Le protocole réparti de Distributed Oz

Iliès Alouini — Peter Van Roy

Université catholique de Louvain
Département d'ingénierie informatique
B-1348 Louvain-la-Neuve, Belgium
Email : {ila, pvr}@info.ucl.ac.be
Web : www.info.ucl.ac.be/{~ila, ~pvr}

RÉSUMÉ. *Distributed Oz est un langage de programmation conçu pour écrire des applications réparties. Cet article présente un modèle, le* graphe distribué*, qui définit l'exécution répartie du langage. Dans ce modèle, chaque catégorie d'entité du langage Oz (donnée avec état, donnée sans état, et donnée à affectation unique) est implémentée par un protocole qui défini son comportement réseau. Le modèle explique l'intéraction de ces entités, et en particulier, comment des opérations réparties sur une entité peuvent entrainer la création de nouvelles références distantes vers d'autres entités. Nous définissons le protocole pour les données sans état (procédures et enregistrements) dans ce modèle; les deux autres protocoles ont été défini dans des articles antérieurs. Le graphe distribué fait partie du système Mozart, qui implémente Oz.*

ABSTRACT. *Distributed Oz is a programming language designed for building distributed applications. This paper defines a model, called the* distribution graph*, for the distributed execution of the language. Within this model, each category of Oz language entity (stateful, stateless, and single assignment) is implemented by a protocol that defines its network behavior. The model explains how these language entities interact, and in particular, how distributed operations on one language entity can result in the creation of new remote references to other language entities. We define the protocol for stateless data (procedures and records) within this model; the other two protocols have been published previously. This model is the foundation of the Mozart programming system, which implements Oz.*

MOTS-CLÉS : *système Mozart, programmation distribuée, protocoles distribués, objets mobiles, langage Oz, langages data-flow*

KEY WORDS : *Mozart system, distributed programming, distributed protocols, mobile objects, Oz language, data-flow languages*

1. Introduction

Dans l'état actuel de l'art, le développement d'une application distribuée nécessite une connaissance additionnelle par rapport au développement d'une application sur une seule machine. La principale caractéristique d'un calcul distribué est son partitionnement sur un ensemble de sites. Par exemple, une architecture client-serveur peut être écrite avec Java RMI [2]. Une application existante peut être connectée avec une autre application en utilisant une implémentation CORBA [9]. Dans les deux cas, ces outils ne sont pas satisfaisants.

En effet, réorganiser la structure de distribution nécessite la réécriture de l'application. En outre, chaque nouveau problème adressé, tel que la conception d'une nouvelle structure de distribution ou l'ajout de tolérance aux pannes à l'application, augmente la complexité de l'application. Un développeur ayant uniquement une expérience dans les systèmes centralisés n'est donc pas préparé pour écrire des applications distribuées. Dans le cadre du langage Distributed Oz [14], notre approche est de séparer la fonctionnalité de l'application des aspects de la structure de distribution, de tolérances aux pannes et de sécurité. Ces derniers aspects doivent être des additions orthogonales à la fonctionalité de l'application. Cette approche est en cours de recherche et de développement.

Les principales contributions de l'article sont d'une part un protocole de gestion d'adressage de toutes les entités du langage Distributed Oz entre un ensemble de sites et d'autre part un protocole distribué pour les données sans état. Le lecteur peut consulter les articles suivants [4, 12, 14] pour des compléments d'informations sur l'implémentation du langage et sur une description d'exemples d'applications distribuées et collaboratives réalisées en Distributed Oz telle qu'un éditeur graphique distribué [17, 14].

La section 2 résume le modèle d'exécution d'Oz. La section 2.1 présente l'exécution d'Oz mis sous forme de graphe, sans la notion de répartition. La section 2.2 étend ce graphe pour exprimer la répartition du programme. Dans le graphe distribué, une entité répartie est représentée par un ensemble de nœuds, appelé sa structure d'accès. La section 3 présente l'architecture globale de Mozart, l'implémentation répartie d'Oz. Cette implémentation est définie par des protocoles répartis entre les nœuds du graphe distribué. Chaque entité du langage a son propre protocole distribué entre les nœuds formant sa structure d'accès. Nous allons voir dans la section 4 qu'un nœud supplémentaire, le nœud site, permet d'exprimer la création de la structure d'accès d'une entité. Cette section présente notre contribution avec un protocole global responsable de la gestion de l'adressage global des données. Dans la section 5, nous proposons la formalisation et l'implantation d'un protocole de données sans état. Dans la section 6, nous comparons le protocole d'adressage de données avec des travaux connexes. Nous concluons dans la section 7.

2. Distributed Oz et le graphe du langage

Nous présentons un aperçu du langage centralisé Oz. Le langage Oz est basé sur un modèle d'exécution concurrent avec une sémantique entrelacée [4, 13] . Ce langage présente trois innovations. Premièrement, il utilise la synchronisation data-flow par la notion de données à affectation unique (variables logiques) comme mécanisme de base de contrôle. Deuxièmement, il fait une distinction claire entre les données avec état (les cellules) et les données sans état (procédures et enregistrements). Enfin, le modèle unifie la programmation fonctionnelle, la programmation logique et la programmation orienté-objet.

Les exécutions du langage sont définies en terme d'un noyau de langage détaillé dans [13, 19, 18, 4]. La figure 1 définit la syntaxe abstraite d'une instruction *S* dans le noyau du langage Oz. Nous distinguons cinq types de données. Les *variables logiques* déclarées avec une portée explicite par l'instruction **`local`**. Les *valeurs* (enregistrements, nombres, noms, etc.) sont introduites explicitement et les *procédures* sont définies à l'exécution par l'instruction **`proc`**. Un nom est créé par l'appel `NewName`. Ce nom est une constante unique dans le système. Une *cellule* ou état est créée par `NewCell`. Cette cellule est un pointeur vers une variable. Les cellules sont mises à jour par `Exchange` et lues par `Access`. Les

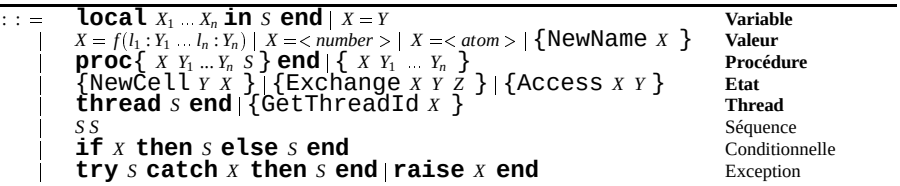

Figure 1. Le noyau du langage Oz

threads sont créés par l'instruction **`thread`**. Chaque thread a un identificateur unique.

Les autres instructions définissent le flux de contrôle associé au langage. Les instructions dans une *séquence* sont réduites séquentiellement dans un thread. L'instruction **`if`** définit une *conditionnelle* dont la condition bloque jusqu'à ce qu'elle soit vraie ou fausse. L'instruction **`try`** définit une portée et l'instruction **`raise`** déclenche une *exception*. D'autres types de données, définies avec les instructions de base du noyau, existent dans Oz tels que les objets, les classes, les verrous réentrants, et une variété de canaux appelée ports [13, 4].

2.1. *Le graphe du langage*

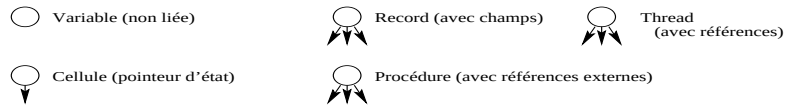

Figure 2. Les entités du langage comme nœuds de graphe

On représente un programme Oz et ses données par un graphe. Une exécution est modélisée par des transformations de graphe. Chaque entité du langage correspond à un nœud du graphe du langage. La figure 2 détaille les différentes entités sous forme de nœuds d'un graphe. Le langage de graphe est décrit indépendamment de toute notion de répartition des nœuds entre un ensemble de sites.

2.2. *Le graphe distribué du langage*

L'implémentation répartie d'Oz intègre des protocoles répartis qui implémentent les entités du langage. L'exécution répartie dans Distributed Oz est représentée par une transformation de graphe. Le graphe distribué du langage introduit la notion de *site* associé au nœud du graphe du langage. Nous supposons un ensemble fini de sites et nous annotons chaque nœud par son site. Si un nœud est référencé par un nœud d'un autre site, alors ce nœud est transformé en un ensemble de nœuds. Cet ensemble est appelé une *structure d'accès* pour ce nœud.

Dans la figure 3, le nœud `N2` est référencé par les sites 1 et 3. Cette référence est transformée en une référence à l'ensemble des nœuds `{P1,P2,P3,M}` de la structure d'accès. Une structure d'accès est composée d'un *nœud proxy* `Pi` pour chaque site référençant le nœud original et d'un *nœud manager* `M` pour la structure globale. Le graphe résultant incluant les

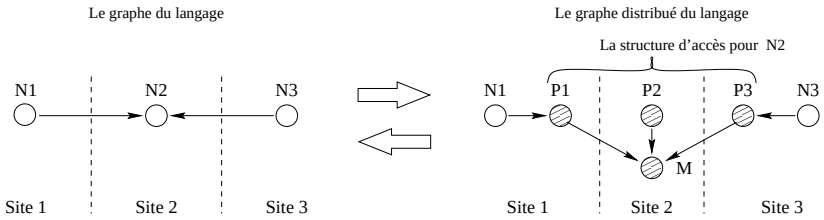

Figure 3. Une structure d'accès dans le graphe distribué

nœuds locaux et les structures d'accès est appelé graphe distribué.

C'est au niveau de ce graphe distribué qu'une exécution distribuée est modélisée. Le comportement distribué d'une entité du langage (nœud du graphe) est définie par un protocole distribué entre les nœuds de la structure d'accès. Dans [4, 12] et la section 5, nous prouvons formellement que la sémantique distribuée de chaque entité du langage respecte sa sémantique centralisée . Ceci exprime une des caractéristiques fondamentales de Distributed Oz appelée la *transparence vis-à-vis du réseau*.

3. La mise en œuvre de Distributed Oz

Cette section résume l'architecture globale du système Mozart sur un site. Cette présentation a pour but de situer les protocoles, détaillés dans les sections suivantes par rapport à l'ensemble du système. Cette architecture, décrite dans la figure 4, est totalement implantée [14] et elle est utilisée pour le développement d'applications réparties [17].

Cette architecture est organisée en couches superposées. Chaque couche implémente des opérations en utilisant les fonctionnalités offertes par la couche inférieure et fournit à la couche supérieure une interface à ces opérations. Les entrées et les sorties de chaque couche sont indiquées en italique dans la figure 4. Nous présentons brièvement les composantes de chaque couche ainsi qu'une description sommaire de leur fonctionnalités.

- *Application*: elle représente le byte-code de l'application à exécuter.
- *La couche de graphe*: elle est composée de deux parties: la couche de graphe du langage ou émulateur et de la couche de graphe distribué du langage. Elle offre une abstraction vis-à-vis de l'exécution des opérations sur les entités du langage. En effet, l'opération sur une entité distribuée du langage est vue comme une opération sur une entité centralisée par la couche application. Nous disons alors que le langage possède la propriété de la *transparence vis-à-vis du réseau*.
- *La couche graphe de langage ou émulateur*: cette couche implémente les opérations sur chaque entité du langage sur le graphe du langage. L'émulateur est basé sur la technologie de byte code et a des performances similaires et même meilleures que les émulateurs Java actuels [11]. En particulier les threads Oz sont beaucoup plus 'légers' que les threads Java.
- *La couche de graphe distribué du langage*: cette couche implémente la sémantique distribuée de chaque entité. Un protocole distribué est associée à chaque entité. Nous dis-

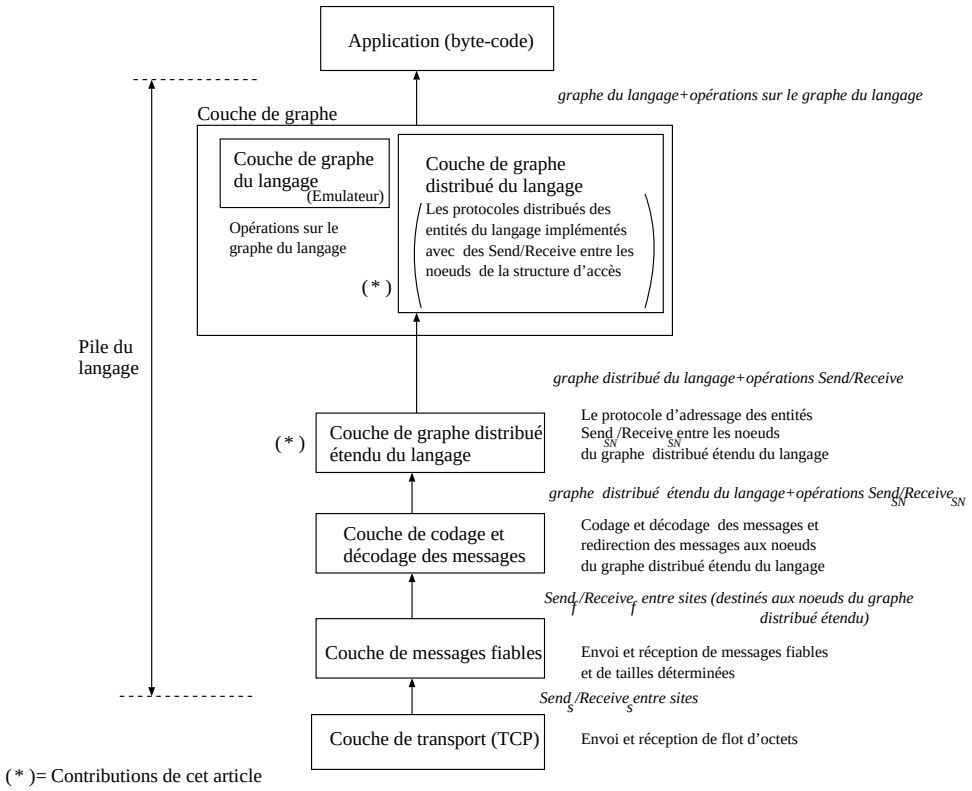

Figure 4. L'architecture du système sur un seul site

tinguons trois catégories d'entités:

- **Données sans état**: enregistrements (records), nombres, procédures, classes. Ces entités ne changent pas, elle peuvent donc être copiées entre les sites. La section 5 présente en détail le protocole distribué de ces entités.
- **Affectation unique**: variables logiques. Le comportement distribué est décrit par un algorithme efficace d'unification distribuée [12]. Pour être précis, les variables logiques fournissent une affectation multiple consistante, c'est-à-dire, les affectations doivent être consistantes entre elles. Nous gardons la terminologie à affectation unique pour des raisons de simplification d'appelation.
- **Données avec état**: cellules, objets, ports, verrous. Un protocole distribué d'état mobile assure que la mise à jour de cet état est vue de manière cohérente par l'ensemble des sites. L'article [4] décrit ce protocole et son utilisation pour les objets mobiles.

Chaque protocole est constitué d'un ensemble d'actions sur les nœuds de la structure d'accès du graphe distribué et un ensemble d'opérations *Send/Receive* entre ces nœuds. Ce protocole offre une sémantique distribuée de l'entité qui implémente exactement la sé-

mantique centralisée.

• *La couche de graphe distribué étendu du langage*: elle fait l'objet de la section 4 de l'article. Elle s'occupe de l'adressage des entités au niveau des sites et de la gestion des structures d'accès. Elle offre une couche de gestion d'adressage des données sur l'ensemble des sites. Elle est définie par un protocole sur un graphe distribué étendu du langage avec un nœud appelé *SiteNode*. Elle offre l'implémentation des opérations *Send*/*Receive* à la couche de graphe distribué du langage.

• *La couche de codage et de décodage des messages*: Elle implémente le codage et le décodage des messages et s'occupe de la redirection des messages aux nœuds du graphe distribué étendu.

• *La couche de messages fiables*: elle implémente le transfert de messages de longueur déterminée entre les sites. Elle est responsable aussi de la redirection des messages vers nœuds du graphe distribué étendu du langage.

• *La couche transport*: elle représente l'interface réseau pour le système d'exploitation du site. Cette interface offre un protocole standard de communication tel que le TCP/IP.

4. Le protocole d'adressage des données de Distributed Oz

Le graphe distribué du langage présenté à la section 2.2 modélise les exécutions distribuées. Les structures d'accès sont créées automatiquement lorsque les nœuds du graphe possèdent des références distantes à partir d'autres sites. Un protocole distribué implante la sémantique distribuée de chaque entité du langage. Ce protocole est décrit par des actions associées aux nœuds de la structure d'accès et des messages envoyés entre ces nœuds. Il implante exactement la sémantique centralisée de l'entité du langage. Par exemple, l'opération `{Exchange C X Y}` sur une cellule a la même comportement qu'elle soit réalisée sur le site local où se trouve initialement la cellule `C`, ou qu'elle soit réalisée sur un site distant possédant une référence vers `C`.

Nous détaillons dans cette section le protocole de gestion de la structure d'accès ainsi que le support d'adressage des entités sur l'ensemble des sites. Les nouvelles entités créées au sein d'un site sont toujours locales. Quand un message contenant une référence à une entité locale est envoyé vers un site donné, une entité distribuée est créée. Nous disons alors que l'entité locale est *globalisée*. Le message peut contenir un sous-graphe du graphe distribué. A l'envoi du message, chaque référence locale dans le message doit être *globalisée* avant l'envoi du message. A la réception du message, un nouveau nœud nommé proxy est créé pour cette entité. Nous montrons comment cette opération de *globalisation* fait partie des opérations *Send* et *Receive* des protocoles distribués de Distributed Oz.

4.1. *L'adressage via la structure d'accès*

Un schéma à deux niveaux d'adressage est utilisé pour référencer les nœuds du graphe. Le premier au niveau d'un site donné utilise un adressage local des nœuds. Le deuxième niveau utilise un adressage global des nœuds sur l'ensemble des sites. La transformation entre adresse locales et globales est réalisée d'une manière totalement automatique en maintenant les invariants suivants: Les nœuds d'un même site ont toujours une référence par des adresses locales. Les nœuds d'un site distant ont toujours une référence par des adresses

globales. Ces adresses globales ne sont jamais changées car aucun nœud du graphe distribué ne se déplace d'un site à un autre [1]. Un nœud ayant une référence distante doit avoir une adresse globale. Un nœud possède une référence distante si et seulement si, il est référencé par un autre site ou par un message en transit. Si un nœud n'a plus de référence distante, alors son adresse globale est récupérée.

Chaque structure d'accès a un nom noté *GlobalName*. Ce nom global est unique sur l'ensemble du système. Ce nom global encode le site du manager M. Un nœud proxy est identifié par la paire *(GlobalName,SiteId)* où *SiteId* représente un identificateur du site du proxy et *GlobalName* le nom de la structure d'accès. Un pointeur réseau est une adresse d'un nœud manager ou l'adresse d'un nœud proxy. Le tableau suivant détaille la représentation des adresses des nœuds managers et des nœuds proxies.

Description	Nom	Représentation
Nom de la structure d'accès	*GlobalName*	*(Index,SiteId)*
Pointeur réseau vers un manager	*GlobalAddr*	*GlobalName*
Pointeur réseau vers un proxy	*GlobalAddr*	*(GlobalName,SiteId)*

Les nœuds managers sont indéxés dans une table notée *ManagerTab*. Cette table associe pour chaque *GlobalName* l'adresse locale du manager. *LocalAdrr* désigne une adresse locale. Nous définissons une autre table notée *ProxyTab*. Cette dernière associe à chaque adresse globale *GlobalName* une adresse locale qui correspond à un proxy. Ces tables sont notées respectivement *ProxyTab: GlobalName* $\rightarrow$ *LocalAddr* et *ManagerTab: GlobalName* $\rightarrow$ *LocalAddr*.

4.2. *La notion de* SiteNode

Nous étendons le graphe distribué par l'ajout d'un nœud unique par site appelé *SiteNode*. Ce nœud particulier réalise la *globalisation* d'un nœud local. Ce *SiteNode* contient donc les deux tables *ProxyTab* et *ManagerTab*. Un protocole distribué de gestion d'adressage des nœuds est définie par deux règles détaillées dans la section suivante. Le ramassage de miettes ("garbage collection") peut engendrer l'inverse de la globalisation, c'est-à-dire la *localisation* d'une entité du langage. Ceci fait disparaître la structure d'accès et le nœud Proxy devient de nouveau un nœud local.

La figure 5 détaille la correspondance entre la couche application, la couche de graphe distribué du langage et la couche de graphe distribué étendu. Un protocole distribué d'une entité du langage est défini au niveau de la couche de graphe distribué. Le protocole est défini par un ensemble d'actions associées aux nœuds de la structure d'accès et un ensemble de messages entre ces nœuds. A ce niveau d'abstraction, on ne se préoccupe pas du problème d'adressage des nœuds du graphe. C'est seulement au niveau de la couche inférieure que l'adressage global des nœuds est décrit. Les opérations *Send*/*Receive* sont implémentées par des opérations $Send_{SN}/Receive_{SN}$. Le protocole de gestion des structures d'accès est décrit à la section 4.3. Ce protocole utilise un nouveau nœud appelé *SiteNode* ajouté au graphe distribué du langage. Le graphe résultant est appelé graphe distribué étendu. La figure 5 montre aussi les trois niveaux d'abstraction possibles pour l'exécution d'une opération sur une entité *N*. Au niveau supérieur, elle fait partie du graphe du langage. Au deuxième niveau elle

1. Néanmoins, la mobilité existe [4].

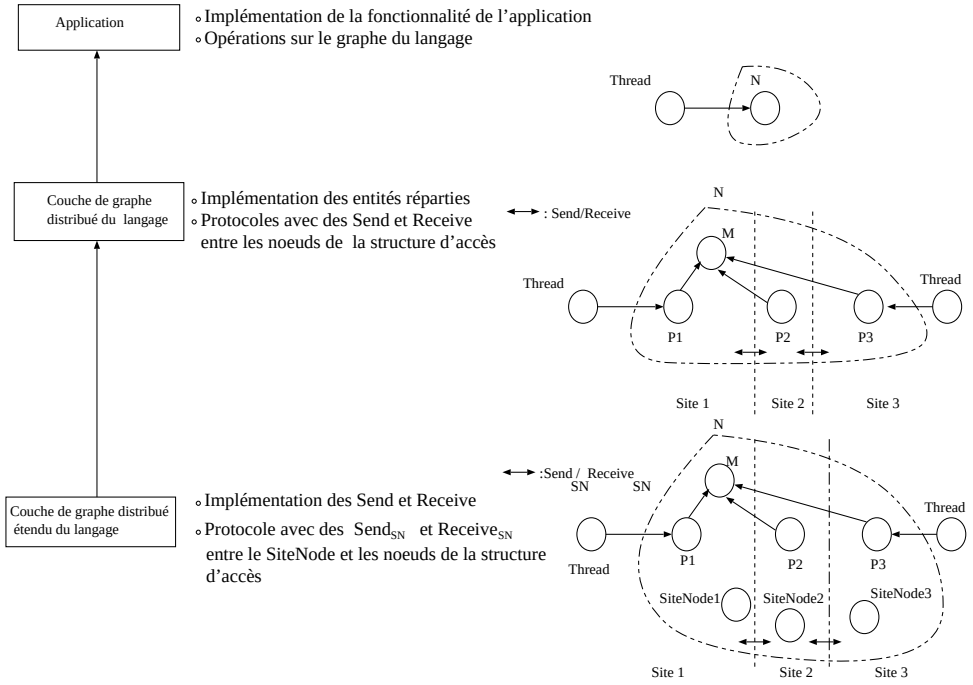

Figure 5. Les différents niveaux de l'exécution d'une opération

est implémentée par un protocole distribué opérant sur le graphe distribué du langage. Au niveau inférieur, elle est réalisée au sein d'un protocole de gestion d'adressage des nœuds du graphe distribué étendu.

Nous décrivons maintenant les procédures de transformation d'adresses des nœuds du graphe distribué du langage. Un nœud du graphe distribué possède les attributs suivants:

Attributs	type
manager	*GlobalAddr* (uniquement pour les Proxy)
type	*Local, Proxy, Manager*
oztype	*Record, Procedure, Thread, Cell, Variable*
content	*vide ou LocalAddr* (selon type du nœud)
refs	*liste des références* (liste de sous-graphes) (uniquement pour les Local)

La procédure **Replace** de la figure 6 transforme une adresse locale *laddr* vers un nœud en un nom global *gname*. Toute adresse locale contenue dans un message est transformée en un nom global. A l'arrivée du message, un nom global contenu dans le message est transformé en une adresse locale sur le site d'arrivée. *Node(laddr)* désigne la structure du nœud référencé par *laddr*. La procédure duale **Replace'** de la figure 6 transforme un nom global en une adresse locale.

```
define  Replace(in SiteNode,in laddr,out gname)
  if  Node(laddr).type=Proxy
  then  gname:=Node(laddr).id
  else
   (globalname,localaddr):=CreateManager()
   Node(laddr).type:=Proxy
   Node(laddr).id:=globalname
   SiteNode.ManagerTab:=SiteNode.ManagerTab ∪ (globalname,localaddr)
   SiteNode.ProxyTab:=SiteNode.ProxyTab ∪ (globalname,laddr)
   gname:= globalname
  fi
```

```
define  Replace'(in SiteNode,in gname,out laddr)
    if  not  In(SiteNode.ProxyTab(gname))  then
     laddr:=CreateProxy(gname)
     SiteNode.ProxyTab:=
          SiteNode.ProxyTab ∪(gname,laddr)
    else
     laddr:=SiteNode.ProxyTab(gname)
    fi
```

Figure 6. Définition de ***Replace*** et de ***Replace'***

4.3. *Le protocole d'adressage des données*

Nous considérons respectivement un nœud *SiteNode* unique par site, un nœud manager *M*, un ensemble de nœuds proxy P_i avec $1 \leq i \leq k$, et un ensemble de nœuds thread T_i avec $1 \leq i \leq m$. Chaque nœud peut réaliser des opérations locales. Ces nœuds sont liés par un réseau *N* modélisé par un multi-ensemble contenant des messages de la forme $d : m$ où *d* identifie une destination (*SiteNode*, proxy, manager, ou thread) et *m* un message. *SNode(N)* dénote le *SiteNode* du site qui contient le nœud N.

Nous décrivons le protocole distribué d'adressage de Distributed Oz. Ce protocole est défini par des $(Send_{SN}/Receive_{SN})$ entre les *SiteNode* et les nœuds de la structure d'accès et par un ensemble d'actions associées au nœud *SiteNode*.

Le tableau suivant définit l'implantation des opérations $Send/Receive$ entre les proxies $P1$ et $P2$ de la couche du graphe distribué par des opérations $Send_{SN}/Receive_{SN}$ de la couche de graphe distribué étendu. Un message envoyé par $P1$ vers $P2$ est implanté par trois envois de messages (deuxième colonne du tableau).

Messages de la couche de graphe distribué	Messages de la couche de graphe distribué étendu
P1 exécute Send(P2,msg(Content))	*1- P1 exécute* $Send_{SN}$*(SNode(P1),msg(P2,msg(Content)))* *2- SNode(P1) exécute* $Send_{SN}$*(SNode(P2),msg(P2,msg(Content')))* *3- SNode(P2) exécute* $Send_{SN}$*(P2,msg(Content"))*
$Receive(GlobalAddr, msg(Content))$	$Receive_{SN}(GlobalAddr, msg(Content))$

SNode(P1) désigne le *SiteNode* du site qui contient *P1*. *Content* est une référence à un graphe *G* à envoyer d'un site 1 à un site 2. *"msg"* est un constructeur utilisé pour emballer le message originale, *Content*, dans la couche de graphe étendu. $Content'$ est une référence à un graphe G' obtenu à partir de *G* où chaque référence locale d'un nœud de *G* est transformée en une référence globale. $Content''$ est une référence au graphe G'' au site 2 obtenu à partir de G' où chaque référence globale d'un nœud de G'' est transformée en une référence locale su site de $P1$.

4.3.1. *Les règles du protocole d'adressage des données*

Un protocole est défini par un ensemble de règles de réduction de la forme: $\frac{\text{Condition}}{\text{Action}}$

Chaque règle est définie dans le contexte d'un seul nœud. L'exécution suit un modèle entrelacé. A chaque pas d'exécution, une règle avec une condition valide est sélectionnée.

Les actions sont exécutées atomiquement. Une condition d'une règle consiste en des conditions booléennes sur l'état du nœud et une condition optionnelle de réception de message *Receive*(*d*,*m*). La condition *Receive*(*d*,*m*) signifie que *d* : *m* est arrivé à *d*. L'exécution d'une règle avec une condition de réception de message supprime le message *d* : *m* du réseau et exécute la partie action de la règle.

Une action d'une règle consiste en une séquence d'opérations sur l'état du nœud avec des opérations optionnels d'envoi de messages. L'action *Send*(*d*,*m*) envoie d'une manière asynchrone un message *m* au nœud *d*, c'est-à-dire ajoute le message *d* : *m* au réseau. L'opération *Receive*(*d*,*m*) bloque jusqu'à qu'un message arrive à *d*. Quand le message arrive, il est supprimé du réseau et l'exécution continue.

Nous supposons que le réseau et les nœuds sont équitables dans le sens suivant. Le réseau est asynchrone et les messages destinés à un nœud donné prennent un temps fini arbitraire et peuvent arriver dans un ordre quelconque. Toute règle qui est infiniment souvent applicable sera tôt ou tard réduite.

---

(1) Réception des messages au *SiteNode*

$$\frac{Receive_{SN}(SiteNode, msg(GlobalAddr, Content))}{Send_{SN}(SNode(GlobalAddr), Map(Content, Replace(SiteNode)))}$$

(2) Envoi de *SiteNode* à *SiteNode*

$$\frac{Receive_{SN}(SiteNode, msg(GlobalAddr, Content'))}{Send_{SN}(GlobalAddr, Map(Content', Replace'(SiteNode)))}$$

Figure 7. Les règles du protocole d'adressage

La fonction *Map* transforme toutes les références de tous les nœuds d'un graphe en conservant la structure du graphe vis-à-vis de l'attribut *refs* d'un nœud. Le protocole de la figure 7 détaille le traitement des messages au *SiteNode*. La règle (1) applique la fonction *Map* au sous-graphe référencé par *Content*. Après cette *globalisation*, le sous-graphe obtenu est envoyé au nœud *SNode(GlobalAddr)*. La règle (2) réalise l'opération duale de la règle (1).

5. Le protocole des données sans état

Cette section détaille le protocole distribué associé aux donnés sans état (les procédures et les enregistrements). La section 3 présente les autres protocoles distribués, associés aux données avec états et à affectation unique. Le protocole proposé préserve la sémantique centralisée du langage. Nous rappelons la sémantique centralisée des différentes opérations associées aux données sans état et nous étendons cette sémantique en une sémantique distribuée simple. Nous présentons ensuite un protocole distribué. Enfin, nous caractérisons certaines exécutions distribuées. En particulier, diverses stratégies de copie de procédures entre les sites sont détaillées. Toutes les stratégies satisfont le protocole distribué. Ce qui les différencie est le "scheduling" des règles de réduction.

5.1. *Le contexte général*

Toutes les exécutions sont décrites par la réduction de règles de transition. L'exécution d'instruction est une opération atomique décrite par une règle dont la syntaxe respecte le schéma: $\frac{(\textit{instructions})}{(\textit{store})} \,\|\, \frac{(\textit{nouvelles instructions})}{(\textit{nouveau store})}$

La règle devient applicable pour une instruction (opération) quand le store courant correspond au store présent dans la règle. Nous rappelons que le store est un espace partagé comportant trois compartiments: les variables logiques avec leur liaison (si elles sont liées) notée $X \leftarrow V$, les procédures et les cellules. Puisque le langage est concurrent, la réduction est en général non-déterministe. L'équité entre threads implique qu'une règle a la garantie d'être appliquée tôt ou tard.

Soit un ensemble de sites indexés par des entiers $1,\ldots,k$. Le store distribué est noté $\sigma \equiv \sigma_1 \wedge \ldots \wedge \sigma_k$. σ_i dénote le store au site i. S désigne un ensemble d'instructions. Une configuration distribuée est notée $C = (S,\sigma)$. Une configuration dans un site i est notée $C_i = (S_i,\sigma_i)$. S_i désigne un ensemble d'instructions d'une configuration au site i. L'effet d'une réduction est le remplacement d'une configuration par une autre configuration. Pour simplifier les notations, l'indice i peut être omis dans le cas où l'ensemble des sites est réduit à un singleton.

5.2. *La sémantique centralisée*

Nous décrivons tout d'abord la sémantique centralisée des procédures. Il s'agit donc de définir la sémantique de deux opérations; la définition et l'application de procédures. Dans Oz, aussi bien la définition que l'application de procédures est réalisée dynamiquement en cours d'exécution.

La règle R^n_{def} exprime la sémantique de la définition de P. $\overline{X}$ dénote un vecteur de variables $X_1 \ldots X_n$ (liste des arguments formels de P). L'effet de l'exécution de **proc**$\{P\,\overline{X}\,\}\,E$ **end** engendre l'instruction $P = n$. Cette action essaye de lier à la variable P un nouveau nom noté n (définition en cours d'exécution) et ajoute au store le code de P noté $\overline{X}/E$ en indiquant le nom correspondant. Il est à noter ici que le nom n est unique dans tout le système. L'action $P = n$ peut "échouer" et engendrer ainsi une exception si P est déjà lié à une autre valeur que n puisque P se comporte comme une variable logique (donnée à affectation unique).

$$R^n_{def}\ \frac{\textbf{proc}\ \{\,P\,\overline{X}\,\}\,E\ \textbf{end}}{\sigma} \,\|\, \frac{P = n\ (n \text{ nouveau nom})}{\sigma \wedge (n : \overline{X}/E)} \qquad R^n_{app}\ \frac{\{P\,\overline{Y}\}}{\sigma \wedge (P \leftarrow n) \wedge (n : \overline{X}/E)} \,\|\, \frac{E[\overline{Y}//\overline{X}]}{\sigma \wedge (P \leftarrow n) \wedge (n : \overline{X}/E)}$$

La règle R^n_{app} exprime la sémantique de l'application de P. L'application d'une procédure est exprimée par l'instruction $\{P\,\overline{Y}\}$. Il faut vérifier d'abord que le store contienne le code de nom n et aussi que la variable P soit liée à n. Dans ce cas, l'instruction est réduite et tous les arguments formels de P sont substituée par Y. L'opération de substitution multiple est notée $//$. Le store reste inchangé après l'application d'une procédure. L'accès à un enregistrement est exprimé par l'application d'une procédure définie. Elle respecte donc la même sémantique que les procédures.

5.3. *La sémantique distribuée*

Nous présentons dans ce paragraphe une sémantique distribuée pour la définition et l'application de procédures. Intuitivement, puisqu'il s'agit de données sans état, les procédures et les enregistrements peuvent être dupliqués sur un ensemble de sites en préservant la cohérence du système. Un indice est ajouté à chaque instruction. Cet indice représente le numéro du site où l'instruction est exécutée. Les règles locales de définition de procédures et d'application restent inchangées sur un site i. Ces règles sont notées respectivement $R^{n,i}_{def}$ et $R^{n,i}_{app}$.

$$R^{n,i}_{def}\ \frac{(\mathbf{proc}\ \{\ P\ \overline{X}\ \}\ E\ \mathbf{end})_{\mathbf{i}}}{\sigma}\ \Big\|\ \frac{(P=n)_i}{\sigma\wedge(n:\overline{X}/E)_i} \qquad R^{n,i}_{app}\ \frac{\{P\ \overline{Y}\}_i}{\sigma\wedge(P\leftarrow n)_i\wedge(n:\overline{X}/E)_i}\ \Big\|\ \frac{E[\overline{Y}/\overline{X}]_i}{\sigma\wedge(P\leftarrow n)_i\wedge(n:\overline{X}/E)_i}$$

La règle suivante définit la copie d'une procédure d'un site i à un site j différent de i. Nous notons ici qu'aucune hypothèse sur la présence ou non du code de n n'est explicitée. Cette sémantique n'interdit pas qu'une copie d'une procédure soit envoyée plusieurs fois à un site.

$$R^{i,j}_{copy}\ \frac{\mathbf{skip}}{\sigma\wedge(P\leftarrow n)_i\wedge(P\leftarrow n)_j\wedge(n:\overline{X}/E)_i, i\neq j}\ \Big\|\ \frac{\mathbf{skip}}{\sigma\wedge(P\leftarrow n)_i\wedge(P\leftarrow n)_j\wedge(n:\overline{X}/E)_j,(n:\overline{X}/E)_i\ , i\neq j}$$

5.4. *L'algorithme distribué*

Cette section présente un protocole distribué implantant la sémantique distribuée définie précédemment. Nous utilisons les notations de la section précédente. Nous supposons un ensemble de nœuds manager M_i, $1 \leq i \leq k$, et un ensemble de nœuds proxy P_i,$1 \leq i \leq k'$. L'implémentation du protocole est décrite au niveau de la couche graphe du langage distribué. Le tableau de la figure 8 décrit les attributs des nœud Manager et des nœuds Proxy. Nous rappelons que le protocole suppose que la structure d'accès est déjà créée. Nous nous limitons à la présentation des attributs des nœuds uniquement utilisés par le protocole. La figure 8 détaille l'ensemble des règles du protocole de copie de données sans état. La règle (1) demande la copie de code au manager par le message *requestcode*. A la réception du message *requestcode* au manager, le contenu de proxy associée au manager (le code) est copié (règle (2)). La règle (3) met à jour les attributs du nœud proxy. Le nœud change de type et devient local.

Interface avec les threads

La règle (1) se déclenche dès que *P.content=Empty*. Donc, elle est applicable dès la création de P. Il s'agit d'une stratégie de copie ardente. La section 5.5 détaille plusieurs stratégies associées aux exécutions distribuées. Nous pouvons modifier cette règle pour réaliser une copie uniquement à la demande d'un thread.

$$\frac{P.content=Empty\wedge\ Receive(T,request)}{Send(P.manager,requestcode(P))}$$

La règle $\frac{P.content\ \neq Empty}{Send(T,proceed)}$ réactive le thread en attente en envoyant un message *proceed*.

Un thread exécutant l'application d'une procédure dont le code est vide se bloque. La figure 9 suivante illustre trois étapes notées 1), 2) et 3) d'application du protocole. Un proxy $P2$ demande le code de n puis un proxy $P3$ demande le code de n. Deux structures d'accès

(1) Demande de code de procédure par P

$$\frac{P.content=Empty}{Send(P.manager,requestcode(P))}$$

(2) Traitement du message au manager M

$$\frac{Receive(M,requestcode(P))}{Send(P,M.content)}$$

(3) Réception du code à P

$$\frac{Receive(P,Content)}{\begin{array}{l}P.content := Content\\ P.type := Local\end{array}}$$

Manager M

Attribut	Valeur initiale
content	*(la procédure)*

Proxy P_i

Attribut	valeur initiale
manager	*(son manager)*
content	Empty
type	Proxy
oztype	Procedure

Figure 8. Les règles du protocole et les attributs d'un nœud

sont créées pour n. Cette redondance est utile pour remédier aux problèmes de pannes d'un site.

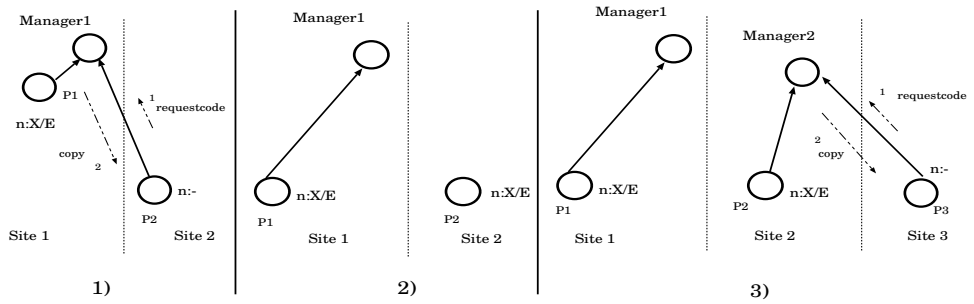

Figure 9. Exemple

5.5. *Les stratégies de copies paresseuse et ardente*

Dans cette section, nous caractérisons certaines exécutions distribuées. Les règles formelles de la sémantique distribuée définissent un cadre général de copie de données sans état. En pratique, nous nous intéressons à des propriétés sur ces exécutions. Par exemple, une propriété interéssante d'une procédure est de pouvoir l'envoyer au plus une fois à un site donné. Nous commençons par définir la notion de relation de causalité.

Relation de causalité

Pour simplifier les notations, nous supposons que les exécutions distribuées ne contiennent que des définitions, des applications et des copies procédures. Une exécution distribuée finie ou infinie à partir d'une configuration initiale C_0 est notée $D : C_0 \xrightarrow{r_1} \ldots \xrightarrow{r_i} C_n \ldots$. $i = 1,..,n,\ldots$ r_i dénote une règle de transition. Etant donnée une exécution D et $r, r' \in D$, $r \prec r'$ signifie que r précède r' dans D. Toute exécution distribuée D vérifie la relation de

causalité suivante:

(A) $\forall i, j, n, \forall R_{app}^{n,j} \in D, \exists R_{def}^{n,j} \in D \vee \exists R_{copy}^{n,i,j} \in D, \quad (R_{copy}^{n,i,j} \vee R_{def}^{n,j}) \prec R_{app}^{n,j}$

La relation (A) exprime le fait que l'application d'une procédure au site j est précédée obligatoirement par sa définition au site j ou par une copie du code d'un site i au site j. Plusieurs stratégies peuvent être associées à la copie des données sans état entre sites. Nous distinguons trois stratégies principales de copies:

— *La stratégie paresseuse ("lazy")*: la copie est réalisée au moment de l'utilisation de la donnée. Soit un thread possédant une référence distante à une procédure de nom *n*. Au moment de l'application de la procédure par ce thread, le code de *n* est inexistant dans le site du thread. Une demande d'une copie de code de *n* est alors envoyée au site contenant le code de *n*. La stratégie paresseuse implique donc une réplication du code de la procédure à la demande.

— *La stratégie ardente immédiate ("eager immediate")*: la copie est réalisée au moment de la *globalisation de la donnée* (section 4.3.1). Dans ce cas, nous disons qu'il s'agit d'une stratégie ardente immédiate. C'est le cas par exemple des enregistrements, qui n'ont pas de références globales (pas de structure d'accès). Ils sont répliqués lors de la globalisation de leur référence.

— *La stratégie ardente ("eager")*: la copie de la procédure est initiée à la réception de la donnée, c'est-à-dire, après l'arrivée d'une référence de procédure à un site qui n'a pas encore demandé cette procédure. Une demande de code est alors envoyée.

Quelques propriétés des stratégies

Selon le contexte, certaines stratégies sont plus utiles que d'autres. Nous nous intéressons essentiellement à deux propriétés:

– La propriété (B) exprime le fait que le code d'une procédure est envoyé au plus une fois à un site donné.

– La propriété (C) exprime une copie de code à la demande.

(B) $\forall i, i', j, n \ \forall R_{copy} \in D \quad R_{copy}^{n,i,j} \not\prec R_{copy}^{n,i',j}$

(C) $\forall i, j, n \quad (\exists\, r_{i'} \in D, \exists\, k \,/\, (r_{i'} \equiv R_{copy}^{n,i,j} \Rightarrow r_{i'+k} \equiv R_{app}^{n,j}))$

Ce paragraphe présente une caractérisation de l'ensemble des exécutions distribuées vérifiant la stratégie paresseuse et ardente. La relation (B) exprime le fait qu'une copie de *n* d'un site j à un site i est réalisée au plus une fois. Une exécution paresseuse ou ardente vérifie la relation (B). La relation (C) décrit la notion de copie de procédure à la demande (stratégie paresseuse). Une application de procédure est précédée par une copie de code uniquement à la demande (au cas où le code n'existe pas dans le site demandeur).

Le nombre d'opérations réseau pour copier une procédure par une stratégie paresseuse est égale à deux messages (section 5).

La stratégie ardente implique la copie d'une procédure *P* lors de la *réception* dans un site d'un message contenant une nouvelle référence à *P*. Elle possède la propriété suivante: une copie au plus d'une procédure est envoyée à un site. La relation (B) est ainsi vérifiée pour les exécutions distribuées ardentes. La stratégie ardente immédiate ne vérifie ni la propriété (B) ni la propriété (C) puisque la copie est envoyée lorsque la référence de la procédure est envoyé à un site. Mais elle possède la propriété de minimiser le nombre de messages

pour la réplication du code. Nous notons ici que c'est uniquement au niveau de la couche de graphe distribué étendu que nous pouvons distinguer la stratégie ardente de la stratégie ardente immédiate.

6. Travaux connexes

Le protocole de gestion d'adressage proposé est à comparer avec l'approche qui définit la notion de mémoire partagée distribuée *Distributed Shared Memory* (DSM) [5, 7]. Ce concept offre un mécanisme de gestion de mémoire pour la programmation parallèle mais plusieurs travaux utilisent ce mécanisme dans le contexte de la programmation distribuée. Afin d'avoir un comportement réseau prévisible des entités d'un langage distribué, celui ci doit être "compatible" avec la couche DSM. En effet, la couche DSM doit offrir un espace partagé d'adressage pour toutes les entités du langage tout en préservant le contrôle des opérations réseau de chaque entité.

Dans [5], nous distinguons entre les couches DSM basées sur la notion de pages et celles basées sur la notion de librairies. Les couches DSM basées sur la notion de pages ne permettent pas de prévoir le comportement réseau des entités puisque les pages ne correspondent pas directement aux entités du langage. Dans cette approche, Munin [6], fournit au programmeur des annotations pour contrôler les opérations réseaux de certaines entités.

Les couches DSM basés sur les librairies fournissent la notion de partage pour des données et peut ainsi éviter les problèmes de granularité et de faux partage. Par exemple, Orca [8] fournit des pointeurs réseau, appelés *"graphes générales"*, et objets partagés. Orca a été conçu pour des applications parallèles. Linda [10, 1] offre des opérations pour insérer et extraire des données non mutables, appelées *"tuples"*, d'un espace partagé. Linda offre ainsi *un modèle de coordination*, et peut être ajouté à tout langage existant. Mais, Linda n'adresse pas le problème de la transparence de la distribution.

Distributed Oz suit l'approche basée sur les librairies. L'espace partagé est une couche DSM (section 4) qui supporte toutes les entités du langage sans un "faux partage". Cette couche est étendue pour fournir d'autres fonctionnalités ne faisant pas partie des couches DSM traditionelles. Premièrement, le langage offre des données à affectation unique (variables logiques) et des objets mobiles. Deuxièmement, le système est ouvert: les sites peuvent se connecter et se déconnecter dynamiquement. C'est une propriété importante du protocole d'adressage globale des données proposée à la section 4. Troisièmement, le système est portable sur une large variété de systèmes d'exploitation et de processeurs. Quatrièmement, le système a été étendu pour la gestion et la détection de pannes [15].

7. Conclusion

Nous avons vu la description d'un protocole distribué d'adressage global de différents types de données entre un ensemble de sites. Ce protocole sert d'interface aux différents protocoles distribués du langage Distributed Oz. Nous avons décrit les interactions entre ces protocoles et les entités du langage. Un protocole distribué de données sans état est détaillé ainsi qu'une formalisation de sa sémantique distribuée. Nous avons montré comment obtenir un langage de programmation distribué en préservant sa sémantique centralisée originale. L'implémentation de Distributed Oz nommée Mozart [14, 19, 18] est disponible et elle est

utilisée comme plate-forme pour la programmation distribuée.

Remerciements

Ce travail a été réalisé dans le cadre du projet `Pirates` financé par la Région Wallonne (Belgique). Les auteurs tiennent à remercier les lecteurs anonymes pour leurs critiques, leurs conseils et l'intérêt qu'ils ont porté à ces travaux. Ils tiennent également à remercier Raphaël Collet pour ses commentaires sur l'article.

Bibliographie

[1] Antony Rowstron and Alan Wood. An efficient distributed tuple space implementation for networks of workstations. Euro-Par'96.

[2] Sun Microsystems. *The Remote Method Invocation Specification*. Sun Microsystems, Mountain View, Calif., 1997. Available at `http://www.javasoft.com`.

[3] Luca Cardelli. A language with distributed scope. *ACM Transactions on Computer Systems*, 8(1):27–59, January 1995. Also appeared in POPL 95.

[4] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.

[5] Coulouris, G., Dollimore, J., and Kindberg, T. *Distributed Systems Concepts and Design 2nd ed.* Addison-Wesley, Reading, Mass.1994.

[6] Carter, J. B., Bennett, J. K., and Zwaenepoel, W. Implementation and performance of Munin. In the *13th ACM Symposium on Operating System Principles*. 1991.ACM, New York, 152–164.

[7] Tanenbaum, A. S. *Distributed Operating Systems.* Prentice-Hall, Englewood Cliffs, N.J. 1995.

[8] Bal, H. E., Kaashoek, F. E., and Tanenbaum, A. S. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng. 18*, 3 (Mar.1992.), 190–205.

[9] Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA: The Common Object Request Broker Architecture*. Prentice-Hall PTR, Upper Saddle River, N.J., 1996.

[10] Carriero, N. and Gelernter, D. Coordination languages and their significance. *Commun. ACM 35*, 2 (Feb. 1992.), 96–107.

[11] Martin Henz. *Objects in Oz.* PhD thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, June 1997.

[12] Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer and Gert Smolka. Efficient Logic Variables for Distributed Computing. *ACM Transactions on Programming Languages and Systems*, 1999, To appear.

[13] Gert Smolka. The Oz programming model. In *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[14] DFKI, SICS, UCL and UdS. The Mozart programming system (Oz 3), Available at `http://www.mozart-oz.org`, January 1999.

[15] Per Brand, Peter Van Roy, Raphaël Collet and Erik Klintskog. A Fault-Tolerant Mobile-State Protocol and Its Language Interface. In preparation 1999.

[16] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3), May 1998.

[17] Donatien Grolaux and Peter Van Roy A Collaborative Graphic Editor Based on Transactions. Talk at Oz/Mozart Workshop, June 1998. Unpublished.

[18] Peter Van Roy, Seif Haridi and Per Brand. Distributed Programming in Mozart – A Tutorial Introduction. In Mozart documentation, available at `http://www.mozart-oz.org`, 1999.

[19] Seif Haridi and Nils Franzén. Tutorial of Oz. In Mozart documentation, available at `http://www.mozart-oz.org`, 1999.