Répartition d'une application Ada

Dawit Bekele - Jean Marie Rigaud

Institut de Recherche en Informatique de Toulouse
Université Paul Sabatier
31 062 Toulouse Cedex
email : bekele@irit.fr - rigaud@irit.fr

RESUME. L'importance des systèmes répartis s'est considérablement accrue cette dernière décennie, y compris dans le domaine des systèmes embarqués, cible principale du langage Ada. Les problèmes rencontrés lors de l'utilisation du langage dans la programmation d'applications réparties sont étudiés. Nous présentons ensuite le projet STRAda qui traite de ces problèmes. Nous terminons par une introduction à la nouvelle version du langage Ada, Ada 9X, qui possède quelques nouvelles caractéristiques destinées à la programmation répartie.

ABSTRACT. The importance of distributed systems has risen significantly the last decade, including in the domaine of embedded systems, the main target of Ada. The problems encountered during the use of the language for the programming of distributed applications will be seen. Then, we will present the STRAda project which deals with these problems. We'll finish by the introduction of the new version of Ada, Ada 9X, which possesses some new features for distributed programming.

MOTS CLES : Ada 83, Ada 9X, Systèmes répartis, Calcul parallèle, Tolérance aux fautes.

KEY WORDS : Ada 83, Ada 9X, Distributed Systems, Parallel processing, Fault tolerance.

1. Introduction

La place attribuée aux systèmes répartis n'a cessé de croître durant la dernière décennie. Cette tendance est due à certains facteurs parmi lesquels on peut citer :

- la baisse des coûts des processeurs matériels,
- la puissance sans cesse croissante de ces processeurs,
- la capacité importante et la rapidité des réseaux informatiques.

En revanche, la complexité et les coûts des applications adaptées à de tels systèmes prennent des proportions considérables. La principale raison est qu'il est souvent difficile d'utiliser les concepts et les outils de génie logiciel, plus particulièrement destinés à une programmation non répartie. En effet, il est très rare qu'un langage de haut niveau inclut des abstractions permettant de manipuler aisément la répartition. Ceci a pour conséquence la

prise en compte, directement dans le code de l'application, des informations caractérisant la configuration matérielle du réseau. Une telle structure de l'application nuit gravement à la portabilité et à la maintenabilité de cette application.

Il arrive aussi que le programmeur ne se sente pas directement concerné par la répartition de son application. L'utilisation de plusieurs processeurs est un moyen simple d'accélérer l'exécution de son application. Dans ce cas, la description de la répartition, par le programmeur, devient une opération contraignante. Il s'agit alors d'un problème d'implémentation ou d'optimisation de l'application.

Le langage Ada est actuellement l'un des rares langages possédant un modèle de parallélisme totalement intégré. En revanche, Ada 83 ne possède aucune notion spécifique de la répartition. En particulier, le langage ne définit pas l'unité de répartition (tâche, paquetage, etc.). Il ne contient aucune abstraction adaptée à ce problème (placement des tâches, communication et synchronisation inter-processeurs, etc.). Ces lacunes seront en partie comblées par la prochaine version du langage.

Le but principal du projet STRAda est de fournir un outil de programmation d'applications réparties en Ada permettant de profiter pleinement des avantages inhérents aux systèmes répartis et des apports du langage Ada en génie logiciel. L'architecture d'ordinateurs ciblée est un ensemble de stations de travail, reliées par un réseau local (par exemple un réseau Ethernet avec NFS). C'est donc un système faiblement couplé.

Il existe deux points de vue sur une application répartie. Le premier prend en compte les fonctionnalités et le parallélisme logique de l'application alors que le second concerne l'architecture du système cible. Nous adoptons donc une programmation en deux phases. Dans la première phase, la programmation des fonctionnalités d'une application répartie est faite en Ada. Le parallélisme logique de l'application est donc exprimé en utilisant toute la puissance du modèle des tâches Ada. Dans une deuxième phase de programmation, bien séparée de la première, le programmeur peut éventuellement décider du placement des différentes tâches. S'il le désire, il peut aussi laisser le système d'exécution du langage choisir ce placement d'une manière transparente.

Dans la première partie de cet article, nous présentons les systèmes répartis et les applications pour ces systèmes. Dans la seconde partie, nous étudions comment les projets qui traitent de la répartition avec Ada ont résolu les principaux problèmes posés. La troisième partie permet de présenter le projet STRAda. Enfin, la dernière partie est consacrée à une introduction à Ada 9X.

2. Les systèmes répartis et Ada

2.1. Les systèmes répartis

Il existe plusieurs définitions des systèmes répartis. Celle que nous avons adoptée dans ce chapitre, définit un système réparti comme un système d'ordinateurs faiblement couplés, reliés par un réseau de

communication local. Un système d'ordinateurs faiblement couplés est un ensemble d'ordinateurs ne partageant pas de mémoire.

Les systèmes répartis ont certains avantages importants sur les systèmes centralisés, ce qui les rend intéressants pour certaines applications. Les principaux avantages sont :

- le temps de réponse prévisible ;
- la possibilité de partage des ressources matérielles (disques, imprimantes, etc.) et des informations ;
- le bon facteur performance/coût intéressant pour les applications nécessitant une grande puissance de calcul ;
- l'extensibilité de la configuration matérielle ;
- la tolérance aux fautes.

Le principal inconvénient des systèmes répartis est la complexité de leur conception. Beaucoup de problèmes facilement résolus dans les systèmes centralisés, tels que le partage des fichiers ou la protection, deviennent ici très complexes. Cela est dû essentiellement à la répartition des informations et du contrôle. Les systèmes d'exploitation répartis essayent de rendre transparente cette complexité et leur utilisation vise à être aussi simple que celle d'un système centralisé. Malheureusement, à cause de considérations techniques et de performance, la répartition n'est jamais complètement transparente au programmeur.

2.2. Les applications réparties

Plusieurs types d'applications peuvent s'exécuter sur des systèmes répartis. Certaines veulent profiter des nombreux avantages de ces systèmes alors que pour d'autres, la répartition est inhérente à l'application. Dans cette section, nous introduisons trois grandes classes d'applications pour systèmes répartis :

◆ *Les applications recherchant une grande puissance de calcul*

Quelques applications nécessitent une grande puissance de calcul. Les exemples sont nombreux. Nous ne citerons ici que l'application de prévision météorologique. Auparavant, seuls les calculateurs parallèles (exemple Cray) étaient utilisés pour obtenir cette puissance de calcul. Le gros problème est que le prix de ces calculateurs est très important. Les systèmes répartis fournissent actuellement des puissances de calcul comparables pour des coûts beaucoup moins élevés.

◆ *Les applications recherchant la tolérance aux fautes*

Il existe des applications pour lesquelles la sûreté de fonctionnement est très importante, voire primordiale. Par exemple, un système de contrôle d'un satellite doit fonctionner dans toutes les conditions. En particulier, il faut éviter l'arrêt total du système causé par la panne d'un processeur. Rien ne peut être fait pour éviter cela avec un système monoprocesseur. En revanche, avec un système réparti, un autre processeur peut être utilisé pour effectuer les tâches du processeur en panne.

◆ *Les applications dont la répartition est inhérente*

De plus en plus d'applications par nature réparties voient le jour. C'est le cas du courrier électronique ("email"), des "news", du "netfind", d'un système de réservation de billets d'avions ; la liste n'est pas exhaustive. Le développement de ces applications peut être expliqué par le développement de réseaux de communication, essentiellement les réseaux à grandes distances qui facilitent l'échange d'informations entre les applications.

3. Le langage Ada 83 et la répartition

Ada est conçu aussi bien pour la *"programmation à petite échelle"* que pour la *"programmation à grande échelle"*. En ce qui concerne la programmation à grande échelle, les mécanismes essentiels sont les suivants :

- les paquetages pour la modularité,
- les tâches pour le parallélisme,
- les exceptions pour la fiabilité.

Actuellement, ces caractéristiques sont bien supportées pour les environnements centralisés. En revanche, leur mise en œuvre dans un système réparti pose des problèmes inhérents au langage. En effet, bien que quelques caractéristiques du langage Ada laissent supposer qu'il est conçu pour les systèmes répartis (pragma *Shared*), d'autres montrent qu'il cible tout au plus une architecture fortement couplée. Parmi ces caractéristiques, on peut citer :

- la structure de bloc et les variables partagées qui supposent un segment de mémoire unique ;
- les appels d'entrées conditionnels dont la sémantique suppose une notion de temps unique dans tout le système cible et ne prévoit pas le délai de communication des messages ;
- les paquetages *Standard* et *System*, les clauses de représentation et les attributs du processeur cible qui sont définis pour un seul processeur (ou au mieux un seul type de processeur).

De plus, Ada 83 ne définit pas d'unité de répartition, ni de méthode de placement de l'application sur un ensemble de processeurs. Ces carences obligent le programmeur d'applications réparties à traiter directement des problèmes qui doivent en principe être résolus par le système de compilation (par exemple l'utilisation de primitives de communication de bas niveau d'abstraction), diminuent la réutilisabilité des applications et rendent complexe l'utilisation des avantages des systèmes répartis (par exemple la tolérance aux fautes).

Plusieurs projets ont ajouté des outils dans l'environnement du langage Ada [BEK 94], afin de pallier à ces carences. Dans cette section, nous étudierons brièvement comment les différents projets de répartition avec Ada ont résolu les problèmes en définissant une unité de parallélisme, une méthode de placement, et des mécanismes de communication et de tolérance aux fautes.

3.1. L'unité de répartition

Un programme réparti s'exécute sur plusieurs processeurs. La granularité de l'unité de répartition est souvent liée à l'architecture matérielle ciblée. Les tâches sont pratiquement les seules unités de répartition utilisées dans les différents projets ciblant les architectures fortement couplées. Ces architectures permettent, facilement et sans trop de surcoût, l'implantation de variables partagées entre tâches ainsi que celle d'autres mécanismes de gestion des tâches (création, terminaison, communication, etc.).

Les nœuds virtuels [TED 87], qui forment une granularité moins fine que les tâches, sont les plus utilisés pour les architectures faiblement couplées. Ces unités ne permettent pas le partage de variables et toutes les communications entre nœuds virtuels se font par échange de messages. Dans DIADEM [ATK 88], les nœuds virtuels sont constitués d'un ensemble d'unités de bibliothèque. Le mécanisme utilisé pour projeter la structure en nœuds virtuels sur la bibliothèque Ada s'appuie sur le concept de "bulle" conçu et développé dans le projet Cnet [INV 85], [JAN 88].

3.2. Le placement

Le placement des unités de répartition sur le système réparti cible peut être traité de manière automatique ou programmée.

Le placement automatique (ou transparent au programmeur) de l'application est surtout intéressant pour les applications qui ciblent un système réparti à mémoire partagée. Les processeurs de tels systèmes sont souvent identiques et une tâche peut être exécutée par un processeur quelconque.

Le placement programmé (ou non transparent) est celui envisagé par la plupart des projets (DIADEM, [JHA 89], [HUT 90]). Plusieurs méthodes sont utilisées pour la spécification du placement. La plus simple est l'insertion d'un pragma spécifique dans le code de l'unité de répartition. Trois inconvénients sont à noter. Premièrement, un simple changement dans la configuration matérielle nécessite un changement du pragma dans le code de l'unité. Cela provoque la recompilation de l'unité mais aussi de toutes les unités qui en dépendent, si le pragma se trouve dans la spécification de l'unité. Deuxièmement, les informations concernant le placement se trouvent éparpillées dans l'application. Un changement de configuration matérielle nécessite donc la revue d'une grande partie de l'application. Enfin, le pragma étant une commande au compilateur, le placement est figé dès la compilation. Le placement est statique et interdit toute reconfiguration dynamique.

La deuxième méthode est l'utilisation d'un langage spécifique, appelé langage de configuration. Ce langage permet la spécification du placement des unités, dans une phase bien séparée de la programmation des fonctionnalités de l'application. Il permet d'éviter les trois problèmes soulevés par la méthode précédente. En effet, un changement de configuration matérielle ne nécessite plus la recompilation de l'application puisque la spécification du placement est faite après la compilation. Les

informations concernant le placement étant regroupées dans un programme séparé, leur modification en est facilitée. Le troisième problème est plus délicat. En effet, la plupart des projets proposent un placement statique (DIADEM par exemple). Mais, le langage de configuration peut être utilisé de telle sorte que le placement de l'unité de répartition soit décidé à l'exécution. Le langage APPL permet ainsi de fixer le placement d'un objet créé avec un allocateur, par appel d'une fonction de placement.

3.3. La communication

La communication entre unités de répartition est résolue de diverses manières. Lorsque les processeurs sont fortement couplés, les variables partagées peuvent être utilisées. Dewar et al. [DEW 90] ont montré que l'on peut utiliser les caches et les mémoires locales pour minimiser les mises à jour des variables partagées entre tâches Ada s'exécutant sur des processeurs différents. Lorsque le système ne possède pas de mémoire partagée, ce mode de communication ne peut plus être implanté directement. Il est toujours possible d'utiliser les mémoires partagées réparties ("Distributed Shared Memories") à cette fin, mais l'accès à des variables distantes devient malheureusement trop inefficace. Tous les projets que nous avons étudiés et qui ciblent une architecture faiblement couplée ont choisi des mécanismes de communication basés sur l'échange de messages [CRO 90].

L'extension du concept de rendez-vous Ada à celui du rendez-vous distant et celle de l'appel de procédure à l'appel à distance semblent tout à fait naturelles. Elles devraient permettre de réaliser la propriété de transparence à la condition que l'adaptation du rendez-vous ou de l'appel de procédure soit prise en charge par des systèmes de transformation ou des exécutifs spécifiques. L'implantation de ces extensions peut s'appuyer sur la couche transport du réseau de communication qui fournit généralement des primitives nécessaires à l'échange des messages.

L'implantation du rendez-vous Ada dans un environnement réparti pose des problèmes d'ordre temporel puisque celui-ci fait intervenir la notion de temps. En particulier, une tâche peut attendre un rendez-vous pendant un temps limité. Malheureusement, la notion de temps est très difficile à définir dans un environnement réparti, étant donné qu'il est impossible d'avoir des horloges synchronisées sur les différents processeurs du système. Certaines études ont proposé des techniques de définition de temps global au système. On peut citer les serveurs de temps, la synchronisation périodique des horloges locales et l'exportation de délais dans les appels plutôt que celle des temps absolus [VOL 87]. Toutefois, ces techniques restent encore imprécises.

3.4. La tolérance aux fautes

Deux méthodes pour la prise en compte de la tolérance aux fautes ont été utilisées dans les différents projets.

La première est une approche transparente. Avec cette approche, le

système d'exécution prend en charge la récupération des objets et des services perdus suite à une faute d'un ou plusieurs processeurs du système ou suite à des problèmes de communication. Pour cela, les informations concernant les objets et les services doivent être conservées sur d'autres machines grâce à des copies multiples. L'avantage de cette approche est que le programmeur n'est pas concerné par les problèmes de reconfiguration. Le principal inconvénient est qu'elle ne permet pas le remplacement d'un service perdu après une faute par un autre service mieux adapté. Un second inconvénient est la grande surcharge que cette approche implique. Cela est dû au fait que le système doit gérer la récupération entière de tous les objets et de tous les services, même de ceux qui ne sont pas importants, puisque le programmeur n'a pas les moyens lui permettant de spécifier les services pour lesquels il souhaite la récupération.

Dans la deuxième approche - l'approche non transparente - le système signale la faute à l'application qui doit alors gérer la situation. L'inconvénient de cette approche est que le programmeur doit envisager toute reconfiguration nécessaire en cas de faute. Cette approche a plusieurs avantages : le programmeur peut spécifier un service différent du service perdu. Selon les circonstances, le service de remplacement peut être le même que le service perdu, un service dégradé ou un service plus fiable De plus, comme le programmeur gère la reconfiguration, la surcharge d'exécution est limitée. L'approche non transparente [KNI 88] ne peut être adoptée que lorsque la configuration est faite explicitement par le programmeur ; celui-ci doit être capable de placer les unités initiales mais aussi les unités de remplacement.

Lors de l'implantation de la tolérance aux fautes, les quatre actions suivantes sont nécessaires :

- détecter la faute ;
- évaluer les dommages ;
- décider et effectuer une réaction.

Certains projets utilisent de nouveaux mécanismes pour effectuer ces quatre actions. D'autres utilisent les exceptions Ada. Les exceptions étant les mécanismes utilisés pour le signalement et le traitement de situations exceptionnelles, il semble naturel qu'elles soient utilisées pour la détection, le signalement et le traitement des fautes. Il faut toutefois que le système d'exécution permette la propagation des exceptions d'une tâche à une autre, que ces tâches soient distantes ou non.

3.5. Initialisation et terminaison d'une application répartie

Le langage Ada définit une sémantique d'initialisation et une sémantique de terminaison de l'application répartie. L'initialisation d'une application répartie est principalement liée à l'élaboration des divers paquetages constituant l'application.

3.5.1. Initialisation d'une application

Le langage Ada définit un ordre partiel d'élaboration des différents

paquetages. Tout d'abord un paquetage donné doit être élaboré après les paquetages qu'il utilise. Un ordre d'élaboration des paquetages peut être imposé en utilisant le pragma *Elaborate*. De plus, le langage Ada exige que tous les paquetages soient élaborés avant l'élaboration de la procédure principale. Ces règles peuvent être facilement appliquées dans le cas d'un monoprocesseur. Le problème devient plus complexe lorsque l'environnement est réparti, l'élaboration des paquetages pouvant être faite sur différents processeurs.

Il est logique qu'un paquetage soit élaboré par le processeur du site sur lequel il est placé. Lorsque deux paquetages sont élaborés par des processeurs différents (cas où l'unité de répartition est une unité de bibliothèque), ces processeurs doivent dialoguer afin de respecter l'ordre d'élaboration défini par l'application, sinon l'élaboration d'un paquetage peut faire appel à un service d'un autre paquetage non encore élaboré. Plusieurs solutions sont proposées par [HUT 90].

3.5.2. Terminaison des tâches

Le langage Ada ne précise aucune contrainte sur la terminaison des unités de bibliothèque d'une application. Le problème de la terminaison concerne plus particulièrement la terminaison des tâches. La terminaison des tâches en Ada est un problème délicat ([MRA 87] section 9.4) y compris dans un environnement monoprocesseur. La terminaison d'une tâche dépend de l'achèvement de son exécution, de l'achèvement de son maître et de l'état de ses tâches dépendantes. La complexité est encore accrue dans un cadre multiprocesseurs lorsque les tâches sont placées sur des sites différents. Ce problème a souvent été délaissé dans les divers projets de répartition avec Ada. Wellings et al. [WEL 89] sont les seuls à notre connaissance à proposer une solution. La solution proposée modifie considérablement cette sémantique de terminaison. On ne s'intéresse plus à la terminaison de chaque tâche mais à la terminaison globale de toutes les tâches de l'application répartie, c'est-à-dire à la passivité de tous les processeurs du système à un instant donné. Cette modification de sémantique facilite énormément la recherche d'un algorithme de terminaison, puisque le problème devient un problème de terminaison de calcul réparti classique [TOP 84].

4. Le projet STRAda

Le système STRAda (Système de Transformation et de Répartition Ada [BAZ 92]) complète le langage Ada afin de résoudre simplement le problème de répartition. Il repose sur les principes suivants :

- La tâche, qui est l'unité de parallélisme en Ada, est choisie comme unité de répartition de l'application ;
- Le placement des tâches sur les divers sites est réalisé soit statiquement (mode programmé), soit dynamiquement (mode interactif ou mode automatique) ;
- La communication inter-sites est totalement prise en charge par le système et reste transparente au programmeur.

4.1. Le modèle de répartition

Le modèle des tâches s'appuie sur le parallélisme, la communication, la synchronisation, le non déterminisme et la notion de temps. Le choix des tâches Ada comme modèle de répartition permet d'avoir un modèle puissant pour exprimer le parallélisme logique d'une application et autorise la transparence de la répartition. Les projets de répartition avec Ada ayant fait un autre choix ont été contraints de définir un modèle simplifié, moins puissant que celui des tâches Ada. Par exemple, aucun d'entre eux ne fournit le non déterminisme et la notion de temps.

Avec une programmation en deux phases, le parallélisme est exprimé indépendamment de l'architecture cible. Des modifications de l'architecture ne nécessitent aucune modification du programme Ada. Un programme donné peut donc être facilement adapté à une nouvelle architecture répartie ou même à une architecture monoprocesseur. La portabilité se trouve donc renforcée.

La stratégie retenue dans le projet STRAda est donc le *post-partionnement* [BUR 87] : l'architecture du système est prise en compte une fois que le programme est conçu. La plupart des autres projets, dont Ada 9X, ont préféré le *pré-partitionnement* : la répartition est prise en compte dès la conception du programme Ada. Les arguments avancés par ces derniers concernent le nombre important de problèmes à résoudre de manière transparente (mises à jour de variables partagées, terminaison répartie, etc.) [KER 88] et dont les surcoûts d'exécution engendrés peuvent échapper au programmeur, ce qui n'est pas acceptable pour des applications temps réels critiques. En revanche, la simplicité de la programmation que permet cette approche peut être recherchée pour des applications ayant des contraintes de temps moins sévères.

4.2. Implantation du système STRAda

L'architecture du système STRAda est composée de deux outils : le noyau et le système de transformation. Le noyau de répartition minimal permet d'implanter tous les concepts spécifiques au langage Ada. Un noyau STRAda est placé sur chaque site du système réparti et fournit les différents services de communication par échange de messages. Un autre service important fourni par le noyau est la création de tâches. A cet effet chaque noyau possède un serveur de création.

Le second outil, le transformeur permet de traduire le programme source Ada en un autre programme Ada. Dans le programme transformé, toutes les constructions concernant le parallélisme et la communication ont été remplacées par des appels aux services du noyau STRAda. Ce programme peut s'exécuter dans l'environnement réparti.

4.2.1. Définition d'un noyau réparti adapté au langage Ada

Dans cette section, nous exposons les associations que nous avons faites entre les concepts et les mécanismes des langage Ada et système

UNIX. Nous présentons successivement les aspects parallélisme puis synchronisation et communication.

◆ *Parallélisme*

De manière naturelle, nous avons associé à chaque tâche Ada un processus UNIX. D'un point de vue purement pratique, un tel choix peut paraître coûteux : on aurait pu envisager, comme il est fait dans la plupart des compilateurs, de multiplexer plusieurs tâches au sein d'un même processus UNIX (pseudo-parallélisme). Toutefois, la stratégie d'implantation adoptée possède les avantages suivants :

- il est possible d'envisager un parallélisme réel au sein d'un programme Ada, dans le cadre des stations UNIX multiprocesseurs,
- elle permet une meilleure réutilisation du système d'exploitation sous-jacent (UNIX) : la synchronisation d'une tâche Ada est directement exprimée en terme de synchronisation du processus support.

Une alternative serait l'utilisation de processus légers. Nous admettons que ce choix aurait été meilleur pour la performance des applications. Nous l'avons toutefois écarté pour deux raisons, valables au moment du choix :

- Peu de systèmes UNIX implantaient les processus légers (thread). Ce choix constituait donc un handicap pour la portabilité ;
- Les seuls systèmes qui supportaient les processus légers fournissaient une bibliothèque de threads, en dehors du noyau du système d'exploitation. Ceci posait un problème lors des communications inter-threads : un thread qui demande à communiquer avec un autre thread appartenant à un autre processus, bloque tous les autres threads appartenant au même processus. Ce qui n'est évidemment pas acceptable pour les tâches.

Actuellement, la situation a changé. De plus en plus de systèmes UNIX implantent, à l'intérieur même du noyau, les opérations sur les threads. De ce fait, les communications inter-threads ne perturbent pas le parallélisme de l'application. Cela les rend attractifs pour le noyau STRAda. C'est donc un problème à reconsidérer dans les évolutions futures du système.

◆ *Communication et synchronisation*

Une tâche de l'application se synchronise ou communique avec une autre tâche en utilisant les instructions dédiées du langage Ada (appel d'une entrée ou acceptation d'un rendez-vous sur une entrée, instruction *select* du langage Ada, etc.). Le système de transformation remplace ces instructions par des appels à des primitives de communication du noyau.

Afin d'implanter la version répartie des instructions de synchronisation et de communication du langage Ada, nous avons choisi de manière naturelle les sockets comme support de synchronisation et de communication. A chaque entrée Ada, nous associons un socket et un port sur ce socket. Les primitives UNIX (*sendto, recvfrom, select)* permettent d'émettre ou de recevoir sur une de ces entrées ou de choisir une entrée parmi un ensemble d'entrées.

La solution retenue présente des limitations mais a l'énorme avantage de permettre une réutilisation directe du système de communication de UNIX.

4.2.2. Le système de transformation

Plusieurs approches peuvent être utilisées pour obtenir les programmes exécutables à partir du programme Ada initial. La première consiste à utiliser un système de compilation construit à cet effet. Avec cette méthode, la compilation du programme Ada donne directement un ensemble de programmes exécutables destinés au système réparti.

La deuxième approche consiste à utiliser une approche transformationnelle. Le programme Ada multitâche est traduit en un, ou plusieurs autres programmes d'un langage de haut niveau (appelés programmes transformés). Les programmes transformés sont ensuite compilés afin d'obtenir les exécutables du programme réparti.

Dans STRAda, nous avons donc choisi cette approche transformationnelle avec, comme langage cible, le langage Ada lui-même. Un aspect intéressant de cette approche est la réutilisation, sans aucune modification, de systèmes existants (compilateurs Ada, système d'exploitation UNIX, etc.).

Le transformeur a, en entrée, le programme Ada multitâche écrit pour exprimer les fonctionnalités de l'application. Il produit, en sortie, deux programmes Ada. Le résultat de la compilation et de l'édition de lien de ces deux programmes donne deux exécutables qui représentent respectivement le programme principal et le serveur-création.

4.3. Evaluation

Dans cette partie, nous discutons de l'approche adoptée dans le projet STRAda. Comme nous l'avons vu, l'approche STRAda est tout d'abord transformationnelle : nous n'avons pas cherché à définir un nouveau langage ou un nouveau système mais nous avons étudié la transformation de certaines constructions d'un langage existant en celles d'un système existant. Cette approche nous semble intéressante pour plusieurs raisons :

- elle permet de réutiliser des compilateurs et systèmes connus et largement diffusés, ce qui contribue à une bonne portabilité ;
- elle est facilement adaptable à d'autres systèmes : on pourrait de la même façon envisager la transformation vers Chorus [ZIM 81], Mach [JON 86], ou d'autres noyaux temps réel.

L'implantation des variables partagées a été momentanément ignorée ; cependant, nous remarquerons que :

- d'une part, certains systèmes distribués offrent maintenant l'abstraction de mémoire globale sur un réseau,
- d'autre part, grâce au noyau minimal STRAda, nous pouvons envisager un schéma où les variables partagées sont implantées comme des variables réparties encapsulées dans des tâches accessibles à distance, les accès à ces variables partagées par des

invocations d'entrées définies dans ces tâches.

Un algorithme adapté au modèle de terminaison des tâches Ada a été étudié et formellement validé. Il reste maintenant à l'implanter sur le système STRAda.

5. Ada 9X

Après dix années d'utilisation, une révision de la norme Ada est à l'étude et porte le nom de projet Ada 9X. Ce travail vise les objectifs suivants :

- corriger les imperfections constatées ;
- intégrer de nouveaux mécanismes et de nouveaux concepts rendus nécessaires du fait de l'évolution des modes de programmation [BRO 92].

L'étude faite ici est basée sur les documents intermédiaires [PLA 94], la version finale étant prévue fin 1994.

Les changements importants proposés concernent :

- la programmation orientée objet : Ada 9X supporte l'héritage et le polymorphisme par extension du type article existant.
- la programmation à grande échelle ("in the large") : Ada 9X permet la définition d'une hiérarchie d'unités de bibliothèque limitant ainsi le nombre de recompilations après la modification d'une application.
- la programmation temps réel et parallèle : Ada 9X propose une synchronisation orientée donnée par le mécanisme des articles protégés.

Ada 9X est composé d'un noyau qui doit être supporté par tous les compilateurs et d'annexes spécialisées optionnelles. La répartition est explicitement définie, dans l'annexe dédiée aux systèmes répartis.

Nous étudions ci-dessous l'unité de répartition ainsi que les mécanismes de communication choisis par Ada 9X.

5.1. Unité de répartition et placement

Le modèle de répartition de Ada 9X définit la partition comme l'unité de répartition. Une partition est un ensemble d'unités de bibliothèque. La sémantique définit les règles de composition, d'élaboration, d'exécution et de terminaison d'une partition.

Une application Ada 9X est formée d'une ou de plusieurs partitions. Une tâche environnement est associée à chaque partition. Cette tâche appelle le sous-programme principal, s'il y en a, et attend la terminaison de toutes les tâches qui dépendent des unités de bibliothèque de la partition.

Une partition est soit active, soit passive. Les unités de bibliothèque formant une partition active résident et s'exécutent sur le même processeur. En revanche, les unités de bibliothèque formant une partition passive résident sur un module d'adressage commun, accessible aux partitions actives qui l'utilisent.

Une partition active peut nommer un paquetage d'une partition

passive, ce qui permet l'accès aux paquetages communs du système réparti. De même, une partition active peut exécuter des sous-programmes des autres partitions actives. De tels appels sont appelés appels de sous-programmes à distance.

L'implantation décide si l'on peut avoir une ou plusieurs partitions sur le même site ou sur le même espace d'adressage.

L'allocation des partitions sur l'environnement d'exécution cible, c'est-à-dire le placement, n'est pas spécifiée dans l'annexe, elle dépend de l'implantation.

5.2. La communication

La communication entre deux partitions actives se fait généralement par appel de sous-programmes à distance, donc par échange de messages. Lorsqu'il existe une mémoire partagée dans le système réparti, la partition passive permet le partage de données entre partitions actives.

L'appel d'un sous-programme à distance peut être explicite ou implicite. L'appel est explicite lorsqu'on désigne un sous-programme d'un paquetage interface d'une partition distante. Il existe deux formes d'appels implicites :

- par l'utilisation de type accès à des sous-programmes : Le sous-programme effectivement appelé est déterminé dynamiquement et appartient à une partition distante ;
- par liaison dynamique d'un opérateur d'objet, l'objet étant désigné par un type accès : L'objet effectivement appelé est déterminé dynamiquement et appartient à une partition distante. Ceci est une extension de la liaison dynamique classique des langages orientés objets, dans le cadre d'une application répartie.

Par défaut, un appel de sous-programme à distance est synchrone. L'appelant attend la fin de l'exécution du sous-programme appelé avant de poursuivre son exécution. Ada 9X permet la définition de procédures d'appel à distance asynchrones si les paramètres sont tous de mode *in*. Dans ce cas, l'appelant n'attend pas la fin de l'exécution du sous-programme.

Les paquetages appelés paquetages *Remote_Types* sont utilisés pour déclarer les types des paramètres formels des sous-programmes d'appel à distance.

6. Conclusion

Le modèle de répartition Ada 9X est complètement indépendant du modèle de parallélisme. Ce choix simplifie l'implémentation parce que le modèle choisi est bien approprié à la répartition. En revanche, dans le cas d'applications où la répartition est recherchée uniquement pour des raisons d'amélioration du temps de réponse ou pour une meilleure utilisation du matériel, il serait intéressant de dégager le programmeur des préoccupations relatives à la répartition. C'est pour cette classe d'applications que STRAda peut toujours avoir une raison d'être avec Ada 9X. En effet, avec STRAda tout unité parallèle étant potentiellement distribuable, il est possible d'arriver

à une transparence totale de la répartition.

Un prototype du projet STRAda est actuellement opérationnel. Nous avons expérimenté quelques programmes parallèles classiques e.g. une adaptation du problème des philosophes dans un environnement physiquement réparti.

Un aspect pratique qui nous semble intéressant est celui des transformations pour des systèmes ou noyaux temps réels existants ; cela permettrait d'une part, d'écrire ou de réutiliser des applications écrites dans un langage de haut niveau et d'autre part de réutiliser des noyaux temps réels existants et dédiés pour certains types d'architecture.

Enfin d'un point de vue théorique, il serait alors intéressant de valider les transformations correspondantes.

Bibliographie

[ATK 88] C. Atkinson, T. Moreton and A. Natali, *Ada for Distributed Systems*, Cambridge University Press 1988

[BAZ 92] G. Bazalgette, D. Bekele, C. Bernon, M. Filali, J.M. Rigaud & A. Sayah, *STRAda : An Ada Transformation and Distribution System*, Ada : Moving Towards 2000, 11th Ada-Europe International Conference, Zandvoort, The Netherlands, June 1992, Proceedings, Springer-Verlag Heidelberg 1992

[BEK 94] D. Bekele et J.M. Rigaud, *Ada et les systèmes répartis*, TSI (Techniques et Sciences Informatiques), Publication acceptée, prévue 1994

[BRO 92] B. Brosgol, *Ada*, Communication of the ACM, Vol. 35 - n° 11 Nov. 1992, pp. 41-89

[BUR 89] A. Burns and A. Lister A. Wellings, *A review of Ada tasking*, Lect. Notes Compt. Science n° 262

[CRO 90] J. Cross, M. Kamrad and S. Fernandez, *Distributed Communications*, SIGAda, Ada Letters (Fall 1990) Vol 10, n° 9, pp. 85-93

[DEW 90], R. Dewar, S. Flynn, E. Schonberg and N. Shulman, *Distributed Ada on Shared Memory Multiprocessors*, Distributed Ada : developments and experiences, pp. 222-234, Cambridge University Press 1990

[DOB 90] B. Dobbing, *Distributed Ada, A Suggested Solution for Ada 9X* SIGAda, Ada Letters (Fall 1990) Vol 10, n° 9, pp. 94-102

[HUT 90] A.D. Hutcheon and A.J. Wellings, *The York distributed Ada Project*, Distributed Ada : developments and experiences, pp. 67-104, Cambridge University Press 1990

[INV 85] P. Inverardi, F. Mazzanti and C. Montangero, *The Use of Ada in the Design of Distributed Systems*, Ada International Conference, Paris, pp. 72-84, The Ada Companion Series, Cambridge University Press 1985

[JAN 88] H.-St. Jansohn, *Ada for Distributed Systems*, SIGAda, Ada Letters (1988) Vol 8, n° 7, pp. 101-103

[JHA 90] R. Jha and G. Eisenhauer, *Honeywell Distributed Ada - Approach*, Distributed Ada : developments and experiences, pages 137-157, Cambridge University Press 1990

[JON 86] M.B. Jones and R.F. Rashid, *Mach and machmaker : kernel and language support for object oriented distributed systems.*, In N. Meyriwitz, editor, OOPSLA Proc. on object-oriented programming systems, languages and applications, pages 67-77. ACM, sep 1986

[KER 88] Kermarrec Y., *Une aproche de simulation des systèmes ditsribués: les composants logiciels en Ada*, Thèse de l'Université de Renne I, Juin 1988

[KNI 88] J. C. Knight and M. E. Urquhart, *A New Approch To Fault Tolerance in Distributed Ada Programs*, SIGAda, Ada Letters (1988) Vol 8, n° 7, pp. 123-126

[MRA 87] Manuel de Référence du langage de programmation Ada, Alsys, fev 1987
[PLA 94] Programming Language Ada, Language and Standard Libraries, Draft (1 June 1994),Version 5.0

[TED 87] M. Tedd, S. Crespi-Reghizzi and A. Natali, *Ada for multi-microprocessors*, Cambridge University Press 1984

[TOP 84] R. Topor, *Termination Detection for distributed Computations*
Inf. Proc. Letters (Jan. 1984) pp. 33-36

[VOL 87] R. Volz, *Timing Issues in the Distributed Execution of Ada Programs*, IEEE Trans. on Computers c-36 (4), pp 449-459 (Apr. 1987)

[WEL 89] A.J. Wellings, A.D.Hutcheon, *Elaboration and Termination of Distributed Ada*, Ada Europe, Madrid (1989)

[ZIM 81] H. Zimmermann, J. S. Banino, A. Caristan, M. Guillemont and G. Morisset, *Basic concepts for the support of distributed systems : The Chorus Approach, IEEE Catalog NO. 80-83218, pp. 60-67, Apr 81*, Computer Society Press