HAL
open science

Construction et configuration d'applications réparties

Luc Bellissard

To cite this version:

Luc Bellissard. Construction et configuration d'applications réparties. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 1997. Français. NNT : . tel-00004918

HAL Id: tel-00004918

https://theses.hal.science/tel-00004918

Submitted on 20 Feb 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Institut National Polytechnique de Grenoble
ENSIMAG

THESE

pour obtenir le grade de

DOCTEUR de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Discipline : Informatique, Systèmes et Communication

présentée et soutenue publiquement par

BELLISSARD LUC

le mardi 9 Décembre 1997

CONSTRUCTION ET CONFIGURATION D'APPLICATIONS REPARTIES

Directeur de thèse:

Pr. Michel Riveill

JURY

Pr. J.P. Verjus, INPG	(Président)
Pr. O. Nierstrasz, Université de Bern, CH	(Rapporteur)
Pr. P. Estraillier, Université Paris VI	(Rapporteur)
Pr. R. Balter, Université Joseph Fourier	(Examinateur)
Pr. M. Riveill, Université de Savoie	(Directeur de Thèse)

Remerciements

Je tiens à remercier toutes les personnes qui ont rendu cette thèse possible par leur aide et leurs contributions.

Mes premiers remerciements sont adressés à Jean-Pierre Verjus, professeur à l'Institut National Polytechnique de Grenoble et directeur de l'unité de recherche INRIA Rhône-Alpes, pour m'avoir fait l'honneur de présider ce jury.

Je remercie également Pascal Estraillier, professeur à l'Université Paris VI, et Oscar Nierstrasz, professeur à l'Université de Berne, pour avoir accepté d'être rapporteurs de mon travail et d'avoir apporté un jugement constructif.

J'adresse mes sincères remerciements à Roland Balter, professeur à l'Université Joseph Fourier et responsable du projet SIRAC, pour son soutien de tous les instants, son intérêt constant pour le sujet de ce travail et pour la motivation qu'il insuffle à l'ensemble de son équipe.

Michel Riveill, professeur à l'Université de Savoie, tient une place particulière dans ces remerciements pour l'encadrement de ce travail, sa confiance et son écoute aux angoisses d'un doctorant (c'est le terme consacré de nos jours).

La liste des personnes que j'aimerais remercier maintenant est longue, car l'ambiance chaleureuse au sein du projet SIRAC provient essentiellement de l'ensemble de ces membres. Je ne pourrais donc citer exhaustivement tout le monde, et je m'en excuse par avance, mais toutefois un certain nombre de personnes ont eu un rôle particulier tout au long de ce travail :

Fabienne Boyer, Jean-Yves Vion-Dury pour leur aide précieuse et le travail constructif que nous avons mené de front, ainsi que tous les participants au travail sur Olan, Marie-Claude et Vladimir, sans qui l'environnement de construction n'aurait atteint le niveau actuel,

André Freyssinet, Serge Lacourte et Marc Herrmann, nos partenaires industriels, avec qui je souhaite pouvoir continuer notre collaboration enrichissante et passionnée, ainsi que tout "essspécialement" Maria, la nouvelle venue,

Noël De Palma (dit nono), Denis et Sarah, la relève bouillonnante qui agrémente les soirées du projet de blagues irremplaçables,

Enfin Jacques Mossière (ah les cigares de Jacques...), Sacha Krakowiak et Xavier Rousset pour leurs conseils emplis de sagesse, Daniel pour sa connaissance des terres orientales et nordiques, et Béatrice pour résoudre nombre de tracasseries administratives.

Pour conclure, je garde une place toute particulière pour ma famille : France, ma fille Arielle et tous ses grand-parents.

Chapitre 1 Introduction 1

I. Motivation 1

II. Historique sur l'Intérêt de la Répartition 2

III. Programmation Constructive et Configuration d'Applications réparties 3

IV. Plan du manuscrit et Présentation du travail réalisé 4

Chapitre 2 Modèles et Principes de Construction et Configuration d'Applications Réparties 5

I. Problèmes 8

II. Langages de définition d'interfaces 9

II.1. Application au modèle Objet : CORBA 10
II.1.1. Principes élémentaire du modèle Objet 10
II.1.2. Exemple de CORBA 10
II.1.3. Avantages et Inconvénients pour la programmation répartie 13
II.2. Application au modèle Composant: COM/DCOM 14
II.2.1. Principes élémentaires du modèle Composant 14
II.2.2. Exemple de DCOM 16
II.2.3. Avantages et Inconvénients pour la programmation répartie 19
II.3. Conclusion 21

III. Langages d'interconnexion de modules 22

III.1. Polylith 22
III.1.1. Présentation 22
III.1.2. Exemple d'Utilisation 23
III.1.3. Avantages et Inconvénients 25
III.2. Conclusion 26

IV. Langages de Configuration 28

IV.1. CONIC 29
IV.1.1. Modèle de programmation 29
IV.1.2. Exemples d'Utilisation 30
IV.1.3. Avantages et Inconvénients 32
IV.2. Darwin 33
IV.2.1. Présentation 33
IV.2.2. Exemple d'Utilisation 36
IV.2.3. Avantages et Inconvénients 38
IV.3. Conclusion 40

V. Langages de description d'architecture 41

V.1. UniCon 42
V.1.1. Présentation 42
V.1.2. Exemple d'Utilisation 46
V.1.3. Avantages et Inconvénients 48
V.2. Rapide 49
V.2.1. Présentation 49
V.2.2. Exemple d'Utilisation 51
V.2.3. Avantages et Inconvénients 52
V.3. Wright 53

VI. Discussion 55

VI.1.1. Intégration de Logiciels 55

VI.1.2. Description de l'Architecture 56
VI.1.3. L'Adhérence aux ressources 58
VI.1.4. Génération d'Exécutables 58

Chapitre 3 Langage de Configuration OCL 61

I. Architecture générale 61

II. Le langage OCL 62

II.1. Choix de conception 62
II.2. Intégration de logiciels 63
II.2.1. Composant Primitif 63
II.2.2. Modules 66
II.2.3. Avantages et Inconvénients 69
II.3. Description de l'architecture 70
II.3.1. Composant Composite 70
II.3.2. Interconnexion de Composants et Connecteurs 71
II.3.3. Schémas d'instantiation dynamique 73
II.3.4. Interconnexions complexes 74
II.3.5. Avantages et Inconvénients 75
II.4. Adhérence aux Ressources 76
II.4.1. Attributs d'administration 76
II.4.2. Répartition des Composants 77
II.4.3. Règles de Décision 78
II.4.4. Avantages et Inconvénients 80
II.5. Génération de l'image exécutable 80

III. Comparaison avec les modèles et langages existants 83

III.1. CORBA et DCOM 83
III.2. Polylith 84
III.3. Darwin 84
III.4. UniCon 85
III.5. Rapide 86

Chapitre 4 Support de Configuration Olan 87

I. Architecture Générale 87

I.1. Principales Fonctions 87
I.2. Découpage Fonctionnel 88
I.3. Choix d'Implémentation 90

II. Mise en Œuvre des Composants 90

II.1. Principes de Fonctionnement 90
II.1.1. Module Logiciel vers le Composant 91
II.1.2. Composant vers un Composant 94
II.1.3. Autres Entités de OCL 94
II.2. Principes de Configuration 94

III. La Machine à Configuration 95

III.1. Interface de la Machine à Configuration 95
III.2. Déploiement 99
III.2.1. Choix du Site 100
III.2.2. Placement 101
III.3. Installation 102
III.4. Interconnexions 106
III.5. Scripts de Configuration 106
III.6. Scripts de Lancement 109

IV. Discussion 110

IV.1. Déploiement en environnement hétérogène 110
IV.2. Limites Actuelles 111

Chapitre 5 Exemples d'Utilisation 113

I. Application CoopScan 113

I.1. Présentation 113
I.2. Description OCL 115
I.3. Avantages et Inconvénients de l'Utilisation d'Olan 118
I.4. Réalisation 120

II. Application Mailer 121

II.1. Présentation 121
II.2. L'Environnement AAA (Agent Anytime Anywhere) 122
II.3. Description OCL 123
II.4. Avantages et Inconvénients de l'Utilisation d'Olan 128
II.5. Réalisation 129

III. Evaluation 129

Chapitre 6 Conclusion 131

I. Objectif et Démarche de Travail 131

II. Evaluation 132

III. Perspectives 134

Bibliographie 137

Chapitre 1
Introduction

I. Motivation

La construction d'applications qui reposent sur des systèmes distribués est un processus de plus en plus critique et important aujourd'hui. Ceci découle de facteurs tels que la généralisation et l'évolution des réseaux et technologies de communication au sein des entreprises mais aussi de l'évolution des modes d'échanges entre entreprises par le biais du réseau mondial Internet ou des Intranet. Le concept d'Intranet ou réseau interne de communication entre membres d'une même entreprise - ou d'un groupe d'entreprises partenaires - est devenu en peu d'années une base que tout système d'information moderne et performant ne peut ignorer. Les bouleversements induits par ces facteurs ne sont pas triviaux, tant au niveau de la modification du comportement et des méthodes de travail des utilisateurs qu'au niveau de l'adaptation des outils informatiques existants à ces nouvelles donnes. Outre les coûts de changement d'habitudes et de formation des utilisateurs à ces nouvelles techniques que l'on comprend aisément, les coûts d'évolution des applications informatiques sont eux aussi à prendre en considération. Ils dérivent essentiellement des problèmes de migration d'anciennes applications vers de nouveaux systèmes, de mise en place de la coopération entre des applications qui n'ont pas été pensées ni architecturées pour cela, de coopération entre des plates-formes matérielles hétérogènes,...

Certes, ces changements technologiques, en particulier la généralisation de l'Internet et l'accès universel par le World Wide Web, ont été accompagnés par l'arrivée de nouveaux modes de construction d'applications réparties où l'accès à des machines distantes est devenu relativement aisé. Le langage Java [Fla96] en est un exemple. Ce langage à objet s'impose peu à peu comme le mode de programmation universel. Il s'appuie sur trois caractéristiques majeures que sont une portabilité quasi universelle sur de multiples environnements informatiques, un apprentissage extrêmement facile et rapide et une intégration totale de toutes les fonctions d'accès et d'utilisation des réseaux et de l'Internet. Malgré tout, la construction d'applications réparties complexes pose toujours des problèmes difficiles que le Web et sa suite "d'innovations" n'ont pas la prétention de résoudre du jour au lendemain.

Considérons le cas concret du Réseau de Gestion des Télécommunications (RGT) de France Télécom. Le système d'information de cette entreprise, de taille importante dès le départ, a gagné en complexité au fil du temps. Ce réseau est en charge de la gestion des informations comptables (informations sur les abonnés, gestion de la facturation et collecte des consommations), des performances, de la sécurité, ... Le souci majeur des concepteurs de tels systèmes informatiques est de minimiser les redondances d'informations et de faire coopérer les différentes applications. Par exemple, la gestion des abonnés et la gestion de la sécurité ont des implications communes, car certaines données peuvent être partagées. Le second soucis concerne l'évolution et l'adaptation de tels systèmes aux nouveaux produits de France Télécom, aux nouvelles technologies, aux nouveaux services, aux nouveaux types de matériels (e.g. éléments de réseaux, matériels de réception),... Par exemple, en prévision de l'arrivée de nouveaux services du type Vidéo à la demande (VoD) dans les

foyers des abonnés de France Télécom, de nombreuses applications informatiques sont à (re)développer pour la fourniture de vidéo, pour l'administration des nouveaux matériels (réseaux hauts débits pour la transmission d'images et de sons, console d'accès aux services de VoD, ...). Ces applications doivent impérativement coopérer avec des applications existantes tels que la gestion de la facturation, la sécurité, ... Pour compliquer le processus, toutes ces applications sont intrinsèquement réparties du fait de la dissémination géographique des abonnés, des matériels d'acheminement des données, des agences commerciales, des centres de surveillance et de maintenance de France Télécom, ...

Cet exemple permet d'identifier un certain nombre de problèmes auxquels les architectes, concepteurs et développeurs de systèmes distribués sont confrontés. On peut les résumer par les interrogations suivantes :

- Comment fournir une vue globale d'une application distribuée à l'architecte tout en permettant un développement incrémental et progressif ?
- Comment garantir la facilité d'évolution de telles applications, en permettant de modifier les composants logiciels ou de permettre d'en ajouter ?
- Comment intégrer et faire coopérer des logiciels existants en garantissant un fonctionnement cohérent des nouvelles comme des anciennes applications informatiques ?
- Comment minimiser les coûts de développement inhérents au facteur de répartition en cachant au mieux les problèmes qui en découlent ?

II. Historique sur l'intérêt de la Répartition

La programmation d'applications réparties est une réalité depuis de nombreuses années. Les premiers systèmes de ce type reposaient (et parfois reposent toujours) sur un ordinateur principal contenant l'ensemble des données dites de production de l'entreprise dont le rôle est de permettre la production et la consultation des informations de l'entreprise. A ces ordinateurs, de type "*mainframe*", les utilisateurs se connectent au travers de terminaux passifs reliés point à point à l'ordinateur. Le mode de fonctionnement de ces machines suit un modèle maître esclaves où toutes les capacités d'accès, de traitement et de calcul sont dédiées au serveur. Ce mode de fonctionnement a dû être révisés par plusieurs facteurs dont le coût élevé des serveurs et l'arrivée progressive des réseaux locaux et des postes de travail de type ordinateurs personnels.

Les réseaux locaux étaient essentiellement constitués d'ordinateurs personnels dédiés au travail bureautique (traitement de textes, etc.) et leur utilisation pour l'accès au système d'information de l'entreprise s'est naturellement imposée de par la facilité d'utilisation (interfaces homme-machine conviviales, réseaux homogène permettant une gestion plus simplifiée). Peu à peu, le système d'information a migré vers des serveurs adaptés au fonctionnement en réseau local et le mode d'accès client-serveur est devenu incontournable. Le réseau local s'est peu a peu transformé en réseau d'entreprise où les serveurs dédiés aux bases de données ont conservé une place importante avec une prolifération d'autres services partagés tels que la messagerie électronique, les serveurs de fichiers, ... Le travail coopératif, où plusieurs utilisateurs travaillent de concert par le biais des outils informatiques à des taches communes, fait son entrée progressive dans le quotidien des entreprises grâce à ces nouveaux outils.

L'évolution naturelle aujourd'hui d'actualité est représentée par l'Internet et l'Intranet. Les barrières imposées par les réseaux d'entreprises ont explosé par l'interconnexion de millions d'entreprises, d'universités et de particuliers à un réseau mondial. L'Intranet utilise les mêmes technologies que l'Internet tout en préservant la confidentialité des activités d'une entreprise ou d'un groupe d'entreprises. Si on prend un point de vue économique de cette évolution, elle s'inscrit parfaitement dans le phénomène de mondialisation et de suppression des frontières au sein des entreprises et de la société.

Ce bref aperçu est avant tout caricatural, toutes les configurations matérielles et tous les types de réseaux coexistent encore mais il brosse un portrait de l'évolution de l'environnement d'exécution des applications réparties, ce qui a eu pour conséquence un changement radical de leur programmation. Ce changement de programmation a été aussi influencé par l'évolution des modèles de programmation tels que la programmation modulaire, puis la technologie orientée objet et l'utilisation de composants logiciels. Les besoins liés à la répartition se sont adaptés à ces évolutions des langages de programmation ce qui amène aujourd'hui une certaine confusion mais surtout un besoin notoire d'outils et de méthodes qui prennent tous ces aspects en compte.

III. Programmation Constructive et Configuration d'Applications réparties

La programmation constructive, dénommée aussi programmation par composition de logiciels, est un axe de recherche qui n'est apparu que récemment. De tout temps, le travail autour des langages de programmation et des méthodes pour construire des logiciels s'est focalisé autour de la mise en œuvre des composants logiciels d'une application en reléguant au second plan l'étude de l'assemblage de ces composants pour former une application. Des études récentes ont montré l'importance de la notion d'Architecture d'applications qui peut être définie comme *"... la structure des composants d'un programme/système, leurs interrelations, et les principes et lignes directrices gouvernant leur conception et leur évolution au fil du temps*."[Gar95]. La notion d'architecture devient vitale car elle se pose en trait d'union entre le cahier des charges d'une application, les méthodes de conception et la mise en œuvre. Elle permet de remonter de manière compréhensible et synthétique la complexité d'un système logiciel tout en ouvrant des portes à une certaine souplesse d'évolution des applications[Nie95a]. Cette souplesse se traduit par la possibilité de supprimer, remplacer et reconfigurer des composants sans perturber les autres parties de l'application. Les axes de travail autour de ce thème sont multiples, mais nous pouvons isoler trois grandes directions[Nie95b] : l'étude des modèles de composition, la spécification de langages pour la composition et les outils et méthodes qui doivent être associées aux principes de composition.

L'objectif de ce document est d'appliquer les principes de la programmation par composition aux applications réparties. L'idée fondamentale de ce travail est de profiter des avantages d'une architecture clairement explicitée pour cacher la complexité de programmation de telles applications, adapter les besoins des applications aux ressources disponibles sur le système distribué, et faciliter son évolution au fil du temps. Le but n'est pas de s'attaquer aux principes de réalisation des composants logiciels comme cela a été de nombreuses fois effectuées mais de faciliter le processus d'assemblage de composants éventuellement existant. C'est pour cela que nous avons accordé une certaine attention aux processus d'intégration de logiciels hétérogènes.

IV. Plan du manuscrit et Présentation du travail réalisé

Dans un premier temps, l'objectif de ce mémoire est de présenter les concepts de programmation constructive et de configuration d'applications qui fournissent des réponses à un certain nombre des problèmes que nous avons énoncé dans les sections précédentes. L'utilisation de ces concepts permet de définir un processus complet de construction d'applications qui englobe la phase de conception et d'architecture, celle de développement et d'intégration de logiciels, celle de coopération entre logiciels, celle d'installation et de mise en place des applications sur un système distribué et enfin la phase d'administration, de surveillance et de maintenance d'un applicatif informatique. Le chapitre 2 contient ainsi une classification de différents modèles et langages de construction et de configuration d'applications réparties.

Dans une seconde étape, nous présentons dans ce manuscrit l'environnement de configuration Olan qui fait l'objet de ce travail. Nous présentons dans le chapitre 3 le modèle de configuration d'application Olan et le langage de configuration associé OCL (Olan Configuration Language) qui permet de décrire l'architecture d'applications réparties et de configurer l'utilisation de composants logiciels hétérogènes et ainsi que leur placement sur un réseau de machines interconnectées. Le chapitre 4 décrit l'architecture et la mise en œuvre du support de configuration réparti Olan dont le rôle est d'installer et de déployer une application décrite en OCL et de mettre en place les structures permettant l'exécution de l'application. Le chapitre 5 présente une évaluation de l'utilisation de l'environnement Olan pour la construction d'applications en s'appuyant sur deux exemples concrets. Enfin, nous présenterons quelques conclusions dans le chapitre 6.

Chapitre 2
Modèles et Principes de Construction et Configuration d'Applications Réparties

L'objectif de ce chapitre est de brosser un aperçu des différentes méthodes de construction et configuration d'applications réparties parmi les tendances actuelles tant au niveau industriel qu'universitaire.

La programmation d'applications réparties a suivi un cheminement parallèle à l'évolution des langages de programmation sans que les spécificités liées à la répartition ne soient réellement prises en compte dans ces langages. A l'origine, la communication entre applications résidant sur des sites différents était complètement réalisée à l'intérieur du code des applications par l'utilisation explicite de mécanismes de types *socket*[ATT90] ou plus tard de type *RPC*[Bir84]. Les langages de programmation classiques n'ont jamais pris en compte la distribution. La répartition des modules sur des sites ainsi que les communications sont restées des tâches de programmation conséquentes. Tout au plus, un programmeur averti pouvait découper son application de telle sorte que les modules dédiés à la communication soient distincts de ceux propres au fonctionnement de l'application. Les langages orientés objets n'ont rien apporté de plus à ce processus hormis une séparation plus claire entre les interfaces d'objets et leur réalisation.

Ce besoin d'intégrer plus fortement la répartition dans la conception des applications se fait ressentir de façon plus pressante au fur et à mesure de la généralisation des réseaux de communications. La première réponse à ce souci fut de fournir des mécanismes plus évolués de communication tels que les courtiers d'objets (Object Request Broker)[OMG95a]. Au niveau des langages sont apparus les langages de définition d'interfaces[Sno89] qui permettent de décrire les fonctions d'accès à des objets ou des modules de manière homogène, peu importe le langage de programmation ou la plate-forme d'exécution. L'étape suivante (qui s'est déroulée en parallèle) fut de fournir une certaine vision de l'architecture de l'application en terme des entités logicielles nécessaires au fonctionnement de l'application, de leur placement sur les sites et de leurs intercommunications. Au travers de ce chapitre, nous allons mettre en évidence cette évolution et présenter quelques projets qui y ont contribués.

Revenons d'abord sur les méthodes de plus bas niveau de construction et de programmation d'applications réparties. Elles sont caractérisées par l'utilisation de langages de programmation (C, ADA,...) ou objets (C++, Smalltalk,...) et par la décomposition d'une application en modules ou en une hiérarchie de classes. Au sein du code de chaque module ou objet, des mécanismes pour communiquer entre machines sont utilisés, par exemple, les sockets ou un mécanisme de RPC. Il n'y a aucune vision claire de l'architecture de l'application ni de la répartition des modules sur différents sites. L'étude détaillée du code est le seul moyen de l'obtenir.

Voici un petit exemple en langage C de l'utilisation de *sockets* pour effectuer un appel depuis un client vers un serveur. Le thème de l'exemple est le suivant : un annuaire fournit une fonction `Lookup` et un client interroge l'annuaire via cette fonction pour récupérer l'adresse électronique d'une personne.

```
  Struct entree {
  char *cle;
  char *email;
};
static Struct entree annuaire[] = {
  {"Luc", "Luc.Bellissard@inrialpes.fr"},
  {"Michel", "Michel.Riveill@univ-savoie.fr"},
  {"Vlad", "marangoz@imag.fr"),
  ...
  NULL
}
/* Procédure qui retourne l'adresse émail d'un utilisateur */
char *lookup( cle)
char * cle {
  int i = 0;
  while (annuaire[i].cle != NULL) {
          if (strcmp(cle, annuaire[i].cle) == 0)
                  return &annuaire[i].email;
          i++;
  }
  return NULL;
}

void main() {
  int sock, length;
  Struct sockaddr_in serveur, client;
  int msgsock;
  char buf[STRLEN],email[STRLEN];
  int client_addr_len, rval;
// creation de la socket
  sock = socket(AF_INET, SOCK_STREAM,0);
  if (sock==-1) {
          perror("ouverture socket");
          exit(1);
  }
// liaison de la socket
  serveur.sin_family = AF_INET;
  serveur.sin_addr.s_addr = INADDR_ANY;
  serveur.sin_port = 7777;
  if (bind (sock,(struct sockaddr *)&serveur,
          sizeof(serveur)) == -1) {
          perror("bind");
          exit(1);
  }
  length = sizeof(serveur);
  if (getsockname(sock, (struct sockaddr *)&serveur,&length) == -1) {
          perror("getsockname"); exit(1);
  }
// écoute des demandes de client
  listen(sock,5);
  do {// boucle d'attente de connexion par le serveur
          client_addr_len = sizeof(client);
          /* acceptation de la connexion cliente et ouverture d'une socket temp */
          msgsock = accept(sock,(struct sockaddr *)&client,&client_addr_len);
          memset(buf,0,sizeof(buf));
          if (msgsock == -1) perror("accept");
          rval = read(msgsock,buf,STRLEN);/*lecture de l'entrée d'annuaire demandée
*/
          buf[rval]='\0';
          sprintf(email,"%s",buf);
          sprintf(email,"%s",lookup(email));/* recuperation de l'entrée */
          write(msgsock,email,strlen(email));/* écriture sur la socket cliente */
          close(msgsock);/* fermeture de la socket cliente */
  } while (TRUE);
  exit(0);
};
```

Figure 1. Code C du service d'Annuaire


```
  /* Programme client permettant le questionnement du serveur d'annuaire pour
récupérer l'adresse d'une personne */
#define SRVHOST "toua.inrialpes.fr"
#define SRVPORT 7777

void main() {

  char cle[STRLEN];
  int sock;
  struct sockaddr_in serveur, client;
  struct hostent *hp, *gethostbyname();
  char buf[STRLEN];
  int rval;
/* création de la socket */
  sock = socket(AF_INET,SOCK_STREAM,0);
  if (sock == -1) {
          perror("ouverture de la socket");
          exit(1);
  }
/* connection a la socket du serveur, adresse est fixe en dur dans ce programme */
  serveur.ssin_family = AF_INET;
  hp = gethostbyname(SRVHOST);
  if (hp == (struct hostent *)0) {
          fprintf(stderr,"%s: hote inconnu\n",
                  SRVHOST);
          exit(2);
  };
  memcpy( (char *) &serveur.sin_addr,
          (char *) hp->h_addr, hp->h_length);
  serveur.sin_port = htons(SRVPORT);
/* connection au serveur */
  if (connect(sock, (struct sockaddr *) &serveur,
           sizeof(serveur)) == -1) {
          perror("connection socket");
          exit(1);
  }
/* demande a l'utilisateur un nom de personne dont l'addresse doit etre trou"eee
*/
  printf("Nom de la personne a rechercher ?\n");
  scanf(cle,"%s");
/* ecriture du nom de la personne sur la socket */
  if (write(sock,cle, strlen(cle)) == -1 )
          perror("ecriture sur la socket");
/* lecture du resultat sur la socket */
  do {
          if ( (rval = read(sock,buf,STRLEN)) == -1)
                  perror("lecture sur la socket")
          if (rval == 0)
                  printf("Fin de connexion.\n");
          else {
                  buf[rval]='\0';
                  printf("L'adresse est %s",buf);
          }
  } while (rval>0);
/* Fermeture de la socket */
  close(sock);
  exit(0);
};
```

Figure 2. Code C du client du service d'Annuaire

Cet exemple permet le lancement d'un serveur Annuaire (dont le code est fourni en Figure 1) sur un site choisi par l'utilisateur. Ce lancement est directement effectué sur le site désiré. D'un autre côté, le client lance son programme client sur le site qu'il choisit et le programme s'occupe de réaliser la communication en espérant que le serveur existe, que la configuration de la socket est correcte (l'adresse doit être disponible, le client et le serveur doivent être branchés sur la même adresse, le format du message doit être toujours le même, ...). Tout ceci amène un nombre important de contraintes et peu de flexibilité tant au niveau de la mise en place, de l'évolution et de l'utilisation de l'application.

I. Problèmes

Au travers de ce petit exemple de programmation de type client serveur, nous pouvons tirer un certain nombre de remarques sur le mode de programmation réparti "classique".

- La modification de la répartition des objets (ou des modules) voire même le changement de mode de communication implique un travail conséquent dans le code existant. Il faut le modifier, recompiler, changer des variables, et vérifier si les programmes serveurs existent bien sur le site désiré.
- Le changement de l'outil de communication entre sites demande un remaniement profond du code. L'évolution de programmes due au changement du mécanisme de communication n'est ni facile ni rapide a réaliser. Par exemple le passage des *sockets* vers un mécanisme de RPC, ou bien le traitement de la reprise de la communication en cas d'erreur de transmission, demande de réécrire pratiquement entièrement le programme de test précédent.
- L'ajout de nouvelles fonctionnalités à un programme existant demande aussi un profond remaniement du code ainsi qu'une recompilation et une nouvelle édition de lien. Dans l'exemple précédent, il faudra mettre en place un dialogue plus compliqué entre le client et le serveur si l'information qui transite par les *sockets* est différente du simple nom de la personne et son adresse en retour. Par exemple, le type des paramètres échangés peut être autre que des chaînes de caractères.
- La structure de l'application n'est que peu visible. L'information disponible se résume à la présence de deux fichiers C. Existe-t-il un serveur, plusieurs clients, ... Ce genre d'information n'est pas visible, il est difficile de comprendre le fonctionnement de l'application sans étudier le code.
- Le problème d'hétérogénéité des plates-formes, du code et d'interopérabilité entre sites doit être complètement pris en charge par le programmeur à l'intérieur du code source. L'exemple précédent montre que le code est en langage C. Les types de paramètres ainsi que les valeurs seront-ils identiques d'un compilateur à un autre, d'une machine à une autre ? La communication avec du code écrit dans un autre langage nécessite la mise en place d'un protocole de communication entre les langages en codant les informations transitant par la *socket*. Par exemple, une chaîne de caractères envoyée par un programme C sera-t-elle la même si le programme est écrit en LISP ? .

II. Langages de définition d'interfaces

IDL est un langage qui permet de décrire complètement des interfaces d'objets ou de modules que l'on cherche à accéder de manière uniforme peu importe la plate-forme, le site ou leur langage de programmation. La richesse d'un IDL[Lam87][Sno89] est de fournir une méthode pour exprimer la sémantique des structures de données partagées et des opérations pour y accéder. L'utilisation d'un IDL, couplé à un compilateur de ce langage, permet de produire automatiquement le code pour accéder aux données et opérations des objets partagées par un ensemble d'utilisateurs sur un ensemble de sites, et ainsi de garantir l'homogénéité et la consistance d'accès aux objets par toutes les composantes de l'application répartie. Ce code produit correspond généralement à la notion de talons dans les RPC, avec une distinction entre talons clients (appelés par l'objet initiateur de la requête) et les talons serveurs (utilisés pour l'accès depuis le site du serveur) : ces deux types de talons sont usuellement désignés respectivement par les mots "stub" et "skeleton". Le talon (stub) permet d'emballer les paramètres au niveau du client initiateur d'un appel dans un format homogène, permettant de faire transiter les informations par le réseau. Une fois emballé, le talon effectue l'appel au mécanisme de communication réparti, qui le transmet au programme serveur. Avant d'atteindre le programme, les paramètres transitent par le squelette dont le rôle est de déballer les paramètres sous une forme compréhensible par le programme serveur. Le squelette effectue ensuite l'appel au programme serveur.

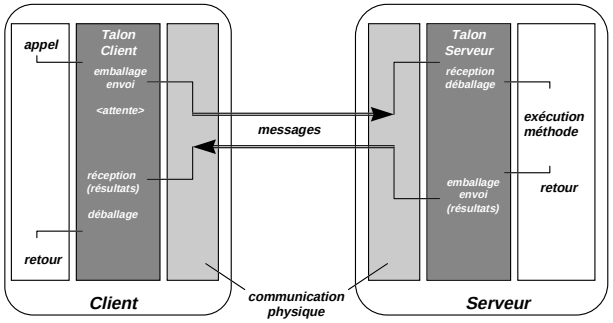

Figure 3. Schéma d'appel réparti

Depuis 1992, L'OMG, au travers de son effort pour imposer l'architecture CORBA (Common Object Request Broker Architecture)[OMG95a], propose un IDL pour décrire les interfaces des objets qui sont utilisés au travers d'un courtier de requêtes d'objets (ORB pour Object Request Broker). D'autre part, la notion d'IDL est aussi utilisée par des environnements d'appel de procédure à distance (RPC) dont celui du RPC de DCE[OSF95] (Distributed Computing Environment) ou du RPC de Sun[Sun90]. Nous n'étudierons pas les mécanismes de RPC mais nous nous contenterons de deux mécanismes de plus haut niveau qui font une large part à l'utilisation d'IDL : CORBA de l'OMG et DCOM (Distributed Component Object Model)[Rog97] de Microsoft. Ces deux mécanismes sont de plus haut niveau qu'un RPC, car le modèle de programmation des programmes serveurs reposent sur un modèle de programmation à objets pour CORBA et un modèle à base de composants pour DCOM.

II.1. Application au modèle Objet : CORBA

L'OMG propose dans l'architecture CORBA l'utilisation d'un IDL pour décrire des objets ou modules accessibles au travers d'un courtier d'objet (ORB). L'avantage d'un tel langage est de décrire de manière uniforme les structures de données échangées et leur projection dans les langages de programmation de la mise en oeuvre des objets[1]. L'IDL permet de définir des types, des constantes ainsi que des interfaces d'objets. Les interfaces contiennent un ensemble d'opérations ou méthodes, retournant ou non un résultat typé et contenant une signature également typée. L'IDL de l'OMG permet aussi de définir des exceptions et des levées d'exception puisque le modèle objet sur lequel est basé CORBA intègre des traitements d'exception.

Avant de continuer la présentation de CORBA pour la programmation d'applications réparties, revenons quelques instants sur les caractéristiques du modèle de programmation objet et sur ses avantages et inconvénients.

II.1.1. Principes élémentaire du modèle Objet

L'IDL de CORBA est basé sur le modèle de programmation à objets dont les principales caractéristiques sont l'encapsulation, l'héritage et le polymorphisme. En se référant à [Gar96], on peut donner les définitions suivantes à ces trois propriétés :

Encapsulation: intégration dans un même objet des données (ou état) et des opérations en ne laissant visible à l'extérieur que les interfaces constituées par les signatures des opérations publiques.

Héritage: permet de réutiliser une classe de base pour définir une nouvelle classe à des fins de spécialisations du comportement. Deux types d'héritage existent. L'héritage de type consiste à définir un nouveau type par ajouts de fonctions ou d'opérations. Seules les signatures des opérations sont héritées, l'implémentation pouvant être identique ou différente. L'héritage de classe au contraire permet de réutiliser des implémentations. Une classe est la réalisation d'un type avec en plus les attributs et les méthodes non publiques. Une sous-classe contient alors, en plus de ses propres attributs et méthodes, ceux de la classe de base.

Polymorphisme: permet d'attacher des codes différents à une opération en fonction de la classe de l'objet qui la met en œuvre ou de la signature de l'opération. Le choix de ce code dépend essentiellement de la classe de l'objet qui est appelé ou du type des paramètres envoyés lors de l'appel de l'opération.

II.1.2. Exemple de CORBA

CORBA est une architecture qui permet à un client d'invoquer des opérations fournies par des objets serveurs. La construction d'un objet serveur commence par la description de son interface IDL qui permet ensuite de générer le talon et le squelette, puis la réalisation de la mise en oeuvre des opérations de l'interface.

1 Il existe aujourd'hui un certain nombre de projection définie par l'OMG dont celles pour les langages C, C++, Smalltalk, Java, ...

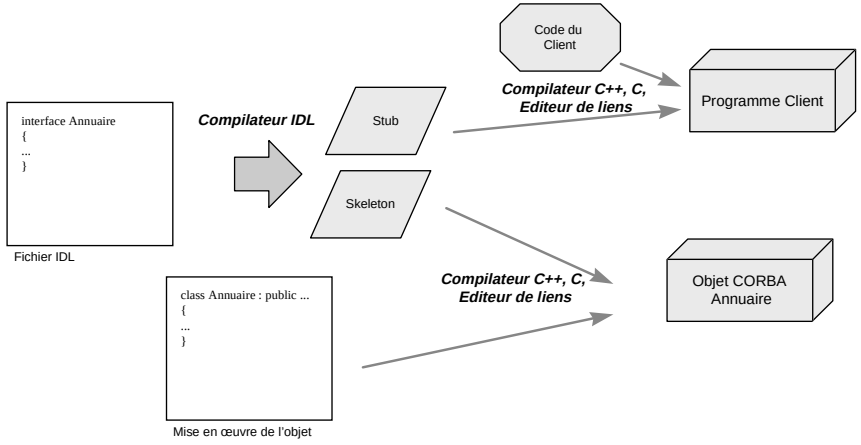

Figure 4. Processus de définition d'un objet dans un environnement CORBA

Le compilateur de l'IDL permet donc de générer automatiquement le talon et le squelette (stub et skeletons) nécessaire pour accéder à distance à un objet CORBA. Le client, pour pouvoir utiliser un objet serveur, doit préalablement lier son code à celui du talon d'appel de l'objet serveur. Il effectue ensuite un appel à l'ORB pour récupérer la référence au serveur (Oid) du serveur puis effectue les appels aux méthodes de l'objet serveur via le talon.

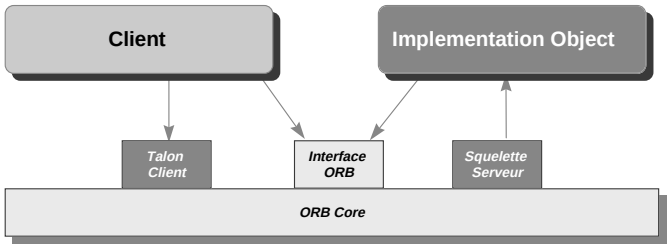

Figure 5. Architecture d'un courtier d'objet CORBA

Le client appelle l'interface du courtier d'objet pour récupérer la référence du serveur. L'appel est ensuite effectué vers le serveur via le talon qui emballe les paramètres et transmet la requête vers le squelette de l'objet serveur grâce aux primitives et fonctions de l'ORB. Celui-ci est en charge de localiser le serveur et d'utiliser un mécanisme de communication entre le site du client et celui du serveur.

Voici un exemple de description d'objet en IDL pour le système d'annuaire présenté plus haut.

```
Module SystemeAnnuaire {

exception UserNotFound {
string message;
string userName;
};

Interface Annuaire {
        void start_serveur(); // permet le lancement du serveur d'annuaire
        string Lookup( in string cle)
                raise (UserNotfound); // retourne l'adresse de la personne dont
                                // on fournit la clé en paramètre avec un
                                // déclenchement d'exception si la personne n'est
                                // pas dans l'annuaire
};
}
```

Figure 6. Déclaration en IDL de l'Annuaire

Nous présentons dans la le code du client ainsi que celui du serveur annuaire pour le courtier d'objet ILU[Jan96], compatible CORBA, mais possédant des règles de projection un peu différentes de celles de l'OMG. Ici le code du serveur est en Python[Ros95] alors que celui du client est en C.

```
import ilu, Annuaire_skel ## modules d'utilisation de ilu et celui du squelette
class Annuaire (Annuaire_skel.Annuaire):
        def start_serveur(self):
        ## lancement du serveur et de l'objet annuaire
                server = ilu.CreateServer("monServeur");
                handle = "L'Annuaire" ## nom de l'objet annuaire
                self.IluPublish()
                server.RunMainLoop() ## serveur en attente de requête

        def Lookup(self, cle):
        ## retourne l'adresse en fonction de la cle
                try:
                        return self.annuaire[cle]
                except KeyError:
                        raise Annuaire_skel.UserNotFound ## définie dans le talon
```

Figure 7. Code du Serveur Annuaire avec un ORB

Le serveur contient dans sa méthode `start_serveur` le code nécessaire pour lancer un serveur d'objet sur un site et la publication de son existence au sein de ce serveur. Ainsi, tout client ILU pourra interroger le serveur de nom ILU pour récupérer une référence universelle de l'objet Annuaire.


```
 void main() {
        Annuaire theAnnuaire;
        CORBA-Environment env;
        char cle[255], email[255];
        // Appel à ILU_C_LookupObject() pour récupérer la référence de l'objet
serveur Annuaire
        // On sait que le serveur s'appelle "monServeur" et que l'annuaire
s'appelle "L'annuaire".
        theAnnuaire = ILU_C_LookupObject("monServeur",
"L'Annuaire",Annuaire__MSType);
        if ( theAnnuaire == NULL) {
                fprintf(stderr, "Couldn't find Annuaire object _n);
                return(NULL);
        }
        // Demande de recherche de noms
        while (1) {
                fscanf(stdout,"Nom de la personne : %s", cle);
                Annuaire_Lookup(theAnnuaire, cle, &email, &env);
                if (! ILU_C_SUCCESSFUL(&env)) {
                        fprintf(stderr, " Exception signalée <%s>.\n",
                                ILU_C_EXCEPTION_ID(&env));
                if (email != NULL) {
                        fprintf(stdout,"Son email est : %s \n",email);
                } else fprintf(stderr,"Personne inconnue.\n");

        }
 }
```

Figure 8. Code du client ILU de l'Annuaire

Le client doit avant tout récupérer une référence vers le serveur Annuaire pour pouvoir appeler ses méthodes. Pour cela, il interroge le serveur de nom pour trouver un objet de nom "L'Annuaire" dont l'interface correspond à celle du talon. La fonction `Ilu_C_LookupObject` permet de faire ceci. Par la suite, le code d'appel du serveur est complètement insensible à la présence du courtier d'objet, mais le client doit connaître le nom ainsi que l'interface de l'objet distant. De plus son code doit être lié à celui du talon de l'Annuaire.

Il existe aussi des moyens plus dynamique pour l'appel à des objets serveur notamment avec les mécanismes d'invocation dynamique DII(*Dynamic Interface Invokation*) et DSI (*Dynamic Skeleton Invokation*). La DII permet à un client d'appeler n'importe quel serveur sans avoir son code lié statiquement au talon d'appel nécessaire. La DSI permet à un serveur d'intercepter l'appel en provenance de n'importe quel talon sans que le code du squelette ne soit lié au code du serveur. Ceci peut être intéressant lorsqu'une nouvelle version d'un serveur possédant une interface différente, doit préserver le fonctionnement de clients utilisant des anciens talons de ce serveur. Ces deux mécanismes permettent de s'affranchir de la phase d'édition de lien des talons et squelettes mais l'effort de programmation pour les utiliser est plus conséquent car une partie des fonctions présentes dans les talons ou squelettes doivent être programmées manuellement.

II.1.3. Avantages et Inconvénients pour la programmation répartie

Dans le cadre d'applications réparties, l'approche objet apporte un certain nombre d'avantages dont le principal est l'encapsulation et la séparation de la mise en œuvre et de l'interface. La description séparée de l'interface permet de générer automatiquement les talons et squelettes de transformation des appels de méthodes entre différentes plates-formes ce qui enlève au programmeur ce travail fastidieux tout en permettant la liaison aisée entre les talons et la mise en œuvre de l'objet.

Toutefois, dans le cadre de la réutilisation de classes, l'approche objet semble insuffisante. L'héritage de classe est en effet un contrat mal défini car il reporte les structures de l'objet dans les objets qui en dérivent, brisant par la même le concept d'encapsulation. Tout changement au niveau du

code de la classe de base impose la recompilation de toutes les classes dérivées. Tout ajout de fonctions à la classe de base demande aussi la recompilation des classes dérivées. Il est donc impossible de faire évoluer une hiérarchie de classe sans avoir accès au code et sans recompiler cette hiérarchie ce qui est une opération lourde dans le cas de programmes de taille et de complexité importante. Cette difficulté est connue sous le nom du problème de la *classe de base fragile*.

En ce qui concerne la lisibilité de l'architecture de l'application, le modèle objet n'apporte que peu de solutions hormis la connaissance de l'ensemble des types et classes utilisés par l'application. Les relations entre les instances de ces classes, de même que le schéma d'instantiation des classes ne peuvent être découverts sans une profonde étude du code des différentes classes. Il est de même pratiquement impossible de connaître le nombre d'instances de classes requis par une application à moins de mettre en place des mécanismes d'espionnage de l'exécution des objets de l'application. Dans le cas d'applications réparties, ceci nous semble un handicap car certaines relations d'utilisations entre instances peuvent permettre de répartir la charge d'utilisation des machines et surtout de colocaliser sur un même site des objets qui communiquent fréquemment. Le modèle objet ne répond pas à ceci car aucune notion de regroupement d'instances (et non pas de classes) n'existe.

CORBA est quant à lui un "super" RPC objet qui fournit en standard un modèle de communication de type client/serveur : des appels synchrones entre des clients et des objets serveurs. La modification du mode de communication entre les objets n'est pas aisée. Il existe des extensions à CORBA pour effectuer des appels asynchrones entre objets, en particulier le service d'événement[OMG95b] ou des services de communications de groupes[Fel97] mais le problème de ces extensions est qu'elles demandent un travail important de changement du code des objets pour les utiliser. CORBA n'apporte aucune solution pour la modification du modèle ou du protocole de communication entre objets distribués sans modification de la mise en œuvre d'objets distribués.

II.2. Application au modèle Composant : COM/DCOM

II.2.1. Principes élémentaires du modèle Composant

Pour palier aux problèmes de la programmation à objets tels que la fragilité de la classe de base ou la difficulté de la surcharge, le modèle à composants offre un mode de programmation relativement adapté à la distribution car il part de l'hypothèse qu'un composant est une entité de taille plus importante qu'un objet. On dit aussi que la granularité d'un composant est plus importante que la granularité d'un objet. Cette différence de granularité se justifie essentiellement par le fait qu'un composant a pour but d'encapsuler du logiciel au niveau du binaire, peu importe la méthode de programmation employée pour ces binaires. Il est fort possible d'encapsuler un logiciel qui a été programmé avec une méthode objet et qui contient une multitude d'objets, s'utilisant les uns les autres pour produire un ensemble de fonctions autour d'un même thème. Cette différence de granularité est donc plus intéressante dans le cas d'applications distribuées car on regroupe des objets[2] fortement dépendants au sein d'une même entité distribuable. Le but est donc de minimiser, dès la phase de

2 Le terme objet est pris au sens large : des classes, des modules, des bibliothèques, des exécutables, ...

conception des composants, les appels entre composants, réduisant ainsi les appels entre sites lors de l'exécution de l'application.

Un composant est une entité de réutilisation de logiciels au niveau du binaire. Il n'est pas nécessaire d'avoir accès au code source du logiciel pour pouvoir l'utiliser de manière cohérente. L'accès au «binaire » est facilité par la présence d'une ou plusieurs interfaces associées au composant. Ces interfaces sont décrites avec un IDL, ce qui comme dans le cas de l'architecture CORBA permet de palier à l'hétérogénéité des plates-formes d'exécution et du langage de programmation du logiciel encapsulé.

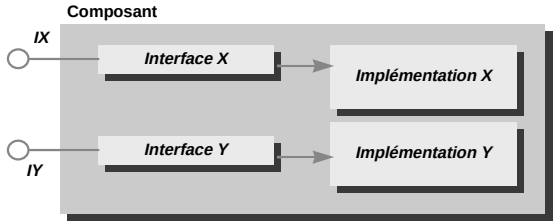

Figure 9. Structure d'un composant

L'encapsulation est la base même du modèle à composant : les données sont cachées et accessibles par le biais unique de services définis dans une interface.

L'héritage existe sous l'unique forme de l'héritage de type. Il est possible de définir des interfaces par héritage d'autres interfaces. Par contre, l'héritage de classe n'existe pas. Deux principes les remplacent : l'agrégation et la délégation permettent de «composer » un composant à partir d'autres composants. On parlera alors de composants internes et de composants externes.

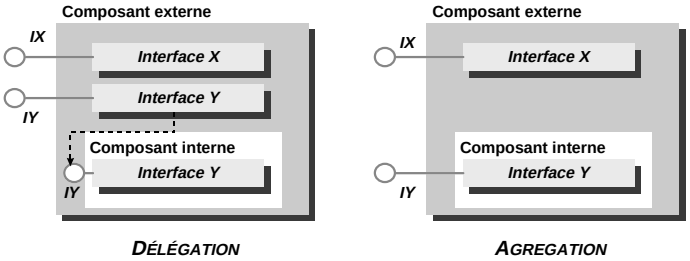

Figure 10. Délégation et Agrégation

> ***L'agrégation** est l'action d'exposer au niveau du composant externe le ou les interfaces des composants internes.*

L'intérêt de l'agrégation est de faciliter l'évolution d'un composant par l'ajout de nouvelles fonctions sans modifier l'ensemble des fonctions initialement offertes. Une version d'un composant présentant l'interface IX peut proposer dans une version postérieure des fonctions supplémentaires contenues dans l'interface IY. Les anciennes fonctions sont toujours présentes ce qui garantit le

fonctionnement des applications clientes existantes alors que de nouvelles fonctions sont disponibles grâce au composant interne ajouté par agrégation.

> ***La délégation*** *est un principe permettant d'utiliser un composant existant tout en l'encapsulant et surtout en permettant d'effectuer des traitements additionnels ou des restrictions sur les fonctions offertes par ce composant interne.*

L'intérêt de la délégation est de remplacer partiellement l'héritage de classe, en proposant un mécanisme de réutilisation de fonctions fournies par un autre composant, tout en permettant d'enrichir ou de restreindre son comportement. Ainsi, la délégation permet d'ajouter du code avant et après l'appel à la fonction du composant interne. Par rapport à l'héritage, le contrat entre la fonction du composant externe et celle du composant interne est moins ambigu car on sait exactement que le code additionnel se situe au niveau du composant externe.

Le polymorphisme est assuré par les multiples interfaces et le mécanisme de délégation qui permet de surcharger des opérations par changement de composant interne ou par ajout de traitements avant l'appel de la fonction.

II.2.2. Exemple de DCOM

DCOM (Distributed Component Object Model)[Rog97][Gri97] est une mise en oeuvre du modèle à composant précédemment décrit. Il est distribué par Microsoft. DCOM permet la description des interfaces de composants au travers d'un langage de définition d'interfaces MIDL et fournit les bibliothèques et mécanismes pour permettre la programmation et l'exécution de composants dans un environnement réparti.

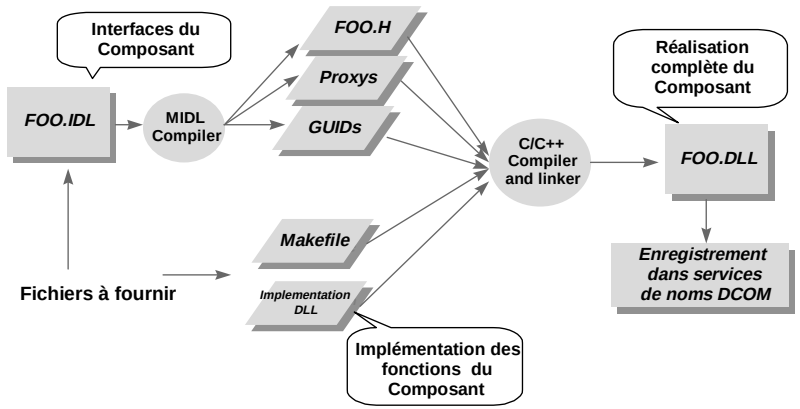

Figure 11. Processus de définition de composant dans l'environnement DCOM

DCOM, version répartie de COM, utilise comme mécanisme de base de la communication le RPC de DCE. MIDL est un langage de définition d'interface proche de celui fourni avec le RPC DCE. Il permet de générer les talons clients (appelés proxy) et les talons serveurs (stub). Le client qui désire accéder aux services d'un composant doit préalablement lier son code à celui du proxy. Le proxy est ensuite considéré par le client comme le composant lui-même. Les appels vers le composant sont

complètement pris en charge par le proxy. Contrairement à CORBA, il n'y a pas d'appel direct par le client à des fonctions des bibliothèques DCOM car tout composant offre une interface générique de recherche d'une instance de composant possédant des fonctions désirées. Cette interface, la `QueryInterface`, est en quelque sorte l'équivalent COM du mécanisme de DII (*Dynamic Invokation Interface*) de CORBA : une instance de composant est recherchée de manière dynamique sur l'ensemble du système distribué susceptible d'accueillir l'exécution des composants.

Voici l'exemple de l'interface de l'annuaire décrite avec MIDL.

```
 import "unknwn.idl";
 // Interface Ilookup d'accès à l'annuaire
 [
        object,
        uuid(32bb8323-b41b-11cf-a6bb-0080c7B2d682),
        helpstring("ILookkup Interface d'accès à l'annuaire"),
        pointer_default(unique)
 ]
 interface Ilookup : Iunknown
 {
        HRESULT Lookup([in string] wchar_t* cle, [out string] wchar_t**
resultat);
 }
```

Figure 12. Interface MIDL de l'Annuaire

La première partie de l'IDL de l'annuaire contient l'identifiant unique de l'interface sur le réseau de machines interconnectées. Cet identifiant permet à toute machine de récupérer la référence de cette interface. La seconde partie du fichier contient l'interface et la définition des services accessibles. Cette définition est proche de l'IDL du RPC de DCE, ce qui rend la syntaxe relativement éloignée de celle de l'IDL de CORBA. Les paramètres sont spécifiés deux fois avec une syntaxe IDL (entre []) et la syntaxe C++ correspondant aux en-têtes des classes de mise en œuvre de ce composant.

La description MIDL génère automatiquement les fichiers C++ des proxys et des talons, ainsi que des fichiers de définition des interfaces requises pour écrire la mise en œuvre des composants.

Les deux figures suivantes sont un exemple complet de la mise en œuvre du composant Annuaire en C++, ainsi que l'appel de ce composant depuis un client.

```
 // Composant Annuaire
 #include "Ilookup.h" // produit ar MIDL
 ... // déclaration de variables du module
 class CAnnuaire : public Ilookup {
 public:
        CAnnuaire();
        ~CAnnuaire();
        virtual HRESULT stdcall QueryInterface(const IID& iid, void** ppv);
        virtual ULONG stdcall AddRef();
        virtual ULONG __stdcall Release(); // fonctions de Iunknown

        // Fonctions de Ilookup
        HRESULT __stdcall Lookup (wchar_t * pcle, wchar_t** presultat);
 }

 // Structures de données de l'annuaire
 struct entree {
        wchar_t *cle,
        wchar_t* email
 } entree;
 static struct entree annuaire[] = {
        {"Luc", "Luc.Bellissard@inrialpes.fr"}, ..., NULL }

 // Mise ne oeuvre des méthodes de l'interface ILookup
 HRESULT __stdcall  CAnnuaire::Lookup (wchar_t* pcle, wchar_t** presultat) {
        int i = 0;
        while annuaire[i].cle != NULL {
                if (wcscmp(pcle, annuaire[i].cle) == 0) {
                        *presultat = &annuaire[i].email;
                        return S_OK;
                }
                i++;
        }
 }
 // Implémentation des autres méthodes dont la QueryInterface, AddRef et
ReleaseRef
 HRESULT __stdcall CAnnuaire::QueryInterface(const IID& iid, void** ppv) {
        if ((iid == ID_IUnknown) || (iid == ID_ILookup))
                *ppv = static_cast<ILookup*>(this) ;// On retourne par défaut
l'interface Ilookup
        else {
                *ppv = NULL; return E_NOINTERFACE;
        }
        reinterpret cast<IUnknown*>(*ppv)->AddRef();
        return S_OK;
 }
 ...
```

Figure 13. Code C++ de l'Annuaire pour DCOM

On voit que le code du serveur comme du client est devenu relativement différent de celui proposé dans les figures Figure 1 et Figure 2. L'annuaire est un composant COM. Pour cela, il doit être défini par une ou plusieurs interfaces dans laquelle doit obligatoirement être présente la fonction `QueryInterface`. Cette fonction dont le détail de mise en œuvre se trouve dans la Figure 13, retourne à un programme client un pointeur vers l'interface qu'il demande, à condition que cette interface soit supportée par ce composant. Cette fonction permet à des clients de découvrir dynamiquement les composants susceptibles de répondre à l'interface qu'il désire. De plus, la réalisation des fonctions propres au composant, telle que la fonction `lookup` doit suivre les règles de COM : utilisation de pointeurs, du format de retour d'erreurs, des types existants dans COM comme le `wchar_t` pour les chaînes de caractères, ...


```
 // Exemple de programme lient de l'annuaire
 #include "Ilookup.h"
 ...
 int main() {
        COSERVERINFO ServerInfo;
        CoInitialize(NULL); // Initilisation de COM
        Ilookup* pILK = NULL;
        memset(&ServerInfo, 0, sizeof(ServerInfo);
        ServerInfo.pwszName = "aroe.inrialpes.fr"; // La structure de
localisation des serveurs DCOM est prête

        MULTI_QI mqi[1]; // Strutures pour l'appel DCOM de la QueryInterface
        mqi[0].pIID = IDD_ILookup;
        mqi[0].pItf = NULL; // pointeur sur l'interface retournée
        mqi[0].hr = S_OK;
        // Création du composant Annuaire
        HRESULT hr = ::CoCreateInstanceEx(CLSID_Component1,
                                                NULL,
                                                CLSCTX_REMOTE_SERVER,
                                                &ServerInfo,
                                                1, // Nb d'interface que l'on veut
recupérer
                                                &mqi); // structure de récupératon
des interfaces
        if (SUCCEEDED(hr)) {
                wchar_t cle[255];
                wchar_t email[255];
                while (1) {
                        cout <<"Nom de la personne";
                        cin >> cle;
                        pILK = mqi[0].pItf; // pointeur vers l'interface Ilookup
                        hr = pILK->Lookup(cle, &email); // appel de la fonction
vers le composant
                        if (SUCCEEDED(hr)) {
                                cout<< "Son email est :"<<email<<endl;
                        } else cout<< "Personne inconnue"<<endl;
                        pILK->Release();
                }
        } else cout << "Impossible de créer un Annuaire"<<endl;
        CoUnitialize();
        return 0
 }
```

Figure 14. Code du Client de l'Annuaire pour COM

Le client est en charge de créer le composant à distance en spécifiant le serveur de création. Par la suite, les bibliothèques DCOM qui fournissent toutes ces fonctions de création, d'appel,... vont demander à la machine distante de créer le composant, puis les proxys et les talons d'appel seront mis en place. L'intérêt de DCOM est de pouvoir spécifier depuis le client l'endroit où le serveur sera créé, puis de laisser DCOM s'occuper des créations et lancements des serveurs distants. Par rapport à CORBA, DCOM peut piloter à distance les créations de composants, alors que CORBA demande qu'il y ait initialement un objet serveur en charge de récupérer des requêtes de créations.

On se rend compte malgré tout que la réalisation de composant DCOM est un travail non trivial. L'effort de développement est conséquent et il doit suivre à la lettre les consignes Microsoft. Il existe toutefois des bibliothèques de "*template*" qui simplifient le développement de nouveaux composants.

II.2.3. Avantages et Inconvénients pour la programmation répartie

Les avantages d'un modèle à base de composants dans un contexte réparti sont avant tout liés à la granularité du logiciel qui est encapsulé. Le composant permet l'accès à des portions de logiciel plus volumineuses que ce que permet un objet. Ce logiciel peut aussi se présenter de différentes sortes : hiérarchies de classes, bibliothèques de fonctions, binaires ou exécutables. Le composant unifie la méthode d'accès à ces différents types de logiciel tout en restant suffisamment souple pour une

utilisation dans divers contextes, i.e. dans diverses applications. Cette souplesse vient en particulier de la présence d'une interface clairement séparée du logiciel ainsi que la possibilité d'en offrir de multiples au niveau d'un composant.

Tout comme dans l'approche CORBA, les interfaces sont définies par un IDL, relativement indépendant du langage de programmation, ce qui confère au composant la possibilité d'être automatiquement réparti grâce à la génération des talons d'accès. Les principes de délégation et d'agrégation sont des principes de réutilisation très adaptés à la programmation répartie car ils permettent une évolution des composants sans modifier l'application cliente. En particulier, une telle évolution ne demande pas de recompilation de l'application cliente ni d'adaptation en cas d'ajouts de nouvelles fonctions.

Malgré tous ces avantages, il reste un problème majeur qui est la gestion de la complexité de l'application. Comment gérer un ensemble de composants, comment savoir quels sont les composants nécessaires pour le bon fonctionnement de l'application ? De plus, l'interface des composants, tout comme celle des objets, ne fournit qu'une information de type opérationnelle, i.e. les fonctions que le composant peut remplir.

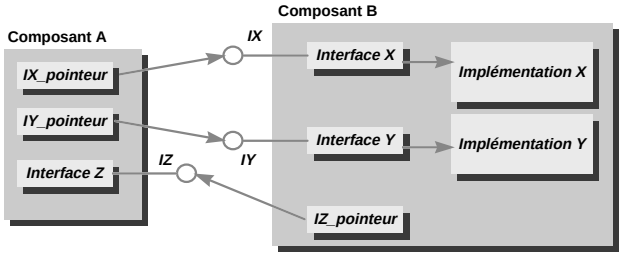

Figure 15. Interdépendance de composants

Imaginons qu'un composant doit communiquer avec un autre (voir Figure 15), i.e. que l'utilisation d'un composant dans une application impose d'avoir accès à un autre composant. Le mécanisme d'agrégation peut répondre partiellement à ce souci en intégrant les deux composants de la Figure 15 dans un seul composant agrégat. Mais si l'on pousse à l'extrême l'utilisation de l'agrégation, une application répartie sera constituée d'une partie cliente et de quelques «gros » composants qui eux-mêmes contiennent d'autres composants. Ceci peut être gênant car l'entité de distribution est le composant et l'agrégation de DCOM impose que tous les composants internes soient sur le même espace d'exécution. Comme dans le modèle objet, il manque les informations relatives aux relations et interdépendances entre les composants.

Pour revenir plus précisément au cas de COM/DCOM, un autre inconvénient de ce système est la difficulté d'écriture des composants même. Tous les gains d'utilisation d'un modèle à base de composant sont en parties remis en cause par le travail d'intégration de code dans des composants et d'assemblage des composants entre eux pour former une application complète. De plus, le problème d'interdépendance des composants que nous venons d'exposer au paragraphe précédent, ne facilite pas l'évolution d'une application ni la réutilisation des composants. Le composant COM n'est pas suffisamment autonome dans son fonctionnement.

II.3. Conclusion

L'utilisation d'un IDL a pour principal intérêt de décrire des structures de données partagées et les opérations pour y accéder. Cette description a pour caractéristique d'être complètement homogène peu importe le langage de programmation des modules, le type de logiciel et la plate-forme d'exécution destinée à l'accueillir. Le problème d'un IDL est qu'il ne s'agit que d'un outil partiel pour la description d'applications réparties. Il ne sert qu'à la description des entités logicielles susceptibles d'être utilisées dans un environnement réparti tel que CORBA ou COM. L'utilisation de l'IDL n'est qu'une étape, le reste du travail de construction d'application est de définir l'architecture de l'application, i.e. la structure de l'application en terme d'instances d'objets ou de composants nécessaires, à leur répartition et à la manière dont ces instances interagissent les unes avec les autres. Les sections suivantes présentent différents types de langages dont le rôle n'est pas de se substituer aux langages de programmation mais d'en être complémentaire afin de fournir aux concepteurs des outils qui permettent de définir l'architecture répartie de l'application.

III. Langages d'interconnexion de modules

La notion de MIL est apparue avec la constatation que la création de modules et leur connexion pour former des systèmes ou applicatifs plus gros et plus complexes étaient nécessaire et que ce processus était distinct de la programmation même des composants logiciels [DeR76] . Un premier langage a été proposé en 1975 qui permettait de connecter les différents modules logiciels d'une application de manière indépendante et séparée de la programmation classique des modules. Dans un MIL, un module exporte et importe des ressources, qui peuvent être de types aussi variées que des définitions de types, de constantes, de variables ou de fonctions. Le compilateur d'un MIL assure l'intégrité du système vis a vis de la vérification des types échangés entre modules. Par exemple, le compilateur vérifie que si un module utilise la ressource qu'un autre module fournit, le type de la ressource est compatible. Il peut aussi vérifier que si un module déclare fournir une ressource, celle-ci existe effectivement, que si un module utilise une ressource, il y a effectivement accès lors de l'exécution. Le rôle du compilateur du MIL peut alors s'apparenter à un "super" éditeur de liens qui est susceptible de prendre en compte des facteurs tels que la répartition de modules sur divers sites, la génération automatique de code permettant d'adapter des données en provenance de modules écrits dans différents langages de programmation et de les faire transiter entre des espaces d'exécutions distincts, sur une même machine ou non.

III.1. Polylith

III.1.1. Présentation

Polylith[Pur94] est un projet de recherche dont une des composantes est un MIL permettant de générer des applications réparties. Ces applications utilisent des bus logiciel pour la communication entre modules s'exécutant sur différents sites. La particularité de Polylith par rapport à des MIL antérieurs est que les modules peuvent être mis en œuvre dans des langages de programmation différents et que les mécanismes qui permettent de relier les modules lors de l'exécution sont plus complexes que de l'édition de liens de procédures. Ces particularités demandent au MIL plus de richesse d'expression en particulier pour ce qui concerne la mise en œuvre des modules : leur langage de programmation, les chemins d'accès au code de ces modules, les options de compilation, les informations sur la plate-forme d'exécution,... doivent être spécifiés

L'architecture de Polylith repose sur un compilateur du MIL et un ensemble de bus logiciel. Un bus logiciel est un ensemble d'agents qui contient et isole l'interface de tous les mécanismes d'exécution nécessaires à l'application. Le comportement et la mise en œuvre de tout mécanisme de communication ou d'exécution de l'application peuvent être ainsi modifier en changeant de bus logiciel sans toucher à la définition ni à la mise en œuvre des modules de l'application. L'utilisation d'un ensemble de bus logiciel demande la définition de la notion de bus abstrait. Le bus abstrait spécifie de manière rigoureuse la minière dont sont gérés les composants logiciels, les types de données autorisés pour l'écriture d'interface de modules, ainsi que les mécanismes de contrôle de communication et d'interconnexion de modules. Le bus abstrait est donc l'intermédiaire entre les composants logiciels de haut niveau, i.e. définis par le programmeur sans contrainte au niveau du support d'exécution, de

programmation et de communication, et les composants de bas niveau tels que le système d'exploitation, les mécanismes de communication, que le programmeur ne doit pas manipuler.

Le compilateur du MIL effectue alors les opérations de vérifications des interconnexions de modules et permet de générer le code nécessaire pour permettre l'exécution de l'application au-dessus d'un bus logiciel de Polylith. L'utilisation d'un bus logiciel particulier dépend essentiellement de la manière dont le concepteur de l'application désire que ses modules communiquent ainsi que la manière voulue de véhiculer les informations provenant des modules. Chaque bus défini un format d'échange des données entre modules (peu importe le langage de programmation des modules) ainsi que le mécanisme et le protocole de communication utilisés lors de l'échange de données entre modules. L'intérêt de l'environnement Polylith est donc de définir clairement le bus abstrait.

III.1.2. Exemple d'Utilisation

L'exemple que nous avons utilisé jusqu'à présent (exemple en partie inspirée du papier [Pur94]) se décompose maintenant en deux types de programme : le programme de l'interconnexion des modules client et serveur d'annuaire utilisant la syntaxe du MIL, et les programmes de la mise en œuvre de ces modules qui doivent fonctionner avec l'environnement Polylith et son bus logiciel.

Le programme MIL est composé en premier lieu de la définition des interfaces des deux modules, en terme de ressources qui sont fournis par le module (mot clé `define`) et de ressources dont le module requiert l'utilisation (mot clé `use`). Ici les ressources sont uniquement des interfaces des fonctions fournies ou requises pour le bon fonctionnement des modules. Chacun de ces modules contient aussi l'information sur la localisation des fichiers contenant la mise en œuvre des modules (attributs `SOURCE` de la description). L'application finale est constituée de la liste des modules nécessaires au fonctionnement (modules client et annuaire) pour lesquels il est possible de spécifier le site d'exécution (attribut `LOCATION`), ainsi que les liaisons entre les services requis et les services fournis par ces différents modules. Cette spécification permet au compilateur du MIL de générer le code agissant sur les primitives du bus logiciel pour d'une part demander la création des modules sur le site désiré et d'effectuer l'opération de liaison dynamique entre les services de ces modules. Le type de cette liaison dépend de la répartition des modules : deux modules présents dans un même processus utiliseront le mécanisme de liaison classique entre fonction alors que des modules présents dans des modules différents utiliseront un mécanisme d'IPC ou de RPC si leur site de placement sont distincts.

```
 module annuaire {
 define interface Lookup : PATTERN= string
        RETURNS = ↑{string}
} : SOURCE=annuaire.c

 module client {
 use interface Lookup : PATTERN = string
} : SOURCE=client.c

 module application {
 tool client : LOCATION=toua.inrialpes.fr
 tool annuaire : LOCATION=dyade.inrialpes.fr
 bind client.Lookup annuaire.Lookup
}
```

Figure 16. Description MIL des modules et de l'application

Par rapport à l'IDL de CORBA, la grande particularité de Polylith est de déclarer les opérations requises par le composant alors qu'un IDL se contente de spécifier les fonctions fournies par l'objet ou le composant. De plus, chaque module logiciel, qu'il fasse office de client ou de serveur doit posséder

une interface décrite dans ce formalisme. Ceci est une différence importante par rapport aux mécanismes de la famille des ORB où seuls les objets fournissant des fonctions - ou objets serveurs- sont généralement décrits avec l'IDL. D'un point de vue pouvoir d'expression, l'IDL de Polylith pêche par sa syntaxe propre de définition des signatures de services et par l'absence de toute une série de propriétés que l'on retrouve dans les IDL modernes, telles que les types nommés, l'héritage de type, les exceptions,... A la décharge de Polylith, ce ne sont pas ces propriétés qui sont visées par le projet de recherche mais plutôt le principe de description séparée des interconnexions.

Revenons sur l'annuaire. Le code de mise en œuvre de l'annuaire et du client est grandement simplifié par rapport à celui contenu dans les Figure 1 et Figure 2 car tout l'aspect communication à distance via des sockets est supprimé, il est pris en charge par le bus. La distinction entre appels distants ou locaux est faite à l'exécution par le bus logiciel en fonction de la localisation des modules. On s'aperçoit que dans le programme client, seul l'appel à une fonction Lookup est conservé, peu importe où cette fonction est définie. Il n'y a aucun lien à l'intérieur du code client qui fait référence au programme serveur d'annuaire. C'est le compilateur du MIL qui vérifiera la validité de la liaison spécifiée dans le programme de la Figure 16 et qui générera le code permettant au bus logiciel de mettre en place lors du début de l'exécution cette liaison. Nous pouvons noter aussi que l'utilisation d'un bus spécifique n'est pas indiquée dans le MIL, c'est lors de l'édition de lien que les modules et le bus sont liés.

```
  Struct entree {
  char *cle;
  char *email;
};
static struct entree annuaire[] = {
  {"Luc", "Luc.Bellissard@inrialpes.fr"},
  {"Michel", "Michel.Riveill@univ-savoie.fr"},
  {"Vlad", "marangoz@imag.fr"),
  ...
  NULL
}

/* Procedure qui retourne l'addresse email d'un utilisateur
Les addresses sont contenues dans la table annuaire */
char *lookup( cle)
char * cle {
  int i = 0;
  while (annuaire[i].cle != NULL) {
         if (strcmp(cle, annuaire[i].cle) == 0)
                return &annuaire[i].email;
         i++;
  }
  return NULL;
}
```

Figure 17. Programme serveur d'Annuaire utilisable par le MIL


```
  Struct entree {
  char *cle;
  char *email;
};

  void main() {
  char cle[256];
  char *adresse;

  /* demande a l'utilisateur un nom de personne dont l'addresse doit etre trou"eee
*/
  printf("Nom de la personne a rechercher ?\n");
  scanf(cle,"%s");

         if (adresse = Lookup(cle)) { /* Lookup sera lié au travers du MIL */
  /* L'adresse a été trouvée */
         printf("L'adresse est %s",adresse);
  } else
         printf("%s n'a pas été trouvé\n",cle);
} /* fin du main *
```

Figure 18. Programme du Client utilisable par le MIL

Enfin, nous pouvons remarquer qu'il existe une projection entre la signature des services au niveau du MIL et les types utilisés par les modules écrits ici en langage C. Cette projection suit des règles propres à l'environnement Polylith.

III.1.3. Avantages et Inconvénients

L'approche MIL a pour principal avantage le découpage explicite en modules de l'application afin d'exhiber une certaine architecture de l'application. La définition de l'architecture est faite dans un langage indépendant du langage de programmation des modules ce qui a pour conséquence, comme lors de l'utilisation d'un IDL, de pouvoir plus facilement réutiliser les modules et d'intégrer des modules ayant des formes ou des langages de programmation divers.

Le second apport majeur de Polylith est la séparation complète entre l'utilisation d'un mécanisme de communication et le code des fonctions du module. La configuration de la répartition des modules et de leur mode de communication est dirigée par des annotations dans le MIL et par l'utilisation de tel ou tel bus de message en fonction des besoins de l'application. La programmation de la communication entre modules est complètement absente du code du programmeur de modules, elle est entièrement prise en compte par le bus logiciel alors que son utilisation est spécifiée dans le MIL. La notion de bus abstrait de Polylith permet par la même, de changer le mode de communication (par exemple via un ORB, un RPC,...) sans toucher au code des modules. De plus, ce bus s'occupe de créer les modules sur les différents sites au moment de l'installation de l'application. Il n'y a pas besoin, comme avec un mécanisme de RPC ou CORBA, de mettre manuellement en place des serveurs avant utilisation, tout cela est pris en compte par le bus logiciel.

Néanmoins la démarche de Polylith possède quelques inconvénients :

- La description des modules est statique, il n'y a pas de moyens ou niveau du MIL pour décrire la dynamique de création et suppression des modules. Les structures d'exécution des modules existent dès le lancement de l'application. De plus, un module est unique dans une architecture Polylith, la notion d'instances de module n'existe pas.
- La communication entre modules est relativement figée pour deux raisons. Premièrement, le MIL ne décrit que la communication entre un module demandeur vers un module fournisseur de service : c'est une liaison de type 1 vers 1 comme dans le modèle client/serveur de base.

Plus contraignant, il est impossible de définir des schémas de communication plus complexe tel la diffusion d'une demande vers un groupe de modules, le choix dynamique des destinataires d'une demande de service en fonction des propriétés ou attributs de modules. En effet, dans CORBA ou DCOM, des services supplémentaires peuvent être ajoutés à l'architecture de base pour offrir ce genre de propriétés. Ici, de part l'absence de ces propriétés dans le MIL, il n'est pas possible de définir des architectures qui en tiennent compte. La deuxième raison vient du fait que toute communication au sein d'une description Polylith utilise le même bus de message pour communiquer. On ne peut pas configurer au sein d'une même architecture l'utilisation de divers modes de communication. Il est juste possible de changer le bus logiciel utilisé mais cette opération demande une nouvelle édition de liens entre les modules logiciels, le code généré par le compilateur MIL et les bibliothèques du bus logiciel.

- Les informations de répartition des modules sont simples, car seul le nom du site d'exécution d'un module est paramétrable. Ceci est donc comparable à ce que propose DCOM, mais une répartition plus fine, dans laquelle les utilisateurs ou les processus sont pris en compte, n'est pas possible avec le MIL. Or, nombre d'applications requièrent l'exécution de certains services ou composants par des utilisateurs privilégiés, dans des processus séparés, et ce principalement pour des raisons de sécurité. Il nous semble dommage qu'un MIL ne prenne pas cela en compte alors que le principe même de séparer la programmation des modules de leur répartition, est parfaitement adapté à ces soucis.

Enfin, les informations d'encapsulation des modules ne reflètent aucunement les caractéristiques du modèle d'exécution du module. Communique-t-il de façon synchrone, asynchrone ? Existe-t-il un flot d'exécution propre au module ou aucun ? Tout ceci pose un problème pour l'intégration et la répartition de logiciels existants.

III.2. Conclusion

Les MIL sont une étape importante vers la définition de l'architecture d'une application, en particulier les applications réparties. En effet, ce sont des langages complémentaires des langages de programmation qui permettent de décrire l'ensemble des modules logiciels nécessaires ainsi que les relations entre les interdépendances fonctionnelles des modules. Pour cette raison, on peut comparer le rôle d'un MIL à celui de guide des actions de l'éditeur de lien qui gère en sus la répartition des modules et l'utilisation d'un mécanisme de communication entre sites. Les MIL produisent ainsi une image exécutable de l'application, prête à être utilisée au-dessus d'un support d'exécution spécifique construit, dans le cas de Polylith à partir de la notion de bus abstrait. Ce bus permet la distribution des composants, tant du point de vue du déploiement sur les sites que de la communication entre modules. Toutefois, la description de l'architecture ne contient que peu d'aspects dynamiques. Il manque en particulier la description du schéma d'instantiation des modules au cours de l'exécution de l'application ou la configuration de la communication en fonction de facteurs dynamiques tels que des modifications d'attributs, des intervenants,...

L'idée fondamentale des MIL est de remplacer l'éditeur de lien pour en diriger l'utilisation à partir des spécifications de l'utilisateur. Le bénéfice de cette idée est d'obliger d'une certaine manière le programmeur à exhiber l'architecture logicielle de son application pour en faciliter par la suite la maintenance et l'évolution. Ceci permet aussi de faire un pas vers une meilleure réutilisation des

modules, car toutes les dépendances avec le monde extérieur sont clairement inscrites dans l'interface. Enfin, la description avec le MIL permet aussi de répartir les composants en substituant une phase d'édition de lien classique par la mise en place d'une communication distante entre sites.

Malgré tout, le concept de base du module n'est pas vraiment en adéquation avec les techniques plus récentes de programmation comme les objets ou les composants, car il ne permet pas de définir des architectures d'applications à base d'instances de classes ou de composants. Il ne peut donc exister deux modules identiques au sein d'une même application dont les interconnexions ou la répartition sont différentes. De plus, le mécanisme de communication n'est pas spécifié dans le langage d'interconnexion de modules, il fait partie de l'environnement de génération d'une application et son changement demande une nouvelle phase d'édition de lien du code des modules avec ce nouveau mécanisme.

La section suivante se propose d'étudier des langages dits langages de configuration dont l'idée de base est identique à celle des MIL, à savoir de séparer la description des interconnexions entre modules logiciels de leur programmation. Toutefois, ces langages font une avancée significative dans le domaine de la définition de l'architecture d'une application car une partie du comportement dynamique des applications y est pris en compte.

IV. Langages de Configuration

Les langages de configuration ont pour objectif de fournir un support linguistique pour décrire des configurations d'applications.

> *Une **configuration** est la description de l'ensemble des composants logiciels nécessaires pour le fonctionnement d'une application ainsi que celle de leurs communications et de leur contrôle.*

Les composants logiciels d'une application sont définis et construits à partir d'une interface qui comme dans un langage d'interconnexion de module exhibe les informations et les appels que peut fournir le composant ou qui lui sont nécessaires chez d'autres composants. Une configuration d'application contient donc l'ensemble des composants et leur interface associée, ainsi que les interconnexions entre ces composants en terme de liaisons entre les informations ou appels requis et ceux fournis.

La principale différence entre les langages de configuration et les langages d'interconnexions de modules est que le composant est considéré comme une entité instantiable : la description d'un composant au niveau du langage permet de créer de multiples instances d'un composant lors de l'exécution alors que l'entité module des MIL a le même statut que le module des langages de programmation classique : il est utilisable par plusieurs applications mais un module est unique dans un même espace d'exécution. De plus, ces langages introduisent une certaine structuration hiérarchique des modules logiciels de l'application accentuant ainsi leur intérêt pour la définition d'architecture réutilisable.

L'utilisation d'un langage de configuration est un moyen pratique et suffisamment abstrait pour décrire de manière compréhensible un système complexe. Il est d'autant plus adapté pour des systèmes répartis où les composants logiciels peuvent résider sur différentes machines. La configuration contient alors en plus de la structure de l'application, la spécification du schéma d'instantiation des composants et leur placement sur les différents sites du système. L'expression séparée d'une certaine dynamique de l'application, i.e. le schéma d'instantantion, permet alors de spécifier une image de l'application plus proche de ce qui existe réellement à l'exécution. On obtient grâce au langage de configuration une vision compréhensible de la structure de l'application tout en fournissant un moyen synthétique d'exprimer le placement et la répartition des composants. Le couplage d'un tel langage avec un outil de génération des exécutables permet de construire automatiquement des applications en faisant totale abstraction de la répartition au niveau de la programmation, car cet outil est capable de produire le code nécessaire à la communication distante ou locale. De plus, l'évolution de l'application est un processus grandement facilité par un tel langage car il suffit de modifier la structure d'un programme en ajoutant d'autres composants ou en modifiant des interconnexions. La notion de composant des langages de configuration s'approche de celle mise en œuvre par DCOM.

IV.1. CONIC

L'environnement CONIC[Kra89a][Kra89b], développé au Distributed Software Engineering Group, Imperial College de Londres, permet la programmation constructive de logiciels distribués et concurrents. CONIC est un environnement possédant une composante langage, une composante environnement graphique de développement et une composante support pour la configuration et reconfiguration dynamique d'applications. Nous allons au travers de cette section insister sur les deux premières composantes.

IV.1.1. Modèle de programmation

Le composant de CONIC est dénommé module. Tout module est défini par une interface unique qui contient des ports. Un port est un service soit fourni, soit requis par le module pour fonctionner. Chaque port est typé ce qui permet de vérifier sémantiquement l'interconnexion entre deux ports : un port de sortie de type T1 ne peut être connecté qu'avec un port d'entrée de même type T1. La communication entre ports se fait au moyen de messages typés. Il est impératif d'avoir le même type de message envoyé et reçu dans une interconnexion. La syntaxe de définition d'un module est la suivante :

```
 group module Name;
        use < Liste de types utilisés >
        exitport nomPort : typeMessage;
        ...
        entryport nomPort : typeMessage reply typeRetour;
        { Cette partie contient le lien avec du code source. Ce lien n'est
malheureusement pas documenté}
 end
```

Figure 19. Syntaxe des modules en CONIC

Pour définir la configuration d'une application, les modules sont instantiés, placés sur des sites et interconnectés. Il faut noter que la sémantique associée aux modules est le processus. L'instantiation d'un module correspond à la création d'un nouveau processus. L'interconnexion entre modules est l'opération de mise en relation entre un port de sortie et un port d'entrée de même type qui transmettent des messages de même type. Lors de l'exécution, une interconnexion se traduit par un envoi de message entre les deux modules. La description CONIC de la configuration est par la suite transmise au support de configuration de CONIC qui a pour rôle de créer les processus sur les machines cibles, d'y créer les modules désirés et enfin de permettre aux modules de communiquer.

La définition de configurations repose sur trois concepts de modules : les modules primitifs, qui intègrent directement du code et dont le modèle d'exécution associé est le processus, les modules composites et les modules représentants l'application dénommés *System*.

Les modules composites sont pratiquement identiques aux modules primitifs, car leur interface d'utilisation est définie de manière identique. La particularité vient de leur mise en oeuvre qui est composée d'ordres de création de modules (primitifs ou composites) ainsi que des mises en relation entre services requis par ces instances de modules (`exitport`) et services fournis (`entryport`). Cet ensemble de créations d'instances de modules et d'interconnexions est ce qu'on appelle une configuration, éventuellement réutilisable dans d'autres applications si le module est un composite. L'application est ainsi formée d'une hiérarchie de modules, primitifs ou composites. Les *Systems* - ou application - ne possèdent pas d'interfaces mais définissent les interconnexions de plus haut niveau dans la hiérarchie des composants formée par des composites et des primitifs.

```
group module Name;
        use typeDefinition; {les définitions des types}
        exitport and entryports
        use <type de modules> {clauses d'importation es sous modules requis}
        create
                nomInstance : nomModule;
        link
                interconnexion des modules, exitport en partie gauche,
                entryport en partie droite. Les ports proviennent des sous
                composants ou de l'interface de ce module composite
end

system ApplicationName;
        use <liste de types de sous composants>
        create
                <liste de création des sous composants >
        link
                < liste des interconnexions >
end;
```

Figure 20. Syntaxe des composants composites et System en CONIC.

CONIC se distingue des langages d'interconnexion de modules (MIL) par deux propriétés : l'instantiation de composants que nous avons précédemment introduit et le concept de composition. Chaque composant est instantié à partir d'un type défini dans le langage de configuration. Il est ainsi possible d'avoir de multiples instances d'un même type de composant (clause `create`) alors que les modules manipulés par les MIL tel Polylith ont la même sémantique que des bibliothèques dans un système tel que UNIX : il ne peut y en avoir qu'un par processus. La composition consiste à créer de nouveaux modules à partir d'autres modules. Ceci est réalisé par les composites. Il faut toutefois être conscient que la composition introduite par CONIC est légèrement différente de celle du modèle DCOM. Un composite possède une interface qui peut être construite à partir de tout ou partie des interfaces des sous composants, ce qui est comparable à l'agrégation. Néanmoins, rien ne remplace la délégation de DCOM, car on ne peut ajouter des traitements additionnels entre l'interface du composite et le sous composant.

Le facteur de répartition est pris en compte lors de la spécification de la création des modules dans la description d'un système uniquement. Le nom du site hôte de chaque sous composant est alors indiqué. Seuls les modules du système peuvent être placés sur des sites, il n'est pas possible de répartir des sous modules d'un module composite sur des sites différents. Cette limitation est ennuyeuse d'un point de vue réutilisation. Il peut en effet être intéressant de réutiliser des modules composites dont les sous composants ont une politique de placement particulière sans être obligé de la redéfinir pour une nouvelle application.

Le dernier aspect de CONIC concerne la capacité à reconfigurer dynamiquement une application, i.e. à modifier en cours d'exécution les modules utilisés ainsi que leurs interconnexions[Kramer90].

IV.1.2.Exemples d'Utilisation

Reprenons en détail l'exemple de l'annuaire. Nous allons dans un premier temps décrire les différents modules utilisés. Chaque exemple de représentation textuelle est accompagné de la représentation graphique correspondante car l'environnement CONIC fournit des outils de configuration graphique qui permettent, dans une certaine mesure, de faire abstraction du langage et de composer son application d'une manière plus intuitive.


```
group module annuaire ;
        use myTypes : annaireEntry ; {import port and message type}
        entryport lookup : lookupType reply annaireEntry ;
end

group module client ;
        use myTypes : annuaireEntry ; {import port and message type}
        exitport Lookup : lookupType reply annuaireEntry ;
end

group module Extended_Client (maxannuaire = 3);
        use myTypes : annuaireEntry ;
        exitport Lookup_Annuaire[1..maxannuaire] reply annuaireEntry ;
end
```

Figure 21. Exemple de l'annuaire avec CONIC

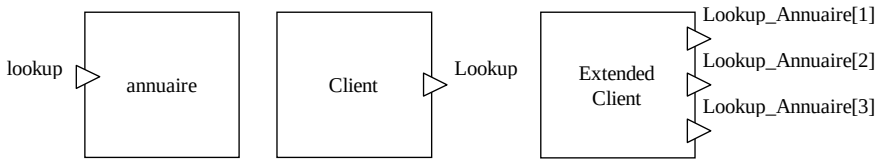

Figure 22. Représentation graphique CONIC de différents modules

Nous présentons ici deux versions de l'application Annuaire. Une version simple comparable à celle déjà utilisée et une version un peu plus complexe qui utilise deux annuaires, chacun contenant les abonnés d'un pays particulier. Il est intéressant de remarquer que le même module Annuaire est utilisé dans les deux versions.

```
System Application ;
        create
                client1 :Client at toua.inrialpes.fr ;
                annuaire: annuaire at dyade.inrialpes.fr ;
        link
                client1.Lookup to annuaire.lookup ;

System Extended_Application ;
        create
                client1 : Extended_Client     at toua.inrialpes.fr ;
                Annuaire_France : annuaire    at dyade.inrialpes.fr ;
                Annuaire_UK : annuaire        at test.dse.doc.ic.ac.uk.
        link
                client1.Lookup_Annuaire[1] to Annuaire_France.lookup ;
                client1.Lookup_Annuaire[2] to Annuaire_UK.lookup ;
```

Figure 23. Configuration CONIC d'une Application

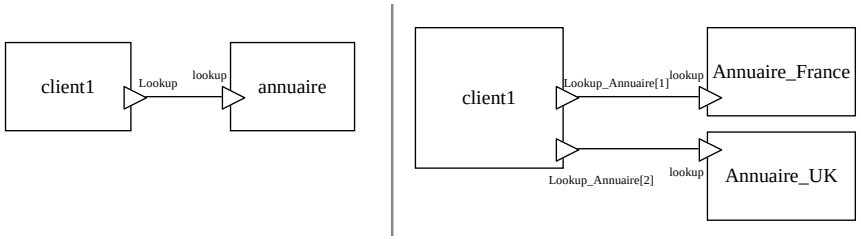

Figure 24. Représentation CONIC de l'application annuaire et annuaire étendu

IV.1.3. Avantages et Inconvénients

L'apport majeur de CONIC est de fournir une description de l'application proche de ce qui existe à l'exécution. Ce facteur permet de coupler des outils d'administration pour des opérations de surveillance de l'exécution de l'application ou pour de la reconfiguration. Ces outils utilisent le même formalisme graphique que celui utilisé pour la conception de l'application. Ce formalisme permet de synthétiser des informations de surveillance pour permettre une localisation rapide de divers problèmes par un administrateur, comme par exemple la panne d'un site, le blocage de l'exécution d'un composant, ou simplement des problèmes liés à la conception ou la répartition de l'application induisant des communications trop nombreuses entre différents sites, etc.

De plus, CONIC propose un embryon de définition de l'architecture d'une application car la structure logicielle, i.e. l'ensemble des modules utilisés, est clairement visible et manipulable. Les modules correspondent aux processus de l'application et les communications entre ces processus sont extériorisées, elles ne sont plus cachées dans la mise en œuvre des modules.

Enfin, la répartition et l'installation des modules sur un système distribué sont automatisées. Le concepteur ou le programmeur de l'application est soulagé de ce travail, car il est pris en charge par le support d'exécution de CONIC. C'est lui qui place les composants sur les sites en fonction de l'indication de placement et il utilise le mécanisme de communication adéquat en fonction de la localisation des modules.

Parmi les inconvénients de CONIC, nous avons choisi de distinguer :

Le problème de définition des interfaces des modules. La syntaxe est éloignée des habitudes des programmeurs, elle est notamment très différente de celle de l'IDL de l'OMG qui elle est relativement proche d'un langage tel que le C ou C++. La définition des types et des paramètres échangés est de plus peu pratique. De plus, il n'existe aucune adaptation de type lors de l'interconnexion de différents composants, car l'environnement CONIC n'offre pas de véritable compilateur, intégrant un système de type.

Le processus d'intégration de code existant n'est pas pris en compte. C'est au programmeur d'adapter les modules pour qu'ils puissent fonctionner avec l'environnement CONIC. Les modules primitifs sont écrits dans une version particulière de Pascal adaptée à l'environnement CONIC L'extériorisation des communications ou du schéma d'instantiation des modules n'est donc qu'en partie effective, car le programmeur doit inclure dans le code des primitifs l'utilisation de primitives particulières à CONIC. L'intégration de code existant n'est donc pas facile avec CONIC.

La communication entre composants n'est pas configurable. Il n'y pas de possibilité d'effectuer des communications à des ensembles de modules, ni d'utiliser des mécanismes de communication autres que l'envoi de message entre processus imposés par le système d'exécution CONIC.

La dynamique des applications, i.e. la création et la suppression de modules n'est pas décrite dans le langage. Il faut, pendant les exécutions, transmettre une nouvelle configuration au système CONIC qui dynamiquement effectue les changements de composants et d'interconnexions. Tout est dirigé manuellement par l'administrateur de l'application. Cette approche n'est pas facilement extensible, car plus le nombre de composants d'une application est important, plus il sera difficile de contrôler manuellement les changements requis.

Enfin, la répartition est identique à ce qui est proposé par Polylith, seul le placement en terme de site d'exécution est permis. Il n'y a pas, par exemple, la notion d'utilisateur pour qui se fait l'exécution d'un module.

IV.2. Darwin

Darwin[Mag93][Mag94a] est un langage de configuration à part entière. Son histoire est directement reliée à celle de CONIC puisque le projet Darwin est la suite directe des travaux de CONIC. Darwin est un langage de description de configurations logicielles. Il permet de décrire des interconnexions complexes entre composants. La particularité de Darwin est de permettre la spécification d'une certaine partie de la dynamique de l'application en terme de schéma de création de composants logiciels avant, après ou en cours d'exécution. Contrairement à CONIC, cette facilité évite toute intervention manuelle de l'utilisateur pour ce genre de changement, et cela ouvre la porte à la programmation du schéma de création de composant en fonction de besoins de l'application ou d'actions de l'utilisateur, de manière séparée du code même des composants.

IV.2.1. Présentation

Le concept de base de Darwin est le composant. Un composant est décrit par une interface qui contient les services fournis et requis. La notion de service s'apparente plus aux entrées et sorties de flots de communication qu'à la notion de fonction des MIL. Deux types de composants existent : les primitifs qui intègrent du code logiciel, et les composites qui sont des interconnexions de composants, comme dans CONIC.

La définition d'un composant suit la syntaxe suivante :

```
 component nom ( liste de paramètres ) {
        provide nomPort <port, signature>;
        require nomPort <port, signature> ;
 ...
        Implémentation, vide dans le cas d'un primitif, composée de déclaration
d'instances et de schéma d'interconnexion dans le cas de composites.
 }
```

Figure 25. Syntaxe Darwin de définition de composants

La sémantique associée à un composant primitif Darwin est le processus. L'instantiation d'un composant correspond à la création d'un nouveau processus avec la possibilité de lui associer un ensemble de paramètres typés d'initialisation déclarés en en-tête des composants. La portée de ces paramètres est son implémentation uniquement. Il est alors possible d'utiliser les valeurs de ces paramètres à l'intérieur de la configuration du composite pour décrire la configuration, comme un nombre d'instance à créer, etc. L'exemple de la section suivante en fait usage.

Les services requis ou fournis (`require` and `provide`) correspondent à des types d'objets de communication que le composant nécessite pour respectivement communiquer avec un autre composant ou recevoir une communication d'un autre composant. En d'autre terme, les services n'ont pas de connotation fonctionnelle, ils désignent seulement le type d'objet de communication utilisé ou autorisé à venir appeler une fonction du composant. Ces objets de communication sont fournis par le support d'exécution réparti Regis[Mag94b] qui permet à des configurations Darwin de s'exécuter. Le type de ces objets est spécifié en tête de la signature du service. L'usage du type `port` est le plus courant : il s'agit d'un objet envoyant des requêtes de manière synchrone ou asynchrone entre

composants répartis ou non. A la charge du programmeur d'utiliser les primitives associées à cet objet à l'intérieur du code des composants primitifs pour requérir une communication avec un autre composant ou accepter une communication en provenance de l'extérieur.

Les composants primitifs sont des classes C++ qui héritent d'une classe de base de Regis, la classe *Process*. A l'intérieur du code des composants, il est fait directement référence aux ports de communication requis ou fournis. Ces ports sont aussi des classes C++ fournies par Regis. On s'aperçoit que l'encapsulation de code existant dans un composant n'est pas une opération immédiate car il faut utiliser de manière explicite les ports de communication de Regis pour interagir avec d'autres composants. Seule la désignation du composant destinataire de cette communication n'a pas à être effectuée dans le code C++, c'est le langage de configuration qui s'en charge. L'encapsulation de code existant est ici une opération tout aussi non triviale que dans le cas de CORBA alors que Polylith, par exemple, automatise cette tâche.

Les composites sont les entités de configuration. En effet, ce sont eux qui contiennent le ou les descriptions des interconnexions de l'application. Une application est d'ailleurs un composant composite. La définition des implémentations des composites s'appuie sur deux constructions syntaxiques de base : l'opérateur `inst` qui déclare une instance d'un composant (sur éventuellement un site particulier) et l'opérateur `bind` qui relie un port requis en partie gauche avec un port fourni en partie droite. La particularité de Darwin par rapport à CONIC est de fournir des constructeurs dans le langage qui permettent de définir des schémas d'instantiation de composants complexes. Par exemple, il est possible de définir des variables tels que des compteurs, des variables booléennes,... Il existe aussi des constructions syntaxiques telles que l'itérateur `forall`, l'opérateur de test `when` (condition) alors,...

Prenons l'exemple suivant directement tiré de [Mag94a]. Le composant de base est un filtre qui possède l'interface décrite sur la Figure 26. Ce composant Filtre a un port d'entrée `left` qui récupère un entier. Si cette entier répond au critère de filtre du composant (critère qui ne nous intéresse pas ici), alors le composant le transmet par le port de sortie `output`, sinon il est transmis via le port `right`.

```
component filter {
        provide left <port, int>;
        require right <port, int>,
                output <port, int>;
}
```

Figure 26. Exemple de composant primitif en Darwin

Le composant Pipeline est une composition d'un nombre paramétrable de filtres. La syntaxe de ce composant est la suivante:


```
component pipeline ( int n) {
        // Interface
        provide input;
        require output;

        // Implémentation
        array F[n] : filter; // definition d'un ensemble d'instances de filtre
        forall k: 0..n-1 {
                inst F[k]; // création d'une instance de filtre
                bind F[k].output -- output; // lien avec le composant pipeline
                when k<n-1
                        bind F[k].right -- F[k+1].left;
        }
        bind    input -- F[0].left;
                F[n-1].right -- output;
}
```

Figure 27. Exemple de Composite en Darwin

Ce composant *pipeline* a pour rôle de "chaîner" des filtres les uns avec les autres, sachant que le nombre de filtre est fixé lors de la création du composant *pipeline*. Les deux opérateurs *forall* et *when* sont utilisés. *Forall* permet de déclarer un tableau de filtres ainsi que leurs interconnexions. Le constructeur *when* permet d'effectuer des déclarations conditionnelles de créations d'instances et d'interconnexions - bien que dans cet exemple seule une interconnexion soit réalisée.

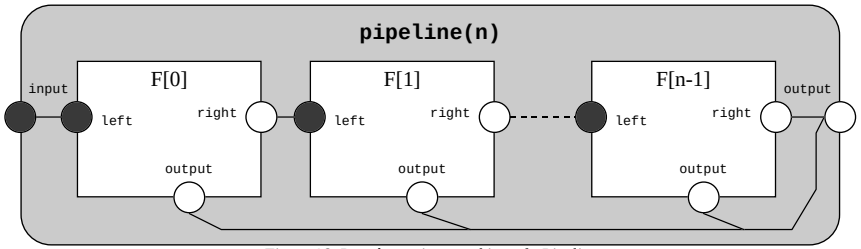

Figure 28. Représentation graphique du Pipeline

Une caractéristique importante de Darwin est de permettre la description de la création dynamique de composants par l'application. Cette caractéristique apporte une amélioration notable par rapport aux MIL et CONIC car l'architecture d'une application n'est plus considérée comme un ensemble de composants prévus dès la phase de conception : il est possible de spécifier les instants et les emplacements de création dynamique de composants logiciels.

Deux types d'instantiation dynamique existent : "l'instantiation paresseuse" et l'instantiation dynamique. L'instantiation paresseuse est une pré-déclaration des instances qui seront effectivement créées non pas lors de l'initialisation du composite mais dès qu'un premier appel vers l'instance est effectué. Ce type d'instantiation permet notamment de décrire des structures dont la taille ne peut être déterminée que dynamiquement. Dans l'exemple suivant du filtre, le nombre total de filtres n'est pas connu au démarrage de l'application, il n'est connu que lors de l'exécution. L'instantiation paresseuse, couplée à l'utilisation de la récursivité d'utilisation de composants[3], permet de résoudre ce problème.

[3] Nous ne détaillerons pas le principe de récursivité mais l'exemple de la Figure 29 montre que l'on utilise au sein du composite `lazypipe` des instances de type `lazypipe`.

```
component lazypipe {
        provide input;
        require output;
        inst
                head : filter;
                tail : dyn lazypipe; //recursivité, on instantie le composant
lui-même
        bind
                input -- head.left;
                head.right -- tail.input; // l'utilisation de cette communication
déclenche la création du composant tail, instances de lazypipe.
                tail.right -- output
                head.output -- output;
}
```

Figure 29. Instantiation dynamique dans Darwin

Dans cet exemple, la première fois qu'une communication partant de la tête vers la queue a lieu, le composant queue est créé puis le service `input` est appelé.

Toutefois, l'instantiation paresseuse ne peut pas toujours être utilisée. Darwin propose alors le concept d'instantiation dynamique. Elle permet à un composant de créer une instance de composant tout en fournissant des paramètres d'initialisation (ce qui n'est pas possible avec l'instantiation paresseuse car on ne positionne pas d'attributs avec l'opérateur `bind`). Un service requis particulier du composant initiateur doit exister. La déclaration de cette instantiation est réalisée dans la clause d'interconnexion. Par exemple, `head.right -- dyn pipeline(4);` La sémantique de cette interconnexion est la suivante. Dès que le composant de la partie gauche demande une création, le composant de la partie droite est créé. Cette demande doit être "codée" par le programmeur du composant initiateur en utilisant un type de port particulier dans le code de la classe C++ associée au composant. Cette interconnexion ne sert à rien d'autre qu'à la création du composant. Il n'y a pas d'appel de service.

IV.2.2. Exemple d'Utilisation

Considérons de nouveau l'exemple de l'annuaire. Voici la définition des composants primitifs client et annuaire. Nous avons étendu les capacités du client pour créer de nouveaux exemplaires de l'annuaire pour des pays différents.

```
Component client (int nbAnnuaire) {
        require lookupAnnuaire[nbAnnuaire] <port, string, string >;
}

Component Annuaire(int pays){
        provide lookup <port, string, string>;
}

component ClientFactory{
        require createClient <component>;
}
```

Figure 30. Exemple de l'annuaire en Darwin

Chaque client peut utiliser deux annuaires particuliers. Il cherche le nom à trouver dans l'un ou l'autre. Il existe aussi un composant de création de client (`ClientFactory`) qui permet de créer dynamiquement un nouveau client désirant utiliser un annuaire.

L'application est formée d'un ensemble de clients dynamiquement créés. Il n'est pas possible de connaître le nombre de clients a priori puisque celui-ci est choisi par la mise en oeuvre du client. Néanmoins un nombre maximum d'annuaire est spécifié dans la configuration.


```
Component Application {
        const int nbMaxAnnuaire = 2;
        inst clientFact : ClientFactory; // createur de clients
        array Annuaires[0..nbMaxAnnuaire-1] : Annuaire; // ensemble d'annuaire

        bind clientFact.createClient -- dyn Client(nbMaxAnnuaire);// création
                        // dynamique de clients
        forall k:[0..nbMaxAnnuaire-1] // itérateur, crée les annuaires
                                      // et les interconnexions entre tout
                                      // client et les annuaires
                inst Annuaires[k] : Annuaire(k);
                bind Client.lookupAnnuaire[k] -- Annuaires[k].lookup;

}
```

Figure 31. Application Annuaire avec Darwin

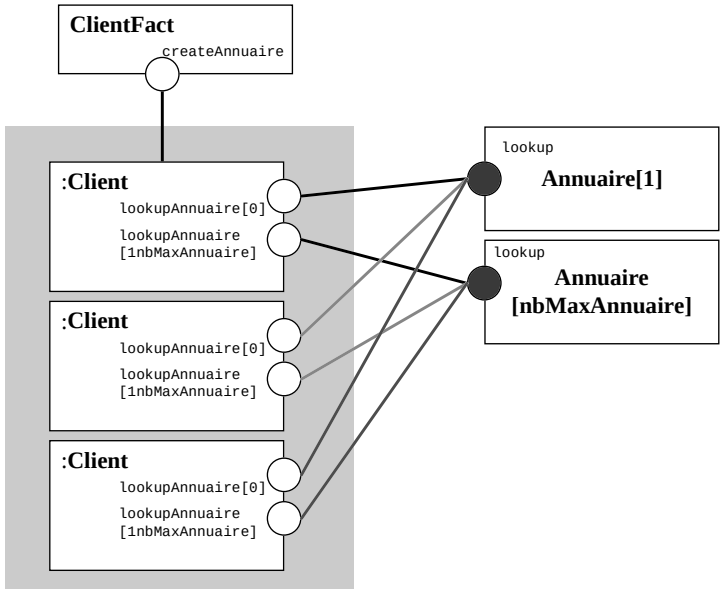

Figure 32. Représentation graphique Darwin de l'Annuaire

Nous pouvons remarquer dans l'exemple précédent l'utilisation du type de composant (ici Client) dans une interconnexion. Cela signifie que toutes les instances de Client ont leur port `lookupAnnuaire[k]` connecté à `lookup`. Regardons maintenant le code associé au composant Client. Il s'agit d'une classe C++ héritant de la classe *Process* fournit par Regis. Le code associé est le suivant :

```
class Client : public process {
public:
        int nbAnnuaire; // récupération de la constante positionné dans Darwin
        portref<string> lookupAnnuaire[nbAnnuaire];
        Client();
}
// constructeur est ce qui est lancé lors de l'instantiation du composant. Il
faut se rappeler que chaque composant est un processus (ou un processus léger si
des composants sont sur un même site).
Client::Client() {
        // Code du client
        char cle[255], email[255];
        int pays;

        while (1) {
                cout<<"Nom de la personne "<<endl;
                cin>>cle;
                cout<<"Pays d'origine"<<endl;
                cin>>pays;

                lookupAnnuaire[pays].send(cle, &email); // envoi de la requête
        }
}
```

Figure 33. Exemple de Mise en oeuvre de composants primitifs pour Darwin

Le lancement du processus associé au client commence par exécuter le code du constructeur de la classe Client. Dans ce constructeur, le programmeur envoi une requête de recherche d'adresse `lookupAnnuaire`. Cette application marche correctement du point de vue de la synchronisation car une requête n'est envoyée qu'à un seul annuaire.

Considérons maintenant une version différente de l'application dans laquelle les annuaires veulent communiquer avec un client. On pourrait vouloir écrire l'interconnexion suivante qui n'est malheureusement pas valide en Darwin, car un composant instancié dynamiquement ne peut pas recevoir de communication de l'extérieur.

```
forall k: [0..nbMaxAnnuaire]

        bind Annuaires[k].demandeService -- Client.service;
```

En effet, nous voyons ici la limite des spécifications de la dynamicité. Il n'est pas possible de définir le client qui va recevoir un appel de service. Un moyen de sélection dynamique du client serait ici nécessaire. Ceci est d'autant plus pénalisant que l'exemple des figures précédentes (Figure 31, Figure 32 et Figure 33) ne marche pas avec la version actuelle de Darwin car une demande de service ne peut recevoir des paramètres en retour, comme ici l'adresse email. Il faudrait donc que les annuaires puissent communiquer avec le client qui a envoyé la requête. Le problème de la sélection des composants dynamiquement créés est donc crucial.

Nous n'avons pas encore abordé la répartition des composants sur des sites différents. Elle se fait lors de la déclaration d'instances de composants en y ajoutant un numéro de site de création. Ce numéro correspond à un site dans les tables d'administration interne du support d'exécution Regis. L'ordre suivant permet de créer un client sur le site numéro 2:

```
inst client : Client( nbMaxAnnuaire) @ 2
```

IV.2.3. Avantages et Inconvénients

Les avantages de Darwin sont en premier lieu liés à la spécification explicite de l'architecture de l'application qui permet de décrire les communications mais aussi le schéma d'instantiation dynamique des composants. La vision de l'architecture, claire et proche des structures présentes lors de

l'exécution, est accessible par le concepteur de l'application comme par l'administrateur pour des opérations de surveillance. La séparation entre le code fonctionnel et le code de communication et de prise en charge de la répartition semble totale et configurable.

Les autres avantages sont liés aux constructeurs du langage Darwin qui permettent de spécifier des configurations complexes, paramétrables et qui peuvent même être contrôlées par des algorithmes. Nous avons vu l'utilisation de conditions et des itérateurs pour définir l'instantiation et les interconnexions. Il nous semble que ces opérateurs sont appropriés lorsque la structure de l'application se prête à une description algorithmique, i.e. que la topologie des interconnexions ou de la répartition sur des sites est a priori connue et qu'elle puisse être définie par ces opérateurs. Par exemple, prenons la répartition de processus sur une machine parallèle dont la topologie des processeurs est sous forme d'hypercube ou d'anneau. Les opérateurs `forall` et `when` permettent aisément de répartir les composants sur ces processeurs. A l'inverse, lorsqu'on se place dans un système réparti où tous les sites sont équivalents sans particularité topologique, et que les arrivées d'utilisateurs induisent la répartition des modules, ces opérateurs ne semblent pas apporter de réponse adéquate.

Un autre avantage concerne le facteur de répartition des composants. Il est pris en compte lors de chaque instantiation de composants. L'utilisation des outils de communication entre composants colocalisés ou distribués est complètement transparente pour le programmeur de l'application. Un support d'exécution associé au langage, Regis, permet de gérer tous ces types de communication en fonction de la localisation des composants ce qui est à comparer au rôle du bus logiciel de Polylith.

Parmi les inconvénients de Darwin, nous pouvons noter :

- L'intégration de code existant est relativement difficile. Le code d'utilisation de l'environnement Darwin/Regis n'est pas transparent pour le programmeur. Le programmeur doit explicitement utiliser les objets de type port pour réaliser ses communications et effectuer les créations de composants lorsqu'ils les sollicitent. De plus, le langage de programmation des composants doit être le même que celui qui définit les classes et objets manipulés par Régis, en l'occurrence C++.

- L'utilisation de divers modes de communication n'est pas configurable. Il faut que le support d'exécution supporte différents types de port. Par exemple [Mag97] propose une utilisation de Darwin au-dessus de CORBA en y adaptant les structures de Regis. Autre exemple, la distinction entre des communications synchrones ou asynchrones n'est pas faite au niveau du langage de configuration, cette possibilité est offerte au programmeur de composant qui utilise les ports de Regis pour communiquer mais rien ne transparaît au niveau de Darwin. Ceci nous semble un frein à l'assemblage de composants car les informations sur le modèle d'exécution des composants est un facteur important de réutilisation et de composition d'architectures qui fonctionnent correctement.

- La description de la dynamicité d'un ensemble de composants est limitée comme l'illustre l'exemple de la section précédente. Il n'est pas possible de supprimer des composants dynamiquement ni de permettre à un composant normal de communiquer avec des composants dynamiquement créés. En effet, l'exemple des annuaires présenté est partiellement invalide car les clients ne peuvent pas appeler des annuaires, seuls les composants dynamiques peuvent communiquer avec l'extérieur. De plus, il n'est pas non plus possible de configurer

une sélection dynamique des intervenants d'une communication, ce qui pourrait être une méthode pour résoudre le problème de communication avec des composants dynamiques.

IV.3. Conclusion

Les langages de configuration sont intéressants à plus d'un titre. Ils permettent de décrire la structure de l'application, les communications entre composants logiciels ainsi que le schéma d'instantiation des composants au fur et à mesure de l'exécution de l'application. Ils apportent donc un plus par rapport aux MIL pour la dynamique de création des composants. De plus, la structure de l'application spécifiée dans le langage, est relativement proche de l'application ce qui permet d'envisager d'utiliser le formalisme de description, graphique ou textuel, pour des opérations d'administration de l'application, par exemple de la surveillance d'exécution, des changements dynamiques de composants,... Pour cette raison aussi, ces langages et leur compilateur associé permettent de générer une image exécutable de l'application mise à la disposition des utilisateurs. Dans les deux exemples étudiés, il existe un support d'exécution qui permet le fonctionnement de l'application mais aussi le déploiement et l'installation des composants et des communications entre composants sur un système distribué.

Par contre, ces langages font quelques hypothèses restrictives à notre point de vue.

Les composants logiciels manipulés dans le langage sont associés à des processus (ou des processus légers - *threads* - selon les supports d'exécution). Ceci ne permet donc pas d'encapsuler dans des composants différents des entités logiciels passives, i.e. qui n'ont pas besoin de flot d'exécution propre pour fonctionner.

Il n'est pas possible de configurer la communication entre composants en terme de mécanismes ou de schéma d'exécution depuis le langage. Il faut toujours effectuer un travail au sein du code des composants. Par exemple, l'utilisation d'un mécanisme de communication tel CORBA ou COM demande des modifications au niveau du support d'exécution et au sein du code des composants, le langage de configuration ne fournissant pas d'abstractions permettant de spécifier ceci. De même, le schéma de synchronisation ou d'ordonnancement des appels n'est pas spécifiable. Nous avons vu que dans Darwin, la distinction entre appel synchrone ou appel asynchrone se fait dans le code des composants, aucune information n'étant remontée dans le langage de configuration. Cette limitation induit un moindre degré de réutilisation des composants et une impossibilité à vérifier avant le déploiement de l'application la validité de l'architecture.

Nous allons étudier dans la section suivante des langages de définition d'architecture qui tentent de combler ces deux inconvénients. Leur particularité est de fournir aux concepteurs un ensemble d'abstractions qui correspondent à celles les plus fréquemment utilisées en génie logiciel. Tous ces langages portent un effort particulier sur la spécification des interconnexions entre modules pour remonter le plus de propriétés possibles de la communication dans le langage de définition d'architecture. Pour cela est introduit le concept de connecteurs.

V. Langages de description d'architecture

Les langages de définition d'architecture (Architectural Definition Languages, ADL) se distinguent des MIL et des langages de configuration par leur objectif premier qui est d'aider les concepteurs de systèmes à structurer et composer leurs éléments logiciels pour former des applications. Traditionnellement, les architectes d'applications se restreignent à la définition des composants logiciels qui forment l'application mais il est apparu que les propriétés d'un tel système dépendent fortement des interactions entre composants[Sha95] . Bien que les concepteurs de systèmes possèdent de nombreuses abstractions informelles, elles ne sont guère présentes dans les langages de programmation classiques. Par exemple, il est fréquent de représenter schématiquement son application en terme de processus, qui communiquent l'un par un RPC, l'autre par de la mémoire partagée, etc. A l'intérieur d'un processus, il est fréquent de définir des boîtes, l'une étant un filtre, l'autre un logiciel de tri, etc. Les ADL reprennent les propriétés et intérêts des MIL et langages de configuration en y ajoutant les abstractions utilisées par les concepteurs d'applications, abstractions qui ne représentent pas les fonctions d'un composant logiciel mais plutôt le type de mécanisme qui permet de le mettre en oeuvre. Les concepts de base manipulés par ces langages sont :

- Le concept de Composants : il est représenté par une interface qui contient les services fournis et requis par le composant et il dérive d'un type qui représente le mode de mise en œuvre, e.g. un processus, une bibliothèque à chargement dynamique, etc.
- Le concept de Communication : il est représenté sous la forme de connecteurs qui interconnectent les interfaces de composant et permettent de vérifier et de définir le comportement de la communication. Cette communication n'est pas incluse dans l'implémentation des composants mais elle est totalement exhibée au niveau de l'ADL. Le connecteur aussi est dérivé d'un type qui représente le mécanisme de communication tel que l'appel de procédure, un appel de procédure à distance, l'utilisation d'un pipe, etc.
- La vérification de l'intégrité de la communication : elle est réalisée à partir des propriétés et des contraintes de chaque connecteur. Elle permet de vérifier statiquement avant la phase d'exécution si les interconnexions entre composants sont valides et si le comportement à l'exécution est conforme à la spécification des interfaces des composants et des connecteurs.
- La composition hiérarchique : elle permet de remplacer facilement les connecteurs et les interconnexions d'une (sous-)architecture pour former une nouvelle architecture.

V.1. UniCon

V.1.1. Présentation

UniCon (Universal Connector Support)[Sha95] est un langage de définition d'architecture dont les principaux concepts sont les composants et les connecteurs. Nous allons tenter de présenter dans cette section une vue synthétique de UniCon.

Composants

Une application est formée par un ensemble de composants. Un composant est avant tout une entité d'encapsulation des fonctions d'un module logiciel ou de données. Elle permet d'exhiber au travers une interface les fonctions et données accessibles comme les fonctions ou données requises par l'implémentation contenue dans le composant. L'interface définit aussi des propriétés sur le mode d'accès et l'utilisation de ces fonctions. Dans UniCon, le composant représente plus qu'un ensemble de fonctions ou de données : il définit aussi le mode de mise en œuvre ou le mécanisme qui permet de fournir les fonctions du composant. Par exemple, les fonctions peuvent être contenues dans une bibliothèque, au sein d'un exécutable, des données peuvent être contenues dans un fichier, dans un segment de mémoire partagée, etc.

Pour représenter ces mécanismes, UniCon utilise un système de type, chaque composant dérive d'un type qui caractérise le mécanisme. Par exemple, un composant peut être de type *Process* si tout ce qu'il encapsule sera contenu dans un processus particulier à l'exécution. Il sera du type *Module* si son implémentation correspond à des ensembles de fonctions contenues dans une bibliothèque de fonctions. Il peut être du type *SeqFile* s'il représente un fichier, *SharedData* s'il correspond à une zone de données partagées, etc. Le Tableau 1 dresse la liste des types de composants disponibles dans UniCon.

Le type du composant est important pour deux raisons : il permet aux outils de génération de l'application de produire automatiquement du code utilisant le mécanisme qu'il représente, et il impose des modes d'accès à ce qu'il contient, données ou fonctions. Ces modes d'accès sont spécifiés par les *Players*. Les *Players* correspondent aux services (fonctions ou données) fournis ou requis par le composant. Ils sont déclarés au niveau de l'interface et sont définis par un type, une signature et des propriétés pour l'interaction tel que le sens de la communication avec le connecteur (input ou output correspondant aux fournis ou requis). Il existe plusieurs types de *Players*, chaque type de composant imposant les types de Players qu'il est en mesure de supporter. Le Tableau 1 montre l'ensemble des types de composants existant dans UniCon ainsi que les types de *Players* acceptés pour chacun d'eux. Chaque type de *Players* correspond à un mécanisme d'accès a un service du composant, ce qui permet aux outils de génération de l'exécutable de produire le code associé. Par exemple, le type *ReadNext* d'un composant de type *SeqFile* permet de produire automatiquement le code de lecture du fichier associé à ce composant.

Type de Composant	***Signification du Type***	***Types de Players autorisés***	***Significations des types de Players***

Module	Correspond à la notion de bibliothèque logiciel ou binaire non lié	RoutineDef, RoutineCall, GlobalDataDef, GlobalDataUse, PLBundle, ReadFile, WriteFile	Correspondent respectivement à la fourniture ou la dépendance d'une fonction, l'utilisation...
Computation	Correspond à la notion de bibliothèque de fonction. Elle se distingue de celle précédente par l'absence de liens avec tous fichiers	RoutineDef, RoutineCall, GlobalDataUse, PLBundle	... d'une variable partageable et la lecture ou écriture sur un fichier. PLBundle correspond à des groupes de fonctions
SharedData	Zone de données partagées. Le mécanisme de partage est défini par les connecteurs d'accès.	GlobalDataDef, GlobalDataUse, PLBundle	Correspondent à des accès ou des utilisations de variables partageables
SeqFile	Fichiers à accès séquentiel.	ReadNext, WriteNext	Correspond à la récupération d'une lecture ou écriture
Filter	Flux de données	StreamIn, StreamOut	Ecriture ou lecture sur un flux
Process	Processus indépendant	RPCDef, RPCCall	Emission ou réception d'un appel de procédure prévue pour fonctionner avec un mécanisme de RPC
SchedProcess	Processus avec contrôle temps réel de son échéancier.	RPCDef, RPCCall, RTLoad	Idem précédemment, contrôleur temps réel
General	Correspond à tout ou n'importe quoi.	Tous les types de Players sont admis	

Tableau 1. Types de Composant

Prenons un exemple complet. Le type *Process* correspond à un processus géré par le système d'exploitation. La communication avec un tel type de composant se fait au travers d'un mécanisme de RPC. Ceci se traduit par l'utilisation obligatoire de *Players* de type *RPCDef* (appel d'une fonction vers le composant) et *RPCCall* (appel d'une fonction depuis le composant) qui permettent la génération des informations pour le système de RPC utilisé lors des communications.

La syntaxe des composants contient donc en premier lieu une interface dérivant d'un certain type, qui elle-même contient la liste des Players du composant. En second lieu, le composant contient une partie décrivant sa réalisation, i.e. le lien entre l'interface et du code logiciel.

```
COMPONENT Name
INTERFACE IS
        TYPE ComponentType
        < Liste des propriétés du composant >
        < Liste des Players du composant>
End INTERFACE
IMPLEMENTATION IS
        < Liste de Propriétés >
        Implementation Primitive | Implémentation Composite
END IMPLEMENTATION
END Name
```

Figure 34. Syntaxe de définition des composants UniCon

La réalisation d'un composant peut être de deux types comme dans CONIC et Darwin : Primitive, i.e. elle fait directement référence à du code source ou un binaire, ou Composite avec alors une implémentation qui contient une architecture, i.e. un ensemble de composants interconnectés via des connecteurs.

Connecteurs

Le connecteur a un rôle majeur dans UniCon car il contient les informations concernant les règles d'interconnexion de composants, telles que le protocole, la spécification des échanges de données, des transformations éventuelles du format d'échange ainsi que le choix du mécanisme nécessaire pour la communication lors de l'exécution. Au niveau du support d'exécution, le connecteur peut prendre des formes variées telles que un accès à des entrées dans un tableau, des tampons, des instructions pour un éditeur de lien, une séquence d'appels systèmes, un serveur acceptant des connexions multiples,... Chacun de ces connecteurs dérive, tout comme les composants, d'un type qui permet au compilateur de pouvoir générer l'application.

La définition d'un connecteur ressemble à celle d'un composant. Chaque connecteur est spécifié par un protocole (l'équivalent de l'interface d'un composant) qui défini les points de branchement autorisés pour les composants. Un protocole est défini par un type parmi ceux fournis par UniCon (c.f. Tableau 2), ce type définissant le moyen d'accès aux informations contenues dans des composants. Associé au type, le protocole définit les *Rôles* potentiels du connecteur, i.e. ses points d'entrées (ou de sorties) auxquels les *Players* d'un composant peuvent être interconnectés. Un *Rôle* est défini par un nom, un type et des attributs supplémentaires tels que une signature, une spécification fonctionnelle ou des contraintes d'utilisation. Du type des *Rôles* dépendent les *Players* autorisés à être branchés, comme le montre le tableau suivant. Ce système de type permet ainsi de vérifier statiquement si les entités d'un système utilisées comme des composants peuvent communiquer entre elles via un connecteur adéquat.

Type de Connecteur	***Types des Rôles et Players autorisés***

Pipe	Source (accepte StreamOut de Filter, ReadNext de SeqFile) Sink (accepte StreamIn de Filter, WriteNext de SeqFile)
FileIO	Reader (accepte ReadFile de Module) Readee (accepte ReadNext de SeqFile) Writer (accepte WriteFile de Module) Writee (accepte WriteNext de SeqFile)
ProcedureCall	Definer (accepte RoutineDef de Computation ou Module) Caller (accepte RoutineCall dde Computation ou Module)
DataAccess	Definer (accepte GlobalDataDef de SharedData ou Module) Caller (accepte GlobalDataUse de SharedData ou Module)
PLBundler	Participant (accepte PLBUNDLe, RoutineDef, RoutineCall, GlobalDataUse, GlobalDataDef de Computation ou Module)
RTScheduler	Load (accepte RTLoad de SchedProcesss)
RemoteProcCall	Definer (accepte RPCDef de Process ou SchedProcess) Caller (accepte RPCCall de Process ou SchedProcess)

Tableau 2. Type de Connecteurs et leurs Rôles dans UniCon

Prenons le type *RemoteProcCall*. Ce connecteur permet de relier des composants de type *Process* ou *SchedProcess*, i.e. des processus. Les *Rôles* définis au niveau de ce connecteur sont de type *Definer* ou *Caller* qui permettent l'interconnexion des *Players RPCDef* et *RPCCall*. Cette règle signifie qu'un processus peut appeler des fonctions (ou recevoir des appels de fonctions) préparées pour une communication interprocessus, et que seul le connecteur de type RemoteProcCall permet de réaliser cette communication en utilisant les points de branchements d'appel ou de réception. Nous verrons dans la section suivante l'utilisation de UniCon pour modéliser le système d'annuaire dans laquelle ce connecteur est largement utilisé.

La syntaxe de définition des connecteurs est la suivante. :

```
CONNECTOR Name
PROTOCOL IS
        TYPE ConnectorType
        < Liste de propriétés >
        ROLE NomRole IS RoleType
                < Liste de Propriétés de définition du role tel que le nombre de
composants connectés, ... >
        END ROLE NomRole
        ROLE .....
        END ROLE ...
End PROTOCOL
IMPLEMENTATION IS
        BUILTIN // On ne peut pas redéfinir des implémentations différentes de
connecteur
END IMPLEMENTATION
END Name
```

Figure 35. Syntaxe de définition des Connecteurs dans UniCon

Dans les versions à notre disposition de UniCon, il est impossible de définir de nouvelles implémentations de connecteurs, il faut obligatoirement choisir parmi l'ensemble prédéfini.

Architecture

Une Architecture est un ensemble de composants interconnectés. L'interconnexion de composants se fait au sein des implémentations d'un composant composite. Une interconnexion correspond à la mise en relation des *Players* des composants par un connecteur. Les *Players* sont connectés aux Rôles du connecteur utilisé et le compilateur de UniCon vérifie si l'interconnexion est conforme au système de typage des entrées et sorties des composants et connecteurs.

Pour faciliter l'utilisation de UniCon, un formalisme graphique existe et permet une représentation visuelle de l'architecture, comme dans le cas de CONIC et Darwin.

V.1.2. Exemple d'Utilisation

Reprenons l'exemple de l'Annuaire que nous avons exploité jusqu'à présent. La définition des composants Clients et Annuaire dans le langage UniCon est la suivante :

```
COMPONENT Annuaire
        INTERFACE IS
                TYPE Process
                PLAYER Lookup IS RPCDef
                        SIGNATURE ("char *", "char**")
                        End Lookup
        End INTERFACE
        IMPLEMENTATION IS
                VARIANT annuaire IS "annuaire.c"
                IMPLTYPE (Source)
        End IMPLEMENTATION
END Annuaire

COMPONENT Client
        INTERFACE IS
                TYPE Process
                PLAYER Lookup_Annuaire IS RPCCall
                        SIGNATURE ("char *", "char **")
                End Lookup_Annuaire
        End INTERFACE
        IMPLEMENTATION IS
                VARIANT client IS "client.c"
                IMPLTYPE(source)
        END IMPLEMENTATION
End Client
```

Figure 36. Description UniCon des Composants de l'Annuaire

La définition des composants contient en même temps la définition de l'interface et de l'implémentation. L'implémentation permet de faire le lien entre le code contenu dans des fichiers source, objet ou exécutable. Le compilateur de UniCon se charge alors de créer le "wrapper", i.e. le programme en charge de réaliser lors de l'exécution les appels au code source. Le code source associé à ces deux composants correspond sensiblement à celui de la Figure 17 et la Figure 18 avec en plus une fonction principale `main` dans le fichier contenant l'annuaire.

Les *Players* offerts par le composant sont typés (ici le type est *RPCCall* ou *RPCDef*, permettent l'accès des fonctions à travers un mécanisme de RPC, UniCon utilisant le mécanisme d'IPC - Inter Process Communication du micro noyau Mach uniquement -) et une signature décrite dans la syntaxe du langage de programmation de l'implémentation. Nous pouvons aussi remarquer que ces composants

ont le type *Process*, unique moyen de faire en sorte que le code encapsulé soit exécuté dans des processus différents.

L'application est un composant composite qui contient la définition des interactions entre ces deux composants. Sa définition est la suivante :

```
 COMPONENT Application
        INTERFACE IS General // pas de typage strict on utilise le type de plus
haut niveau
        END INTERFACE
        IMPLEMENTATION IS
                // Définition des interfaces de composants utilisées
                USE client1 INTERFACE Client
                        PROCESSOR ("toua.inrialpes.fr")
                        ENTRYPOINT (client1)
                END client1
                USE annuaire1 INTERFACE Annuaire
                        PROCESSOR("dyade.inrialpes.fr)
                END annuaire1

                // Définition des interactions entre le client et le serveur par
le biais d'un connector adéquat
                ESTABLISH Remote-proc-call WITH
                        client1.Lookup_Annuaire AS caller
                        annuaire1.Lookup as definer
                END Remote-proc-call
        END IMPLEMENTATION
 END Application

 CONNECTOR Remote-proc-call
        PROTOCOL IS
                TYPE RemoteProcCall
                ROLE definer IS definer
                ROLE caller IS caller
        END PROTOCOL
        IMPLEMENTATION IS
                BUILTIN
        END IMPLEMENTATION
 END Remote-proc-call
```

Figure 37. Définition de l'Application Annuaire en UniCon

La définition de l'application est un composant de type *General*, les autres types présents dans UniCon correspondent chacun à un usage précis alors que ce composant application a pour seul rôle celui de structuration de l'application. L'implémentation du composant application comporte en premier la déclaration des sous composants, sous la forme du nom de l'interface utilisé. L'utilisation de l'interface en lieu et place d'un composant permet au système de choisir parmi les composants ayant une même interface mais des implémentations différentes. Dans le cas des composants primitifs, il n'existe qu'une seule implémentation pour chaque interface alors que les composite peuvent avoir plusieurs *variantes*. La phase de définition des sous composants permet de positionner des attributs propres aux composants tels que le site d'exécution (Attribut *Processor*), le service de lancement du processus (Attribut *EntryPoint* qui pointe sur la fonction main du module encapsulé si rien n'est indiqué comme dans l'exemple),...

Une fois les sous composants déclarés, la seconde partie de l'implémentation contient la description des interconnexions à l'aide de connecteurs. Le but est de relier les *Players* des composants avec les *Rôles* des connecteurs. Le connecteur est décrit en dernier lieu, il ne sert ici qu'au renommage des *Rôles* par rapport au type de connecteur *RemoteProcCall*. L'implémentation du connecteur ne peut être définie dans UniCon (au moins dans la version présentée en [Sha95]), on utilise celle prédéfinie par le système UniCon et il n'est pas encore possible de créer de nouveaux types de connecteurs.

V.1.3. Avantages et Inconvénients

UniCon présente un certain nombre d'avantages pour la construction d'applications en général, réparties en particulier :

- La description de l'architecture d'une application dont la mise en œuvre repose sur des mécanismes a priori suffisants pour le concepteur d'applications Unix : processus, flots (streams), modules, communications par pipe, mémoire partagée, RPC,...
- Le système de typage fort des composants et des interconnexions permet de vérifier dans une phase d'analyse la faisabilité et la validité de l'architecture.
- L'intégration de logiciels existants à différents niveaux : source, objet ou exécutable. La prise en compte de multiples langages de programmation existe bien que la définition des signatures des services soit propre à chaque langage de programmation. Il n'existe pas de syntaxe "unifiante" comme le propose les IDL.
- La séparation complète du code des composants de toutes fonctions de communications qui sont intégrées dans les connecteurs. Les connecteurs ont le double rôle d'outil réalisant la communication mais aussi d'entité de vérification de la validité des interconnexions.

Les inconvénients de UniCon dans le cadre des applications réparties sont en nombre restreints mais nous retenons parmi ceux-ci :

- La description des interfaces n'est pas faite dans un langage homogène pour la définition des signatures des services comme le concept d'IDL le permet. La signature est essentiellement dépendante du langage. Dans le cas de l'utilisation du RPC (RPC de Mach uniquement), la signature doit être exprimée, outre dans le langage du code encapsulé, dans le formalisme du RPC utilisé. Cette seconde description n'est pas directement insérée dans une description UniCon mais spécifiée au travers des Attributs de composant RPCTypesIn et RPCDef dans un fichier séparé.
- Le déploiement d'une application répartie en terme de processus et de site est fait explicitement par le programmeur. Imaginons qu'un composant définissant un module (type Module ou Computation) soit réutilisé dans une autre application pour en faire un composant serveur accessible à distance. Le concepteur de l'application doit redéfinir le composant UniCon correspondant en le typant différemment, en redéfinissant son interface et en modifiant son architecture initiale afin de changer les connecteurs et toutes les interconnexions attenantes pour utiliser des connecteurs de type RemoteProcCall. Cette opération n'est pas immédiate, le changement de l'architecture du système distribuée sous-jacent et de l'application même impose un travail de "reengineering" conséquent.
- La description de la dynamique de l'application, i.e. le schéma de création/suppression des composants est totalement statique. Il n'existe pas d'abstraction au niveau du langage pour la décrire, contrairement à l'instantiation dynamique de Darwin.

En conclusion, UniCon est un langage qui permet la construction d'applications réparties mais qui ne nous semble pas le plus adapté à la répartition. Son système de type des composants et des interconnexions est très intéressant mais il manque de souplesse pour la définition de composants réutilisables. En effet, il n'est pas possible de répartir des composants qui n'ont pas été prévus pour

cela dès leur conception. L'approche est donc ici complètement différente de celle de CORBA ou DCOM où l'objet ou le binaire peuvent ou ne pas être répartis sur un système distribué selon le désir du programmeur. De plus, une architecture UniCon ne se préoccupe pas de la dynamique de l'application. La création de nouveaux modules pendant l'exécution ne peut pas être décrite dans le langage.

V.2. Rapide

Rapide[Luc95] est un langage de description d'architecture dont le but initial est de vérifier par la simulation la validité d'une architecture logicielle donnée. Une application est construire sous la forme de modules ou composants communiquant par échange de messages ou événements. Le simulateur associé à Rapide permet ensuite de vérifier la validité de l'architecture, i.e. l'absence d'interblocage lors de l'exécution et la conservation de l'ordre causal de délivrance des événements lorsque les composants sont assemblés et interconnectés.

Il est important de noter que l'environnement Rapide ne permet pas la génération automatique d'applications, mais il est intéressant de le présenter ici, car c'est un langage de description d'architecture qui met l'accent sur la spécification du comportement dynamique de l'application : création et suppression de composants, interconnexions qui se modifient dynamiquement en fonction des propriétés des composants existants dans l'application, etc.

V.2.1. Présentation

Le concept de base de Rapide est l'événement qui est une information transmise entre composant. L'événement permet de construire des expressions appelées "*event patterns*". Ces expressions permettent de caractériser les événements circulant entre composant. Par exemple, si A est un événement, A > B signifie que B sera envoyé après A. La construction de ces expressions se fait avec l'utilisation d'opérateurs permettant d'exprimer des dépendances entre événements. Parmi les opérateurs présents, on peut trouver l'opérateur de dépendance causal (A -> B si l'événement B dépend causalement de A), d'indépendance (a || B si A et B ne sont pas causalement dépendants), de différence (A ~B si A et B sont différents), de simultanéité (A and B si A et B sont vérifiés),... La notion d'événement correspond à une information transmise. Un événement peut donc correspondre à une demande de service, à une valeur particulière d'un attribut, à une apparition d'un nouveau composant,... Il ne s'agit pas seulement d'un envoi de message entre deux entités logiciels mais à une information quelconque portant sur le comportement des composants de l'application.

La sémantique particulière des événements associés aux opérateurs de construction des expressions d'événements, permettent de caractériser les interconnexions entre composants d'un point de vue échange de paramètres, ordonnancement et sélection dynamique des instances de composants qui communiquent effectivement entre eux. Considérons l'exemple suivant qui décrit une application contenant des objets serveurs et des objets clients.

```
 with Client, Serveur;
 ...
 -- déclaration des instances de composants de l'application susceptible
d'exister
 ?s : Client; -- fait référence à une instance de Client
 !r : Serveur; -- fait référence à toutes les instances de serveur
 ?d : Data; -- fait référence à un bloc de paramètre d'un certain type Data
 ...
 -- Une règle d'interconnexion
 ?s.Send(?d) => !r.Receive(?d);;
 -- Si un client transmet un événement de type Send avec ce type de paramètres,
alors, l'événement est transmis à tous les serveurs de l'application avec ces
paramètres.
```

Figure 38. Exemple d'expression d'événement Rapide

L'exemple utilise deux composants : le client peut émettre un événement *Send* avec des paramètres de type Data, et le serveur réagit à l'événement *Receive*. L'interconnexion précédente dit que dès qu'il existe un client qui émet *Send* avec n'importe quelles données de type Data, alors la partie droite de la règle est exécutée, i.e. tous les serveurs du système reçoivent la même donnée *?d*. Les opérateurs ? et ! permettent de définir des variables (des objets ou composants utilisés) avec une sémantique particulière. ? indique que l'on choisit un composant du bon type parmi ceux présents dans le système alors que ! indique que tous les composants de ce type présents dans le système sont choisis. Notons que le programme précédent ne spécifie pas les instances devant être créées, il utilise un ensemble d'instances susceptibles d'être présentes dans le système.

Le second concept de Rapide est le composant ou module qui est défini par une interface. Une interface correspond à la notion de type de composant de UniCon. Une interface est constituée d'un ensemble de services. Les services sont des fonctions qui peuvent être appelées ou que le composant requiert. Ces services peuvent être de type "*provides*", "*requires*", "*action*". Les "*provides*" sont des services qui peuvent être appelés de manière synchrone par d'autres composants. Les "requires" sont les services que le composant demande de manière synchrone à d'autres composants. Un appel synchrone entre deux composants correspond à une connexion entre un service "*requires*" et un "*provides*". Les "*actions*" correspondent à des appels asynchrones entre composants. Deux types «d'*action*» existent : les actions "in" et les "out" qui sont des événements acceptés et envoyés par un composant. Tous ces services ont la propriété d'être observables par l'architecture - i.e. le regroupement des composants interconnectés - et donc le système Rapide est en mesure de générer des traces d'exécution lors de phases de vérification et de simulation d'architectures. Outre la définition des services, l'interface contient une section de description du comportement (clause "*behavior*"). Le comportement est la description du fonctionnement observable du composant, par exemple l'ordonnancement des événements ou des appels aux services. C'est grâce à cette description que Rapide est en mesure de simuler le fonctionnement de l'application.

Enfin, une application est représentée par son architecture. Une architecture contient la déclaration des instances de composants ou modules. Toutes les instances sont représentées par des variables. Cette déclaration de variables est relativement dynamique car on définit un objet devant être présent dans l'architecture ou un ensemble (borné ou non) d'objets de tel ou tel type. Il n'est pas obligatoirement nécessaire de définir exactement les instances de composants présents, car cela peut se faire dynamiquement au fur et à mesure de l'exécution. Le reste de l'architecture contient la spécification des règles de connections entre objets. Ces règles sont composées d'une partie droite et d'une partie gauche. La partie gauche contient l'expression d'événements qui doit être vérifiée avant que les événements contenus dans la partie droite ne soient déclenchés, i.e. envoyés dans le système vers leurs destinataires. Par exemple, dans la Figure 38, la partie gauche contient un événement

correspondant à l'envoi par un composant `Client` du message `Send` avec un certain type de paramètres. La partie droite contient le déclenchement de l'appel du service `Receive` de tous les composants `Serveur` du système. Les parties gauches et droites peuvent être interconnectées par trois sortes opérateurs :

to : connecte deux expressions d'événement simples, i.e. ne définissant qu'un événement possible vers un composant. Si la partie gauche est vérifiée, alors l'expression de la partie droite permet le déclenchement de l'événement vers l'unique composant désigné par cette expression. Dans la Figure 38, il ne peut y avoir de connexion simple car la partie droite définit un ou plusieurs composants destinataires, `!r` désignant tous les serveurs existants dans le système.

||> : connecte deux expressions quelconques. Dès que la partie gauche est vérifiée, tous les événements contenus dans la partie droite sont déclenchés. Ils sont envoyés vers l'ensemble des destinataires désignés dans cette expression. L'ordre d'évaluation de cette règle de connexion est quelconque. Un déclenchement de cette règle de connexion est indépendant des autres déclenchements antérieurs ou postérieurs. L'ordre d'observation de ces déclenchements n'est pas significatif. Cet opérateur de connexion est appelé opérateur de diffusion.

=> : possède le même rôle que l'opérateur précédent mais ici l'ordre d'évaluation des règles est contrôlé. Un déclenchement de cette règle est causalement dépendant des déclenchements antérieurs de cette même règle. Cet opérateur de connexion est appelé opérateur pipe-line.

L'aspect intéressant de Rapide est sensiblement différent des autres langages étudiés tout au long de ce chapitre, car les aspects structuration, répartition ou génération d'exécutables ne sont pas visés. Par contre, il est intéressant d'étudier ce langage, car tout en se basant sur des abstractions identiques aux ADL (composants possédant une interface, interconnexions exprimées en dehors des composants,...), il permet d'exprimer toute la dynamique de l'application en terme d'ensembles d'instances de composants, de règles d'interconnexions qui évoluent en fonction du comportement des composants,... Essayons maintenant d'illustrer les caractéristiques du langage avec notre application pilote.

V.2.2. Exemple d'Utilisation

Nous n'entrerons pas vraiment dans les détails de Rapide qui sont encore nombreux notamment pour tous ce qui est expressions d'événements et règles pour la simulation. La section suivante présente l'exemple de l'annuaire qui est étendu comme celui présenté dans la section relative à CONIC.

La définition des composants correspond à la définition de leur interface, i.e. le type à partir duquel ils sont mis en oeuvre. Nous supposerons que le type *string* est déjà défini.

```
type Annuaire is interface
        provides -- une liste de services
        function Lookup ( cle : string)
                return string;
        function Pays ()
                return string;
...
end;
type Client is interface
        requires -- une liste de fonction demandée
        function Lookup_Annuaire (cle : string, pays : string)
                return string;
...
end;
```

Figure 39. Composants de l'Annuaire avec Rapide

La première architecture contient un client et un annuaire. La connexion entre les deux composants utilise l'opérateur de connexion de base *to* et les expressions d'événements sont simples.

```
 With Annuaire, Client
 architecture Application is
        client : Client; -- le client
        annuaire : Annuaire; -- l'annuaire
        ?C : Client; -- un client déclencheur d'événement
        ?cle : string; -- une clé transmise
        ?pays : string;
 connect
        ?C.Lookup_Annuaire(?cle, ?pays) to annuaire.Lookup(?cle);;
        -- un client qui génére l'événement Lookup_Annuaire avec un paramètre de
type string, est connecté à l'annuaire à qui on appelle la fonction Lookup avec le
même paramétre.
 end Application;
```

Compliquons l'exemple : on autorise l'existence de plusieurs clients qui appellent un ensemble d'annuaires propre à chaque pays. Le pays de l'annuaire est retourné par la fonction `Pays` de l'interface Annuaire, le pays demandé par le client est fourni en paramètre de la fonction requise `Lookup_Annuaire`.

```
 With Annuaire, Client
 architecture Application is
        clients : array [1..NUM_CLIENTS] of Client;
        annuaires : array [1..NUM_ANNUAIRE] of Annuaire;
        ?C : Client; -- un client déclencheur d'événement
        ?A : Annuaire; -- un annuaire possible
        ?cle : string; -- une clé transmise
        ?pays : string;
 connect
        ?C.Lookup_Annuaire(?cle, ?pays) and ?A.Pays() == ?pays
                ||>  ?A.Lookup(?cle);;
 end Application;
```

Figure 40. Architecture étendue de l'Annuaire en Rapide

Dans ce dernier cas (Figure 40), un client qui effectue une demande d'adresse en fournissant une chaîne de caractères correspondant au pays, peut avoir sa demande satisfaite s'il existe un annuaire dont la fonction Pays retourne la même chaîne de caractères. Dans un tel cas, cet annuaire est appelé. Cet exemple illustre ainsi la sélection dynamique de participants d'une interconnexion. Il est ainsi possible de modéliser des appels vers un composant qui possède des propriétés particulières, comme son existence, des valeurs d'attributs, la présence d'une fonction dans son interface, etc.

V.2.3. Avantages et Inconvénients

Rapide est un langage de description d'architecture, car il permet de modéliser une application en ne raisonnant que sur des ensembles de composants logiciels manipulés par une interface et sur le concept d'interconnexion entre ces composants. Il se situe donc complètement dans la classe de langage que nous étudions où la structure de l'application est clairement exhibée et les communications entre les éléments de cette structure sont indépendantes de la programmation des éléments.

L'intérêt de Rapide par rapport aux autres langages étudiés est de fournir des réponses au problème de la description de la dynamique d'une application, en terme de schéma de création des composants logiciels, de désignation dynamique des participants d'une interconnexion. Ceci est permis par l'utilisation de règles d'interconnexion qui sont "déclenchées" par le comportement des composants logiciels, que ce soit lors d'un changement d'état ou une demande de communication avec d'autres composants. De plus, Rapide prend totalement en compte différents modèles d'exécution en proposant

au niveau de l'interface des composants différents types d'appels de services (principalement synchrones et asynchrones) et en utilisant différents opérateurs d'interconnexion. Enfin, Rapide est basé sur un formalisme de description qui permet des vérifications statiques et dynamiques de l'exécution d'une application. Cette caractéristique est particulièrement appréciable dans le cas d'architecture de grande taille à forte complexité car elle évite l'installation et le déploiement d'une application pour effectuer les tests de validité de l'architecture.

Il faut noter que Rapide ne permet pas de générer une application. Son but est avant tout de vérifier la validité des architectures par des techniques de simulation de l'exécution. Nous ne pouvons donc pas vraiment effectuer de comparaison par rapport aux précédents langages mais certaines propriétés des applications, ainsi que certains de ses opérateurs, nous semblent pouvoir être intégrés à des ADL afin de renforcer la description de la dynamique de l'application.

V.3. Wright

Nous ne ferons pas une présentation exhaustive de Wright[All94] car ce langage de définition d'architecture est plutôt orienté vers la vérification formelle d'une architecture composée de composants et de connecteurs, chacun étant défini à l'aide d'un calcul proche de CSP[Hoa85]. L'utilisation de ce calcul permet d'assurer la faisabilité d'une interconnexion entre des composants et des connecteurs. Le principe de définition des composants et connecteurs est proche de celui de UniCon : les composants possèdent des ports pour communiquer avec l'extérieur et les connecteurs offrent des rôles. Wright porte son effort sur l'expression des règles d'interconnexion, i.e. la mise en relation des ports de composants et des rôles des connecteurs. Wright offre un formalisme qui permet de vérifier si les ports et rôles peuvent être compatibles d'un point de vue échange de paramètres comme du point de vue du modèle d'exécution de la communication. Ce calcul permet aussi de garantir l'absence d'interblocage dans une architecture.

Ci dessous, nous vous présentons un rapide aperçu de la syntaxe de Wright sans entrer dans les détails du calcul basé sur CSP.

```
System Annuaire
        Component Annuaire
                Port Lookup = spécifications de l'action de ce service
                Computation = Spécification du fonctionnement de l'annuaire
        Component Client
                Port Lookup_Annuaire = spécifications de l'action de ce service
                Computation = Spécification du fonctionnement du client
        Connector C-S-connector
                Role client = spécification du protocole
                Role serveur = spécification du protocole
                Glue = Protocole de communication, fonctionnement du connecteur
Instances
        annuaire : Annuaire
        client : Client
        cs : C-S-connector
Attachments
        annuaire.Lookup as cs.serveur;
        client.Lookup_Annuaire as cs.client
End Annuaire
```

Figure 41. Description Wright de l'annuaire

Il faut noter que Wright n'est pas dédié pour la production d'une image exécutable de l'application. Il autorise des vérifications mais ne construit pas d'application. De plus, la répartition n'est pas prise en compte car Wright s'occupe essentiellement de définir le comportement des composants et connecteurs en terme de processus communicants. Si des composants sont sur des sites différents, Wright

considère qu'il s'agit de composants dans des processus différents. L'intérêt de présenter Wright dans ce chapitre est de montrer l'existence d'un outil formel pour la vérification d'architecture d'applications. Un tel formalisme peut être couplé avec un langage de type UniCon ou Olan[Vio97a][Vio97b] pour réaliser un certain nombre de vérifications avant le déploiement de l'application sur les noeuds du système distribué.

VI. Discussion

L'objectif de cette section est de faire une synthèse des avantages et inconvénients des différentes approches et langages que nous venons d'étudier. Cette comparaison portera sur les points suivants :

- L'intégration de logiciels existants
- La description de l'architecture de l'application
- L'adhérence aux ressources du système distribué et en particulier la distribution des composants logiciels

La génération automatique de l'image exécutable de l'application distribuée.

Intégration de Logiciels

L'intégration de logiciel est l'action de décrire de manière homogène des entités logiciels hétérogènes du point de vue de leur type (e.g. du code source, des bibliothèques, des binaires, des applications,...), du langage de programmation comme de la plate-forme d'exécution. CORBA intègre du code source provenant de divers langages de programmation et homogénéise leur accès. DCOM s'occupe plutôt d'intégrer des composants sous diverses formes (source, bibliothèques, binaire) à condition que ce code soit chargeable dynamiquement. Polylith intègre du code compilable écrit dans divers langages de programmation, sans modification aucune sur leur structure interne car il agit au niveau de l'édition de lien. CONIC et Darwin ne se préoccupent pas vraiment de ce problème d'intégration, seul du code Pascal pour CONIC et C++ pour Darwin peuvent être utilisé. Enfin UniCon se présente comme un intégrateur de tout type de logiciel, à condition qu'il s'exécute sur Unix.

En plus des aspects langages de programmation, les composants logiciels de granularité importante (telle qu'une application complète) possèdent leur propre modèle d'exécution, de communication ou de synchronisation avec l'extérieur. Ces composants peuvent être actifs ou passifs, contenus dans des processus, des ensembles de processus. Toutes ces informations doivent d'une manière ou d'une autre être «remontées » par le processus d'intégration. CORBA et DCOM ne s'occupent pas de ceci, ils fournissent un modèle de programmation et d'exécution auquel les composants doivent s'adapter. Polylith ne remonte que peu d'information à ce sujet pour la simple raison que les entités logiciels de base sont des binaires objets. CONIC et Darwin se caractérisent par l'hypothèse forte que tout composant est associé à une activité (processus ou processus léger). Aucune information n'est remontée au niveau de leur interface au sujet de leur modèle d'exécution et de communication. UniCon se distingue par son système de type qui indique explicitement le modèle d'exécution associé au composant : un processus, un module logiciel passif, des données ou même un fichier. Ces informations constituent le type du composant.

L'action d'intégration de logiciels comporte une phase d'adaptation du système de type propre à l'entité avec celui du langage de construction de l'application répartie. Cette phase d'adaptation peut être relativement simple comme dans le cas de Polylith ou UniCon car la description des composants permet d'indiquer la provenance de divers types de logiciels ou plus complexe comme dans le cas des

ORB ou DCOM car les objets sont décrits avec un IDL qui génère différentes structures d'exécution nécessaires pour communiquer entre machines éventuellement différentes.

Caractéristiques	CORBA	COM	Polylith	Conic/ Darwin	UniCon	Olan
Hétérogénéité des langages de programmation	✔ (C, C++, Java, Smalltalk, etc.)	✘	✔ (C, C++, Lisp, ? ?)	✘ (un environnement Pascal et C++)	✔ (C, C++ pour le moment)	✔ (C, C++, Java, Python)
Hétérogénéité des entités logiciels	✘ Code Source	✘ Bibliothèques à chargement dynamiquement (*DLL*) Exécutable	✘ Code Source	✘ Code Source	✔ Code Source, Bibliothèques, Exécutables	✔ Code Source, Bibliothèques, Exécutables
Prises en comptes des modèles d'exécution	✘	✘	1 Composant = 1 processus	1 Composant = 1 processus	✔ *Explicite.* (notion de processus, de modules,...)	✔ *implicite* (notion de services synchrones et asynchrones)
Outils d'aide à l'intégration	Projection IDL vers les langages	Bibliothèques de classes	Description dans le MIL	Bibliothèque de classes	Description dans le MIL	Description dans le MIL

Tableau 3. Comparaison de l'Intégration de logiciels

Description de l'Architecture

La description de l'architecture est une des caractéristiques principales des différents modèles présentés. Toutefois, les variations de la notion d'architecture existent d'un projet à l'autre. Nous avons essayer de les classifier selon quatre axes :

- La description de l'ensemble des composants. Existe-t-elle, est-elle structurée, qu'est-ce qu'un composant ?
- La séparation des communications et des composants. Existe-t-elle, est-elle configurable, quelles sont les abstractions qui lui sont associées ?
- La description du schéma d'instantiation des composants, i.e. comment les composants sont-ils créés dans le temps.
- La description de communications complexes entre les composants : Est-il possible d'avoir plusieurs intervenants dans une communication, de décrire le choix dynamique des intervenants, de décrire des règles d'interconnexions entre composant pour restreindre les erreurs potentielles d'assemblages de composants ne pouvant interagir, de permettre de l'interconnexion entre composants non prévus pour interagir ?

La description de l'ensemble des composants d'une application varie de manière importante d'un langage à l'autre. CORBA et DCOM apportent les solutions les plus simples, seules les interfaces des objets sont décrites de manière indépendante de leur mise en œuvre. La vision de l'application se résume à l'ensemble des interfaces qui sont potentiellement utilisées. Polylith permet de décrire de manière plus complète l'ensemble des modules logiciels réellement nécessaire à l'application. Toutefois, aucune structuration de l'ensemble des modules n'existe, l'application est un ensemble «plat » de modules. CONIC, comme Darwin et UniCon, vont un pas en avant dans la description de l'architecture car ils raisonnent sur des instances de composants qui sont réellement utilisées lors de l'exécution. L'architecture n'est plus simplement une manière d'identifier les modules logiciels requis mais aussi les structures d'exécution qui sont créées. De plus, la structuration de ces instances existe en proposant de modéliser une application comme une hiérarchie de composants, certains contenant le code et les autres agissant comme des regroupements de sous composants. Rapide et Wright n'offrent aucune structuration hiérarchique de l'ensemble des composants.

La séparation du code de la communication de celui de mise en œuvre des composants logiciels est une autre caractéristique des langages de définition d'architecture. Dans le cas d'applications réparties, ceci permet de créer ces composants logiciels sans se soucier de l'utilisation ou de la programmation des communications à distance entre sites. L'environnement de construction d'application associé aux langages précédemment étudiés, permettent dans de nombreux cas de générer automatiquement l'utilisation de tel ou tel mécanisme de communication à distance. Ceci est un facteur important de simplification de la construction d'applications réparties. De plus, cela permet aussi de pouvoir modifier les propriétés de la communication entre composants sans toucher à leur mise en œuvre. CORBA et DCOM n'entrent donc pas dans ce cas de figure car les communications entre objets sont complètement noyées dans le code des objets ou composants. Polylith, CONIC et Darwin permettent de spécifier les liaisons entre composants, i.e. les informations concernant les composants appelants et appelés dans une communication. Ceci permet notamment de répartir a posteriori les composants car il n'y a aucun lien entre ces composants. Ils ne se désignent pas mutuellement car ceci est fait par le MIL ou le langage de configuration. Toutefois, ces trois projets ne permettent pas de spécifier facilement les autres propriétés d'une communication telles que le mécanisme utilisé, les adaptations de transmission de données (propriété très utile car interconnecter des composants qui n'ont pas été initialement conçus de concert demande souvent des adaptations entre signatures d'appels), l'ordonnancement d'appels, les propriétés de synchronisation, etc. Polylith permet de changer de mécanisme de communication en utilisant des versions différentes de ses bus logiciels. Darwin permet de programmer dans son support d'exécution associé des objets de communication différents. Toutes ses opérations ne sont pas immédiates et demandent d'agir au sein de la mise en œuvre du support d'exécution. Aster[Iss96][Iss97] est un MIL proche de Polylith qui permet de choisir automatiquement le support de communication en fonction de certaines propriétés requises par les composants. Un formalisme de description de contraintes de qualité de services requises pour les communications est utilisé pour décider par inférence du meilleur support de communication. UniCon permet de spécifier certaines propriétés des communications entre composants, notamment le mécanisme de communication utilisé et les composants autorisés à utiliser ces mécanismes. Rapide et Wright vont plus loin car l'ordonnancement des communications, la synchronisation, le modèle d'exécution de la communication peuvent être spécifiée. Il faut toutefois noter que ces deux langages ne permettent pas pour l'instant de produire une image exécutable de l'application.

Le dernier point pour la définition d'architecture d'applications concerne la spécification de certains aspects dynamiques comme le schéma de création des différents composants au fur et à mesure de

l'exécution de l'application ou l'évolution des interconnexions de composants lorsque l'application le demande. Polylith et UniCon n'apportent aucune solution à ces problèmes car l'architecture qu'ils définissent est complètement statique. Tous les composants de l'application sont créés au démarrage de l'application. CONIC, mais surtout Darwin, offrent la possibilité de spécifier au travers d'algorithmes paramétrables la manière de créer des composants. Darwin permet aussi de spécifier la création dynamique des composants pendant l'exécution de l'application au lieu de les installer tous en début d'exécution comme le font les MIL ou UniCon. Toutefois Darwin ne permet pas d'exprimer de la dynamique au niveau des communications. Une interconnexion de composant ne peut pas évoluer dynamiquement, ni ne peut paramétrer les destinataires ou émetteurs avec des attributs qui évoluent lors de l'exécution. Ces genres de caractéristiques sont au cœur de Rapide, qui permet de décrire toutes sortes d'interconnexions très dynamiques.

L'Adhérence aux ressources

L'adhérence aux ressources est ici à prendre au sens de la capacité de répartir un ensemble de composants qui forment une application sur un système distribué, qui possède des caractéristiques particulières (nombre de machines, types de machines, topologie d'interconnexions des machines, autorisations d'accès, etc.) et peut évoluer avec le temps.

Le principal attrait des langages que nous venons d'étudier est de définir des modèles de construction permettant de répartir des composants dans une phase séparée de celle de conception. Ceci permet ainsi de s'affranchir des contraintes liées à la répartition lors de la programmation et de produire une architecture d'application qui puisse s'adapter dans un deuxième temps au système distribué qui va l'héberger. Une troisième phase est à envisager : celle qui permet à un administrateur du système ou de l'application de modifier au fil du cycle de vie de l'application la répartition des composants logiciels sans avoir à toucher au code de mise en œuvre des composants.

CORBA permet de modifier les sites qui hébergent les objets de l'application à la condition qu'un serveur de nom permettent de retrouver des objets en cours d'exécution sur les sites du système. Toutefois, la distribution des composants est faite de manière entièrement manuelle, lors du lancement des différents serveurs. DCOM fonctionne un peu différemment car il permet à des clients de lancer des composants à distance. Il est donc possible de programmer la répartition des clients. L'intérêt est de contrôler la répartition par programme mais l'inconvénient est que le changement du schéma de répartition des composants demande de se plonger dans le code qui l'effectue. Polylith, CONIC et UniCon permettent de spécifier les sites d'exécution des composants. Le support d'exécution associé à ces langages procède, lors de l'installation des composants, à la création à distance de composants. Certains support comme celui de Polylith et CONIC mettent en place les structures de communications adéquates entre les composants en fonction de leur localisation. Enfin, Darwin permet de spécifier aussi le site d'exécution mais il offre une légère subtilité. Tous les sites du système distribué sont désignés par des nombres. Il est alors possible de piloter le positionnement des composants dans le programme de configuration avec des algorithmes. La localisation n'est plus simpliste, car elle peut être définie par algorithme en fonction de la topologie du système.

Génération d'Exécutables

Les langages que nous venons d'étudier ont tous pour but de permettre la construction d'applications réparties mais il est possible de les diviser en trois groupes : les langages pour programmer une

application répartie, ceux pour la décrire et la générer et ceux pour la décrire et vérifier la validité de son comportement.

La première catégorie de langage comprend les IDL et le mécanisme de répartition qu'ils permettent d'utiliser. CORBA et DCOM sont les exemples que nous avons présentés. Ils ne permettent pas vraiment la génération automatique complète de l'application car le facteur de répartition n'est pas transparent pour le programmeur. Il faut qu'il prenne en charge l'utilisation du mécanisme de communication ainsi que les opérations de répartitions et d'installation des composants.

La seconde catégorie de langage regroupe les MIL, les langages de configuration et UniCon. Ces langages permettent de générer automatiquement l'image exécutable de l'application répartie. Une description de l'architecture associée au code des composants logiciels permet d'installer et de faire exécuter l'application sur le système répartie. Ceci se fait à des degrés divers selon les projets mais tous utilisent un support d'installation et d'exécution spécifique pour y arriver. La génération de code est différente d'un projet à l'autre. Dans le cas de Polylith, des vérifications de types sont faites au niveau des interconnexions et le compilateur génère du code qui pilote le bus logiciel sous-jacent. Dans le cas de CONIC et de Darwin, le compilateur (que l'on devrait appeler le préprocesseur) génère des ordres qui sont exécutés par le support sous-jacent. Enfin UniCon permet de générer du code utilisant au mieux tous les mécanismes présents sur un système de type Unix et ainsi d'éviter au programmeur tout le travail d'utilisation de ces mécanismes.

Enfin, la dernière catégorie comprend les langages de définition d'architecture (Rapide et Wright) dont le but est de vérifier et simuler le comportement de l'application distribuée construite par assemblage de composants logiciels. Ces langages ne permettent malheureusement pas de générer automatiquement le code logiciel correspondant aux propriétés énoncées dans le langage.

Chapitre 3
Langage de Configuration OCL

I. Architecture générale

L'environnement de configuration Olan est un ensemble d'outils destiné à configurer des applications réparties. Le terme configuration désigne d'une part l'action de définition et de spécification de l'architecture de l'application et de son mode d'exécution, et d'autre part l'action d'installer, de déployer et d'exécuter l'application répartie sur le système informatique qui l'héberge. L'environnement de configuration Olan comprend à cet effet deux catégories d'outils : les outils de description de l'architecture de l'application et les outils d'installation et d'exécution regroupés au sein du support de configuration Olan.

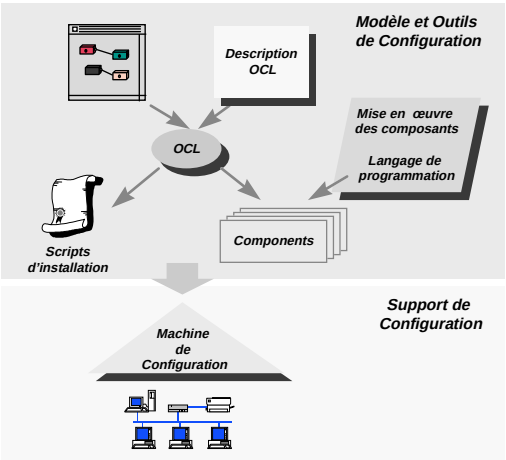

Figure 42. Architecture générale de l'environnement Olan

Outils d'Architecture

La définition de l'architecture de l'application repose sur un modèle d'assemblage de composants logiciels. Un langage de configuration, OCL (Olan Configuration Language), dérive de ce modèle. Il permet de spécifier des architectures d'applications dont les abstractions de base sont les composants, entités d'intégration et de structuration de logiciels, et les connecteurs, entités de gestion de la

communication entre composants. Les outils mis à disposition du concepteur de l'application sont un compilateur du langage de configuration OCL et des outils visuels de conception d'architecture.

Ces outils ont pour cibles la génération d'une image exécutable de l'application décrite ainsi que la génération d'instructions pour l'installation et le déploiement de l'application sur un ensemble de sites. Ces cibles peuvent ensuite être exploitées par le support de configuration Olan.

Support de Configuration

Le support de configuration est composé d'une machine de configuration dont le but est d'interpréter les cibles produites par le compilateur pour effectuer l'installation des composants logiciels sur différents sites et de mettre en place les canaux de communication entre ces composants en totale concordance avec les spécifications contenues dans l'architecture.

Dans ce chapitre, nous vous présentons la première partie du travail réalisé pour cette thèse. Il s'agit de la brique de base de l'environnement de configuration Olan : le modèle de construction et de configuration et le langage de configuration OCL. Nous allons présenter les différentes abstractions du modèle Olan, les constructions syntaxiques associées ainsi que la chaîne de configuration d'application. L'exemple de l'application Annuaire utilisé tout au long du chapitre précédent, sert d'illustration à notre discours. Il a été progressivement étendu par rapport à l'exemple initial afin de montrer l'intérêt et la souplesse du langage OCL à décrire et construire des applications complexes en partant de briques logiciels relativement simples. Toutefois, pour des raisons de longueur ; seules des portions de l'exemple sont inclues dans ce chapitre.

II. Le langage OCL

II.1. Choix de conception

A l'origine de l'environnement, les actions de recherche du projet étaient principalement dirigées autour des systèmes d'exploitation répartis à objets[Bal91]. L'utilisation de tels systèmes était principalement destinée aux concepteurs et programmeurs d'applications réparties qui manipulaient un langage de programmation spécifique au système, intégrant dans le cas de Guide les concepts objets et la répartition[Kra90]. L'expérience de programmation d'applications sur ce type de systèmes nous a vite montré que les objets programmés étaient généralement de petite taille et pouvaient, par le biais du système, être «aléatoirement » répartis sur l'ensemble des machines interconnectées. Le programmeur avait la liberté de définir le site d'exécution de l'objet mais par manque de structures pour exprimer le regroupement d'objets, le choix de la répartition était très difficile. Il était fréquent, à cause du modèle d'exécution du système Guide, d'avoir des objets fortement interdépendants - i.e. qui s'appellent mutuellement - présents sur des sites distincts.

L'idée première de l'environnement Olan était de fournir au programmeur des abstractions pour regrouper a priori des objets et les localiser sur des sites. Ces abstractions devaient être manipulées au niveau d'un langage distinct des langages de programmation pour pouvoir récupérer du code existant et ne pas à avoir à inventer un nouveau langage de programmation, ni à effectuer le travail conséquent de modifier un langage de programmation existant.

Rapidement le concept de composant logiciel, en tant que regroupement d'objets répartis fonctionnellement corrélés, s'est imposé. Le composant nécessite la présence d'une interface homogénéisant les accès à des mises en œuvre utilisant divers langages de programmation, rôle de l'IDL. Le passage vers un langage de type Langage d'Interconnexion de Modules (MIL) s'est fait en conformité avec nos objectifs d'indépendance de la description et de l'implémentation d'une application avec le support de communication, car un MIL permet d'exprimer les caractéristiques des communications entre modules logiciels de manière externe.

Pour toutes ces raisons dictées en grande partie par les enseignements des actions de recherche passées, nous avons pris pour base de définition de OCL, d'une part l'IDL de CORBA pour la définition des interfaces d'accès, en particulier son système de type et ses projections vers les langages courants de programmation, et d'autre part Darwin qui offre, en plus de la description des interconnexions, une hiérarchisation des composants logiciels permettant de structurer l'application de manière plus fine que dans un MIL.

Nous allons montrer dans la suite, les principales caractéristiques du langage OCL au travers des quatre angles de vue exhibés dans le chapitre précédent :

- l'intégration de logiciel,
- la description de l'architecture,
- l'adhérence aux ressources physiques, en particulier la répartition, et
- la génération des images exécutables des applications.

II.2. Intégration de logiciels

II.2.1. Composant Primitif

Le composant primitif est l'entité d'encapsulation de logiciel existant. C'est lui qui fournit une interface d'accès à un ensemble d'entités logicielles qui sont décrites au travers de la construction syntaxique *module* que nous présentons dans la section suivante. Le composant primitif est composé d'une interface et d'une description des modules encapsulés. Cette dernière description est appelée *l'implémentation*.

Interface

L'interface permet de décrire de manière la plus complète possible l'accès aux opérations et aux structures de données fournies par le composant. Ce n'est pas son unique rôle, car elle contient aussi des informations sur toutes les opérations requises par le composant afin de garantir un fonctionnement correct. Ces opérations requises sont assimilables en même temps aux requis de Polylith - références externes résolues traditionnellement par un éditeur de lien - et aux ports de communications requis de Darwin - information sur le mode de la communication avec d'autres composants -. La terminologie OCL associée à ces opérations est le service qui contient à la fois les indications sur rôle de l'opération et le mode de communication devant être utilisé pendant l'exécution. Le choix du mode de communication est fait en conformité avec le schéma de communication des

modules intégrés. Par exemple, lorsque le module effectue un appel asynchrone, le service devra refléter ce mode d'appel.

Dans OCL, un service est définit par son "sens" d'accès, un nom unique dans l'interface où il est défini, une signature qui comprend des paramètres typés et le type de synchronisation associée à la communication. La signature utilise la syntaxe IDL de l'OMG que ce soit pour les types ou le sens des paramètres (`in` et `out`). Le Tableau 4 résume les différents types de service existants en OCL avec une explication sur le mode de communication associé à chacun de ces services.

Type de service	Sens d'appel	Synchronisation	Flot d'exécution
provide ●	réception	synchrone	flot appelant
require ○	demande	synchrone	flot appelant
react ■	réception	asynchrone	flot propre au composant
notify □	demande	asynchrone	flot appelant asynchrone

Tableau 4. Services d'une Interface OCL

Ce tableau demande quelques informations complémentaires. Chaque type de service est un reflet du modèle d'exécution de la mise en œuvre de ce service. En effet, un appel de méthode ou de fonction standard correspond à un appel synchrone, le flot appelant allant exécuter la fonction appelée. Ce schéma correspond aux deux types de services `provide` et `require`, le second correspondant à l'appelant et le premier à l'appelé. Néanmoins, il est fréquent que le service appelé s'exécute avec son propre flot d'exécution de manière asynchrone par rapport à l'appelant. L'utilisation des services `react` et `notify` sert à décrire ce type de schéma de communication. Il existe toutefois une contrainte au niveau de OCL concernant les services asynchrones : ils ne peuvent recevoir de paramètres en retour, leur signature ne peut contenir que des paramètres de type `in`. Ce dernier schéma de communication est comparable aux «actions » de Rapide ainsi que les types de méthodes «oneway » de CORBA.

L'interface peut aussi contenir des définitions d'attributs. Les attributs sont des variables typées correspondant ou non à des variables contenues dans l'implémentation du composant. Les attributs permettent de rendre accessibles au niveau du langage de configuration certaines données contenues initialement dans le code intégré et qui peuvent éventuellement changer en cours d'exécution. Cette propriété est intéressante, car elle ouvre la porte aux choix dynamiques de composants dans des interconnexions par rapport à la valeur de l'attribut qui peut varier en cours d'exécution. Les attributs actuellement disponibles dans OCL ne peuvent pas être modifiés dans le langage de configuration, ils sont uniquement modifiables par le code primitif.

Voici l'exemple OCL de l'interface du composant Annuaire. L'annuaire possède un attribut `Pays` qui indique le lieu géographique des personnes contenues dans l'annuaire. Il offre un service de récupération de l'adresse d'une personne, `Lookup`. Pour illustrer les différents types de services, nous introduisons un nouveau service par rapport à l'exemple décrit dans le chapitre précédent. Ce service envoie, de manière asynchrone, un rapport des statistiques d'utilisation de l'annuaire, `SendStats`.

```
From oil::types import * // Inclusion de fichiers de définition d'interfaces.

Typedef sequence<char> statStruct; // déf d'un type séquence illimitée de
caractères

// Définition de l'interface de l'annuaire
Interface AnnuaireItf {
        readonly attribute string Pays;
        provide Init()
        provide Lookup( in string cle, out string email);
        notify SendStats(in statStruct statReport);
}
```

Figure 43. Interface OCL de l'Annuaire

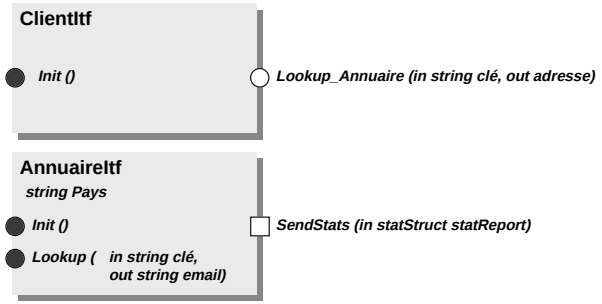

Figure 44. Représentation graphique des interfaces OCL

Dans le modèle Olan, nous avons adopté une représentation graphique proche de celle de Darwin associée à une représentation textuelle. Elle sert de base aux outils de visualisation de l'environnement Olan. La synoptique utilisée est indiquée dans le Tableau 4.

Le concept d'interface de OCL est proche de celui de l'IDL avec toutes les informations propres à des langages de configuration. L'utilisation de la syntaxe et du système de type de l'IDL permet de fournir le plus d'information possible au monde extérieur - dans la limite des possibilités d'un système de types et d'un langage déclaratif -.

Implémentations

La mise en œuvre d'un composant correspond à un ensemble d'entités logicielles de divers types. OCL permet la description de cet ensemble d'entités, chacune d'entre elle correspondant à un `module`. La section *Implementation* d'un composant primitif contient la liste des modules utilisés ainsi que la liaison explicite entre les services déclarés dans l'interface et ceux définis dans les modules. Il en est de même pour les attributs de l'interface dont la valeur est celle d'un attribut contenu dans un des modules. En outre, il existe une contrainte concernant le contenu de l'implémentation : les modules doivent tous être de même type, par exemple des fichiers source C ou des exécutables,...

La syntaxe d'une implémentation est illustrée avec l'exemple suivant de l'annuaire. Une implémentation est identifiée par un nom et elle "*implémente*" une interface. L'opérateur syntaxique "**:**" est ici pour indiquer la relation entre l'identifiant de l'implémentation et une interface. Le reste de l'implémentation contient les liaisons entre les services et attributs de l'interface et ceux présents dans les modules. L'absence de liaison entre une déclaration de l'interface et un module déclenche des avertissements au niveau du compilateur du langage, car cela peut induire un possible dysfonctionnement du composant.


```
from oil::annuaire import AnnuaireItf // inclusion de la déf de l'interface

primitive implementation AnnuaireImpl : AnnuaireItf
        use AnnuaireMod // liste des modules utilisés
{
        // liste de la projection de l'interface vers les modules
        Pays -> AnnuaireMod.Pays; // L'attribut Pays existe dans le modules
        Init -> AnnuaireMode.Init()
        Lookup() -> AnnuaireMod.Lookup();
        AnnuaireMod.Stat() -> SendStats();
}
```

Figure 45. Implémentation Primitive de l'Annuaire en OCL

Nous pouvons noter que cette implémentation ne contient qu'un seul module, leur nombre n'étant pourtant pas limité. La clause `use` contient le nom des modules intégrés, modules qui doivent donc avoir été antérieurement définis. Les clauses de projection matérialisées par l'opérateur "**->**" ressemblent du point de vue syntaxique aux interconnexions que nous verrons dans une section suivante. Il faut toutefois noter qu'aucune transformation de paramètres n'est autorisée, la projection étant une simple association entre services de l'interface et des modules. Il est par contre possible de procéder au renommage des services ou attributs.

Composant

Le composant dans le modèle OCL peut être qualifié de glu entre une interface et une implémentation. Cette glu est similaire au concept de type de composant de Darwin ou UniCon dont les architectures sont composées d'instances de ces types. La syntaxe OCL d'un composant est la suivante :

```
component Annuaire {
        interface AnnuaireItf;
        implementation AnnuaireImpl;
}
```

Figure 46. Composant OCL

Ce niveau de syntaxe supplémentaire, "**`component`**", n'est pas facilement justifiable pour les composants primitifs, car finalement une implémentation réalise une interface est donc le composant pourrait à juste titre être assimilé à son implémentation ou à son interface comme dans Polylith, Darwin et UniCon. Néanmoins, cela apporte une souplesse d'évolution des composants, car le changement d'une implémentation, à interface égale, ne perturbe aucunement les autres composants de l'application ni l'architecture globale de l'application. De plus, nous verrons dans la section II.4. que cela permet d'associer des politiques de répartition différentes pour les mêmes types de composants.

Intérêts

Le composant primitif est la brique de base du modèle de construction Olan, car c'est lui qui permet de décrire et de faciliter l'accès aux entités logicielles. La description qu'il fournit au travers d'une interface est homogène peu importe l'entité encapsulée, son langage de programmation ou sa plate-forme d'exécution. Ils peuvent être utilisés dans n'importe quelle architecture à condition de satisfaire ses besoins fonctionnels et de modèle d'exécution.

II.2.2. Modules

Le module est l'abstraction OCL qui décrit une entité logicielle. Un module est défini par une interface, le type de l'entité encapsulée et des informations sur la localisation et les paramètres

d'utilisation des fichiers constituant le module. L'interface est identique à celle des composants primitifs sans restriction particulière. Le type de l'entité doit être indiqué explicitement en début de déclaration du module : elle a pour syntaxe une chaîne de caractères qui permet à l'environnement de compilation de charger le générateur de code spécifique à ce langage ou cette entité. Par exemple, le type `"C"` permet de gérer le code d'emballage d'un fichier source écrit en langage C, `"Python"` permet de faire la même chose pour du code Python[Ros95]. Les informations du module dépendent du type de l'entité mais elles contiennent généralement le chemin d'accès au fichier source ou à l'exécutable, des options de compilation ou de lancement, etc.

Type de module	Informations	Etat
Python	path: chemin d'accès au fichier sourceFile: nom du fichier source	Existant
C, C++	path : idem précédent sourceFile : idem précédent options : options de compilation	Existant
Java, AAA[4]	java : chemin de l'interpréteur classPath : chemin d'accès aux classes Java destination : chemin de création des classes Java version : version nécessaire du compilateur Java options : options de lancement du compilateur java class : nom de la classe Java intégrée	Partiellement existant
Exécutable	path : idem précèdent command : nom de l'exécutable options : paramètres de lancement	En cours de réalisation

Tableau 5. Types de modules OCL

Le module contenu dans le composant Annuaire indique l'emplacement des fichiers source contenant le code de l'annuaire. Ce code est écrit en Python et la syntaxe du module est la suivante :

```
from oil::annuaire import AnnuaireItf

module "Python"  AnnuaireMod : AnnuaireItf { // on réutilise l'interface du
                                              // primitif
        path: "${OLAN_SRC}"+"/examples/annuaire";
        sourceFile: "AnnuaireMod.py";
}
```

Figure 47. Module Annuaire en OCL

L'environnement Olan utilise toutes les informations du module pour générer automatiquement les fichiers d'emballage du code source afin de les intégrer dans les applications construites avec OCL. L'emballage de code source est fortement dépendant du type du module. Toutefois, il est possible de distinguer deux concepts : les talons (stub) et les «empaqueteurs » (wrapper).

[4] Nous verrons dans le chapitre ? ? à quoi correspond le type AAA.

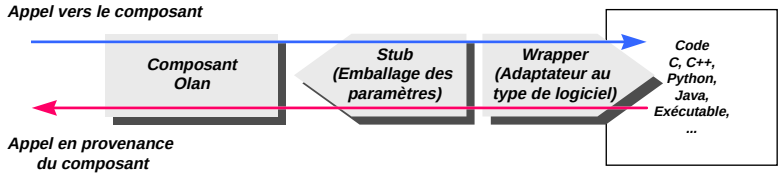

Figure 48. Intégration des modules.

Les talons transforment les paramètres du format propre à l'entité vers le format interne de représentation des paramètres de l'environnement et vice versa. Le wrapper permet de diriger les appels de services ou d'attributs vers l'entité. Il peut donc effectuer des opérations d'appels de fonction jusqu'au lancement d'exécutable ou de processus pour honorer la demande de service. Chaque wrapper est dépendant du type de l'entité. Ils permettent aussi de faire le lien entre une communication requise par le logiciel avec le talon et d'autres composants de l'environnement Olan.

```
## Exemple de fichier python
import AnnuaireRetrieval ## contient le code d'accès à l'objet annuaire
import threads

pays = "" ## C'est l'attribut du composant primitif
annuaire = {} ## C'est l'objet contenant l'accès aux informations
countIncrement = 0 ## Nombre d'accès à l'annuaire

def init ():
        '''oil:  provide () '''
        global pays
        global annuaire
        ## Ouverture de l'annuaire désiré dont le code est dans un autre module
        annuaire = AnnuaireRetrieval.openAnnuaire(pays)
        ## annuaire est un objet dont une méthode permet de recuperer
        ##l'adresse désirée
        ## et une autre permet de recevoir les statistiques.

def Lookup(cle):
        '''oil: provide (in cle, out email) '''
        global annuaire, countIncrement
        ## Toutes les N fois, on envoi un rapport d'utilisation
        global countIncrement
        countIncrement = countIncrement + 1
        if countIncrement >= N:
                ## Construction du rapport, c'est une structure dont les champs
                ## sont Nombre d'accès, personnes la plus fréquemment utilisée
                ## La projection d'une structure en Python est un dictionnaire
                ## dont les clés sont les noms de champs
                threads.start_new_thread(SendStats, (annuaire.getReport()))
                ## Appel asynchrone (par rapport au processus initial) en
                ## créant un flot concurrent d'exécution pour l'envoi des stats.
                countIncrement = 0
        return annuaire.getAdresse(cle)

def SendStats():
        '''oil: notify (in statReport) '''
        ## Il n'y a pas de code, le wrapper permet de dérouter tout appel à
        ## cette fonction vers
        ## du code qu'il implante lui-même et qui communique avec l'extérieur
```

Figure 49. Module Python de définition de l'Annuaire

Prenons l'exemple du module Python définissant la mise en œuvre de l'Annuaire (cf. Figure 49). Ce module offre les fonctions `Lookup` et `Init` (`provide`) et requiert l'envoi asynchrone des statistiques d'utilisation de l'annuaire par la notification `SendStat`. Dans le cas d'un fichier source écrit en Python, le talon et le wrapper du module sont générés entièrement automatiquement. Les indications du type `''' provide (in cle, out email) '''` sont des chaînes de documentation (une caractéristique du

langage Python) qui permettent à l'environnement de compilation Olan de produire automatiquement le code du wrapper pour diriger les appels de l'extérieur vers ces fonctions ou de diriger des appels requis vers le wrapper. Le cas de fichiers C n'est pas aussi immédiat que Python, le wrapper n'est pas entièrement automatisé, le programmeur doit compléter un squelette. Nous n'entrerons pas plus dans les détails, car ce n'est pas l'objet de cette thèse et le travail de réalisation n'est que partiellement réalisé, en particulier le langage C, les exécutables ou les bibliothèques ne sont que partiellement terminés. Ce même argumentaire s'applique aussi au problème de projection des interfaces vers un langage de programmation donné qui ne suit pas exactement les indications de l'OMG.

II.2.3. Avantages et Inconvénients

L'approche de construction du composant primitif OCL est intéressante pour plusieurs raisons. Du point de vue de l'architecture d'une application, le composant primitif est la brique utilisée pour l'assemblage de composants. Rien ne le différencie d'un point de vue interface des autres composants. Seule son implémentation est particulière. Ce point de vue est identique à l'approche choisie par UniCon et Darwin où l'interface du composant primitif ne se distingue pas des autres composants. La différence entre le composant primitif OCL et les primitifs Darwin ou UniCon vient de l'implémentation. Dans Darwin, le code associé doit se trouver dans le constructeur d'une sous-classe particulière de celles contenues dans les bibliothèques fournies par le support d'exécution. Dans UniCon, l'implémentation primitive est à rapprocher du concept de modules de OCL, elle indique clairement les informations pratiques d'accès au code. La limitation de UniCon vient de l'impossibilité d'inclure plusieurs modules dans un même primitif alors que OCL le permet. De plus, le changement du code intégré dans un primitif demande la construction d'un nouveau composant alors que les modules dans OCL permettent de garantir la conservation de la définition du primitif tout en changeant de module. De plus, un même module OCL peut être utilisé dans plusieurs primitifs différents.

La réutilisation d'une entité logicielle au travers d'un composant primitif est facilitée par les informations sur le modèle d'exécution de l'accès ou des demandes vers ou en provenance des services, d'une manière similaire à Rapide[Luc95]. Il est facile de voir qu'un service de type `react` indique la présence au sein du primitif d'un (ou plusieurs) flot d'exécution propre qui prend en compte et réalise le service demandé. L'information est moins clairement exposée que dans UniCon qui affiche explicitement le type du module (types *Process*, *Computation*, *Module*,...) mais cela permet de mixer au sein d'un primitif des services réalisés par un flot externe traversant le composant ou par un flot interne propre au composant. Pour plus d'informations au sujet du modèle d'exécution des composants primitifs, le lecteur pourra se reporter à [Vio97a][Vio97b] où la sémantique opérationnelle de OCL y est présentée. Le problème de notre approche reste l'adéquation entre les différents services et le code effectif des modules, à la charge du programmeur de définir des appels et demandes synchrones ou asynchrones en fonction du code existant.

Les inconvénients de notre approche sont essentiellement liés au travail de création du module. Il faut d'une part décrire les interfaces, les implémentations primitives, les modules et la glu, i.e. le composant. Ce travail est donc relativement long par rapport à la description Darwin et dans une moindre mesure UniCon. Ce travail est compensé par la possibilité d'inclure plusieurs modules différents, plusieurs langages de programmation en retravaillant au minimum le code que l'on intègre. Dans le cas du langage Python, le travail d'intégration du code se limite à la documentation des

fonctions requises ou fournies. Dans le cas du langage C ou Java, il faut compléter le wrapper et utiliser la projection de OCL vers ce langage.

II.3. Description de l'architecture

III.3.1. Composant Composite

Les principes de bases de OCL sont relativement proches des langages de configuration et d'architecture. Une application est composée de composants primitifs mais aussi de composants composites qui servent à la fois d'entité de description de la configuration et d'entité de structuration d'une application en modules ou composants coopérants. Les composites permettent de former une hiérarchie de composants, hiérarchie partiellement ou totalement réutilisable dans diverses applications. Le concept d'application de OCL est un composite particulier, le sommet de la hiérarchie. On peut comparer l'application au concept de *System* de CONIC ou d'*Architecture* de Rapide.

Chaque composite est formé d'une interface identique à celle des primitifs décrite dans la section I.2.1. Ils sont aussi formés d'une mise en œuvre ou "*Implémentation*" qui contient la déclaration des "*sous composants*" nécessaires ainsi que les interconnexions entre eux.

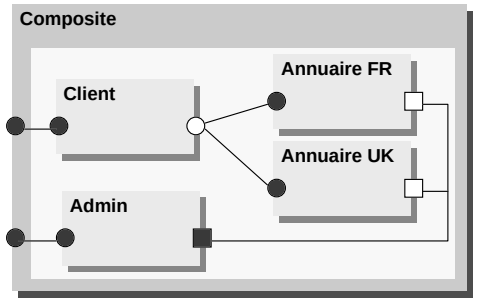

Figure 50. Principe de construction des composites dans OCL

L'implémentation d'un composite contient en premier lieu les déclarations des instances de composants utilisées avec éventuellement des paramètres de création de ces composants. Ces déclarations correspondent au schéma d'instantiation des sous composants d'un composite lorsque ce dernier est créé. Ce schéma est essentiellement statique, car tous les sous composants sont créés à un seul instant, lors de la création du composite. L'utilisation du constructeur `instance` permet de déclarer les sous composants à instantier.

La seconde fonction de l'implémentation est de définir la projection des services et attributs de l'interface du composite vers les sous composants susceptibles de les réaliser ou d'en avoir besoin. Cette projection se fait de la même manière que pour les composants primitifs en utilisant l'opérateur d'interconnexion ->. L'exemple du composite contenant l'architecture de l'application annuaire illustre ces deux concepts : la création de sous composants et la projection.

```
 Implementation AppliImpl : AppliItf
        use AnnuaireItf, ClientItf, AdminItf // Inclusion des interfaces des sous
composants nécessaires
 {
        // Définition des sous composants
        client = instance ClientItf;
        annuaireFr = instance AnnuaireItf("FRANCE");
        annuaireUk = instance AnnuaireItf("UK");
        admin = instance AdminItf;

        // Projection des services de l'interface vers les services des sous
composants
        LanceClient() => client.Init();
        LanceAdmin() => admin.Init();
        ...
```

Figure 51. Implémentation d'un composite en OCL

Nous pouvons remarquer au travers de cet exemple que la déclaration d'instances utilise des interfaces et non pas des composants. L'association entre l'interface et un composant qui réalise cette interface est faite plus tard, lors de la définition de la glu, i.e. du composant. Cette caractéristique permet de changer simplement les implémentations de composants sans modifier l'architecture, i.e. l'implémentation d'un composite.

```
 Component Appli {
        interface AppliItf;
        implementation AppliImpl {
                AnnuaireItf is Annuaire;
                AdminItf is Admin;
                ClientItf is Client;
        }
 }
```

III.3.2. Interconnexion de Composants et Connecteurs

L'interconnexion est l'action de spécifier avec qui et comment les composants communiquent pendant l'exécution. Chaque interconnexion contient l'initiateur de la communication et le ou les destinataires. Une interconnexion ne décrit jamais le moment exact de la communication. L'interconnexion a pour unique rôle de dire que dès lors que l'initiateur effectue une demande de communication, celle-ci se passera en utilisant tel mécanisme ou protocole pour envoyer une requête vers le ou les services des destinataires.

Une interconnexion met en jeu deux parties : la partie gauche contient le service du composant initiateur d'une communication, la partie droite les services des composants destinataires. Pour le moment, le modèle d'interconnexion OCL ne permet de décrire que des communications de type 1 composant vers N composants. Des études sont en cours pour étendre à des schémas d'interconnexion plus complexes de type M vers N [Mar97]. Tout comme dans UniCon, l'objet qui spécifie et met en œuvre la communication est le *Connecteur*. La clause `using` d'une interconnexion indique le type de connecteur utilisé. Pour finir, il est aussi possible d'ajouter à l'interconnexion une clause `do` de transformation de paramètres qui permet à des services de signatures différentes de pouvoir être malgré tout mis en relation.

Reprenons l'exemple de l'annuaire. L'application est composée de plusieurs instances de composants : le client, l'administrateur et les annuaires. Son implémentation contient la définition des interconnexions de ces composants.


```
        ....
        client.Lookup_Annuaire (cle, adresse) -> annuaire.Lookup(cle, adresse)
                using syncCall();
        annuaire.SendStats( statReport) -> admin.GetReport( report)
                using asyncCall()
                do { // clause de transformation de paramètres
                        report = annuaire.Pays + " : "+ statReport;
                        // concaténation du nom du pays en début de rapport.
                        // Le rapport est une séquence illimité de caractères,
                        // d'ou la possibilité d'utiliser les opérateurs de
                        // concaténation de chaîne +
                }
        ...
```

Figure 52. Interconnexions dans l'application Annuaire

Le client est connecté à un annuaire par le biais d'un connecteur effectuant un appel synchrone, `syncCall`. Les signatures sont identiques et les services sont un `require` et un `provide`. L'interconnexion est donc valide comme le montre le Tableau 6 qui dresse la liste des connecteurs existant dans OCL ainsi que leurs caractéristiques. La seconde interconnexion utilise un connecteur asynchrone, car les services sont de type `notify` et `react`. On peut toutefois remarquer la présence d'une clause de transformation qui ajoute en entête du rapport transmis d'un annuaire vers l'administrateur, le nom du pays de l'annuaire. Dans cette clause, l'opérateur de concaténation entre chaînes est utilisé ainsi que la valeur d'un attribut de l'annuaire.

La validité d'une interconnexion est vérifiée par les caractéristiques de chaque connecteur qui imposent les services que l'on peut relier ainsi que les signatures compatibles de ces services. Le Tableau 6 contient les différents connecteurs qui existent dans OCL. Nous pouvons remarquer dès à présent que la répartition n'entre pas en ligne de compte dans le choix du connecteur. Un connecteur définit des propriétés de la communication. Pour chaque connecteur, il existe différents objets utilisés lors de l'exécution, un pour la communication locale au sein d'un même processus, un pour la communication locale entre des processus différents et un pour la communication entre sites. Le choix de l'un ou l'autre objet pour l'exécution s'effectue lors de la phase de déploiement de l'application en fonction de la localisation des composants impliqués dans l'interconnexion.

Nom	Appelant(s)	Appelé(s)	Propriétés	Mécanisme	Paramètres
syncCall	1 require	1 provide	Communication Synchrone Signatures compatibles ou clause do	Local: appel de fonction ou méthode IPC: utilisation de ILU Distant: utilisation d'ILU	Aucun
asyncCall	1 notify	1 react	Communication asynchrone	Local: appel de fonction dans un thread séparé	Aucun

			Signatures compatibles ou clause do	IPC : utilisation d'ILU en asynchrone Distant : utilisation d'ILU en asynchrone	
randSyncCall	1 require	1 à n provide	Idem syncCall, mais choix aléatoire d'un seul destinataire	idem syncCall	Aucun
createIn-Collection	1 require	1 provide au sein d'une collection	Création d'une instance dans une collection, puis appel synchrone du service appelé. Signatures compatibles ou clause do	idem syncCall en fonction du site de création du composant de la collection	Valeurs des attributs du composant créé.
deleteIn-Collection	1 require	1 provide au sein d'une collection	Appel du service appelé synchrone puis suppression du composant de la collection	Idem appel synchrone.	Aucun
aaa	1 notify d'un composant primitif contenant des modules "AAA"	1 react d'un composant primitif contenant des modules "AAA"	Appel asynchrone Signatures compatibles. Pas de clause do	Bus logiciel AAA prenant en charge la répartition des composants.	Aucun

Tableau 6. Connecteurs OCL

III.3.3. Schémas d'instantiation dynamique

Les caractéristiques d'instantiation statique des langages de définition de Module ou d'interconnexion sont intéressantes mais trouvent rapidement leur limite dans des applications réalistes de grande taille. Il est en effet impensable d'installer un ensemble de composants avant l'exécution si ces composants ne sont jamais utilisés. De plus, il peut être intéressant de créer des ensembles de composants ayant les mêmes fonctions mais dont le nombre varie pour des raisons de disponibilité par exemple. Darwin fournit des réponses partielles à ces soucis avec l'instantiation paresseuse d'une part et l'instantiation dynamique d'autre part. L'instantiation paresseuse est l'action de déclarer une instance en retardant l'instant de sa création au premier accès. L'instantiation dynamique est la possibilité de créer des instances n'importe quand si un composant client le demande via un service particulier de création. Nous avons vu que l'instantiation paresseuse de Darwin ne répond pas à tous les besoins de création dynamique et que l'instantiation dynamique possède des inconvénients majeurs tels que l'impossibilité par un composant client d'appeler des services des instances créées dynamiquement.

Dans OCL, les deux concepts existent : l'instantiation paresseuse identique à Darwin et l'instantiation dynamique qui diffère passablement du modèle de Darwin. Nous avons aussi introduit dans OCL le concept de "*Collections*". Ce sont des ensembles, bornés ou non, de composants ayant la même interface. La cardinalité de l'ensemble est contrôlable par l'architecte de l'application, car une collection permet d'ajouter ou de supprimer des composants en cours d'exécution. Il existe à cet effet deux connecteurs spécifiques, qui lorsqu'ils sont utilisés pour la communication, déclenchent la création ou la suppression d'une instance de composant dans la collection, selon les propriétés que l'on peut trouver dans le Tableau 6.

Dans l'exemple suivant, nous avons modifié l'application annuaire en demandant la création de l'objet client lors de sa première utilisation et en incluant les annuaires dans une collection non bornée en fonction des demandes de l'utilisateur. L'interface du client a été étendue pour prendre en compte la demande de création d'un nouvel annuaire pour un pays donné.

```
interface ClientItf {
        provide Init();
        require Lookup_Annuaire(      in string pays,in string cle,
                                      out string adresse);
        require Create_Annuaire(in string pays);
}
...
implementation AppliImpl : AppliItf
        use AnnuaireItf, AdminItf, ClientItf
{
        client = dyn instance ClientItf(); // instantiation paresseuse
        annuaires = collection [0..n] of AnnuaireItf;
        admin = instance AdminItf();
...
        client.Create_Annuaire(p) -> annuaires.Init()
                using createInCollection(p);
...
}
```

Figure 53. Instantiation dynamique et Collections dans OCL

On peut remarquer dans cet exemple que l'appel par le client du service `Create_Annuaire` provoque l'utilisation d'un connecteur particulier dont l'action est une création de composant dans une collection, puis l'appel du service désigné en partie droite de l'interconnexion pour cette instance nouvellement créée. On remarque l'utilisation d'un paramètre au niveau du connecteur qui sert à l'initialisation des attributs de l'instance créée.

III.3.4. Interconnexions complexes

Cet exemple utilise une collection non bornée de composants réalisant l'interface Annuaire. On peut immédiatement faire remarquer que rien pour l'instant ne permet l'accès à un composant particulier de la collection. Comment les distinguer, comment s'adresser à un composant plutôt qu'un autre au sein de la collection ? Cette section s'efforce de répondre à ces questions en proposant une clause supplémentaire attachée à l'opérateur d'interconnexion. Cette clause est la clause de désignation associative `where` qui permet de désigner un ou plusieurs composants en fonction de la valeur de leurs attributs. Cette clause est particulièrement adaptée à l'utilisation de composants à l'intérieur de collections. La syntaxe de la clause `where` comprend un ensemble de comparaisons entre des noms d'attributs d'instances ou de collections, des expressions ou des paramètres transitant par l'interconnexion. La partie gauche des expressions de comparaison contient toujours un attribut de composants destinataires de l'interconnexion. Il est aussi possible de combiner plusieurs expressions de comparaison avec les opérateurs logiques standard.

L'exemple suivant montre comment un client envoie une requête vers un annuaire de la collection en fonction du paramètre `pays`. De plus, nous pouvons voir qu'une interconnexion contenant en partie gauche une collection agit de la même façon que s'il s'agissait d'un composant normal : le composant au sein de la collection qui initie la communication envoie une demande vers le composant administrateur.

```
...
client.Lookup_Annuaire(pays,cle,adresse) ->
        annuaires.Lookup(cle,adresse)
        using randSyncCall()
        where {
                annuaires.Pays == pays; // comparaison entre l'attribut
                                        // pays des composants de la
                                        // collection et le paramètre pays
                                        // envoyé par le client
        };

annuaires.SendStats( statReport) -> admin.GetReport( report)
        using asyncCall()
        do {
                report = annuaires.Pays + " : "+ statReport;
        }
...
```

Figure 54. Désignation Associative dans OCL

III.3.5. Avantages et Inconvénients

La définition des architectures avec OCL offre un degré d'expression équivalent à ceux de tous les langages étudiés dans le chapitre II avec une approche parfois proche (comme avec CONIC, Darwin) ou un peu différente (comme UniCon et son système de type fort, Rapide avec ses expressions d'événements,...). Toutefois, il apporte de la souplesse et des nouveautés dans l'expression de la dynamique d'une application. Des composants identiques au demeurant, peuvent être combinés de manière riche avec les clauses de transformation, les collections et la désignation associative, offrant des possibilités nouvelles de description d'architecture tout en conservant des briques de bases - les primitifs -, réutilisables dans de multiples contextes et en gardant une description proche de ce qui existe lors de l'exécution. Par rapport à UniCon, OCL se veut plus souple en terme de création, définition des composants, en évitant l'utilisation de type de composants. Il garde malgré tout la possibilité d'effectuer un grand nombre de vérification sur l'architecture, en particulier avec les connecteurs et les règles d'interconnexion qui y sont attachées. Nous avons volontairement évité d'introduire les constructeurs de configuration de Darwin tels l'itérateur, les conditions, la récursivité sans raison autre que la complexité déjà suffisante de OCL.

Cette souplesse a un coût : celui de fournir un langage plus complexe, avec de multiples niveaux d'écriture (l'interface, l'implémentation, la glu,...) mais aussi d'obliger la création d'outils et d'un support d'exécution : il faut effectuer des vérifications et des conversions de types compliquées, posséder un formalisme de définition des connecteurs afin de valider les interconnexions tout en permettant l'utilisation de multiples connecteurs, gérer la désignation associative tant au niveau validité dans le langage mais aussi mécanisme d'exécution réalisant cette fonction essentiellement dynamique.

Nous n'avons de plus pas abordé le problème des erreurs et des exceptions qui ne manqueront pas d'exister, car d'une part elles peuvent être levées par les composants primitifs et ou par les clauses de transformations, de désignation associative et d'instantiation peuvent toujours produire des erreurs. Des études sont en cours pour définir les exceptions au niveau de l'interface et proposer une clause d'interception et de traitement des exceptions au niveau des interconnexions de composants.

II.4. Adhérence aux Ressources

OCL permet d'associer à l'architecture de l'application des informations concernant l'utilisation de l'environnement d'exécution sur lequel elle sera déployée. Ces informations permettent une meilleure utilisation du système d'exécution par rapport à des critères propres à l'application comme ceux de localisation des composants, d'utilisation de machines performantes, etc. Elles permettent une adaptation des ressources du support d'exécution aux besoins de l'application. L'avantage de l'utilisation d'un langage de configuration est d'une part de rendre indépendant du code logiciel tous les besoins concernant l'utilisation des ressources du système mais aussi d'utiliser la description de l'architecture, la structure des composants logiciels pour exprimer ces besoins.

Dans OCL, nous avons décidé de nous attacher au problème du placement des différents composants d'une application dans des processus, répartis sur un ensemble de sites. A la différence de CONIC, Darwin ou Polylith, le placement sur des sites n'est pas le seul souci d'un administrateur d'applications. La répartition de composants dans des processus est aussi importantes pour des raisons d'obligation (un exécutable ne peut souvent fonctionner que dans un processus propre) ou de sécurité (séparation des données utilisateurs dans des contextes d'exécution différents). La base de notre proposition consiste à spécifier pour chaque composant (ou groupe de composant) le site mais aussi l'utilisateur pour lequel se composant doit s'exécuter. L'association entre un site et un utilisateur permet ainsi de définir le processus qui accueille le code du composant. De plus, nous avons essayé de permettre l'expression de critères de placement plutôt qu'une désignation directe d'un nom de site ou d'utilisateur. Ceci permet ainsi de définir des ensembles de sites ou d'utilisateurs susceptibles d'accueillir des composants et ainsi d'utiliser au mieux les ressources du système distribué disponibles lors de chaque lancement de l'application.

Le second choix du langage OCL est de décrire cette répartition non pas dans les parties d'implémentation des composants mais dans une partie spéciale, dédiée à tous les problèmes d'adaptation aux ressources d'exécution : la section d'administration (**`management`**). Un composant est la glu entre l'interface, l'implémentation et la section d'administration. Il est ainsi possible de modifier les critères de placement des composants de manière indépendante de sa définition ou de son implémentation. La répartition de composants existants dans une nouvelle application est ainsi facilitée car seule cette section est à modifier.

La spécification du placement dans la section d'administration repose sur deux concepts : les attributs d'administration caractérisent la ressource que l'on veut utiliser (par exemple, un site) et les critères de placement qui sont des propriétés que la ressource doit vérifier pour accueillir le composant.

II.4.1. Attributs d'administration

Chaque composant possède deux attributs d'administration, `Node` et `User`. Ils permettent de spécifier les contraintes imposées pour le choix du site d'exécution et de l'utilisateur pour qui l'exécution du composant aura lieu. Le choix d'un site et d'un utilisateur permet de choisir un espace d'exécution du composant appelé *Contexte*. Tous les composants dont le site et l'utilisateur choisis sont identiques, sont placés dans le même contexte. Un contexte est généralement constitué d'un processus. Néanmoins, en fonction du type de logiciel intégré par les composants, il est possible que le Contexte gère plusieurs processus. C'est le cas notamment si le logiciel intégré est un exécutable indépendant. Ces attributs sont accessibles par tous les composants, leur déclaration OCL est la suivante :

```
typedef struct Tnode {
        string name; // nom IP complet de la machine.domaine.fr
        string IPAdr; // Adresse IP
        string platform; // type de machine, conforme au retour de la fonction
                                // posix uname
        string os; // type de système d'exploitation parmi posix, nt, mac, ...
        short osVersion; // numéro de version du système d'exploitation
        long CPULoad; // Charge CPU moyenne du site au moment du déploiement
        long UserLoad; // Charge Utilisateur moyenne du site au moment du
                       // déploiement
}
management attribute TNode Node;

typedef struct Tuser {
        string name; // nom d'accès de l'utilisateur
        long uid; // identifiant de l'utilisateur
        sequence <long> grpId; // liste des identifiant des groupes de
                                        //l'utilisateur
}
management attribute Tuser User ;
```

Figure 55. Attributs d'administration de OCL

A la lecture de la figure précédente, on peut remarquer que chaque attribut contient des champs qui caractérisent la ressource (le site ou l'utilisateur). Ces champs reçoivent des valeurs en provenance de capteurs implantés dans le support d'exécution qui effectuent la mesure des différents champs. Ces mesures sont ensuite utilisées pour décider du contexte de création et d'exécution du composant comme nous le verrons dans le chapitre 4. Certains champs de l'attribut `Node` contiennent des valeurs variables comme la charge moyenne en utilisateur d'un site. Elles contiennent en fait des moyennes permettant de donner une idée de l'utilisation du site et sont utilisées lors du démarrage de l'application.

Les attributs d'administration sont fournis en standard dans OCL, chaque composant pouvant décider de leur utilisation ou non. Il n'est actuellement pas prévu d'ajouter de nouveaux capteurs ou attributs dans la version de l'environnement Olan à moins de régénérer un support d'exécution possédant de nouveaux capteurs.

II.4.2. Répartition des Composants

Les critères de répartition des composants sont déclarés dans une section `management`. Des expressions sont associées à chaque champ de l'attribut d'administration choisi, et elles permettent d'effectuer la décision lors du déploiement de l'application. Ces expressions indiquent une valeur ou un ensemble de valeurs que doivent vérifier les champs des attributs dans le contexte d'exécution choisi. Ces expressions sont de la forme :

```
[Nom Composant .] Nom attribut . champ operateur expression;
avec    operateur :=  == |  != | < | > | <= | >=
        expression := (expression ) | valeur | in [ensemble]
        valeur := valeur_arithmetique | valeur_chaine
                valeur_chaine := chaine | expression_reguliere
Les expressions régulière sont des chaîne de caractères utilisant les opérateurs
* et ? similaires à ceux présents dans les shell Unix.
```

Figure 56. Syntaxe OCL des expressions d'Administration

Il est ainsi possible d'exprimer des critères sur les attributs du composant dont la section d'administration dépend ou bien sur les sous composants s'il s'agit d'un composite. Cette dernière caractéristique permet d'exprimer des conditions de placement d'un ensemble de sous composants les uns par rapport aux autres, par exemple la colocalisation dans un même processus, le placement sur des processus différents d'un même site, sur des sites différents,...

Reprenons l'exemple de l'annuaire. Nous imposons que les différents annuaires soient localisés sur un ensemble donné de sites. De plus, nous désirons que le client soit sur un site différent de celui des annuaires, car les sites des serveurs d'annuaires ne peuvent supporter la présence d'activités clientes pour des raisons de sécurité. Enfin, le composant d'administration ainsi que les annuaires appartiennent à un utilisateur privilégié.

```
management AnnuaireMgmt : AnnuaireImpl {
        Node.name == "db?.inrialpes.fr"; // un site dont le nom commence par
                                         // db, puis un caractère dans le doamine
                                         //inrialpes.fr
        Node.CPULoad <= 10; // La charge moyenne constatée lors de
                            //l'installation doit être inférieure à 10
                            //(echelle de 0 à l'infini, la charge 100 étant
                            // une utilisation standard.
        Node.UserLoad <= 10; // Nombre d'utilisateurs moyen inférieur à 10
        User.name == "admin"; // appartient à l'utilisateur privilégié
}

// Composant de surveillance des exécutions
management AdminMgmt : AdminImpl {
        User.name == "admin";// appartient à l'utilisateur privilégié
}

management AppliMgmt : AppliImpl{
        client.Node != annuaires.Node ; // Le client est sur un site différent
                                        // de la collection d'anunaires.
                                        // la collection des annuaires peut être
                                        // localisée sur plusieurs sites
                                        // en fonction des créations d'annuaires.
}

component Appli {
        interface AppliItf;
        implementation AppliImpl {
                ClientItf is Client;
                AnnuaireItf is Annuaire;
        };
        management AppliMgmt;
}
```

Figure 57. Répartition de l'application Annuaire en OCL

Il est aussi possible d'utiliser des attributs du composant pour la définition des critères de placement. Par exemple, imaginons que le placement des annuaires se fasse sur un site appelé `db_<pays>.inrialpes.fr` avec *pays* la valeur de l'attribut de l'annuaire lors de sa création. La section `management` suivante permet de le faire :

```
management AnnuaireMgmt : AnnuaireImpl {
        Node.name == "db_"+pays+".inrialpes.fr";
...
}
```

Ceci permet de paramétrer le placement du composant par rapport à des attributs qui peuvent varier d'une application à une autre. Cela permet aussi de fixer la répartition individuelle des composants dans les collections.

II.4.3. Règles de Décision

Le support d'exécution Olan utilise les critères définis dans la section d'administration pour placer chaque composant primitif de l'application. Or, nous avons vu dans la section précédente qu'il en existe pour les composants primitifs mais aussi pour les composites. De plus, la spécification du placement n'est pas obligatoire, il est possible de laisser le choix du placement au support d'exécution. Il est donc nécessaire d'avoir une politique d'évaluation de ces règles et d'éviter des conflits entre les critères exprimés à différents niveaux de la hiérarchie de composants.

Au niveau du support d'exécution, seuls les composants primitifs sont placés. Nous verrons que les structures d'exécution associées aux composites sont placées sur tous les contextes qui contiennent des sous composants. Le compilateur de OCL effectue une analyse des critères de placement présents à tous les niveaux de la hiérarchie pour définir les critères des primitifs que le support d'exécution est en mesure d'interpréter. L'analyse se fait attribut par attribut, puis champ par champ. Pour cela, les expressions exprimées aux niveaux les plus bas de la hiérarchie sont celles utilisées en premier lieu. S'il existe une expression portant sur le même champ d'un attribut à un niveau supérieur, cette dernière est ignorée. S'il en existe portant sur des champs d'attributs dont rien n'est défini, alors l'expression est ajoutée aux différents composants des niveaux inférieurs. La Figure 58 montre un exemple de l'analyse par le compilateur. La partie supérieure de la figure montre l'ensemble des expressions tel qu'il est spécifié dans les sections d'administration. Chaque composant contient des expressions associées aux champs des attributs d'administration. La partie inférieure de la figure montre les expressions effectivement associées aux composants primitifs de l'application après l'analyse par le compilateur. Nous pouvons voir que chaque primitif contient les expressions définies à son niveau mais aussi au niveau de tous les composites qui le contiennent. Les composants `CPRIM1` et `CPRIM2` illustrent la règle précédemment énoncée qui dit que si une expression associée à un champ existe déjà à des niveaux inférieurs, celles associées au même champ contenues dans les niveaux supérieurs sont ignorées (expr3 du composant C1).

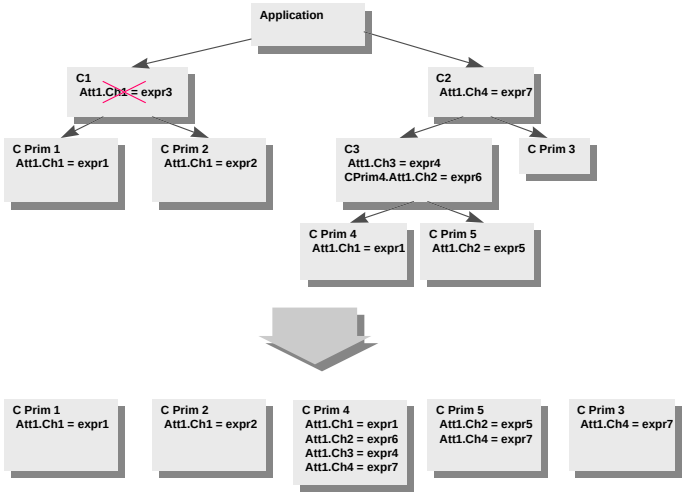

Figure 58. Evaluation des directives d'administration

Cette méthode d'analyse des critères de placement permet d'éviter tout conflit entre règles exprimées à des niveaux différents. Dans le mode de construction d'application avec OCL, il est fort possible qu'un programmeur réutilise un composant défini par un autre programmeur. Les conflits peuvent donc être fréquents. Nous avons décidé de privilégier les règles exprimées au niveau les plus bas, car nous pensons que si le concepteur de programme spécifie un critère de placement pour un primitif, alors ses motivations doivent être importantes, car pourquoi vouloir empêcher de le placer n'importe où. Nous pensons que ce choix ne doit pas être surchargé par un autre choix ultérieur provenant d'un autre concepteur. Ceci est évidemment pénalisant dans un contexte de réutilisation de composant, mais

alors rien n'empêche de réécrire de nouveaux critères de placement pour les composants qu'un programmeur récupère.

II.4.4. Avantages et Inconvénients

Notre approche semble intéressante par rapport aux approches des autres langages car elle offre plus de souplesse dans l'expression du placement, elle est indépendante de l'architecture de l'application et elle permet d'exprimer à la fois le placement sur des sites mais aussi sur des contextes d'exécution propres aux utilisateurs. Par rapport à CONIC, Darwin, Polylith, UniCon, il est possible de caractériser les sites de placement au lieu de les nommer directement de manière univoque. UniCon permet lui aussi d'exprimer le placement dans des contextes utilisateurs mais ceci est totalement intégré dans l'architecture en déclarant un composant de type `Process`. Il n'est toutefois pas possible de définir l'utilisateur pour qui le composant s'exécute ce qui ôte une caractéristique de configuration intéressante pour des applications réparties.

Il existe un certain nombre de reproche possible à notre approche. L'expression de la répartition se fait à de multiples niveaux de la hiérarchie ce qui est un plus en terme de souplesse mais cela demande au concepteur de l'application la connaissance des règles d'évaluation que nous venons de citer. De plus, le choix du site se fait lors du déploiement ce qui nous le pensons laisse plus de liberté d'expression. Toutefois, ce qui est dynamique n'est pas toujours contrôlé par le concepteur, il existe un facteur d'incertitude à propos de la faisabilité du déploiement. Cela dépend du moment du déploiement, de l'état des sites et de la politique de choix de sites contenu dans le support. Par exemple, il est possible d'obtenir un déploiement à un instant donné alors que cela ne l'est plus quelques instants plus tard.

II.5. Génération de l'image exécutable

L'environnement de configuration Olan permet de générer une image de l'exécutable de l'application comme le fait Polylith, Darwin et UniCon. Ceci est possible car toutes les informations propres au fonctionnement de l'application sont incluses dans une configuration OCL, en particulier :

Les chemins d'accès au code des composants logiciels ainsi que des informations sur leur type,

Les composants logiciels de l'application ainsi que leur schéma d'instantiation,

Les mécanismes de communication entre composants ainsi que les interconnexions, et

Le placement des composants dans des processus sur des sites.

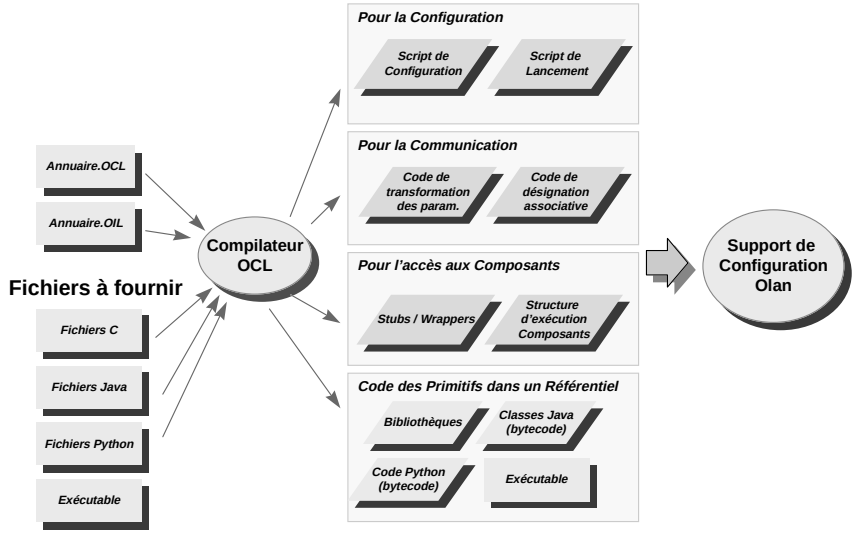

Figure 59. Processus de développement Olan

La Figure 59 montre les différents fichiers requis pour l'application. Il faut d'une part la description de toutes les interfaces des composants ainsi que la définition des modules, des implémentations et des composants. Il faut de plus tous les fichiers, bibliothèques ou exécutables intégrés dans les modules. Ces fichiers sont ensuite pris en compte par le compilateur OCL qui génère tout ce qui est nécessaire à l'exécution de l'application. Les fichiers générés sont alors compilés et enregistrés dans un référentiel. Ces fichiers peuvent être classés en trois groupes de fichiers : ceux liés à l'accès aux composants logiciels, ceux liés à la communication entre composants et ceux liés à la configuration proprement dite et au lancement de l'application. Il reste aussi une dernière catégorie de fichier qui contient le code intégré sous une forme utilisable pour l'exécution de l'application. Par exemple, un fichier C est compilé en une bibliothèque à chargement dynamique. Des classes Java ou du code Python sont précompilés dans le format pivot intermédiaire. Ces fichiers aussi sont inclus dans le référentiel.

Code pour l'accès aux Composants

Les fichiers générés par le compilateur OCL qui permettent l'accès aux composants contiennent la mise en œuvre d'une classe pour chaque composant qui est utilisée lors de l'installation du composant. Cette classe permet de créer des objets dans le support d'exécution, objets servant de base à la configuration des interconnexions avec les autres composants, mais aussi à charger les fichiers nécessaires à l'accès aux logiciels intégrés dans les modules. Cette classe est la structure d'exécution associée à chaque composant.

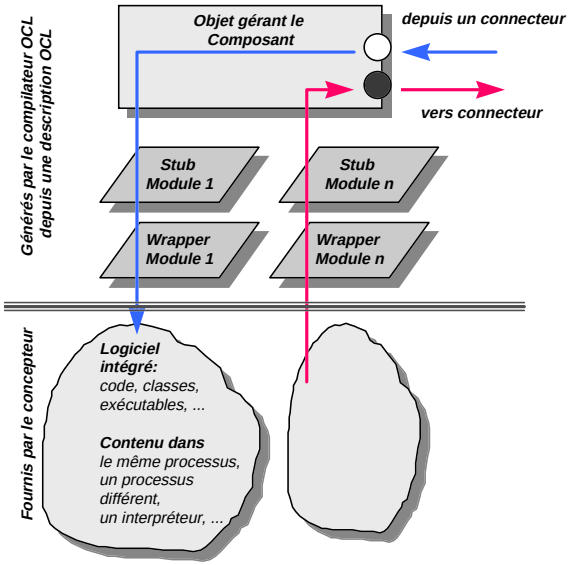

Figure 60. Accès aux logiciels intégrés

Le compilateur génère aussi pour chaque module le talon (*stub*) et l'empaquetteur (*wrapper*) pour permettre l'accès aux fonctions du logiciel et faire ressortir les appels en provenance du logiciel vers l'objet gérant le composant. Ce dernier connaît les connecteurs avec qui il est interconnecté.

Le code du wrapper n'est pas une simple redirection d'appel vers le code car en fonction du type de logiciel intégré, cela n'est pas suffisant. Par exemple, lorsque le logiciel intégré est un exécutable, ce code permet à chaque appel de lancer un processus contenant l'exécutable. Dans le cas de modules contenant des classes Java, le wrapper permet de lancer un interpréteur Java (sauf s'il est déjà lancé) et de dialoguer avec le support de configuration intégré à cet interpréteur pour créer le composant et dialoguer avec les objets Java. Ce cas de figure a été réalisé dans le cadre de l'expérimentation AAA que nous détaillerons dans le chapitre V.

Nous verrons en détail dans le chapitre IV la mise en œuvre de la classe associée au composant ainsi que le fonctionnement du stub et du wrapper selon le type de logiciel intégré.

Code pour la Communication

Le code nécessaire à la transmission des requêtes d'un composant vers d'autres n'est pas directement généré par le compilateur. Nous avons vu que des objets connecteurs servent à cela, ces objets étant fournis par le support de configuration Olan et présents physiquement dans une bibliothèque de connecteurs. Toutefois, dans certains cas d'interconnexions, il est nécessaire d'ajouter des traitements au niveau des connecteurs, en particulier lorsque l'interconnexion contient une clause de transformation de paramètres ou lorsque la désignation associative est utilisée. Dans ce cas de figure, le compilateur génère du code qui est inséré lors de la mise en place des interconnexions au sein du connecteur.

Code pour la Configuration

Le code de configuration fait partie de ce que nous présentons dans le chapitre suivant. Il s'agit de deux choses : un script permettant au support d'exécution Olan de mettre en place la configuration, i.e. d'installer les composants sur un ensemble de sites en fonction des critères d'administration spécifiés, de mettre en place les interconnexions entre composants par le biais de connecteurs et de permettre aux utilisateurs d'utiliser l'application. La seconde chose fournie par le compilateur est le script de lancement de l'application par le ou les utilisateurs désireux de créer une nouvelle application ou de joindre une session d'application en cours d'exécution.

III. Comparaison avec les modèles et langages existants

III.1. CORBA et DCOM

La comparaison entre OCL et les mécanismes CORBA et DCOM se situe essentiellement au niveau du modèle de programmation des entités logicielles, du pouvoir d'expression du langage de définition d'interface et de l'expression de la répartition. Il n'est pas pertinent de comparer la définition des architectures, car CORBA ou DCOM ne s'attachent pas les définir.

Le modèle de programmation de CORBA est basé sur le modèle de programmation à objets. Il utilise ainsi les concepts d'encapsulation, d'héritage et de polymorphisme pour définir les entités distribuables. Le modèle de programmation des composants de OCL se rapproche plus de celui de DCOM, car l'héritage de classe n'existe pas et le polymorphisme n'est pas utilisé explicitement. En revanche, OCL peut composer de nouveaux composants à partir de sous composants, ce qui est une combinaison de l'agrégation et de la délégation de DCOM. En effet, l'agrégation est totalement permise puisque des interfaces de sous composants peuvent être exhibées au niveau de l'interface d'un composite. La délégation est en partie permise, car il est possible de cacher tout ou partie des interfaces des sous composants de celle du composite. En revanche, il n'est pas possible d'ajouter des traitements supplémentaires avant ou après l'appel à une fonction puisque OCL n'est pas un langage de programmation. Tout au plus, est-il possible d'assembler des sous composants entre eux à l'intérieur d'un composite. De plus, le modèle de composant de DCOM et de OCL diffère par le fait qu'un composant OCL ne possède qu'une unique interface.

Au niveau du pouvoir d'expression du langage de définition d'interface, OCL utilise largement l'IDL de l'OMG, du point de vue de la syntaxe, du système de type et des projections normalisées vers les langages de programmation C, C++ et Java. La syntaxe de l'IDL a été étendue pour prendre en compte l'expression des services requis ainsi que des services asynchrones de type `react` ou `notify`. La différence majeure avec IDL (cette différence s'applique aussi à MIDL de DCOM) est l'impossibilité d'inclure dans les paramètres des services OCL des références à des composants ou des interfaces, alors que CORBA ou DCOM font une grande utilisation de ceci. Cette différence fondamentale s'explique par la nature même de OCL. Il n'est pas possible de passer des références d'objets ou de composants vers d'autres composants, car alors deux problèmes se posent. En premier lieu, les composants sont répartis mais la gestion des communications à distance se fait uniquement avec des connecteurs. Si une référence de composants est fournie en paramètre à un autre composant distant, comment celui-ci utilise-t-il la référence dans son code et accède au composant à distance. Le support de configuration Olan ne prend pas ce cas en charge, rien n'étant offert au programmeur pour

manipuler des composants dans son code. Le but même de l'environnement Olan est d'offrir aux programmeurs des moyens pour intégrer leur code et composer des architectures dont les éléments logiciels sont répartis par l'environnement Olan. Le but n'est pas de fournir un langage de programmation permettant d'utiliser des composants répartis. Le deuxième problème que pose le passage de composants en paramètres est les "effets de bord" que cela induit par rapport à la définition de l'architecture. Imaginons que le programmeur puisse manipuler des composants éventuellement distants dans le code des primitifs. Il y aurait alors des communications entre composants qui ne figureraient pas dans la définition OCL de l'architecture. L'architecture OCL ne serait alors pas un reflet exact de ce qui se passe lors de l'exécution.

Enfin, au niveau de l'expression de la répartition, celle-ci n'existe pas dans CORBA, car il n'est pas possible de contrôler par programme le lieu d'exécution d'objets serveurs ou clients. Dans DCOM, il est possible de le faire dans le code d'un programme. Il est possible de lancer un composant à distance ou de récupérer les références vers des composants s'exécutant à distance. Le problème de cette approche est que le schéma de répartition des composants est programmé puis compilé et inclus dans le binaire de l'application. Il n'est donc pas aisé de le modifier statiquement (phase de recompilation) et impossible de le modifier dynamiquement. OCL apporte donc des solutions en spécifiant la répartition de manière indépendante de la programmation des composants.

III.2. Polylith

Polylith est un langage d'interconnexion de module dont le rôle est de décrire les différents modules de l'application et les liaisons entre les modules. Ses principaux intérêts sont de permettre d'encapsuler des modules existants et de les rendre accessibles au travers d'une interface explicite, et de décrire explicitement les actions que l'éditeur de lien doit effectuer pour produire une application. De plus, Polylith permet de répartir ces modules en substituant les mécanismes de liaison classique par des moyens de communication entre processus ou entre sites en fonction de la localisation des modules. N dernier lieu, il est important de noter que Polylith génère une image exécutable de l'application, qui s'appuie sur un bus logiciel pour gérer le facteur de répartition à l'exécution ainsi que le facteur d'hétérogénéité des langages de programmation des modules.

En revanche, Polylith est quelque peu incomplet dès lors que l'on parle d'architecture de l'application. Il n'offre qu'une vision structurelle de l'application, i.e. les parties de logiciels qui sont nécessaire pour l'exécution et leurs dépendances fonctionnelles. Il n'y a pas de vision de la dynamique de l'application, i.e. des instances créées, de l'évolution possible des liaisons entre les modules, etc. En fin, la communication entre les modules s'effectue par le biais du bus logiciel qui utilise un mécanisme de communication particulier. Il n'est pas possible de spécifier différents types de communication au sein d'un même bus (une synchrone, une asynchrone, un protocole de diffusion,...) car rien n'existe au niveau du MIL pour caractériser cette communication. Le concept de connecteur a ici une importance fondamentale.

III.3. Darwin

OCL et Darwin sont deux langages relativement proches. En effet, la conception du langage OCL s’est initialement inspirée de Darwin, car son formalisme graphique ainsi que la simplicité des

concepts nous ont parus être une base intéressante. Les différences fondamentales entre OCL et Darwin sont :

La sémantique des composants primitifs. Dans Darwin, un primitif est un processus et il encapsule du code préparé selon un format contraignant. OCL cherche à intégrer divers types de logiciels sans contraindre l'écriture de ces logiciels. De plus, le modèle d'exécution d'un composant primitif Olan n'est pas imposé, il peut s'agir d'entités actives telles un processus ou passives telles des bibliothèques à chargement dynamique.

Darwin ne possède aucun système de type. Il n'y a pas de spécification des paramètres échangés entre composants ce qui ôte à Darwin la faculté de faire un minimum de vérification lors de l'interconnexion de composants. La description des signatures des opérations dans Darwin consiste en une chaîne de caractères. Deux services peuvent être interconnectés si leurs chaînes respectives sont identiques. Des clauses de transformations de paramètres telles celles présentes dans OCL sont donc impossible à spécifier dans Darwin.

La sémantique d'interconnexion. Dans Darwin, l'objectif de l'interconnexion est de mettre en relation les ports d'entrées et de sorties des composants. Seule la désignation de l'émetteur et du récepteur d'une communication est configurée par Darwin. L'utilisation d'un mécanisme de communication est programmée à l'intérieur du code des composants primitifs. Il n'y a donc pas d'entité similaire au connecteur de OCL qui permette de spécifier de manière indépendante du code des primitifs toute la communication entre composants : intervenants de la communication, mécanisme de communication utilisé, protocole, transformation de paramètres, etc.

Il existe aussi des différences importantes entre ces deux langages : le schéma d'instantiation dynamique, les interconnexions complexes, le pouvoir d'expression du placement des composants,... En particulier, OCL offre le concept de collections qui remplace avantageusement l'instantiation dynamique de Darwin, car elle peut être contrôle par un connecteur et non pas par un service de création particulier comme dans Darwin, elle permet de créer comme de supprimer des composants et surtout elle permet l'accès par des composants clients aux composants contenus ce que ne permet pas l'instantiation dynamique de Darwin.

Il faut toutefois noter à l'actif de Darwin, qu'il offre des opérateurs de configuration tels que l'itérateur, les conditions,... qui sont absents de OCL. Ces opérateurs permettent de définir une certaine algorithmique des interconnexions ou du placement mais nous pensons que ces opérateurs sont adaptés à des applications dont la structure d'interconnexion et de placement est régulière. Par exemple, ceci est plus adapté au placement de composants sur un réseau processeur dont la topologie peut facilement être décrite avec de tels opérateurs.

III.4. UniCon

UniCon est le langage de définition d'architecture le plus complet que nous ayons étudié. Il permet de faire remonter au niveau du concepteur de l'application des abstractions qu'il manipule régulièrement, telles que la notion de processus, de pipe, de filtre,... Un système de type strict permet d'associer chacune de ces notions aux composants. Les connecteurs sont eux aussi typés, chaque type représentant un mode de communication particulier : un RPC, un pipe,... Des règles d'interconnexions strictes autorisent ou n'autorisent pas l'utilisation d'un connecteur avec des composants. Ainsi,

UniCon effectue toutes les vérifications d'assemblage des abstractions et mécanismes utilisés pour faire fonctionner une application éventuellement répartie. L'effort du programmeur se focalise sur l'écriture de l'algorithmique des composants, le concepteur ou l'architecte effectue l'assemblage des composants.

L'approche de OCL est un peu différente car il n'existe pas de typage des composants au sens de UniCon. Le composant est une entité logicielle sur laquelle aucune contrainte comparable aux types UniCon existent. Un composant est homogène : il peut être a priori utilisé sur tout type de machine, avec tout type de connecteurs,... Toutefois, des informations sont présentes au niveau de l'interface pour indiquer le modèle d'exécution interne

La différence entre UniCon et OCL est leur philosophie. UniCon tente de fournir au concepteur d'application un cadre et des règles pour automatiser et garantir l'intégrité de fonctionnement des différents mécanismes de programmation et de communication que l'application nécessite. OCL propose un modèle d'intégration de logiciel existant et de répartition sur un environnement hétérogène. Nous pensons que le modèle de composant Olan est plus adapté aux applications réparties, car il permet de ne pas lier directement les composants à un type d'architecture donné, un site précis, ... contrairement à ce que fait UniCon. Le placement des composants est ainsi configurable, modifiable d'une application à l'autre, d'une exécution à l'autre. UniCon a une approche stricte à ce sujet, une fois le choix de répartition et de placement effectué, rien ne peut changer.

La seconde différence entre UniCon et OCL vient de la description de la dynamique de l'application. UniCon considère que le schéma d'instantiation des composants et des interconnexions est essentiellement statique, défini une fois pour toute par l'architecture. Les possibilités de décrire la dynamique de création et suppression de composant n'est pas possible. Il en est de même pour la configuration des intervenants d'une interconnexion en fonction de choix dynamique.

III.5. Rapide

La comparaison entre OCL et Rapide ne peut se situer que du point de vue de l'expression des interconnexions et de la dynamique de l'application. Les langages ne suivent pas les mêmes buts : le second vise à spécifier des architectures en vue de simuler leur comportement, le premier les décrit pour générer une image exécutable. Les soucis d'intégration de code, de placement ne sont donc pas de mise dans Rapide. L'effort de Rapide se situe autour de la description des architectures dont la dynamique des communications comme des créations et suppressions des composants est très forte. Rapide apporte un certain nombre de solutions originales à la description de systèmes complexes qui évoluent fortement au cours de l'exécution. Chaque interconnexion est ainsi décrite comme une règle qui est évaluée et déclenchée dès que les conditions d'exécution s'y prêtent. Ces règles permettent entre autres de créer des composants, de désigner dynamiquement les intervenants d'une communication,... en s'appuyant sur un formalisme d'écriture et d'évaluation des règles.

Nous pensons que les idées de Rapide, couplées aux objectifs complémentaires de OCL peuvent mener à un environnement complet de configuration dont la simulation serait partie intégrante du processus de déploiement.

Chapitre 4
Support de Configuration Olan

Les langages de définition d'architecture comme les langages de configuration permettent d'éviter tout un travail de réalisation au niveau du code des composants logiciels et offrent des abstractions de haut niveau qui permettent l'utilisation de mécanismes d'exécution divers sur lesquels repose l'application. Dans OCL, ces abstractions sont les connecteurs, l'instantiation dynamique, la désignation associative, le placement dans des processus,... Ceci implique l'existence d'un compilateur qui génère du code additionnel aux composants logiciels pour arriver à faire fonctionner l'application. Le compilateur Olan génère le logiciel de configuration des composants et d'accès à du logiciel hétérogène (cf. section 0 du chapitre précédent). Dans UniCon, le compilateur effectue aussi la génération du code final de l'application en incluant la programmation des connecteurs, des interconnexions, etc. Toutefois, le cas des applications réparties demande la présence d'un support système qui permet le déploiement des composants et la gestion des communications entre composants. Polylith utilise un environnement de génération d'application, PolyGen[Cal90] ainsi que des bus abstraits pour la communication. CONIC et Darwin reposent sur un support d'exécution. OCL possède lui aussi un support appelé support de configuration qui offre des fonctions particulières que nous allons détailler dans la suite de ce chapitre. Nous présenterons tout d'abord l'architecture générale du support de configuration, puis nous détaillerons la mise en œuvre des composants et de la machine à configuration qui effectue la mise en place de l'application.

I. Architecture Générale

I.1. Principales Fonctions

Le support de configuration Olan a pour charge de permettre la configuration et l'exécution d'applications décrites par OCL. Ses différentes fonctions sont :

- le déploiement des composants logiciels en fonction des indications de placement contenues dans la description OCL,
- l'installation des composants logiciels à l'endroit choisi et le positionnement des attributs,
- la mise en place des interconnexions entre composants, et
- la gestion des exécutions.

Le déploiement

Le déploiement est l'action de choisir les sites et les processus qui hébergeront les différents composants de l'application. Le support de configuration interprète le script de configuration fourni par le compilateur OCL. Ce script contient en premier lieu toutes les indications de création des composants auxquelles sont associés les critères de placement des composants. Le support est alors en charge de trouver un site qui soit capable de créer le composant, i.e. qui ait accès au logiciel intégré dans le composant et qui réponde aux critères de placement.

L'installation

L'installation du composant consiste à créer l'objet composant que le compilateur produit et ‡ charger le logiciel intégré ainsi que les talons et les empaquetteurs associés. De plus, le support positionne les valeurs des attributs des composants.

Les Interconnexions

La dernière phase de mise en place de l'application consiste à créer les interconnexions de composants. Il s'agit ici de mettre en place les objets connecteurs et de connecter les services d'entrées et de sorties des composants aux connecteurs adéquats. La phase de mise en place d'un connecteur contient le choix du mécanisme de communication en fonction du placement des composants, la création des objets permettant d'utiliser le mécanisme de communication choisi et l'insertion de l'éventuel code de transformation des paramètres et de désignation associative.

L'exécution

Le support a pour ultime tâche la gestion du lancement de l'application, de la connexion et la déconnexion des utilisateurs habilités à joindre une application en cours d'exécution.

I.2. Découpage Fonctionnel

Pour garantir ces différentes fonctions, le support Olan est découpé en différentes parties comme le montre la Figure 61.

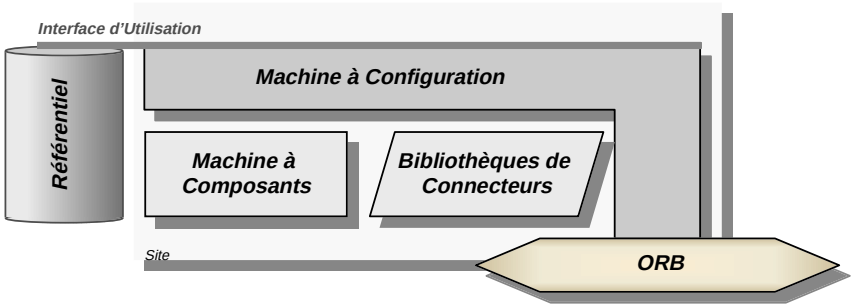

Figure 61. Architecture du Support Olan

la Machine à Configuration

Le rôle de la machine à configuration est de fournir une interface pour la configuration d'applications réparties. Elle offre des primitives correspondant aux fonctions principales de configuration précédemment présentées : déploiement, installation, interconnexions et gestion des exécutions. Ces primitives sont l'unique entrée pour l'utilisation du support de configuration Olan. Elles sont utilisées principalement par le script de configuration puis par le script de lancement générés par le compilateur OCL.

La machine à configuration est présente sur tous les sites autorisés à héberger des exécutions d'applications. Elle a des besoins propres de communication entre chacun de ces sites pour effectuer d'une part le choix du placement, i.e. le déploiement, et d'autre part l'installation à distance des composants. En effet, si le site de placement d'un composant n'est pas le même que le site de lancement de l'application, il faut d'une manière ou d'une autre pouvoir créer ce composant à distance sur la machine choisie. C'est la raison d'être du courtier d'objet (ORB) dans l'architecture. La communication entre machines de configuration ne sert que lors de la phase d'installation de l'application. Lors de l'exécution, les composants de l'application communiquent par le biais de mécanismes cachés dans les connecteurs en utilisant si nécessaire d'autres mécanismes ou protocoles que ceux fournis par le courtier d'objet.

La machine à configuration pilote ensuite la machine à composants, car elle n'effectue pas elle-même la création de composants ou le chargement des modules logiciels. Elle pilote aussi l'utilisation de la bibliothèque de connecteurs pour la mise en place les interconnexions.

La Machine à Composants

La machine à composants s'occupe de créer les composants logiciels ainsi que la chaîne d'accès aux modules logiciels intégrés. Au travers de primitives, elle permet à la machine à configuration de créer les structures générées par le compilateur sur le site choisi par cette machine. Parmi ces structures, la machine à composant gère, outre celles dérivant les composants, celles pour la gestion du comportement dynamique de l'application relatif aux composants comme les collections, le mécanisme de désignation associative, l'instantiation dynamique dans les composites.

Il existe un exemplaire de cette machine sur chaque site, mais il est important de noter que les communications entre machines à composants ne se font que par l'intermédiaire des primitives internes de la machine à configuration. Il n'y a donc aucune adhérence particulière de cette machine avec un mécanisme de communication, elle est indépendante de la distribution.

Bibliothèques de Connecteurs

La bibliothèque de connecteurs contient des classes d'objets permettant d'acheminer des communications entre composants selon les règles d'interconnexions du langage OCL. Ces objets s'adaptent au placement des composants pour d'une part choisir le mécanisme de communication adéquat et d'autre part se répartir sur les différents contextes d'exécution des composants.

De plus, ces objets sont susceptibles de contenir du code, dynamiquement chargé lors de leur création, qui gère la transformation éventuelle de paramètres ou l'utilisation du mécanisme de désignation associative pour acheminer ou non les communications.

Le Référentiel

La machine à configuration et la machine à composants utilisent un référentiel commun à tous les sites, qui contient tout ce qui est propre aux applications ainsi que la bibliothèque des connecteurs. Il permet de stocker le code produit par le compilateur OCL et c'est grâce à cet espace commun que les différentes machines retrouvent les données et structures qu'elles doivent mettre en place.

I.3. Choix d'Implémentation

A l'origine du développement du support de configuration, nous avons imposé dans le cahier des charges la portabilité maximum de l'environnement sur différentes plates-formes d'exécution, en l'occurrence sur plusieurs Unix (AIX v4, Solaris 2.5) et sur Windows NT 4.0. Ce critère nous semblait plus important que des critères de performances, car ce support devait démontrer l'intérêt de la configuration d'applications réparties sur supports hétérogènes plutôt que son coût d'exploitation.

C'est pour ces raisons que nous avons adoptées comme langage de programmation du support de configuration Python[Ros95]. C'est un langage semi-interprété, indépendant de la plate-forme d'exécution, produisant du byte-code et dont la caractéristique principale est la dynamicité. C'est un langage modulaire, à typage dynamique, orienté objet, réflexif, intégrant en standard toutes sortes de traitements sur des tuples, listes ou dictionnaires. Ce choix s'est avéré justifié, car le portage est immédiat et le temps de développement est très court par rapport à des langages compilés.

Pour le support de communication, nous avions la contrainte de faire communiquer des machines de configuration de la manière la plus simple possible sur des systèmes d'exploitation différents, afin de minimiser l'effort de programmation de la communication entre sites. Pour cela, nous avons choisi d'utiliser le courtier d'objet ILU (Inter Language Unification)[Jan96], un ORB interopérable avec CORBA, fonctionnant sur de multiples plates-formes, intégrant des objets mis en œuvre au travers de multiples langages de programmation (C, C++, Lisp, Modula-3, Python, Java) et "last but not least", domaine public. Les structures d'exécution de la machine d'exécution ont été programmées avec des objets en Python.

Dans la suite du chapitre, nous allons présenter le modèle d'exécution et de configuration de composants interconnectés par des connecteurs, pour ensuite présenter l'architecture, la mise en œuvre et le fonctionnement de la machine à configuration.

II. Mise en Œuvre des Composants

II.1. Principes de Fonctionnement

Nous avons vu au chapitre précédent les classes et modules générés par le compilateur pour intégrer du code existant. En particulier, nous avons vu que le compilateur produit une classe associée à chaque composant. Cette classe est instanciée lors du déploiement et possède deux fonctions : permettre les communications avec le code intégré ainsi que les communications avec d'autres composants via des connecteurs.

A cet effet, chaque composant possède deux tables, `inCall` et `outCall`, indexées par le nom des services définis dans l'interface OCL du composant. La table `inCall` est indexée par les noms des services fournis (`provide` et `react`), la table `outCall` par le nom des services requis (`require` et `notify`). Chaque élément de cette table contient la référence de l'objet ou module logiciel à qui l'appel doit être transmis. Plus précisément, la table `inCall` contient la référence de la fonction à appeler dans le talon, alors que la table `outCall` contient la référence vers la partie émettrice d'un connecteur. Enfin, tout appel à un composant est intercepté, et en fonction du service demandé, l'appel est retransmis à l'objet ou fonction dont la référence est contenue dans une de ces deux tables.

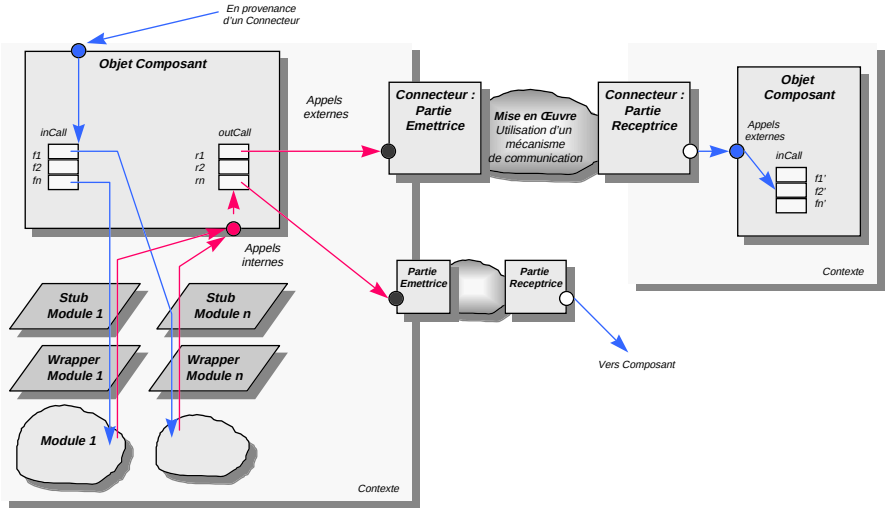

Figure 62. Communication entre Composants

L'intérêt de ces indirections est de permettre la modification dynamique des associations composant/code intégré et composant/connecteurs. De plus, la communication avec un connecteur est, du point de vue du composant, similaire à une communication locale. L'aspect répartition est pris en compte par la mise en œuvre du connecteur uniquement. Etudions maintenant plus précisément la chaîne d'appel entre le composant et les modules intégrés (et vice versa), puis la communication entre composants.

II.1.1. Module Logiciel vers le Composant

Chaque module logiciel intégré communique avec des entités logicielles contenues dans d'autres composants via l'objet composant qui lui est associé. La communication entre le composant et le code passe par deux modules. Le talon (*stub*) emballe et déballe les paramètres propres à chaque entité dans un format homogène et linéarisable. L'empaquetteur (*wrapper*) transmet l'appel en provenance du talon (resp. en provenance du logiciel) vers l'entité (resp. vers le talon). Le degré de complexité de cette dernière transmission dépend essentiellement de la nature du logiciel que l'on désire atteindre car nous verrons que dans le cas du langage Java, les objets s'exécutent au sein d'un interpréteur Java dans

un processus différent de celui qui contient les objets composants. Nous allons donc étudier le cas de logiciels écrits en Python, C (ou C++) et en Java.

Module Logiciel Python

C'est le cas le plus simple. L'environnement d'exécution est lui-même écrit en Python, donc il n'y a aucune difficulté de communication entre le wrapper et le code. Le wrapper (tout comme le talon) possède une fonction pour chaque service de l'interface du composant. Les fonctions associées aux services fournis effectuent directement l'appel aux fonctions correspondantes dans le code (fonctions qui doivent évidemment exister). Dans le cas des services requis, le programmeur doit inclure dans le code Python une fonction sans code ni paramètre, de même nom que le service requis (le lecteur pourra se reporter à l'exemple présent sur la Figure 63). Le wrapper contient quant à lui du code Python qu'il insère dynamiquement à la place de la fonction ci-dessus définie. Ce code effectue l'appel au talon du composant adéquat. Ceci est rendu possible, car Python permet de modifier dynamiquement le code des fonctions ou classes. Le travail d'intégration de code Python est donc minime pour le programmeur, le compilateur OCL faisant la presque totalité du travail. De plus, un module Python ne contient aucune référence aux structures générées par le compilateur, ce qui permet de réutiliser ce module dans tout autre contexte que celui de l'environnement Olan.

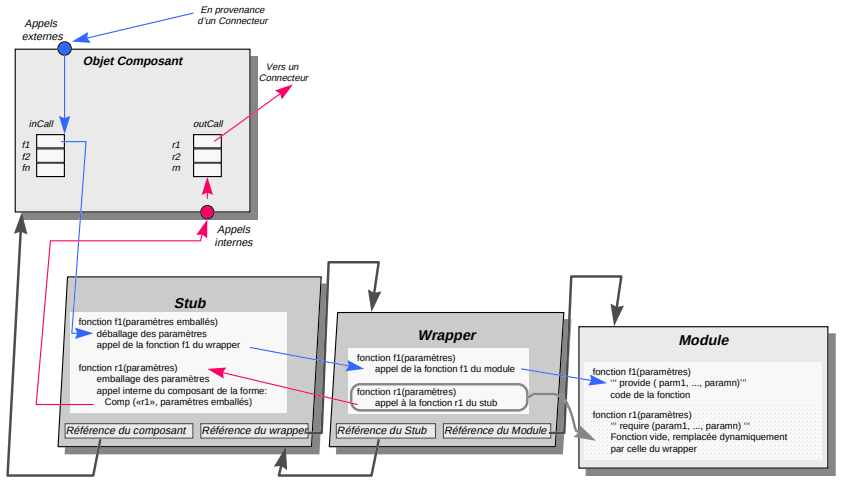

Figure 63. Intégration automatique de code Python

Module Logiciel C

Ce cas est peu différent du cas Python, car l'interpréteur Python est lui-même écrit en C. Il existe une méthode pour faire communiquer des modules C avec un interpréteur Python à l'unique condition que les modules C soient contenus dans des bibliothèques à chargement dynamique[Ros95b]. Le programmeur doit fournir dans le module C une fonction par service fournis et requis ainsi qu'une variable contenant les références pour appeler le wrapper. Celles qui correspondent à des services fournis contiennent le code de ce service, les autres doivent effectuer l'appel au wrapper selon la méthode imposée par la communication entre du code C et un interpréteur Python[Ros95b]. Ce travail

doit être en grande partie effectué par le programmeur, sans aide de la part du compilateur OCL comparable à celle qu'il fournit pour du code écrit en Python ; Il faut donc être conscient que le programmeur C doit suivre une certaine discipline pour permettre l'intégration de son code.

Module Logiciel Java

Dans ce cas, l'accès à des objets Java est une opération plus délicate car le composant et son talon s'exécutent au sein d'un interpréteur Python, alors que les objets Java sont au sein d'un interpréteur Java, les deux interpréteurs étant dans deux processus différents.

La mécanique du wrapper est donc plus complexe car il doit fournir un mécanisme de communication entre processus. La Figure 64 schématise ce processus de communication.

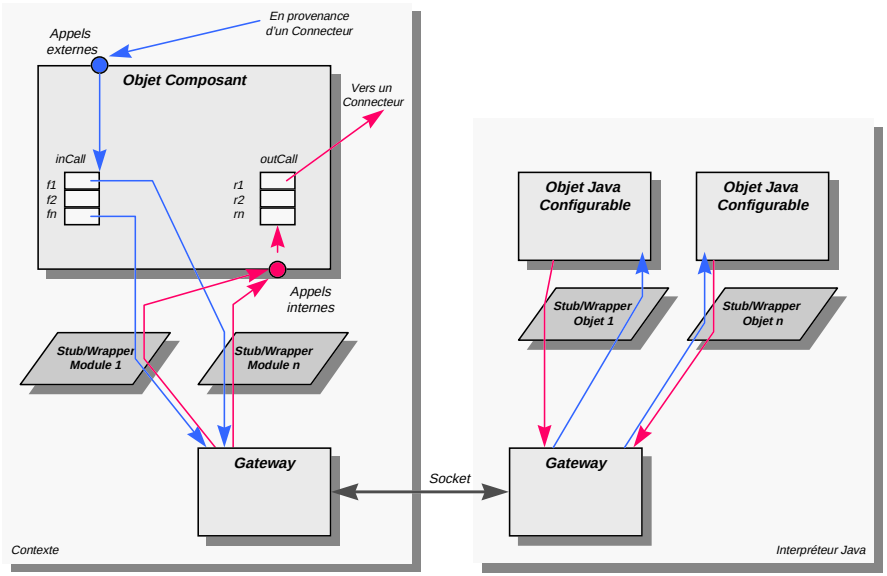

Figure 64. Intégration de code Java

La communication entre le monde Olan et l'interpréteur Java se fait par socket. Ce canal de communication est accessible au travers d'un objet `Gateway` présent à la fois dans le monde Olan et dans l'interpréteur Java. Cet objet retransmet les appels et les oriente vers les bons objets dans le monde Java ou les bons talons dans le monde Python. La décomposition en talons et empaquetteurs n'est plus aussi claire que dans les cas de figure précédents. En effet, l'emballage et déballage des paramètres sont faits dans le monde Java car on transmet entre processus des informations linéarisées. En revanche, l'empaquetteur agit dans les deux mondes car c'est lui qui connaît l'objet de transmission `Gateway` et l'utilise de manière transparente au programmeur d'objets Java.

II.1.2. Composant vers un Composant

Une fois les composants et la chaîne d'accès au code intégré créés, la communication entre composants transite par des objets connecteurs (cf. Figure 62). Chaque interconnexion donne lieu à la création d'un objet connecteur, même si plusieurs connecteurs de même type coexistent au sein d'un même processus. Ces objets sont scindés en deux :

- une interface de communication avec les composants, représentée par les objets "partie émettrice" et "partie réceptrice",
- la mise en œuvre de la communication en fonction du type de connecteur spécifié dans le programme OCL et du placement des composants.

Ce principe d'interconnexion entre composants ajoute des indirections supplémentaires dans la communication. Il existe toutefois des cas où il est possible de supprimer certaines indirections. En effet, pour un connecteur appel de fonction synchrone, il est possible de ne pas faire appel à un objet connecteur si les composants sont localisés dans un même processus et si aucune clause de transformation ou de désignation associative n'a été spécifiée dans le programme OCL. Nous ne détaillerons pas plus la mise en œuvre des connecteurs, car ceci déborde du cadre de ce travail. Le lecteur pourra se reporter à [Mar97].

II.1.3. Autres Entités de OCL

Dans un programme OCL, il existe d'autres entités manipulées comme par exemple les composites, les collections, etc. Chacune de ces entités à une existence à l'exécution par une structure ou un objet qui la représente. La raison de l'existence de ces objets est de reproduire à l'intérieur du support d'exécution la même structure hiérarchique des composants, ceci afin de pouvoir utiliser par la suite des outils de supervision de l'exécution de l'application dont la base est la même représentation textuelle ou visuelle que celle de la conception. De plus, en fonction de l'entité, il est parfois impératif d'avoir un objet lors de l'exécution. Dans le cas des collections, cet objet permet d'effectuer les créations et suppressions de composants.

II.2. Principes de Configuration

Les actions de configuration dans le support Olan correspondent essentiellement à la création des structures décrites précédemment et au remplissage des différentes tables des composants. Plus précisément, le support Olan effectue les opérations suivantes :

- Le placement et l'installation des composants dans des processus.
- La mise en place de la chaîne d'accès au code à travers des talons et empaquetteurs. Les tables internes des objets composants sont remplies avec les références d'accès à cette chaîne. Dans certains cas, cette action peut être complexe (par exemple avec des objets Java), car des processus externes peuvent être lancés,...
- La mise en place des objets connecteurs et le choix des mécanismes de communication en fonction du placement des composants. Les tables externes des composants sont alors remplies avec les références vers les interfaces d'accès aux connecteurs.

III. La Machine à Configuration

III.1. Interface de la Machine à Configuration

L'utilisation de la machine à configuration s'effectue par le biais d'une série de primitives appelées par les scripts de configuration et les scripts de lancement que le compilateur OCL génère. La liste des primitives offertes est la suivante :

```
CreatePrimitiveComponent( string oclName,
             uInfoIncomplet user,
             nInfoIncomplet node)
```

Permet de créer le composant primitif de nom *oclName* sur un site répondant aux critères de placement contenus dans *user* et *node*.

Cette primitive est la base même de la configuration d'applications. Elle permet de demander la création effective d'un composant primitif en tenant compte des critères de placement, en terme de sites comme d'utilisateurs. Les informations liées au choix du composant, comme les modules logiciels qu'il intègre, sont contenues dans le nom OCL du composant (cf. ci-dessous la section sur la désignation). Cette primitive ne fonctionne correctement qu'à l'unique condition que le nom OCL fasse référence à des fichiers présents dans le référentiel.

```
CreateCompositeComponent(string oclName,
             stringList internalCompNameList,
             uInfoIncomplet user,
             nInfoIncomplet node);
```

Permet de créer le composant composite de nom *oclName*, contenant les sous composants dont les noms sont contenus dans *internalCompNameList*. Le composant composite est une structure dupliquée dans chacun des contextes contenant un sous composant.

Cette primitive permet de créer un composant composite qui contient un ensemble de sous composants déjà présents dans le support Olan. L'intérêt d'un composite se justifie par la volonté d'avoir à l'exécution la même structure de composant que celle de la description OCL de l'application, et par le fait d'avoir des connecteurs interconnectés, éventuellement différents de ceux présents à l'intérieur du composite.


```
SetAttribute(string oclName,
             string attributeName,
             oilParam value);
```

Permet de positionner l'attribut de nom *attrName* du composant de nom *oclName*. La valeur de l'attribut est fournie dans le paramètre value ; Il est transmis sous forme de chaîne de caractères correspondant à la linéarisation du format interne Olan de transmission des paramètres.

Les attributs sont une caractéristique importante d'un langage de configuration car ils permettent de personnaliser l'utilisation d'un composant. On ne peut donc se passer de leur présence à l'exécution car ils servent entre autre chose pour la désignation associative. Un attribut correspond à une variable, déclarée au niveau de l'objet composant mais aussi à l'intérieur des modules intégrés dans un primitif.

```
CreateDynamicComponent(string oclName,
             stringList configurationCode,
             uInfoIncomplet user,
             nInfoIncomplet node);
```

Permet de pré-créer le composant de nom *oclName*, en vue de l'instantiation dynamique de OCL. Le second paramètre *configurationCode* contient une liste de chaînes. Chaque chaîne contient une primitive de configuration des sous composants devant être créés lors de la création effective de ce composant.

Cette primitive a le même effet que la création d'un primitif ou d'un composite. Toutefois, cet effet est retardé au moment du premier accès à un des services de ce composant. Si ce composant est un composite, tous les composants internes sont alors créés à cet instant. La création du composant et de ses sous composants se fait par le biais du code de configuration stocké lors de l'utilisation de cette primitive.

```
CreateCollection(string oclName,
             uInfoIncomplet user,
             nInfoIncomplet node);
```

Permet de créer l'objet Collection de nom *oclName* sur le site indiqué. Cet objet permet ensuite, au travers de connecteurs particuliers, de créer des instances de composants en son sein.

L'intérêt de cette primitive est évident : elle met en place le concept de collection dans une application. Une collection est similaire à un composite avec la contrainte de ne posséder que des composants de même type. De plus, elle effectue la création comme la suppression des composants internes.

```
Bind ( string oclName,
             coupleList src,
             coupleList dest);
```

Permet de mettre en place une interconnexion en utilisant le connecteur de nom *oclName*. Le paramètre *src* contient la liste des composants émetteurs de la communication et le nom du service émetteur. Le paramètre *dest* contient quant à lui la liste des couples (composant, nom du service destinataire) de la communication.

Cette primitive est la seconde primitive essentielle du support Olan. Elle permet de mettre en place une communication entre des services de composants. L'étude de la signature de la primitive montre qu'il est prévu de spécifier plusieurs services à la source de l'interconnexion, i.e. d'avoir une interconnexion de type N composants vers M. Les connecteurs existants dans le support Olan ne le permettent pas actuellement, mais ceci sera possible dans l'avenir.

```
Map ( string oclName,
             coupleList src,
             coupleList dest);
```

Permet de mettre en place une liaison entre un composite et des sous composants en utilisant le connecteur de nom *oclName*. Le paramètre *src* contient la liste des couples composant émetteur de la communication et nom du service émetteur. Le paramètre *dest* contient quant à lui la liste des couples composants/services destinataires de la communication.

La présence du `Map` sert à distinguer la projection entre les services d'un composite et les services des sous composants qui les réalisent par rapport aux interconnexions mises en place par un `Bind`.

```
ExecuteComponentService(string nodeOfOrigin,
             long userIdOfOrigin,
             string componentOclName,
             string portName,
             oilParam params);
```

Permet l'exécution du service de nom *portName* du composant *componentOclName* en fournissant les paramètres (si nécessaire) de lancement *params* dans le format Olan linéarisé. Il faut indiquer dans les deux paramètres initiaux, l'identifiant de l'utilisateur qui demande l'exécution du service et le site de lancement de l'application.

Cette dernière primitive est la troisième d'importance car elle permet de lancer l'exécution de l'application. Le lancement est effectué par un utilisateur depuis n'importe quel site par le biais du script de lancement. Cette primitive de la machine à configuration demande de fournir en paramètre l'identifiant de l'utilisateur ainsi que le site de lancement car il n'est pas certain que l'objet de la machine à configuration qui initie l'exécution soit forcément sur ce site.

Schéma de Désignation

Nous avons vu lors de la présentation des primitives de la machine à configuration, que les entités manipulées dans le langage OCL sont nommées. En effet, deux types de désignation existent : la désignation OCL et la désignation Olan. Le premier type correspond aux entités d'un programme OCL que le compilateur manipule et génère les ordres de création, le second servant à l'intérieur du support pour adresser les différents objets créés. Toute entité manipulée par le compilateur possède un nom OCL unique. Ce nom, attribué par le compilateur, est de la forme suivante :

```
<nom de la classe>:<nom du module>:<nom de l'instance>
```

Le *nom de l'instance* est propre à chaque entité de OCL. Un composant de type C aura pour nom C_2 s'il s'agit du second composant de ce type à être utilisé comme instance. Le *nom du module* contient le fichier du référentiel dans lequel se trouve le code associé à l'entité. Le *nom de la classe* contient le nom de la classe de l'objet composant, connecteur,... devant être créé lors de l'exécution. Cette classe est contenue dans le module adéquat du référentiel.

Chaque entité existante dans une description OCL possède un nom OCL unique. Ce dernier contient en particulier les informations pour récupérer le code présent dans le référentiel pour l'entité. Par exemple, le nom `"comp:mod:inst1"` associé à un composant primitif de nom *comp*, désigne une instance *inst1*, dont le code pour l'exécution se trouve dans le module *mod* du référentiel.

Au niveau du support, le schéma de désignation a pour base le nom OCL mais il est possible qu'à une entité OCL unique correspondent plusieurs objets dans le support Olan. Par exemple, chaque interconnexion OCL donne lieu à l'utilisation d'un connecteur qui a un nom OCL. Lors de l'installation du connecteur, il existe au moins deux objets qui le représentent : la partie émettrice et la partie réceptrice. Le nom Olan sert donc à différencier des objets ayant le même nom OCL. Pour cela, le nom Olan est de la forme :

```
<nom du contexte> | <nom OCL>
```

Le *nom du contexte* défini de manière unique les espaces d'exécution des composants et des connecteurs. Chaque entité d'une description OCL n'existe qu'à un exemplaire dans un contexte donnée. Le nom Olan permet donc d'identifier de manière unique chaque entité existante lors de l'exécution.

Il contient, outre le nom OCL, le nom du contexte d'exécution dans lequel cet objet se trouve. Il est ainsi possible de connaître la localisation de n'importe quel objet grâce à son nom. Ce schéma de désignation est possible, car les entités Olan ne migrent pas et il n'existe pas plusieurs objets correspondant à une même entité OCL dans un même contexte d'exécution.

Informations de Placement

Les critères de placement spécifiés dans le langage OCL sont transmis à la machine à configuration par le biais des deux structures `nodeInfoIncomplet` pour le choix du site et `userInfoIncomplet` pour le choix de l'utilisateur. Ces structures ont pour définition :

```
typedef struct nodeInfoIncomplet {
    string name;
    string IPAdr;
    string platform;
    string osType;
    string osVersion;
    string CPULoad;
    string UserLoad;
}

typedef struct userInfoIncomplet {
    string name;
    string uid;
    string grpId;
}
```

Chacun des champs de ces structures contient des chaînes de caractères. Ces chaînes sont des expressions en langages Python, évaluées lors du choix du placement des composants, qui permettent au support Olan de décider si un site répond aux critères de placement. Chacune de ces expressions retourne une valeur booléenne. Ces expressions sont automatiquement générées par le compilateur OCL.

Prenons le cas de l'application Annuaire que nous avons abondamment utilisée lors des deux chapitres précédents. Dans la Figure 57, nous avons décrit certaines clauses d'administration. Ces clauses permettent au compilateur de générer les paramètres de type `nodeInfoIncomplet` et `userInfoIncomplet`, pour chacun des composants primitifs en fonction des règles indiquées dans la section *II.4.3. Règles de* Décision, p.78.

```
 Paramètre node:
 Structure exprimée avec la syntaxe Python. Chaque champ contient une chaîne de
caractères, qui est une expression Python évaluable. En particulier, l'expression
du champ name utilise l'opérateur Python de "slicing" de chaînes de caractères. La
première partie retourne les deux premiers caractères, la seconde du caractères 3
au dernier.
  {     'name' : "(node.name [1:] == "db") and (node.name[3:-1] ==
".inrialpes.fr")",
        'IPAdr' : "true", ## toujours vrai, tous les sites sont valables
        'platform' : "true",
        'osType' : "true",
        'osVersion' : "true",
        'CPULoad' : "node.CPULoad <= 10",
        'UserLoad' : "node.UserLoad <= 10"
        }
 Paramètre user:
 {      'name' : "user.name == 'admin'",
        'uid' : "true",
        'grpId": "true"
        }
```

Figure 65. Exemple de paramètres de placement

III.2. Déploiement

Le déploiement est rendu possible par deux serveurs que met en œuvre la machine à configuration : les *Serveurs Olan* et les *Configurateurs de Session*. Les *Serveurs Olan* sont des objets communs à toutes applications. Il en existe un par site et ils ont pour charge de gérer l'accès aux informations concernant le site, la machine,... ainsi que la liste des applications en train de fonctionner sur un ensemble de site. Chaque application en cours d'exécution est représentée au niveau de la machine à

configuration par un *Configurateur de Session*. Une application en cours d'exécution est appelée une *Session*, permettant ainsi d'avoir plusieurs *Sessions* indépendantes d'une même spécification OCL d'une application. C'est ce Configurateur de Session qui implémente le code associé aux primitives de la machine à configuration : il représente l'application et interprète les ordres de configuration qui y sont associés.

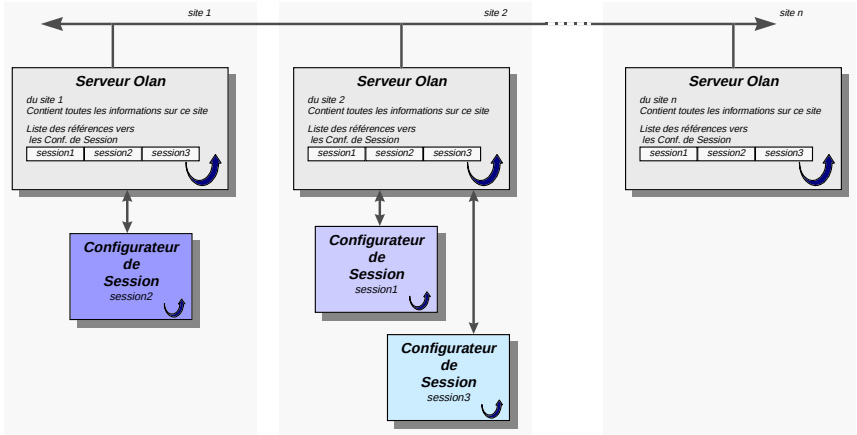

Figure 66. Serveurs Olan et Configurateur de Session

Chaque Serveur Olan peut communiquer avec les autres serveurs. Ceci est utilisé notamment pour la connexion d'un nouveau Serveur Olan ou la déconnexion d'un serveur, permettant ainsi d'augmenter ou de diminuer dynamiquement le nombre de sites susceptibles d'accueillir des applications. Les applications, chacune représentée par un Configurateur de Session, ne dialoguent pas entre elles. Le Configurateur de Session dialogue uniquement avec les serveurs Olan lorsque notamment il les interroge pour le choix d'un site de placement pour un composant.

III.2.1. Choix du Site

Nous avons vu que les primitives de la machine à configuration liées à la création de composants demandent en paramètres les critères de placement produits par le compilateur OCL. Ces critères sont sous forme d'expression évaluable dynamiquement par chacun des Serveurs Olan afin de retourner l'adéquation du site qu'il gère avec ces critères.

Le serveur Olan d'un site connaît les valeurs de chacun des champs des attributs d'administration, en l'occurrence `Node` et `User`. Il est ainsi capable d'évaluer les expressions pour chacun des champs des attributs d'administration `Node` et `User`. L'évaluation est faite par tous les serveurs Olan. Un poids est attribué pour chaque site par son serveur Olan. Un poids de 0 élimine le site. Le site ayant le poids le plus élevé sera choisi afin d'héberger le composant en question.

Les valeurs des poids pour chacun des champs sont paramétrables par l'administrateur du système pour chaque site ou pour l'ensemble, car ils permettent de définir la politique de disponibilité de chacun des sites. Par exemple, la Figure 67 contient la définition de la politique standard d'utilisation

d'un site. L'administrateur des sites Olan est capable de remplacer cette politique par une autre par le biais d'une interface graphique liée aux serveurs Olan.

```
 Choix du Site, Attribut Node
        Si <expression associée au champs name> alors poids += 20

        Si <expression associée au champs IPAdr> alors poids +=20

        Si <expression associée au champs platform> alors poids +=25
        sinon site inéligible, poids = 0

        Si <expression associée au champs os> alors poids +=30
        sinon site inéligible, poids = 0

        Si <expression associée au champs osVersion> alors poids +=25
        sinon site inéligible, poids = 0

        Si <expression associée au champs CPULoad> alors poids +=10

        Si <expression associée au champs UserLoad> alors poids +=10

 Choix de l'Utilisateur, Attribut User
        Si  non <expressions associées aux champs name, uid ou grpId> alors site
inéligible, poids = 0
```

Figure 67. Politique Standard de Disponibilité d'un Site

Le Configurateur de Session, qui est le représentant de l'application, interprète les ordres de configuration, demande à l'ensemble des serveurs Olan connus le poids correspondant au composant et décide alors du meilleur site répondant à ces critères en choisissant celui de poids le plus élevé. En cas d'égalité de poids, le site local, i.e. l'hôte du Configurateur de Session est celui préféré.

III.2.2. Placement

Les Contextes

Une fois choisis l'utilisateur et le site sur lequel un composant doit être placé, le Configurateur de Session y crée un *Contexte*. Un contexte est un espace de configuration et d'exécution de composants. Les composants sont placés puis installés au sein d'un contexte. Il en est de même pour les composites et les collections. Les connecteurs sont quant à eux des objets multi-contextes, car, conformément à leur schéma de fonctionnement (cf. section *II. Mise en Œuvre des* Composants, pp.90), ils sont présents à la fois dans le contexte du composant émetteur comme dans ceux des destinataires.

Il ne peut exister plus d'un contexte par utilisateur sur un site donné au sein d'une session d'application. Un contexte est donc identifié de manière unique par le nom de la session de l'application, l'identifiant de l'utilisateur et le nom du site. Lors du choix du placement d'un composant, si un contexte existe déjà avec les mêmes caractéristiques (utilisateur, site et session), il n'y a pas de nouvelle création de contexte, il est directement utilisé. En conséquence, tous les composants d'un utilisateur, sur un site pour une session d'application donnée, sont regroupés dans un même contexte.

Les Configurateurs de Contextes

Chaque contexte est géré par un objet particulier appelé le *Configurateur de Contexte*. Son rôle est d'effectuer toutes les opérations de configuration intervenant à l'intérieur du contexte dont il a la charge : les créations de composants, la mise en place du logiciel intégré des primitifs, les demandes d'interconnexion,... Le Configurateur de Session interprète le script de configuration, choisi les

contextes de placement, effectue la création des contextes si nécessaire, puis transmet les opérations de configuration au contexte le plus apte à les effectuer.

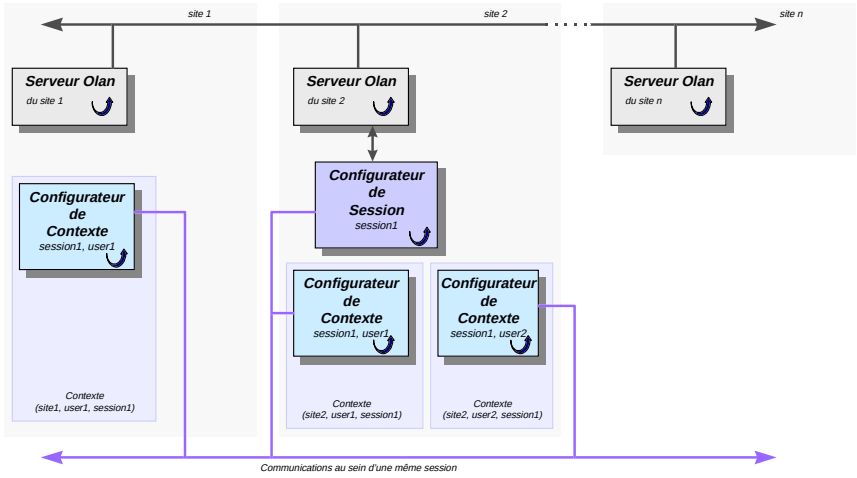

Figure 68. Découpage en Contexte d'une Application

Il est possible que pour certaines opérations un Configurateur de Contexte ne puisse l'effectuer en totalité par lui-même. Par exemple, dans le cas de la mise en place d'une interconnexion, le contexte contenant le composant émetteur est capable de mettre en place la partie émettrice du connecteur mais d'autres Configurateurs de Contexte doivent effectuer la création des parties réceptrices. Chaque Configurateur de Contexte est donc capable de communiquer avec tous les autres Configurateurs de Contextes de la session de l'application, ainsi qu'avec le Configurateur de Session de l'application. Leur mise en œuvre est un objet serveur ILU qui est capable de communiquer avec tous les autres Configurateurs de la session.

III.3. Installation

L'installation de l'application consiste à créer les structures d'exécution correspondantes aux différents composants de l'application, aux collections ainsi qu'aux structures qui prennent en compte l'instantiation dynamique. La création de ces structures commence tout d'abord par l'appel des primitives de configuration contenues dans le script de configuration de l'application. Ces primitives correspondent à des méthodes du Configurateur de Session de l'application. Le Configurateur de Session choisi le site, comme nous l'avons décrit dans la section précédente, demande la création du contexte et du Configurateur de contexte si nécessaire, puis appelle la méthode correspondante de l'objet Configurateur de Contexte. Ce dernier va effectuer alors la création du composant, de la collection ou d'une structure pour l'instantiation dynamique. Pour ce faire, il appelle la machine à Composants qui se charge de retrouver le code dans le référentiel et met en place l'objet composant ou collection ainsi que les modules de code encapsulés s'il s'agit d'un composant primitif. C'est à ce moment qu'existe dans le contexte, des objets ou modules comparables à ceux décrit dans la Figure 62 et Figure 63. Dans certains cas, le Configurateur doit avant l'installation, vérifier si le code intégré

dans un primitif n'a pas des besoins particuliers comme l'accès à des exécutables particuliers ou le lancement d'un programme. Par exemple, dans le cas de primitifs intégrant des objets Java, le Configurateur de Contexte vérifie s'il faut lancer un interpréteur Java. Dans l'affirmative, un nouveau processus est lancé et les mécanismes de communication entre le contexte et l'interpréteur sont mis en place (cf. section *Module Logiciel Java*, pp.93)

Nous allons maintenant présenter brièvement les cas d'installation des composites, des collections et des composants à instantiation dynamique. Nous ne développerons pas outre mesure cette section qui concerne essentiellement la Machine à Composants du support Olan, mais il nous paraît intéressant de fournir au lecteur un aperçu du fonctionnement des ces entités.

Composites et Collections

Les composites sont des composants qui n'ont aucune fonction algorithmique : ils servent essentiellement à structurer des ensembles de composants en une hiérarchie et à regrouper des composants ayant des propriétés applicatives corrélées. Nous avons abordé la raison de leur présence dans la section *II.1.3. Autres Entités de* OCL pp.94. Principalement, les composites existent pour conserver la structure hiérarchique au sein du support afin de pouvoir s'appuyer sur elle pour des opérations de supervision de l'application. Les outils envisagés pour ce faire, reposeront sur la même représentation visuelle que celle utilisée par le concepteur, permettant de faire remonter les informations sur le fonctionnement de l'application de manière plus synthétique et plus pertinente. La seconde raison est expliquée plus loin et illustré dans la Figure 69. Elle concerne possibilité de relier des services en enchaînant l'utilisation de plusieurs connecteurs.

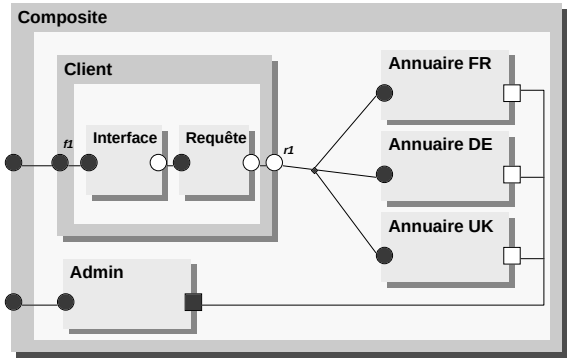

Figure 69. Utilisation des Composites

L'exemple se réfère à l'application Annuaire dans laquelle le composant Client est un composant composite. A l'intérieur, un client est composé d'une interface graphique qui effectue le dialogue avec l'utilisateur et d'un composant de préparation et d'envoi de requêtes vers l'extérieur. Le but de l'application consiste toujours à envoyer la requête vers un ensemble d'annuaires, la sélection de ceux-ci se faisant grâce à la désignation associative. Nous pouvons voir ici que le composite permet d'utiliser un connecteur avec une sélection de l'annuaire à l'extérieur du client, alors qu'à l'intérieur ceci est invisible.

A l'exécution, il est intéressant d'avoir l'objet composite qui permet de configurer différemment les interconnexions entre lui-même et d'autres composants de celles entres ses sous composants et lui-même.

Lors de l'exécution, un composite est représenté par plusieurs objets identiques, dupliqués sur chacun des contextes contenant un de ses sous composants. Chaque objet ressemble aux objets composants des primitifs avec comme propriétés supplémentaires la connaissance des références de chacun de ses sous composants et de chaque duplicata. La Figure 70 schématise la structure d'un composite.

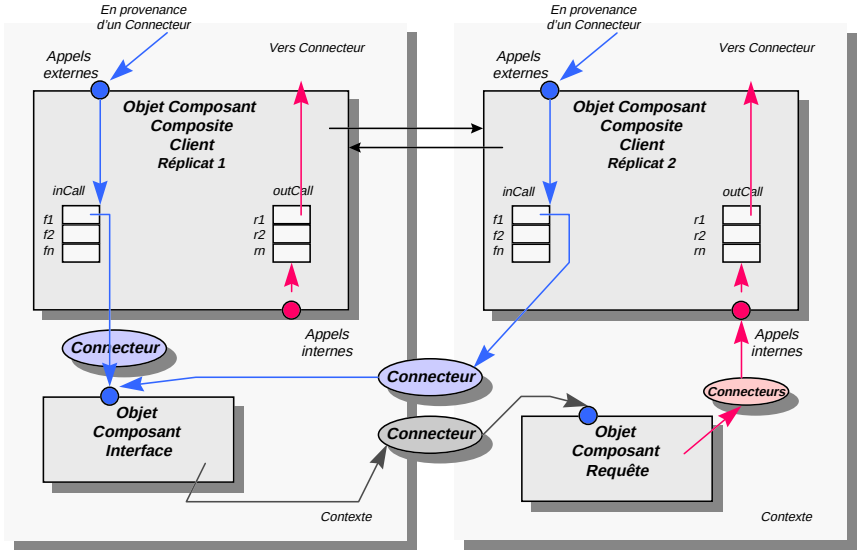

Figure 70. Structure des Composites

Un appel à n'importe quel duplicata de composite est redirigé vers le sous composant correspondant même s'il n'est pas dans le même contexte. C'est pour cette première raison que les références contenues dans les tables `inCall` des duplicatas pointent sur des connecteurs, car il faut éventuellement effectuer des appels vers un contexte différent. De plus, le lien entre ces tables et les sous composants correspondent à des projections définies dans le langage OCL. Or ces projections peuvent éventuellement utiliser des connecteurs spécifiques ou des clauses de désignation associative, ce qui implique l'utilisation d'un connecteur à l'exécution. Ces liens sont mis en place par la primitive `Map` de la Machine à Configuration.

La création d'un nouveau sous composant sur un contexte différent de ceux sur lesquels le composite est dupliqué, induit la création d'un duplicata sur ce contexte. Le composite est donc une entité extensible. L'intérêt de ceci est représenté sur la Figure 71 car cela permet de minimiser les appels entre contextes différents.

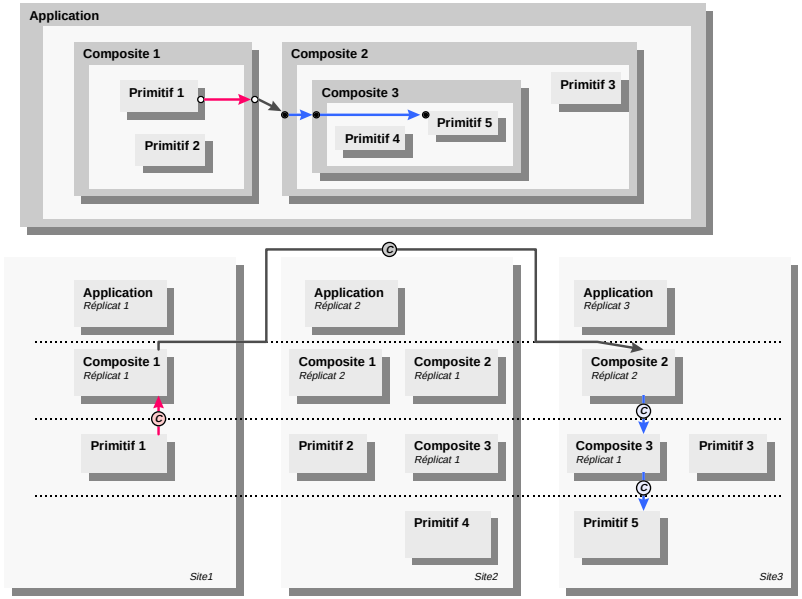

Figure 71. Chaîne d'appel avec des Composites

Le Composant *Primitif1* doit communiquer avec le composant *Primitif5*. Ils sont situés sur des sites différents, à des niveaux de hiérarchie différents et dans des composites différents. D'après notre architecture, ceci est coûteux car il est nécessaire de faire des appels par des connecteurs d'un sous composant vers son composite jusqu'au sommet de la hiérarchie, puis de redescendre vers le destinataire de la communication en traversant aussi des composites et des connecteurs. Pour minimiser ceci, la duplication des objets composites permet de ne conserver que des appels locaux jusqu'au dernier appel vers le primitif destinataire. Ici, la communication passe du *Primitif1* vers le duplicata du *Composite1* du même contexte. Puis, la communication doit s'étendre sur le *Composite2* qui n'est pas dans le même contexte. Le flot part alors vers le duplicata du *site3*, puis vers le duplicata de *Composite3* du même contexte et enfin vers le *Primitif5*. Il est évident que l'appel entre deux primitifs est coûteux car il y a plusieurs indirections via des connecteurs, mais l'appel entre sites ne s'effectue qu'une seule fois.

La duplication des objets composites est adaptée au cas des composites, car les informations dupliquées évoluent peu pendant l'exécution : le contenu des tables `_inCall` et `_outCall` ne sont modifiés seulement lors de la création du composite et dès qu'un sous-composant à instantiation dynamique est créé. La mise en cohérence des différentes tables n'est donc pas un facteur gênant pour l'exécution de l'application de par leur caractère peu fréquent.

Les collections sont identiques du point de vue de la structure des composites avec en sus la faculté de créer et de supprimer un des sous composants.

Instantiation Dynamique

La dernière structure qui peut être mise en place lors de la phase d'installation correspond à celle d'un composant qui doit être instanciée dynamiquement lors de sa première utilisation. La structure correspondante est un objet dont l'interface extérieure est identique à un objet composant primitif ou composite, selon le cas. Il est appelé "*objet dynamique*". Toutefois, cet objet ne contient aucun module logiciel ou sous composant. Il contient par contre le code de configuration correspondant à sa création ainsi que celui correspondant à la création de ce qu'il contient.

Lors du premier appel à cet objet, le code de configuration est transmis au Configurateur de Contexte, qui le transmet au Configurateur de Session. Ce dernier l'évalue, décide des sites, puis retransmet aux différents Configurateurs de Contexte les opérations de configuration convenables. Les modules logiciels d'un primitif ou bien les sous composants d'un composite sont alors créés. Les interconnexions internes sont mises en place et en dernier lieu les interconnexions vers ou depuis "l'objet dynamique" sont défaites puis mises en place de nouveau.

III.4. Interconnexions

La phase d'interconnexion correspond à la mise en place des connecteurs entre les composants. Les connecteurs interviennent pour toutes communications entre composants. Ces communications interviennent à deux niveaux : lors d'une interconnexion entre deux composants contenus dans un même composite, et au niveau de la projection entre l'interface d'un composite et des sous composants. Ces deux opérations correspondent respectivement aux primitives `Bind` et `Map` de la Machine à Configuration.

La mise en place d'une interconnexion est initiée par le Configurateur de Contexte du composant émetteur dans une interconnexion. La main est ensuite passée au gestionnaire de la bibliothèque de connecteurs qui met en œuvre un algorithme de reconnaissance des contextes dans lesquels une partie émettrice ou réceptrice des connecteurs doit être créée. Puis, une fois que ces différentes parties existent, le code du connecteur et la configuration du mécanisme de communication demandé est effectuée en fonction du type de chaque connecteur. Au niveau des composants, les "branchements" entre les parties émettrices ou réceptrices, et les tables de sorties `outCall` et `inCall` sont alors effectués.

Nous ne détaillerons pas plus dans ce manuscrit le fonctionnement interne de la bibliothèque de connecteurs et le fonctionnement de l'utilisation du mécanisme de communication qui fait l'objet d'un travail autre.

III.5. Scripts de Configuration

Le script de configuration est produit par le compilateur OCL. C'est actuellement le "seul utilisateur" de la machine à configuration. Il contient une suite d'appel à des primitives de configuration que le Configurateur de Session doit appeler. Le résultat de l'appel de ces primitives est le déploiement et l'installation complète de l'application.

Le script de configuration est construit en suivant un ordre précis pour éviter tous problèmes lors de son interprétation. En effet, l'ordre d'utilisation des primitives de la machine à Configuration est important : une interconnexion ne peut pas être effectuée si les composants ne sont pas existants, un composite ne peut être créé sans les sous composants, etc. Pour éviter tout problème, la construction

du script par le compilateur repose sur un parcours à partir des feuilles de la hiérarchie de composants. Les composants primitifs sont d'abord créés, puis les composites du niveau supérieur. Ensuite les interconnexions entre les primitifs sont mises en place, puis les projections avec ses sous composants et finalement les attributs des sous composants sont positionnés. Les autres niveaux de la hiérarchie se font sur le même principe, de manière récursive.

Prenons l'exemple de l'application Annuaire tiré des **Figure 47** et **Figure 48** auquel on a ajouté des clauses d'administration comparables à celles de la **Figure 53**. Le script de configuration est le suivant :


```
## Création des Primitifs
## Création du composant primitif Client sans contrainte de placement
CreatePrimitiveComponent('annuaire_ClientImpl:annuaire_ClientImpl:_1',
                                 None, None);
## Création du 1er composant primitif Annuaire, avec contrainte de placement
nInfo = ( 'name' :    "(node.name [1:] == "db") and
                      (node.name[3:-1] == ".inrialpes.fr")",
        'IPAdr' : "true",
        'platform' : "true",
        'osType' : "true",
        'osVersion' : "true",
        'CPULoad' : "node.CPULoad <= 10",
        'UserLoad' : "node.UserLoad <= 10"
        )
uInfo = ('name' : "user.name == "admin",'uid' : "true",'grpId": "true")
CreatePrimitiveComponent('annuaire_AnnuaireImpl:annuaire_AnnuaireImpl:_2',
                      uInfo, nInfo)
## Création du second Annuaire avec les mêmes contraintes de placement
CreatePrimitiveComponent('annuaire_AnnuaireImpl:annuaire_AnnuaireImpl:_3',
                      uInfo,  nInfo)
##Création du composant d'administration avec des contraintes de placement
uInfo = ('name' : "user.name == "admin","uid" : "true","grpId": "true")
CreatePrimitiveComponent('annuaire_AdminImpl:annuaire_AdminImpl:_4',
                      uInfo, None)

## Création des Composites de niveau supérieur
## Création du Composite représentant l'application
CreateCompositeComponent('annuaire:AppliImpl:_0',
                      ['annuaire_ClientImpl:annuaire_ClientImpl:_1',
                      'annuaire_AnnuaireImpl:annuaire_AnnuaireImpl:_2',
                      'annuaire_AnnuaireImpl:annuaire_AnnuaireImpl:_3',
                      'annuaire_AdminImpl:annuaire_AdminImpl:_4'],
                      None, None )

## Création des Projections
## LanceClient() -> client.Init()
Map ('annuaire:AppliImpl_syncCall_0:AppliImpl_0_48',
        [('annuaire:AppliImpl:_0', 'LanceClient', 'impl')],
        [('annuaire_ClientImpl:annuaire_ClientImpl:_1', 'Init', 'itf')])

## LanceAdmin() -> admin.Init()
Map ('annuaire:AppliImpl_syncCall_1:AppliImpl_1_49',
        [('annuaire:AppliImpl:_0', 'LanceAdmin', 'impl')],
        [('annuaire_AdminImpl:annuaire_AdminImpl:_4', 'Init', 'itf')])

## Création des interconnexions
## client.Lookup(cle, email, pays) ->
##             annuaireFr.Lookup(cle, email) ,
##             annuaireUk.Lookup(cle, email)
##             using syncCall
##             where (annuaireFr.Pays) == pays or (annuaireUk.Pays == pays)
Bind('annuaire:AppliImpl_syncCall_2:AppliImpl_2_50',
        [('annuaire_ClientImpl:annuaire_ClientImpl:_1', 'Lookup_Annuaire',
                'itf')],
        [('annuaire_AnnuaireImpl:annuaire_AnnuaireImpl:_2', 'Lookup', 'itf'),
         ('annuaire_AnnuaireImpl:annuaire_AnnuaireImpl:_3', 'Lookup', 'itf')])

## annuaireFr.Sendstats() -> admin.GetReport() using asyncCall
Bind('annuaire:AppliImpl_asyncCall_0:AppliImpl_0_51',
        [('annuaire_AnnuaireImpl:annuaire_AnnuaireImpl:_2', 'SendStats', 'itf')]
        [('annuaire_AdminImpl:annuaire_AdminImpl:_4', 'GetReport', 'itf')])

## annuaireUK.Sendstats() -> admin.GetReport() using asyncCall
Bind('annuaire:AppliImpl_asyncCall_1:AppliImpl_1_52',
        [('annuaire_AnnuaireImpl:annuaire_AnnuaireImpl:_3', 'SendStats', 'itf')]
        [('annuaire_AdminImpl:annuaire_AdminImpl:_4', 'GetReport', 'itf')])

## Positionnement des Attributs
SetAttribute('annuaire_AnnuaireImpl:annuaire_AnnuaireImpl:_2',
        '_a_Pays',
        "('string*', 'FRANCE')")
SetAttribute('annuaire_AnnuaireImpl:annuaire_AnnuaireImpl:_3',
        '_a_Pays',
        "('string*', 'UK')")
```

Figure 72. Script de Configuration de l'Annuaire

La syntaxe du script est en Python. On peut remarquer les noms OCL, générés automatiquement par le compilateur. Ils contiennent en première partie le nom du module, en seconde le nom de la classe associée à l'objet et enfin un identifiant de l'instance. Le module est recherché dans le référentiel et il contient le code de l'objet correspondant. Les connecteurs sont aussi chargés par ce procédé. Chaque interconnexion est représentée par un objet particulier qui contient le code additionnel au connecteur de base pour effectuer les transformations de paramètres ou la désignation associative. Ceci rend les clauses de désignation associative ou de transformation de paramètres invisibles au niveau des primitives de configurations car elles sont contenues dans le code associé au connecteur.

On peut voir la construction des critères de placement conformément à ceux décrits en section *Informations de Placement*, p.98. Enfin, remarquons les primitives de positionnement de la valeur d'un attribut. Le nom de l'attribut est fourni ainsi que la valeur qu'il doit prendre, valeur décrite dans le format interne Olan linéarisé de description des paramètres.

III.6. Scripts de Lancement

Le script de lancement permet à un utilisateur de lancer l'exécution d'une session d'une application ou bien de rejoindre une session en cours d'exécution d'une application. Son utilisation se fait de la manière suivante :

```
nom_script [-help] nom_Session nom_Service [paramètres]
```

Lance une session de l'application Olan de nom `nom_script`. Cette session est désignée par `nom_Session`. Le service appelé en premier se nomme `nom_Service` et il doit être défini dans le programme OCL comme un service de lancement de l'application. Si le service demande des paramètres, ils doivent être fournis selon le format pivot de représentation des types sous forme d'une chaîne de caractères.

Ce script est généré automatiquement par le compilateur OCL. Voici son algorithme :


```
Récupération des arguments, en particulier le nom de la session, le
nom du service appelé et les éventuels paramètres.

Vérification de l'existence du service comme point d'entrée de
l'application.

Vérification de la validité des paramètres

Vérification de l'existence de la session.

Si la session existe,
Alors
    récupération de la référence vers l'objet Configurateur de
    Session et appel de la primitive ExecuteComponentService avec les
    paramètres requis.
Sinon
    Création d'un objet Configurateur de Session et enregistrement de
    la Session dans les serveurs Olan

    Interprétation du script de configuration par l'objet
    Configurateur de Session. Les contextes, puis les composants,
    collections, interconnexions, ... sont alors mis en place.

    En cas d'impossibilité de déploiement, un message prévient
    l'utilisateur et lui demande de modifier la configuration
    initiale de l'application

    Lancement de l'application par l'appel à la primitive
    ExecuteComponentService avec les paramètres requis.
```

L'intérêt de ce script est de cacher à l'utilisateur la complexité de lancement d'un programme et de permettre la création de nouvelles sessions d'applications comme la connexion à une session existante.

IV. Discussion

IV.1. Déploiement en environnement hétérogène

En règle générale, les supports de communication distribuée du type RPC, CORBA, bus de message demandent au démarrage de l'application de lancer manuellement chacun des objets sur les sites désirés. Il n'y a donc pas de phase de déploiement, tout se fait manuellement. Le déploiement de composants dans un environnement distribué consiste en premier lieu en la création de composant à distance sur un site capable de supporter l'exécution de l'application. DCOM, les bus logiciels de Polylith, CONIC ou Regis sont des supports d'exécution de composants qui autorisent cette action. La création de composants, et en particulier le choix du site de création, est dirigée soit par le programme lui-même comme dans le cas de DCOM, soit par un langage externe comme le MIL de Polylith ou les langages CONIC et Darwin. Tous ces outils ont en commun de devoir définir exactement le site qui supporte la création, de manière absolue. Ils ne permettent pas de choisir un site de manière indirecte parmi un ensemble ayant des caractéristiques communes. Ceci pose un problème, car il existe toujours un risque que le déploiement ne puisse être effectué, par exemple si une machine est momentanément hors service, si la machine a été remplacée,...

Le support Olan offre au niveau de ses primitives la possibilité d'exprimer des critères de choix de sites en fonction de propriétés des machines, des systèmes d'exploitation ou même de charge moyenne d'une machine. La désignation indirecte du site permet alors un degré de souplesse du déploiement car il est possible de choisir parmi des sites aux caractéristiques équivalentes et donc de garantir un nombre de déploiements potentiels plus important.

De plus, la spécification du site est indépendante de l'implémentation des composants, car elle est contenue dans une section séparée alors que Polylith, CONIC, Darwin ou UniCon incorporent le placement dans la configuration des composants. Ceci est aussi un gage de souplesse car il n'est pas nécessaire de modifier la description de l'architecture mais seulement les clauses d'administration qui contiennent les indications de placement des composants.

IV.2. Limites Actuelles

Le niveau de réalisation actuel du support Olan ne permet pas de mettre en place toutes les propriétés exprimées dans un programme OCL. En particulier, le concept de collection est en cours de réalisation. Le service de désignation associative ainsi que son mode d'utilisation n'est pas encore défini ce qui explique le peu d'information à ce sujet dans ce présent manuscrit. Ce travail fait l'objet d'une autre thèse au sein du projet SIRAC.

Algorithme de placement

L'algorithme de choix des sites au niveau du support est relativement simple. Le choix s'effectue uniquement au niveau des composants primitifs en fonctions des critères de placement générés par le compilateur. Il n'est pas possible d'utiliser des critères de placement dépendant de plusieurs composants comme la co-localisation au niveau du support de configuration. Ceci est aussi en cours de réalisation.

En cas d'impossibilité de trouver un site, il n'existe pas encore d'autre solution que d'arrêter l'installation de l'application pour définir une nouvelle configuration. Nous avons envisagé de rendre ce processus interactif avec un administrateur d'applications qui pourrait décider, avec l'aide du système, du meilleur placement possible parmi un ensemble déduit par le support. Il reste un certain nombre de pistes à suivre mais nous pensons que l'architecture présentée permet d'envisager l'une ou l'autre piste. En effet, le choix des sites se fait au niveau du Configurateur de Session, qui peut interpréter totalement le script de configuration avant de lancer les ordres aux différents Configurateurs de Contexte. Il est donc possible de mettre en œuvre un algorithme qui effectue plusieurs choix de placement avant d'installer effectivement l'application sur des sites, dans notre cas des contextes.

Optimisations de la chaîne de communication

La communication entre des composants peut être à juste titre considérée comme coûteuse, car dans le meilleur des cas, deux composants communiquent via les objets composants, puis les objets connecteurs. Il y a donc beaucoup d'indirections pour faire communiquer deux portions de code ensemble. Ce prix à payer nous semble acceptable au regard des facilités de configuration et de réutilisation de composants dans différentes architectures.

Toutefois, cela commence à devenir moins acceptable dès lors que les composites entre en jeu. En effet, les composites n'ont aucune fonction algorithmique, ce sont des entités de structuration et

d'administration. Leur autre intérêt est de permettre la combinaison de plusieurs types de connecteurs dans une chaîne de communication entre composants primitifs. Malgré cela, la communication entre composants contenus dans des composites différents doit traverser les objets composites, ajoutant par la même des indirections supplémentaires. Ces indirections sont d'autant plus coûteuse qu'elles sont mises en œuvre via des connecteurs. Nous avons pensé à faire un certain nombre d'optimisation lorsque cela était possible. Par exemple, la suppression des objets connecteurs lorsqu'un appel de méthode pouvait être utilisé, la suppression du passage par les composites pour n'effectuer que les appels entre primitifs. Ce dernier cas n'est possible que si les connecteurs utilisés tout au long de la communication sont identiques. Toutes ces optimisations sont en cours d'étude, sachant que certaines solutions rendront l'application plus performante mais ne permettront peut-être plus les extensions envisagées autour de la supervision de l'exécution et plus tard de la reconfiguration en cours d'exécution d'une application.

Chapitre 5
Exemples d'Utilisation

L'objectif de ce chapitre est de rapporter l'expérience de programmation d'applications de taille raisonnable avec l'environnement de programmation Olan. Nous nous attacherons à deux applications particulières : CoopScan[Bel96], une application d'édition coopérative, et le Mailer, une application de filtrage automatique de courrier électronique au travers d'entités spécifiques appelées des agents.

Nous allons présenter ces deux applications en proposant leur architecture décrite en OCL et en effectuant une évaluation des avantages et inconvénients de l'utilisation de l'environnement Olan pour les construire.

I. Application CoopScan

I.1. Présentation

CoopScan[Bel96] est une architecture générique pour la construction d'applications coopératives. Ces applications ont pour objectif d'utiliser des programmes initialement mono-utilisateur et de les intégrer dans un environnement réparti où plusieurs utilisateurs interagissent autour de tâches communes. CoopScan offre un modèle d'architecture pour faire coopérer des logiciels existants en fournissant diverses politiques de coopération entre utilisateurs.

L'application que nous décrivons ici est un environnement de téléconférence qui supporte le travail simultané de plusieurs utilisateurs. Ceux-ci peuvent interagir à travers d'un canal audio de communication et par des documents partagés. Nous ne détaillerons par toute l'architecture de l'application qui peut être trouvée dans [Ben97] mais nous étudierons le cas d'un scénario simplifié où les utilisateurs ne communiquent qu'à travers des documents partagés. Plusieurs types de documents sont accessibles au travers d'éditeurs de grande diffusion comme Xfig, XEmacs, Thot, ... CoopScan apporte des fonctions de coopération WYSIWIS (What You See Is What You See), permet la connexion et déconnexion dynamique de participants et gère des rôles et droits d'accès aux logiciels comme aux documents partagés.

Le fonctionnement de l'application est le suivant. Un utilisateur qui entre dans le processus de coopération intègre une session qui lui donne accès aux documents partagés. L'utilisateur possède alors deux composants logiciels : l'application, qui est le programme d'édition avec son interface graphique, et le *Contrôleur* qui gère la coopération entre les utilisateurs et l'application. Le composant contenant l'application intercepte toutes les actions de l'utilisateur et les transmet au contrôleur qui les autorise ou non. Ainsi, le composant contrôleur est à même de distribuer les droits d'intervention des utilisateurs dans l'application. Il faut noter que chaque utilisateur possède un composant application et

un composant contrôleur. Il y a une duplication complète de ces composants et CoopScan propose des politiques et protocoles de dialogues et de mise en cohérence des informations de ces modules dupliqués. En règle générale, ces deux composants sont localisés sur un même site pour un utilisateur donné.

La coopération fonctionne ainsi : une action effectuée par l'utilisateur au travers de l'interface graphique de l'éditeur, est transmise au contrôleur local qui vérifie si l'utilisateur possède les autorisations requises pour cette action. Dans la négative, l'action est invalidée au niveau de l'interface graphique de l'éditeur et l'utilisateur est prévenu de cette interdiction d'action. Dans le cas positif, cette action est alors diffusée aux autres contrôleurs appartenant aux autres utilisateurs et généralement présents sur des sites différents. Ces contrôleurs reçoivent un descriptif fidèle de l'action effectuée par l'utilisateur pour en demander la reproduction à l'identique sur les interfaces utilisateurs que chacun d'entre eux gère. Ceci permet d'assurer le caractère WYSIWIS de la coopération. De manière plus fine, le contrôleur est lui-même scindé en plusieurs entités : l'Agent local, l'Agent de Connexion et des Agents distants. L'agent local est l'interface entre l'application et le contrôleur de la coopération, l'agent de connexion gère les connexions et déconnexions de participants et les agents distants sont les objets qui représentent l'application distante et permettent de rejouer les actions reçues d'autres contrôleurs. Il existe un agent distant pour chaque utilisateur distant présent dans la coopération.

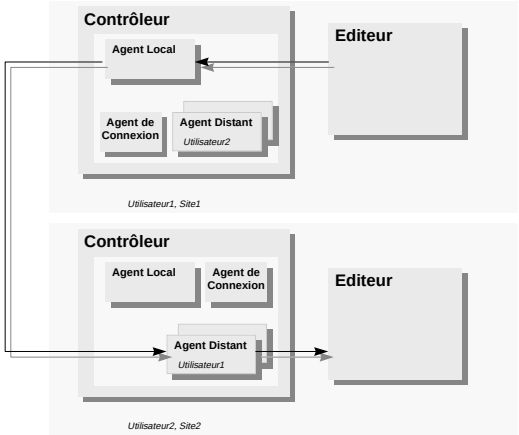

Figure 73. Principe de la Coopération dans CoopScan

La Figure 73 montre une communication entre l'éditeur de l'utilisateur1 et les éditeurs distants (ici celui de l'utilisateur2). L'action effectuée par l'utilisateur1 sur son éditeur est transmise à son agent local, qui décide ou non de la diffuser aux éditeurs distants en passant par les agents distants qui le représentent.

I.2. Description OCL

L'Editeur

Le composant Editeur est un composant primitif dont le rôle est d'encapsuler l'accès et l'utilisation de l'application d'édition utilisée dans CoopScan. Le code de ce composant est composé de modules écrit en langage C ainsi que de l'exécutable de l'éditeur structuré Thot[Qui94]. Cet éditeur est une application ouverte car elle possède une API ainsi qu'un mécanisme de notification (callback) permettant d'envoyer vers l'extérieur un événement chaque fois que l'utilisateur effectue une action sur l'interface graphique. Ce mécanisme de notification s'appelle External Call Facility (ECF)[Qui94]. A chaque envoi d'événement, l'ECF attend un retour de type booléen qui, s'il a la valeur vraie, permet à l'interface de l'éditeur de valider l'action entreprise par l'utilisateur. Dans le cas contraire, l'action n'est pas autorisée par l'éditeur. Les modules C permettent d'une part le lancement de l'exécutable Thot, l'interception des événements provenant de l'éditeur ainsi que le dialogue avec l'API.

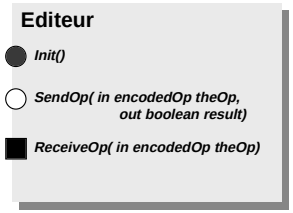

Figure 74. Composant éditeur

Vu de l'extérieur, le composant primitif possède l'interface suivante :

```
interface Editor {
        provide Init();
        require SendOp(in encodedOperation theOperation, out boolean result);
        react ReceiveOp(in encodedOperation theOperation);
}
```

Figure 75. Interface de l'Editeur

Nous pouvons remarquer que le service `ReceiveOp`, qui appelle l'API de l'éditeur pour lui demander d'effectuer des actions, est un service de type `react`. Cela signifie qu'il existe un flot d'exécution propre à l'intérieur du composant primitif, qui traite la réception de l'opération pour la transmettre à l'éditeur.

Contrôleur de la Coopération

Le composant contrôleur de la coopération est un composite scindé en trois types de composants : l'agent de connexion, l'agent local et les agents distants. Ces différents types d'agents sont des composants primitifs dont les rôles respectifs sont les suivants :

L'agent local récupère les informations en provenance de l'éditeur pour retourner un booléen qui indique si l'utilisateur possède les autorisations pour effectuer cette action. Il gère donc la cohérence des actions utilisateurs au sein d'une session de coopération. Si l'action est autorisée, celle-ci est alors transmise vers l'extérieur du composant Contrôleur, en l'occurrence vers d'autres utilisateurs.

L'agent Distant a pour fonction de récupérer des actions en provenance de l'extérieur du composant Contrôleur pour les retransmettre vers le composant de l'application. Il peut dans certains cas avoir une influence sur le contenu des actions provenant de l'extérieur en fonction des droits de coopération que possède l'utilisateur.

L'agent de Connexion est en charge de gérer l'arrivée ou le départ de l'utilisateur d'une session, ainsi que de récupérer les informations de connexion ou de déconnexion en provenance de l'extérieur du composant Contrôleur.

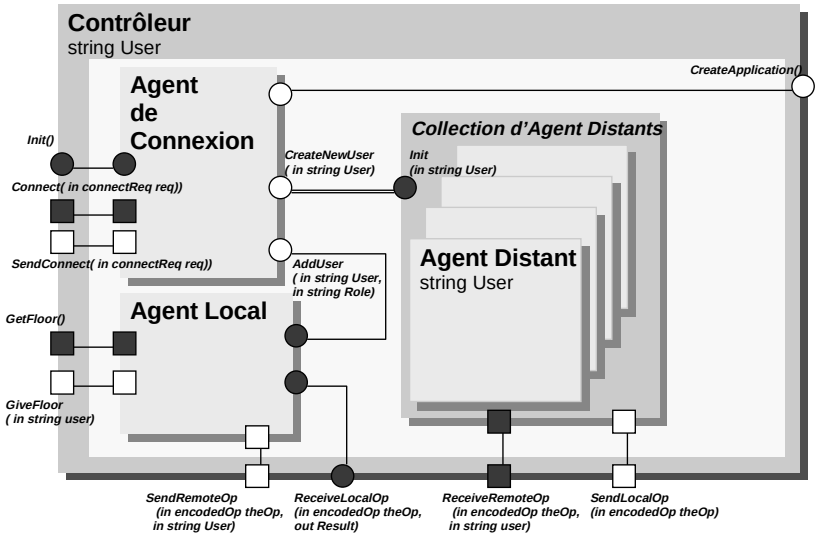

Figure 76. Le Contrôleur de CoopScan

Le contrôleur est un composant composite qui utilise les trois agents précédemment décrits. Son architecture interne illustré sur la Figure 76 montre trois groupes de services : les services pour gérer les connexions et déconnexions à une session de coopération (services `Connect` et `SendConnect`), les services pour le passage des droits de coopération (services `GiveFloor` et `GetFloor` qui permettent de transmettre le droit de parole d'un utilisateur à un autre) et les services pour transmettre ou recevoir les opérations effectuées sur un éditeur et devant être rejouées (services `ReceiveLocalOp` pour recevoir une opération de l'éditeur local, `SendRemoteOp` pour envoyer cette action vers d'autres utilisateurs, `ReceiveRemoteOp` pour recevoir une opération d'un autre utilisateur et `SendLocalOp` pour rejouer une action sur l'éditeur local). Nous pouvons voir aussi l'utilisation des attributs pour conserver le nom des utilisateurs. Au niveau du contrôleur, l'attribut `User` correspond à l'utilisateur possédant le contrôleur, alors que les attributs `User` des agents distants contiennent le nom de l'utilisateur distant représenté par cet agent. Ces derniers attributs servent pour l'accès aux différents agents distants de la collection. Lorsqu'une opération est reçue, elle contient en paramètre le nom de l'utilisateur qui envoie l'opération. La désignation associative permet alors de choisir l'agent distant correspondant à cet utilisateur (cf. Figure 77)

```
implementation ContrImpl : ContrItf
        use LocalAgentItf, DistantAgentItf, ConnectionAgentItf
{
        local = instance LocalAgentItf();
        distants = collection [0..inf] of DistantAgentItf();
        connect = instance ConnectionAgentItf;
...
        ReceiveRemoteOp(op,user) => distants.ReceiveOp(op)
                        where distants.User == user
                        using asyncCall();
...
        connect.CreateNewUser(u) => distants.Init(u)
                        using createInCollection();
...
}
```

Figure 77. Configuration du Composite Contrôleur

La Session Utilisateur

La session utilisateur est un composite qui regroupe les deux composants précédents que sont l'éditeur et le contrôleur. Il représente tous les composants appartenant à un utilisateur sur un site, coopérant au sein d'une session de l'application. Ce composite permet ainsi de cacher la complexité des composants internes pour ne rendre visible qu'un certain nombre de services.

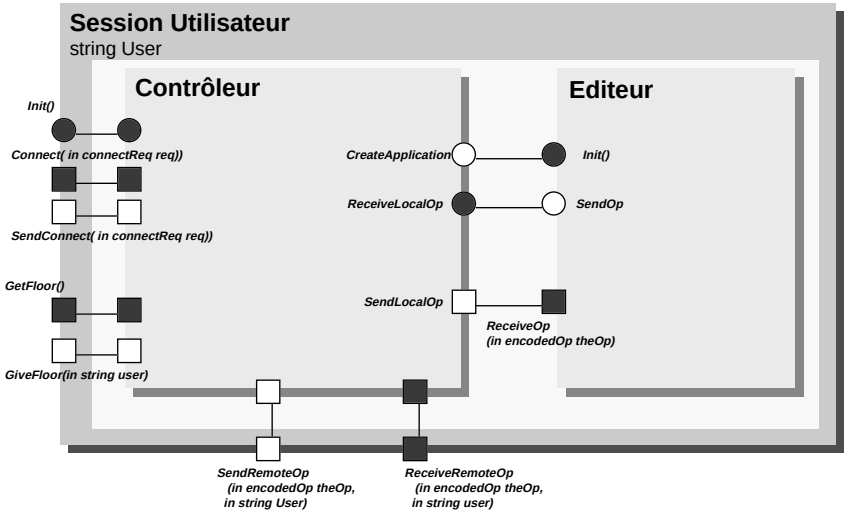

Figure 78. Composant Session Utilisateur de CoopScan

Ces services sont ceux liés à la connexion et déconnexion des utilisateurs, ceux liés au passage de droit de parole entre les utilisateurs et ceux dédiés à la transmission des actions communes à chaque utilisateur.

L'application CoopScan

L'application est une collection de session utilisateur. A chaque arrivée d'utilisateur, une nouvelle instance de composant dans la collection est créée. Celle-ci rejoint la session existante si elle n'est pas seule dans la collection. Un protocole de connexion dynamique de participants est alors utilisé par le

biais des agents de connexion présents dans le composant Contrôleur. En régime continu d'utilisation, la transmission du droit de parole se fait par les services `GiveFloor` et `GetFloor` grâce à la désignation associative de l'utilisateur destinataire du jeton de parole. Les opérations effectuées sur un éditeur par un utilisateur sont diffusées à l'ensemble des participants de la session de coopération. Pour cela, un connecteur particulier `broadcastAsyncCall` a été créé pour diffuser un événement à un ensemble de destinataires.

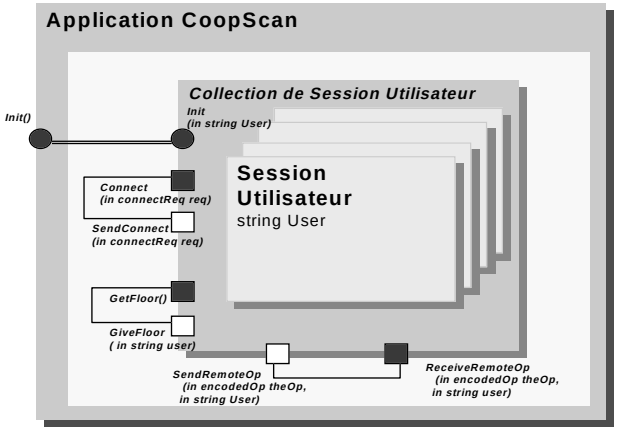

Figure 79. Application CoopScan

La configuration OCL qui est associée à ce composant est la suivante :

```
implementation CoopImpl : CoopItf
        use UserSessionItf
{
        sessions = collection [0..inf] of UserSessionItf();
        Init() => sessions.Init()
                using createInCollection();

        sessions.SendConnect(req) => sessions.Connect(req)
                                using broadcastAsyncCall();
        sessions.GiveFloor(user) => sessions.GetFloor()
                                where sessions.User == user
                                using asyncCall();
        sessions.SendRemoteOp() => sessions.ReceiveRemoteOp()
                                using broadcastAsyncCall();
}
```

Figure 80. Code OCL de l'application CoopScan

La distribution est ici implicite. Il n'y a pas de critères de placement spécifiques pour les composants, tout se fait en fonction des utilisateurs et du site de lancement du programme de connexion à une session par ces derniers.

I.3. Avantages et Inconvénients de l'Utilisation d'Olan

Les avantages que nous avons pu dégager de l'utilisation de l'environnement Olan au travers de cette expérience proviennent essentiellement de la souplesse de création de différentes architectures. Nous

avons présenté dans la section précédente, une version de l'application supportant un nombre non limité d'utilisateur. La politique de coopération est complètement intégrée dans le code de l'agent local, c'est lui qui donne les jetons aux différents utilisateurs en passant en paramètre le nom de l'utilisateur qui reçoit le droit de parole. Nous avons, en utilisant le même agent local, créé l'architecture présentée en Figure 81, qui distribue le jeton en fonction d'une politique en anneau, i.e. un utilisateur après l'autre reçoit le droit de parole.

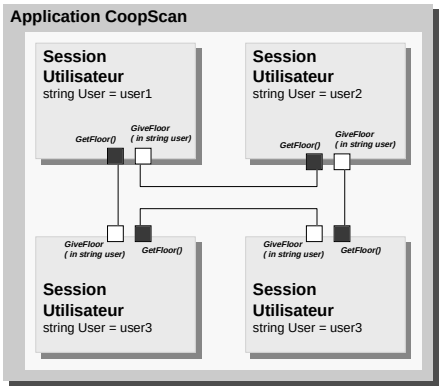

Figure 81. Politique de droit de parole en Anneau

Nous avons aussi expérimenté une autre politique de passage de jeton en créant un utilisateur privilégié, le président, qui reçoit le droit de parole à chaque fois qu'un utilisateur s'en sépare. La Figure 82 montre l'architecture de cette application. Notons que les composants sont les mêmes d'une application à l'autre, seule la configuration des interconnexions change.

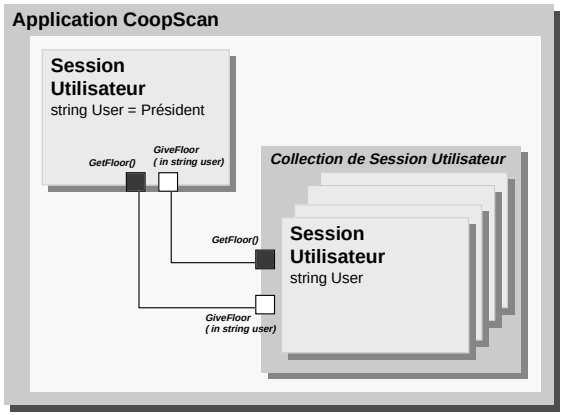

Figure 82. Politique de droit de parole avec un Président

L'autre avantage provient de l'utilisation des connecteurs et de la répartition implicite des composants. En effet, en l'absence de clause d'administration, les composants sont créés sur le site de

lancement de l'utilisateur. Ils peuvent donc être ou ne pas être répartis sur divers sites. Ceci n'a aucune influence sur la programmation de la mise en œuvre des composants, ni sur la définition de l'architecture. Le fait d'utiliser les connecteurs permet de s'affranchir de la programmation des communications comme de la répartition des composants.

I.4. Réalisation

Les composants de l'application CoopScan ont été écrits en C et C++. La configuration et le déploiement de l'application ont été effectués sur une version antérieure du support de Configuration Olan, que nous n'avons pas décrit dans ce manuscrit. Ce support était construit au dessus de la plate-forme répartie à objet OODE[Bul94]. OODE fournissait un langage concurrent orienté objet, extension de C++, qui autorisait la programmation transparente d'objets répartis, partageables avec des mécanismes de contrôle de la concurrence. OODE permettait ainsi d'utiliser l'appel de méthode entre objets de manière identique à un appel local peu importe leur localisation. Dans cette version du prototype, un mécanisme d'événements asynchrones entre objets[Len96] ainsi qu'une version préliminaire de la désignation associative existait[Mar95]. Par contre, le compilateur du langage OCL n'était pas disponible ce qui nous imposait d'écrire manuellement le script de configuration et de lancement de l'application. L'application CoopScan sous cette forme a fait l'objet d'une démonstration lors de la journée du Trophée AFCET sur les transferts de technologies entre le monde de la recherche et le monde industriel.

Cette version a été abandonnée suite à l'arrêt par Bull du développement et du support de OODE d'une part, et à notre volonté d'utiliser une plate-forme plus répandue et standard comme CORBA.

II. Application Mailer

II.1. Présentation

L'objectif de cette application est de réaliser un gestionnaire de messagerie électronique configurable aisément par un usager non-spécialiste. Ce gestionnaire réutilise des outils de messagerie existants, dans l'optique de ne pas modifier les habitudes d'un usager. Le gestionnaire de messagerie considéré dans cette application est Netscape, auquel diverses fonctions supplémentaires sont ajoutées pour le traitement du courrier, comme la possibilité de filtrer automatiquement, de ranger les messages dans des répertoires particuliers, de réémettre automatiquement des messages, etc. L'environnement Olan, et en particulier le formalisme graphique de Olan, permet ainsi à l'utilisateur d'assembler divers éléments de base (eg. l'élément de tri, un élément de rangement,...) pour construire à sa guise une chaîne de traitement. En phase d'utilisation de l'application, tous ces ajouts sont utilisés de manières transparentes, seuls des rapports d'utilisation de la chaîne de traitement sont régulièrement envoyés à l'utilisateur.

Les éléments manipulés par l'utilisateur terminal sont des agents AAA. Leurs propriétés sont détaillées dans la section suivante mais en première approche, ce sont des objets de petites tailles, facilement modifiables, et pouvant être aisément insérés ou supprimés d'une application en cours de fonctionnement. Le reste de l'application est composé du logiciel nécessaire à intégrer l'environnement de messagerie électronique de l'utilisateur et à permettre l'utilisation des agents.

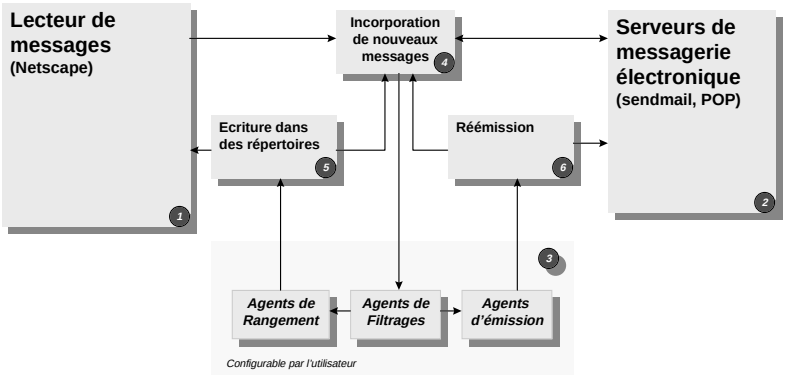

Figure 83. Architecture de l'application Mailer

L'application est ainsi composée de six parties fonctionnelles :

Le lecteur de messages, qui représente l'interface avec l'utilisateur ; C'est à ce niveau que l'utilisateur consulte ses nouveaux messages.

Le serveur de messages qui s'occupe du flux du courrier électronique. Il réceptionne les messages en provenance du monde extérieur et permet d'en envoyer. Cette partie utilise le mécanisme de messagerie présent sur la machine de l'utilisateur.

Le traitement des messages qui contient l'ensemble des agents définis par l'utilisateur et qui forment la chaîne de traitement.

Le gérant d'incorporation, qui réceptionne les messages en provenance du serveur à la demande de l'utilisateur, pour les préparer à passer par la chaîne de traitement. Cet élément gère aussi les erreurs de la chaîne de traitement afin d'éviter toute perte de messages. Il gère aussi la création d'un rapport des opérations, envoyé à l'utilisateur lorsqu'une série de messages a été traitée par les agents.

L'élément d'écriture qui gère les demandes de rangement d'un message dans un répertoire spécifique de l'application.

L'élément de réémission qui gère les demandes d'envoi de courrier électronique vers un autre usager.

II.2. L'Environnement AAA (Agent Anytime Anywhere)

L'environnement AAA est un support d'exécution pour des entités réactives appelés agents. Les agents sont des objets qui se comportent selon un modèle «événement -> réaction » : un événement exprime un changement d'état significatif auquel un ou plusieurs agents peuvent réagir. Dans ce modèle, un événement est représenté par une notification. Une notification est un objet passif émis par un agent qui est signalé à l'agent, pour que la réaction correspondante puisse être exécutée. Les notifications sont le seul moyen que les agents peuvent utiliser pour communiquer. Les notifications sont amenées jusqu'aux agents via un bus de message appelé `Channel`. Le `Channel` est piloté par le moteur AAA qui garantit l'atomicité, l'ordre des transmissions de notifications et le caractère transactionnel des réactions. L'ensemble constitué par le `Channel` et le moteur AAA est appelé un serveur d'agent.

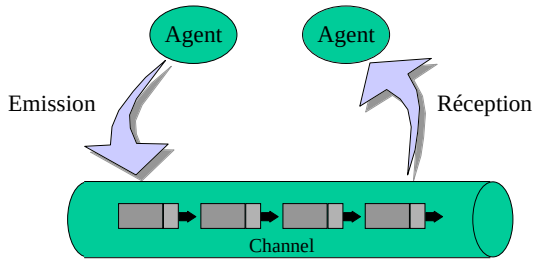

Figure 84. Architecture du serveur d'agents AAA

Ce mécanisme fonctionne dans un environnement distribué. Plusieurs serveurs d'agents peuvent s'exécuter sur différentes machines. Les serveurs d'agents gèrent la création, et la destruction des

agents sur chaque site et ils permettent l'exécution des réactions des agents en fonctions des notifications reçues. Chaque agent est attaché à un seul serveur d'agents, seules les notifications sont transmises entre sites.

L'objectif de cette expérimentation est de rendre configurable l'utilisation des agents tout en intégrant dans l'environnement Olan la construction d'applications fondées sur l'environnement AAA. Il faut noter que l'environnement AAA est écrit en Java et que les Agents sont des objets Java héritant de certaines classes de base assurant un comportement minimal commun compatible avec les mécanismes du serveur d'agents. Nous avons donc ajouté à l'environnement Olan un nouveau type de modules primitifs permettant de décrire des Agents en tenant compte de leurs règles de programmation. Nous avons aussi étendu le support de configuration pour permettre de configurer des agents dans l'environnement AAA par le biais de la machine à configuration Olan.

II.3. Description OCL

L'application Mailer a pour particularité de ne pas être intrinsèquement répartie. Il n'y a pas de véritable raison pour repartir les différents composants ou agents qui la forment. L'intérêt de cette application est donc d'intégrer l'utilisation du support à agents AAA et donc de montrer la possibilité d'extension de l'environnement Olan pour prendre en compte différents modèles de programmation et différents mécanismes de communication.

Deux grandes classes d'entités existent : les agents et les composants. Nous ne détaillerons pas les fonctions et la réalisation des différents composants car ce processus est similaire à celui de l'application CoopScan. Nous nous focaliserons sur la description des agents.

Les Agents

Les agents AAA ont un modèle de programmation particulier. Ils communiquent avec d'autres objets par l'envoi de notification au bus de message. Le code associé à un agent est activé lorsque celui-ci reçoit une notification en provenance d'un autre agent ou d'un autre composant. Au niveau du langage OCL, les agents doivent posséder une interface spécifique construite sur le modèle suivant :

- Ils possèdent un unique service de type react dont le nom est obligatoirement React. Les paramètres de ce service sont des notifications. Les notifications correspondent généralement à des structures, mais tout type nommé OCL (i.e. défini par un typedef) est utilisable.
- Ils possèdent ensuite des services de type notify, qui correspondent à l'envoi d'une ou plusieurs notifications vers un autre agent ou un composant.
- Ils peuvent contenir au sein de la signature des services des unions de notifications, correspondant à un ensemble de notifications autorisées.


```
 Interface AgentItf {
        react React( in NotifUnion)
        notify N1(in NotifUnion1);
        ...
 }
 enum NotifNames { Name1, Name2, ...}; // Définition des noms de notifications
accepté par le port react
 // Définition de l'ensemble des notifications acceptés par le port React. C'est
une union accessible par le selecteur NotifNames.
 typedef union NotifUnion
        switch(NotifNames)
 {
        CASE Name1:
        // définition du type associé à cette notification
        CASE Name2:
        ....
 }
```

Figure 85. Description OCL de l'interface d'un agent

Les agents sont alors considérés comme des composants primitifs OCL composés uniquement d'un seul module de type 'AAA'. C'est cette dernière propriété qui permet au compilateur OCL de distinguer un agent de tout autre composant primitif. Il peut ainsi générer automatiquement un squelette de la classe Java qui correspond à l'agent. Cette classe est alors conforme au modèle de programmation AAA, assurant ainsi un fonctionnement correct avec le moteur AAA.

Trois types d'agents existent dans l'application Mailer. L'agent de filtre reçoit une notification de nouveau message arrivé, l'analyse pour décider s'il répond aux critères de filtres qui lui sont associés. L'agent de tri reçoit une notification de nouveau message pour demander son rangement dans un répertoire de l'utilisateur, répertoire dont le nom est contenu dans un de ses attributs. Enfin, l'agent de redirection reçoit une notification de nouveau message pour l'envoyer vers un autre utilisateur. Etudions le code OCL de définition de l'agent de Filtre. Son interface, ainsi que les différents types de notification sont décrits dans la Figure 87. La représentation du composant de filtre est la suivante.

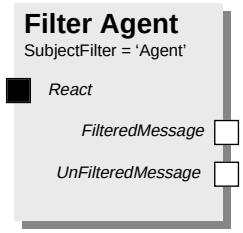

Figure 86. Représentation de l'agent de Filtre

On peut remarquer que l'agent réagit à une notification de type `MailNotification`. Celle-ci contient le contenu du message qui doit être traité. Le message est donc reçu dans sa globalité par l'agent de filtre qui ensuite le retransmet à l'extérieur en envoyant une notification de même type soit sur le service `FilteredMessage` ou sur le service `UnFilteredMessage`. La syntaxe complète en OCL est la suivante:

```
 // structure du message
 struct mailStruct {
        string Message_id;      // identificateur du message permettant au
lecteur de mail d'acceder aux differents messages par le biaus de liens
hypertextes
        short num_mail;         // numeros du mail
        string Date;            // champ date du message
        string From;            // champ From du message
        string To;              // champ To du message
        string Subject;         // champ Subject du message
        string Cc;              // champ Cc du message
        string Sender;          // champ Sender du message
        string Content_type;    // type du contenu (champ content-type du
message)
        string Boundary;        // si c'est un mail en plusieurs parties, cette
chaine est non vide et contient la ligne separateur
        string Otherfields;     // Header restant apres avoir enleve les champs
date,from,to,subject,cc,sender,received et content-length
        string Firstline;       // Premiere ligne du message (From ..)

        string intermediaire;  // lignes intermediaires entre debut du multipart
et entete
        sequence <subsetStruct> liste_blocs; // contient les differentes parties
du mail (1 seul si non multipart)
 } ;

 enum mailNotificationSelectors {GetMessage};
 typedef union
 SmailNotification switch(mailNotificationSelectors)
 {
  case GetMessage:
         mailStruct message;
 } mailNotification;

 interface FilterAgentItf {
 /* attributs */
        attribute   string FromFilter;
        attribute   string ToFilter;
        attribute   string SubjectFilter;
        attribute   string CCFilter;
        attribute   string BodyFilter;

        react React(in mailNotification message);
        notify FilteredMessage(in mailNotification message);
        notify UnFilteredMessage(in mailNotification message);
 };
```

Figure 87. Description OCL de l'interface de l'agent de filtre

```
 /************************ agent de filtrage ********************/
 module "aaa" FilteringAgent_Mod : FilteringAgent_Itf
 {
         java: "/usr/local/java/bin/java";
         classPath: "${CLASSPATH}";
 };

 primitive implementation "AAA" FilteringAgent_Imp : FilteringAgent_Itf
         uses FilteringAgent_Mod
 {
 attribute FilteringAgent_Mod.FromFilter = FromFilter;
 attribute FilteringAgent_Mod.ToFilter = ToFilter;
 attribute FilteringAgent_Mod.CCFilter = CCFilter;
 attribute FilteringAgent_Mod.SubjectFilter = SubjectFilter;
 attribute FilteringAgent_Mod.BodyFilter = BodyFilter;

         React() => FilteringAgent_Mod.React();
         FilteringAgent_Mod.FilteredMessage() => FilteredMessage();
         FilteringAgent_Mod.UnFilteredMessage() => UnFilteredMessage();

 };
 component FilteringAgent {
         implementation FilteringAgent_Imp;
 };
```

Figure 88. Définition OCL de l'Agent de Filtrage

Intégration des logiciels de messagerie

Si on se réfère à la Figure 83, on s'aperçoit que les agents ne sont pas les seuls composants existants. Nous avons besoin de divers composants dont celui qui intègre le lecteur de courrier électronique, celui pour l'accès au mécanisme de réception et d'envoi de message,... Nous avons déjà décrit le rôle de chacun de ces composants dans la section précédente.

La caractéristique commune de tous ces composants est qu'ils ne sont pas mis en place par l'utilisateur de l'application. Ce sont essentiellement les briques de base de l'application à base d'agents. Ils permettent de faire transiter les messages par une série d'agents. Tous ces composants sont regroupés dans un composite qui offre au niveau de son interface les points de branchements pour les agents.

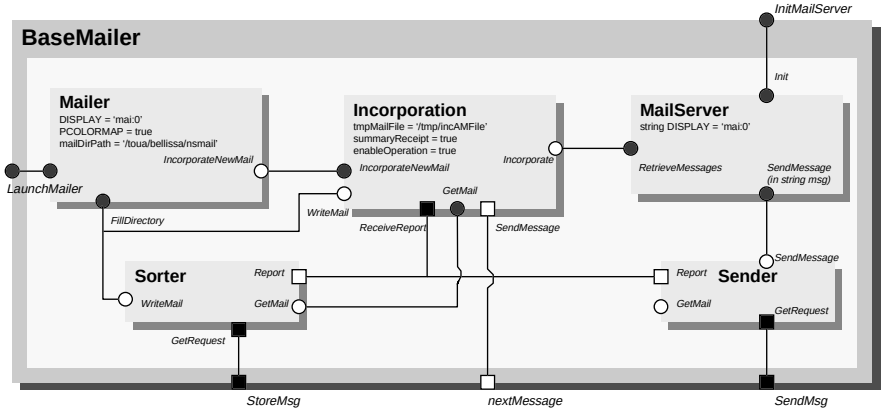

Figure 89. Intégration de l'existant dans l'application Mailer

En particulier, il permet d'envoyer une notification `nextMessage` contenant un message qui peut alors transiter par la chaîne de filtrage. Il offre aussi deux services, l'un pour le rangement du message (`StoreMsg`) et l'autre pour l'envoi d'un message vers un autre destinataire (`SendMsg`).

L'implémentation de ce composite contient des composants dont certains possèdent des attributs configurables. Ils permettent notamment à l'installateur de l'application de positionner les valeurs propres à chaque utilisateur, comme le répertoire de stockage des messages de l'utilisateur dans le composant du lecteur de message, la création d'un rapport d'opération au niveau du composant d'incorporation, etc.

Configurer l'application

La configuration de l'application consiste à interconnecter des agents avec le composite contenant les logiciels existants. Ce dernier permet l'envoi de chaque message de la boîte aux lettres de l'utilisateur vers des agents, et il permet aussi à des agents de piloter le stockage de messages dans des répertoires ou l'envoi d'un nouveau courrier vers un autre utilisateur.

Configurer l'application consiste donc à spécifier la chaîne de traitement des messages en ajoutant des agents de filtre, de rangements ou de réémission. Ajouter un agent signifie déclarer l'instance et positionner les attributs de l'agent, puis mettre les interconnexions en place.

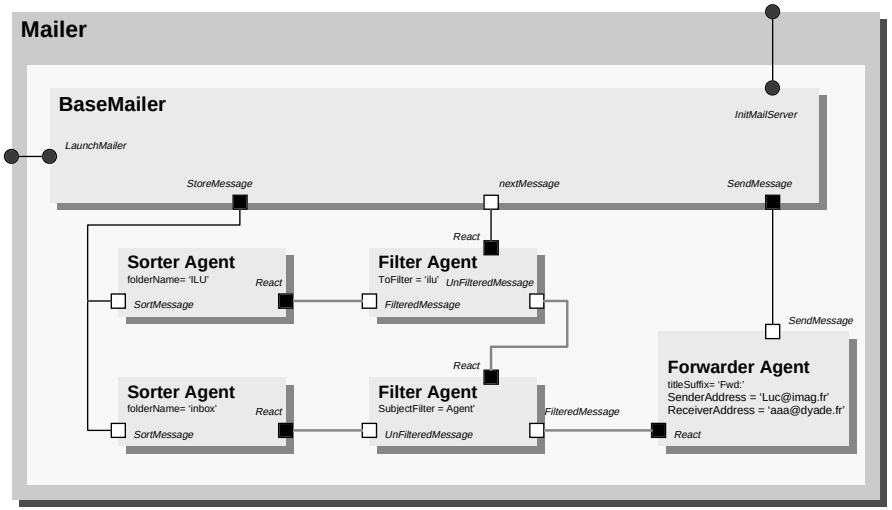

Figure 90. Configuration de l'application Mailer

L'exemple illustré sur la Figure 90 montre une chaîne de filtrage dont le fonctionnement est le suivant : un message transite par le premier agent de filtre qui vérifie si le destinataire contient la chaîne "`ilu`". Dans un tel cas, le message est transmis à un agent de rangement dans le répertoire de nom `ILU`, agent qui transmet le message et le nom du répertoire au composite `BaseApplication`. Si le message ne vérifie pas le critère du premier agent de filtre, il passe dans un second agent de filtre qui vérifie si le champ sujet contient la chaîne "`Agent`". Dans le cas négatif, le message est rangé dans le répertoire `inbox`, en demandant l'opération à un agent de rangement. Dans le cas positif où il vérifie le critère de filtre, le message est réémis à une liste de diffusion `aaa@dyade.fr` avec comme entête de message, le sujet du message réémis précédé de la chaîne de caractère "`Fwd`:".

Le programme OCL qui découle de cet exemple est le suivant :


```
implementation MyAppl
        use ...
{
        existant = instance BaseMailerItf;
        agent1 = instance FilterAgentItf( ToFilter = "ilu")
        agent2 = instance FilterAgentItf(SubjectFilter = "Agent");
        agent3 = instance SorterAgentItf(folderName = "ILU");
        agent4 = instance SorterAgentItf(folderName = "inbox");
        agent5 = instance ForwarderAgentItf( titleSuffix = "Fwd:",
                                SenderAddress = "Luc.Bellissard@inrialpes.fr",
                                ReceiverAddres = "aaa@dyade.fr");

        existant.nextMessage() => agent1.React();
        agent1.FilteredMessage() => agent3.React() using aaa();
        agent1.UnFilteredMessage() => agent2.React() using aaa();
        agent2.FilteredMessage() => agent5.React() using aaa();
        agent2.UnfilteredMessage() => agent4.React() using aaa();
        agent3.SortMessage() => existant.SortMessage();
        agent4.SortMessage() => existant.SortMessage();
        agent5.SendMessage() => existant.SendMessage();
}
```

Figure 91. Configuration OCL du Mailer

Notons que la communication entre agents utilise un connecteur particulier, le connecteur "`aaa`", qui permet d'utiliser le mécanisme du `Channel` d'un serveur d'agents en lieu et place du modèle de connecteurs présenté dans le chapitre précédent.

II.4. Avantages et Inconvénients de l'Utilisation d'Olan

Le principal intérêt de l'utilisation de l'environnement Olan est de permettre de modifier ou faire évoluer l'application aisément. En effet, dans notre cas de figure, l'ajout ou le retrait d'agents de traitement de message intervient à tout moment lorsque l'utilisateur en ressent le besoin. L'environnement Olan permet ainsi d'ajouter des fonctions à l'application sans aucune programmation en soit à envisager. Il suffit de réutiliser des agents existants, de configurer leurs attributs et de mettre les interconnexions en place. Evidemment la démarche de permettre à l'utilisateur d'ajouter ses propres fonctions dans un logiciel n'est pas nouvelle. Prenons la dernière version du logiciel de navigation Netscape 4.0 : elle permet entre autre de définir des filtres sur les messages électroniques pour les ranger dans des répertoires, comme les fonctions que nous proposons dans notre application. La définition de ces fonctions utilisateurs se fait à travers d'interface graphique classique (boutons, menus déroulants,...) et ne permet pas d'effectuer des traitements autres que ceux prédéfinis. Ici, avec l'environnement Olan, deux intérêts nous semblent primordiaux: les possibilités illimitées de combinaisons de composants au travers d'un formalisme graphique facile à prendre en main, et la possibilité de fabriquer de nouveaux agents sans entrer dans le code du reste de l'application.

Le problème qui est en suspens concerne l'absence actuelle de l'outil graphique d'assemblage de composants, ce qui impose de programmer la description OCL à la main, en particulier la définition des types et des notifications d'agents. Le travail n'est donc pas aussi facilement manipulable par un usager non averti que ce que nous avons annoncé initialement. Il faut un outil graphique qui cache toute la complexité du langage OCL.

II.5. Réalisation

La réalisation de l'application repose sur la version du support de configuration présentée dans le chapitre précédent. Les composants primitifs qui intègrent les logiciels de messagerie comme le serveur de courrier électronique, le lecteur de message, ont été réalisés en Python en C. Tous les agents sont écrits en Java et fonctionnent avec un serveur d'agents AAA, lui-même écrit en Java. L'interopérabilité entre agents écrits en Java et composants C ou Python fonctionne.

III.Evaluation

Les deux applications que nous venons de présenter possèdent des caractéristiques relativement différentes. La première est une application multi-utilisateurs, où les composants sont répartis sur un ensemble de sites au fur et a mesure des arrivées de nouveaux utilisateurs. La seconde application ne fait pas usage de la répartition mais démontre l'intérêt de la représentation de l'architecture pour faire évoluer une application sans avoir à programmer du logiciel. Il existe d'autres exemples d'applications que nous avons partiellement développées mais les deux présentées ici sont un échantillon représentatif de certaines propriétés de l'environnement Olan.

Le principal intérêt de l'utilisation d'Olan pour ces applications est la description explicite, indépendante du code de mise en œuvre, de l'architecture de l'application. Elle permet de connaître clairement tous les composants logiciels nécessaires au fonctionnement, ainsi que les modes de communication requis entre ces composants logiciels. Les indications de placement ajoutent à l'architecture des informations sur le type de machine et la localisation des composants les uns par rapport aux autres. Les types de connecteurs donnent des informations sur le modèle et le mécanisme propre à chaque communication entre composants.

La définition de l'architecture est basée sur une structuration hiérarchique des composants. Ceci permet de réutiliser des composants dans diverses applications sans connaître leur contenu. Par exemple, le composant `IntegratedMailer` de l'application de gestion du courrier électronique permet de réutiliser des composants de lecture de message ou d'envoi de message sans savoir ce qu'il y a à l'intérieur. L'utilisateur peut alors se concentrer sur les composants agents dont il a besoin pour mettre en place sa chaîne de traitement des messages.

Enfin, le dernier point intéressant lié à l'environnement Olan et au langage OCL est de fournir une vision structurelle de l'application sans oublier des aspects dynamiques de son comportement. Les aspects dynamiques auxquels on peut se référer sont la création et suppression de composant pilotées par l'application pendant l'exécution. Le concept de collection permet de représenter des ensembles de composants dont le dénominateur commun est la même interface. Cette facilité nécessite alors de pouvoir aussi s'adresser aux instances de composants créées lors de l'exécution de manière dynamique, i.e. pilotée par l'application. C'est pour cela que la désignation associative a été introduite afin de permettre de s'adresser dynamiquement à un composant plutôt qu'un autre. Dans l'application CoopScan, ceci est souvent utilisé pour spécifier les communications entre composants contenus dans des collections.

On s'aperçoit ainsi que la définition d'une architecture avec OCL intègre des aspects statiques, i.e. des instantiations ou des communications connues a priori, lors de la construction de l'application, mais aussi des aspects dynamiques où le schéma d'instantiation des composants ou certaines parties

des communications sont potentiellement définies, mais ont effectivement lieu lors de l'exécution à la seule condition que les composants de l'application le décident. Nous appelons l'ensemble des liens ainsi décrits et spécifiés, les "liens structurels" dans une application. Néanmoins, il existe un autre type de lien qui apparaît souvent lors de la programmation d'applications : les "liens conjoncturels". Ces liens existent lorsqu'un composant communique avec un autre composant en lui passant en paramètre un identifiant ou une référence vers un autre composant, afin que le récepteur puisse le manipuler dans son implémentation. Or ceci est complètement impossible à faire en OCL pour deux raisons. La première est qu'il n'est pas possible de spécifier des types de composants comme paramètre de service avec OCL. La seconde raison est que, si la première limitation était supprimée, il faudrait encore que la mise en œuvre d'un composant, i.e. le code logiciel qu'il intègre, soit capable d'utiliser et de manipuler des références vers des composants. Or un composant peut être réparti ou non, en fonction des spécifications, mais rien n'est fourni au programmeur pour manipuler des références universelles, i.e. des références à des composants locaux ou distants. Il faudrait dans un tel cas de figure que le support de configuration Olan fasse office de support d'exécution avec la prise en compte transparente de références de composants au sein des divers langages de programmation que l'on intègre. C'est quelque chose que nous avons toujours refusé, car l'intérêt de notre approche est aussi d'utiliser divers mécanismes de communication entre composants sans imposer un support d'exécution réparti particulier dont les propriétés soient conformes au modèle d'architecture Olan. L'environnement Olan n'apporte donc pas de réponse au problème des liens conjoncturels.

Le second problème de l'environnement concerne les performances du support d'exécution qui peut dépendre de la granularité du logiciel intégré dans les composants primitifs. En effet, dans la plupart des cas, un composant de forte taille communique moins souvent avec l'extérieur qu'un simple objet. Les performances des communications, dégradées par le passage des multiples couches de composants et de connecteurs du support de configuration Olan, sont acceptables si les communications entre composants sont minoritaires, en terme de temps global d'exécution, par rapport au temps utilisé par le logiciel contenu dans les primitifs. Dans nos expérimentations, la granularité du logiciel intégré est variée : un composant représente des applications complètes ou des bouts de code importants, dans le cas de CoopScan ou de Netscape, par exemple. Il peut aussi contenir une simple classe comme dans le cas des agents. L'utilisation de Olan dans le cadre de CoopScan s'est avéré acceptable en terme de temps de réponse pour l'utilisateur. Dans le cas de l'application Mailer, les multiples couches pour faire communiquer les composants avec les agents (passage entre Java et les composants Python,...) pénalisent l'utilisateur final. La demande d'incorporation de messages qui prend quelques millisecondes avec l'application Netscape seule, requiert quelques secondes avec les agents et le support Olan. Nous n'avons pas de mesures exactes, ni de comparaisons chiffrées, seule une évaluation qualitative du confort d'utilisation nous permet d'affirmer ceci.

Chapitre 6
Conclusion

I. Objectif et Démarche de Travail

L'objectif de ce travail est de brosser un aperçu de différents outils de construction et de configuration d'applications répartis. La construction d'application est l'action d'intégrer du logiciel (existant ou non) dans des composants pour ensuite les assembler afin de former une application. La configuration consiste à identifier les composants logiciels nécessaires à l'application, à les paramétrer en vue du fonctionnement désiré et à décrire le placement de l'application sur un réseau de machines interconnectées. Ce travail a permis d'identifier un certain nombre d'outils pour effectuer l'une ou l'autre, voire ces deux opérations. L'action de construction est traditionnellement une tâche incombant aux langages de programmations qui généralement fournissent des abstractions pour définir un composant logiciel et pour programmer les communications entre ces composants. L'application des principes issus de tels langages a permis l'émergence de mécanismes de communication répartie, tels les RPC, CORBA ou DCOM, dont le but est de simplifier la tâche du programmeur dans la gestion du facteur de répartition. En particulier, toute la gestion de l'accès aux interfaces de communication réseau ainsi que le problème de l'hétérogénéité des données ou des plates-formes est pris en charge par ces mécanismes. Toutefois, cette approche langage de programmation associée à des mécanismes de communication à distance n'apporte que peu de solutions aux problèmes de la configuration, tels la facilité d'assembler des composants hétérogènes sans modifier le code de l'application, la vision de l'architecture logicielle de l'application ou la paramétrisation du modèle de communication. De plus, de tels mécanismes demandent une compétence certaine de la part du concepteur et programmeur d'applications réparties.

Ce travail a débuté par l'étude de ces mécanismes ainsi que celle d'une classe de langage d'assemblage ou de composition. Ces langages - langages d'interconnexion de modules (MIL), langage de configuration ou langage de description d'architecture (ADL) - ne remplacent pas les langages de programmation mais permettent de spécifier l'intégration de composants logiciels pour former des applications, bien qu'ils n'adressent pas tout le problème de la répartition. Ils permettent de décrire les composants logiciels requis sans fournir d'outil pour leur programmation. Ils permettent de spécifier les interconnexions entre composants, i.e. les communications qui potentiellement auront lieu pendant l'exécution. Ils peuvent aussi générer l'image exécutable de l'application en prenant en compte toutes les spécifications pour produire automatiquement du code lié au facteur de répartition. Le programmeur peut ainsi se concentrer sur l'algorithmique des composants, l'architecte de l'application faisant acte d'assembler les composants selon ses impératifs.

Nous avons abordé l'étude de ces langages et outils selon quatre angles de vision :

- les capacités d'intégration de logiciels existants et la gestion de leur hétérogénéité,
- les capacités à définir l'architecture de l'application en terme de composants logiciels utilisés et de spécification de la communication entre ces composants,

- la capacité à spécifier le plus simplement et souplement possible la répartition des composants,
- la capacité à produire une image exécutable de l'application et la facilité d'installer et de déployer automatiquement l'application sur un système distribué.

Nous avons ensuite porté notre effort sur la définition d'un langage de définition d'architecture, OCL, dédié à l'intégration de logiciels existants pour former des applications réparties. Ce langage permet notamment de décrire une architecture proche de ce qui existe réellement lors de l'exécution afin de maintenir un lien fort entre la vision du concepteur et la réalité de l'exécution. Pour cela, OCL permet de décrire certains comportements dynamiques de l'application comme l'instantiation de composants pendant le déroulement de l'application ou la communication conditionnelle entre composants en fonction de propriétés ou d'attributs qui peuvent être modifiés par la mise en œuvre des composants. OCL s'efforce aussi de proposer des solutions dans le domaine de l'intégration de logiciels existants, en permettant d'encapsuler du code sous forme binaire ou non programmé en langage C, Python ou Java à l'intérieur des composants. Un système de définition d'interface couplé à différentes projections vers ces langages facilitent le travail du programmeur. Ces composants peuvent ensuite être assemblés en les interconnectant, peu importe le type de logiciels qu'ils contiennent. La notion d'interconnexion permet alors de définir le modèle de communication entre ces composants ainsi que d'éventuelles adaptations de paramètres entre interfaces. Le mécanisme utilisé dépend alors du modèle de communication désiré mais aussi de la localisation des composants. La définition de cette localisation se fait aussi dans le langage OCL, qui permet non seulement de définir le nom du site hôte de manière absolu mais aussi de manière indirecte en indiquant des propriétés requises du site qui hébergera l'exécution du composant.

Ce langage a donné lieu au développement d'un compilateur qui produit le code exécutable nécessaire à l'intégration des logiciels, aux interconnexions mais aussi au placement de l'application sur des machines interconnectées. L'action de placer les composants et d'installer des applications en vue de leur exécution a nécessité le développement d'un support réparti dont le rôle est d'interpréter des demandes de création ou d'interconnexion de composants issus de la définition de l'architecture en OCL. Il qui met en œuvre un algorithme de sélection des sites hôtes et effectue la création des différentes structures requises à l'exécution de l'application. Parmi ces structures, nous avons identifié le code d'accès aux logiciels intégrés mais aussi le code correspondant d'utilisation des mécanismes de communication spécifiés dans les interconnexions.

La phase finale de ce travail est l'utilisation de l'environnement Olan pour construire et configurer des applications d'exemples. Nous avons détaillé dans ce manuscrit l'architecture d'une application d'édition coopérative et d'une application de filtrage de courrier électronique qui ont été construites, installées et déployées avec les outils Olan.

II. Evaluation

Le principal apport de ce travail est la définition d'un environnement complet pour la construction, la configuration et le déploiement d'applications en environnement réparti. L'environnement se scinde en deux classes d'outils : des outils de type génie logiciel avec un modèle de construction d'application associés au langage de configuration OCL et un compilateur, et des outils de type "système" qui permettent l'installation et le déploiement d'applications avec le support de configuration Olan. Au

niveau des outils génie logiciel, le travail décrit dans ce document est surtout axé autour du modèle de construction et du langage de configuration OCL. Les contributions d'OCL portent sur les axes suivants :

- intégration de logiciels hétérogènes,
- description de l'architecture d'une application avec la possibilité d'inclure des aspects dynamiques du comportement comme le schéma d'instantiation des composants avec les collections ou la désignation associative de composants lors des communications,
- placement des composants de l'application dans des contextes (caractérisation conjointe du site et de l'utilisateur pour qui le composant s'exécute), et
- génération du code suffisant à l'exécution de l'application sur un ensemble de sites.

Toutefois, nous avons identifié certaines limitations de modélisation d'applications.

- L'architecture de l'application contient essentiellement les informations sur la structure de l'application. Les interconnexions entre composants s'appuient sur cette structure et nous avons volontairement souhaité que cette structure ne soit pas connue par la mise en œuvre des composants. Il est ainsi impossible qu'un composant manipule dans son code primitif un autre composant ; il doit obligatoirement s'adresser à son interface. En conséquence, on ne peut transmettre en paramètre des références de composants que nous appelons des liens conjoncturels entre composants. Ceci demande une certaine gymnastique au concepteur d'application pour s'adapter à ce mode de conception qui ne peut définir que les liens appelés liens structurels.
- La prise en compte des exceptions n'est pas définie, ce qui dans un contexte réparti est problématique, car les sources d'erreurs sont multiples et aléatoires.
- Une application dans le modèle Olan est une hiérarchie de composants qui peuvent être utilisés dans de multiples contextes. Néanmoins, cette réutilisation se situe au niveau du code des composants : il n'est pas possible de réutiliser des composants ou des applications en cours d'exécution, sur un site donné, avec une configuration particulière. Il n'y a pas de système de désignation d'entités en cours d'exécution avec lesquelles une hiérarchie de composants puisse communiquer. Un programme OCL produit une application qui fonctionne seule, indépendamment de toutes autres applications en cours d'exécution sur des sites du système distribué.

Au niveau du support de configuration, nous avons aussi identifié des problèmes essentiellement liés au mode d'intégration des composants hétérogènes. Tout accès vers ou depuis du logiciel intégré (composant primitif) est transformé dans un modèle d'exécution homogène imposé par le support de configuration. Il existe donc de multiples couches logiciels entre des composants : un pour la transformation du modèle d'exécution, un pour la transformation de paramètres, un ou plusieurs pour la communication entre les composants : ceci induit des performances dégradées. Il est de plus dommage de ne pas utiliser au mieux les propriétés des composants primitifs pour éviter le passage par ces multiples couches ; si deux composants contenant des objets CORBA veulent communiquer, ils ne peuvent actuellement le faire directement car ils doivent se conformer au modèle Olan.

Au niveau des applications, les expérimentations entreprises ont montré que l'architecte qui se conforme au mode de description Olan des applications gagne en souplesse, en capacité d'évolution et de production de multiples variantes de son application. Par contre, il existe un compromis à trouver pour la granularité du code intégré dans ces composants. Le modèle Olan autorise l'intégration d'application complète comme de simple classe. Or chaque communication entre composant passe par la chaîne de communication mise en place par le support de configuration, en particulier les connecteurs. Il y a donc un coût inhérent au caractère configurable des communications entre composants qui ne peut pas vraiment se justifier si des composants sur un même site communiquent très fréquemment ensembles. Or nous n'avons pas de mesure sur la taille idéale du composant ou le surcoût des communications pour donner des lignes générales de conception d'applications avec Olan.

Enfin, il existe des classes d'applications auxquelles nous n'avons pas vraiment regarder l'adéquation du modèle Olan pour les programmer. En particulier, il nous semble difficile d'utiliser le langage OCL pour prendre en compte des applications utilisant du code ou des objets mobiles car justement l'intérêt de OCL est concerne la structure et les liens structurels. Nous n'avons pas non plus de réponses pour des applications de type transactionnel car rien ne permet de contrôler le comportement de la mise en œuvre d'un composant, ceci est à la charge du concepteur de l'application en espérant que les informations que OCL fournis au niveau de l'interface soient suffisantes.

III. Perspectives

Ce travail a fait l'objet d'un développement conséquent mené avec les membres de l'équipe Olan. Toutefois, certains points ne sont pas complètement achevés, en particulier la désignation associative et la génération des clauses de transformations de paramètres. Un travail sur la désignation associative a déjà été mené dans un autre environnement et l'incorporation dans l'environnement Olan est en cours. De plus, un certain nombre des problèmes que nous avons rencontrés font parties des axes de travail en cours. En particulier, nous étudions la possibilité de prendre en compte les exceptions dans le modèle de construction Olan en évaluant leurs impacts sur les concepts existants, comme les clauses de transformations de paramètres, les clauses de désignation associatives. Nous étudions aussi la possibilité de résoudre le problème des liens conjoncturels que nous avons abordé en fin de chapitre 5. Pour cela, nous essayons de définir les conditions de passages de références de composants en paramètres des services. Enfin, le dernier axe sur le langage OCL et le modèle de construction Olan consiste à prendre en compte l'existence de plusieurs applications au sein du langage OCL et de permettre la coopération entre celles-ci. Ceci permettra de décrire par la même l'architecture complète d'un système distribué en montrant clairement toutes les applications existantes qui s'exécutent.

Du point de vue réalisation, le travail majeur qui vient de débuter consiste à adapter le modèle de construction Olan pour réaliser un environnement complet de développement d'applications à base d'Agents dans le cadre du GIE Dyade (groupement d'intérêt économique entre Bull et l'INRIA). Le rôle du GIE est de transférer des résultats de recherche au sein des produits Bull. Les agents que nous avons définis dans le cadre de l'action AAA (c.f. section *II.2. L'Environnement AAA (Agent Anytime* Anywhere), p.122) sont des entités logicielles considérées comme des valeurs ajoutées à des applications existantes. L'intérêt des agents est de rendre ces applications plus souples et facilement évolutives, avec en prime un moyen de communiquer à distance si elles ne sont pas réparties. Le transfert de technologie issue de l'expérience acquise autour de la configuration d'applications réparties ainsi que les outils que nous avons présenté dans ce manuscrit, consiste à simplifier le

modèle Olan et le langage OCL pour les besoins spécifiques des applications à base d'agents, de fournir un outil visuel de configuration de ces applications et d'intégrer dans le support d'exécution d'agents AAA des fonctions de configuration, d'installation et de déploiement. Ce développement se fait en utilisant comme application de test celle de traitement du courrier électronique et des applications issues des lignes de produits Bull.

Trois pistes de recherche nous semblent intéressantes de poursuivre autour de la construction et de la configuration d'applications réparties :

La reconfiguration dynamique d'applications, ou comment modifier en cours d'exécution des éléments de la configuration, comme la mise en œuvre d'un composant, des interconnexions, des valeurs d'attributs ou le placement des composants. C'est un thème de recherche en plein essor où le couplage entre un langage d'architecture et un support de reconfiguration semble être une piste prometteuse.

L'intégration de logiciels existants qui se présentent sous des formes extrêmement variées : un binaire, une application utilisant système de fenêtres particulier (e.g. X-Windows,...), un serveur CORBA, un composant serveur DCOM, un serveur HTTP,... Nous avons insisté dans cette étude sur l'intégration de langages de programmation hétérogènes qui pouvaient induire un modèle d'exécution particulier (cas de l'utilisation d'un interpréteur Java). Ceci nous pousse à croire à la possibilité de remonter de manière plus générale ces types de logiciels au niveau du langage d'architecture (par un système de type à la UniCon[Sha95] par exemple) pour permettre une intégration de multiples types de logiciels et ainsi aider le concepteur d'application pour le travail d'encapsulation.

L'utilisation d'un calcul pour la vérification de la validité d'une architecture et des interconnexions qu'elle contient. Cet effort ne porte pas sur la vérification de la conformité des types au sein des interfaces mais sur la compatibilité des modèles d'exécution des composants (principalement des modèles actifs ou passifs, synchrones ou asynchrones). Des travaux préliminaires ont été déjà conduits au sein du projet Olan qui nous poussent à espérer des résultats intéressants et des outils de type simulateur d'exécution à la Rapide[Luc95].

Bibliographie

[All94] Allen R., Garlan D., "Formal Connectors", CMU Tech. Report CMU-CS-94-115, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA, March 1994.

[ATT90] AT&T, "System V Release 4 Programmer's Guide: Networking Interfaces", Unix Press-Prentice Hall, 1990

[Bal91] Balter R. et al., "Architecture and Implementation of Guide, an Object Oriented Distributed System", *Computing Systems*, vol.4(N.1), Hiver 1991, pp.31-67

[Bel96] Bellissard L., Ben Atallah S., Boyer F., Riveill M., "Distributed Application Configuration", *in Proc. of the 16th IEEE Intn'l Conference on Distributed Computing Systems (ICDCS'96)*, Hong Kong, April 1996

[Ben97] Ben Atallah S., "Architectures Système pour la Construction et l'Exécution de Collecticiels", Thèse de Doctorat en Informatique, Université de Savoie, Juin 1997.

[Bir84] Birell A.D., Nelson B.J., "Implementing Remote Procedure Calls", *ACM Trans. On Computing Systems*, vol.2(No.1), Feb. 1984, pp. 39-59

[Bul94] Bull Open Software Systems, "OODE : une Plate-Forme Objets pour les Applications Réparties", in AFCET, Ed. AFCET, Paris, France, Nov. 1994

[Cal90] Callahan J. R., Purtilo J. M., "A Packaging System for Heterogeneous Execution Environments.", *Technical Report* (CS-TR-2542), Dept. of Computer Science, Univ. of Maryland, October 1990

[DeR76] DeRemer F., Kron H.H., "Programming-in-the-large vs. Programming-in-the-small", *IEEE Trans. Software Engineering*, vol.SE-2(N.2), June 1976, pp. 114-121.

[Fel97] Felber P., Guerraoui R., Schiper A., "A CORBA Object Group Service", *ECOOP Workshop on Corba : Implementation , Use and Evaluation*, June 1997, *to be published in IEEE Computer Science Press Book*.

[Fla96] Flanagan D., "Java in a Nutshell, a Desktop Quick Reference for Java Programmers", O'Reilly & Associates Ed., 103 Morris Street, Suite A, Sebastopol, CA 95472 USA, February 1996, ISBN: 1-56592-193-6, (http://www.ora.com/info/java).

[Gar95] Garlan D., "First International Workshop on Architectures for Software Systems: Workshop Summary", ACM SIGSOFT Software Engineering Notes, SEN 1995.

[Gar96] Gardarin G, Gardarin O., "Le Client Serveur", Ed. Eyrolles, Paris, 1996

[Gri97] Grimes R., "Professional DCOM Programming", Wrox Press Ltd., 30 Lincoln Road, Olton, Birmingham, B27 6PA, UK, 1997, ISBN 1-861000-60-X

[Hoa85] Hoare C.A.R., "Communicating Sequential Processes", Prentice Hall, 1985

[Iss96] Issarny V., Bidan C., "Aster: a Framework for Sound Customization of Distributed runtime Systems", *Proc. Of the 16th IEEE Intn'l Conference on Distributed Computing Systems (ICDCS'96)*, Hong Kong, April 1996.

[Iss97] Issarny V., Bidan C., Saridakis T., "Aster: Un système de configuration distribuée au-dessus de CORBA", L'objet, 1997.

[Jan96] Janssen B. Spreitzer M.., "ILU 2.0alpha10 Reference Manual", Xerox Corporation Palo Alto CA, Feb. 1996, (http:\\ftp.parc.xerox.com\ilu)

[Kra89a] Kramer J., Magee J., Sloman M., "Constructing Distributed Systems in CONIC", *IEEE Trans. Software Engineering*, vol.SE-15(N.6), June 1989, pp.663-675.

[Kra89b] Kramer J., Magee J., Ng K., "Graphical Configuration Programming", *IEEE Computer*, vol.22(N.10), Oct. 1989, pp. 53-65

[Kra90] Krakowiak S. and al., "Design and Implementation of an Object Oriented, Strongly Typed Language for Distributed Applications", Journal of Object Oriented Programming (JOOP), vol.3(N.3) Sept-Oct 1990.

[Lam87] Lamb D. A., "IDL: Sharing intermediate Representations", ACM Transaction on Programming Languages and Systems, vol.9(N.3), July 1987, pp.297-318.

[Len96] Lenormand E., "Communication par Evénements dans les Modèles à Objets", Thèse de Doctorat en Informatique, Université Joseph Fourier, Grenoble France, Nov. 1996.

[Luc95] Luckham D. C., Vera J., "An Event-Based Architecture Definition Language", *IEEE Trans. Software Engineering*, vol.SE-21(N.9), September 1995, pp.717-734.

[Mag93] Magee J., Dulay N., Kramer J., "Structuring Parallel and Distributed Programs", *IEE Software Engineering Journal*, vol.8(N.2), March 1993, pp.73-82

[Mag94a] Magee J., Dulay N., Kramer J., "A Constructive Development Environment for Parallel and Distributed Programs", in Proc. of the IEEE Intn'l Workshop on Configurable Distributed Systems (IWCCS'94), Pittsburgh PA, USA, March 1994.

[Mag94b] Magee J., Dulay N., Kramer J., "Regis: A Constructive Development Environment for Distributed Programs", *IEE Distributed Systems Engineering Journal*, vol.1(N.5), Dec. 1994, pp.304-312

[Mag97] Magee J., Tseng A., Kramer J., "Composing Distributed Objects in CORBA", *in Proc. of Intn'l Symposium of Autonomous Distributed Systems* (ISADS'97), Berlin, April 97.

[Mar95] V. Marangozov, "Conception et Réalisation d'un service de désignation associative pour la construction d'applications coopératives", Rapport de DEA d'Informatique, Institut National Polytechnique de Grenoble, Grenoble, France, Juin 1995.

[Mar97] Marangozov V., Bellissard L., Vion-Dury J.-Y., "Connectors : a Key Feature For Building Distributed Component-based Architectures", *in Proc. Of the 2nd European research Seminar on Advanced Distributed Systems (ERSADS'97)*, Zinal, Switzerland, March 1997

[Nie95a] Nierstrasz O., Dami L., "Component-Oriented Software Technology", in Object-Oriented Software Composition, Eds. Nierstrasz O. and Tsichritzis D., Prentice Hall, Englewood Cliffs NJ USA, 1995, pp.3-28

[Nie95b] Nierstrasz O., Meijler T. D., "Research Directions in Software Composition", ACM Computing Surveys, vol.27(N.2), June 1995, pp.262-264.

[OMG95a] Object Management Group, "The Common Object request Broker Architecture", Object Management Group, Revision 2.0, 1995

[OMG95b] Object Management Group, "CORBAServices: Common Object Services Specification", Object Management Group, Revision Edition March 31, 1995

[OSF95] Open Software Foundation, "OSF DCE : Introduction to OSF DCE", Open Software Foundation, Revision 1.1, 1995

[Pur94] Purtilo J. M., "The POLYLITH Software Bus", *ACM TOPLAS*, vol.16(N.1), Jan. 1994, pp.151-174.

[Qui94] Quint V., Vatton I., "Making Structured Document Active", Electronic Publishing, vol.7(N.2), June 1994, pp.53-74

[Rog97] Rogerson D., "Inside Com", Microsoft Press, One Microsoft Way, Redmond WA 98052-6399 USA, 1997, ISBN 1-57231-349-8

[Ros95] Rossum Van R. "Python Reference Manual", Dept. AA., CWI, P.O. Box 94079, 1090 GB Amsterdam, NL, Release 1.3, Oct. 1995 (http:\\www.python.org\)

[Ros95b] Rossum Van R. "Extending and Embedding Python", Dept. AA., CWI, P.O. Box 94079, 1090 GB Amsterdam, NL, Release 1.3, Oct. 1995 (http:\\www.python.org\)

[Sha95] Shaw M., DeLine R., Klein D. V., Ross T. L., Toung D. M., Zelesnik G., "Abstractions for Software Architecture and Tools to Support Them", *IEEE Trans. Software Engineering*, vol.SE-21(N.4), April 1995, pp. 314-335.

[Sno89] Snodgrass R., "The Interface Description Language : Definition and Use", Computer Science Press, Rockville MD, USA, 1989.

[Sun90] Sun Microsystems, "Network Programming Guide", Sun Microsystems Inc., Mountain View CA, 1990

[Vio97a] Vion-Dury J.-Y., Bellissard L., Marangozov V., "A Component Calculus for Modeling the Olan Configuration Language", *to appear in Proc. Of COORDINATION'97*, Berlin, Germany, September 97

[Vio97b] Vion-Dury J.-Y., Bellissard L., Marangozov V., "A Component Calculus for Modeling the Olan Configuration Language (Extended Version)", *Technical Report INRIA*, to appear, September 97

RESUME en Français

L'objectif de cette thèse est d'étudier les modèles de construction et de configuration d'applications réparties. La configuration d'une application correspond à la fois à l'identification de l'architecture logicielle, en terme de composants logiciels et des communications nécessaires, ainsi qu'à son adaptation aux ressources disponibles dans un environnement réparti pour produire une image exécutable de l'application.

Le résultat de ce travail est en premier lieu le langage de configuration Olan (OCL) qui apporte des solutions aux problèmes de définition de l'architecture d'une application, d'intégration de logiciels existants hétérogènes, d'adaptation aux facteurs de répartition, notamment aux ressources et mécanismes répartis disponibles, et enfin à la génération de l'image exécutable. En second lieu, ce travail propose un support système permettant d'installer et de déployer l'applications selon les directives et contraintes spécifiées au travers de l'architecture, sur un système réparti hétérogène.

TITRE en Anglais

Engineering and Configuration of Distributed Applications

RESUME en Anglais

This work aims at studying various models for the engineering and the configuration of distributed applications. The word configuration refers here to both the identification of the software architecture, in terms of software components and communication requirements, as well as the adaptation to the available resources and the underlying constraints of the targeted distributed environment.

The result of this work is at first the definition of the Olan Configuration Language (OCL) which major contributions are a mean of highlighting the application architecture, focusing on some aspects of the application dynamic behavior, a solution to the integration of heterogeneous legacy software, the configuration of the application related to the distribution concerns, such as the available resources or remote communication mechanisms. All this contribution finally aims at generating the executable image of the application. In a second hand, this work has also defined a system support that enables the remote installation and deployment of an application architecture according to the OCL specification onto a heterogeneous distributed environment.

MOTS-CLES

Systèmes réparties, Construction d'Applications réparties, Langages de définition d'Architecture

DISCIPLINE

Informatique

LABORATOIRE de RATTACHEMENT

Laboratoire SIRAC
INRIA Rhône Alpes, 655 Avenue de l’Europe, F-38330 Montbonnot Saint-Martin.