



Université Joseph Fourier

U.F.R Informatique &
Mathématiques Appliquées



Institut National Polytechnique
de Grenoble

ENSIMAG

I.M.A.G.

École Doctorale
Mathématiques et Informatique

DEA Informatique :
SYSTÈMES ET COMMUNICATIONS

Projet présenté par

Aline Senart

**Aspects Dynamiques dans les Architectures Logicielles en
Environnement Réparti**

Effectué au laboratoire SIRAC

Date : 20/06/2000

Jury : Ph. Morat
Ch. Boitet
S. Krakowiak
L. Bellissard
N. De Palma

Remerciements

Je remercie toutes les personnes qui ont rendu ce DEA possible et je tiens à remercier plus particulièrement :

Mr Philippe Morat pour avoir accepté de participer à ce jury ;

Mr Christian Boitet pour avoir bien voulu juger mon travail ;

Mr Sacha Krakowiak pour la relecture de mon rapport et ses brillants conseils de rédaction. Je le remercie également pour son jugement sur ce travail ;

Mr Bellissard et Mr De Palma, pour leur encadrement et leurs encouragements tout au long de cette année ;

Je ne peux oublier de remercier l'ensemble de l'équipe Sirac et notamment Roland pour m'avoir accueillie au sein du projet, Sara, Fabienne, Vania et Manu pour leur gentillesse et leur soutien, Olive pour nos nombreux et interminables éclats de rires. Je tiens également à remercier Philippe et Christophe pour m'avoir supporté au cours de cette année, pas toujours facile.

Une pensée particulière à tous mes amis PicaTeamiens pour les nombreuses parties réseaux, les parties de billard du dimanche soir et leur passion illimitée pour les petits hommes verts...

J'adresse mes derniers remerciements à ma famille pour son soutien et ses encouragements, ainsi qu'à Olivier pour m'avoir remonté tant de fois le moral et pour nos nombreuses discussions...

Table des matières

1	Introduction	1
1.1	Contexte	1
1.2	Vers les descriptions d'architectures logicielles	2
1.2.1	Architecture	3
1.2.2	Programmation par composants	3
1.2.3	Langages de description d'architecture	3
1.3	Problématique	5
1.4	Démarche et organisation du document	6
2	Langages de description d'architecture	7
2.1	UniCon	7
2.1.1	Modèle et notations	8
2.1.2	Évolution dynamique	14
2.1.3	Atouts	15
2.1.4	Limites	15
2.1.5	Bilan	16
2.2	Acme	16
2.2.1	Éléments de l'architecture	17
2.2.2	Traduction d'une description d'architecture	19
2.2.3	Évolution dynamique	21
2.2.4	Atouts	21
2.2.5	Limites	21
2.2.6	Bilan	22
3	Langages de configuration	23
3.1	Darwin	23
3.1.1	Composants	24
3.1.2	Opérateurs particuliers	26
3.1.3	Évolution dynamique	27
3.1.4	Système à l'exécution	28
3.1.5	Atouts	29
3.1.6	Limites	29
3.1.7	Bilan	31
3.2	Olan	31
3.2.1	Modèle de composants	31
3.2.2	Modèle d'interactions	35

3.2.3	Évolution dynamique	37
3.2.4	Système à l'exécution	38
3.2.5	Atouts	39
3.2.6	Limites	40
3.2.7	Bilan	40
4	Langages de description dynamiques	41
4.1	Wright	41
4.1.1	Modèle et notations	42
4.1.2	Évolution dynamique	45
4.1.3	Atouts	47
4.1.4	Limites	48
4.1.5	Bilan	49
4.2	Rapide	49
4.2.1	Concepts	49
4.2.2	Évolution dynamique	52
4.2.3	Atouts	54
4.2.4	Limites	54
4.2.5	Bilan	55
5	Aspects dynamiques des ADL : analyse expérimentale et proposition	56
5.1	Synthèse des approches existantes	56
5.2	Objectifs	59
5.3	Environnement	60
5.3.1	Support d'exécution A3	60
5.3.2	Environnement Olan	61
5.4	Application pilote	62
5.5	Limitations	66
5.6	Expression de la dynamicité dans Olan	66
6	Utilisation de règles pour la configuration dynamique d'applications	68
6.1	Hypothèses relatives aux services à l'exécution	68
6.2	Règles actives	69
6.2.1	Événement	70
6.2.2	Condition	71
6.2.3	Action	71
6.3	Expression de la configuration dynamique au moyen de règles actives	72
6.3.1	Syntaxe des règles	72
6.3.2	Expression dans l'ADL	72
6.3.3	Configuration dynamique	73
6.4	Modèle d'exécution des règles	76
6.5	Discussion	78
6.5.1	Conservation du formalisme ?	78
6.5.2	Modélisation de l'Action	78

6.5.3	Simplifications apportées	79
7	Conclusion	80
7.1	Objectif et démarche de travail	80
7.2	Évaluation	81
7.3	Perspectives	82
A	Utilisation simplifiée de CSP	87

Table des figures

1.1	<i>Cycle de vie d'une application répartie</i>	2
2.1	<i>Modèle d'architecture</i>	8
2.2	<i>Types de composants</i>	8
2.3	<i>Représentation de l'application bancaire dans UniCon</i>	11
2.4	<i>Exemple d'utilisation d'Acme</i>	16
2.5	<i>Traduction de Wright vers Rapide via Acme</i>	20
3.1	<i>Présentation des composants de l'application bancaire en Darwin</i>	24
3.2	<i>Représentation graphique de la configuration</i>	26
3.3	<i>Représentation Olan du composite Application</i>	34
4.1	<i>Exemple de connecteur Wright</i>	43
4.2	<i>Deux configurations possibles pour la topologie dynamique</i>	45
4.3	<i>Représentation d'un serveur de noms en Rapide</i>	53
5.1	<i>Représentation de l'annuaire téléphonique</i>	60
5.2	<i>Architecture du serveur d'agents A3</i>	61
5.3	<i>Environnement Olan</i>	62
5.4	<i>Description OCL du client de l'annuaire téléphonique</i>	63
6.1	<i>Actions associées à des méthodes d'un contrôleur</i>	75
6.2	<i>Action associée à un script</i>	75

Liste des tableaux

2.1	<i>Types de composants dans UniCon et leurs players</i>	10
2.2	<i>Types de connecteurs dans UniCon et leurs rôles</i>	13
3.1	<i>Types de service d'Olan</i>	32
4.1	<i>Opérateurs de dépendance entre événements</i>	51
5.1	<i>Caractéristiques des différents ADL</i>	57
5.2	<i>Dynamacité possible pour les abstractions architecturales</i>	58
5.3	<i>Lieu de dynamacité pour les différents ADL</i>	58
6.1	<i>Opérations de reconfiguration</i>	69
6.2	<i>Types d'événements primitifs</i>	69

Chapitre 1

Introduction

Ce document présente le travail effectué dans le cadre du projet de DEA, mené au cours de l'année universitaire 1999/2000. Ce travail s'intègre dans le projet Sirac¹, commun à l'INRIA², à l'Institut National Polytechnique de Grenoble et à l'université Joseph Fourier. Le domaine de recherche couvert par Sirac est la construction de systèmes et d'applications informatiques répartis. Les recherches sont menées dans les deux domaines suivants :

- construction d'applications réparties adaptables ;
- support système pour serveurs en grappes.

Notre DEA s'inscrit dans le contexte du premier axe de recherche, la construction d'applications réparties adaptables.

Les applications tendent à devenir de plus en plus dynamiques. Les bouleversements induits par la généralisation des réseaux, la répartition et la mobilité impliquent un nombre variable de clients et de services. Le souci majeur des applications est donc de pouvoir s'adapter à ces variations et supporter la dynamique. Notre sujet de DEA s'intéresse à une classe de langages dont l'objectif est de décrire l'architecture d'applications. Nous souhaitons inclure au niveau de ces langages, et plus particulièrement un, une description de l'évolution des architectures logicielles. Les notions d'architecture et de langages de description d'architecture sont détaillées dans la suite du document.

1.1 Contexte

La conception, le développement et la maintenance des applications de grande complexité posent des problèmes difficiles à résoudre. Les concepteurs sont confrontés à des contraintes de réutilisation de code existant, d'installation d'applications dans des contextes matériels ou logiciels qui peuvent varier avec le temps, des contraintes d'administration, d'évolution des applications, *etc.* La conception d'applications par le biais des langages de programmation classiques (C, Ada...) permet difficilement de respecter ces contraintes. Ces langages sont intéressants pour programmer des composants logiciels d'une application mais

¹ Systèmes Informatiques Répartis pour Applications Coopératives

² Unité de Recherche Rhône-Alpes

sont inadéquats pour intégrer les composants en les faisant communiquer et coopérer. D'autre part, à l'opposé des applications traditionnelles, le réalisateur d'applications distribuées ne peut avoir une vision globale de l'état et du comportement de son application. De ce fait, des problèmes sont rencontrés pour l'intégration, l'installation et l'administration dans un environnement réparti.

La programmation constructive, dénommée aussi programmation par composition de logiciels, est au centre d'études récentes sur le développement d'applications distribuées. Cette méthode se propose de construire les applications, en termes d'assemblage de composants logiciels autonomes et hétérogènes. Différentes phases caractérisent le processus de construction d'une telle application. Ce processus est schématisé dans la figure 1.1.

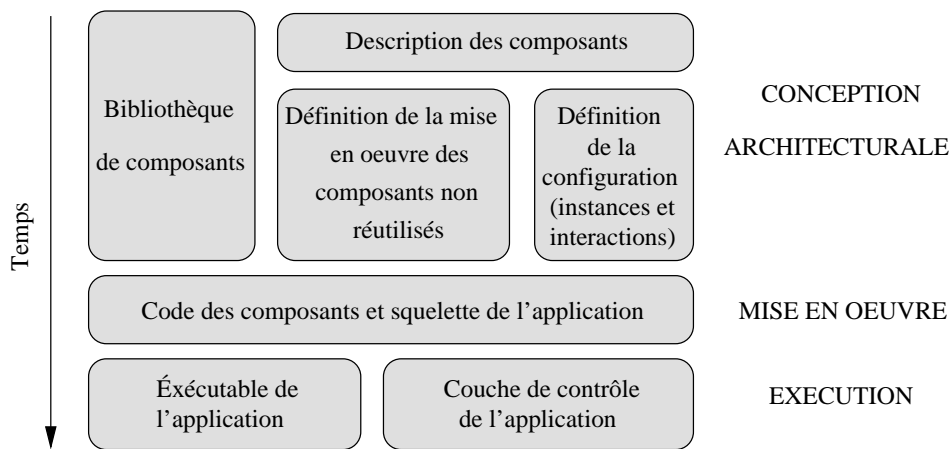


FIG. 1.1 – Cycle de vie d'une application répartie

La phase de conception consiste dans un premier temps à analyser et spécifier les besoins de l'application. C'est généralement réalisé par des méthodes de conception ou de modélisation (*p. ex.* l'UML [BRJ97]). En complément de ces méthodes, les langages de description d'architecture (ADL) apportent une vision globale de la structure de l'application. Ils permettent de spécifier son architecture à partir d'une bibliothèque de composants.

Le compilateur génère ensuite le code binaire et le squelette du système et participe, par là même, à la mise en œuvre de l'application.

Celle-ci peut alors être déployée sur un ensemble de sites distribués. Comme c'est illustré dans le schéma, un service offre la possibilité de contrôler de d'administrer, au sein de l'environnement d'exécution, le comportement de l'application. Ce service facilite ainsi la maintenance de l'application.

1.2 Vers les descriptions d'architectures logicielles

Des études récentes [ISZ98, ADG98, OMT98] ont montré l'importance de la notion d'architecture. Cette dernière permet d'exposer de manière compréhensible et synthétique la complexité d'un système logiciel et de faciliter l'assemblage de composants logiciels. Les axes de travail autour de ce thème sont

multiples, l'idée fondamentale est de profiter des avantages d'une architecture clairement explicitée pour ne laisser qu'en second plan la programmation d'applications. C'est pour cela que nous accordons dans cette partie une certaine attention à la notion d'architecture et présentons un nouveau modèle de programmation s'appuyant sur des langages de construction et de configuration, les ADL (Architecture Description Language).

1.2.1 Architecture

De nombreuses recherches ont permis d'introduire une dimension nouvelle à la notion d'architecture de logiciels. La définition généralement admise de ce concept est qu'une « (...) *architecture spécifie les modules du système (appelés composants du système) et l'interaction entre ces composants afin de satisfaire les besoins d'un système* » [LVM95]. Cette notion est essentielle puisqu'elle est utile pour la construction et l'administration de systèmes. Plus particulièrement, elle permet l'évolution de l'application (modification des constituants d'un système, de leur placement...), la définition de contraintes sur le système, et facilite la mise en œuvre de l'application en automatisant son processus d'installation. En outre, la structure à un haut niveau d'abstraction exposant des collections de composants interagissants, permet la spécification de propriétés importantes du système (protocoles d'interaction, localisation des données...).

1.2.2 Programmation par composants

La programmation par composants est une technique de construction d'applications par intégration d'entités logicielles. Les composants sont conçus pour être réutilisés de manière homogène dans différentes applications et contextes d'exécution, sans modification de leur implémentation.

Cette approche permet ainsi d'envisager un processus de développement d'applications à deux niveaux : un premier correspondant à l'architecture globale du système et le second correspondant à l'écriture des composants. La phase de programmation macroscopique définit l'assemblage de composants logiciels formant l'application. Elle est à l'origine de l'architecture du système. Enfin, la phase de programmation microscopique permet la réalisation d'un composant logiciel donné à l'aide de langages de programmation classiques.

Nous pouvons constater qu'il manque, au niveau de la phase macroscopique, un modèle de spécification de l'assemblage de composants. Un grand nombre de chercheurs ont proposé des notations formelles pour représenter et analyser les concepts architecturaux. Souvent appelés ADL, ces notations fournissent à la fois une base conceptuelle et une syntaxe concrète pour la caractérisation d'architectures logicielles.

1.2.3 Langages de description d'architecture

L'intégration d'entités logicielles requiert un langage commun, indépendant des langages de programmation. Sans un tel langage, le développeur d'un système distribué doit rendre lui-même les entités logicielles inter-opérables au sein

d'un environnement hétérogène. Les ADL permettent de spécifier des architectures d'applications en offrant un moyen pratique et abstrait pour une description compréhensible d'un système complexe. Ils permettent l'expression, dans un mode déclaratif, de la configuration d'une application en termes de composition d'unités logicielles élémentaires et d'éventuelles contraintes de déploiement.

Chacun de ces langages fournit une manière de décrire l'architecture d'un logiciel selon son utilisation finale. Certains langages ont le mérite d'aider à la vérification formelle de propriétés, de permettre la simulation d'une architecture, ou alors d'aider de manière directe à la mise en œuvre et l'exécution du programme. Un « Architecture Description Language » se compose communément [MT97] de trois éléments : les composants, les connecteurs et les configurations.

Composant

Le concept de composant est l'élément central du modèle d'assemblage. Le composant étend la notion d'objet, qui possède de nombreuses qualités, parmi lesquelles l'héritage et le polymorphisme. Cependant, les limites du modèle à objet commencent à apparaître, spécialement dans un contexte réparti. En ce qui concerne la lisibilité de l'architecture de l'application, ce modèle n'apporte que peu de solutions hormis la connaissance de l'ensemble des types et classes utilisés. Le schéma d'instanciation des objets et la politique de répartition ne sont pas clairement exposés. D'une manière générale, aucune vision claire de l'architecture n'est proposée. Il est de même impossible de définir des assemblages d'objets. L'héritage des classes est en effet insuffisant car il reporte les structures de l'objet dans les objets qui en dérivent, brisant par là même le concept d'encapsulation. Une autre limitation est à remarquer. Les objets sont décrits par une interface, au même titre que les fichiers *.h* du langage C, ou les spécifications Ada. Cependant, cette interface reste incomplète, elle ne spécifie pas les méthodes externes nécessaires à l'objet. La description de l'architecture n'est pas non plus découplée de l'implémentation de l'objet. Elle est exprimée directement dans le code des objets et entrave, de ce fait, la réutilisation et l'intégration de ces modules logiciels dans des systèmes différents.

Le composant se veut plus spécialisé, il n'implante que du code spécifique de l'application réalisée, encore appelé « code métier ». C'est avant tout une entité d'intégration et de structuration de logiciels, dans la mesure où il peut encapsuler du code ou permettre une hiérarchisation de l'application³, spécifiée indépendamment de l'implémentation du composant. Les composants logiciels sont, au même titre que les objets, décrits par une interface qui explicite la liste des procédures et données, les types et éventuellement les événements pouvant être visibles par les autres composants du système. Cette interface exhibe, en plus, les informations et les appels qui lui sont nécessaires chez les autres composants du système. Par ailleurs, les composants dérivent d'un type qui représente le mode de mise en œuvre, *e.g.* un processus, une bibliothèque à chargement dynamique, *etc.* Cet aspect de typage impose des modes d'accès particuliers vis à vis du composant et autorise des vérifications, dans la phase d'analyse, de

³Dans la plupart des langages de description d'architecture, ces deux fonctionnalités découlent de la nature propre du composant (*primitif* ou *composite*).

la faisabilité et de la validité de l'architecture. La description d'un composant au niveau du langage permet de donner lieu à de multiples instances, créées dynamiquement lors de l'exécution de l'application.

Interaction

Le connecteur, entité de gestion des interactions entre les composants, permet de définir le comportement des communications. Toutefois, certains langages de description d'architecture ne fournissent pas de connecteur permettant de spécifier le mécanisme d'interaction. La modification du mode de communication entre ces composants n'est alors pas aisée. Un changement du mécanisme de communication entre sites demande effectivement un remaniement profond de la réalisation de ces objets distribués.

Les connecteurs, ou interconnecteurs, sont donc des abstractions essentielles pour la maintenance et la réutilisation des composants. Ils permettent de séparer la phase de programmation en deux niveaux : la programmation classique des différents modules logiciels et le codage des connexions entre ces modules. Ils contiennent les informations concernant les règles d'interconnexion de composants, telles que le protocole, la spécification des échanges de données ainsi que les transformations éventuelles du format d'échange. Chaque connecteur dérive, comme les composants, d'un type qui représente le mécanisme et les propriétés de communication : type de communication, règles de synchronisation, schéma d'ordonnancement, *etc.*

Composition hiérarchique

Une composition hiérarchique, ou *configuration*, est une description de l'ensemble des composants logiciels nécessaires pour le fonctionnement d'une application, ainsi que celle de leurs communications. Elle permet de gérer la complexité de l'application en raffinant peu à peu son architecture en un ensemble hiérarchique. Une configuration contient en outre toutes les informations relatives à l'instanciation des composants logiciels et les interconnexions entre ces composants. De ce fait, sa manipulation permet de former facilement une nouvelle architecture.

1.3 Problématique

Une application est administrée de manière à exploiter au mieux les caractéristiques et ressources du site ou de l'environnement (machines, disques, réseaux de communication, utilisateurs, *etc.*) où elle est déployée. Ses caractéristiques pouvant évoluer, il est souhaitable de pouvoir redéfinir la configuration de l'application en cours d'exécution. La structure de l'application spécifiée dans le langage de description d'architecture est relativement proche de l'application, ce qui permet d'envisager l'utilisation du formalisme de description pour des opérations d'administration de l'application, par exemple de surveillance d'exécution, de changements dynamiques de composants. Cependant, peu de langages permettent d'introduire des stratégies d'administration au niveau de l'ADL. Les

solutions proposées ne s'appliquent qu'à une description statique d'une architecture logicielle répartie, elles ne prennent que très partiellement en compte l'évolution de cette architecture. Effectivement, la description de l'architecture correspond à la représentation du système au moment initial, au moment de son installation.

Notre travail consiste donc à mettre en oeuvre une solution dans le cadre particulier du langage OCL [BPF], de l'environnement Olan [BAKR95]. Pour cela, il est nécessaire d'introduire un moyen d'expression pour la dynamique ainsi que des notions de reconfiguration, au niveau du langage de description. Un certain nombre de questions sont, à cette occasion, soulevées : Quelles éléments de reconfiguration sont nécessaires ? Quel moyen d'expression pour les politiques d'administration ? Quel formalisme choisir ? Où définir ces stratégies dans l'ADL ? Quid de l'infrastructure à l'exécution ?

1.4 Démarche et organisation du document

Ce document se propose de présenter une solution à l'expression de la dynamique d'une application, au niveau d'un langage de description d'architecture particulier, OCL. Nous avons commencé par une étude des différents langages existants, étude que nous avons abordée en se concentrant particulièrement sur les mécanismes de reconfiguration et de dynamique. Tout au long de cette taxonomie, nous avons développé un exemple concernant une application bancaire, constituée d'un compte en banque et du titulaire de ce compte, un client. Cette illustration nous a permis de comparer, de manière plus détaillée, les similitudes et différences entre ces langages. Nous avons ensuite porté notre effort sur la réalisation d'un exemple concret, au cœur de l'environnement Olan, afin de déterminer précisément les lacunes rencontrées au niveau de son langage de description d'architecture, OCL. Cette étape nous a permis de mieux cibler notre problématique et de définir exactement nos besoins dans le cadre d'applications réparties. La phase finale de ce travail est la définition d'un modèle de règles, intégré au modèle de composants d'OCL, qui est dédié à la gestion de la dynamique des applications.

Les chapitres 2, 3 et 4 exposent une taxonomie de différents modèles de construction et de configuration d'applications réparties. Au cours de cette étude, un accent particulier est mis sur la gestion et le contrôle de l'application au niveau des langages de description et de configuration d'architectures. Le chapitre 5 contient une synthèse de ces différents langages et propose, à travers une expérimentation concrète, une analyse qui conduit à une solution possible. Dans le chapitre 6, nous présentons notre proposition, dédiée à l'introduction de politiques d'administration au niveau de ces langages. Enfin, nous proposons quelques conclusions dans le chapitre 7.

Chapitre 2

Langages de description d'architecture

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 soient réellement prises en compte. Ces dernières années ont vu apparaître de nouvelles techniques orientées objets, cependant, elles n'ont rien apporté de nouveau pour la prise en compte de la distribution. Ce besoin d'intégrer plus fortement la répartition dans la conception des applications se fait de plus en plus ressentir, notamment au niveau des systèmes à grande échelle et le phénomène se trouve amplifié avec le développement des réseaux de communication et des services autour de l'Internet. L'ingénierie logicielle s'oriente alors vers les langages de description d'architecture. Ce chapitre a pour objectif d'exposer les grands principes de ces langages. Nous nous efforcerons de montrer leurs avantages et ferons une étude de différents projets qui dérivent de ce concept. Ce chapitre s'intéresse particulièrement à une classe de langages n'apportant pas de solution pour exprimer la dynamique d'une application.

2.1 UniCon

La particularité de UniCon (Universal Connector Support) est de supporter un large éventail de styles d'architecture, ou modèles, trouvés dans les applications d'aujourd'hui et de construire à partir d'eux des systèmes complexes. Une description d'architecture dans UniCon s'adresse à deux principaux concepts, les composants et les connecteurs [SDZ96].

Dans cette partie, nous allons présenter, au-delà de ces notions, une vue plus générale de UniCon en nous attardant sur les différents concepts introduits par les langages de description d'architecture. Nous établirons enfin une synthèse sur les avantages et inconvénients des différentes approches de ce langage.

2.1.1 Modèle et notations

Le modèle décrit le système sous forme de deux entités principales, les composants et les connecteurs. Ils ont une structure analogue¹ (cf. figure 2.1) :

- un nom ;
- une spécification (*interface* ou *protocole*) contenant un type et une collection d'entités de composition (*players* ou *rôles*) ;
- une implémentation.

Toutes ces notations sont présentées dans la suite du document.

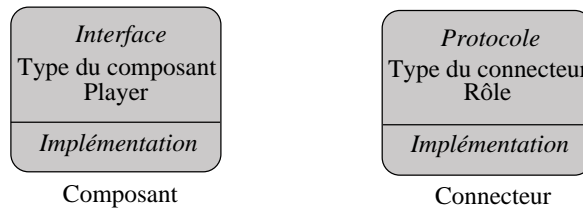


FIG. 2.1 – *Modèle d'architecture*

Composants

La première fonction d'un composant est d'intégrer du code logiciel. Son interface spécifie les fonctions que le composant exporte et importe dans son implémentation. Un composant définit aussi son type, ce qui permet de nombreuses vérifications au niveau de l'architecture. Cet aspect de typage semble important dans ce langage, nous y reviendrons dans la suite. Alors que l'application distribuée est spécifiée par un architecte, les implémentations de composants sont laissées à la disposition du concepteur de l'application. Comme dans la plupart des langages de description d'architecture, deux types d'implémentations de composants existent (cf. figure 2.2).

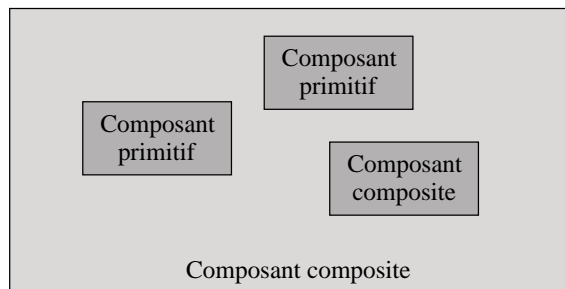


FIG. 2.2 – *Types de composants*

Les éléments primitifs font directement référence à du code source ou binaire. Les composants composites, quant à eux, ont une implémentation qui consiste en une liste de composants et connecteurs, des instructions de composition et

¹A part pour les connecteurs composites qui n'existent pas encore.

des connexions avec le composite englobant. En fait, un composant composite encapsule les informations sur la structure globale d'un sous-ensemble d'une application. Il permet de structurer l'application, dans le sens où il présente la hiérarchisation du système en termes d'entités plus petites, sous-composants primitifs ou composites.

Interface L'interface définit les propriétés du composant à plusieurs niveaux. Elle permet d'exhiber d'une part, les fonctions et données accessibles, et d'autre part, les fonctions et données requises par l'implémentation du composant. D'autre part, elle contraint la manière dont le composant doit être utilisé en imposant des modes d'accès aux fonctions et données qu'il possède. Ces modes d'accès sont spécifiés par des entités visibles, les *players*. Les players correspondent aux services² fournis et requis par le composant vis à vis des autres composants de l'application. Ils sont définis par un nom, un type et des attributs optionnels, comme la signature.

Type UniCon présente un système de typage fort pour ses composants, dans la mesure où chaque composant dérive d'un type qui le caractérise. Outre l'expression de la fonctionnalité globale du composant, les types expriment des restrictions sur les propriétés, en termes de formes prescrites pour la communication, le partage de données et autres interactions. Le tableau 2.1 synthétise ces mécanismes et liste tous les types de composants actuellement disponibles et les players autorisés pour chacun d'entre eux³.

Afin de comprendre plus finement l'utilisation du typage des composants dans UniCon, prenons l'exemple d'un composant *Module*. Ce type fait appel à la notion de bibliothèque, contenant un ensemble de fonctions. Ceci implique l'utilisation obligatoire d'un des players autorisés par ce type. Les players *RoutineDef* et *RoutineCall* correspondent aux procédures respectivement exportées ou importées. De même, *GlobalDataDef* et *GlobalDataUse* sont les players d'importation et d'exportation de données. *ReadFile* et *WriteFile* fournissent les fonctions de lecture et d'écriture sur un fichier. Cependant, il est aussi intéressant de définir une ou plusieurs collections de players. UniCon propose alors les *PLBundle*, players abstraits pour un groupe de fonctions⁴.

Notations La syntaxe des composants contient en premier lieu une interface. Celle-ci est composée de trois parties différentes : le type du composant, des propriétés optionnelles figurant sous forme de liste et enfin la définition des players. Le type du composant est l'expression d'une classe de fonctionnalités que le concepteur de l'architecture souhaite fournir au travers du composant. Il

²Les services sont des points d'entrée et de sortie des composants, seuls éléments visibles de l'extérieur du composant.

³Le lecteur pourra se référer à [SDK⁺95, page 17] pour plus de détails.

⁴Nous ne détaillerons pas ici les autres types de composant et players associés. Pour indication, *Process* sera le type d'un composant encapsulant un processus en cours d'exécution. *Computation* et *SharedData* sont des sous-types de *Module*, se restreignant dans le premier cas aux fonctions et dans l'autre aux données partagées. *SeqFile* représente un fichier, *Filter* gère les flux de données...

Type de composant	Type de players autorisés
Module	RoutineDef, RoutineCall, GlobalDataDef, GlobalDataUse, PLBundle, ReadFile, WriteFile
Computation	RoutineDef, RoutineCall, GlobalDataUse, PLBundle
SharedData	GlobalDataDef, GlobalDataUse, PLBundle
SeqFile	ReadNext, WriteNext
Filter	StreamIn, StreamOut
Process	RPCDef, RPCCall
SchedProcess	RPCDef, RPCCall, RTLoad
General	All

TAB. 2.1 – *Types de composants dans UniCon et leurs players*

restreint les nombres, types et spécifications des players qui peuvent être définis. Les propriétés sont des attributs spécifiant des informations supplémentaires sur le composant dans sa globalité. Quant aux players, ils sont les unités sémantiques visibles à travers lesquelles les composants peuvent interagir avec d'autres.

Dans une deuxième partie, le composant contient son implémentation, c'est-à-dire le lien entre l'interface et le code logiciel. L'implémentation d'un composant peut prendre deux formes différentes, primitive ou composite. L'implémentation primitive fait référence à un fichier source, indépendant du langage de description UniCon, qui contient l'implémentation. Ce fichier source peut être du code dans un langage de programmation traditionnel, un script shell, *etc.* Voici la syntaxe de UniCon pour un composant primitif :

```

COMPONENT componentName
  INTERFACE IS
    TYPE componentType
    <Liste de Propriétés>
    <Liste de Players>
  END INTERFACE

  IMPLEMENTATION IS
    <Liste de Propriétés>
    <Implémentation Primitive ou Composite>
  END IMPLEMENTATION
END componentName

```

Pour l'implémentation composite, une description d'un ensemble de composants ainsi que leur composition au moyen de connecteurs sont spécifiées. Nous allons maintenant détailler notre exemple d'application bancaire. C'est un composant composite constitué de deux composants primitifs : un titulaire d'un compte et son compte lui-même. Ces deux composants interagissent au moyen d'un connecteur qui les relie. Plus précisément, les rôles des connecteurs sont reliés aux players de chaque composant.

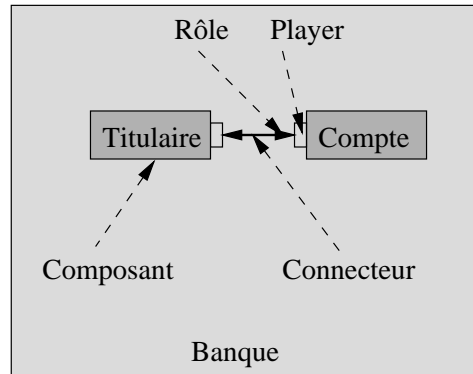


FIG. 2.3 – Représentation de l'application bancaire dans UniCon

La figure 2.3 représente cette application. Les composants primitifs sont représentés par des boîtes foncées. Le composant plus clair qui les englobe est l'application elle-même, décrite par un composant composite. Le trait reliant `Titulaire` et `Compte` schématise le connecteur. Ses rôles (les flèches) sont attachés aux players des composants, représentés par des petits rectangles. Ci-dessous est joint le code des composants `Titulaire` et `Banque`.

```

COMPONENT Titulaire
  INTERFACE IS
    TYPE Process
    PLAYER consultation IS RPCDef
      SIGNATURE ("int"; "int")
    END consultation
  END INTERFACE

  IMPLEMENTATION IS
    VARIANT titulaire IN "titulaire.c"
    IMPLTYPE (Source)
    END titulaire
  END IMPLEMENTATION
END Titulaire

```

Ce composant contient une interface et une implémentation primitive. Il est de type *Process* et offre un player possible pour l'accès à des fonctions par le mécanisme RPC. Il peut demander à un autre composant (notamment un compte) la consultation de son solde. Nous pouvons remarquer que la signature

de players est écrite dans la syntaxe du langage utilisé pour la programmation du code source. L'attribut *IMPLTYPE* spécifie le langage de programmation du code source (C).

```

COMPONENT Banque
  INTERFACE IS
    TYPE General
  END INTERFACE

  IMPLEMENTATION IS
    // COMPUSES : Instanciation des composants
    USES titulaire INTERFACE Titulaire
      PROCESSOR ("caroline.inrialpes.fr")
      ENTRYPOINT (titulaire)
    END titulaire
    USES compte INTERFACE Compte
      PROCESSOR ("dyade.inrialpes.fr")
      ENTRYPOINT (compte)
    END compte
    // CONNECT : Connexion utilisée
    ESTABLISH RPC WITH
      titulaire.consultation AS appellant
      compte.solde AS appelé
    END RPC
  END IMPLEMENTATION
END Banque

```

Les éléments composites fournissent le mécanisme de construction d'un système à partir de composants primitifs, comme ci-dessus, ou de sous-systèmes qu'il inclut dans son architecture. Une composition est constituée des :

- instanciations des composants dans la partie *COMPUSES* ;
- instanciations des connecteurs dans la partie *CONNUSES* ;
- associations de players de certains composants avec les players du composant composite établies dans la partie *BIND* ;
- définitions des interactions entre les composants dans la partie *CONNECT*.

L'application bancaire n'ayant pas besoin de typage précis, nous avons utilisé *General* qui sert à la structuration du logiciel. L'implémentation de l'application comprend l'instanciation des sous-composants primitifs *titulaire* et *compte*, de type *Titulaire* et *Compte* respectivement. Au cours de ces instanciations, des propriétés propres au type des composants peuvent être positionnées, telles le site d'exécution par l'attribut *PROCESSOR* (par défaut, c'est la site local), le point d'entrée pour le lancement des processus (par défaut, le *main* du module), le niveau de priorité, *etc.* Juste après la définition des composants utilisés, l'implémentation contient la déclaration des interactions entre ces composants. La clause *ESTABLISH* permet de décrire une interaction entre les deux composants par le biais d'un connecteur RPC.

Connecteurs

Un connecteur permet d'établir des connexions entre les composants. Il spécifie les protocoles d'interactions et tous les autres mécanismes nécessaires pour les communications : spécifications des échanges de données et choix du mode de communication. Il est défini par un *protocole* qui spécifie le type d'interaction que le composant fournit. Par ailleurs, le connecteur possède, comme le composant, une implémentation. Actuellement, seul un type d'implémentation de connecteur est possible, c'est le connecteur primitif.

Protocole Le protocole définit les interactions permises parmi la collection de composants de l'application et fournit des garanties sur ces interactions. En fonction du type du connecteur, le protocole définit des *rôles*. Ces entités représentent les points de branchement auxquels les *players* d'un composant peuvent se connecter. Un rôle est décrit au moyen d'un nom et d'un type. Il comporte aussi une liste d'attributs optionnels comme une signature, des spécifications fonctionnelles et des contraintes d'utilisation. Le protocole doit, par conséquent, contenir le type du connecteur, les assertions qui imposent des restrictions à ce connecteur (*p. ex.* des règles d'ordonnancement), et les *rôles* qui servent de points d'entrée ou de sortie du protocole.

Type de connecteur	Type de rôles et players autorisés
Pipe	Source (StreamOut de Filter, ReadNext de SeqFile) Sink (StreamIn de Filter, WriteNext de SeqFile)
FileIO	Reader (ReadFile de Module) Readee (ReadNext de SeqFile) Writer (WriteFile de Module) Writee (WriteNext de SeqFile)
ProcedureCall	Definer (RoutineDef de Computation ou Module) Caller (RoutineCall de Computation ou Module)
DataAccess	Definer (GlobalDataDef de SharedData ou Module) User (GlobalDataUse de SharedData, Computation ou Module)
PLBundler	Participant (PLBundle, RoutineDef, RoutineCall, GlobalDataUse, GlobalDataDef de Computation, Module ou SharedData)
RemoteProcCall	Definer (RPCDef de Process ou SchedProcess) Caller (RPCCall de Process ou SchedProcess)
RTScheduler	Load (RTLload de SchedProcess)

TAB. 2.2 – Types de connecteurs dans *UniCon* et leurs rôles

Type Les rôles sont les unités sémantiques visibles à travers lesquelles les connecteurs font les interconnexions entre les composants. Les rôles identifient le type d'interaction qu'un connecteur peut établir, le type des composants avec lesquels ils peuvent être connectés, le type de players qu'ils supportent. Plus précisément, certains rôles requièrent des players particuliers. Le tableau 2.2 synthétise les types de connecteurs actuellement supportés, les rôles permis pour chaque et les players que les rôles supportent.

Par exemple, pour le type de connecteur *RemoteProcCall*, les rôles de type *Definer* ou *Caller* doivent être définis au niveau du protocole du connecteur. Ils permettront l'interaction entre les players *RPCDef* et *RPCCall*.

Notations Dans l'exemple précédent, le connecteur n'était pas encore défini. Sa description est spécifiée ici, par le code suivant :

```
CONNECTOR RPC
  PROTOCOL IS
    TYPE RemoteProcCall
    ROLE appelé is definir
    ROLE appelant is caller
  END PROTOCOL

  IMPLEMENTATION IS
    BUILTIN
  END IMPLEMENTATION
END RPC
```

Nous pouvons remarquer que cette notation est très similaire à celle des composants. Notamment au niveau de la structure, nous retrouvons la séparation entre le protocole (interface pour le composant) et l'implémentation. Le protocole contient dans sa définition le type dont dérive le composant, une liste optionnelle de propriétés et une liste de rôles. L'implémentation du connecteur ne figure qu'en deuxième partie. Celle-ci ne correspond qu'à des éléments primitifs, UniCon ne permettant pas encore l'utilisation de connecteurs composites.

La description de ce connecteur rend le renommage des rôles de *RemoteProcCall* possible. Il est à noter aussi qu'aucun mécanisme de UniCon ne permet de définir des implémentations personnalisées de connecteurs. Seule l'implémentation *BUILTIN* est disponible.

2.1.2 Évolution dynamique

Une architecture UniCon est définie par une énumération statique de composants et de connecteurs. Aucun aspect dynamique n'apparaît donc. La création et la suppression de nouveaux composants/connecteurs pendant l'exécution ne peut pas être décrite dans ce langage.

2.1.3 Atouts

UniCon est un langage qui présente un nombre non négligeable d'avantages. Parmi eux, nous pouvons retenir :

- Le système de typage fort est intéressant pour plusieurs raisons. Le compilateur de UniCon vérifie statiquement si les interconnexions sont valides et conformes au typage des entrées et sorties des composants et des connecteurs. De plus, ce typage impose un certain mode d'accès aux entités du système en restreignant leur accès. Enfin, chaque type de players et rôles correspond à un mécanisme d'accès aux composants et connecteurs, permettant aux outils de génération de produire le code associé ;
- Le mécanisme de communication n'est pas inclus dans le code des composants mais tient une place particulière au niveau du langage de description d'architecture. Par conséquent, une même définition d'un connecteur peut gérer les échanges entre des composants divers et variés, à partir du moment qu'il sont autorisés à interagir via ce type de communication. Outre la gestion de la communication, cette séparation entre composants et connecteurs permet de valider l'architecture au niveau des interconnexions ;
- Les sources, objets et exécutables peuvent être encapsulés dans les composants. Cette intégration à différents niveaux semblent offrir des mécanismes suffisants pour la conception d'applications. De nombreuses primitives de construction très utiles sont disponibles : les processus, filtres, pipes...
- La symétrie entre les composants et les connecteurs facilite la conception d'architectures.

2.1.4 Limites

Malgré les nombreux avantages présentés par UniCon, un certain nombre d'inconvénients sont à relever :

- Bien entendu, la limitation majeure de UniCon correspond à la description statique de l'architecture qu'il fournit. Ceci implique que tous les composants doivent être instanciés initialement et qu'aucun ajout ni suppression ne sont autorisés par la suite ;
- Trop de rigidité induit par le typage fort est aussi à déplorer. Bien que les concepteurs d'applications réparties aient de bonnes abstractions informelles pour les interactions, ils sont restreints à l'utilisation d'abstractions par défaut, ou implicites, plutôt qu'un choix délibéré ;
- UniCon supporte seulement un ensemble prédéfini de types pour les composants et les connecteurs. L'ensemble n'est pas complet et demande de faire une taxonomie approfondie sur le sujet. Il est toutefois prévu de rendre cet ensemble extensible. La définition de nouvelles abstractions et constructions pour les composants et connecteurs est aussi prévue dans le futur ;
- Bien que l'implémentation devrait supporter un bon nombre de langages, le C est actuellement le seul accepté. Les interfaces sont entièrement dé-

- pendantes du langage utilisé, *p. ex.* la signature est exprimée dans le langage du code encapsulé. Ainsi, un composant est difficilement réutilisable dans un contexte différent de celui de sa création. De nombreux changements par le concepteur de l'architecture seraient nécessaires dans le codage du composant. Par exemple, si le concepteur veut rendre son composant accessible à distance, il doit apporter des modifications sur les connecteurs pour que leur type soit *RemoteProcCall* et certainement sur l'interface des composants afin de rendre les communications possibles ;
- Les seuls composants primitifs disponibles dans la version de [SDZ96] sont les unités de compilation *BUILTIN*, il n'est pas encore possible d'en définir des personnalisés. De même, UniCon supporte les composants composites mais pas encore les connecteurs composites.

2.1.5 Bilan

UniCon est un langage de description d'architecture qui semble intéressant sur de nombreux aspects, particulièrement pour la séparation entre le composant et le connecteur, mais son système de typage reste trop contraignant et manque de souplesse. De plus, une architecture UniCon ne présente pas d'abstraction au niveau du langage pour décrire la dynamique de l'application. Elle ne permet ni de définir la présence variable d'un composant, le marquant explicitement comme optionnel, ni de spécifier des composants alternatifs. D'autre part, aucun ajout ou suppression de composants et connecteurs n'est possible à l'exécution.

2.2 Acme

Le projet Acme a commencé au début de l'année 1995. Ce langage de description d'architecture est assez particulier puisqu'il se propose de combiner les fonctionnalités de multiples ADL [GMW97]. La figure 2.4 le présente comme un intermédiaire entre différents langages de description d'architecture. Bien qu'il soit essentiellement utilisé comme un format intermédiaire, le langage Acme permet aussi la construction de nouveaux langages de description ainsi que la programmation de nouveaux outils à partir d'une structure standard. Actuellement Acme fournit une structure générique, extensible pour la représentation, la génération et l'analyse de descriptions d'architectures.

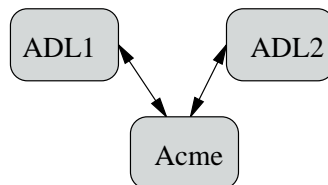


FIG. 2.4 – *Exemple d'utilisation d'Acme*

L'élaboration de ce langage, capable de supporter l'utilisation de descriptions d'architectures d'origines diverses, permet l'utilisation d'une palette plus grande d'outils. Effectivement, non seulement les outils développés autour d'Acme sont

disponibles mais aussi les outils de construction et d'analyse provenant des autres ADL.

2.2.1 Éléments de l'architecture

Les éléments de l'architecture fournis par le langage Acme ont été choisis de sorte à correspondre aux concepts communément admis par les différents ADL. Le langage est construit autour d'un noyau de sept entités pour la représentation architecturale : les composants, connecteurs, systèmes, ports, rôles, représentations et rep-maps.

Les composants sont des blocs de base pour la construction d'une description d'un système, au même titre que les composants de UniCon. Ils représentent les éléments primitifs de calcul et d'enregistrement de données du système. Des exemples typiques de composants sont les processus, serveurs, systèmes de fichiers, bases de données, flots de données, filtres, *etc.*

Les ports permettent aux composants d'exposer leurs fonctionnalités. Chaque port identifie un point d'interaction entre le composant et son environnement. Un port est utilisé pour représenter ce qui est traditionnellement considéré comme une interface, c'est-à-dire un ensemble d'opérations disponibles et fournies pour le composant. Un port peut représenter une interface aussi simple qu'une signature de procédure et ou une interface plus complexe comme une diffusion multicast d'événements.

Les connecteurs représentent les interactions entre les composants. Ils servent d'intermédiaire pour la communication et la coordination des activités. Des exemples simples d'interaction sont les pipes, appels de procédures, diffusion d'événements... Mais des interactions plus complexes peuvent être représentées, comme une requête SQL entre une base de données et une application.

Les rôles définissent les interfaces des connecteurs. Chaque rôle du connecteur décrit les participants de l'interaction. La plupart des connecteurs sont binaires, ils possèdent deux rôles comme appelé et appelant d'une communication RPC, ou émetteur et récepteur pour un envoi de messages.

Les systèmes représentent les configurations composées d'éléments de l'architecture. Une description d'un système en Acme est un assemblage de composants et connecteurs, décrivant les entités de l'application et la nature de leurs interactions. Les systèmes permettent aussi de définir des propriétés de plus haut niveau, telles la tolérance aux fautes.

Les représentations permettent de décrire plus précisément un élément du système, en produisant une vue de cet élément. Une représentation se situe à deux niveaux distincts. Elle peut d'abord correspondre à une vue hiérarchique d'un élément (composant ou connecteur) en présentant le contenu de son sous-système. C'est alors un raffinement de cet élément, une représentation à un niveau de détail plus élevé.

Une représentation peut par ailleurs être une vue d'un système qui a été filtrée selon certains aspects. Par exemple, dans un système contenant des

connecteurs variés, une représentation particulière montrerait une seule sorte d'interaction, masquant les autres.

Les **rep-maps** définissent les correspondances entre la représentation interne d'un système et les interfaces des composants qui le constituent. Cette entité fournit simplement une association entre les ports internes et les ports externes d'un composant. Les rep-maps trouvent leur équivalence dans les liaisons de l'ADL Olan [BAKR95].

Annotations

Actuellement, de nombreuses discussions sont ouvertes pour savoir ce qui doit être ajouté aux informations relatives à l'architecture. Chaque langage possède son propre ensemble d'informations auxiliaires qui déterminent la sémantique à l'exécution, les types (*p. ex.* les types de données communiquées entre les composants), les protocoles d'interaction... Pour s'accomoder de cette grande variété d'informations complémentaires, Acme propose un système d'annotations basé sur des listes de propriétés. Chaque propriété a un nom, un type optionnel et une valeur. Les sept entités, présentées précédemment comme éléments de base, peuvent ainsi être annotées.

Pour Acme, ces propriétés sont des valeurs ou données non interprétées. Elles sont utiles seulement si un outil issu d'un autre ADL les utilise pour une manipulation, analyse, ou transformation. Acme prédéfinit des types simples comme les entiers, chaînes et booléens. Pour les autres types, des propriétés sont attachées indiquant par quel langage elles pourront être interprétées. Par conséquent, les outils utilisent les attributs *name* et *type* pour voir si la propriété les concerne. Si l'outil ne comprend pas une propriété donnée, il la laisse non interprétée pour les autres outils.

Essayons maintenant d'illustrer ces caractéristiques avec notre application pilote. Reprenons notre architecture d'un client et son compte en banque, dialoguant par l'intermédiaire du connecteur RPC. Le concept **Titulaire** est déclaré comme n'ayant qu'un port **consultation** pour faire un envoi de requêtes et le **Compte** ne possède lui aussi qu'un seul port pour les recevoir. Le connecteur a, quant à lui, deux rôles désignés par **callee** et **caller**. La topologie du système est déclarée par une liste d'attachements entre les ports et rôles. Pour cet exemple, quelques propriétés ont été ajoutées. L'une d'elle indique comment spécifier le style des éléments (Aesop et UniCon). De même, la propriété du protocole RPC est déclarée être issue du langage Wright et sera seulement compréhensible pour un outil qui connaît ce langage. L'encapsulation du code des composants est introduit par l'attribut **source-code**.

```
System Banque = {  
  
Component Titulaire = {  
    Port consultation;  
    Properties{Aesop-style : style-id = client-server;  
               UniCon-style : style-id = cs;  
               source-code : external = "titulaire.c"}}}
```

```

Component Compte= {
  Port solde;
  Properties{max-concurrent-clients : integer = 1;
             source-code : external = "compte.c"}}

Connector rpc = {
  Roles{caller, callee};
  Properties{synchronous : boolean = true;
             max-roles : integer = 2;
             protocol : Wright = "..."}}

Attachments{
  Titulaire.consultation to rpc.caller;
  Compte.solde to rpc.callee}
}

```

Finalement, Acme fournit à la fois la base structurelle pour la caractérisation d'architectures, mais aussi des facilités d'annotations pour des informations supplémentaires concernant des ADL spécifiques. Cette méthode permet à un ensemble d'ADL de partager des informations comprises par tous, et tolérer la présence d'informations qui ne seront exploitées que par quelques uns d'entre eux.

2.2.2 Traduction d'une description d'architecture

Principe

Acme joue le rôle d'une représentation intermédiaire qui peut agir aux moyens d'outils variés de traduction. Une architecture d'un système donné est, de ce fait, partagée parmi tous les langages supportés par les traducteurs d'Acme. Il en découle que tous les outils développés dans le contexte d'Acme et tous ceux fournis avec des ADL compatibles sont alors applicables et disponibles pour cette architecture. Cela permet notamment d'élargir les propriétés et fonctionnalités de chacun de ces ADL. Actuellement, Acme permet de faire des correspondances entre UniCon, MétaH, Aesop, Rapide et Wright. D'autres langages sont sur le point d'être intégrés.

De même, n'importe quel langage peut être traduit, utilisant Acme comme pivot intermédiaire, vers un langage cible. Pour cela, trois étapes sont nécessaires. D'abord, le langage de description source est traduit en Acme, tout en conservant ses spécifications sous formes d'annotations dans son propre langage. Ensuite ces annotations sont traduites vers le langage de destination. Enfin, la dernière étape consiste à transcrire le langage Acme enrichi de propriétés vers le langage cible. Toutes les étapes sont plus ou moins directes. Toutefois faire un pont entre deux langages constitue la difficulté essentielle des étapes intermédiaires. Dans la partie suivante, un exemple sera développé soulignant les problèmes rencontrés.

Schéma d'intégration de Rapide et Wright⁵

Étant développeurs de Wright, les auteurs d'Acme ont voulu utiliser les avantages de Rapide pour les simulations et animations de descriptions d'architectures. Il est évident que réunir ces deux langages sera bénéfique puisqu'ils présentent tous deux des aspects complémentaires [GW98].

Des différences importantes entre Wright et Rapide soulèvent cependant des difficultés, à la fois au niveau de la structure des architectures et au niveau de la sémantique. Voici quelques problèmes qui se sont posés ainsi que leur solution :

- Une des difficultés est de savoir reproduire la définition de nouveaux connecteurs en Rapide, tel qu'il est possible de le faire en Wright. Il apparaît deux façons de gérer cette situation : limiter la classe de descriptions architecturales à celles utilisées par Rapide, ou représenter les connecteurs complexes comme composants et utiliser les connecteurs événementiels pour lier les différentes parties en Rapide. Cette dernière solution a été retenue afin de permettre d'élargir la classe de connecteurs en Rapide ;
- Faire correspondre le style fonctionnel de Wright au style impératif de Rapide a été réalisé grâce à des programmes standards de transformation ;
- Contrairement à Wright, Rapide impose des restrictions sur l'utilisation du parallélisme imbriqué dans ses descriptions. L'utilisation d'opérateurs algébriques CSP a permis d'éliminer le parallélisme imbriqué avant la traduction.

Cette liste est bien entendue non exhaustive et n'est présentée que pour fournir un aperçu des problèmes rencontrés. La figure 2.5 présente un environnement prototype dans lequel les deux langages sont intégrés en utilisant Acme comme langage intermédiaire.

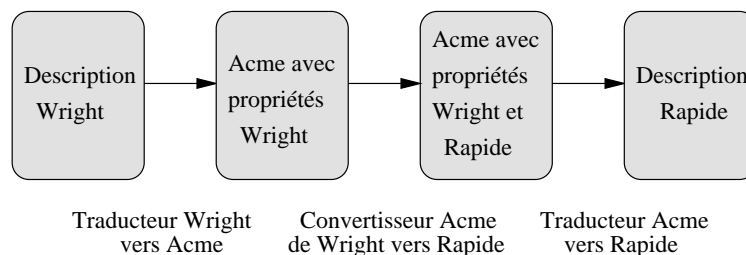


FIG. 2.5 – Traduction de Wright vers Rapide via Acme

La traduction de Wright vers Acme consiste en une correspondance des éléments Wright avec ceux d'Acme et d'un ajout de propriétés qui correspondent à la description du comportement de Wright. Puisque ces deux langages ont une vue similaire de la structure, cette étape est presque directe.

Dans la deuxième étape, les propriétés qui caractérisent le comportement de Wright sont transformées pour créer des propriétés caractérisant le comportement de Rapide. Pour faire ceci, des fragments de code Rapide doivent être créés à partir des annotations Wright stockées dans la représentation d'Acme. Ceci est réalisé par l'utilisation d'une table qui pour chaque construction Wright

⁵Ces deux langages sont décrits en section 4.2 et 4.1 respectivement.

établit la traduction quasi immédiate en spécifications Rapide.

Enfin, dans la dernière étape consiste à traduire les structures Acme annotées avec des fragments de Rapide en des structures natives Rapide. Le comportement du composant devient une implémentation de *Module* en Rapide.

2.2.3 Évolution dynamique

Il est important de reconnaître que même si Acme est basé sur l'idée de représentation d'une architecture comme une structure statique, cela ne signifie pas que les représentations sont seulement statiques et les architectures fixes. Acme peut être aussi utilisé comme moyen de représenter des architectures reconfigurables en exprimant des configurations possibles. Par exemple, un système peut inclure des propriétés qui décrivent les composants pouvant être ajoutés à l'exécution et la manière dont ils seront rattachés aux éléments de la configuration courante.

De plus, d'après les descriptions de Garlan et de son équipe Able, Acme peut aussi être utilisé pour représenter des configurations alternatives. Une description Acme peut être utilisée pour décrire un ensemble d'architectures, pas seulement une seule fixe. Par exemple, la description d'un composant en Acme peut inclure quelques implémentations différentes, dont une seule sera finalement dans l'implémentation de l'architecture.

Outre ces aspects, la traduction d'un langage entièrement statique vers un langage plus dynamique, tel Darwin (*cf.* section 3.1) ou encore mieux Rapide, permet d'ouvrir les portes vers une application plus évolutive. En d'autres termes, à toute application entièrement statique peuvent être ajoutées des propriétés dynamiques après traduction dans un langage adéquat.

2.2.4 Atouts

- De nombreux, voire la plupart des outils de conception et d'analyse d'architecture nécessitent une représentation pour décrire, enregistrer, manipuler les concepts ou abstractions d'une architecture. Malheureusement, développer de bonnes représentations est difficile, long et coûteux. Acme peut atténuer le coût et la difficulté de construction d'outils en fournissant un langage et une boîte à outils qui peuvent être utilisés comme base à de nouveaux outils de construction. Acme fournit une fondation solide, extensible et une infrastructure qui permet aux constructeurs d'éviter la reconstruction d'une infrastructure standard ;
- De plus, Acme, par son origine, est un langage d'échange générique, permettant à des outils développés sous ce langage d'avoir la capacité d'être compatible avec une grande variété d'autres langages avec peu ou pas d'effort supplémentaire. En utilisant un ensemble varié d'outils, les architectes gagneront plus de force d'expression.

2.2.5 Limites

- Les éléments peuvent être annotés avec des propriétés qui se présentent comme des informations descriptives mais ne fournit pas de modèle spé-

- cifique pour décrire le comportement d'un système ;
- Acme ne présente pas d'intégration complète des différents langages de description. De plus, certains types de systèmes décrits dans un ADL ne pourront pas être transcrits dans un autre, toutes les structures architecturales ne sont pas prévues. Cependant, plutôt qu'essayer de fournir une couverture complète, les auteurs de Wright essayent de trouver le schéma qui permette à un large sous-ensemble d'être accessible mutuellement dans un système intégré ;
 - En permettant la création de nouveaux ADL, Acme perd un peu son objectif qui était d'uniformiser les différents langages ;
 - Ce langage ne présente aucun aspect nouveau pour l'évolution et la dynamique des applications. Il s'appuie beaucoup sur la dynamique des autres ADL mais ne propose rien. Des recherches sont en cours et s'ouvrent vers des modèles sémantiques plus riches : contraintes logiques temporelles pour l'expression d'aspects dynamiques et familles de propriétés [GMW97].

2.2.6 Bilan

De nombreux langages de description d'architecture ont été développés, chacun fournissant des fonctionnalités complémentaires pour le développement et l'analyse des architectures. Malheureusement, chaque ADL et son ensemble d'outils associés opèrent chacun de son côté, rendant difficile les intégrations à ces outils et les partages de descriptions d'architectures. Acme résulte d'un effort joint de la communauté de recherche dans les architectures logicielles afin de définir un format d'échange commun pour les outils travaillant sur les architectures. Ce nouveau langage permet de combiner les capacités d'analyse de multiples langages de description d'architecture. Idéalement, il peut aussi servir comme base d'ADL génériques et analyses structurelles, permettant aux programmeurs d'outils de faire des analyses compatibles à de multiples ADL.

Toutefois, Acme manque de maturité et bien qu'il s'appuie sur des principes intéressants, il n'apporte pas de véritable solution quant à l'uniformité des ADL. Il sert plus de médiateur entre chaque langage et n'apporte aucune nouveauté point de vue structuration architecturale ou construction langagière.

Chapitre 3

Langages de configuration

Les langages de configuration fournissent un modèle de notation pour spécifier des configurations d'application. 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. La principale différence entre les langages de description d'architecture et les langages de configuration provient de leur vision différente du composant. Dans le deuxième modèle, le composant est considéré comme une entité instanciable : la description d'un composant au niveau du langage permet de créer de multiples instances d'un composant lors de l'exécution. La configuration contient, en plus, la spécification du schéma d'instanciation des composants et leur placement sur les différents sites du système. Dans ce chapitre, nous allons présenter deux langages de configuration. Chacun décrit une application qui va s'exécuter en environnement réparti. Ils ont pour cela introduit des éléments dynamiques, permettant de rester plus proche de l'exécution de l'application.

3.1 Darwin

Darwin [ADG98] est un langage de description d'architecture fournissant une notation simple pour spécifier la structure de systèmes distribués construits à partir de composants. Il supporte des compositions logicielles à travers la description de composants logiciels interconnectés. À partir de cette description, les composants peuvent être instanciés à des fins de former une architecture spécifique exécutable. Ce langage fournit une sémantique basée sur le π -calcul¹ et supporte la description d'une certaine dynamique de l'application en termes de schéma de création de composants logiciels en cours d'exécution.

La figure 3.1 illustre notre application bancaire en termes de composants et interconnexions décrits dans le langage Darwin. Dans un premier temps, nous ne nous intéresserons qu'à un seul **Titulaire** et un seul **Compte**. Ces composants sont contenus dans le composant **Banque** et sont liés au moyen d'une interconnexion. Le rond blanc représente un service requis et le noir un fourni. La notion de service sera introduite dans la prochaine section.

¹Nous ne détaillerons pas dans ce document cette sémantique. Se référer à [MK96, page 5].

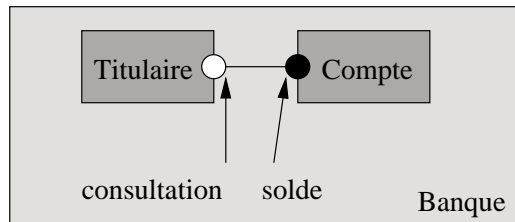


FIG. 3.1 – Présentation des composants de l'application bancaire en Darwin

3.1.1 Composants

Les principales abstractions gérées par Darwin sont les composants. Darwin les décrit en termes de services qu'ils fournissent et qu'ils requièrent pour leur permettre de communiquer avec d'autres composants. Deux types de composants existent : les primitifs et les composites.

Les composants primitifs

Les composants primitifs sont avant tout des entités d'encapsulation de fonctions et données d'un module logiciel. La description d'un composant est constituée de son nom et d'une interface qui déclare les services fournis et requis (*provide* ou *require*) par le composant. Voici la description des composants *Compte* et *Titulaire* dans le langage Darwin :

```

component Compte{
  provide solde <port, int>;
}
component Titulaire {
  require consultation <port, int>;
}

```

À partir de ce modèle décrivant sa nature et ses services, le composant primitif sera représenté à l'exécution par une classe C++. Celle-ci hérite d'une autre classe *process*. Elle est issue des classes de base de Régis [MDK94], support d'exécution réparti de Darwin.

Les composants composites

Un composant composite, lui, est construit à partir de composants primitifs et d'autres composites. Son utilisation sert, en tout premier lieu, à décrire la composition hiérarchique de l'application. Ainsi, la structure finale du système est représentée par un composant composite. Ces composites peuvent être alors considérés comme des entités de configuration puisqu'ils contiennent la description complète de l'application. À l'exécution, ces composites sont représentés par un ensemble d'instances de composants s'exécutant en concurrence.

La description d'un composant composite dans le langage est constituée de

son nom suivi d'une liste de paramètres typés². La portée de ces paramètres est l'implémentation du composant uniquement. Il est alors possible d'utiliser la valeur des paramètres à l'intérieur du composite pour décrire, par exemple, le nombre d'instances de composants à créer³. Dans la description de ce composant apparaissent aussi les services requis ou fournis. Vient ensuite l'implémentation du composant composite, vide dans le cas de composants primitifs (**Titulaire** et **Compte**), mais qui pour les composites (**Banque**) est constituée de déclarations d'instances et de schémas d'interconnexion. 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 à l'aide du constructeur `--`.

```

component Banque{
  inst
    titulaire : Titulaire ;
    compte : Compte ;
  bind
    titulaire.consultation--compte.solde ;
}

```

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

```
inst titulaire : Titulaire@2
```

Les services des composants

Les composants interagissent au moyen de services. La notion de connecteurs n'apparaît pas dans ce langage. En effet, chaque interaction est représentée par un lien entre un service requis et un service fourni entre des composants différents. La notion de service s'apparente ici plus à des flots de communication qu'à la notion de fonctions que l'on retrouve dans Wright (*cf.* section 4.1). Les services n'ont aucune 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 Régis qui permet à des configurations Darwin de s'exécuter.

Le type des services est spécifié dans leur signature. L'usage du type **port** est le plus courant : il s'agit d'un objet envoyant des requêtes de manière synchrone ou asynchrone entre les composants répartis ou non. Darwin, par défaut, effectue un simple test d'équivalence des noms et des types, vérifiant que le service requis est compatible avec le fourni. Très peu de vérifications étant faites, nous pouvons considérer que l'aspect de typage dans Darwin n'est pas très bien développé.

²Le composant figurant dans cet exemple ne possède pas une telle liste.

³Dans la suite du document, un exemple en fait usage.

3.1.2 Opérateurs particuliers

Darwin fournit des constructeurs dans le langage qui permettent de définir les schémas d'instanciations de composant plus complexes, dans le sens où des structures plus évoluées peuvent être définies (*p. ex.* des compteurs). Pour présenter ces différents opérateurs, nous allons nous appuyer sur notre exemple en l'étendant un peu. Considérons pour cela un nombre n de clients ayant chacun un compte qu'ils veulent consulter, sauf le dernier client qui partage son compte avec son conjoint. Une représentation graphique est proposée avec la figure 3.2.

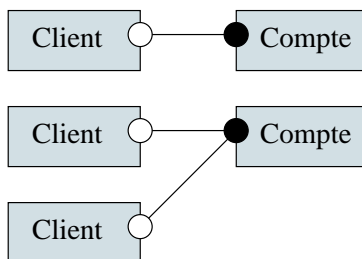


FIG. 3.2 – Représentation graphique de la configuration

L'opérateur `forall` permet de faire des itérations sur des variables entières. Sa sémantique est équivalente à un « pour tout ». Un autre opérateur `when` permet d'évaluer une condition et de réagir selon cette évaluation.

Dans l'exemple ci-dessous, nous utilisons les mêmes composants `Client` et `Compte` que ceux définis précédemment. Le nombre de comptes n'est pas connu au démarrage de l'application, il n'est fixé, par le paramètre n , que lors de la création du composant composite `Banque`. Le code associé à cette application est le suivant :

```
component Banque(int n) {  
  // Interface vide  
  
  // Implémentation  
  // Définition d'un ensemble d'instances de  
  // clients et comptes en banque  
  array : Cl[n] : Client ;  
  array : Cpt[n] : Compte ;  
  forall k : 0..n  
    inst Cl[k] ;  
    Cpt[k] ;  
  bind Cl[k].consultation--Cpt[k].solde ;  
  when k=n-1  
    bind Cl[k].consultation-- Cpt[k-1].solde ;  
}
```

Deux nouveaux opérateurs sont utilisés ici. `forall` permet de déclarer un tableau de clients et un tableau de comptes et faire des interconnexions entre leurs composants. Le constructeur conditionnel `when` permet d’effectuer des déclarations de création d’instances et d’interconnexions selon la validité de la condition.

3.1.3 Évolution dynamique

Darwin apporte une amélioration notable par rapport à des langages comme UniCon puisqu’il autorise la spécification des instants et des emplacements de création des composants et interactions d’une application plutôt que de considérer le système comme étant fixe par rapport à sa phase de conception. Il fournit deux principaux mécanismes pour la description de structures dynamiques : l’instanciation paresseuse et dynamique.

Instanciation paresseuse

La première de ces instanciations est l’instanciation paresseuse dans laquelle un composant fournissant un service n’est pas instancié tant que l’utilisateur de l’application n’essaie pas d’accéder à ce service.

```
component Banque{
  inst
    titulaire : Titulaire;
    compte : dyn Compte;
  bind
    titulaire.consultation--compte.solde;
}
```

Les instances non dynamiques sont créées normalement lors de l’instance du composant composite. Seules les instances déclarées dynamiques sont effectuées quand le service qu’elles fournissent est sollicité. Initialement, un seul `titulaire` est instancié. L’instance du `compte` n’est pas directe puisqu’elle est déclarée comme *lazy* par le mot clé `dyn`. Quand le `compte` est appelé pour la première fois par son service fourni par l’expression `compte.solde`, l’instanciation est déclenchée. L’instanciation paresseuse permet donc, par une spécification de structures dynamiques, de décrire la configuration potentielle à l’exécution en déclarant des instances qui seront peut-être créées par un appel à leur service. Ce type d’instanciation permet notamment de décrire des structures dont la taille ne peut être déterminée que dynamiquement. Malheureusement, ce mécanisme ne permet d’instancier qu’un seul composant par clause d’interconnexion, contrairement à l’instanciation dynamique que nous détaillerons par la suite.

La combinaison de l’instanciation paresseuse et de la récursivité peut être utilisée pour décrire une grande variété de structures apparaissant couramment dans les programmes parallèles (recherche dans un arbre, diviser pour conquérir). Elle peut aussi être trouvée, par exemple, dans les systèmes distribués où de nombreuses sauvegardes doivent être effectuées et les serveurs de stockage de

données sont créés à la demande. Nous ne détaillerons pas plus le principe de récursivité.

Instanciation dynamique directe

L'instanciation paresseuse ne pouvant pas toujours être utilisée, Darwin propose alors le concept d'instanciation dynamique directe.

Dans cet exemple, un client de la banque réalise la création dynamique d'un nouveau compte. La déclaration de cette instanciation dynamique est réalisée dans la clause d'interconnexion `titulaire.creation--dyn Compte`. Pour cela, un service requis particulier (`creation`) du composant initiateur (`titulaire`) doit exister. Dès que le composant de la partie gauche (`titulaire`) demande une création au travers un service, le composant de la partie droite (`Compte`) est créé. Cette interconnexion ne sert à rien d'autre qu'à la création du composant, aucun appel de service n'est effectué.

```
component Banque{
  inst
    titulaire : Titulaire
  bind
    titulaire.creation--dyn Compte ;
}
```

Il est à noter que le lien est spécifié pour le type du composant à instancier (`Compte`) plutôt que pour une instance de ce type. L'instanciation dynamique directe, par ce fait, permet de multiples instanciations au moyen d'une seule clause. Le problème majeur de cette technique est l'inaccessibilité des composants une fois qu'ils sont créés dynamiquement. En d'autres termes, les composants qui sont créés statiquement ne peuvent pas communiquer avec ceux qui sont instanciés dynamiquement. L'inverse est cependant possible. Il ne serait donc pas possible avec Darwin de déclarer des liens vers des services que les composants dynamiques fournissent. Nous voyons ici les limites de la dynamique. Un moyen de sélection dynamique du compte serait ici nécessaire.

3.1.4 Système à l'exécution

Nous rappelons que Régis [MDK94] est le support d'exécution de Darwin permettant à des configurations de s'exécuter.

Le langage de programmation des composants est le même que celui utilisé dans Régis. Les composants primitifs sont des classes C++ qui héritent d'une classe de base de Régis, la classe `process`. A l'intérieur des composants, il est fait directement référence aux ports de communication requis ou fournis. Ces ports sont aussi des classes C++ fournies par Régis. Regardons maintenant le code associé au composant `Client` :

```
class Client : public process {
  public :
```

```

    portref<int> consultation;
    Client();
}

Client : :Client(){
    int numeroCompte;
    int montant;
    string nom;

    while(1){
        consultation.send(numeroCompte, &montant);
    }
}

```

Notons que le programmeur doit obligatoirement utiliser les objets de type `port` pour réaliser les communications et effectuer les créations de composants lorsqu'il les sollicite.

3.1.5 Atouts

Parmi les avantages de Darwin, nous pouvons noter :

- Tout d'abord, Darwin a la capacité de construire une configuration dynamique. En effet, ce langage présente deux constructions qui permettent de décrire un schéma d'instanciation dynamique de composants et de relier ces instances au reste du système par un moyen de communication. Chacune de ces constructions permet de faire une spécification explicite de l'architecture dynamique de l'application dans des contextes d'utilisation très différents, offrant une certaine flexibilité à l'architecte ;
- Des constructeurs du langage, comme *forall* et *when*, alliés à la récursivité, permettent de spécifier des configurations complexes et paramétrables. Il nous semble que ces opérateurs sont appropriés lorsque la topologie des interconnexions ou la répartition sur les sites est connue à l'avance. L'utilisation de conditions et d'itérateurs permettent, par exemple, de distribuer aisément les composants sur des processeurs répartis sur un anneau ;
- L'utilisation des mécanismes de communication entre les composants co-localisés ou distribués est complètement transparente pour le programmeur de l'application. Le facteur de répartition des composants est pris en compte lors de chaque instanciation de composants. Le support d'exécution Régis, associé au langage, permet de gérer tous ces types de communication en fonction de la localisation des composants en offrant plusieurs mécanismes d'interaction.

3.1.6 Limites

- La description de la dynamique d'un ensemble de composants est limitée. 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, seuls les composants dynamiques peuvent communiquer avec l'extérieur. Une sélection dynamique des intervenants d'une communication dynamiquement créés n'a malheureusement pas été envisagée. Enfin, certaines structures architecturales, *p. ex.* les structures cycliques, ne peuvent pas être décrites. Insérer un nouveau composant dans l'anneau demanderait de rompre un lien existant, en supprimant une connexion. De plus, pour la réinsertion, il faudrait coordonner les composants afin d'éviter les effets indésirables ;

- Darwin ne fournit pas de base adéquate pour l'analyse du comportement d'une architecture. L'implémentation des composants sont des boîtes noires non interprétées. Les types de services dépendent de la plate-forme Régis et la sémantique est alors là-aussi non interprétée dans Darwin. Le modèle devrait fournir plus de moyens pour décrire des propriétés associées à un composant ou ses services. En outre, le typage dans Darwin n'est pas bien intégré. Une simple vérification du typage des services ne paraît pas suffisant ;
- Nous nous apercevons que l'encapsulation du code existant dans un composant n'est pas une opération immédiate car il faut utiliser de manière explicite les ports de communication de Régis pour interagir avec d'autres composants. Le code d'utilisation de l'environnement Régis/Darwin n'est donc pas transparent pour le programmeur. 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 le C++. Seule la désignation du composant destinataire de cette communication n'a pas à être effectuée en C++, c'est le langage de description qui s'en occupe ;
- Le support de Darwin pour les styles d'architecture est par ailleurs limité par sa faiblesse relative à la notion de connecteurs. Le modèle de connexion avec le principe fourni/requis construit un schéma asymétrique pour l'interaction. Il implique qu'il y ait toujours un parti dans une interaction qui soit responsable de la définition du protocole. Donc le schéma d'interaction peut être difficilement décrit indépendamment du composant qui le fournit. De plus, l'utilisation de divers modes de communication n'est pas configurable par le concepteur de l'architecture. Il faut que ce support d'exécution supporte les bons types de communication et il n'est malheureusement pas possible d'étendre la classe des types supportés. Un autre problème relatif à l'interaction des composants est à soulever. Il s'agit de la distinction entre les communications synchrones et asynchrones. Elle n'est pas faite au niveau du langage de description, cette possibilité est offerte au programmeur de composants qui utilise les ports Régis 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 des composants est un facteur important de réutilisation et de composition d'architectures.

3.1.7 Bilan

Darwin propose une vision de l'architecture claire et fonctionnelle. À travers l'utilisation de constructions conditionnelles et itératives, il autorise une distribution sur les sites plus aisée. En outre, l'utilisation de paramètres pour déterminer la structure du système à l'initialisation est possible. Mais l'apport principal de Darwin est de permettre des descriptions de structures dynamiques qui évoluent en cours d'exécution. Ce langage de description d'architecture ouvre la porte à la programmation du schéma de création de composants en fonction des besoins de l'application, de manière séparée du code du composant.

3.2 Olan

Olan [Bel97, BAKR95] est un environnement de programmation qui vise à faciliter le développement, la configuration et le déploiement d'applications réparties construites par assemblage de composants hétérogènes. Son langage de description d'architecture, OCL (Olan Configuration Language), repose sur un modèle d'assemblage de composants logiciels, dont la conception initiale a été inspirée de Darwin et de sa notion de composants interconnectés. Cependant, certaines limitations de ce langage ont été levées, avec entre autre, un début de spécification du comportement dynamique de l'application, et l'insertion de la notion de connecteur dans l'architecture. Nous allons présenter dans ce chapitre le langage de description OCL, ainsi que l'architecture et l'implémentation du système d'exécution Olan, dédié à l'installation et l'exécution des applications.

3.2.1 Modèle de composants

Olan permet de spécifier les 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. Le concept principal véhiculé par le modèle d'assemblage est le composant. Un composant englobe la définition d'aspects statiques (au travers d'une interface) et dynamiques (au travers d'une réalisation et d'une configuration). Les composants décrits en Olan dérivent d'une classe **Component**. Cette classe Java peut donner lieu à de multiples instances, créées dynamiquement lors de l'exécution de l'application. Deux sortes d'implémentation sont disponibles : les primitifs encapsulent directement les modules logiciels ou classes écrits dans des langages de programmation comme Java, alors que les composites contiennent les interconnexions de composants, qu'ils soient composites ou primitifs.

Interface

Les dépendances fonctionnelles du composant avec le monde extérieur sont exposées au niveau de l'interface. Celle-ci permet de décrire de manière complète l'accès aux opérations et aux structures de données fournies par le composant. Ce n'est pas son rôle unique, elle contient également les informations sur toutes les opérations requises par le composant en question, afin de garantir un fonctionnement correct. La terminologie associée à ces opérations est le *service* qui

contient les indications sur le rôle de l'opération et le mode de communication devant être utilisé pendant l'exécution. La sémantique exacte d'un service dépend de la manière dont il sera défini et accessible à l'exécution. Son type caractérise le middleware, ou support d'exécution, qui sera employé. Le tableau 3.1 détaille des types de service aujourd'hui existants dans OCL. Voici un exemple de service qui utilise le support A3⁴.

```

service CompteService : A3Service{
    notifications = aaa.CompteNotificationSet ;
    file = ${SRC_DIR}/aaa/CompteNotificationSet.java ;
}

```

Un service de type A3 spécifie toutes les notifications devant être gérées au sein du service. La notion de notification est proche de celle d'événement et représente des données significatives dont la diffusion peut entraîner l'exécution de réactions de la part de composants présents et intéressés par ces données. La localisation du fichier Java implémentant ces notifications est donnée par le chemin d'accès `file`. Ce fichier permet de faire la correspondance entre les services fournis et requis spécifiés dans l'interface, et l'implémentation de ces services. [BPF] présente plus précisément les différents types de service existants.

Support	Type de service	Description des opérations
AAA	A3Service	Type Java sérialisable
Corba IDL	IDLService	Type IDL de l'OMG
Java RMI	JRemoteService	Type Java sérialisable
Java rARI	ARIRemoteService	Type Java sérialisable

TAB. 3.1 – *Types de service d'Olan*

L'interface peut aussi contenir des définitions d'attributs publics. Les attributs sont des variables typées qui permettent de rendre accessibles au niveau du langage certaines données contenues initialement dans le code intégré et qui peuvent éventuellement changer en cours d'exécution. Certains attributs sont prédéfinis (tout composant possède par exemple un « site de chargement »), tandis que d'autres sont définis directement par le réalisateur. L'intérêt majeur de ces attributs est de permettre le nommage des composants. Voici l'exemple Olan de l'interface du composant Compte.

```

Interface CompteItf{
    attribute String personne ;
    attribute Location localisation ;
    provided CompteService solde ;
    provided CompteService init ;
}

```

⁴Une description succincte de ce support est fournie en section 5.3.1

Le compte en banque possède un attribut `personne` qui indique le nom de la personne propriétaire du compte, `localisation` indique le site d'exécution du composant. Les opérations `solde` et `init` sont des notifications implémentées au sein du fichier `CompteNotificationSet.java`.

Implémentation

Un type caractérise la nature de l'implémentation du composant. En fonction du type, l'interface et les propriétés doivent être soigneusement choisies. Le tableau montre les types disponibles ainsi que leurs services associés.

Type	Description	Services
A3Agent	Agent A3	A3Service
CORBAObject	Objet CORBA distribué	IDLService
RMIObject	Objet Java RMI	JRemoteService
ARIObject	Objet Java ARI	ARIRemoteService
CorbaAgentProxy	Passerelle entre objet CORBA et A3	A3 et IDLService

Deux sortes de composants sont identifiés : les composants primitifs et composites.

Composant primitif Un composant primitif est une entité logicielle encapsulant la mise en œuvre d'entités logicielles (classes, modules...) qui forment une application. La réalisation d'un composant primitif comprend son interface et une implémentation des entités programmées qu'il encapsule, écrites en Java ARI (modèle RMI en asynchrone), Java RMI, A3 ou CORBA. La syntaxe d'une implémentation est illustrée dans l'exemple suivant de la banque. Elle est identifiée par son nom, et indique le type dont le composant dérive et la localisation des fichiers à encapsuler.

```

Component Compte : A3Agent{
    A3Server location ;
    Interface CompteItf{
        attribute String personne ;
        attribute Location localisation ;
        provided CompteService solde ;
        provided CompteService init ;
    }
    Implementation{
        agent = aaa.Compte ;
        file = ${SRC_DIR}/aaa/Compte.java ;
    }
}

```

La mise en œuvre du composant dans le fichier `Compte.java` permet de faire une liaison explicite entre les services déclarés dans l'interface et ceux définis dans le code du composant.

Composant composite Une application est constituée de composants primitifs mais aussi de composants composites qui servent à la fois d'entités de description de la configuration et d'entités de structuration d'une application en composants coopérants. Les composites permettent de former une hiérarchie de composants, hiérarchie partiellement ou totalement réutilisable dans diverses applications. Le concept d'application d'OCL est similaire à Darwin, il s'agit d'un composant composite placé au sommet de la hiérarchie.

Chaque composite est formé d'une interface identique à celle des primitifs. De la même façon, un composite est composé d'une mise en œuvre (*configuration*) qui contient la déclaration des sous-composants nécessaires avec éventuellement des paramètres de création de ces composants, ainsi que les interconnexions entre eux. Ce schéma est essentiellement statique, car tous les sous-composants sont créés à un seul instant, lors de la création du composant composite qui les englobe. L'utilisation du constructeur `instance` permet de déclarer les sous-composants à instancier. Un composite présente en plus les interconnexions entre eux. La figure 3.3 illustre le composite `Application` de l'application bancaire.

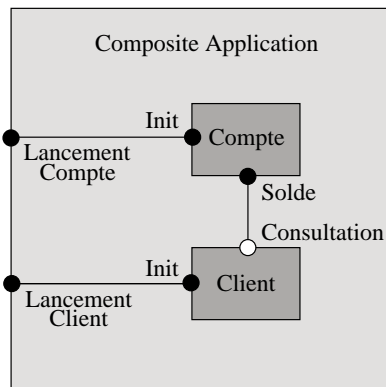


FIG. 3.3 – Représentation Olan du composite `Application`

Les lignes horizontales et verticales représentent les connecteurs dont la description est établie dans la prochaine section. Voici le code associé à cette figure :

```
Component class Application{
interface
    provided appService lancementCompte ;
    provided appService lancementClient ;
configuration
    Compte compte ;
    Client client ;
//Instanciation des sous-composants
    compte = new Compte(localisation = caroline.inrialpes.fr) ;
    client = new Client(localisation = dyade.inrialpes.fr) ;
//Connexions
    lancementCompte => compte.init ;
    lancementClient => client.init ;
```

```

    client.consultation => compte.solde using A3bus ;
}

```

Nous pouvons noter à ce niveau la structuration du composant composite. Il exhibe au niveau de l'interface deux services fournis pour configurer et lancer les composants `Compte` et `Client`, qui seront instanciés dans la partie `configuration`. Au cours de cette instanciation, l'attribut `localisation` est fixé, afin de définir le site d'exécution de ces composants.

3.2.2 Modèle d'interactions

Connecteur

Les connecteurs sont des objets qui contrôlent les interconnexions de composants et spécifient le protocole requis à l'exécution, identifiant le médium utilisé pour les communications locales ou distantes. Similairement aux notions de rôles introduits dans UniCon (*cf.* chapitre 2.1), chaque connecteur Olan spécifie l'ensemble de composants dont il accepte la connexion. Des mots-clé identifient le type de service (`in` pour les services d'entrée et `out` pour ceux de sortie) attendu par le connecteur.

Actuellement, il n'existe pas de langage ou de formalisme, comme dans [AG94b], pour exprimer les connecteurs. Ils sont fournis par le support d'exécution et sont utilisables directement dans le langage Olan. Quelques connecteurs prédéfinis (`builtin`) dans l'environnement Olan sont proposés mais, dans la version actuelle, il n'est pas permis d'intégrer des connecteurs propres à l'utilisateur :

```

connector A3bus{
    in A3Service input ;
    out A3Service output ;
    implementation {builtin}
}

connector CORBAbus{
    in CORBAService <A> input ;
    out CORBAService <A> output ;
    implementation {builtin}
}

```

Le premier type de connecteur est utilisé pour la communication au sein des `A3Service`, et le second autorise des interactions entre services CORBA. L'utilisation d'une entité distincte pour la gestion des communications au niveau de l'ADL, séparée du code des composants, favorise de nombreuses vérifications. La validité d'une interconnexion est effectivement vérifiée par les caractéristiques de chaque connecteur (services imposés, signatures compatibles). Par exemple, les services connectés aux extrémités du connecteur (`in` et `out`) doivent être de même type (identifié par `<A>` pour les connecteurs CORBA).

Nous pouvons remarquer dès à présent que la répartition n'entre pas en ligne de compte dans le choix du connecteur. Un connecteur définit seulement les propriétés de communication.

Connexion

Les connexions sont les communications effectives qui prennent place entre les composants. Elles peuvent être vues comme des instances d'un type de connecteur avec une implémentation spécifique dont les sources et destinations sont spécifiées. Une interconnexion est plus précisément l'action de spécifier avec qui et comment les composants communiquent pendant l'exécution. Chaque interaction contient l'initiateur de la communication et le ou les destinataires. Une connexion ne décrit jamais le moment exact de la communication. L'interconnexion a pour unique rôle de dire que dès lors l'initiateur effectue une demande de communication, celle-ci sera effectuée en utilisant tel mécanisme ou protocole pour envoyer une requête vers les services destinataires. Une interconnexion met en jeu deux parties : une partie gauche contenant le service du composant initiateur d'une communication, la partie droite, les services des composants destinataires. La clause `using` d'une interconnexion indique le type de connecteur utilisé.

Reprenons l'exemple du composant composite `Application`. Son implémentation contient la définition d'une interconnexion entre les sous-composants `client` et `compte` :

```
client.consultation => compte.solde using A3bus ;
```

Dans cette clause, le client est connecté au composant `compte` par le biais d'un connecteur de type `A3`. Les services de type `A3Service` reliés étant `provided` et `required`, l'interconnexion est bien valide.

Liaison

Les liens définissent les correspondances entre les services du composite englobant, et les services de ses sous-composants définis à l'intérieur du composite. Ils correspondent aux traits horizontaux de la figure 3.3. Ces liens mettent en avant la manière dont le composant composite agit en tant que capsule pour sa configuration interne.

```
lancementCompte => compte.init ;  
lancementClient => client.init ;
```

Dans notre illustration, deux liens entre services fournis sont établis. La syntaxe de ces liaisons est assez similaire aux connexions, si ce n'est que le nommage des services du composite n'est pas préfixé par le nom de ce composant.

3.2.3 Évolution dynamique

Des aspects dynamiques de la configuration d'une application peuvent être exprimés dans un programme OCL. Deux types d'instanciation dynamique existent : l'instanciation paresseuse et dynamique. Ce sont des opérations comparables à celles qui ont été présentées dans Darwin. Nous faisons dans cette section un rapide rappel de leurs caractéristiques et présentons un nouveau schéma d'instanciation dynamique, introduit par les auteurs d'OCL. Ce concept offre à l'architecte de l'application le contrôle d'un ensemble de composants ayant la même interface.

Instanciation paresseuse

L'instanciation paresseuse permet de déclarer des composants qui ne seront pas tout de suite instanciés. Ces composants sont créés lorsque le service qu'ils fournissent est sollicité. Ce type de mécanisme ne permet d'instancier qu'un seul composant par interconnexion, contrairement à l'instanciation dynamique.

Instanciation dynamique

En opposition à l'instanciation paresseuse, de multiples instances peuvent être effectuées dynamiquement à partir d'une seule clause d'interconnexion. Pour cela un service requis particulier doit exister. L'interconnexion ne sert qu'à l'instanciation dynamique, aucun appel de service n'est effectué.

Collection

Les collections, ou les groupes, sont des ensembles dynamiques de composants identiques. La désignation des composants peut prendre deux formes. Tout d'abord, les membres de la collection peuvent être dynamiquement déterminés par le fait qu'ils satisfont certaines propriétés (un composant récepteur d'un message, un composant soumis à certaines règles d'administration...), les composants sont alors gérés implicitement. Une deuxième forme de désignation consiste à percevoir les composants comme une seule entité et la désignation concerne donc tous les membres de la collection. La clause suivante définit une collection de comptes en banque.

```
collection = Collection[0..n] of Compte ;
```

Les collections ont été introduites pour faciliter l'utilisation de multiples composants de même type. Son nombre à l'exécution repose sur un intervalle défini par un nombre minimum et maximum. Cette cardinalité est passée en paramètre au constructeur de la collection, au moment initial. À sa création, la collection est vide. On spécifie, par cet intervalle de valeurs, le nombre d'instances de composants que la collection pourra contenir à l'exécution. Deux opérateurs particuliers permettent d'ajouter ou de supprimer des composants d'une collection en cours d'exécution, ce sont **new** et **delete**, qui sont appliqués à un

identificateur.

```
collection.new(idf) ;  
collection.delete(idf) ;
```

De plus, un mécanisme spécial peut effectuer la création d'un nouveau composant de la collection avant d'accéder à un service spécifique. Quand l'initiateur demande à l'application de se lancer, une première instance du composant est créée et son service `init` est activé.

```
initiateur.lancement => collection.init using createInCollection ;
```

Enfin, pour faire un appel de service à distance du client vers un serveur de la collection, une méthode de nommage par association est utilisée, basée sur la vérification d'attributs existants à l'exécution. Le code suivant montre une utilisation possible.

```
client.envoiRequete(requete, idf) =>  
    collection.receptionRequete(requete)  
    where collection.idf = idf using methodCall ;
```

Ainsi, l'atout principal des collections et de créer des ensembles de composants ayant les mêmes fonctions mais dont le nombre varie. Cela autorise la manipulation d'entités qui peuvent évoluer dynamiquement et qui sont accessibles à travers une seule interface.

3.2.4 Système à l'exécution

Les supports d'exécution d'Olan sont des plate-formes standards, possédant un service de déploiement. Parmi ces plate-formes, nous comptons A3, CORBA, *etc.* Chaque composant primitif, en fonction de son type, doit être construit selon certaines conventions.

Par exemple, pour chaque attribut des composants A3 défini dans OCL, doit correspondre un attribut dans le fichier Java encapsulé, ainsi que deux méthodes pour y accéder, `setX()` et `getX()` :

```
en OCL :  
attribute string nom ;
```

```
en Java :  
protected String nom ;
```

```
String getNom(){  
    return nom ;  
}
```

```

void setNom(String value){
    nom=value ;
}

```

D'autres contraintes doivent être respectées, notamment pour les services et les implémentations.

3.2.5 Atouts

- La description que le composant primitif fournit au travers d'une interface est homogène et indépendante de l'entité encapsulée, de son langage de programmation ou de sa plate-forme d'exécution. Ils peuvent être utilisés dans n'importe quelle architecture à condition de satisfaire ses besoins fonctionnels et son modèle d'exécution. Ceci est une amélioration conséquente par rapport à UniCon où le langage de programmation n'était pas transparent au niveau des composants. De plus, OCL est plus souple en terme de définition de composants, son typage est moins contraignant. Il garde malgré tout la possibilité d'effectuer un grand nombre de vérifications sur l'architecture, en particulier avec les connecteurs et les règles d'interconnexion qui y sont attachées ;
- En outre, OCL apporte des spécifications pour la communication des composants par le biais de connecteurs, concept inexistant dans Darwin. Même si cette notion n'est pas encore bien définie, elle apporte déjà des réponses quant à la gestion des divers aspects de la communication, comme le protocole de communication utilisé entre les composants. De plus, la définition d'une entité de communication séparée de l'implémentation des composants permet de rendre la manipulation des communications plus souple, de vérifier la validité de la connexion et favorise la réutilisation de code ;
- OCL présente des différences importantes avec Darwin : le schéma d'instanciation dynamique, les interconnexions... En particulier, le langage offre le concept de collections qui remplace avantageusement l'instanciation dynamique de Darwin, car elle peut être contrôlable par un connecteur et non par un service de création particulier comme Darwin. Elle permet de créer comme de supprimer des composants et surtout elle permet l'accès par des composants clients aux composants contenus dans la collection, ce que ne permet pas l'instanciation dynamique de Darwin. Ainsi, concernant la gestion dynamique d'instances de composants, l'environnement Olan est assez complet. Néanmoins, il manque de nombreux opérateurs pour les reconfigurations dynamiques, notamment pour la suppression de composants ou les modifications d'attributs ;
- Enfin, Olan offre un environnement complet de construction d'applications. Toutes les étapes du cycle de vie d'une application distribuée sont représentées, incluant la construction, l'installation, le déploiement, la configuration du système. L'ADL est à l'origine de ce processus. Il permet de définir une interface homogène permettant d'exprimer l'hétérogénéité du modèle d'exécution et du langage de programmation du composant encapsulé.

3.2.6 Limites

- Les inconvénients de l’approche Olan sont essentiellement liés au travail de création des composants. Il faut écrire les interfaces, les implémentations primitives dans un langage de programmation, et décrire l’application, les composites et primitifs, avec OCL. Ce travail est donc relativement long par rapport à la description Darwin ou UniCon ;
- Dans l’interface d’un composant est défini le type de service qu’il possède, définissant par la même le support d’exécution que le composant utilise (*cf.* tableau 3.1). Cette information est suffisamment pertinente pour anticiper le modèle de communication nécessaire. On aurait pu imaginer une correspondance directe entre le type de composant et le type de connecteur, évitant à l’architecte de préciser quel connecteur utiliser dans les clauses d’interconnexion.

3.2.7 Bilan

Olan est un langage qui permet aux applications d’être exprimées comme une vue hiérarchique de composants interagissants. Le modèle architectural est basé sur celui de Darwin [MDEK95], étendu à la notion de connecteurs et aux collections. Cet environnement est idéal pour le développement d’applications réparties puisqu’il génère, à partir d’intégrations de morceaux logiciels hétérogènes, une application déployable.

Chapitre 4

Langages de description dynamiques

L'utilisation d'un langage de description d'architecture est un moyen pratique pour décrire la dynamique d'une application. Il est d'autant plus adapté pour des systèmes répartis où la configuration de l'application peut être reconfigurée en fonction de la charge des sites à l'exécution. L'évolution de l'application est de plus un processus grandement facilité par un langage dynamique puisqu'il suffit de spécifier les modifications à apporter à l'architecture initiale. Malheureusement, les deux langages étudiés dans ce cadre n'offrent pas la possibilité de mener une application à l'exécution. Il est tout de même intéressant d'étudier leurs mécanismes de gestion de la dynamicité.

4.1 Wright

Wright est un langage de description orienté vers la vérification des protocoles entre les composants, plutôt que sur la correction fonctionnelle de l'architecture globale [ADG98]. Ce langage n'est pas dédié à la production d'une image exécutable de l'application, il porte effectivement son accent sur la vérification formelle du système.

Allen et Garlan, ses auteurs, opposent deux styles d'architecture [AG94b] : celle dite d'implantation, qui est basée sur l'inclusion de composants, appels de procédure et décomposition hiérarchique des fonctionnalités d'un système, comme le langage Darwin, et celle dite d'interaction qui s'oriente vers les interconnexions des composants. Wright se présente comme cette dernière vision architecturale puisqu'il privilégie les interactions entre les entités de l'architecture. Ainsi, il s'éloigne des langages de programmation traditionnels où les appels de procédures et partages de données sont courants et propose d'exprimer d'autres notions architecturales plus complexes telles les systèmes à base d'événements, les structures de flots de données et protocoles client/serveur.

4.1.1 Modèle et notations

Nous ferons une présentation un peu plus exhaustive du modèle Wright puisque son principe de définition des composants et des connecteurs diffère sensiblement des langages étudiés précédemment. Wright modélise des structures architecturales en utilisant des composants, connecteurs et configurations, chaque abstraction étant définie à l'aide d'un calcul proche de CSP [Hoa85]. CSP décrit le comportement d'une architecture logicielle à travers un modèle algébrique de processus. L'utilisation de ce calcul permet de vérifier la faisabilité d'une interconnexion entre les composants et les connecteurs. Des notions de CSP sont présentées en Annexe A.

Composants

Les composants intègrent du code logiciel. Par exemple, notre titulaire d'un compte en banque émet des requêtes pour consulter son compte et reçoit en retour son solde. Dans Wright, la description d'un composant possède deux parties, l'interface et la partie `COMPUTATION`. Une interface est constituée d'un ensemble de ports. Chaque port représente une interaction dans laquelle le composant participe. Dans notre exemple, le client possède un seul port, celui pour consulter son solde. La spécification d'un port est une description partielle du composant [All97], une projection du comportement du composant sur ce point particulier de l'interface. La spécification complète du composant se trouve dans `COMPUTATION`, lieu d'analyses des propriétés. La partie `COMPUTATION` d'une description décrit le véritable comportement du composant. Elle reprend les interactions décrites au sein des ports et montre les relations entre les ports ainsi que leur lien avec les actions internes du composant. Ci-dessous se trouve un exemple de description du composant `Titulaire` inspiré de [AG97] :

```
COMPONENT Titulaire
  PORT consultation = requete? x → reponse! y →
  consultation  $\sqcap$   $\surd$ 
  COMPUTATION = calculInterne → consultation.requete? x →
  consultation.reponse! y → COMPUTATION  $\sqcap$   $\surd$ 
```

Le port `consultation` figure comme un point logique¹ d'interaction entre le composant `Titulaire` et son environnement. Il exprime le comportement que le composant fournira à travers ce point particulier de l'interface à son environnement. Ici, il souhaite pouvoir renouveler les actions « émission d'une requête `x` » et « réception du résultat `y` » ou terminer son processus quand il le désire (choix non déterministe \sqcap).

La partie `COMPUTATION` ressemble beaucoup au codage du port `consultation` puisqu'il est le seul port présenté par le composant. Le comportement observé à travers ce port est mis en relation avec le comportement interne du composant.

¹Les ports sont des entités logiques : il n'est pas nécessaire qu'un port réalise une fonction dans le système où il se trouve.

Les actions qui ne sont pas internes sont préfixées par le port où leur processus a lieu. Par exemple, **requete** et **reponse** sont préfixées par **consultation**.

Connecteurs

Un connecteur détermine les interactions parmi une collection de composants. Par exemple, un pipe représente un flot de données séquentiel entre deux filtres.

Une description Wright d'un connecteur est divisée en un ensemble de rôles et une spécification **GLUE**. Les rôles définissent le comportement local souhaité ou attendu pour les ports qui vont s'attacher aux extrémités du connecteur.

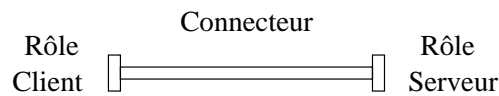


FIG. 4.1 – *Exemple de connecteur Wright*

Ainsi, le connecteur de la figure 4.1 admettra sur son rôle gauche la connexion d'un composant ayant le comportement d'un client et sur son rôle droit celui d'un serveur. Dans notre mise en scène, le client pourrait être le titulaire d'un compte en banque et le serveur pourrait être le compte lui-même. Les connecteurs sont en attente de propriétés particulières pour les participants potentiels à l'interaction. Cette attente est spécifiée au niveau des rôles. Ceux-ci définissent le protocole qui doit être spécifié par ceux qui s'y attacheront. En général, un port n'a pas besoin d'avoir le même comportement que le rôle qu'il remplit, il peut en effet utiliser seulement un sous-ensemble des capacités du connecteur. La **GLUE** d'un connecteur, quant à elle, décrit comment les participants interagissent au cours de cette communication. Comme pour **COMPUTATION**, elle relie les différentes actions comprises dans son interface.

CONNECTOR Connecteur

ROLE client = **requete!** x → **reponse?** y → client □ ✓

ROLE serveur = **demande?** x → **resultat!** y → serveur □ ✓

GLUE = (client.**requete?** x → serveur.**demande!** x →
serveur.**resultat?** y → client.**reponse!** y → **GLUE**) □ ✓

Le rôle du **client** décrit le comportement de l'utilisateur du service dans la communication. C'est un processus qui peut répétitivement appeler le service et recevoir le résultat, ou terminer. L'utilisation de l'opérateur décisionnel □ représente un choix laissé au rôle lui-même.

Similairement au **client**, le **serveur** est défini comme un processus qui, soit accepte répétitivement une invocation, retournant par la suite une réponse, soit peut terminer avec succès. À cause de l'utilisation de l'opérateur déterministe □, le choix est effectué par l'environnement de ce rôle (qui se résume à la glu et aux autres rôles). En comparant les deux opérateurs de choix, on peut s'apercevoir qu'une distinction formelle est faite entre les situations où un rôle

donné est obligé de fournir des services (le cas du **serveur**) et la situation où il peut profiter s'il le désire de services (**client**).

Le processus **GLUE** coordonne le comportement entre les deux rôles en indiquant comment tous les événements interagissent ensemble. La glu décrit comment les activités des rôles **client** et **serveur** sont coordonnées et présente leur séquençement. Le lecteur pourra noter que les événements qui sont en entrée (*resp.* sortie) au niveau des rôles (**reponse? y**) se trouvent en sortie (*resp.* entrée) dans la glu (**client.reponse! y**). Ceci s'explique aisément en se ramenant aux spécifications des deux parties d'un connecteur. Un rôle décrit le comportement d'un composant qui s'y attache alors que la glu s'intéresse au comportement du connecteur.

Configuration

Pour décrire l'architecture complète d'un système, les composants et connecteurs d'une description Wright sont combinés dans une configuration. Une configuration est une collection d'instances de composants reliés au moyen de connecteurs. Dans l'exemple suivant, des styles sont utilisés. Un style architectural permet la réutilisation d'un ensemble de types et de règles pour la description d'une architecture. Plus précisément, ils déterminent un vocabulaire à base de types de composants et connecteurs et spécifient les contraintes topologiques. Ces contraintes définissent un ensemble de prédicats que chaque configuration, conforme à ce style, doit satisfaire. Ici, les définitions précédentes des composants et du connecteur sont réutilisées. Les contraintes topologiques spécifient qu'un attachement entre toutes les instances des types **Compte** et **Titulaire** doit exister.

```

STYLE Titulaire-Compte
  COMPONENT Titulaire    // Définition du Titulaire
  COMPONENT Compte      // Définition du Compte
  CONNECTOR Connecteur  // Définition de Connecteur
  CONSTRAINTS
     $\exists !c \in \text{Component}, \forall t \in \text{Component} : \text{TypeCompte}(c) \wedge$ 
     $\text{TypeTitulaire}(t) \Rightarrow \text{connected}(c, t)$ 
END STYLE

```

La configuration du système est décrite en trois étapes. La première est la définition d'une ensemble de types de composants et connecteurs. Cette étape fait appel au style **Titulaire-Compte** prédéfini. Ensuite, un ensemble d'instances de ces types est déclaré, représentant les entités qui apparaîtront réellement dans la configuration. Il figure une seule instance **titulaire** dans cette illustration, ainsi qu'une seule instance **compte** et un connecteur. Dans la phase finale, les instances de composants et connecteurs sont combinées pour décrire quels sont les ports de composants attachés aux rôles des connecteurs. La consultation du **compte** par le **titulaire** est reliée au rôle **client** du connecteur. De même,

le port `solde` est attaché au rôle `serveur`. Ainsi, le prédicat exprimé dans les contraintes du style `Titulaire-Compte` est vérifié. Le code suivant montre comment notre simple système de banque peut être défini dans la description Wright.

```

CONFIGURATION Banque
STYLE Titulaire-Compte
INSTANCES
  titulaire : Titulaire
  compte : Compte
  connecteur : Connecteur
ATTACHEMENTS
  titulaire.consultation as connecteur.client ;
  compte.solde as connecteur.server ;
END Banque

```

4.1.2 Évolution dynamique

En guise d'illustration de la dynamicité de Wright, reprenons le style décrit précédemment. Pour cela, nous allons considérer maintenant une nouvelle configuration dans laquelle deux serveurs interviennent (*cf.* figure 4.2), représentant les comptes en banque : un serveur primaire qui peut tomber en panne à tout moment et un secondaire, répliqué, qui sert de serveur de secours. Initialement, le connecteur relie un serveur primaire et un titulaire de compte. Dès que le serveur primaire tombe en panne, la connexion est défaite. Elle est ensuite rétablie vers le serveur secondaire jusqu'à la restauration du primaire. Pour simuler cette situation, des modifications sont apportées dans la description des composants et connecteurs à l'intérieur du style.

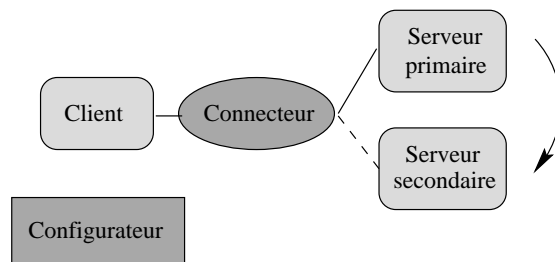


FIG. 4.2 – Deux configurations possibles pour la topologie dynamique

D'abord, des événements spéciaux de contrôle sont introduits dans l'alphabet des composants et connecteurs. Ces événements de contrôle sont utilisés dans une vue séparée de l'architecture, le programme de configuration. Ce dernier fait la correspondance entre les événements émis et les reconfigurations à effectuer sur l'architecture. L'idée de base est d'exprimer, au niveau des abstractions (composants et connecteurs), le contexte dans lequel elles admettent des

reconfigurations ; la gestion de ces reconfigurations, elle, est effectuée au sein du contrôleur.

Dans la description suivante, le compte primaire indique dans quel état il est susceptible de tomber en panne avec `control.down` et se rétablir avec `control.up`. Effectivement, il peut s'interrompre à tout moment sauf au cours d'une transaction (entre `demande` et `retour`) et ne peut se rétablir qu'après un arrêt. Le connecteur peut, lui, être reconfiguré (`changeOk`) au cours d'une transaction (entre `request` et `reply`).

```

COMPONENT ComptePrimaire
  PORT p = ✓ □ (demande? x → resultat! y → p □ control.down
    → (✓ □ control.up → p))
  COMPUTATION = ✓ □ (p.demande? x → calculInterne →
    p.resultat! y → COMPUTATION □ control.down →
    (✓ □ control.up → COMPUTATION))

CONNECTOR Connecteur
  ROLE client = requete! x → reponse? y → client □ ✓
  ROLE serveur = (demande? x → resultat! y
    → serveur □ control.changeOk → serveur) □ ✓
  GLUE = (client.requete? x → serveur.demande! x
    → GLUE
    □ serveur.resultat? y → client.reponse! y → GLUE)
    □ ✓
    □ control.changeOk → GLUE

```

Dans ce programme de reconfiguration, le type du client est identique à celui utilisé précédemment. Le connecteur est modifié, ainsi que les comptes². Ce nouveau style est donc utilisé dans le configurateur. Des modifications de la topologie de l'architecture seront effectuées en appliquant des actions de base comme `new`, `del`, `attach` et `detach` sur les instances de types architecturaux contenus dans le style.

Au début de la description du configurateur, le style utilisé est déclaré. Ensuite, la séquence initiale d'actions (`new` et `attach`) modélise la configuration initiale du système. **Arrêt** décrit deux situations possibles : le système peut marcher correctement et terminer avec succès ou une erreur peut apparaître avant la terminaison de l'application. Si le compte primaire dysfonctionne, le secondaire s'active et le connecteur passe en mode de reconfiguration. Le compte primaire est alors détaché du connecteur et il est remplacé par le serveur de secours. La nouvelle configuration est ainsi établie jusqu'au rétablissement du primaire. Enfin, **Rétablissement** se comporte de manière similaire, les rôles entre des deux comptes étant inversés.

²Le compte secondaire n'a pas été présenté ici. Il est assez similaire au compte primaire et ne présente, par ce fait, pas beaucoup d'intérêt.

Donc, le configurateur décrit les configurations possibles (une initiale et deux configurations qui s'alternent). L'annotation « style » spécifie l'ensemble de types de composants et connecteurs que le configurateur utilise ainsi que les contraintes qu'il doit satisfaire. Dans cet exemple, il est facile de voir que la contrainte du style est vérifiée dans les trois configurations possibles.

```

CONFIGURATOR ClientServeur
  STYLE Titulaire-Compte
  // Instances et attachements
  new.C : Client →
  new.P : ComptePrimaire →
  new.S : CompteSecondaire →
  new.Conn : Connecteur →
  attach.C.p to Conn.client
  attach.P.p to Conn.serveur → Arrêt
  WHERE
  Arrêt = (P.control.down → S.control.up →
    Conn.control.changeOk → Style Titulaire-Compte
    detach.P.p from Conn.serveur →
    attach.S.p to Conn.serveur → Rétablissement) → ✓
  Rétablissement = (P.control.up → S.control.down →
    L.control.changeOk → Style Titulaire-Compte
    detach.S.p from Conn.serveur →
    attach.P.p to Conn.server → Arrêt) → ✓

```

Ainsi, Wright centralise la gestion et le contrôle de la dynamique dans le configurateur. Les abstractions architecturales, composants et connecteurs, exposent au sein de leur description les contraintes liées à leur reconfiguration. Un découplage entre la gestion dynamique de l'application et les composants est ainsi établi. Il permet de garder une définition des composants proches de ceux qui sont statiques à l'exécution, permettant certainement une meilleure réutilisation de ce code logiciel. Cependant, comme le configurateur est le seul responsable de la gestion de la dynamique, il aura sa taille proportionnelle à celle du système considéré.

4.1.3 Atouts

- Wright met l'accent sur la spécification du comportement dynamique des applications : création, suppression de composants, interconnexions qui se modifient dynamiquement en fonction de propriétés, *etc.* Des éléments de reconfiguration, constituant un vocabulaire de contrôle, ont pour cela été ajoutés au vocabulaire de base. Wright sépare, au niveau du langage, la reconfiguration dynamique d'une architecture de ses spécifications non dynamiques. Cette séparation entre le comportement et la structure assouplit la maintenance de l'application. Cette solution semble très intéressante et complète, malheureusement peu de détails concernant l'introduction de ce

vocabulaire ainsi que sur le fonctionnement du configurateur sont présentés dans la littérature ;

- Wright présente les connecteurs comme des entités sémantiques particulières. L'essence de cette approche est de fournir une notation qui donne un statut explicite à la connexion, augmentant par ce fait leur indépendance vis à vis des composants. Ceci permet à l'architecte de comprendre la fonctionnalité d'un connecteur indépendamment du contexte dans lequel il sera utilisé ;
- Un des intérêts de présenter Wright dans ce chapitre est de montrer l'existence d'un outil formel pour la vérification d'architecture d'applications. Effectivement, Wright offre un formalisme qui permet de vérifier si les ports et les rôles peuvent être compatibles d'un point de vue échange de paramètres comme du point de vue modèle d'exécution de la communication. Il supporte aussi l'analyse statique d'absence d'interblocage ;
- Un aspect non présenté dans ce document paraît être une bonne extension du langage. Les expressions telles `write!x`, pour lesquelles les composants sont à l'initiative de l'apparition de l'événement sont soulignées. Ils s'opposent aux événements observés ou subis par le composant, qui restent dans leur état naturel. Cette facilité notationale permet de comprendre rapidement quels sont les événements initiés ou observés par les composants ; La compréhension globale du système est facilitée.

4.1.4 Limites

- Il faut noter que Wright n'est pas dédié à la production d'une image exécutable de l'application. Il autorise les vérifications mais ne construit pas d'applications. Ainsi, de nombreux aspects désirés dans notre langage de description d'architecture ne sont pas pris en compte. Notamment pour la répartition, Wright s'occupe essentiellement de définir le comportement des composants et des connecteurs en termes de processus communicants et oublie de les répartir sur des sites d'exécution. Si les composants sont sur des sites différents, Wright considère qu'il s'agit de composants dans des processus différents. Par ailleurs, contrôler la reconfiguration au sein d'une entité monolithique ne paraît pas être une solution adéquate si l'on souhaite déployer une configuration distribuée à grande échelle ;
- En outre, Wright n'apporte pas de solution pour la hiérarchisation et la structuration de l'application. Les composants composites qui existent dans la plupart des langages n'ont pas d'équivalent ici. Malheureusement, toutes les architectures devant passer à l'échelle ont besoin d'encapsulation de sous-systèmes dans des composants. Cette limitation réduit donc grandement la classe d'applications utilisables ;
- Wright s'appuie sur un modèle algébrique de processus, compréhensible dans des architectures simples mais qui se complique très rapidement pour décrire des applications plus complexes.

4.1.5 Bilan

Wright est un langage architectural qui se concentre sur le comportement du système. Celui-ci est caractérisé en termes d'événements significatifs qui peuvent prendre place dans les calculs des composants et des interactions entre ces composants (les connecteurs). Ce langage a été récemment étendu[ADG98] et propose maintenant une bonne solution pour l'expression de la dynamique des applications.

4.2 Rapide

Rapide est un langage de modélisation d'architectures, dont la sémantique est basée sur des échanges de messages caractérisés par des événements partiellement ordonnés. La simulation de modèles dans Rapide permet de valider une architecture à l'aide de cette sémantique et de vérifier certaines propriétés comme l'absence d'interblocage lors de l'exécution de l'application ou la conservation de l'ordre causal de délivrance d'événements. Il est important de noter que l'environnement Rapide ne permet pas la génération automatique d'applications, mais il très intéressant dans la mesure où il met l'accent sur la spécification du comportement dynamique de l'application.

4.2.1 Concepts

La principale caractéristique de Rapide par rapport aux autres langages de description d'architecture étudiés est qu'il utilise des règles d'interconnexion, déclenchables selon le comportement des composants. Dans cette partie, nous abordons le modèle événementiel et les règles de Rapide. Sont également présentés le modèle de composants et d'architecture, permettant de décrire la structure des applications.

Composants

Chaque composant, ou module, possède un ensemble d'interfaces qui décrivent les styles d'événements qui peuvent être échangés avec les autres composants de l'architecture.

Une interface est constituée d'un ensemble de services. Ces services sont des fonctions qui peuvent être appelées ou que le composant requiert. Ils peuvent être de type *provides*, *requires* ou *action*. Les *provides* peuvent être appelés de manière synchrone par les composants. Les *requires* sont les services que le composant demande de manière synchrone à d'autres composants. Un module peut donc communiquer de manière synchrone avec un autre module si leurs services *requires* et *provides* sont connectés. Le dernier type de service possible est l'action. Une action correspond à un appel asynchrone entre composants. Deux types d'actions existent : les *in* et les *out* qui sont respectivement des événements acceptés et envoyés par un composant.

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'ordonnement 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.

Architecture

Une application est représentée par son architecture. Une architecture consiste en des déclarations d'instances de composants, des connexions entre ces composants et des contraintes sur le comportement de l'architecture. Voici la structure d'une architecture :

```
architecture name is
//Déclarations
connections
//Connexions
[constraints]
//Contraintes optionnelles
end name
```

Une architecture contient la déclaration des instances de composants ou modules. Toutes les instances sont représentées par des variables. Rapide introduit deux types de variables un peu particulières [LV95] : les *placeholder* et les *iterator*. Le nom des variables placeholder débute toujours par « ? », ce qui permet de les distinguer des variables classiques. Ce sont des variables typées, leur déclaration est similaire à des variables ordinaires. Cependant, elles ont une sémantique différente. Elles servent à la sélection dynamique, dans le sens où elles désignent un objet qui est susceptible d'être présent. Par exemple, `?c` : **C**lient fait référence à une instance de **C**lient. L'itérateur « ! » désigne, quant à lui, une conjonction d'instances d'un certain type. Ainsi, `!s` : **S**erveur désigne toutes les instances de **S**erveur présentes dans l'application. Ces déclarations de variables sont relativement dynamiques 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 de règles de connexions entre les objets et des contraintes optionnelles. Nous verrons par la suite les caractéristiques des interconnexions. Par contre, nous n'étudierons pas les contraintes, les détails de Rapide étant trop nombreux pour être tous abordés ici.

Événements

Le concept de base de Rapide est l'événement qui est une information transmise entre composants. L'événement permet de construire des expressions appelées *event patterns*. Ces expressions permettent de caractériser les événements circulant entre les composants. 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 causale, d'indépendance, *etc.* L'ensemble de ces opérateurs est répertorié dans le tableau 4.1. Un événement peut correspondre à une demande de service, à une

Opérateur	Sémantique
$A > B$	B est envoyé après A
$A -> B$	B dépend causalement de A
$A B$	A et B ne sont pas causalement dépendants
$A \sim B$	A et B sont différents
A and B	A et B sont vérifiés simultanément

TAB. 4.1 – *Opérateurs de dépendance entre événements*

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 logicielles 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 de composants d'un point de vue échange de paramètres, ordonnancement et sélection dynamique des instances de composants qui communiquent effectivement entre eux.

Règles

Une règle est composée 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. Entre ces deux parties, deux sortes de connexion sont possibles, les simples et complexes. Une connexion simple définit une interaction entre deux services (**to**). Une connexion complexe définit de manière plus générale les interconnexions de composants. Il existe deux principaux types de connexions complexes : les connexions *agent* ($||>$) et *pipe* (\Rightarrow).

to connecte deux expressions d'événements 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.

$||>$ 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, c'est-à-dire qu'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.

\Rightarrow 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 règle. Cet opérateur de connexion est appelé opérateur *pipe-line*.

Essayons maintenant d'illustrer ces règles au sein de notre application pilote.

```
with Client, Serveur ;
...
//Déclaration des instances des composants de l'application
susceptibles d'exister
?s : Client; //Référence à une instance de Client
!r : Serveur; //Référence à toutes les instances de Serveur
?d : Data; //Référence à un bloc de paramètres d'un
          certain type Data
//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.
```

L'interconnexion précédente spécifie l'existence d'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`. L'opérateur `?` 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. Les parties gauches et droites peuvent être interconnectées par trois sortes d'opérateurs (`to`, `||>` et `=>`). Dans notre exemple, 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.

4.2.2 Évolution dynamique

Dans cette partie, nous allons développer un exemple un peu plus complet qui présente une architecture dynamique. Trois types de composants interviennent : des clients, des fournisseurs et un serveur de noms. Les fournisseurs peuvent s'enregistrer auprès du serveur de noms, indiquant les services qu'ils remplissent. Le serveur de noms enregistre leur référence ainsi que leurs fonctions. Un client peut à tout moment demander au serveur un fournisseur d'un service qui l'intéresse. Il obtient en résultat un nom de fournisseur qu'il ne peut déréférencer. Pour cela, il doit envoyer une requête à l'architecture, qui est le seul module pouvant effectuer les déréférencages et générer donc la bonne requête au fournisseur. Nous ne nous intéresserons dans cet exemple qu'à la description de l'architecture.

```
with Client, Provider, nameServer ;
architecture Network is
//Déclarations
  ns : nameServer ;
```

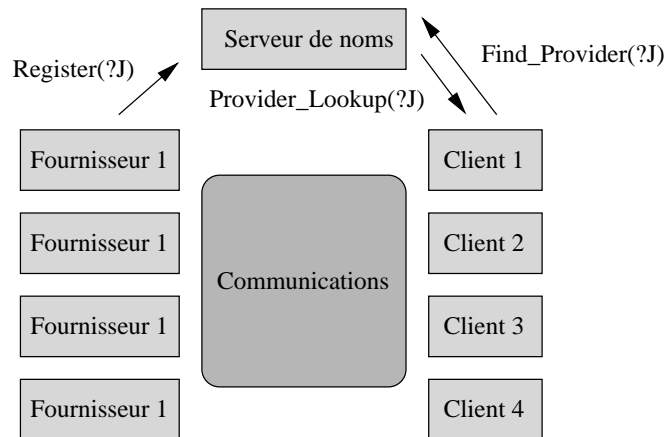


FIG. 4.3 – Représentation d'un serveur de noms en *Rapide*

```

clients : array[1..NUM_CLIENTS] of Client ;
providers : array[1..NUM_PROVIDERS] of Providers ;
?P : Provider ;
?J : Job ;
?C : Client ;
?N : &Provider ;
?param : Parameters ;
//Connexions
connect
  ?P.Register(?J) To ns.Register_Provider(?J, &?P) ;
  ?C.Find_Provider(?J) To ns.Provider_Lookup(?J) ;
  ?C.Request_Job(?J, ?param, ?N) To *?N.Do_Job(?J, ?param) ;
end Network ;

```

La première règle connecte toutes les demandes d'enregistrement (**Register**) des fournisseurs au service **Register_Provider** du serveur de noms. Cette interaction met en jeu plusieurs fournisseurs et un seul serveur de noms : l'utilisation de l'opérateur **to** est ainsi justifiée.

La seconde règle déclenche un événement **Provider_Lookup** vers le serveur de noms quand l'événement **Find_Provider** du client est détecté. Le client attend ici un résultat. Cette règle peut être considérée comme un alias de la fonction du client vers celle du serveur de noms.

Enfin, la dernière règle sert au déréférencage du nom des fournisseurs. Quand un client effectue une demande de service à l'un des fournisseurs, le serveur de nom déréférence le nom du fournisseur et renouvelle la demande de service correctement.

Cet exemple souligne l'importance des règles pour la dynamique de l'application. Elles suffisent à caractériser le comportement de chaque élément de l'architecture. Cet exemple illustre aussi la sélection dynamique des 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.*

4.2.3 Atouts

- Ce langage se situe complètement dans la classe de langages 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. 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 ;
- La simulation d'un modèle génère un ensemble d'événements apparaissant à l'exécution avec des relations causales et temporelles. Cela fournit de nombreuses opportunités pour l'analyse de modèles de systèmes complexes, particulièrement concernant les aspects de distribution et de comportement concurrent. Cette caractéristique est particulièrement appréciable puisqu'elle évite l'installation et le déploiement d'une application de grande taille pour effectuer les tests de validité de l'architecture ;
- 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'interconnexion qui évoluent en fonction du comportement des composants... 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.

4.2.4 Limites

- Dans les articles [LKA⁺95, LVM95, LV95, Luc96], aucun opérateur de création, suppression, migration de composants ou modification d'interconnexion n'est disponible. Ceci est un frein à la dynamique de l'application. En effet, il est regrettable de ne pas pouvoir reconfigurer l'application ;
- 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 ;
- Comme Wright, Rapide ne propose aucun élément de structuration de l'application.

4.2.5 Bilan

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 composants. Son système de règles d'interconnexion permet d'exprimer la dynamique des applications mais nous ne pouvons pas vraiment effectuer de comparaison par rapport aux langages de configuration précédents, Rapide ne proposant pas la génération de l'application. Toutefois, certaines propriétés des applications, ainsi que certains de ses opérateurs nous semblent très intéressants et le système de règles paraît extrêmement prometteur.

Chapitre 5

Aspects dynamiques des ADL : analyse expérimentale et proposition

L'étude des systèmes existants permet de mettre en évidence un certain nombre d'insuffisances concernant la dynamique des applications. Quelques projets ont malgré tout abouti à des propositions intéressantes. Ce chapitre synthétise les différentes approches proposées par les langages de description d'architecture. Ensuite, nous détaillons nos objectifs par rapport à cette étude et présentons une application pour les illustrer. Le support d'exécution et l'environnement Olan choisis pour la mise en œuvre de cette application seront exposés. Cette expérimentation met l'accent sur les limitations actuelles du langage OCL et nous permet de cibler plus précisément le modèle d'expression désiré pour la dynamicité.

5.1 Synthèse des approches existantes

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, filtres, *etc.* Des règles d'interconnexion strictes autorisent ou n'autorisent pas l'utilisation d'un connecteur avec des composants. Ainsi, UniCon effectue toutes les vérifications d'assemblage des entités et mécanismes utilisés pour faire fonctionner une application répartie. L'approche de Darwin est un peu différente car il n'existe pas de typage de composants au sens de UniCon. Un composant est une entité logicielle homogène sur laquelle aucune contrainte comparable aux types n'existe. UniCon tente de fournir au concepteur de l'application un cadre et des règles pour automatiser et garantir l'intégrité de fonctionnement des différents mécanismes de programmation.

OCL et Darwin proposent, quant à eux, une description dynamique de l'architecture. Ces deux langages permettent d'exprimer une partie de l'évolution initiale de l'application, essentiellement le schéma d'instanciation des composants. UniCon considère ce schéma d'instanciation comme statique, défini une

fois pour toute par l'architecte. Le tableau 5.1 présente les caractéristiques de tous les langages étudiés. Il s'intéresse bien entendu aux aspects dynamiques, mais la comparaison de ces langages porte aussi sur la hiérarchisation de l'application ainsi que la présence de la notion de connecteur et la génération automatique d'une image exécutable de l'application distribuée. Tous ces aspects semblent essentiels pour le développement de systèmes répartis. D'une part, l'intégration de logiciels comporte une phase de structuration en proposant de modéliser une application comme une hiérarchie de composants. Cette modélisation permet de ne pas considérer l'application comme un ensemble plat de composants. D'autre part, la séparation du code de la communication et celui de la mise en œuvre des composants logiciels permet de créer des composants logiciels sans se soucier de l'utilisation ou de la programmation des communications distantes entre sites. Enfin, la description de l'architecture, associée au code des composants logiciels permet, dans certains langages, d'installer et de faire exécuter l'application sur un système réparti. Cette caractéristique offre des facilités pour le déploiement, puisqu'une génération automatique de l'exécutable est effectuée.

Langage	Dynamacité	Hiérarchisation	Connecteur	Déploiement
UniCon	non	non	oui	non
Darwin	oui	oui	non	oui
Olan	oui	oui	oui	oui
Acme	possible	oui	oui	possible
Wright	oui	non	oui	non
Rapide	oui	oui	non	non

TAB. 5.1 – *Caractéristiques des différents ADL*

Les caractéristiques d'instanciation statique des langages de description d'architecture 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'instanciation paresseuse d'une part et l'instanciation dynamique d'autre part. L'instanciation paresseuse est l'action de déclarer une instance en retardant l'instant de sa création au premier accès. L'instanciation dynamique est la possibilité de créer des instances n'importe où, quand par exemple, un client le demande via un service particulier de création. Nous avons vu que l'instanciation paresseuse de Darwin ne répond pas à tous les besoins de création dynamique et que l'instanciation 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. Le concept de « collection » a été aussi introduit, apportant des réponses concernant la gestion dynamique d'un groupe de composants (*cf.* tableau 5.2). Les collections sont des ensembles, bor-

nés 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.

Quant à Acme, son statut particulier permet difficilement de le classer par rapport aux autres langages. Il se présente comme un intermédiaire entre différentes architectures exprimées au moyen de langages différents. Cependant, il fournit aussi des entités architecturales standards permettant d'exprimer l'architecture au moyen de composants et connecteurs. Seulement, la seule dynamique qu'il offre aux architectes est issue d'autres langages, aucun apport n'est à signaler.

Langage	Composant	Groupe	Connexion	Attribut
UniCon	aucune	aucune	aucune	aucune
Darwin	instanciation	aucune	aucune	complète
Olan	instanciation	gestion	aucune	complète
Acme	possible	possible	possible	possible
Wright	complète	aucune	complète	modification
Rapide	complète	gestion	complète	modification

TAB. 5.2 – *Dynamacité possible pour les abstractions architecturales*

La comparaison avec les langages de description d'architecture 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 les architectures en vue de simuler leur comportement, les premiers décrivent les architectures pour générer une image exécutable (à part UniCon). Les soucis d'intégration de code et de placement ne sont donc pas pris en compte dans Rapide. L'effort de Rapide se porte autour de la description des architectures dont la dynamique des communications est très forte. Rapide offre un certain nombre de solutions originales à la description de solutions 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.

Langage	Composant	Configurateur
UniCon	non	non
Darwin	oui	non
Olan	oui	non
Acme	possible	possible
Wright	non	oui
Rapide	oui	non

TAB. 5.3 – *Lieu de dynamacité pour les différents ADL*

Enfin, concernant Wright, on peut noter qu'il n'est pas dédié à la production d'une image exécutable de l'application. Il autorise des vérifications, de la manière à UniCon, mais ne construit pas d'application. De plus, la répartition n'est pas prise en compte, Wright s'occupe essentiellement de définir le comportement de composants et connecteurs en terme de processus communicants. Pour la gestion de la dynamique, Wright introduit dans son algèbre à base de processus un vocabulaire de reconfiguration. Toute la dynamique introduite est gérée au sein d'une entité monolithique, le configurateur (*cf.* tableau 5.3). Seules des spécifications et contraintes sont introduites dans la description des composants. Rapide, Olan et Darwin intègrent, quant à eux, tous les aspects dynamiques au sein des composants.

Pour conclure, nous considérons qu'un certain nombre de propositions sont intéressantes mais pas toujours suffisantes. Wright et Rapide expriment bien la dynamique de l'application mais ne s'intéressent qu'à sa vérification comportementale, ils ne génèrent pas d'image exécutable et ne prennent pas en compte la répartition des composants. De la même manière, les langages Darwin et Olan proposent des schémas d'instanciation qui ne répondent pas à tous les besoins de création dynamique. Compte tenu des enseignements apportés par cette étude, nous essayons d'exprimer, dans la section suivante, nos objectifs pour l'expression de la dynamique des applications.

5.2 Objectifs

Pour donner un exemple de ce que l'on souhaite obtenir, nous exposons ici un exemple d'application. Elle est composée d'un ensemble de serveurs dupliqués proposant un service d'annuaire téléphonique. Tous les serveurs possèdent une copie de la base de données. Par souci d'intégrité, ces serveurs doivent être mis en cohérence pour tout ajout d'entrée dans la base. Cette base de données contient des couples de type `<String nom, int numTéléphone>`, représentant respectivement le nom d'une personne et son numéro de téléphone.

Dans cette mise en œuvre, des clients peuvent se connecter à tout moment à un serveur particulier et effectuer une consultation de la base de données. De même, ils ont la possibilité de s'enregistrer auprès de l'annuaire téléphonique en fournissant leur nom et leur numéro de téléphone. C'est dans cette situation qu'une mise en cohérence entre les différents serveurs est nécessaire.

Nous avons essayé dans cette phase de programmation de séparer la phase de contrôle de l'application (*p.ex.* mise en cohérence des serveurs) du code des composants. Effectivement, ce code n'a pas lieu d'être ni dans les composants `client`, ni dans les `serveur`. Le contrôle est lié à l'application dans sa globalité et non pas à l'un de ses constituants. Nous avons donc regroupé tous les aspects dynamiques dans un composant séparé, que nous avons appelé `controller`.

Quatre styles de dynamicités, gérées par notre contrôleur, ont été introduites dans cet exemple. Nous les détaillons ci-dessous, une représentation visuelle est offerte par la figure 5.1 :

- Régulièrement, des clients peuvent arriver dans le système (1). À la détection de ces nouveaux composants, le contrôleur enregistre leur arrivée

- (2);
- Le contrôleur crée alors une liaison dynamique (3) entre le client nouvellement arrivé et un des serveurs disponibles. À ce moment là, le client a la possibilité d'accéder à l'annuaire téléphonique. Il est libre de formuler des requêtes auprès du serveur auquel il est rattaché pour obtenir des numéros de téléphone;
 - Il a aussi la possibilité de s'enregistrer dans la base de données. Pour cela, il doit fournir au serveur son nom et son numéro de téléphone. Chaque enregistrement d'un client déclenchera une réaction de la part du contrôleur (4). Ce composant maintient à jour les données de tous les autres serveurs qui n'ont pas enregistré cette nouvelle entrée.

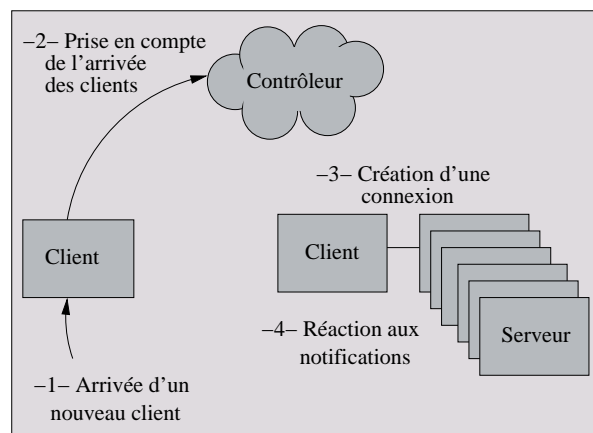


FIG. 5.1 – Représentation de l'annuaire téléphonique

Dans cet exemple, nous avons introduit deux formes de dynamique bien distinctes. D'une part, nous souhaitons décrire des politiques d'administration et de contrôle en spécifiant le comportement exact de l'application à l'exécution. D'autre part, nous aimerions pouvoir exprimer des reconfigurations à apporter au système initial en réaction au comportement de l'application.

5.3 Environnement

Notre expérimentation a été effectuée avec des composants particuliers (des *agents*), s'exécutant sur le support d'exécution A3 [BPF⁺99]. Ce support a été choisi pour son modèle « Événement-réaction » bien adapté à l'exemple, puisqu'il permet de réagir à des événements significatifs, comme l'arrivée de clients dans le système. La présentation du support d'exécution A3 et de l'environnement Olan est détaillée dans la suite.

5.3.1 Support d'exécution A3

L'environnement A3 (Agent Anytime Anywhere) [PBF⁺99] est un support d'exécution pour des entités réactives appelées *agents*. Les agents sont des objets qui se comportent selon le 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 un agent destinataire afin que la réaction correspondante soit exécutée. Ce système de notifications est le seul moyen laissé à la disposition des agents pour la communication. Les notifications sont amenées jusqu'aux agents via un bus à message appelé **Channel** (figure 5.2). Ce **Channel** est piloté par le moteur A3 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 d'exécution A3 est appelé serveur d'agents.

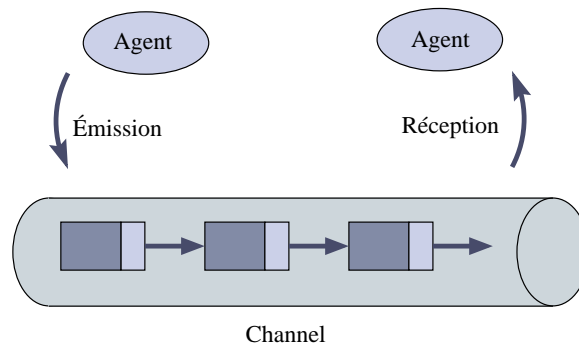


FIG. 5.2 – Architecture du serveur d'agents A3

L'architecture distribuée est composée d'un ensemble de serveurs d'agents répartis sur différents sites. Chaque agent est attaché à un moteur d'exécution, les seules entités mobiles sont les notifications. 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 fonction des notifications reçues.

5.3.2 Environnement Olan

Le langage de description d'architecture OCL permet l'expression, dans un mode déclaratif, de la configuration d'une application en termes de composition d'unités logicielles élémentaires et les éventuelles contraintes de déploiement sur les environnements d'exécution. À ce langage sont associés des outils pour la vérification de la validité de l'assemblage, pour la génération du code binaire de l'application distribuée, l'automatisation du déploiement et des services systèmes pour supporter le processus de déploiement, d'administration et de reconfiguration de l'application. La figure 5.3 représente l'ensemble de ces outils. L'environnement de configuration Olan est effectivement constitué d'un ensemble d'outils destinés à configurer des applications réparties. Le terme configuration désigne ici, d'une part, l'action de définition et de spécification 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. Cet environnement comprend en effet deux sortes d'outils. Les outils dédiés à la description de l'architecture de l'application comportent un outil de construction de composants (primitifs et composites) et un outil de configura-

tion d'applications qui permet de modifier l'assemblage des composants et fixer la valeur d'attributs de ces composants. Le deuxième type d'outils concerne plus l'installation et l'exécution des applications. Il regroupe un outil de déploiement permettant l'exécution des composants sur des sites et un outil d'administration et de reconfiguration des applications agissant une fois qu'elles sont déployées.

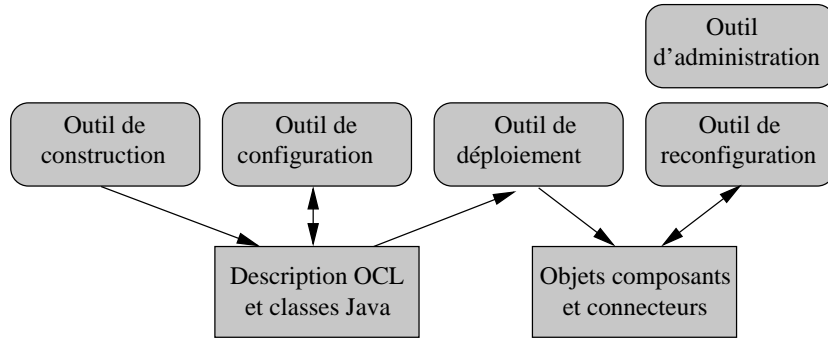


FIG. 5.3 – *Environnement Olan*

Construire une application avec Olan consiste d'abord à décrire la configuration à l'aide du langage OCL. Le réalisateur ou l'administrateur pourra retrouver la même vision de l'application à la conception qu'à la configuration. À cette étape, une mise au point avant le déploiement est effectuée. Au travers de cet outil de configuration, l'application pourra être réglée et paramétrée de manière à prendre en considération les contraintes spécifiques de l'environnement où elle est déployée. Ensuite, l'outil de déploiement installe sur des sites répartis les objets composants et connecteurs afin qu'ils évoluent librement. L'exécution de ces entités est surveillée et contrôlée. Un outil de reconfiguration, associé à celui d'administration permet de modifier l'application en cours d'exécution.

Nous souhaitons utiliser cette fonction d'administration (ou de monitoring) pour surveiller des événements significatifs pour l'application. Dès la détection de ces événements dans notre expérimentation, une analyse et un traitement sont effectués par le composant contrôleur. La consommation de ces événements déclenche essentiellement des modifications de l'architecture (*p. ex.* création de connexion), à l'aide de l'interface de reconfiguration.

5.4 Application pilote

Une mise en œuvre de l'application a été effectuée et des tests ont été réalisés pour un ensemble variable de composants et de serveurs. La programmation des différents composants a été réalisée dans le langage Java, en tenant compte des règles de programmation induites par le support d'exécution A3. Les agents A3 ont un modèle de programmation particulier. Ils communiquent avec d'autres objets par l'envoi de notifications au bus à message. Au niveau du langage OCL, les agents doivent posséder une interface spécifique construite sur le modèle suivant :

- Ils possèdent un ou plusieurs services de type **A3Service**. Les paramètres de ces services sont des notifications et le fichier où ces notifications sont

définies ;

– Chaque service fourni ou requis de l’interface de l’agent dérive d’un `A3Service`. Tout d’abord, nous allons nous intéresser aux éléments statiques de l’architecture. Étudions, dans un premier temps, le code OCL de définition de l’agent `client`. Son interface est représentée dans la figure 5.4.

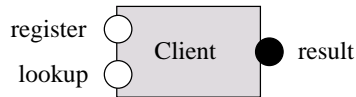


FIG. 5.4 – Description OCL du client de l’annuaire téléphonique

Le client expose au niveau de son interface son nom, adresse et numéro de téléphone, ainsi que son site hôte. Cette dernière information est prise en compte pour le déploiement de cet agent. L’implémentation de l’agent spécifie l’emplacement du fichier Java résultant de l’application OCL.

```
component Client : A3Agent{
  /* Interface */
  Interface ClientItf{
    attribute String name ;
    attribute String address ;
    attribute int number ;
    attribute Location location ;
    required ClientService register ;
    required ClientService lookup ;
    provided ClientService result ;
  }
  /* Implémentation */
  Implementation{
    agent = aaa.test.Client ;
    file = ${SRC_DIR}/aaa/test/Client.java ;
  }
}
```

Nous rappelons que les services définissent les opérations ou ensemble d’opérations qui peuvent être accessibles ou utilisées par les composants. La sémantique exacte d’un service dépend de son type. Un service fourni ou requis par les agents A3 correspond à un ensemble de notifications qui peuvent être envoyées et reçues par les agents. Ces notifications sont implémentées dans cet exemple par une classe qui étend `aaa.agent.NotificationSet`. Nous aurions pu envisager d’associer une classe pour chaque service fourni ou requis.

```
Service ClientService : A3Service {
  notifications = aaa.test.ClientNotificationSet ;
  file = ${SRC_DIR}/aaa/test/ClientNotificationSet.java ;}
```

Chaque méthode de cette classe Java crée une notification qui peut être envoyée ou reçue directement par les agents. La notification correspondant aux paramètres passés est instanciée.

```
package aaa.test ;
public class ClientNotificationSet extends aaa.agent.NotificationSet{
    public static RegisterNot(int number, String name){
        return new RegisterNot(number, name) ;
    }
    public static RegisterNot(int number, String name, String address){
        return new RegisterNot(number, name, address) ;
    }
    public static LookupNot(String name){
        return new LookupNot(name) ;
    }
    public static ResultNot(int number){
        return new ResultNot(number) ;
    }
    public static ResultNot(int number, String address){
        return new ResultNot(number, address) ;
    }
}
```

Il est intéressant maintenant de regarder quels sont les sources Java encapsulées du fichier `Client.java`. Le code suivant est en est un extrait :

```
public class Client extends Agent implements Initializable{
    public String name ;
    public String address ;
    public int number ;
    public Client(short to, String OCLName){
        super(to) ;
    }
    public void react(AgentId from, Notification not){
        if (not instanceof Init) {
            sendTo(server, new RegisterNot(...)) ;
            ...
            sendTo(server, new LookupNot(...)) ;
        }
        else if (not instanceof ResultNot) {
            //Traitement du résultat
        }
        else super.react(from, to) ;
    }
}
```

Le lecteur pourra remarquer que le client envoie à un certain moment une notification pour s'enregistrer auprès du serveur. Il effectue aussi un `lookup`

pour demander un numéro de téléphone particulier. La réponse de sa requête sera contenue dans une notification qu'il va recevoir de type `ResultNot`.

Le code du serveur n'est, quant à lui, pas très intéressant. Le serveur se contente de stocker dans une table de hachage, le nom et le numéro de téléphone des clients qui s'enregistrent. En ce qui concerne la recherche de numéros de téléphone, le serveur consulte simplement sa table et renvoie le résultat au client. La description du contrôleur en OCL n'est pas non plus d'un grand intérêt.

Nous présentons, par conséquent, directement le code simplifié du contrôleur en Java. Nous rappelons que son rôle consiste à déclencher des actions de reconfiguration suite à des événements produits dans le système. Nous avons utilisé concernant les aspects de surveillance, l'outil d'administration d'Olan, qui permet l'abonnement aux occurrences d'un type d'événement se produisant dans l'application.

```
public class Controller extends Agent {
    //Constructeur
    public Controller (short to){
        super(to);
    }
    protected void reconfigurationPolicy(MonitoringReport ev){
//Traitement de l'arrivée d'un client
        if (ev instanceof ConnectNot){
            ...
//Enregistrement du client
            conf.RegisterAgent(ev.not.OCLName, ev.not.ag);
            ...
//Connexion à une serveur
            bind(ev.not.OCLName, "server", servers.get(index));
            ...
//Abonnement auprès de l'outil d'administration
            aux notifications outputSubscribe, de type register
            sendTo(ev.not.ag,
                new OutputSubscribeNot(SubscribeNot.ADD,"server",
                    "RegisterNot"));
        }
//Traitement d'un register observé
        else if (ev.not instanceof RegisterNot){
            for(...){
                sendTo(servers.get(i).getId(),
                    new RegisterNot(ev.not.getName(),
                        ev.not.getNumber()));
            }
        }
        else ...
    }
}
```

}

En réaction à la création d'un nouveau client, le contrôleur enregistre son arrivée et le connecte à un serveur d'annuaire. Il s'abonne ensuite aux notifications « register » émises par le client, en destination du serveur auquel il est connecté. Afin de mettre en cohérence les différents serveurs, le contrôleur crée une notification et la distribue aux autres serveurs. Dans cet exemple, la méthode `reconfigurationPolicy` a été ajoutée pour réagir aux événements surveillés et détectés par l'outil d'administration. Les événements concernés ici sont : demande de connexion au service téléphonique (`connectNot`) et envoi d'une notification `register` vers un des serveurs de l'application.

5.5 Limitations

Nous pouvons remarquer tout de suite que la dynamique introduite dans cette illustration est exprimée au sein du code encapsulé du contrôleur et non au niveau de sa description en OCL. En effet, le mode opératoire situé dans le code Java des composants n'est pas exprimable dans le mode déclaratif de ce langage de description d'architecture. Une description en OCL, dans l'état actuel, ne permet pas de modifier la configuration de l'application une fois que son déploiement est effectué.

Au niveau du langage OCL, il manque des moyens d'expression pour :

- l'utilisation d'opérateurs d'itération, comme `for` et `while`, et de tests (`if`) pour décrire un changement complexe de la configuration ;
- la reconfiguration de l'application, en termes d'opérateurs (*p.ex.* pour la création dynamique de connexions) ;
- la réaction à des événements externes (*p.ex.* pour l'arrivée des clients) ;
- la réaction à des événements internes (*p.ex.* pour la réaction à l'enregistrement d'un client auprès d'un serveur).

5.6 Expression de la dynamique dans Olan

Compte tenu des observations faites sur les systèmes étudiés et sur l'expérimentation menée, il apparaît clairement le besoin, au niveau du langage de description, de réagir à la détection de faits particuliers, en exécutant des opérations de reconfiguration. Cette capacité de réaction à des situations significatives ne demande aucune connaissance particulière de la provenance de l'événement. C'est pourquoi, la notion de règles, similaire à celle de Rapide, nous semble intéressante. En outre, le concept Événement-Réaction permet une implémentation dynamique de façon relativement simple et naturelle. Enfin, les règles peuvent être utilisées pour réaliser diverses fonctions, tant au niveau de la description de la dynamique de l'architecture, qu'au niveau des applications elles-mêmes. En effet, elles peuvent permettre, par exemple, de gérer les interactions entre les utilisateurs ou entre les applications et le système, de vérifier des contraintes d'intégrité. L'étude des travaux de Sloman et Lupu [LS97, Slo94, MSS97] nous a

conforté dans cette idée. Le système GEM basé sur des politiques d'administration utilise des règles et événements pour définir, par exemple, les autorisations et obligations des employés d'une entreprise. La notion d'administration du système est une approche assez similaire à celle d'administration d'applications souhaitée. En considérant cette argumentation, nous avons opté pour l'utilisation effective de règles dans OCL.

A partir de ce choix, plusieurs formalismes s'offrent à nous. Nous pouvons reprendre les travaux effectués pour le projet Rapide, se diriger vers le formalisme des systèmes experts ou bien coder la dynamique sous forme de règles directement dans la description du composant.

Nous avons sélectionné un tout autre style de règles. Notre choix c'est porté sur les règles actives de type Événement-Condition-Action, issues des bases de données. Elles peuvent être vues comme une généralisation des règles de production nées des travaux réalisés dans le domaine de l'intelligence artificielle [Yam98, LS98]. Les systèmes experts utilisent en effet des langages de règles de production sous la forme Condition-Action. De plus, ce formalisme permettra d'exprimer sous forme déclarative la dynamique de l'application.

Chapitre 6

Utilisation de règles pour la configuration dynamique d'applications

Avant de proposer une modélisation de nos règles pour la dynamique dans Olan, nous présentons les hypothèses, faites au préalable, relatives aux services souhaités à l'exécution. Ensuite, les fonctionnalités actives architecturées autour de la notion de règles sont présentées. Le formalisme E-C-A (Événement, Condition et Action) sur lequel sont basées les règles est détaillé et la syntaxe et l'expression des règles sont exposées. Nous présentons enfin le modèle d'exécution associé à nos règles et concluons ce chapitre par une discussion concernant notre proposition.

6.1 Hypothèses relatives aux services à l'exécution

Nous rappelons dans cette section le contexte d'exécution dans lequel nous nous plaçons et formulons des hypothèses sur les services disponibles. Comme nous l'avons vu précédemment, l'environnement Olan nous propose de nombreux outils, notamment des outils dédiés à la construction et à la configuration, au déploiement et à l'administration d'applications. Associé à l'outil d'administration, un service de reconfiguration est disponible. Le tableau 6.1 expose les différentes opérations fournies par ce service.

Nous souhaitons utiliser un large éventail d'opérateurs fourni par ces outils pour exprimer entre autre la dynamique des applications. Nous avons accès au vocabulaire d'une configuration OCL (précédemment introduite dans la section 3.2). Cependant, une configuration ne contient que le vocabulaire de base, formé d'instanciation de composants et de création de connexions et de liaisons. C'est le service de reconfiguration qui a la charge d'offrir les opérations de reconfiguration. Nous faisons donc l'hypothèse de disposer d'un vocabulaire constitué d'éléments de configuration et de reconfiguration.

Nous supposons également que le système possède un service complet de gestion d'événements. En d'autres termes, le système doit proposer un service de production et de détection d'événements et doit permettre l'abonnement à

Opération	Description
new	Ajout d'une instance d'un composant
delete	Suppression d'une instance d'un composant
bind	Ajout d'une connexion
unbind	Suppression d'une connexion
setAttribute	Modification de la valeur d'un attribut
clone	Clonage d'un composant
move	Déplacement d'un composant d'un site vers un autre

TAB. 6.1 – *Opérations de reconfiguration*

un certain type d'événement. Le tableau 6.2 synthétise les types d'événements auxquels on souhaite réagir. Ces événements peuvent être émis par les services d'administration et de surveillance à l'exécution¹.

Type d'événement	Description
OUTPUTREPORT	Envoi d'une notification à travers le rôle d'un composant
INPUTREPORT	Réception d'une notification par le rôle d'un composant
STATUSREPORT	Modification de l'état d'un agent ou d'un connecteur
CREATE	Création d'un composant
DESTROY	Destruction d'un composant
CLONE	Clonage d'un composant
MOVE	Déplacement d'un composant d'un site vers un autre
ATTACH	Établissement d'une connexion entre deux composants
DETACH	Détachement d'une connexion entre deux composants
SETATTRIBUTE	Modification de la valeur d'un attribut d'un composant

TAB. 6.2 – *Types d'événements primitifs*

Après avoir spécifier les hypothèses dans lesquelles nous nous situons, nous pouvons envisager de définir nos règles actives.

6.2 Règles actives

L'évolution des systèmes de base de données tend vers une approche particulière qui vise à rendre ces systèmes actifs [Col96]. Le terme « Actifs » signifie dans le contexte des bases de données, capables de réagir à des événements et déclencher des actions appliquées à la base de données. Le comportement actif d'un système peut être représenté de manière générale par le schéma *FAIT* -> *RÉ-ACTION* signifiant : si un événement ou une condition (*FAIT*) se produit alors traiter la *RÉ-ACTION* correspondante. Dans le domaine des bases de données,

¹Les événements OUTPUTREPORT, INPUTREPORT et STATUSREPORT sont actuellement traités par l'outil d'administration.

divers mécanismes implantant ce schéma ont été proposés. Les mécanismes de déclencheurs (triggers) ont été, par exemple, introduits pour gérer la cohérence de la base. D'autres travaux se sont ensuite intéressés à l'intégration de cette notion de règle. Les bases de données actives sont aujourd'hui principalement basées sur ce mécanisme à base de règles, dites règles actives.

Les règles actives implantent le schéma FAIT -> RÉACTION sous le formalisme Événement-Condition-Action (E-C-A). La sémantique générale associée à une règle est :

```
LORSQUE l'événement E est produit,  
SI la condition C est satisfaite,  
ALORS exécuter l'action A.
```

Nous détaillons dans la partie suivante les trois composants d'une règle active.

6.2.1 Événement

Un événement est défini comme une situation particulière pour certaines règles, pouvant se produire à tout moment dans une application. Conceptuellement, la notion d'événement est attachée aux notions de types d'événement et occurrences d'un type d'événement[Col96]. Un type d'événement réfère à une catégorie de faits à laquelle l'événement appartient alors que l'occurrence d'un type d'événement est perçu comme une instance de ce type. Par exemple, la « création d'un composant » est un type d'événement et la « création d'un composant primitif de la classe `Client.java` ayant `c0` comme attribut » en est une occurrence ou un sous-type. Dans la suite du document, l'emploi du terme « événement » correspondra en fait à une occurrence d'un type d'événements.

Événements primitifs

Les événements primitifs sont en général regroupés en deux catégories : les événements internes provenant directement de l'application et les événements externes. Actuellement, des types d'événements utilisateurs ne peuvent pas être définis explicitement, nous avons limité notre travail à des types d'événements simples. Une bibliothèque d'événements prédéfinis est d'ailleurs disponible pour l'utilisateur. Le tableau 6.2 identifie les événements primitifs d'Olan. Une étude plus approfondie des besoins devrait permettre de rendre cette liste exhaustive pour des utilisations futures.

Les événements internes sont constitués par tous les événements émanant directement de l'application. Les événements externes (aussi appelés événements système ou utilisateurs) ne sont pas liés au comportement de l'application mais plutôt à des phénomènes extérieurs au système, par exemple l'heure, un contrôleur de température. La prise en compte de ces événements permet d'étendre le champ d'application des règles. Ces événements sont détectés à l'extérieur de l'application et doivent être toutefois signalés ou notifiés au système qui va pouvoir déclencher les règles associées.

Un environnement est associé à chaque événement. Ce contexte est un ensemble d'informations permettant d'expliciter les conditions de production de cet événement : son type, sa valeur. Ces informations contextuelles peuvent être utilisées dans les parties Condition et Action de la règle déclenchée par l'événement. Ainsi, pour les **STATUSREPORT**, un nom d'entité (qui peut être le nom d'une classe, d'un type), l'ancienne et la nouvelle valeur concernées par l'événement constituent son contexte. Ainsi, à chaque événement correspond une structure avec des champs particuliers. Les structures détaillées de tous les événements ne sont pas présentées ici.

Événements composites

Les événements composites consistent en des événements combinés entre eux par des opérateurs de composition comme la conjonction, la disjonction et la séquence. Soient **E1**, **E2**, ..., **En** des types d'événements et **e1**, **e2**, ..., **en** leurs instances respectives.

Conjonction e, instance du type d'événement **E1 AND E2**, se produit si **e1** et **e2** se produisent dans n'importe quel ordre.

Disjonction e, instance du type d'événement **E1 OR E2**, se produit si **e1** ou **e2** (ou les deux) ont lieu.

Séquence e, instance du type d'événement **E1 ; E2**, se produit si le dernier événement qui compose **e1** a lieu avant le dernier événement qui compose **e2**.

6.2.2 Condition

La Condition permet de préciser les situations pour lesquelles il faut exécuter l'action de la règle : l'action n'est exécutée que si la condition est vérifiée. C'est une expression optionnelle² à évaluer sur un ensemble de données provenant de n'importe quel élément de l'application, voire du contexte d'exécution de cette application (*p.ex.* y a-t-il des composants déployés sur un site en dysfonctionnement ?). Le résultat d'évaluation de la condition est la conjonction des résultats booléens de chaque sous-expression.

6.2.3 Action

La partie Action décrit les traitements à réaliser, suite à la production d'un événement et à l'évaluation à vrai de la Condition. C'est généralement un ensemble d'opérations macroscopiques effectuées sur l'architecture initiale de l'application. Ce sont donc des opérations de remodelage de l'architecture, telles que des créations de connexions ou des suppressions de composants.

²En cas d'absence de la partie Condition, la condition d'une règle est supposée toujours vérifiée.

6.3 Expression de la configuration dynamique au moyen de règles actives

Nous venons de présenter les concepts de base associés aux règles actives. Nous avons vu notamment comment sont définies les différentes parties de ces règles. Cette section s'intéresse non plus à la description des règles actives mais à leur utilisation. Elle présente leur syntaxe et leur moyen d'expression dans le langage de description d'architecture. Nous abordons dans une dernière partie la configuration dynamique de l'application, et nous détaillons les manières de l'exprimer.

6.3.1 Syntaxe des règles

Nous présentons dans cette partie la syntaxe de nos règles. Nous nous sommes inspirés des systèmes NAOS [CC96], un système actif pour le SGBD O₂, et ODAS [CVSGR], un service ouvert actif. Nous nous sommes restreints à un modèle très simple de règles, il est illustré ci-dessous.

```
ON <selector>
IF <boolean expression> THEN
DO <action>
```

En écrivant une règle, l'architecte doit définir les parties Événement, Condition et Action. L'emplacement `selector` spécifie le type des événements déclencheurs de la règle. La clause `IF` spécifie une condition devant être vérifiée. Si la condition est évaluée positivement, l'action de la règle est exécutée. La partie Action définit l'ensemble des modifications à effectuer à partir de la configuration initiale, en réaction à la production du ou des événements.

6.3.2 Expression dans l'ADL

Il serait intéressant de définir les règles indépendamment de la description des composants pour des soucis de réutilisabilité et maintenabilité (compilation séparée). Mais des questions sont soulevées. Où les définir ? Comment les rendre réutilisables ? adaptables ? Comment leur passer des paramètres ? Ce sont des problèmes trop complexes et le temps qui nous est imparti n'est pas suffisant pour pouvoir les aborder. Les actions déclenchées par les règles sont en outre trop souvent spécialisées, elles sont difficilement réutilisables dans des contextes différents.

C'est pourquoi, l'expression des règles actives est embarquée dans la description des composants composites. Ces composants ayant connaissance de leur architecture interne, ils sont chargés de gérer le comportement de leurs sous-composants.

```
Component class className{
//Déclaration des services fournis et requis
interface
```

```

...
//Instanciation des sous-composants et interconnexions
configuration
...
//Définition des règles actives
rules
...
}

```

La syntaxe des composants composites est illustrée ci-dessus. Elle met en évidence les trois parties nécessaires à la description d'un composite : la déclaration de ses services fournis et requis dans l'interface, la configuration de ses sous-composants en termes d'instanciations et connexions ainsi que la définition de leurs règles actives.

6.3.3 Configuration dynamique

La configuration dynamique est exprimée dans la partie Action des règles actives. Plusieurs approches ont été envisagées.

Description complète des actions dans l'ADL

Une première solution pour l'expression de l'Action est d'utiliser les éléments d'une configuration OCL. L'ensemble d'opérateurs autorisés appartiendrait au vocabulaire de configuration étendu à des opérations de reconfiguration (*cf.* tableau 6.1) définies dans une API fournie par l'environnement. Pour offrir une certaine souplesse d'utilisation, des itérateurs (**for**, **while**), des structures conditionnelles (**if**) et des éléments structurés (*p. ex.* un tableau) seraient introduits. Ainsi, nous obtenons une grammaire permettant un remodelage dynamique de l'architecture de l'application.

Afin d'illustrer cette approche, reprenons l'exemple du chapitre précédent concernant un annuaire téléphonique. Comment exprimer la dynamique des points 1, 2 et 3 à l'aide de notre langage OCL étendu ? Tout d'abord, les règles ont été encapsulées dans la description du composant composite représentant l'application. Toutes les instances présentes dans sa configuration (c'est-à-dire ses sous-composants) sont ainsi connues. C'est pourquoi l'utilisation de l'instance **serveur(i)** est directe dans la règle ci-dessous. Ensuite, nous pouvons remarquer que le type de l'événement déclencheur de la règle est « CREATE client *c* ». Nous avons nommé le type d'événement (**c**) afin d'introduire des facilités de désignation dans le reste de la règle, même si celle-ci est assez simple. La réaction associée à cette création est de connecter (opération => fournie par le vocabulaire de reconfiguration) le client à un des serveurs de la collection.

```

composite Application {
attribute int nbServers;
Configuration{
    //Instanciation de serveurs au sein

```

```

    d'une collection
    serv = Collection[0..nbServers] of Server;
    for (int i; i<nbServers; i++){
        serv.new(location = i);
    }
    ...
}
}
Rules{
    ON CREATE client c
    DO c => serv[i];
}
}

```

Cette solution permet de reconfigurer l'application à l'exécution selon son comportement, un premier objectif est déjà atteint. Cependant, il pourrait être intéressant d'exprimer des appels de services dynamiquement. Par exemple, essayons de formuler la dynamique du point 4. Sa sémantique est de réagir à une notification de type **Register** d'un client vers un serveur. Cette réaction consiste à appeler le service **register** des autres serveurs présents dans l'application afin de les mettre en cohérence. Or, aucun mécanisme pour appeler des services de composants n'est défini. Nous introduisons donc ici un moyen d'accès à ces services. Dans notre exemple, on appelle **register**, le code associé à la réaction d'un serveur quand il reçoit une notification de type **RegisterNot** d'un client. Le problème c'est qu'habituellement, ce code n'est pas visible au niveau du langage de description.

```

ON OUTPUTREPORT client c
IF c.type = «Register» THEN
DO for(.. i ..)
    serv[i].RegisterNot(c)

```

Dans le cadre du support A3, il n'est pas très difficile de comprendre le fonctionnement de cette notation. Effectivement, la notification **RegisterNot** du client est visible au niveau de son interface. Ainsi, notre appel **RegisterNot(c)** se charge d'instancier la classe du même nom, avec les bons paramètres (ici, le client). Pour le serveur, l'action du contrôleur sera transparente. Il recevra une demande d'enregistrement pour un client et mettra donc à jour sa base de données. Des solutions similaires existent pour les autres supports d'exécution.

Définition des actions dans le code du contrôleur

Une autre possibilité concernant les appels de service est d'associer les actions d'une règle à du code défini dans un service de contrôle de l'application. Nous pouvons imaginer que l'application contient un tel service, qui pour chaque action déclenchée appelle une méthode interne. Ce service aura la forme d'un

composant est sera déployé en même temps que l'architecture initiale de l'application. La figure 6.1 montre une règle contenant des opérations de configuration et de reconfiguration classiques ainsi que des actions définies dans le contrôleur. Chaque action A_n utilisée dans la règle du composant composite appelle une méthode $A_n()$ qui lui est associée dans le code du contrôleur. Ainsi, toutes les actions qui ne sont ni des opérations de configuration, ni des opérations de reconfiguration sont définies à l'extérieur de la règle.

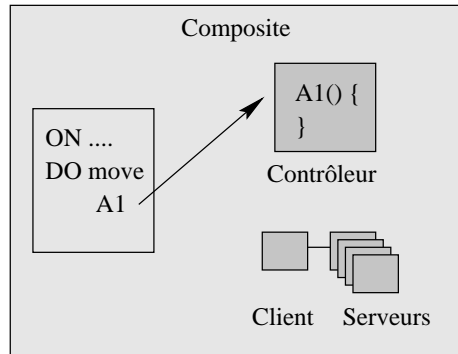


FIG. 6.1 – Actions associées à des méthodes d'un contrôleur

Définition des actions sous forme de script externe

Une dernière solution serait de faire correspondre l'action à une classe Java ou un fichier script de manière plus générale. Le langage JavaScript semble par exemple bien approprié. La règle aurait donc la forme :

```
ON OUTPUTREPORT client c
IF c.type = «Register» THEN
DO action = ${SRC_DIR}/aaa/Action
```

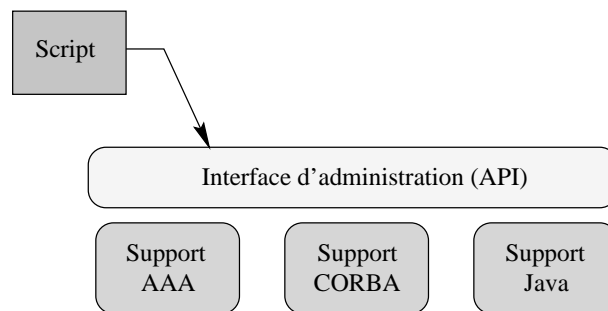


FIG. 6.2 – Action associée à un script

Le script pourra être chargé dynamiquement et des modifications en cours d'exécution pourraient être envisagées. Ce script contiendrait les reconfigurations et les appels de services. Cette dernière solution étant la plus simple à

concevoir et la plus facile d'utilisation, ce sera certainement celle que l'on retiendra.

6.4 Modèle d'exécution des règles

Dans cette partie, nous présentons les principaux aspects qui interviennent dans la définition du modèle d'exécution d'un système de règles actives. Un tel modèle permet de décrire quand et comment est exécutée une règle, une fois qu'elle est déclenchée. [Cou96] a proposé 21 dimensions pour caractériser le modèle d'exécution. Chaque dimension peut prendre un ensemble de valeurs. Toutefois, nous ne pouvons pas spécifier ces valeurs, elles nous sont imposées par notre support d'exécution. Nous discutons ici de quelques unes de ces dimensions.

Granularité du déclenchement

Une règle est associée lors de sa définition à un type d'événement. Lorsque plusieurs occurrences de ce type d'événement sont produites au cours de l'exécution de l'application, deux stratégies existent quant au nombre de déclenchements de la règle considérée :

- Sémantique par instance : déclencher l'exécution de la règle autant de fois que d'événements produits ;
- Sémantique ensembliste : déclencher la règle qu'une seule fois pour l'ensemble des événements générés depuis la dernière exécution de la règle.

La granularité du déclenchement de nos règles dépend du support associé à leur exécution. Dans Olan, les supports A3 et CORBA ont une politique par instance. Les règles sont exécutées pour toutes les occurrences du type de l'événement déclencheur.

Consommation des événements

Une règle est susceptible d'être déclenchée et exécutée plusieurs fois au cours de l'exécution de l'application par des occurrences successives d'un même type d'événement. Chaque événement est conservé jusqu'à la première exécution de la règle qu'il a déclenchée. Un événement est donc pris en compte au moins une fois au cours de l'exécution. Les éventuelles exécutions ultérieures de la même règle peuvent s'envisager selon deux politiques :

- Consommation : l'événement est consommé par la première exécution de la règle qu'il a déclenchée ;
- Préservation : après avoir été pris en compte une première fois, l'événement est préservé et reste visible dans l'environnement des exécutions ultérieures de la règle.

Olan se place dans un modèle de consommation locale. Cette politique particulière de consommation permet de laisser l'événement visible pour d'autres règles alors qu'il est consommé localement.

Modes de couplage

Pour compléter la définition du modèle d'exécution des règles actives, nous devons répondre à deux questions essentielles : où et quand s'exécutent les parties Condition et Action d'une règle relativement à l'occurrence de l'événement déclenchant d'une part, et relativement l'un à l'autre, d'autre part.

Le mode de couplage entre une Condition et une Action spécifie quand doit être exécutée l'Action par rapport à l'évaluation de la Condition. Si la condition est vérifiée, l'action est immédiatement mise en exécution dans OCL. Le mode de couplage C-A est immédiat. Nous devons également spécifier le mode de couplage E-C, c'est-à-dire préciser quand doit être évaluée la Condition d'une règle par rapport à la détection de l'événement déclenchant. Pour simplifier l'utilisation des règles, nous nous plaçons aussi dans un mode immédiat. Ainsi, la Condition est évaluée juste après la détection de l'événement, et si elle est satisfaite, l'action est alors immédiatement exécutée. D'autres combinaisons peuvent être retenues, mais nous nous astreignons à proposer un modèle d'exécution simple pour le moment.

Stratégie d'ordonnement

Lorsque plusieurs règles sont candidates à l'exécution, le système doit décider quelles règles exécuter et dans quel ordre. Il s'agit de déterminer un ordonnancement des règles considérées qui permette d'obtenir une exécution déterministe et prévisible. Dans notre cas, l'ordonnement des règles est lié aux instants de production des événements déclencheurs. Les supports d'exécution d'Olan ne garantissent pas un ordre d'exécution déterministe des réactions déclenchées.

Effet net des opérations

L'effet net d'une opération est le bilan des actions effectuées successivement sur une même entité, seul l'effet net obtenu en fin d'exécution est considéré. Par conséquent, si une opération crée deux entités *a* et *b*, puis détruit *a*, l'effet net de l'opération sera la création de l'entité *b*. Nous ne considérons pas les effets nets.

Dans la plupart de ces dimensions, nous nous reposons sur le support d'exécution et nous ne pouvons pas fixer des valeurs librement. En revanche, il semble possible de prendre en compte certains de ces aspects. Par exemple, il paraît assez simple de prendre en compte un modèle de couplage différé, en repoussant l'évaluation de la condition ou l'exécution de la règle à un point ultérieur de l'exécution. Nous n'en tenons pas compte pour le moment mais nous pensons que ce serait un point intéressant à développer.

6.5 Discussion

6.5.1 Conservation du formalisme ?

La question principale soulevée par notre proposition est de savoir si nos règles événementielles trouvent leur place au sein de la description des composants. Bien entendu, ce choix délibéré répond à notre besoin d'exprimer des stratégies d'administration au sein même de l'ADL. Mais, qui va réellement spécifier le comportement de l'application ? L'architecte ? Est-ce vraiment son rôle de définir le comportement de l'application ? Ne serait-ce pas plutôt au concepteur de décrire ce comportement ? Ce sont des questions ouvertes, difficiles à répondre. Seule l'expérimentation pourra y répondre.

Il est aussi légitime de se demander si le langage obtenu après ajout de la description du comportement de l'application reste dans la classe de langages d'origine, à savoir les ADL. Bien entendu, la portée de ces langages n'est pas bien définie. Chaque langage ayant des objectifs différents (vérification de propriétés, simulation), le contexte d'utilisation des ADLs couvre un large éventail de possibilités. Nous pourrions donc considérer la fusion entre la description de la structure et du comportement des architectures logicielles comme naturelle. Toutefois, une nouvelle classe de langages s'intéressant à la description des changements de l'architecture est en train d'émerger. De tels langages s'orientent vers l'expression de modifications à apporter à des systèmes déployés, en cours d'exécution. C2 [MTW96, MORT96, OMT98, OT98] entre dans le cadre de ces « Architecture Modification Language ». Hélas, les propositions du projet C2, et d'une manière générale celles des AML, ne sont pas encore bien détaillées dans la littérature et il est difficile de comparer leur approche à celle des ADL dynamiques. Dans quel cadre nous plaçons nous, ADL ou AML ?

6.5.2 Modélisation de l'Action

Un choix de conception assez difficile a été soulevé concernant la partie Action des règles. Trois solutions ont été envisagées, chacune présentant des atouts particuliers.

La première a entre autre l'avantage d'inclure les règles entièrement dans le composant composite, et ne demande pas une définition extérieurs. Cette intégration totale offre plus de facilités d'implémentation. En revanche, le vocabulaire utilisé au sein des règles est limité. Les éléments de configuration et de reconfiguration étendus aux opérateurs d'accès aux services des composants ne sont pas suffisants. L'utilisateur ne peut, par exemple, pas exprimer des modifications sur les paramètres du système (*p.ex.* l'horloge) ou effectuer des calculs mathématiques. Mais, il reste à définir si ce genre d'opérations est souhaitable et si l'on ne s'égare pas un peu...

La deuxième solution apporte, quant à elle, l'avantage de ne pas limiter la capacité d'expression des actions. La puissance d'expression des actions repose effectivement sur le code encapsulé dans le composant chargé du contrôle de l'application. Les actions sont donc exprimées au moyen de langages de programmation classiques. Par ailleurs, le mécanisme de règles (et notamment la

partie Action) fait partie de l'architecture de l'application, et peut donc potentiellement interagir avec les autres composants du système (et donc accéder à leurs services). La réutilisabilité des composants n'est par ailleurs pas touchée, le composant composite étant fourni avec l'ensemble de ses sous-composants (dont celui de contrôle). Le gros désagrément de cette approche est essentiellement dû au fait que la règle est éclatée, elle n'est pas centralisée dans le composite. La maintenance n'est alors pas facilitée.

Le principe du script constitue la dernière solution proposée. Il semble être le moins contraignant à utiliser, de par la liberté d'expression qu'il offre et par sa programmation proche de l'environnement d'exécution. Le principal apport de cette approche est relatif à l'utilisation d'un langage de script prédéfini. Ainsi, il n'est pas nécessaire d'introduire une nouvelle forme de langage, ou d'étendre un existant, comme cela était fait dans la première solution. Cette orientation permettrait l'utilisation de tous les opérateurs nécessaires, sans avoir besoin de les redéfinir (*p.ex.* `for`, `loop`). Néanmoins, la réutilisabilité des composants est détériorée, le script devant être fourni avec le composite. La description de l'Action est dans cette solution totalement à l'extérieur du composite, n'est-il pas possible d'aller un pas plus loin ? Pourquoi ne pas placer la règle entière à l'extérieur, la définir dans un script ? Il nous semble que cette implantation est intéressante mais elle soulève un certain nombre de difficultés, parmi lesquelles un manque de connaissance du système.

6.5.3 Simplifications apportées

Notre étude a porté son attention sur les possibilités offertes par le modèle d'exécution des règles actives. L'extrême simplicité du modèle Événement-Condition-Action n'est qu'apparente, elle masque en réalité de nombreuses questions auxquelles le modèle d'exécution doit répondre : Quand exécuter une règle par rapport à l'événement qui l'a déclenchée ? Que faire lorsqu'une règle est déclenchée par plusieurs événements ? Que faire lorsque plusieurs règles sont déclenchées simultanément ? Il est difficile de traiter ces questions de manière satisfaisante. Nous avons donc laissé les supports d'exécution nous imposer leur sémantique d'exécution simple (déclenchement par instance, consommation des événements locale, couplage immédiat-immédiat).

Les événements déclencheurs de nos règles actives sont une combinaison d'éléments appartenant à un vocabulaire prédéfini. Nous n'avons pas permis aux utilisateurs de définir leurs propres événements. Si nous l'avions fait, nous aurions perdu un aspect essentiel, à savoir le typage des événements. Gérer l'exécution des règles aurait alors été une tâche ardue. Par exemple, si l'événement correspondant au `CREATE` est décrit dans une classe Java, il faut pouvoir produire l'événement (et donc instancier la classe) au moment de création de composants particuliers. Or comment exprimer la sémantique associée à la classe afin de pouvoir effectuer son instanciation au moment opportun ?

Chapitre 7

Conclusion

7.1 Objectif et démarche de travail

L'évolution des systèmes informatiques vers des configurations réparties est un phénomène général qui concerne tous les secteurs d'activités. Elle est favorisée par la généralisation et l'accessibilité des réseaux de communication ainsi que la mobilité des machines. Paradoxalement, les applications distribuées dynamiques sont encore peu répandues, en raison de la complexité de programmation et du manque d'outils pour le développement de telles applications. Ce besoin de dynamicité et d'administration se fait de plus en plus ressentir et conduit à reconsidérer les méthodes de développement des applications.

L'administration d'applications réparties demande une connaissance aigüe de l'architecture des applications. L'ensemble de cette connaissance peut être initialement exprimée au niveau des langages de description d'architecture. Leur but est de fournir une certaine vision de l'architecture de l'application en termes d'entités logicielles nécessaires au fonctionnement de l'application, et de leurs intercommunications. Ces langages sont traditionnellement utilisés dans une phase de construction de l'application. Notre objectif était de prendre en compte des stratégies de contrôle et d'administration de manière déclarative au sein des ADL pour pallier le problème de la dynamicité des applications.

Ce travail a donc débuté par l'étude de ces langages. Notre intérêt s'est particulièrement porté sur les propositions d'expression de la dynamique des applications, c'est-à-dire l'évolution en cours d'exécution de la configuration des applications. Nous avons ensuite ciblé notre effort sur une mise en œuvre d'un exemple d'application nécessitant des reconfigurations dynamiques (instanciations de composants, création de liens dynamiques) au sein de l'environnement Olan. Cette expérimentation a souligné les faiblesses de son langage de description d'architecture, OCL, en termes de dynamicité. Cette démarche nous a permis de nous plonger directement dans un environnement avant de proposer des solutions concrètes. Nous avons ainsi acquis une vision plus claire de la problématique.

Nous avons précisé un certain nombre d'hypothèses liées à notre environnement d'exécution avant de nous plonger au cœur de notre travail. Il a principalement consisté à définir des règles pour le langage de description OCL,

permettant d'exprimer l'évolution dynamique d'une architecture logicielle en environnement réparti. Notre modèle s'appuie sur les travaux effectués en base de données. Plus précisément, il utilise des règles actives basées sur le formalisme E-C-A (Événement-Condition-Action). Ce modèle réactif apporte toute la souplesse nécessaire pour exprimer le comportement des applications à l'exécution. Notre travail a ainsi donné lieu à un ensemble de réflexions et spécifications pour l'introduction de règles actives dans OCL. Des solutions ont été proposées pour les différents problèmes soulevés. L'intégration de cette proposition au sein d'Olan est le prochain pas à franchir. Son implémentation est prévue pour les mois à venir. L'utilisation de règles actives pour exprimer la dynamique des applications pourra de ce fait être validée par l'expérimentation.

Pour cette proposition, nous nous sommes limités à un modèle d'exécution simple. Il est induit par le support d'exécution et n'est pas configurable par l'utilisateur. Nous n'avons pas non plus exploré toutes les possibilités offertes par le modèle E-C-A. La partie Action pourrait par exemple être exprimée différemment selon l'utilisation que l'on veut en faire.

7.2 Évaluation

L'apport de dynamique dans OCL fait d'Olan un environnement très riche et complet. En effet, il combine les avantages de plusieurs ADL :

- UniCon pour la vérification statique des types ;
- Wright pour son vocabulaire de reconfiguration ;
- Darwin pour ses schémas d'instanciations dynamiques.

Les aspects dynamiques proposés dans OCL sont principalement la création et la suppression de composants et de connecteurs ainsi que les modifications d'attributs pilotées par l'application pendant l'exécution. Le concept de collection, qui n'existe pas dans Darwin, permet de représenter les ensembles de composants dont le dénominateur commun est la même interface. Cette facilité permet de pouvoir s'adresser aux instances de composants créées lors de l'exécution de manière dynamique. Outre son aspect dynamique, OCL permet la répartition des composants et produit une image exécutable de l'application.

Cependant, l'utilisation d'OCL ne semble pas aussi aisée que Rapide. Les événements susceptibles de se produire sont prédéfinis et ne correspondent pas forcément à l'attente des architectes. Par ailleurs, la gestion des règles nous est imposée par le support d'exécution de l'application, alors que Rapide permet le contrôle de leur évaluation (*p.ex.* en ordonnant causalement les événements). En outre, notre proposition n'est ni extensible, ni adaptable. Nos règles actives permettent d'exprimer la reconfiguration de l'application à l'exécution. Toutefois, nous ne pouvons pas définir de nouvelles règles pendant l'exécution, ni en supprimer (extensibilité). Il est de même impossible de les activer ou désactiver afin qu'elles soient ou non déclenchables, et le modèle d'exécution n'est pas manipulable (adptabilité).

Notre proposition a permis de faire une étude approfondie des problèmes (formalisme, expression de la configuration dynamique) que pose la mise en oeuvre d'un service d'administration et de reconfiguration et d'éclaircir leur

portée en proposant un éventail de solutions. Le formalisme adopté semble par ailleurs convenir, il permet l'expression de la dynamique souhaitée. La contribution majeure de notre travail est de permettre une intégration simple de règles avec un grand pouvoir d'expression de la dynamique. L'étude que nous avons faite nous a apporté une bonne connaissance des mécanismes à mettre en oeuvre à la compilation, le travail d'implémentation n'en sera que plus facile.

7.3 Perspectives

De nombreuses voies de recherches sont ouvertes, les perspectives de travail sont en effet nombreuses. À court terme, il faudrait prévoir la gestion des erreurs et vérifier statiquement l'expression des règles. Dès que l'implémentation du système de règles sera intégrée à Olan, nous serons capable de considérer ces problèmes. Nous envisageons aussi d'établir une spécification formelle de la grammaire avant de se lancer dans l'implémentation. Enfin, il faudrait proposer un modèle d'exécution plus complet et adaptable.

Le problème essentiel mis en évidence par notre travail est une question ouverte : comment garantir l'intégrité des actions ? Les reconfigurations dynamiques posent des problèmes relatifs à la suppression des composants ou des connexions. Comment savoir lors de l'exécution d'une règle si les entités concernées (composants ou connexions) sont encore présentes à ce moment de l'exécution. Le problème n'est pas soulevé quand les reconfigurations affectent l'architecture initiale. Mais dans le cas où les règles sont exécutées sur une configuration qui a déjà évolué, comment faire ?

Rapide pallie ce problème en introduisant des quantificateurs universels (il existe, pour tout) dans sa grammaire, ce qui lui permet d'effectuer des modifications seulement sur les éléments architecturaux présents. La meilleure solution que l'on peut envisager est de définir un service de règles actives. Ce service pourrait offrir un modèle d'exécution fixant certains aspects, parmi lesquels le mode de traitement des événements, la sémantique ensembliste ou par instance, *etc.* Il pourrait aussi garantir la cohérence et l'intégrité de l'application. L'implémentation d'un tel service n'étant pas forcément notre spécialité, nous pourrions de par notre situation limitrophe avec les bases de données, utiliser des systèmes déjà prédéfinis. Néanmoins, un tel service n'existe pas forcément, un gros travail d'adaptation serait à fournir. Effectivement, ce service garantit bien la cohérence et l'intégrité dans le domaine des bases de données, mais dans les applications réparties, nous ne possédons pas la vue ensembliste qui leur est propre, et cela rend les choses d'autant plus difficiles.

Bibliographie

- [ADG98] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998.
- [AG94a] R. Allen and D. Garlan. Beyond definition/use : Architectural interconnection. In *Proceedings of the ACM Interface Definition Language Workshop*, volume 29(8). SIGPLAN Notices, August 1994.
- [AG94b] R. Allen and D. Garlan. Formal connectors. *CMU Tech. Report CMU-CS-94-115*, March 1994. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA15213, USA.
- [AG94c] R. Allen and D. Garlan. Formalizing architectural connection. In *Sixteenth International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994.
- [AG97] R. Allen and D. Garlan. A formal basis for architectural connection. In *ACM Transactions on Software Engineering and Methodology*, July 1997.
- [All97] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.
- [BABR96] L. Bellissard, S. B. Atallah, F. Boyer, and M. Riveill. Distributed application configuration. In *Proceedings of the 16th International Conference on Distributed Computing Systems ICDCS'96*, pages 579–595, Hong-Kong, May 1996.
- [BAKR95] L. Bellissard, S. B. Atallah, A. Kerbrat, and M. Riveill. Component-based programming and application management with Olan. In *Proceedings of Workshop on Object-based Parallel and Distributed Computation*, Tokyo, Japan, June 1995. LNCS Springer Verlag.
- [Bel97] L. Bellissard. *Construction et Configuration d'Applications Réparties*. PhD thesis, Institut National Polytechnique de Grenoble, ENSIMAG, Décembre 1997.
- [BPF] L. Bellissard, N. De Palma, and D. Féliot. The Olan architecture definition language. Internal Report. Version 2.1.
- [BPF⁺99] L. Bellissard, N. De Palma, A. Freyssinet, M. Herrmann, and S. Lacombe. An agent platform for reliable asynchronous distributed

- programming. In *Symposium on Reliable Distributed Systems*, Lausanne, Suisse, October 1999.
- [BRJ97] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison Wesley, December 1997. See www.awl.com/cp/uml/uml.html.
- [CC96] C. Collet and T. Coupaye. Primitive and composite events in NAOS. In *12ièmes Journées Bases de Données Avancées (BDA '96)*, Cassis, France, September 1996.
- [Col96] C. Collet. *Bases de Données Actives : Des Systèmes Relationnels aux Systèmes à Objets*. LSR-IMAG, Université Joseph Fourier, October 1996. Diplôme D'Habilitation à Diriger les Recherches.
- [Cou96] T. Coupaye. *Un modèle d'exécution paramétrique pour systèmes de bases de données actifs*. PhD thesis, Université Joseph Fourier, LSR-IMAG, Grenoble, France, November 1996.
- [CVSGR] C. Collet, G. Vargas-Solar, and H. Grazziotin-Ribeiro. *Open Active Services for Data-Intensive Distributed Applications*. IMAG-LSR.
- [GGM97] J-M. Geib, C. Gransart, and P. Merle. *Corba : des concepts à la pratique*. Collection InterEditions, Editions Masson, October 1997.
- [GMW97] D. Garlan, R. T. Monroe, and D. Wile. Acme : An architecture description interchange language. In *CASCON'97*, pages 169–183, Ontario, Canada, November 1997.
- [GP95] D. Garlan and D. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.
- [GS94] D. Garlan and M. Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.
- [GW98] D. Garlan and Z. Wang. A case study in software architecture interchange. Unpublished, March 1998.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [ISZ98] V. Issarny, T. Saridakis, and A. Zarras. A survey of architecture description languages. Technical report, C3DS Project, June 1998.
- [LKA⁺95] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. In *IEEE Transactions on Software Engineering*, volume 21 n.4, pages 336–355, April 1995.
- [LS97] E. Lupu and M. Sloman. A policy based role object model. In *Proceedings of EDOC'97*, Queensland, Australia, October 1997.
- [LS98] G. F. Luger and W. A. Stubblefield. *Artificial Intelligence. Structures and Strategies for Complex Problem Solving*. Addison-Wesley, 3rd edition, 1998.

- [Luc96] D. C. Luckham. Rapide : A language and toolset for simulation of distributed systems by partial orderings of events. In *Technical Report CSL-TR-96-705*, Stanford University, Computer Systems Laboratory, September 1996.
- [LV95] D. C. Luckham and J. Vera. An event-based architecture definition language. In *IEEE Transactions on Software Engineering*, volume 21 n.9, pages 717–734, September 1995.
- [LVM95] D. C. Luckham, J. Vera, and S. Meldal. Three concepts of system architecture. Unpublished, July 1995.
- [MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Fifth European Software Engineering Conference (ESEC'95)*, Barcelona, September 1995.
- [MDK94] J. Magee, N. Dulay, and J. Kramer. Regis : A constructive development environment for distributed programs. In *IEEE Distributed Systems Engineering Journal*, volume 1 n.5, pages 304–312, December 1994.
- [Men97] T. Menzies. Object-oriented patterns : Lessons from expert systems. volume 27(12), pages 1457–1478. *Software Practice and Experience*, December 1997.
- [MK96] J. Magee and J. Kramer. Dynamic structure in software architectures. In *ACM SIGSOFT'96 : Fourth Symposium on the Foundations of Software Engineering*, pages 3–14, San Fransisco, CA, October 1996.
- [MORT96] N. Medvidovic, P. Oreizy, E. Robbins, J., and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 - style. In *Fourth ACM Symposium on the Foundations of Software Engineering*, pages 24–32, San Fransisco, CA, October 1996.
- [MSS97] M. Mansouri-Samani and M. Sloman. Gem - a generalised event monitoring language for distributed systems. *Distributed Systems Engineering Journal*, 4(2), June 1997.
- [MT97] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Sixth European Software Engineering Conference*, pages 60–76, Zurich, Switzerland, September 1997.
- [MTW96] N. Medvidovic, R. N. Taylor, and E. J. Jr. Whitehead. Formal modeling of software architectures at multiple levels of abstraction. In *California Software Symposium*, pages 28–40, Los Angeles, CA, April 1996.
- [OMT98] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *International Conference on Software Engineering (ISCE'98)*, Kyoto, Japan, April 1998.
- [OT98] P. Oreizy and R. N. Taylor. On the role of software architectures in runtime system reconfiguration. In *International Conference on Configurable Distributed Systems (ICCDs4)*, Annapolis, Maryland, May 1998.

- [PBF⁺99] N. De Palma, L. Bellissard, D. Féliot, A. Freyssinet, and S. Lacomte. An agent-based message-oriented middleware. Internal Report, 1999.
- [Qui] A. Quinn. Javabeans : Components for the java platform. <http://java.sun.com/docs/books/tutorial/javabeans/index.html>.
- [RM96] D. Richards and C. McDonald. Changing the system style of expert systems. In *Proceedings of AI'95 - Workshop on AI and the Environment*, pages 33–40, ADFA Canberra, November 1996.
- [RM99] M. Riveill and P. Merle. *La programmation par composants*. École d'Été : Construction d'Applications Réparties, Autrans, France, August 1999.
- [SDK⁺95] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. In *IEEE Transactions on Software Engineering*, volume 21 n.4, pages 314–335, April 1995.
- [SDZ96] M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations for architectural connections. In *Third International Conference on Configurable Distributed Systems*, May 1996.
- [Slo94] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2(4), 1994.
- [Tho] A. Thomas. Enterprise java beans : Server component model for java. <http://java.sun.com/products/ejb/white-paper.html>.
- [Yam98] T. Yamaguchi. Desire : An interoperative environment for distributed expert systems. In *ECAI'98 Workshop on Applications of Ontologies and Problem-Solving Methods*, 1998.

Annexe A

Utilisation simplifiée de CSP

En CSP, les processus sont définis par des suites d'événements. Alors que cette algèbre possède un ensemble riche de concepts pour la description d'entités communicantes, nous n'utilisons qu'un sous-ensemble incluant :

Des processus et événements Un processus décrit une entité qui peut s'engager dans des communications événementielles. Les événements peuvent être primitifs ou ils peuvent être associés à des données (*p. ex.* $P.e?x$ et $P.e!x$ représentent des données d'entrées et de sorties respectivement sur le port P). L'événement \surd est utilisé pour représenter la terminaison réussie d'un processus ;

Des préfixes Un événement e précédant le processus P est noté $e \rightarrow P$;

Des choix déterministes Le choix entre P ou Q , fait par l'environnement, est noté $P \sqcap Q$. L'environnement correspond aux autres processus interagissant avec le processus courant ;

Des choix non-déterministes Le choix entre P ou Q , fait par le processus lui-même est noté $P \sqcup Q$.

Dans les expressions de processus, \rightarrow est associative à droite et prioritaire par rapport à \sqcap ou \sqcup .

De plus, pour pouvoir utiliser des noms locaux, les λ -expressions, telles $P = \text{let } Q = \text{expr in } R$, sont introduites.