système centralisé. Toutefois, il existe une légère différence de sémantique dans un système distribué: une branche **else** ne peut être sélectionnée que si l'on a pu s'assurer qu'aucune tâche n'était demandeuse sur la ou les entrées ouvertes; par contre une branche **delay** peut être sélectionnée si aucune demande de rendez-vous n'a été reçue par le serveur à l'expiration du délai demandé. Dans un système distribué, où l'interrogation de l'état de tâches distantes peut nécessiter l'échange de messages sur un réseau avec des temps de transmission non négligeables, il est possible de sélectionner une branche **delay** munie d'un délai assez petit alors même que des tâches se seraient trouvées en attente d'appel d'une des entrées ouvertes du **select**, si le temps de réponse du réseau est supérieur à la valeur du délai. Ceci signifie que l'équivalence ci-dessus est une commodité autorisée pour l'implémenteur, mais que le programmeur ne doit pas supposer cette équivalence, les sémantiques pouvant être différentes sur certaines implémentations.

Si nous supposons l'existence d'une valeur spéciale, appartenant au type DURATION, appelée ENDLESS, qui correspond à une durée infinie, nous pouvons établir les équivalences suivantes, compte tenu des remarques précédentes:

Instructions Ada	Instructions équivalentes

select
  accept E1 **do** <Instr. 1>;
or **accept** E2 **do** <Instr. 2>;
  .
  .
else <Instr. **else**>;
end select;
	select
	  accept E1 **do** <Instr. 1>;
	or **accept** E2 **do** <Instr. 2>;
	  .
	  .
	or **delay** 0 **do** <Instr. **else**>;
	end select;

select
  accept E1 **do** <Instr. 1>;
or **accept** E2 **do** <Instr. 2>;
  .
  .
or **accept** En **do** <Instr. n>;
end select;
	select
	  accept E1 **do** <Instr. 1>;
	or **accept** E2 **do** <Instr. 2>;
	  .
	  .
	or **accept** En **do** <Instr. n>;
	or **delay** ENDLESS;
	end select;

accept E **do** <Instr.>;	**select** **accept** E **do** <Instr.>; **or** **delay** ENDLESS; **end select**;
T.E;	**select** T.E; **or** **delay** ENDLESS; **end select**;

select
  T.E; <Instr.>;
else <Instr. **else**>;
end select;
	select
	  T.E; <Instr.>;
	or **delay** 0 **do** <Instr. **else**>;
	end select;

Figure 1: Equivalences

Grâce à ces équivalences[2], nous avons pu réduire les différentes variétés d'instructions liées au mécanisme de rendez-vous à deux instructions seulement: l'attente sélective avec alternative **delay**, et l'appel d'entrée temporisé. Le cas de l'alternative **terminate** sera traité à part, comme une alternative spéciale qui pourrait, du point de vue de notre système, cohabiter avec une alternative **delay**. La vérification de la non présence simultanée d'une alternative **delay** et d'une alternative **terminate** ([LRM 9.7.1(3)]) est du ressort de la partie avant du compilateur.

---

[2] A notre connaissance, ces équivalences ont été proposées pour la première fois dans [Rosen 83]. D'autres auteurs ont les ont depuis plus ou moins reprises ([Leathrum 84] par exemple), sans toujours citer les sources...

3.4 Primitives de synchronisation

Toutes les synchronisations entre tâches sont effectuées dans notre modèle au moyen de deux procédures appelées SIGNAL et WAIT. Malgré leurs noms, ces deux procédures n'ont pas exactement les mêmes fonctionnalités que les procédures *signal* et *wait* habituelles, telles que décrites dans [Dijkstra 68].

Une tâche exécutant une primitive WAIT relâche le contrôle de l'UC. Elle fournit un argument à la procédure, qui est une durée maximale pour la mise en attente. Si rien d'autre ne vient réveiller la tâche auparavant, elle sera réveillée automatiquement à l'expiration de ce délai. Ce délai peut être 0, auquel cas la tâche est immédiatement ré-ordonnancée, et il peut également être la valeur spéciale ENDLESS, pour un délai "infini". Avant d'exécuter une primitive WAIT, la tâche doit s'être correctement chaînée si cela est nécessaire, et elle doit avoir mis son statut à la valeur correspondant à la raison pour laquelle elle se met en attente. La primitive WAIT est le *seul* endroit du système où un changement de contexte de tâche puisse se produire; elle comporte donc le mécanisme d'ordonnancement.

Le passage d'une tâche dans la primitive WAIT se passera donc de la façon suivante:

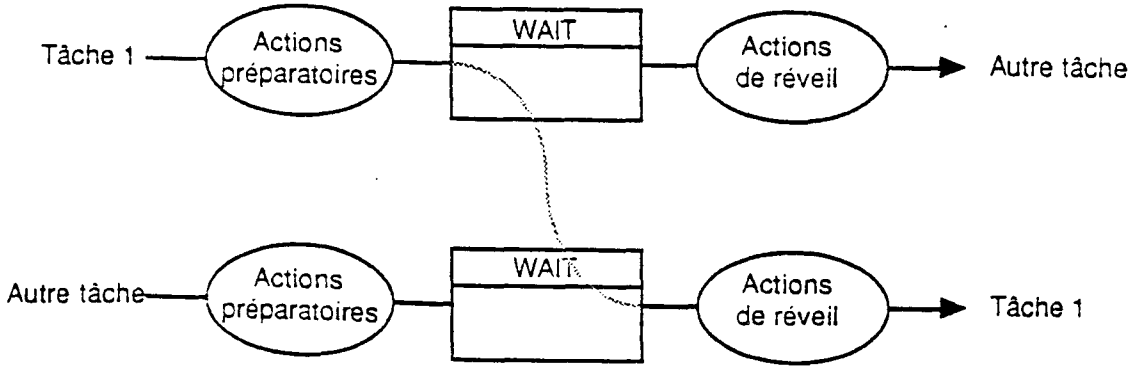


Figure 2: Primitive WAIT

A un instant donné, une tâche entre dans la primitive WAIT, et c'est une tâche différente qui en sort. Plus tard, la première tâche est réveillée, et reprend son exécution là où elle avait été suspendue, à l'intérieur du WAIT; tout le contexte de la tâche a été restauré. La suspension est totalement transparente pour la tâche qui exécute le WAIT.

Une tâche exécute une primitive SIGNAL pour forcer une autre tâche à sortir de l'état d'attente. Elle doit fournir un argument, *l'événement*, qui donne la raison pour laquelle elle réveille l'autre tâche. Cet événement est rangé par la primitive SIGNAL dans le TCB de la

tâche réveillée, sauf s'il s'agit de l'événement spécial NO_EVENT, qui signifie que l'événement courant de la tâche réveillée ne doit pas être modifié. Les cas d'utilisation de cet événement spécial seront vus par la suite. Lorsqu'une tâche est l'objet d'une procédure SIGNAL, elle est automatiquement retirée de tous les chaînages sur lesquels elle se trouve, et mise dans la table d'ordonnancement. Si elle était en attente d'acceptation de rendez-vous, les entrées marquées ouvertes dans sa table de rendez-vous sont fermées. Une tâche a le droit d'exécuter un SIGNAL sur elle-même. Ceci a pour effet de la rechaîner en queue de file d'ordonnancement, donc de lui faire perdre la fin de sa tranche de temps. Ceci se produit lorsqu'une tâche exécute un WAIT(0). Noter que le statut de la tâche est suffisant pour déterminer sur quels chaînages la tâche se trouve, de façon à éviter des recherches inutiles.

Le statut de la tâche n'est pas changé lorsqu'elle est l'objet d'une procédure SIGNAL: cette information est nécessaire pour effectuer certaines actions au moment où la tâche reprend le contrôle de l'UC. Ce n'est qu'à ce moment là que son statut reprendra la valeur ACTIVE.

Lorsqu'une tâche est en attente avec un délai différent de ENDLESS, et que rien ne se passe durant ce délai, elle est automatiquement réveillée à l'expiration de celui-ci, avec un événement appelé TIMER_EVENT. Tout se passe comme si une "tâche horloge" avait exécuté une primitive SIGNAL avec cet événement sur la tâche.

En conséquence, les assertions suivantes sont toujours vérifiées:

- Lorsqu'une tâche est en attente, sont statut donne la raison pour laquelle elle attend; Si elle n'est pas en attente, son statut ne peut être que ACTIVE.
- Lorsqu'une tâche est réveillée, elle trouve toujours dans son événement la raison pour laquelle elle a été réveillée.
- Aucune tâche n'émet de SIGNAL sur une tâche active. Remarquer que cette assertion est vérifiée dynamiquement par le système, et provoque la levée de l'exception prédéfinie (dans notre système) SYSTEM_ERROR dans la tâche signalante en cas de violation.
- Une tâche active ne peut se trouver sur aucun chaînage.

Il peut se produire cependant qu'une tâche ait à attendre plusieurs événements simultanément: par exemple, à la fin d'une partie déclarative, une tâche doit attendre que toutes ses tâches filles aient terminé leur activation avant d'avoir le droit de poursuivre. S'il y a N tâches qui doivent être attendues, ceci peut être réalisé par une séquence:

```
for I in 1..N loop
    WAIT(endless);
end loop;
```

Cependant, ceci provoquera N-1 réordonnancements inutiles de la tâche. De plus, ceci pourrait créer des séquences critiques, car une tâche pourrait essayer d'émettre un SIGNAL sur la tâche pendant que celle-ci est temporairement dans l'état ACTIVE, ce qui est interdit (cf. plus haut). Pour ces raisons, un mécanisme spécial a été mis en place pour une tâche devant attendre plusieurs événements. Elle place dans un élément de son TCB appelé *multiwait_count* le nombre d'événements attendus. Chaque tâche désirant effectuer un primitive SIGNAL utilise alors la procédure MULTI_SIGNAL, qui met à jour l'événement de la tâche de la même façon que SIGNAL, décrémente *multiwait_count*, et ne réveille effectivement la tâche que si *multiwait_count* passe par zéro. Des exemples d'utilisation de ce mécanisme se trouvent dans la description des mécanismes d'activation et de terminaison des tâches.

4. Création et activation des tâches

4.1 L'environnement de tâche

L'élaboration d'une partie déclarative peut mettre en jeu une succession de plusieurs création de tâches, suivie par une activation en parallèle des tâches créées. Ce processus peut cependant être récursif, car la création d'un objet peut déclencher l'évaluation d'un allocateur, qui peut à son tour créer plusieurs tâches qui doivent être activées ensemble, avant que la valeur accès ne soit retournée. L'élaboration d'un paquetage peut également déclencher la création et l'activation d'un certain nombre de tâches durant l'élaboration de la partie déclarative.

L'exemple suivant montre différents points de création et d'activation suivant les contextes:

```
procedure MAIN is
    task type T;

    type ACC_T is access T;
    type ARR_T is array(1..10) of T;
    type ARR_ACC_T is array(1..10) of ACC_T;
    type ACC_ARR_T is access ARR_T;

    task body T is
        ...
    end T;

begin
    declare
        T1  : T;                -- création de T1
        T2  : T;                -- création de T2
        AT1 : ACC_T;            -- création de AT1.all et activation

        package P is
            PT1 : T;            -- création de PT1
            APT : ARR_T;        -- création de 10 tâches
        end P;

        TAB1 : ARR_ACC_T := (others => new T);
                                -- création et activation de TAB1(1), puis de
                                -- TAB1(2), etc.
        TAB2 : ACC_ARR_T := new ARR_T;
                                -- création de 10 tâches, puis activation en
                                -- parallèle
        T3 : T;                 -- création de T3

        package body P is
            PT2 : T;            -- création de PT2
        begin                   -- activation de PT1, PT2, et des 10 tâches
            null;               --de APT
        end;

    begin                       -- activation de T1, T2 et T3
        null;
    end;

end MAIN;
```

Comme nous pouvons le voir dans l'exemple précédent, des tâches sont créées, puis un autre ensemble de tâches peut être créé et activé, puis nous pouvons reprendre la création de tâches appartenant au premier ensemble, et finalement les activer en parallèle. Nous pouvons donc définir des familles de tâches, qui sont créées dans le même contexte et activées ensemble. Mais pendant la création d'une telle famille, une autre famille peut

être créée et activée. Une telle famille est appelée dans notre modèle un *environnement de tâche.*

Pour gérer correctement les créations et les activations comme indiqué ci-dessus, il est nécessaire de maintenir une pile d'environnements de tâches. Les élaborations d'objets tâches provoqueront l'ajout de l'objet dans l'environnement qui se trouve sur le sommet de la pile. L'activation d'une famille consistera à activer toutes les tâches se trouvant dans l'environnement situé au sommet de la pile, puis à dépiler cet environnement. Un nouvel environnement est créé lors de l'entrée dans une partie déclarative ou lors de l'exécution d'un allocateur désignant un type contenant des tâches.

4.2 Création de tâches

La création d'une tâche revient, comme indiqué plus haut, à créer un TCB décrivant la tâche. Ce TCB doit obligatoirement être complet, car, vue de l'extérieur, rien ne distingue une tâche créée, mais non encore activée, d'une tâche activée. En particulier, ses entrées peuvent être appelées, ses attributs interrogés, et elle peut être avortée, comme le montre l'exemple suivant:

```
procedure MAIN is

    task T is
        entry E;
    end T;                  -- La tâche est créée ici

    task body T is
        ...
    end T;

    package P is
    end P;

    package body P is
    begin
        if not T'TERMINATED then          -- Exécuté ici
                                          -- Interrogation d'attribut
            select
                T.E;                      -- Appel d'entrée
            else
                abort T;                  -- Avortement
            end select;
        end if;
    end P;

begin                   -- T aurait été activée ici
    null;
end MAIN;
```

4.3 Activation des tâches

L'activation d'un ensemble de tâches appartenant au même environnement de tâche va consister à initialiser correctement le contexte des tâches, à leur permettre d'élaborer leur partie déclarative tout en suspendant la tâche parent, puis à réactiver la tâche parent, éventuellement en levant l'exception TASKING_ERROR dans celle-ci si l'une au moins des tâches activées s'est terminée à cause de la levée d'une exception lors de son activation ([LRM 9.3(3)]). Remarquer que comme un environnement de tâche différent est créé pour chaque partie déclarative et pour chaque allocateur, toutes les tâches apppartenant à un même environnement de tâches dépendent du même maître.

Chaque tâche activée est initialisée avec les informations appropriées concernant son maître, chaînée sur la chaîne correspondante, et entrée dans la table d'ordonnancement. La tâche parent est alors initialisée avec son *multiwait_count* à la valeur du nombre de tâches activées, son *current_event* à NO_EVENT, et son *status* à ACTIVATING. Le parent exécute alors une primitive WAIT à délai infini.

Les tâches nouvellement activées prendront le contrôle de l'UC et élaboreront leur partie déclarative. A la fin de cette élaboration, elles exécutent une primitive MULTISIGNAL sur leur parent, avec l'événement NO_EVENT si l'élaboration s'est bien passée, ou TASKERR_EVENT si une exception a été levée durant cette élaboration. Comme NO_EVENT ne change pas la valeur courante de l'événement de la tâche qui en est l'objet, la tâche parent sera réveillée après que toutes les tâches filles aient terminé leur activation avec un événement qui sera NO_EVENT si et seulement si toutes les tâches ont pu s'élaborer avec succès. Si au moins une tâche a levé une exception, la tâche parent sera réveillée avec un statut TASKERR_EVENT, ce qui provoquera automatiquement la levée de l'exception TASKING_ERROR lorsque la tâche parente sera réveillée.

Les échanges de signaux durant l'activation des tâches sont résumés dans la figure suivante:

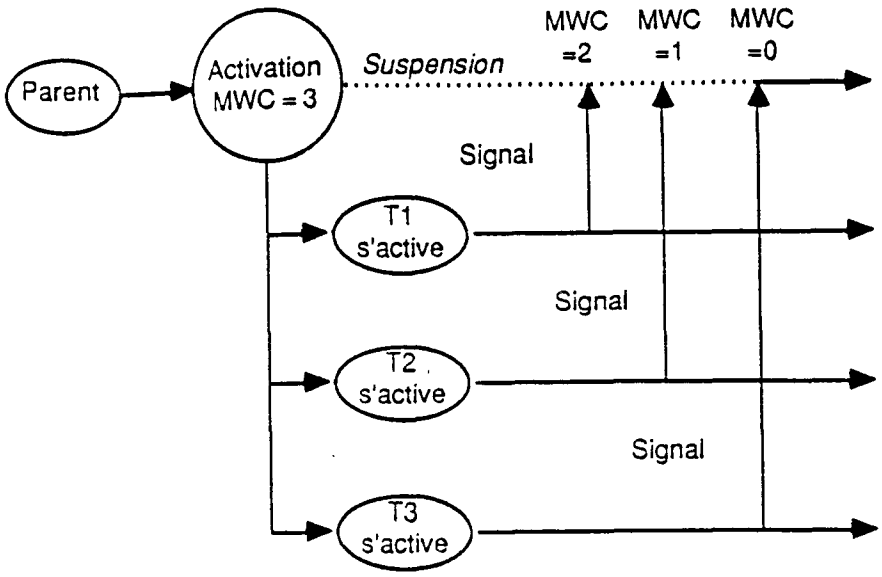


Figure 3: Echange de signaux durant l'activation des tâches

Remarquer qu'il est nécessaire de marquer de façon spéciale les tâches qui se sont fait avorter avant d'avoir été activées, car elles doivent passer immédiatement à l'état terminé sans s'activer (elles ne peuvent avoir de tâches dépendantes, puisqu'elles n'ont pas élaboré de partie déclarative).

5. Gestion des rendez-vous

Toutes les entrées, qu'il s'agisse d'entrées simples ou de membres de familles d'entrées, sont identifiées par un *indice absolu d'entrée*, unique pour chaque entrée d'un même type tâche. Il n'y a rien de spécial aux entrées appartenant à une famille, si ce n'est que leur indice absolu d'entrée doit parfois être calculé dynamiquement, car le sous-type d'indice d'une famille d'entrée peut être dynamique. Dans le restant de ce chapitre, les explications seront données en terme d'indices absolus d'entrée.

Chaque tâche possède dans son TCB une table de rendez-vous. Cette table possède une entrée pour chaque indice absolu d'entrée de la tâche, chacune d'elles consistant en une paire de pointeurs de tâches, appelés *first* et *last*. *First* pointe sur la première tâche de la chaîne des tâches en attente sur l'entrée considérée, il est nul si la chaîne est vide. *Last* pointe sur la dernière tâche de la chaîne, il est égale à *first* s'il n'y a qu'une seule tâche sur la chaîne, et il est non significatif si *first* est nul.

Lorsque le propriétaire de l'entrée est en attente d'acceptation de rendez-vous, c'est à dire que personne n'a appelé une de ses entrées ouvertes, le *first* correspondant aux entrées ouvertes a une valeur conventionnelle servant à signaler aux autres tâches que l'entrée est ouverte.

Lorsqu'une tâche est en attente d'appel de rendez-vous, elle est reliée à la tâche appelée sur la chaîne correspondante. Comme lorsqu'une nouvelle tâche effectue un appel d'entrée elle est toujours chaînée en queue (afin de garantir l'ordre correct de service des appels d'entrée, [LRM 9.5(15)]), le fait d'avoir un pointeur sur la dernière tâche de la chaîne évite d'effectuer une recherche dans la liste afin de déterminer son dernier élément. La tâche appelante est chaînée par un lien double, constitué de *next_task*, pointeur sur la prochaine tâche en attente sur la même entrée, et *previous_task*, pointeur sur la tâche précédente en attente sur la même entrée. *Next_task* est nul si la tâche est la dernière de la chaîne (elle est donc pointée par le *last* du propriétaire), et *previous_task* est nul si la tâche est la première de la chaîne (elle est donc pointée par le *first* du propriétaire). Le but de ce double chaînage est de permettre d'enlever facilement la tâche de la chaîne lors de l'expiration du délai dans un appel d'entrée temporisé, et, à un moindre degré, lorsque la tâche est avortée.

Lorsqu'une tâche exécute une instruction **accept**, elle inspecte les entrées de sa table de rendez-vous correspondant aux alternatives ouvertes pour voir s'il existe des tâches en attente. Si aucune n'est trouvée, la tâche indique qu'elle est prête à accepter un appel d'entrée en mettant les *first* correspondant à la valeur conventionnelle, puis exécute une primitive WAIT, dont la durée dépend des cas de figure (cf. équivalences). Sinon, la tâche appelante est sortie de la chaîne et mise en tête de la chaîne *serviced* (voir gestion de l'avortement). Si la tâche appelante était chaînée sur l'horloge (ce que l'on sait sans avoir à inspecter la chaîne d'horloge, grâce à l'indicateur *on_clock*), la temporisation est annulée. Le rendez-vous est ensuite exécuté, et à la fin, l'appelant (qui est la première tâche sur la chaîne *serviced*) est réveillé par une primitive SIGNAL(ENDRDV_EVENT).

L'appelant inspecte l'entrée dans la table de rendez-vous du propriétaire, et si celui-ci est en attente (fait connu par la valeur spéciale de la tête de chaîne), il est réveillé par une primitive SIGNAL dont la valeur d'événement est l'index absolu d'entrée. Dans tous les cas, l'appelant se chaîne sur l'entrée correspondante de la table de rendez-vous.

Les échanges de signaux lors d'un rendez-vous sont résumés dans la figure suivante:

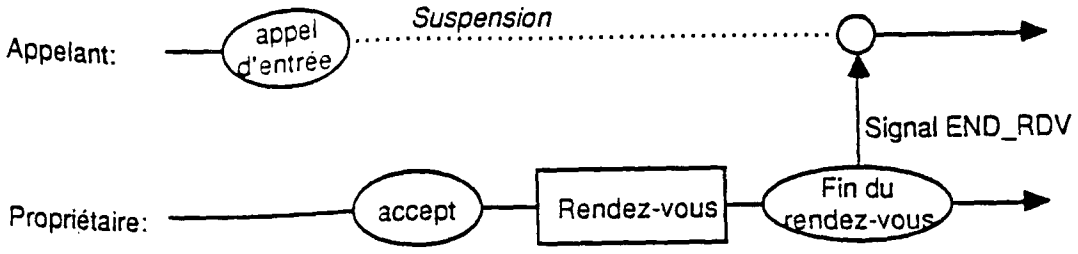


Figure 4: Rendez-vous, appelant arrivant en premier

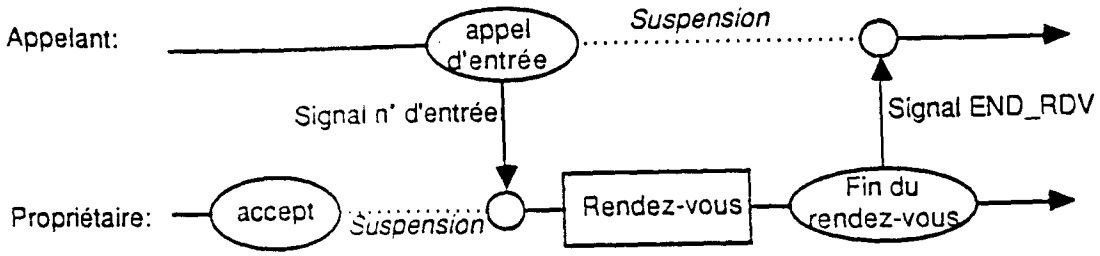


Figure 5: Rendez-vous, propriétaire arrivant en premier

On remarquera sur ces figures que les parties droites sont semblables: la séquence de fin de rendez-vous ne dépend pas de qui est arrivé en premier.

On pourrait craindre l'existence d'une séquence critique entre le moment où une tâche qui était en attente de rendez-vous est l'objet d'un SIGNAL et celui où elle reprend effectivement le contrôle de l'UC, car dans cet intervalle son statut continue à indiquer qu'elle est en attente de rendez-vous, alors qu'en fait un tâche a déjà appelé l'entrée. Mais en fait, aucun rendez-vous n'est possible après qu'elle ait été réveillée, puisque toutes les entrées ont été marquées fermées par la primitive SIGNAL. Aucune condition de course ne peut donc se produire si des tâches appellent une entrée dans cet intervalle.

6. Gestion des priorités

La priorité d'une tâche peut changer pendant l'acceptation d'un rendez-vous, car le rendez-vous doit être exécuté avec la plus haute des priorités des deux tâches concernées [LRM 9.8(5)]. Comme le corps de l'instruction **accept** peut contenir d'autres instructions **accept**, qui peuvent eux-mêmes impliquer d'autres changements de priorité, il n'est pas possible de préserver la priorité précédente dans le contexte du propriétaire, sauf à gérer

une pile spéciale pour celà, avec les complications que celà suppose (gestion des débordements de pile, etc.). Une solution plus simple serait de sauvegarder la priorité sur la pile d'exécution, mais il serait alors difficile de la retrouver dans le cas de levée d'exception, la pile étant alors redescendue d'office. Il existe fort heureusement une solution plus simple: la priorité du propriétaire est préservée dans le TCB de l'appelant (élément *called_priority*). Cette solution est parfaitement sûre, puisqu'une tâche ne peut être sur plusieurs chaînes à la fois, et que l'appelant ne peut être déchaîné avant la fin du rendez-vous, *même dans le cas d'une exception ou d'un avortement.*

La modification de la priorité du propriétaire est effectuée par celui-ci si le rendez-vous est immédiatement possible au moment où il exécute l'instruction **accept**, mais autrement elle est effectuée par l'appelant *avant* d'effectuer la primitive SIGNAL sur le propriétaire lorsque celui-ci est en attente, de façon à lui permettre d'être réordonnancé immédiatement avec la plus haute priorité. Si la priorité était changée par le propriétaire au moment où il est réveillé, il se trouverait en effet réordonnancé avec sa priorité propre, et ne passerait en priorité élevée qu'après avoir pris le contrôle de l'UC.

7. Dépendances et terminaison; avortement

7.1 Achèvement d'une construction maîtresse

Lorsqu'une tâche quitte une construction maîtresse, elle doit vérifier si la tête de chaîne des tâches dépendant de cette construction est nulle. Si oui, il n'existe aucune tâche dépendante, et la construction est quittée immédiatement. Si non, la tâche suit la chaîne pour vérifier si toutes les tâches de la chaîne (c'est à dire toutes les tâches dépendant de la construction maîtresse que l'on est en train de quitter) sont soit dans l'état TERMINATED, soit dans l'état TERMINATABLE (ce qui signifie qu'elles sont en attente sur un **select** avec une alternative **terminate**, et que leurs propres sous-tâches sont elles-même toutes TERMINATED ou TERMINATABLE). Si certaines sous-tâches sont encore actives (c'est à dire ni TERMINATED, ni TERMINATABLE), la tâche se met à l'état COMPLETED, et exécute une primitive WAIT avec un délai infini. Elle positionne également *multiwait_count* au nombre de tâches encore actives. Chaque sous-tâche dépendant de la construction maîtresse considérée qui se terminera à partir de ce moment exécutera une primitive MULTISIGNAL sur la tâche maîtresse, qui ira revérifier lorsqu'elle est réveillée si toutes les sous-tâches sont terminées. Remarquer que cette séquence doit être réexécutée lorsque la tâche sera réveillée, car il n'y a pas de garantie que lorsque *multiwait_count* passe par 0, toutes les sous-tâches soient terminées; en effet, de nouvelles sous-tâches ont pu être

créées pendant que la tâche maîtresse était en attente par l'exécution par une sous-tâche d'un allocateur sur un type contenant des tâches, le type de l'allocateur étant déclaré dans la construction maîtresse.

L'exemple ci-dessous donne un exemple de ce cas de figure:

```
procedure MAIN is

    pragma PRIORITY(8);

    task type T is
        pragma PRIORITY(6);
    end T;

    type A_T is access T;

    task MECHANTE is
        pragma PRIORITY(7);
    end MECHANTE;

    task body MECHANTE is
    T1 : A_T;
    begin
        T1 := new T;
    end MECHANTE;

    task body T is
    begin
        null;
    end T;

begin
    null;
end MAIN;
```

La figure ci-dessous montre le déroulement des événements lors de l'exécution de ce programme:

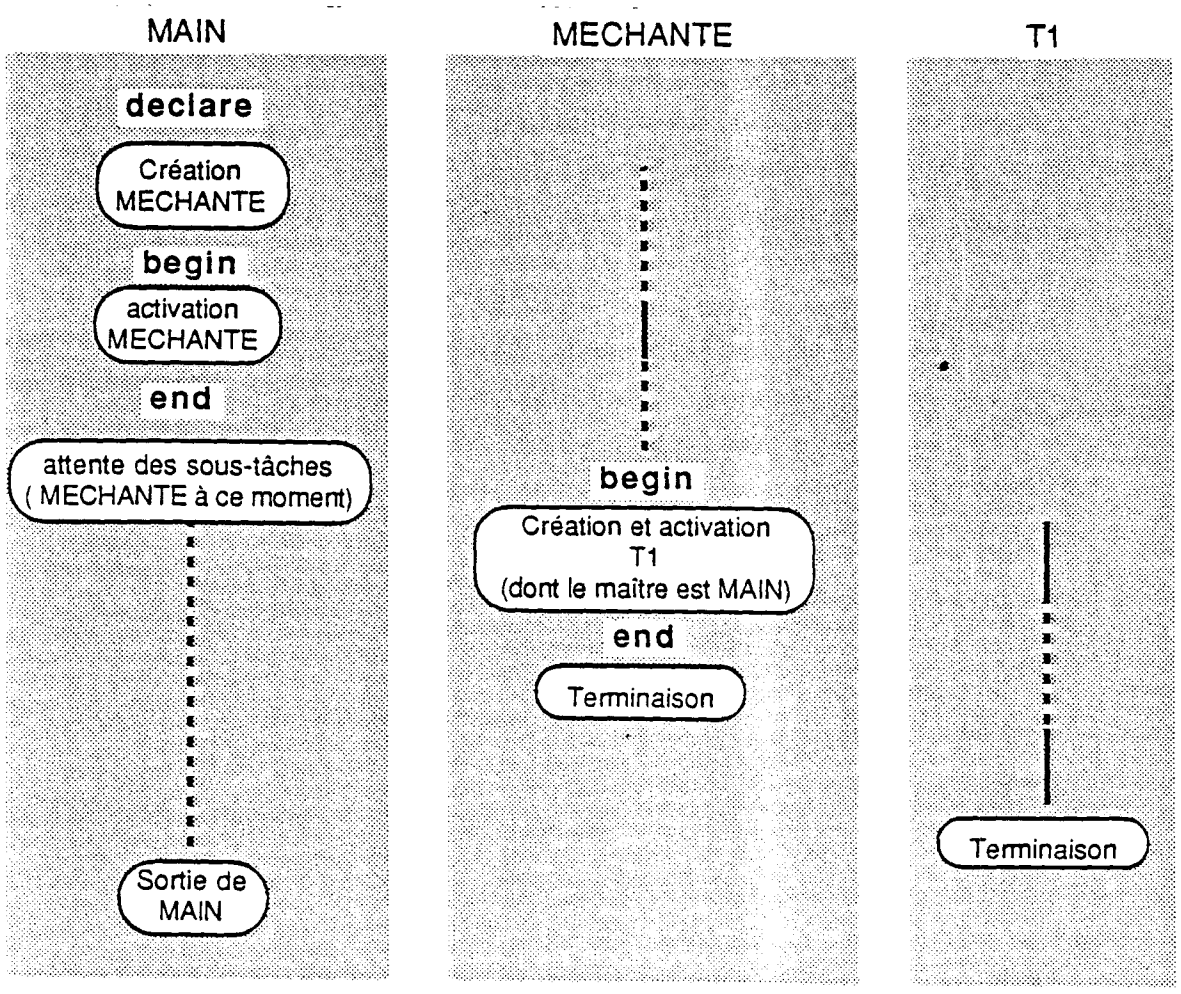


Figure 6: Déroulement des événements

Le compte de tâches n'est donc en fait que la borne inférieure du nombre de tâches qui doivent se terminer avant que la construction maîtresse ait une chance de pouvoir être quittée. En fait, ce compte pourrait être 1 (ce qui revient à utiliser SIGNAL au lieu de MULTISIGNAL), et la tâche maîtresse serait réveillée chaque fois qu'une sous-tâche se termine. Notre solution permet cependant d'éviter la plupart des réordonnancements inutiles de la tâche maîtresse. Il est même vraisemblable qu'en pratique, la tâche maîtresse ne sera réveillée que lorsque toutes les sous-tâches seront terminées, le cas mentionné ci-dessus où cela ne serait pas vrai devant être rarissime en pratique.

Si toutes les sous-tâches sont soit TERMINATED, soit TERMINATABLE, la tâche maîtresse exécute une primitive SIGNAL sur toutes les tâches dans l'état TERMINATABLE avec l'événement TERMINATE_EVENT, et initialise son *multiwait_count* au nombre de tâches ainsi signalées. Une fois réveillée, chacune des sous-tâches sélectionnera l'alternative correspondant à cet événement, et répétera le même processus sur ses propres sous-tâches, puis exécutera une procédure MULTISIGNAL sur sa tâche maîtresse. Lorsque toutes les sous-tâches seront terminées, la tâche maîtresse sera réveillée à son tour. Ce mécanisme garantit donc un ordre correct de terminaison. Les échanges de signaux lors de ce mécanisme de terminaison sont résumés dans la figure suivante:

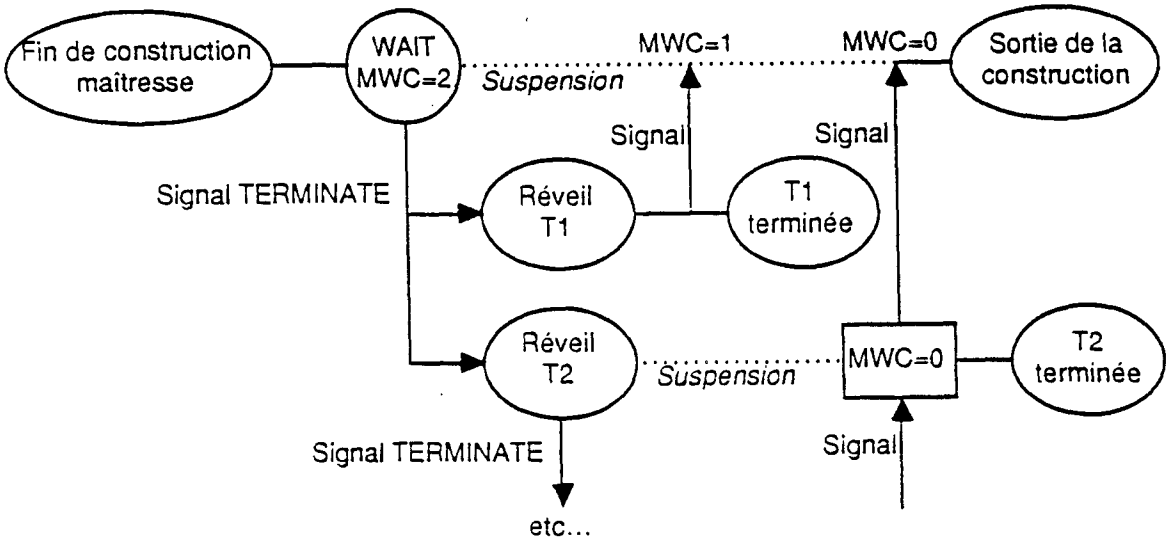


Figure 7: Echange de signaux durant la terminaison

7.2 Select avec alternative terminate

Nous allons maintenant décrire les événements qui se produisent lorsqu'une tâche exécute une attente sélective avec une alternative **terminate**.

Toute tâche possède dans son TCB un élément appelé *dependent_not_term*, qui contient le nombre de tâches dont celle-ci est la tâche maîtresse, et qui ne sont ni dans l'état TERMINATED, ni dans l'état TERMINATABLE. Cet élément ne fait pas double emploi avec les chaînes de dépendances: en effet, les conditions de terminaison dans le cas d'une alternative **terminate** [LRM 9.4(7-10)] exigent de connaître toutes les tâches qui dépendent d'un maître considéré. Or, si un **select** ne peut apparaître dans une procédure interne [LRM 9.5(8)], il peut apparaître dans un bloc qui peut être une construction maîtresse; il ne suffit donc pas de tenir compte des tâches dont la tâche est également la

construction maîtresse. Il est donc apparu plus simple de maintenir ce compte, plutôt que d'aller rechercher dans la pile toutes les chaînes de dépendances.

Si la tâche possède des sous-tâches qui ne sont ni TERMINATED, ni TERMINATABLE, ce qui est équivalent à dire que son *dependent_not_term* est différent de 0, alors nous sommes sûrs que la tâche ne peut sélectionner l'alternative **terminate** et se terminer. Elle se met en attente de rendez-vous avec le statut SELECTING_TERM.

Si son *dependent_not_term* est nul, alors il est *possible* que les conditions requises pour terminer soient remplies. La tâche positionne alors son statut à TERMINATABLE, et décrémente le *dependent_not_term* de sa tâche maîtresse, puisque maintenant cette tâche maîtresse possède une tâche active de moins. A partir de ce moment, la suite des événement va dépendre du statut de la tâche maîtresse.

- Si celle-ci est achevée, et à la fin de la construction maîtresse dont dépend la tâche exécutant le **select** (sinon elle est considérée active pour cette tâche), les conditions de terminaison peuvent être remplies. La tâche maîtresse est l'objet d'une primitive MULTISIGNAL, et la sous-tâche exécute une primitive WAIT à délai infini. Après son réveil, la tâche maîtresse (ré)exécutera la séquence de vérification décrite au paragraphe précédent, et tout se passera comme si la tâche maîtresse venait d'entrer dans l'état achevé, sauf qu'il y a maintenant une tâche TERMINATABLE de plus.
- Si celle-ci est dans l'état SELECTING_TERM, et que son *dependent_not_term* est maintenant nul, ce qui signifie que la sous-tâche considérée était la dernière tâche active dépendant de cette tâche maîtresse, alors le statut de la tâche maîtresse est changé en TERMINATABLE, et le même processus est répété sur la tâche maîtresse de celle-ci.
- Dans tous les autres cas, les conditions de terminaison ne peuvent être vérifiées, et la sous-tâche exécute simplement une primitive WAIT avec un délai infini.

Un point extrêmement important est qu'il faut être capable de reconstruire un état cohérent du système lorsqu'une tâche en attente sur un **select** avec alternative **terminate** est réveillée par l'appel d'un de ses points d'entrée. Si la tâche était dans l'état SELECTING_TERM, rien n'a été changé dans son environnement, et elle est juste réveillée. Si au contraire elle était dans l'état TERMINATABLE, elle redevient active, le *dependent_not_term* de sa tâche maîtresse doit être incrémenté. Si cette tâche maîtresse était elle-même dans l'état TERMINATABLE, son état est changé en SELECTING_TERM, et le *dependent_not_term* de sa propre tâche maîtresse est incrémenté, et ainsi de suite

jusqu'à trouver un maître qui n'est pas dans l'état TERMINATABLE (les règles du langage garantissant qu'il doit forcément y en avoir un, car une des constructions maîtresses indirectes est obligatoirement différente d'un corps de tâche, puisqu'une tâche n'est pas une unité de compilation [LRM 10.1(2)]).

REMARQUE:

Le mécanisme de terminaison décrit dans [ROSEN 83] est plus simple que celui décrit ici, car il était basé sur le manuel de référence de Juillet 1982. Dans ce manuel, une restriction [LRM.82 9.7.1(12)] interdisait la présence d'un **select** muni d'une alternative **terminate** dans un bloc qui était susceptible d'être une construction maîtresse pour des tâches. Dans ces conditions, lors de l'exécution d'un **select** avec alternative **terminate**, il ne pouvait y avoir de sous-tâches dont la tâche soit une tâche maîtresse indirecte, et le fait que *dependent_not_term* soit nul était une condition suffisante pour assurer que toutes les sous-tâches étaient terminées, sans qu'il soit nécessaire d'aller ré-exécuter la séquence de terminaison du maître pour compter les sous-tâches. Cette restriction a été levée dans la norme de Janvier 1983.

7.3 Avortement

La principale difficulté dans l'implémentation du mécanisme d'avortement vient de la nécessité [LRM 9.10(6)] qu'une tâche avortée alors qu'elle est engagée en tant qu'appelant dans un rendez-vous ne soit pas terminée avant la fin du rendez-vous. Un point intéressant qui est souvent négligé lorsque l'on parle de l'instruction **abort** est que la tâche avortée n'est *pas* terminée: même si elle n'est pas engagée dans un rendez-vous, elle n'est qu'achevée, ce qui signifie qu'elle doit attendre la terminaison de ses propres sous-tâches avant de se terminer elle-même. Cela peut ne pas être immédiat si certaines de ses sous-tâches sont elles mêmes engagées dans un rendez-vous, et il est possible de vérifier par programme le respect de cette règle[3].

Lorsqu'une tâche est avortée, elle commence par avorter ses sous-tâches [LRM 9.10(4)]. Elle est ensuite enlevée de toutes les chaînes sur lesquelles elle se trouve. Si la tâche est engagée dans un rendez-vous en tant que propriétaire, il faut lever l'exception TASKING_ERROR dans le client correspondant. En fait, la séquence d'instruction de l'instruction **accept** peut elle-même contenir d'autres instructions **accept**, ce qui fait que la

[3] Un nombre important des tests que nous avons proposés à l'ACV porte sur ce point.

tâche peut être engagée dans plus d'un rendez-vous simultanément. Pour cette raison, il existe une chaîne, gérée en pile, des appelants courants (*serviced*). Lors d'un avortement, il suffira de lever TASKING_ERROR dans toutes les tâches de cette chaîne.

Un test est ensuite effectué pour voir si la tâche avortée est engagée dans un rendez-vous en tant que client. Ce fait est connu d'après son *status*. On détermine d'autre part qu'il y a des sous-tâches engagées dans un rendez-vous par le fait qu'après avoir avorté les sous-tâches, le *dependent_not_term* n'est pas nul; si tel est le cas, c'est bien qu'une sous-tâche n'a pu se terminer (en effet, le *dependent_not_term* n'est *pas* décrémenté dans ce cas, car une tâche anormale est une tâche achevée, mais non terminée [LRM 9.10(5)]) . Si elle n'est pas engagée dans un rendez-vous, ni aucune de ses sous-tâches, alors la tâche peut se terminer; elle exécute la séquence normale d'actions pour une tâche qui se termine: rendre l'espace de ses sous-tâches, et décrémenter le *dependent_not_term* de son maître.

Si la tâche, ou l'une de ses sous-tâches, est engagée dans un rendez-vous, alors son état est mis à ABNORMAL, et aucune autre action n'est engagée. Toute tentative de communication avec une tâche anormale lèvera l'exception TASKING_ERROR. Une tâche anormale n'est plus CALLABLE, mais pas encore TERMINATED [LRM 9.9(2-3)].

Lorsqu'une tâche anormale est réveillée, ce qui ne peut se produire qu'à la fin du rendez-vous dans lequel elle était engagée, il est possible qu'elle puisse se terminer, mais pas obligatoirement, car elle peut elle même avoir des sous-tâches engagées dans un rendez-vous; si elle peut se terminer, il est possible que cela rende possible la terminaison d'un certain nombre de ses maîtres directs et indirects. Aussi, la tâche réveillée va-t-elle remonter la chaîne des maîtres jusqu'au plus lointain se trouvant dans l'état ABNORMAL, et le ré-avorter. Ceci aura en particulier l'effet de l'avorter elle-même, puisqu'elle n'est plus engagée dans un rendez-vous, en tant que sous-tâche du maître avorté, ainsi que tous les autres maîtres intermédiaires, à moins qu'il n'y ait quelqu'autre membre de la famille encore engagé dans un rendez-vous.

Ce mécanisme a l'avantage d'être simple à implémenter, et le problème des performances du système ne se pose pas, car le processus de ré-avortement ne sera mis en oeuvre que dans des cas d'avortement rarissimes, l'avortement devant lui-même être une instruction rare, et de toute façon peu urgente. On peut se poser la question de savoir s'il arrivera jamais à ce mécanisme d'être mis en oeuvre intégralement à part dans la suite de validation...

Remarquons enfin que l'algorithme que nous proposons ici avorte effectivement les tâches dès l'exécution de l'instruction **abort**. Une solution exigeant moins de précautions aurait été de laisser les tâches s'exécuter jusqu'au prochain point de synchronisation [LRM 9.10(6)]. Toutefois cette solution nous paraît assez peu commode pour l'utilisateur, puisque l'avortement d'une tâche prise dans une boucle infinie ne la force pas à se terminer...

8. Gestion d'horloge

Toutes les tâches suspendues par une instruction **delay**, par un appel d'entrée temporisé, ou par une instruction **select** munie d'une alternative **delay**, sont reliées entre elles par une chaîne appelée la chaîne d'horloge. L'élément du TCB utilisé pour ce chaînage est appelé *clock_chain*.

Nous supposerons pour notre modèle que nous disposons d'une interruption matérielle permettant d'effectuer une séquence d'instructions à intervalles réguliers, et que la tête de chaîne d'horloge est une variable globale (CLOCK_HEAD) du système accessible par cette séquence.

Lorsqu'une tâche est suspendue sur une attente de délai non nul, non infini, elle est chaînée sur *clock_chain*. Ceci est indiqué dans son TCB par le positionnement de *on_clock*. Le but de cet indicateur est d'éviter d'avoir à parcourir inutilement la chaîne d'horloge lorsqu'une tâche est l'objet d'une primitive SIGNAL alors qu'elle n'est pas sur la chaîne. Les tâches sur *clock_chain* sont ordonnées par ordre d'heure de réveil croissante. La structure générale de la chaîne est donnée par la figure suivante:

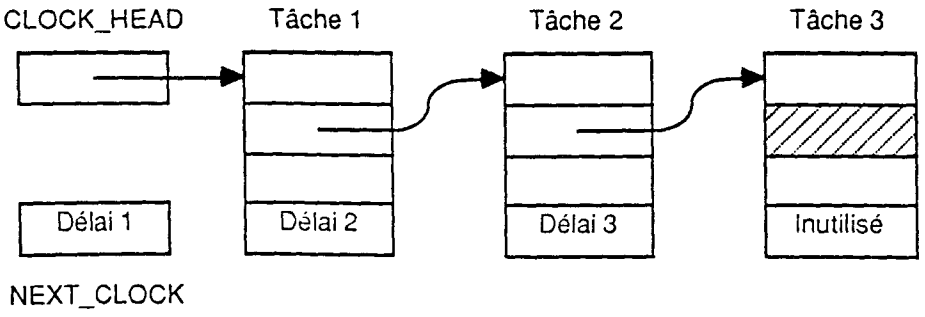


Figure 8: Structure de la chaîne d'horloge

Chaque tâche contient dans son TCB un pointeur sur la tâche suivante de la chaîne (*clock_chain*), et un compteur différentiel représentant le nombre de tops d'horloge avant de réveiller la suivante. Ainsi, le compteur de T1 représente le nombre de tops d'horloge à

attendre avant de réveiller T2 *après que T1 ait été réveillé*, etc.. Le temps restant avant de réveiller la première tâche de la chaîne est gardé dans une variable globale (NEXT_CLOCK). Le fait de garder dans le contexte d'une tâche l'heure de réveil de la prochaine tâche au lieu de la sienne propre facilite les insertions et les suppressions dans la chaîne, car cela permet d'utiliser un seul pointeur.

Cette structure permet de limiter au maximum le nombre d'opérations à effectuer par la séquence d'interruption: il suffit en effet de décrémenter une seule variable globale. Si cette variable n'est pas nulle, la séquence d'interruption est immédiatement terminée. Si elle est nulle, elle effectue une primitive SIGNAL sur la première tâche de la chaîne, ainsi que sur les suivantes tant que le *next_clock* de la tâche signalée est nul (cas de plusieurs tâches devant être réveillées à la même heure). CLOCK_HEAD et NEXT_CLOCK sont finalement regarnis à partir des valeurs de *clock_chain* et du compteur de la dernière tâche réveillée.

9. Conclusions sur le modèle proposé

L'idée de modéliser un système de gestion des tâches n'est pas originale, et pratiquement chaque conférence Ada propose une présentation de ce type.

Riccardi a décrit le modèle développé à Florida State University dans plusieurs publications [Riccardi 84,85]. On y retrouve des principes généraux assez proches des nôtres: une chaîne triée est utilisée pour relier les tâches en attente de délai; des états appelés *passive* et *asleep* correspondent d'assez près à nos états SELECTING_TERM et TERMINATABLE. Un compteur appelé AWAKE_COUNT ressemble beaucoup à notre compteur *dependent_not_term*. Toute la gestion est faite par des chaînages entre tâches, et il est difficile de se rendre compte de la validité du modèle proposé car seuls les cas "normaux" de rendez-vous et d'attente sont décrits. L'avortement est à peine effleuré, et le traitement des exceptions dans les rendez-vous n'est pas mentionné.

Leathrum [Leathrum 84] propose un modèle plus original, où il simplifie les différentes sortes d'instructions liées au rendez-vous au moyen d'équivalences très proches des nôtres, mais où en plus il tente de symétriser le rendez-vous en décrivant une syntaxe commune entre l'appel et l'acceptation d'entrée. Son modèle utilise deux queues différentes pour l'attente simple (instruction **delay**) et les clauses **delay** d'instructions **select**, mais une seule queue pour toutes les tâches en appel d'entrée. Ce dernier choix nous paraît dangereux, car il implique que toute tâche acceptant une entrée doit explorer cette liste pour chercher un éventuel client en attente.

Ogor enfin [Ogor 85] propose un modèle à but pédagogique assez complet, mais qui nous a paru contenir un nombre assez important de chaînages et d'états, et relativement plus complexe que celui que nous proposons. Seule une version simplifiée de ce modèle a été réalisée.

Tous les modèles de gestion des tâches ont en fait de grandes ressemblances, ce qui semblerait prouver qu'il n'y a pas beaucoup de solutions différentes pour implémenter les primitives de base du parallélisme d'Ada.

Néanmoins, nous pensons que le modèle que nous avons présenté peut se comparer favorablement avec ceux qui ont été publiés, et qu'il serait intéressant de le reprendre comme base d'un noyau d'exécutif temps réel pour plusieurs raisons:

1- Généralement les modèles proposés sont purement théoriques, beaucoup d'entre eux n'ayant même jamais été compilés. Notre modèle a passé avec succès tous les tests de la version 1.7 de l'ACVC.
2- A l'heure actuelle, les compilateurs existants concernent tous des machines cibles tournant sous un système d'exploitation. Un modèle tel celui décrit ici est nécessaire pour les applications nécessitant un éxécutif temps-réel autonome, afin de permettre d'utiliser Ada avec des machines cibles nues (matériel embarqué, centraux téléphoniques...).
3- Le système actuel peut permettre le développement d'un modèle pour machines multi-processeurs. C'est ce point que nous allons développer maintenant.

Les tâches, dans notre modèle, interagissent de trois façons: par l'intermédiaire de la primitive SIGNAL qui permet d'envoyer une information élémentaire dans le contexte d'une autre tâche, par inspection directe du contexte de la tâche appelante pour interroger son état, enfin lors des opérations de chaînages qui nécessitent une intervention directe sur le contexte des tâches.

La primitive SIGNAL est typiquement un envoi de message: elle se prête donc bien à une implémentation sur système distribué. Les interrogations d'état ne posent pas de problème de conflit d'accès, puisqu'il s'agit uniquement de lectures. Toutefois, il existe des périodes critiques entre le moment où un état est interrogé et le moment où les actions correspondantes sont prises: par exemple, une tâche peut trouver que le propriétaire d'une **entry** est en attente, décider de l'appeler, mais pendant ce temps là un délai du propriétaire a expiré et l'entrée n'est plus ouverte.

Ce problème est très général, et a été discuté par [Volz 85]. Egalement, le problème de la constitution des chaînages dans un système distribué risque de poser des difficultés d'implémentation. Notre modèle est cependant assez favorable dans la mesure où il propose relativement peu d'opérations de chaînages, et où une partie des interactions a lieu par échange de messages.

Il y a enfin une dernière classe de machines, intermédiaire entre les mono-processeurs et les systèmes distribués: les machines multi-processeurs à mémoire commune. Ici, les chaînages et les interrogations peuvent être effectués au moyen de simples verrouillages de tables. Notre modèle de LOCK/UNLOCK permet de repérer précisément les séquences où un tel verrouillage serait nécessaire. Un cas particulier de ce type de machine est l'Ultracomputer développé à New York University [Gottlieb 83], sur lequel une implémentation d'Ada est en cours [Schonberg 85]. Notre modèle est utilisé pour réaliser la gestion des tâches, et bien que rien n'ait encore été publié sur le sujet, il semblerait que sa structure se soit révélée bien adaptée à l'architecture de l'Ultracomputer.

Chapitre V.
Implémentation du système de gestion des tâches

Ce chapitre va maintenant décrire comment a été réalisé en pratique le modèle décrit au chapitre précédent, ainsi que les structures de plus bas niveau, comme le mécanisme d'échanges d'informations entre tâches. Les instructions de la Machine Ada spécifiques à la gestion des tâches sont également documentées.

Le système de gestion de tâches peut être divisé logiquement en deux couches: le *noyau* qui se charge de la gestion effective des tâches, comme l'ordonnancement, les échanges de signaux, et qui est la simple traduction du modèle décrit ci-dessus, et l'*interface* qui fournit l'accès au noyau pour l'Ada-machine, ainsi que des services particuliers à un mécanisme de compilation mais qui n'interviennent pas dans le modèle de gestion de tâches, comme le passage de paramètres entre tâches lors d'un rendez-vous. Cette distinction n'est cependant utile que pour clarifier l'exposé, et pour le cas où l'on souhaiterait reprendre le modèle décrit ici sur une machine réelle; dans notre implémentation, les deux aspects sont souvent imbriqués. En effet, dans une machine interprétée, il est important de diminuer le nombre de passages dans la boucle principale de l'interprète pour améliorer l'efficacité de la machine. Ceci signifie qu'il est préférable de générer un code compact, utilisant un grand nombre d'instructions de haut niveau sémantique[1]. Nous avons donc été amenés à définir des instructions qui correspondraient, dans une implémentation sur machine nue, à une suite d'instructions terminées par un appel système.

[1]La seule limitation au nombre d'instructions est théoriquement la taille réservée au code d'opération, 8 bits dans notre cas, autorisant en principe 256 instructions différentes. Pour des raisons d'optimisation d'adressage dans le cas de l'IBM/PC, nous avons été amenés à rester un peu en dessous des 256 instructions théoriques (cf. [KRUCHTEN 86]).

1. Adaptation du modèle à une structure d'interprète.

Le premier modèle du système [Rosen 83] avait été conçu dans l'optique d'une implémentation sous forme d'un noyau pour une machine nue, et plus particulièrement pour un micro-ordinateur à espace mémoire restreint. La Machine Ada a des contraintes différentes: l'espace mémoire est virtuellement infini[2], par contre la structure interprétative de la machine est beaucoup plus contraignante quant aux permutations de contexte. En particulier, sur une "vraie" machine, une permutation de contexte rétablit *tout* le vecteur d'état de la tâche. Dans le cas d'une machine interprétée, le vecteur d'état contient des éléments qui ne sont pas contrôlables, en particulier l'état de l'interprète lui-même: par exemple, l'état d'imbrication des procédures de l'interprète ne peut être reconstitué au moment d'un changement de contexte.

Or le changement de contexte se produit au moment où une tâche se met en attente, c'est à dire lorsqu'elle exécute une primitive WAIT. Dans notre modèle, des actions spécifiques doivent être effectuées lors du réveil des tâches. Ces actions sont appelées *actions de réveil* (ou *After Wait Actions*), et sont entièrement déterminées par le statut de la tâche (raison pour laquelle elle a été mise en sommeil) et par son événement (raison pour laquelle elle a été réveillée). Elles peuvent donc être effectuées par l'ordonnanceur au moment où la tâche est réveillée, avant de la ramener au niveau de la boucle principale de l'interprète. Le déroulement d'une mise en sommeil, puis du réveil, se passe alors d'après le schéma suivant:

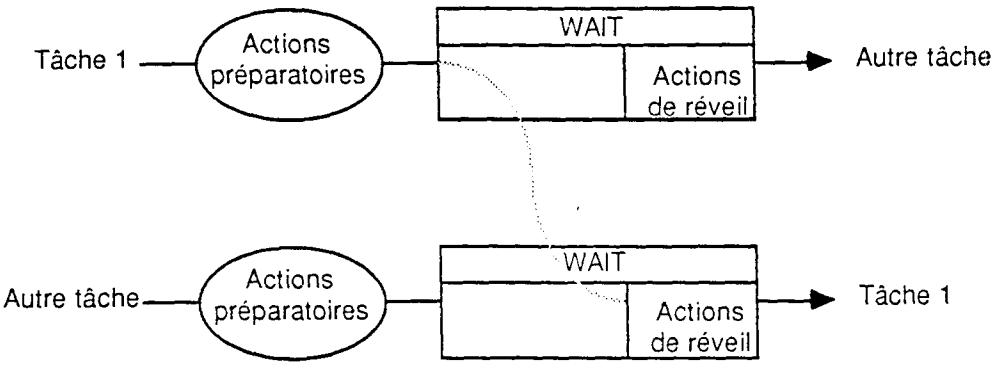


Figure 1: Actions associées à une mise en attente

Du point de vue de l'organisation du système, ce schéma est très différent de celui mis en oeuvre dans notre modèle: en effet, l'ordonnanceur n'avait alors aucune connaissance

[2]Tout au moins en ce qui concerne le prototype SETL sur Vax avec mémoire virtuelle. Les problèmes de mémoire devront être reconsidérés lors d'une implémentation sur machine de type "PC".

de l'état de la tâche réveillée, alors qu'ici c'est lui qui exécute les actions de réveil pour la tâche qu'il va ordonnancer. Une conséquence de cela est que dans l'écriture de l'interprète, un appel à la primitive WAIT doit obligatoirement être la dernière action d'une procédure, et ceci de façon transitive pour toute procédure appelant une autre procédure qui utilise WAIT, puisqu'après un appel à WAIT, c'est une autre tâche qui s'exécute. Après toute mise en attente, il est donc obligatoire de redescendre immédiatement l'interprète au niveau de la boucle principale[3].

2. Structures et mécanismes de base

2.1. Le bloc de contrôle de tâche (TCB)

Chaque tâche possède son propre segment de pile (cf. [KRUCHTEN86]). Le *bloc de contrôle de tâche* (TCB) constitue le bas de cette pile.Toute l'information relative à une tâche étant ainsi comprise dans son segment de pile, une tâche est entièrement caractérisée par son numéro de segment de pile. La *valeur* d'un objet tâche est donc simplement son indice de segment de pile.

[3]Nous verrons qu'il y a une exception à cette règle dans le cas d'**abort**, mais les instructions se trouvant après le WAIT peuvent être exécutées par n'importe quelle tâche, et un éventuel changement de contexte n'est donc pas important.

Le bloc de contrôle de tâche contient toute l'information nécessaire à la gestion des tâches. Ses éléments sont résumés dans la figure suivante:

Nom	Ref. dans l'interprète	Taille	**Commentaire:**
TTB TTO	c_templ_table_base c_templ_table_off	W W	*Base et déplacement* *de la table de correspondance*
NRE	c_nr_entries	W	*Longueur de la queue_table*
STS EVT	c_status c_cur_event	B W	*Statut de la tâche* *Evénement courant*
ONC CCH CCO	c_on_clock c_clock_chain c_clock_count	B W W	*Drapeau = 1 si en attente d'horloge* *Chaînage d'horloge* *Heure de réveil tâche suivante*
DYP	c_dyn_priority	B	*Priorité dynamique*
LKL	c_lock_level	B	*Niveau de verrouillage*
DNT	c_dependent_not_term	W	*Nb de tâches dépendantes non terminables*
MWC	c_multiwait_count	W	*Nombre d'événemens attendus*
NXT PRT	c_next_task' c_previous_task	W W	*Tâche suivante sur la même entrée* *Tâche précédente sur la même entrée*
MYF MYM MBF BLK	c_my_father c_my_master c_master_bfp c_brother_link	W W W W	*Pointeur sur le parent* *Pointeur sur tâche maîtresse* *BFP de la construction maîtresse* *Pointeur sur tâche de même maître*
CLT CLE CLP	c_called_task c_called_entry c_called_priority	W W B	*Tâche possédant l'entrée appelée* *N° absolu de l'entrée appelée* *Priorité de la tâche appelée*
	c_queue_base ...	T	*Début de la queue_table*

B = Octet , W = Mot, T = Table de (W,W)

Figure 2: Définition du bloc de contrôle de tâche

V.4 Implémentation du système de gestion des tâches

La table suivante donne la signification précise de ses éléments:

nr_entries:
Le nombre d'entrées possédées par la tâche, chaque membre d'une famille comptant pour 1. C'est également la longueur de la *queue_table*.

status:
Le statut courant de la tâche.

cur_event:
L'événement courant, plus récent événement qui ait réveillé la tâche. Initialisé à NO_EVENT à la création de la tâche.

on_clock:
Drapeau booléen, vrai si la tâche est chaînée sur la queue d'horloge, c'est à dire en train d'exécuter un délai dont la valeur n'est ni 0, ni ENDLESS.

clock_chain:
Lien pour la queue d'horloge.

clock_count:
Heure absolue du réveil de la prochaîne tâche sur la chaîne (c'est à dire de la tâche désignée par *clock_chain*). Non significatif si *clock_chain* est le pointeur nul.

dyn_priority:
Priorité courante (dynamique) de la tâche (cette priorité peut changer durant un rendez-vous).

lock_level:
Niveau de verrouillage, c'est à dire le niveau d'imbrication courant de primitives LOCK/UNLOCK.

multiwait_count:
Lors d'attente sur événements multiples, contient le nombre de primitives MULTI_SIGNAL dont la tâche doit être l'objet avant d'être réveillée.

next_task,previous_task:
Chaînage double entre tâches en attente sur la même entrée (de la même tâche).

my_father:
pointeur sur la tâche parent.

my_master:
pointeur sur la tâche maîtresse.

master_bfp:
pointeur de bloc de trame du maître, utilisé pour identifier la construction maîtresse dont dépend la tâche.

brother_link:
Chaînage sur la tâche suivante dépendant du même maître (même tâche maîtresse et même construction maîtresse).

called_task,called_entry:
durant un appel d'entrée, pointeur vers la tâche propriétaire de l'entrée, et indice absolu de l'entrée.

called_priority:
durant un rendez-vous, valeur préservée la valeur précédente de la priorité de la tâche appelée.

serviced_tasks:
durant un rendez-vous, pointeur vers la tâche appelante. Il s'agit en fait d'une chaîne gérée en pile, puisque les acceptations d'entrées peuvent être imbriquées.

queue_base:
Base de la *queue_table*, table contenant pour chaque entrée la tête de chaîne des tâches appelant l'entrée.

2.2. Chaînages

Un pointeur de tâche pour un chaînage consiste simplement en un numéro de tâche. Les pointeurs sont donc en fait de simples entiers positifs.

Par convention, la valeur 0 est utilisée comme pointeur nul, et la valeur -1 pour signaler une entrée ouverte dans la table des entrées d'une tâche.

2.3 Interruptions et mécanisme de verrouillage

En dépit du haut niveau de l'interprète, des "interruptions" peuvent se produire, dans le sens que des événements extérieurs peuvent forcer un changement de contexte. Ceci peut se produire à la fin d'un délai d'une tâche dont la priorité d'exécution est supérieure à celle de la tâche qui possède alors l'UC, ou lorsqu'une tâche exécute une primitive SIGNAL dont l'objet est une tâche de plus haute priorité.

La façon dont travaille l'interprète le rend fondamentalement ininterruptible durant l'exécution d'une instruction. De plus, les instructions liées à la gestion des tâches constituent chacune une fonctionnalité complète, ce qui fait qu'aucune séquence critique ne s'étend sur plusieurs instructions. Il n'y a donc pas lieu de prévoir de mécanisme de masquage d'interruptions. Il y a cependant besoin un mécanisme pour retarder le

changement de contexte effectif jusqu'à un endroit autorisé lorsqu'une tâche effectue une primitive SIGNAL sur une tâche de priorité supérieure, car ceci pourrait provoquer un changement de contexte immédiat qui ne serait pas autorisé, puisqu'il reste des actions à effectuer. D'autre part, il est intéressant de marquer les séquences qui seraient critiques dans une implémentation de plus bas niveau pour permettre d'utiliser le modèle sur une machine réelle par la suite.

Ces deux buts sont obtenus simultanément grâce au modèle suivant: chaque niveau de priorité peut être vu comme un *niveau d'interruption*. Chaque tâche est attachée à son niveau d'interruption dans le sens que lorsqu'une interruption sur le niveau correspondant a lieu, la première tâche attachée à ce niveau est activée. La primitive SIGNAL se comporte comme une interruption logicielle sur le niveau de la tâche qui en est l'objet. Deux primitives appelées LOCK et UNLOCK sont utilisées pour inhiber les interruptions. Toute procédure faisant appel à SIGNAL doit exécuter auparavant un appel à LOCK, pour retarder l'éventuel changement de contexte jusqu'au moment où celui-ci est autorisé - au UNLOCK correspondant. Certaines procédures peuvent cependant être appelées à partir de différents points du sytème, et il est souhaitable de garder une indépendance entre les procédures; si elles sont appelées à partir d'un autre module qui a lui même effectué un appel à LOCK, l'interruption doit être retardée jusqu'à l'UNLOCK de l'appelant. Pour cette raison, les masquages d'interruption sont empilé, c'est à dire que toute tâche possède dans son TCB un compteur incrémenté lors de chaque appel à LOCK et décrémenté lors de UNLOCK. Un changement de contexte n'a effectivement lieu que lorsque UNLOCK fait repasser ce compteur à 0. Ainsi, chaque procédure peut protéger son code indépendemment de son contexte appelant: de toute façon, aucun changement de contexte ne peut se produire avant la sortie de la paire de LOCK/UNLOCK la plus extérieure.

Une variable interne à l'interprète, appelée HIGHEST_PRIORITY, contient en permanence la plus haute des valeurs des priorités de la tâche qui s'exécute et de celles qui ont été l'objet de primitives SIGNAL. Cette variable peut être vue comme le plus haut niveau d'interruption actif. Lorsqu'UNLOCK a besoin d'effectuer un changement de contexte, c'est à dire lorsque le compteur est passé à 0 et que HIGHEST_PRIORITY est supérieur à la priorité courante, il exécute simplement un appel à WAIT(0). UNLOCK doit donc, comme WAIT, toujours être la dernière instruction d'une procédure.

La primitive WAIT est elle-même protégée par une paire LOCK/UNLOCK. Comme WAIT est le seul endroit où un changement de contexte puisse se produire, les tâches sont

créées avec leur compteur de verrouillage initialisé à 1, car elles seront activées au milieu du WAIT, sans avoir exécuté le LOCK correspondant.

3. Création et activation de tâches

3.1 Environnements de tâches

Comme expliqué ci-dessus, un *environnement de tâche* est une famille de tâches devant être activées ensemble. Les environnements de tâches constituent une pile. Il n'aurait pas cependant été commode de créer une pile effective uniquement dans ce but; aussi l'avons nous réalisée au moyen de listes chaînées gérées en "premier entré, premier sorti". Un environnement de tâche est un élément alloué dans le tas, constitué d'une chaîne de tâches à activer en parallèle, et d'un pointeur vers l'environnement de tâche précédent. Ce pointeur est nul si l'environnement de tâche est le premier de l'environnement de bloc courant. L'environnement de tâche courant est pointé par l'élément *bf_tasks_declared* de l'environnement de bloc.

Lorsque des tâches sont créées, elles sont chaînées dans l'environnement de tâche courant. L'instruction *activate* activera les tâches chaînées sur l'environnement de tâche courant, puis enlèvera cet environnement de tâche. Les environnements de tâche se comportent donc bien comme une pile dont les élément sont des ensembles de tâches à activer en parallèle. Remarquer cependant que les tâches créées dans la partie spécification d'un paquetage appartiennent logiquement à la même famille que celle créées durant l'élaboration de la partie déclarative du corps de paquetage correspondant (comme les tâches PT1 et PT2 de l'exemple du paragraphe 2.2.1 du chapitre précédent). Par conséquent, la chaîne de tâches courante doit être préservée à la fin de la spécification de paquetage, et restaurée à l'entrée du corps correspondant. Ce travail est accompli au moyen de deux instructions spécialisées: *pop_tasks_declared* dépile un environnement de tâche en préservant la chaîne dans une variable (locale), et *link_tasks_declared* crée un nouvel environnement de tâche, initialisé par une valeur trouvée sur la pile. Cette instruction est également utilisée pour créer de nouveaux environnements de tâche, auquel cas la valeur mise sur la pile est 0 (= fin de chaîne). Sinon, dans le cas du corps de paquetage, la valeur préservée de la tête de chaîne est tout d'abord mise sur la pile, ce qui recrée un environnement de tâche contenant toute les tâches créées lors de l'élaboration de la spécification correspondante.

Par convention, une valeur nulle de *bf_tasks_declared* est équivalente à un pointeur sur un environnement de tâche vide: ceci permet de retarder la création de l'environnement de tâche jusqu'au moment où une tâche est effectivement créée, et en particulier d'éviter la création d'un environnement de tâche vide dans le cas où aucune tâche n'est créée.

Le fait que l'environnement de tâche soit réalisé au moyen d'une pile en mémoire dynamique ne doit pas inquiéter: il s'agit d'une solution adaptée à la Machine Ada, où de toutes façons *toutes* les variables sont allouées en mémoire dynamique. Le surcoût est donc négligeable. Dans une machine avec un mécanisme d'adressage plus conventionnel, il serait certainement préférable d'avoir une pile fixe pour la gestion des environnements de tâches.

3.2 Elaboration des tâches

Bien que le manuel de référence distingue des tâches simples et des types tâche, nous considérerons ici que nous n'avons que des déclarations de types tâches, éventuellement suivies de déclarations d'objets tâches appartenant à ce type. En effet, la partie avant du compilateur transforme la déclaration de tâches simples en déclaration d'un type tâche anonyme, suivie de la déclaration de l'objet correspondant, comme indiqué dans [LRM 9.1(2)].

La compilation de la spécification de tâche crée un type tâche; elle génère donc un patron de type qui sera élaboré par le mécanisme normal d'élaboration de type. Un patron de type pour un type tâche comprend, à part les champs communs *kind* et *size*, la priorité de la tâche, les valeurs de base et de déplacement du descripteur du corps de tâche, le nombre total d'entrées de la tâche, et le nombre de familles. Ceci est suivi par une table de correspondance, permettant de calculer la valeur absolue d'un index d'entrée à partir d'une paire [famille,membre]. Cette table contient, pour chaque famille, le nombre de membres de la famille, et la valeur dans la table des entrées de l'indice correspondant au premier membre de la famille. Les familles étant numérotées à partir de 1 et les membres à partir de 0, un indice absolu d'entrée se calcule simplement comme:

```
indice_absolu := table_de correspondance (2xfamille-1) + membre + 1;
```

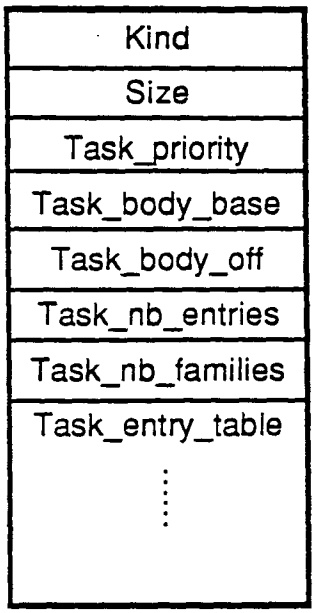


Figure 3: Patron de type tâche

L'élaboration du type consiste à calculer cette table. Noter que ceci ne peut être fait au moment de la compilation, car les sous-types des familles peuvent être dynamiques, et par conséquent le nombre de membres d'une famille également.

Un corps de tâche est compilé de la même façon qu'une procédure (cf. [KRUCHTEN 86]) et génère par conséquent le même descripteur qu'une procédure. Comme pour une procédure, l'élaboration du corps de tâche permet de créer l'ensemble relais associé.

L'élaboration d'une déclaration d'objet tâche crée un objet tâche. Un objet tâche est entièrement défini par son segment de pile. Par conséquent, la création d'un objet tâche consiste à créer un segment de pile initialisé avec le TCB et le contexte adéquats, puis à mettre sur le sommet de la pile du parent le numéro de ce segment de pile.

Les seules parties susceptibles de varier d'un type tâche à l'autre dans le TCB initial sont le numéro de segment de code, la priorité, le nombre d'entrées, et la table d'entrées initiale. Ces éléments sont pris dans le patron de type. Remarquer que l'adresse de départ dans le segment de code est la même pour toutes les tâches. Les champs *my_master* et *master_bfp* du TCB de la tâche sont laissés non initialisés (toujours à 0) à ce moment. Ceci sert à reconnaitre le cas particulier de la tâche qui se fait avorter avant son activation. Comme elle n'a pas encore de maître, les actions sont en effet légèrement différentes dans ce cas.

La tâche nouvellement créée est alors chaînée sur l'environnement de tâche courant. Un environnement de tâche est créé si cela n'avait pas été fait auparavant.

3.3 Bloc de corps de tâche

Afin de permettre la transmission des signaux nécessaires à l'activation d'une tâche, le corps de tâche est modifié par la phase d'expansion du générateur de code. La structure:

```
task body T is
    <déclarations>
begin
    <instructions>
exception
    <traite_exceptions>
end T;
```

est transformée par la phase d'expansion du générateur de code en:

```
task body T is
begin
    declare
        <déclarations>
    begin
        SIGNAL(NO_EVENT);
        <instructions>
    exception
        <traite_exceptions>
    end;
exception
    when others => SIGNAL(TASKERR_EVENT);
end T;
```

Ceci permet de générer automatiquement l'échange de signaux décrits au chapitre précédent.

3.4 Activation de tâches

La chaîne des maîtres relie les tâches dépendant du même maître (même tâche maitresse et même construction maîtresse). La tête de chaîne est un élément dans l'environnement de bloc du maître, appelé SUBTASKS. Les tâche sont reliées par *brother_link*. Avant activation de la tâche, cette chaîne est utilisée pour relier les tâche de même parent. Dans ce cas, la tête est un autre élément de l'environnement de bloc du parent, appelé TASKS_DECLARED.

La procédure ACTIVATE est utilisée pour activer simultanément un ensemble de tâches, liées à partir de l'environnement de tâche courant. Toutes ces tâches dépendent de la même tâche maîtresse et de la même construction maîtresse. Le maître (au sens du LRM) est identifié par le numéro de la tâche maîtresse, et le pointeur d'environnement de bloc correspondant au niveau de la pile lorsque la tâche maîtresse a élaboré la construction maîtresse. Remarquer que comme une construction ne peut être abandonnée tant qu'il reste des tâches actives qui en dépendent, ceci est suffisant pour identifier de façon unique la construction maîtresse. Finalement, ACTIVATE supprime l'environnement de tâche.

4. Implémentation du rendez-vous

Comme indiqué dans la première partie de ce chapitre, un certains nombre de transformations doivent être effectuées pour adapter les différentes instructions d'appel ou d'acceptation de rendez-vous aux deux seules instructions supportées par le modèle, qui sont l'attente sélective avec alternative **delay**, et l'appel d'entrée temporisé.

Certaines de ces transformations sont effectuées par le compilateur (comme de transformer les parties **else** en **delay** 0), alors que d'autres sont internes au système de gestion des tâches (comme celles mettant en jeu un délai ENDLESS). Dans ce dernier cas, il existe des instructions machines spécifiques, dont l'interprétation appelle les routines standard du système de gestion de tâches avec des valeurs particulières des paramètres. Par exemple, il existe une instruction *entry_call* et une instruction *timed_entry_call*, qui toutes les deux appelleront les mêmes routines avec une valeur de délai qui sera ENDLESS pour *entry_call*, et la valeur effective pour *timed_entry_call*. Ceci augmente le nombre d'instructions de la machine, mais diminue la taille du code généré dans les cas les plus fréquents.

4.1 L'interface des rendez-vous

L'interface permettant l'échange de paramètres entre l'appelant et le propriétaire lors d'un rendez-vous est très proche de celui utilisé pour les appels de procédure, et nous renvoyons le lecteur à [Kruchten 86] pour une description plus détaillée de ce mécanisme. Nous n'étudierons ici en détail que les points spécifiquement liés à la gestion des tâches.

4.1.1 Interface côté appelant

Mis à part les paramètres, un appel d'entrée nécessite l'évaluation de l'objet tâche auquel appartient l'entrée et l'entrée elle-même. L'ordre de ces évaluations est spécifié par le LRM, avant l'évaluation des paramètres. Pour effectuer un appel d'entrée, l'appelant va donc devoir empiler la valeur de l'objet tâche appelé, les valeurs de famille et de membre de l'entrée appelée, puis évaluer les paramètres comme pour un appel de sous-programme normal. Si l'appel d'entrée est un appel temporisé ou conditionnel, la valeur du délai est également empilée (0 pour un appel conditionnel), mais cette valeur est immédiatement dépilée par l'instruction machine *timed_entry_call*. Au moment ou l'appel est accepté, la pile aura donc la structure suivante:

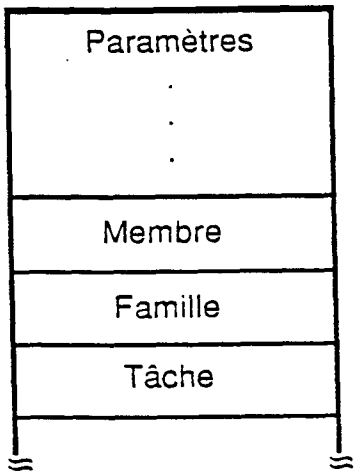


Figure 4: Etat de la pile à l'acceptation d'un appel d'entrée

Lorsque l'appel d'entrée est accepté, les paramètres sont recopiés sur le sommet de la pile du propriétaire. Mais contrairement à un appel de procédure, un appel d'entrée ne crée pas de nouvel environnement de pile. Les paramètres sont gérés comme des variables locales, appartenant à la tâche englobante. Par conséquent, les paramètres ne se trouvent pas au bon endroit sur la pile, puisqu'ils se trouvent au sommet alors que les variables se trouvent sous l'environnement de bloc [Kruchten 86]. Le propriétaire exécutera le nombre requis d'instructions *update_and_discard* pour recopier les paramètres dans les variables locales associées[4].

A la fin du rendez-vous, le propriétaire n'a aucune action spéciale à effectuer, puisque les paramètres ne sont que des adresses, qui ne sont pas changées par le rendez-vous. L'appelant, lors de son réveil, descendra le bloc de paramètres de trois positions dans la

[4] Ce mécanisme est moins coûteux qu'il n'y paraît, car les paramètres comme les variables locales ne sont en fait que l'adresse des variables effectives (cf. [Kruchten 86]).

pile, pour se débarasser des valeurs du membre, de la famille, et de la tâche, qui ne sont plus nécessaires. La pile aura alors l'allure suivante:

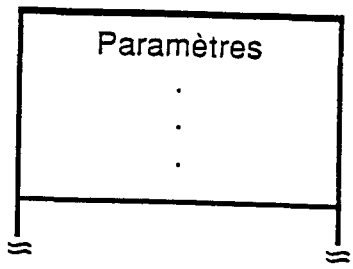


Figure 5: Etat de la pile au retour d'un appel d'entrée

C'est exactement la même structure qu'au retour d'un appel de sous-programme, et le postlude habituel peut être utilisé. Une solution alternative à ce rabaissement de la pile aurait été de générer trois instructions *discard* après le postlude; mais il est apparu plus efficace de se débarasser de ces valeurs directement dans l'interprète plutôt qu'au moyen d'instructions machines.

Si l'appel était temporisé (ou conditionnel), une autre information est nécessaire au retour pour reconnaître si la demande de rendez-vous a pu être effectuée ou non afin de sélectionner la branche correcte du **select**. Si l'appel a été accepté, un 1 est empilé au dessus du bloc de paramètres; sinon, le bloc de paramètre entier est supprimé, car il n'y a pas de postlude dans ce cas, et un 0 est empilé. Les deux configurations possibles de la pile sont alors:

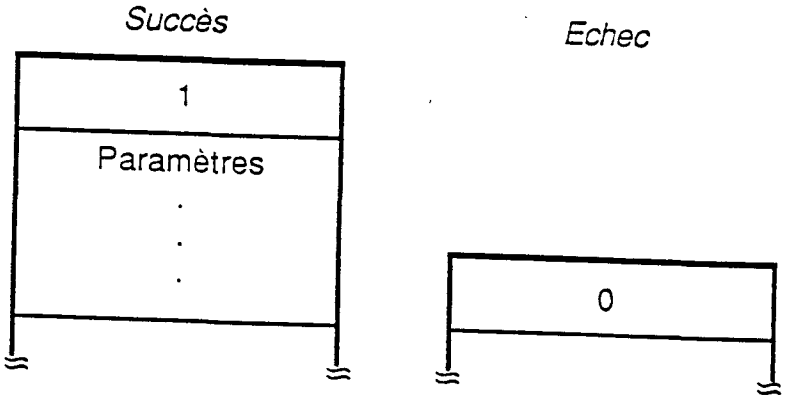


Figure 6: Etat de la pile au retour d'un appel d'entrée temporisé

4.1.2 Interface côté propriétaire

Dans le cas général, le propriétaire exécutera une attente sélective munie soit d'une alternative **teminate**, soit d'une alternative **delay** (cf. équivalences). Chaque alternative peut être gardée, et ces gardes sont dynamiques. La seule chose qui soit connue statiquement est le nombre d'alternatives de l'instruction **select.**

L'exécution d'une instruction **select** est divisée en quatre phases:

1- Choix d'une alternative (acceptation d'entrée, plus court délai, ou alternative **terminate**).
2- Exécution du corps du rendez-vous, ou de la partie **else**, ou du **terminate.**
3- Si rendez-vous: Fin du rendez-vous (réveil de l'appelant).
4- Si rendez-vous: Exécution de la séquence d'instructions optionnelle.

La première phase est exécutée par l'instruction machine *selective_wait.* Cette instruction a un argument qui est le nombre d'alternatives. Un nombre égal de *descripteurs* est empilé. Il y a trois sortes de descripteurs, correspondant aux alternatives **accept**, **delay**, et **terminate**, dont les formats sont donnés par la figure ci-dessous:

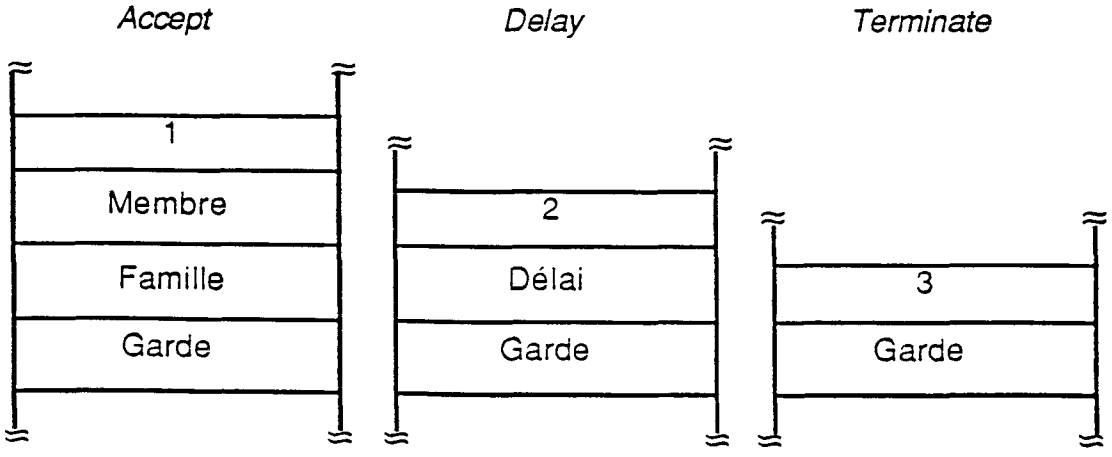


Figure 7: Descripteurs d'alternatives d'instruction **select**

Dans le cas d'une alternative **accept** fermée, les valeurs de famille et de membre sont toutes les deux à 0, car [LRM 9.7.1(15)] indique que la famille et le membre ne sont pas évalués. La même chose s'applique au délai d'une alternative **delay** fermée. Il est cependant nécessaire d'empiler un descripteur quand-même, car l'identification de la branche choisie se fait par son numéro d'ordre dans l'instruction **select**, et le nombre de descripteurs doit être respecté.

Ces descripteurs sont dépilés par l'instruction machine *selective_wait*. Celle-ci va choisir une branche, éventuellement en mettant la tâche en attente. Au retour du *selective_wait*, le sommet de pile contient le numéro de l'alternative sélectionnée, l'identité de l'appelant, et la copie des paramètres. Les états de la pile avant et après l'instruction *selective_wait* se présentent ainsi:

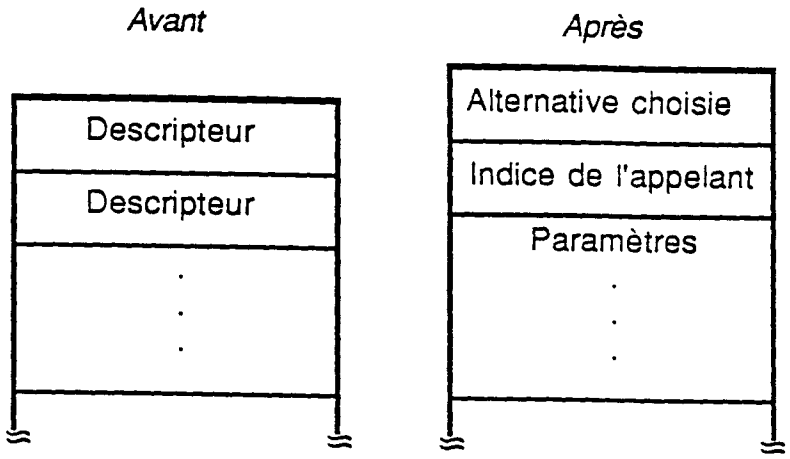


Figure 8: Etat de la pile avant et après une attente sélective

Remarquer que si aucun appel d'entrée acceptable n'a été effectué au moment où le *selective_wait* est exécuté, la tâche sera suspendue, puis réveillée par une primitive SIGNAL donnant la valeur de l'indice absolu de l'entrée appelée. Mais ce qui est nécessaire au *selective_wait* est la position de l'alternative dans l'instruction **select**. Par conséquent, le *selective_wait* doit garder une table de correspondance des indices absolus d'entrée vers le numéro d'alternative correspondante. Cette correspondance est gardée sous la forme de paires [Indice de l'alternative, numéro d'entrée], empilées durant la suspension, sous la forme suivante:

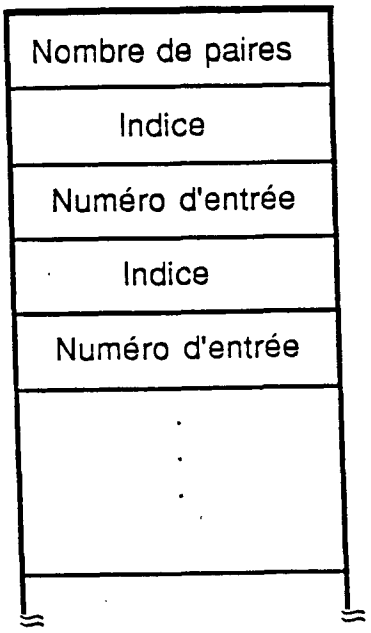


Figure 8: Etat intermédiaire de la pile durant une attente sélective

Remarquer que cet état de la pile n'existe que pendant la suspension de la tâche, et ne peut être vu lorsque la tâche est active.

Après l'attente sélective, une instruction **case** normale est utilisée pour choisir le traitement correspondant à la branche sélectionnée. La séquence d'instructions de chacune des branches de ce **case** comporte, pour les branches correspondant à des **accept**:

- Les instructions correspondant au corps du rendez-vous (phase 2);
- Une instruction *end_rendezvous* (phase 3);
- Les instructions optionnelles (phase 4).

Une branche correspondant à une alternative **else** comporte la suite d'instructions correspondante; une branche correspondant à une alternative **terminate** comporte juste une instruction machine *terminate* qui provoquera la terminaison effective de la tâche.

Bien qu'il soit possible de l'implémenter comme une instruction **select** normale, il nous a paru utile de fournir une interface plus simple pour le cas de l'**accept** simple. Dans ce cas, la même instruction *selective_wait* est utilisée, mais le nombre d'alternatives est mis à 0. La pile ne contient alors qu'une paire [famille,membre] décrivant l'entrée acceptée. A la suite de cette instruction, seul l'index de l'appelant est empilé. Les états de la pile dans ce cas se présentent ainsi:

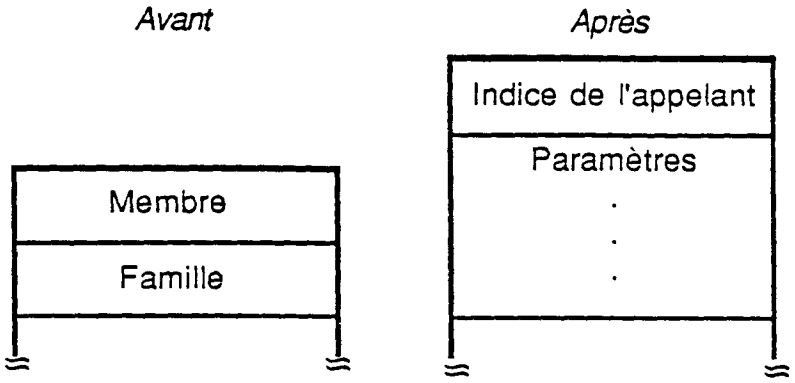


Figure 9: Etat de la pile avant et après un **accept** simple

4.2 Exceptions en cours de rendez-vous

Lorsqu'une exception se produit en cours de rendez-vous, elle doit être propagée à l'appelant [LRM 11.5(4)]. Une instruction spéciale, *raise_in_caller*, est utilisée pour propager l'exception courante à l'appelant, qui est toujours la tâche en tête de la chaîne *serviced_tasks*. Pour permettre ceci, la phase d'expension du générateur de code transforme la séquence:

```
accept E do
    <instructions>
end E;
```

en:

```
accept E do
    begin
        <instructions>
    exception
    when others => RAISE_IN_CALLER;
                   raise;
    end;
end E;
```

5. Implémentation des mécanismes de terminaison

5.1 Fin de construction maîtresse

Lorsqu'une tâche quitte une construction maîtresse, elle exécute une instruction *leave_block*. Cette instruction commence par vérifier si l'élément *subtasks* de l'environnement de bloc, qui est la tête de chaîne des tâches dépendant de cette construction, est nul. Si c'est le cas, la construction est quittée. Sinon, elle vérifie comme indiqué précédemment s'il existe des tâches dépendantes actives. Si c'est le cas, elle se met en attente après avoir décrémenté son compteur ordinal, de façon à réexécuter l'instruction *leave_block* lorsqu'elle sera réveillée. Ceci permet d'éviter une boucle explicite dans le code.

5.2 Terminaison

La terminaison effective de la tâche est provoquée par l'instruction *terminate.* Celle ci admet un argument qui permet de spécifier de quel type de terminaison il s'agit: fin normale du programme principal, ou plus exactement fin de la tâche qui a appelé le programme principal, fin de cette même tâche pour cause d'exception non traitée, fin normale d'une autre tâche, sélection d'alternative **terminate**... Ce renseignement sert à arrêter la Machine Ada en cas de fin du programme et à retourner au système d'exploitation, éventuellement avec un message d'erreur s'il y a eu exception non traitée, ou pour des aides à la mise au point des programmes utilisateur. Il n'intervient en aucune façon dans le mécanisme de terminaison lui-même, qui est unique, et implémenté conformément au modèle décrit ci-dessus.

5.3 Avortement

Deux procédures participent au mécanisme d'avortement: ABORT et KILL. ABORT implémente directement l'instruction **abort**, alors que KILL est chargée d'effectuer l'avortement effectif d'une tâche. La séparation en deux procédures a été rendue nécessaire par le mécanisme récursif mis en jeu par l'avortement. ABORT représente la partie non récursive et KILL la partie récursive. Lors de la génération de code de l'instruction **abort**, les identités des tâches sont évaluées dans l'ordre fourni par l'utilisateur et empilées. Un argument de l'instruction *abort* donne le nombre de tâches à avorter. La procédure ABORT dépilera les tâches les unes après les autres pour les avorter. Remarquer qu'à cause de l'interface de type pile, les tâches seront effectivement avortées dans l'ordre inverse de celui mentionné dans l'instruction **abort**. Ceci est autorisé par le

manuel (les tâches sont avortée dans *un certain ordre qui n'est pas défini par le langage*), mais peut surprendre l'utilisateur.

Remarquer que toutes les identités de tâche doivent être empilées avant d'exécuter l'instruction *abort*, et qu'il n'est pas possible de transformer la séquence:

```
abort A, B;
```

en:

```
abort A;
abort B;
```

En effet, si la tâche qui exécute l'**abort** dépend de la tâche A, l'avortement de B n'aurait pas lieu dans le second cas.

5.4 Libération de l'espace des tâches

Un soin tout spécial a été aporté dans le système de gestion des tâches pour assurer que l'espace appartenant aux tâches (ici leur segment de pile) est totalement restitué lorsqu'il n'est plus nécessaire. Ceci n'est pas une exigence primordiale dans la version courante d'Ada/Ed, puisqu'une tâche terminée n'occupe que l'espace de son TCB, soit 22 mots. Mais ceci peut devenir important si le parallélisme est utilisé à grande échelle dans une implémentation sur micro-ordinateur, ou si le système proposé est utilisé comme modèle d'exécutif temps réel Ada.

La définition très soigneuse de la notion de maître dans le LRM pourrait laisser à penser que la construction maîtresse d'une tâche représent une limite maximale de la zône où l'objet tâche est accessible; lors de la sortie d'une construction maîtresse, il serait donc possible à la tâche maîtresse de libérer l'espace de toutes ses sous-tâches. Malheureusement, lors de l'étude de ce problème, nous avons pu mettre en évidence un contre-exemple à cette règle:

```
procedure MAIN is
    task type T;
    function F return T;

    task body T is
    begin
        null;
    end T;

    function F return T is
        LOCAL : T;
    begin
        return LOCAL;
    end F;

begin

    if F'TERMINATED then
        null;
    end if;

end MAIN;
```

Le maître de la tâche LOCAL est la fonction F, mais comme celle-ci est l'objet de l'instruction **return**, elle est propagée en dehors du corps de la fonction, où elle peut être l'objet d'un attribut ou un argument d'un sous-programme qui aurait un paramètre de type T. Noter que la tâche est obligatoirement terminée lorsqu'elle est propagée à l'extérieur de son maître, et qu'il est difficile d'imaginer une utilisation en pratique de cette possibilité. Il semblait donc que ce problème n'ait pas été envisagé par les concepteurs du langage, et nous avons soumis un commentaire au comité de maintenance du langage demandant qu'une telle construction soit considérée comme *erronée*, de façon à permettre de libérer l'espace de la tâche à la sortie du maître. Malheureusement, le comité a décidé que cette construction était effectivement autorisée par le langage, et devait donc être soutenue par les implémentations.

La solution que nous avons adoptée a consisté à rendre tout l'espace de la tâche à la sortie du maître, en considérant que si l'indentificateur d'une tâche désignait une pile vide, alors cette tâche était terminée. Ceci nous interdit toutefois de réutiliser le numéros de tâche une fois que leur maître a été quitté, ce qui limite le nombre total de tâches lors d'une exécution à 255 (les numéros sont codés sur un octet) alors qu'autrement la limitation aurait été de 255 tâches existant simultanément, sans limitation sur le nombre total de tâches durant une exécution. La modification serait cependant extrêmement simple à réaliser si jamais le comité revenait sur sa décision lors d'une révision du langage.

La libération de l'espace des tâches avortées peut poser des problèmes spécifiques, car une tâche peut non seulement s'avorter directement, mais aussi indirectement en avortant l'un de ses maîtres. La tâche qui a exécuté l'ordre **abort**, si elle s'est avortée elle-même, doit exécuter une primitive WAIT afin de relancer l'ordonnanceur et céder le contrôle de l'UC; or pour ce faire, elle doit encore posséder une pile. Il n'est donc pas possible de rendre son espace sans précautions. Aussi, lors d'un avortement, l'espace des tâches avortées n'est pas libéré, mais l'identité des tâches avortées est gardée dans un ensemble (RELEASABLE_TASKS), et leur espace est rendu à la fin de la procédure ABORT. Si la tâche s'est avortée elle-même, on aura effectué l'appel à WAIT avant, c'est donc la tâche nouvellement ordonnancée qui effectuera la libération de l'espace. Noter que c'est le seul cas où un traitement est effectué après un appel à WAIT.

6. Algorithme d'ordonnancement

6.1 Ordonnancement général; priorités

L'algorithme d'ordonnancement choisi est un simple ordonnancement circulaire hiérarchisé, ce qui signifie que chaque tâche d'un niveau de priorité donné obtient le contrôle de l'UC dans l'ordre premier arrivé - premier servi. Aucune tâche ne peut obtenir le contrôle de l'UC si une tâche de plus haute priorité est candidate [LRM 9.8(4)].

Le sous-type PRIORITY est défini dans le paquetage SYSTEM comme:

```
subtype PRIORITY is INTEGER range 1 .. 10;
```

Ceci autorise donc 10 niveaux de priorité. Ce choix est totalement arbitraire et peut aisément être changé, chaque niveau supplémentaire coûtant une entrée dans la table d'ordonnancement. Compte tenu de l'aspect pédagogique d'Ada/Ed, il est apparu souhaitable d'avoir plusieurs niveaux de priorité, mais il n'a pas semblé nécessaire d'en prévoir un grand nombre. Il existe en fait de façon interne un onzième niveau de priorité, le niveau 0, qui n'est utilisé que par la tâche *idle_task* (v. plus loin). Ce niveau réservé garantit qu'*idle_task* ne peut être ordonnancée que lorsqu'aucune tâche utilisateur n'est candidate.

La table d'ordonnancement est en fait un tuple de longueur 11 (un élément par niveau de priorité), chaque élément étant un tuple contenant des identificateurs de tâches. Lorsqu'une tâche perd le contrôle de l'UC en exécutant une primitive WAIT, une autre tâche est prise dans le premier élément du tuple de plus haute priorité non vide.

Lorsqu'une tâche est réveillée, suite à la réception d'une primitive SIGNAL, elle est mise à la fin du tuple correspondant à son niveau de priorité, ce qui correspond bien à un ordonnancement circulaire.

Dans l'implémentation en C, ce tuple serait remplacé par une table de chaînages doubles, un désignant la tête de chaîne, et l'autre la fin de chaîne des tâches en attente sur le niveau de priorité considéré. Les tâches de même priorité seraient chaînées entre elles par l'élément *next* de leur TCB, qui est toujours disponible dans ce contexte, puisqu'une tâche ne peut être ordonnancée alors qu'elle est en attente de rendez-vous.

Si le pragma PRIORITY n'est pas spécifié, la priorité par défaut du programme principal est de 6, et elle est de 5 pour toutes les autres tâches. La raison de ceci est qu'il nous a paru utile d'avoir des priorités par défaut vers le milieu de l'intervalle autorisé, de façon à permettre des valeurs explicites aussi bien supérieures qu'inférieures aux valeurs par défaut. Le modèle de parallélisme d'Ada étant orienté vers l'organisation du programme en tâches serveuses effectuant des actions pour des clients, et comme le programme principal n'est pas une tâche, et ne peut donc jouer qu'un rôle de client, il nous a paru intéressant de lui attribuer une priorité supérieure à celle des autres tâches. Ainsi, lorsqu'il a obtenu un service d'une tâche, il peut reprendre le contrôle immédiatement, sans avoir à attendre que la tâche serveuse ait fini d'effectuer des travaux de "nettoyage" consécutifs à l'exécution du rendez-vous.

6.2 Gestion du temps

Le principe de gestion d'horloge décrit dans le modèle n'est pas directement applicable, car la machine simulée ne dispose pas d'interruptions matérielles permettant de définir une durée sous forme de nombre de tops d'horloge restant. Par contre, le système d'exploitation nous fournit un moyen de connaître l'heure absolue. Toutes les durées figurant dans la chaîne d'horloge sont donc des heures absolues, et non plus de durées différentielles comme dans le modèle.

En principe, lors de chaque instruction (simulée), il faut vérifier l'heure absolue pour voir s'il est nécessaire de réveiller la première tâche de la chaîne d'horloge. En fait, afin d'éviter des appels inutiles à l'horloge du système, un indicateur global appelé NEXT_CLOCK_FLAG est positionné à True seulement s'il existe, parmi les tâches en attente de délai, au moins une dont la priorité soit supérieure à celle de la tâche en cours d'exécution. Si ce n'est pas le cas, il n'est pas nécessaire d'aller vérifier l'heure, puisqu'en aucun cas il ne serait possible d'ordonnancer une des tâches de la chaîne. Si

NEXT_CLOCK_FLAG vaut True et que l'heure courante est supérieure à NEXT_CLOCK, un appel est effectué à la procédure UPDATE_CLOCK, qui effectuera une primitive SIGNAL sur toutes les tâches de la chaîne dont l'heure de réveil est inférieure ou égale à l'heure courante, avec l'événement TIMER_EVENT. Dans le cas normal d'un programme n'effectuant aucune attente, en particulier le cas des programmes n'utilisant pas les tâches ni l'instruction **delay**, le temps perdu à l'administration de l'horloge se réduit à celui nécessaire au test d'une variable boléenne globale à chaque exécution d'une instruction de la machine Ada. Il paraît difficile de faire moins...

6.3 Partage de temps

La gestion du temps décrite ci-dessus s'applique au cas où une tâche active garde le contrôle de l'UC jusqu'à ce qu'elle se bloque où qu'une tâche de priorité supérieure soit candidate à l'UC. Ceci correspond au mode de fonctionnement par défaut de l'interprète. Il est cependant possible de spécifier une option pour demander le partage de temps entre tâches de même priorité, c'est à dire de spécifier une tranche de temps maximale pendant laquelle une même tâche peut occuper l'UC.

Lorsqu'une tâche prend le contrôle de l'UC, elle charge le *clock_count* de son TCB avec la valeur de la tranche de temps. Si l'option de partage de temps est spécifiée, ce compteur est décrémenté chaque fois que l'horloge est remise à jour. Lors de la sortie de cette procédure, la primitive UNLOCK forcera un appel à WAIT(0) si le compteur est négatif ou nul, ce qui aura pour effet de forcer la tâche à relâcher l'UC pour un tour. Noter que si l'option de partage de temps est validée, NEXT_CLOCK_FLAG sera vrai dès qu'il y aura dans le système une tâche de priorité supérieure *ou égale* à celle de la tâche courante. L'usage de cette option augmentera donc quelque peu le temps perdu à la gestion de l'horloge.

7. Démarrage du programme

Deux tâches au moins sont toujours présentes dans toute exécution de programme Ada. La tâche numéro 1 est appelée *idle_task*. Son seul but est de garantir qu'il y a toujours au moins une tâche ordonnançable dans le système, afin de maintenir la machine Ada active. Sa priorité est 0, ce qui est inférieur à la plus petite priorité utilisateur, de façon à garantir qu'elle ne puisse être activée que si aucune tâche utilisateur n'est candidate. Lorsque *Idle_task* est activée, un test est effectué pour vérifier qu'il existe au moins une tâche sur CLOCK_CHAIN, c'est à dire en attente de délai. Si ce n'est pas le cas, le

programme utilisateur est dans une situation d'étreinte fatale irrémédiable, et l'exécution s'arrête. Il est intéressant de noter que ceci constitue le seul cas d'étreinte fatale définitive en Ada: en effet des étreintes fatales partielles entre tâches peuvent toujours être défaites si certaines tâches qui y participent se font avorter par d'autres tâches demeurées actives. Une conséquence de ceci est qu'il doit être extrèmement difficile d'établir un algorithme destiné à prouver la présence d'étreintes fatales dans un programme Ada si celui-ci comporte une instruction **abort**. Ceci ne doit cependant pas être vu de façon totalement négative, car en fait cela signifie qu'un programme peut décider que certaines tâches sont en étreinte fatale si elles ne sont pas à un rendez-vous au bout d'un temps déterminé, et corriger la situation en les avortant et en les relançant.

La tâche numéro 2 est appelée *main_task*. Celle-ci constitue la tâche anonyme qui appelle l'unité déclarée comme programme principal au moment du binding. Elle commence par élaborer les paquetages de bibliothèque en appelant les procédures générées à cet effet par le compilateur, puis la procédure principale. Elle comporte différents traite-exception pour gérer les exceptions se produisant pendant l'élaboration des paquetages de bibliothèque, ainsi que les exceptions non traitées par la procédure principale. Elle contient également les patrons de type pour les types prédéfinis, ainsi que pour d'autres types et des constantes nécessaires au compilateur. Le comportement de ces tâches peut être décrit par la tâche Ada ci-dessous (STOP est supposée être une procédure prédéfinie provoquant l'arrêt de l'interprète, avec un argument de type STRING donnant la raison de l'arrêt; MAIN est la procédure servant de programme principal):

```
task IDLE_TASK is
    pragma PRIORITY(0);
end IDLE_TASK;

task body IDLE_TASK is

    CONSTRAINT_ERROR  : exception;
    NUMERIC_ERROR     : exception;
    PROGRAM_ERROR     : exception;
    STORAGE_ERROR     : exception;
    TASKING_ERROR     : exception;
    SYSTEM_ERROR      : exception;

    type      INTEGER     is range -2**15 .. 2**15 - 1
    type      BOOLEAN     is (FALSE, TRUE);
    subtype   NATURAL     is INTEGER range 0 .. INTEGER'LAST;
    subtype   POSITIVE    is INTEGER range 1 .. INTEGER'LAST;
    type      CHARACTER   is --etc...
    type      STRING      is array(POSITIVE range <>) of CHARACTER;
    type      DURATION    is delta 0.001 range -2147483.647 .. 2147483.647;
    for DURATION'SMALL use 0.001;

    subtype NULL_INDEX is INTEGER range 1 .. 0;
    NULL_STRING : STRING := "";
    type INTEGER_FIXED is delta 1.0 range  -2147483.647 .. 2147483.647;
    type FLOAT is digits 6;

    task type MAIN_TASK_TYPE is
        pragma PRIORITY(xx);    -- xx declaré dans la procédure principale
    end MAIN_TASK_TYPE;          -- Défaut = 5

    task body MAIN_TASK_TYPE is
    begin
        declare
            -- Déclaration des unités de librairie
        begin
            MAIN;
            STOP("Terminaison normale");
        exception
            when others =>
                STOP("Exception propagée hors du programme principal");
        end;
    exception
        when others =>
            STOP("Exception durant l'élaboration d'unités de librairie");
    end MAIN_TASK_TYPE;

begin  -- IDLE_TASK
    loop
        null;
    end loop;
end IDLE_TASK;
```

Remarquer qu'il n'est pas possible de faire élaborer les unités de librairie par *idle_task*, car la partie instructions d'un corps de paquetage peut exécuter une instruction **delay**, ce

qui provoque la suspension de la tâche qui effectue cette élaboration, et il ne faut jamais suspendre *idle_task*.

L'initialisation de l'interprète consiste à se mettre dans le contexte d'*idle_task* (segment de code 1) et à lancer la machine Ada. Tout le reste s'enchaîne alors automatiquement.

8. Performances de l'implémentation

Le système écrit en SETL sur lequel nous avons travaillé avait pour seule fonctionnalité de servir de prototype au produit définitif, après recodage dans le langage C. Le prototype SETL a été abandonné après notre départ fin Janvier 1985, et l'effort de l'équipe s'est alors concentré sur la traduction en C. Le prototype SETL passait à cette époque avec succès plus de 90% des tests de la suite de validation, et en particulier 100% des tests du chapitre 9 (gestion des tâches).

Le but d'un prototype n'a jamais été la performance, aussi n'avons nous pas fait de mesure rigoureuse sur notre système. Nous avons fait toutefois quelques essais empiriques avec des programmes utilisant la gestion des tâches de façon intensive ("philosophes et spaghettis"), qui semblaient montrer que la Machine Ada s'exécutait environ 10 fois plus vite que l'interprète de haut niveau. Ce type de programme représente la borne inférieure des gains de performances par rapport à l'ancien système: des gains de plusieurs centaines de fois ont été mesurés sur des programmes effectuant des calculs numériques (type "produit de matrices").

Cette différence d'ordre de grandeur du gain de performances obtenu n'est pas surprenante, et ne signifie rien quant aux performances intrinsèques de notre système: l'essentiel des opérations de gestion des tâches s'effectuant à l'intérieur de l'interprète, dans le nouveau comme dans l'ancien système, c'est là que la différence de niveau sémantique entre les représentations du code de l'ancien et du nouveau système intervient le moins; autrement dit cette différence provient essentiellement du fait que c'était sur les programmes mettant en jeu principalement la gestion des tâches que l'ancien système était proportionnellement le moins catastrophique.

Après que le système ait été recodé en C, l'équipe de NYU a mené quelques expériences, trop rapides pour être réellement qualifiées de mesure, en comparant les temps d'exécution de ces mêmes programmes avec un compilateur de production, le compilateur Vax/Ada de DEC. Sur des programmes mettant en jeu essentiellement la gestion des tâches, il est apparu que notre système était environ deux fois plus lent que

Vax/Ada, avec parfois même des temps d'exécution équivalent si ce n'est même légérement plus rapide que le compilateur DEC. Bien sûr, la même remarque que précédemment s'applique, c'est à dire que c'est sur ce type de programmes que la structure d'interprète d'Ada/Ed est la moins pénalisée par rapport au code compilé de Vax/Ada. Toutefois, ce résultat très encourageant, compte tenu du fait que nous comparons un interprète de machine virtuelle à un compilateur de production générant du code machine optimisé, semble montrer que notre modèle de gestion des tâches est pour le moins viable du point de vue des performances. Des tests plus exhaustifs, et surtout l'implémentation de notre modèle dans un compilateur générant du code natif, seraient nécessaires pour se prononcer définitivement sur ce point.

Enfin, du point de vue de la concision et de la simplicité du sytème, tout l'ensemble de la gestion des tâches tient en approximativement 2000 lignes de SETL de bas niveau (c'est à dire n'utilisant que des fonctionnalités "classiques" que l'on trouve dans les langages conventionnels), y compris les commentaires et les déclarations diverses.

Annexe: Description des instructions spécifiques à la gestion des tâches

Abort

Spécification:

Utilisée pour avorter une ou plusieurs tâches.

Interface:

Le nombre de tâches à avorter est obtenu dans l'instruction. Les identités des tâches à avorter sont dépilées.

Activate

Spécification:

Active toutes les tâches déclarées par des déclarations d'objet dans une partie déclarative. Cette instruction est générée à la fin d'une partie déclarative si celle-ci contient des déclarations d'objets susceptibles de contenir des tâches. Remarquer que dans le cas d'enregistrements avec parties variantes, il n'est pas possible de déterminer à la compilation si des objets tâches ont effectivement été déclarés. L'instruction est toujours générée dans ce cas.

Interface:

Dépile un environnement de tâches à partir de l'environnement de bloc courant et active toutes les tâches sur la chaîne correspondante. Le maître de ces tâches est initialisé à la tâche courante.

Activate_new

Spécification:

Active toutes les tâches créées par l'exécution d'un allocateur.

Interface:

Un argument qui est l'adresse du patron de type du type accès est obtenu dans l'instruction. Le maître des tâches est obtenu à partir de ce patron. Autrement, se comporte comme *activate.*

Create_task

Spécification:

Crée un nouvel objet tâche. Une nouvelle pile est créée et initialisée, le numéro de cette pile est retournée sur le sommet de la pile de l'appelant.

Interface:

L'adresse du patron de type du type tâche est obtenue dans l'instruction.

Current_task

Spécification:

Renvoie l'identité de la tâche courante.

Interface:

L'identité de la tâche courante est mise sur le sommet de pile.

End_activation

Spécification:

Signale à une tâche parente qu'une tâche a terminé son activation.

Interface:

Un argument immédiat est obtenu dans l'instruction, qui vaut 0 si l'activation a échoué (exception levée en cours d'activation), 1 si elle a réussi.

End_rendez_vous

Spécification:

Dernière instruction d'un **accept**, sert à libérer l'appelant.

Interface:

L'identité de l'appelant est dépilée.

Entry_call

Spécification:

Appel d'entrée simple.

Interface:

Les indices de membre, de famille, et l'identité du propriétaire sont dépilés.

Link_tasks_declared

Spécification:

Crée un nouvel environnement de tâches, initialisé par une chaîne donnée. Utilisée en tête de corps de paquetage pour initialiser l'environnement de tâche courant avec les tâches déclarées dans la spécification correspondante.

Interface:

Un pointeur sur la tête de chaîne est dépilé.

Pop_tasks_declared

Spécification:

Sauve un environnement de tâches (une chaîne de tâches déclarées, mais non activée) dans une variable. Utilisée pour préserver les tâches déclarées dans une spécification de paquetage qui doivent être ajoutées aux tâches déclarées dans le corps correspondant.

Interface:

L'adresse de la variable où l'environnement de tâches doit être sauvé est obtenue dans l'instruction.

Raise_in_caller

Spécification:

Utilisée pour propager une exception à l'appelant durant un rendez-vous.

Interface:

L'exception à propager est l'exception courante de la tâche, obtenue dans le registre d'exception. La tâche appelante est la première sur la chaîne *serviced_tasks*.

Selective_wait

Spécification:

Réalisation des parties sélection et mise en attente des instructions **select** et **accept**.

Interface:

Le nombre d'alternatives de l'instruction **select** est obtenu dans l'instruction, ou 0 dans le cas d'un **accept** simple. Des descripteurs d'alternatives sont dépilés. Après exécution de l'instruction, la pile contient le numéro de l'alternative choisie, l'identité de l'appelant et les adresses des paramètres.

Terminate

Spécification:

Utilisée pour terminer l'exécution de la tâche courante. S'il existe des sous-tâches actives, le compteur d'instruction est décrémenté de façon à pointer de nouveau sur l'instruction *terminate*, et la tâche est suspendue en attente de la terminaison de ses sous-tâches. Sinon, la tâche est mise dans l'état terminé, et les vérifications nécessaires sont effectuées sur son maître.

Interface:

Un argument donnant le "type" de *terminate* (fin de corps de tâche, alternative **terminate**, fin du programme principal) est obtenu dans l'instruction.

Timed_entry_call

Spécification:

Appel d'entrée conditionnel ou temporisé. L'appel conditionnel est généré comme un appel temporisé avec délai nul.

Interface:

La valeur du délai, les indices de membre, de famille, et l'identité du propriétaire sont dépilés.

Wait

Spécification:

Suspend l'exécution de la tâche pour un certain temps.

Interface:

La valeur du délai est dépilée.

Appendices

A. Appendice F: Caractéristiques dépendant de l'implémentation

0. Limites de l'implémentation

Longueur maximale d'un identificateur: 120
Nombre maximal de caractères dans une ligne de source: 120
Nombre maximal de lignes dans un fichier source: 32767

1. La forme, les endroits autorisés et les effets de chaque pragma dépendant de l'implémentation

Ada/ED ne reconnait pas de pragma dépendant de l'implémentation. Les pragmas définis par le langage sont correctement reconnus et leur légalité est vérifiée, mais, à part LIST et PRIORITY, ils n'ont aucun effet sur l'exécution du programme. Un message d'avertissement est généré pour indiquer que le pragma est ignoré par Ada/ED.

2. Le nom et le type de chaque attribut dépendant de l'implémentation

Ada/ED ne définit aucun attribut dépendant de l'implémentation.

3. La spécification du paquetage SYSTEM

```
package SYSTEM is

   type SEGMENT_TYPE   is new INTEGER range 0..255;
   type OFFSET_TYPE    is new INTEGER range 0..32767;
   type ADDRESS is
      record
         SEGMENT : SEGMENT_TYPE := SEGMENT_TYPE'LAST;
         OFFSET  : OFFSET_TYPE  := OFFSET_TYPE'LAST;
      end record;

   type NAME is (ADA_ED);

   SYSTEM_NAME   : constant NAME := ADA_ED;

   STORAGE_UNIT  : constant  := 16;
   MEMORY_SIZE   : constant  := 2**16 - 1;

   -- Nombres nommes dependant du systeme

   MIN_INT           : constant := -(2**30) - 1;
   MAX_INT           : constant := 2**30 - 1;
   MAX_DIGITS        : constant := 6;
   MAX_MANTISSA      : constant := 63;
   FINE_DELTA        : constant := 2.0**(-30);
   TICK              : constant := 0.01;

   -- Autres declarations dependant du systeme

   subtype PRIORITY is INTEGER range 1..10;
   SYSTEM_ERROR : exception;

end SYSTEM;
```

4. La liste de toutes les restrictions sur les clauses de représentation

Ada/ED ne supporte de clause de représentation que pour le *small* des types point fixe, et un programme contenant toute autre clause de représentation est considéré illégal.

Les *small* valides pour cette implémentation sont de la forme:

$$S = 2^p \cdot 5^q$$

où $-9 <= q <= 9$, avec la condition $2^{(-30)} <= S <= 2^{30}$.

Les attributs de représentations sont reconnus comme demandé par la définition du langage.

5. Les conventions utilisées pour tout nom généré par l'implémentation qui dénote des composants dépendant de l'implémentation

Ada/ED ne fournit aucun nom généré par l'implémentation puisque les clauses de représentation ne sont pas supportées.

6. L'interprétation des expressions qui figurent dans les clauses d'adresse, y compris celles pour les interruptions

Les expressions d'adresse sont supportées par Ada/ED. Le type ADDRESS défini dans le paquetage SYSTEM est un enregistrement constitué de deux champs. Le premier est un octet non signé qui contient le numéro de segment. Le deuxième contient le déplacement dans le segment, dans l'intervalle de 0 à 32767.

Les clauses d'adresse portant sur des entrées (interruptions) ne sont pas supportées.

7. Toutes les restrictions sur les conversions sans vérification

Ada/ED reconnait les utilisation de conversions sans vérification et vérifie leur validité. Cependant, tout programme exécutant une conversion sans vérification est considéré erronné, et l'exception PROGRAM_ERROR est levée.

8. Toutes les caractéristiques des paquetages d'entrées-sorties qui dépendent de l'implémentation

A) Les fichiers temporaires sont supportés. La convention de nommage est: XHHMMSS.TMP, avec:

X= S - SEQUENTIAL _IO
D - DIRECT_IO
T - TEXT_IO
HH - Heure de création
MM - Minute de création
SS - Seconde de création

B) La suppression de fichier est supportée.

C) Un seul fichier interne peut être associé à un fichier externe donné (l'accès multiple aux fichiers n'est pas supporté).

D) Les noms de fichiers utilisés par les procédures CREATE et OPEN sont des noms de fichier standard VMS. La fonction FORM renvoie la chaîne donnée en paramètre FORM lors de la création du fichier. Aucune caractéristique dépendant du système n'est associée à ce paramètre.

E) 17 fichiers au maximum peuvent être ouverts simultanément durant l'exécution du programme.

F) Le fichier d'entrée standard par défaut peut être spécifié au moyen du paramètre DATA de la commande ADA. Si un fichier est spécifié, il doit être possible de l'ouvrir au début de l'exécution du programme, ou sinon l'exception PROGRAM_ERROR sera levée. Si aucun fichier n'est spécifié, SYS$INPUT sera utilisé. Le fichier de sortie standard est SYS$OUTPUT.

G) SEQUENTIAL_IO et DIRECT_IO supportent les types tableau contraints, les types enregistrement sans discrimants, et les types enregistrement avec discriminants ayant des valeurs par défaut.

H) Les entrées-sorties de types accès sont possibles, mais l'utilisation de valeurs accès engendrées dans une autre exécution est erronnée.

I) La terminaison du programme principal provoque la fermeture de tous les fichiers ouverts, et la destruction de tous les fichiers temporaires.

J) LOW_LEVEL_IO n'est pas supporté.

K) Une marque de fin de page est constituée d'un enregistrement de fichier contenant le seul caractère Form Feed (Ascii.FF); l'effet de l'utilisation de ce caractère dans un fichier de données est indéfini.

B. Le langage de programmation SETL

La quasi-totalité du projet NYUADA a été réalisée sur un ordinateur DEC VAX 11/780, en utilisant le langage de programmation SETL.

SETL est un langage de programmation de très haut niveau, d'usage général, qui permet de résoudre un large éventail de problèmes de programmation de façon relativement efficace avec un minimum d'effort. Il a été développé à New York University.

Ses primitives de bases sont celles de la théorie des ensembles (SETL = SEt Theoretic Language). On pourra en trouver des descriptions détaillées dans [Dewar 79, Schonberg 81, Dewar 82], ainsi que des rapports sur son utilisation en tant que langage de prototypage dans [Kruchten 83, Kruchten 84a, Kruchten 84b, Kruchten 85, Rosen 85, Schonberg 86]. Nous nous limiterons ici à la présentation des points les plus significatifs, et aux caractéristiques du langage qui ont le plus contribué à notre travail d'écriture de prototype.

SETL est un langage séquentiel, impératif, avec assignation, faiblement typé, et à allocation dynamique de mémoire, ce qui en fait un cousin de LISP, SNOBOL et APL. Il possède les types simples usuels: nombres, booléens, chaînes de caractères et atomes générés (style **gensym** de LISP), mais ses types de données structurés les plus importants sont les ensembles, les séquences et les relations (en anglais: *sets*, *tuples* et *maps*).

1. Les *ensembles* de SETL ont toutes les propriétes mathématiques usuelles: ce sont des collections non-ordonnées d'objets de types quelconques, ne contenant pas deux fois le même élément, et sur lesquels les opérations ensemblistes classiques: union, intersection, différence, cardinal, ensemble des sous-ensembles, sont définies. Les *constructeurs* d'ensemble et les *itérateurs* sur ensemble sont des structures de contrôle du langage qui opèrent sur des ensembles; par exemple:

   {x **in** S | C(x)}

   représente le sous-ensemble de S dont les éléments x satisfont le prédicat C(x). Les expressions quantifiées sur des ensembles permettent de décrire des boucles de recherche:

   exists x **in** S | C(x)

   est une expression booléenne dont la valeur est TRUE s'il existe un élément x de S qui satisfait C(x).
2. Les *séquences* de SETL sont des suites ordonnées de longueur arbitraire, indexées par des entiers positifs, dont les composants, comme ceux des ensembles, sont de types quelconques. Il est donc possible de construire des ensembles de séquences, des séquences d'ensembles, des séquences de séquences d'ensembles d'entiers, etc... L'insertion et la suppression d'éléments à l'une ou l'autre extrémité d'une séquence permet de les utiliser comme des piles ou comme des files d'attente. La concaténation de séquences les rend similaires

aux listes. Des séquences hétérogènes de longueur fixe sont souvent utilisées là où dans d'autres langages on se servirait d'enregistrements. Les constructeurs de séquences et les itérateurs sur séquences sont similaires à ceux sur les ensembles. Par exemple:

[x: x **in** [2..100] | (**not exists** y **in** [2..x-1] | x **mod** y = 0)]

construit la séquence des nombres premiers inférieurs à 100.

3. Les *relations* en SETL reproduisent fidèlement la notion de relation en théorie des ensembles: une relation est un ensemble de paires ordonnées, c'est-à-dire de séquences de longueur 2. Les premiers composants de ces séquences constituent le domaine de la relation, les seconds la portée. Domaine et portée d'une relation peuvent être des ensembles quelconques: on peut construire des relations dont le domaine est un ensemble de relations, des relations de séquences vers ensembles de chaînes de caractères, etc. Les relations peuvent être utilisées comme fonctions tabulées, et l'application R(x), où R est une relation peut s'évaluer comme une expression ou être cible d'une affectation. Enfin l'image d'un élément x peut être unique (fonctions) ou être un ensemble (correspondance); dans ce dernier cas, c'est l'ensemble-image R{x} qui peut être évalué ou affecté.

Ces notions familières constituent le coeur de SETL. Mais ce qui distingue SETL d'une pure expression mathématique est la contrainte d'exécutabilité. Afin de garantir que toute construction SETL est exécutable, seuls des objets finis peuvent être représentés, et ils doivent être construits explicitements avant de pouvoir être utilisés ailleurs; il n'y a pas d'"évaluation paresseuse" en SETL. Pour pouvoir exprimer des notions d'objets infinis, les mécanismes habituels d'itération, d'appel de procédures et de récursion sont disponibles, dans un cadre syntaxique similaire à PASCAL, mais sans structure de bloc.

Cette simplicité conceptuelle s'appuie sur un environnement d'exécution complexe, capable d'exécuter toutes les primitives sur les ensembles, séquences et relations, quels que soient les types de leurs composants. On peut dire que les ensembles, séquences et relations sont des types de données abstraits pour lesquels le processeur SETL dispose d'une complète implémentation, libérant l'usager de la charge de spécifier lui-même comment ils doivent être représentés et manipulés sur une machine réelle. Ce langage a été développé à New York University entre 1970 et 1980.

6 Appendices

La version du système SETL que nous avons utilisée date de 1978. Elle est disponible sur VAX (sous Unix(R) et sous VMS), sur IBM (sous VMS et CMS, sur DEC20, sur AMDAHL (sous UTS), ainsi que sur machines SUN et ApolIO.

C. Lady Ada Augusta Byron

Ada Augusta Byron, comtesse de Lovelace (1815-1851), était la fille de Lord Byron, le fameux poète anglais. Elle était passionnée de mathématiques, ce qui était scandaleux pour une femme à son époque, et lui attira bien des inimitiés. Amie et disciple de Charles Babbage, elle eut l'intuition de la possibilité d'utiliser des suites d'instructions codées d'après le modèle des métiers Jacquard. Elle écrivit des programmes pour calculer les nombres de Brenouilli sur la machine de Babbage, mais qu'elle ne put exécuter puisque la machine ne fonctionna jamais. Ses programmes furent recodés bien plus tard en PL/1 et fonctionnèrent du premier coup... Elle est donc considérée comme le premier programmeur de l'histoire.

(R) Unix est une marque déposée des Bell Telephone Laboratories.

Bibliographie

[Aho 77] A. V. Aho, J. D. Ullman: *Principles of Compiler Design*, Addison-Wesley, Reading (Mass.), 1977

[Appelbe 82] B. Appelbe, G. Dismukes: *An Operational Definition of Intermediate Code for Implementing a Portable Ada Compiler*, Proceedings of the AdaTEC Conference on Ada, Arlington (Va), Octobre 6-8, ACM, 1982, pp. 266-274

[Barnes 84] J.G.P. Barnes: *Programming in Ada*, International Computer Science Series, Addison-Wesley, 2ème édition, 1984.

[Burns 86] A. Burns, A. Lister, A. Wellings: *A review of Ada Tasking*, Computer Science Report PR.12, University of Bradford, 1986

[Chiabaut 85] J. Chiabaut: *Travaux sur le compilateur Ada/ED*, rapport de stage de troisième année, ENST, 1985.

[Dewar 79] R.B.K. Dewar, A. Grand, S.C. Liu, J.T. Schwartz, E. Schonberg: *Programming by refinement, as exemplified by the SETL representation sublanguage*, ACM Transactions on Programming Languages and Systems, volume 1, no1, pp.27-49, Juillet 1979.

[Dewar 80] R.B.K. Dewar, G.A. Fisher. Jr., E. Schonberg, R. Froehlich, S. Bryant, C.F. Goss, M.G. Burke: *The NYU Ada Translator and Interpreter*, Proceedings of the Compsac'80 Conference, Chicago, IEEE, Octobre 1980.

[Dewar 82] R.B.K. Dewar, E. Schonberg, J.T. Schwartz: *Higher level Programming - An introduction to the Programming Language SETL*, Courant Institute of Mathematical Sciences, Juillet 1982

[Dijkstra 68] E. W. Dijkstra: *Cooperating Sequential Processes*, in: *Programming Languages*, F. Genuys ed., Academic Press, New York, 1968, pp. 43-112.

[DoD 78] United States Department Of Defense: *Department of Defense Requirements for High Order Computer Programming Languages - STEELMAN*, Defense Advanced Research Projects Agency, Arlington (VA), Juin 1978.

[DoD 80] United States Department Of Defense: *Reference Manual for the ADA Programming Language*, Proposed Standard Document, Juillet 1980.

[DoD 82] United States Department Of Defense: *Reference Manual for the ADA Programming Language*, Draft Proposed ANSI Standard Document for Editorial Review, Juillet 1982.

[DoD 83] United States Department of Defense: *Reference Manual for the ADA Programming Language*, ANSI/MIL-STD-1815 A, Janvier 1983.

[Falis 82] E. Falis: *Design and Implementation in Ada of a Run-Time Task Supervisor*, Proceedings of the AdaTEC Conference on Ada, Arlington (Va), Octobre 6-8, ACM, 1982, pp. 1-9

[Froggatt 86] T. Froggatt: *Fixed-Point Conversion, Multiplication, & Division, in Ada.* Systems Designers PLC, April 1986.

[Goodenough 81] J.B. Goodenough: *The Ada Compiler Validation Capability*, IEEE Computer, Juin 1981, pp. 57-64.

[Gottlieb 83] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, M. Snir: *The Ultracomputer - Designing an MIMD Shared Memory Parallel Computer*, IEEE Transaction on Computers, volume C-32, no 2, pp. 175-189, Février 1983.

[Ibsen 83] L. Ibsen, L. O. K. Nielsen, N. M. Joergensen: *A-machine specification*, Rapport interne ADA/RFM/0001, Christian Rovsing A/S, Ballerup (Danemark), March 15, 1983.

[Ibsen 84] L. Ibsen: *A portable virtual machine for Ada*, Software-Practice and Experience, volume 14, no 1, Janvier 1984, pp. 17-29.

[Ichbiah 79a] J.D. Ichbiah: *Preliminary ADA Reference Manual,* SIGPLAN Notices, volume 14, no 6, Juin 1979 (Partie A).

[Ichbiah 79b] J.D. Ichbiah, J.G.P. Barnes, J.C. Heliard,B. Krieg-Brueckner, O. Roubine, B.A. Wichmann: *Rationale for the Design of the ADA Programming Language*, SIGPLAN Notices, volume 14, no 6, Juin 1979, (Partie B).

[Kamrad 83] J.M. Kamrad: *Run Time Organization for the Ada Language System Programs*, proceedings of the Ada/Europe-AdaTEC Joint Conference on Ada, Bruxelles 16-17 Mars 1983, pp. 10(1-23).

[Katwijk 84] J. van Katwijk, J. van Someren: *The Doublet Model: Run-Time Model and Implementation of Ada Types*, SIGPLAN Notices, volume 19, no 1, ACM, Janvier 1984, pp. 78-92.

[Kok 84] Jan Kok et al.: *Proposal for a Standard Mathematical Package for Ada*, Ada Letters, volume IV, no 3, Novembre-Décembre 1984

[Kruchten 83] P. Kruchten, E. Schonberg: *The Ada/Ed system: a large-scale experiment in Software Prototyping using SETL*, Approaches to Prototyping, R. Budde éd., Springer Verlag, 1983.

[Kruchten 84a] P. Kruchten, E. Schonberg: *Le système Ada/Ed: une expérience de prototype utilisant le langage SETL*, Techniques et Sciences Informatiques, volume 3, no 3, 1984, pp. 193-200.

[Kruchten 84b] P. Kruchten, E. Schonberg, J.T. Schwartz: *Software prototyping using the SETL Programming Language*, IEEE Software, volume 1, no 4, 1984, pp.66-75

[Kruchten 85] P. Kruchten: *Le langage de programmation SETL et son utilisation pour la réalisation de prototypes de logiciels*, BIGRE+GLOBULE, no 43/44, Juillet 85.

[Kruchten 86] P. Kruchten: *Une machine virtuelle pour Ada: Architecture*, Thèse de Doctorat, Ecole Nationale Supérieure des Télécommunications, Paris, Octobre 1986.

[Leathrum 84] J.F. Leathrum: *Design of an Ada Run-Time System*, Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments, IEEE, St. Paul(MN), 1984.

[LRM c.s.p(a)] voir [DoD 83]

[NYU 83a] NYUADA (nom collectif): *Semantic actions for Ada*, Technical Report #841983, New York, Courant Institute of Mathematical Sciences.

[NYU 83b] NYUADA (nom collectif): *An executable Semantic Model of Ada*, Technical Report #851983, New York, Courant Institute of Mathematical Sciences.

[Ogor 85] R. Ogor: *Spécification opérationnelle en Ada d'un noyau pour le langage Ada*, Thèse de Docteur-Ingénieur, Université de Rennes 1, Décembre 1985.

[Riccardi 84] G. Riccardi, T. Baker: *A Runtime Supervisor to Support Ada Task Activation, Execution and Termination*, Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments, IEEE, St. Paul(MN), 1984.

[Riccardi 85] G. Riccardi, T. Baker: *A runtime supervisor to support Ada tasking: Rendezvous and delays*, Ada in USE, Proceedings of the Ada International Conference, Paris, Mai 1985.

[Rosen 83] J. P. Rosen: *A kernel for tasks and rendezvous management in Ada*, Proceedings of the Ada-Europe/AdaTEC Joint Conference on ADA, March 16-17, 1983, Bruxelles (Belgique) pp. 8(13-20)

[Rosen 84a] J. P. Rosen: *On the use of TEXT_IO on interactive terminals*, Proceedings of the IEEE conference on Ada applications and environnement, St. Paul, Minnesota Octobre 16-18, 1984, pp. 76-82.

[Rosen 84b] J. P. Rosen: *Arithmétique réelle en Ada*, Actes des journe'es ADA AFCET+ENST, BIGRE + GLOBULE, no 42, Décembre 1984, pp. 67-75

[Rosen 85] J.P. Rosen: *SETL: un langage de très haut niveau pour le prototypage*, Actes des journées AFCET Nouveaux Langages pour le Génie Logiciel, BIGRE+GLOBULE, no 45, Octobre 1985.

[Rosen 86] J.P. Rosen: *Le langage Ada: présentation et situation actuelle*, L'Echo des Recherches, no 123, 1er trimestre 1986.

[Rubine 82] D.H. Rubine: *A Hybrid Ada Interpreter*, Memorandum interne, Bell Laboratories, Murray Hill (N.J.), 1982.

[Schonberg 81] E. Schonberg, J.T. Schwartz, M. Sharir: *An Automatic Technique for Selection of Data Representations in SETL Programs*, ACM Transactions on Programming Languages and Systems, volume 3, no 2, pp. 126-143, Avril 1981.

[Schonberg 85] E. Schonberg, E. Schonberg: *Highly Parallel Ada - Ada on an Ultracomputer*, Ada in USE, Proceedings of the Ada International Conference, Paris, Mai 1985.

[Schonberg 86] E. Schonberg, D. Shields: *From prototype to efficient implementation: a case study using SETL and C*, Proceedings of the 19th Hawai International Conference on System Science, Bruce Shriver ed., IEEE, Janvier 1986

[Volz 85] R. Volz, A. Naylor, T. Mudge, J. Mayer: *Some Problems in Distributing Real-Time Ada Programs across Machines*, Ada in USE, Proceedings of the Ada International Conference, Paris, Mai 1985.

[Wetherell 82] C.S. Wetherell: *The Ada Breadboard Compiler: An Overview*, Memorandum interne no TM82-45412-13, AT&T Bell Laboratories, Murray Hill (N.J.), 1982.

Dépôt légal : 4ème trimestre 1986
Imprimé à l'ENST - PARIS
ISSN : 0751 - 1353