Modèles et environnement pour configurer et déployer des systèmes logiciels

Vincent Lestideau

▶ To cite this version:

Vincent Lestideau. Modèles et environnement pour configurer et déployer des systèmes logiciels. Génie logiciel [cs.SE]. Université de Savoie, 2003. Français. NNT : . tel-00005525

HAL Id: tel-00005525
https://theses.hal.science/tel-00005525

Submitted on 4 Apr 2004

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

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

THESE

présentée à

L'UNIVERSITE DE SAVOIE
ECOLE SUPERIEURE D'INGENIEURS D'ANNECY (ESIA)

par

Vincent LESTIDEAU

pour obtenir le

DIPLÔME DE DOCTEUR DE L'UNIVERSITE DE SAVOIE

(Arrêté ministériel du 30 Mars 1992)

spécialité :

Informatique

Modèles et environnement pour configurer et déployer des systèmes logiciels

---

Soutenue publiquement le 19 décembre 2003 devant le jury composé de :

J.-C. Derniame	Professeur, Institut National Polytechnique de Lorraine	(Président – Rapporteur)
J.-M. Andreoli	Directeur de recherche, CNRS associé, Xerox Research Centre Europe	(Rapporteur)
N. Belkhatir	Professeur, IMAG (Grenoble)	(Directeur de thèse)
F. Oquendo,	Professeur, Université de Savoie	(Directeur de thèse)
R. Balter	Vice-Président et Responsable Scientifique, ScalAgent Distributed Technologies (Echirolles)	(Examinateur)
F. Bernigaud	Industriel, Actoll (Meylan)	(Examinateur)

Thèse préparée au sein du LISTIC et du LSR/ADELE

A ma famille

Remerciements

Je tiens à remercier les membres du jury :

M. Jean-Claude Derniame et M. Jean-Marc Andreoli pour avoir accepté d'être rapporteurs de thèse. Je les remercie pour leur lecture attentive ainsi que pour les remarques et critiques qu'ils ont fait à ce travail.

M. Roland Balter et M. François Bernigaud pour avoir accepté d'être membre du jury de cette thèse.

M. Noureddine Belkhatir pour m'avoir dirigé et soutenu durant ces années de thèse. Merci de m'avoir fait découvrir le monde de la recherche ainsi que celui de l'enseignement.

M. Flavio Oquendo pour m'avoir encadré et conseillé et ceci malgré l'éloignement géographique.

Je voudrais tout d'abord remercier M. Pierre-Yves Cunin pour son aide et ses conseils tout au long de ce travail, ainsi que pour son travail important de relecture du document.

Merci à M. Jacky Estublier et à tous les membres de l'équipe Adèle pour leur accueil et pour m'avoir offert les moyens et l'ambiance nécessaires à l'aboutissement de ce travail.

Merci à tous les membres du groupe déploiement (Madjid, Noëlle, Hacene, Laura, Noureddine, Pierre-Yves) avec qui j'ai partagé le quotidien de la recherche avec ses périodes de crises et d'échanges.

Je remercie Frédéric, Rémy, Laurent, Humberto, Ahn-Tuyet, Sonia, Jean-Marie, Mikaël, Jorge, German, Didier, Samer et quelques autres pour l'aide et le bon temps que l'on a passé ensemble.

Merci aux membres de la société Actoll pour les critiques apportés à ce travail et les échanges fructueux.

J'adresse mes remerciements aux membres de l'équipe Listic pour m'avoir toujours accueilli dans la bonne humeur lors de mes passages à Annecy. Un merci plus particulier pour Valérie pour son soutien administratif.

A celles et ceux, qui par leurs actions, leurs témoignages, ont permis la réalisation de ce travail.

Résumé

Le cycle de vie du logiciel regroupe plusieurs activités comme l'analyse, la conception, la production, le test et le déploiement. Cette dernière activité est un procédé complexe composé de sous activités comme la configuration, l'installation ou la mise à jour. La complexité et l'importance du déploiement a augmenté récemment avec l'évolution des réseaux et la construction d'applications à base de composants. Il est maintenant envisageable d'imaginer des solutions permettant le déploiement automatisé de logiciels en assurant que chaque utilisateur recevra la version du logiciel la plus cohérente et la mieux adaptée à ses besoins et à son environnement tout en respectant les stratégies de déploiement de l'entreprise. Il existe de nombreuses approches et outils de déploiement, mais très peu permettent de couvrir entièrement le cycle de vie du déploiement ou alors en imposant des contraintes fortes.

Cette thèse propose un environnement de déploiement nommé ORYA, c'est à dire une plate-forme offrant un support automatisé aux activités du cycle de vie du déploiement. ORYA est basé sur la réutilisation et l'intégration des outils de déploiement existants. Pour cela, nous proposons une abstraction des différents acteurs et entités du déploiement, ainsi qu'une infrastructure permettant de faire interopérer des outils hétérogènes et ne se connaissant pas. Ce travail traite plus particulièrement de l'activité de sélection avec l'utilisation d'un modèle de composant générique et la mise en place d'un framework basé sur un système d'annotations et de règles. La deuxième activité étudiée en détail est celle de l'installation qui est basée sur un langage de procédés permettant la description et la réalisation des procédés dans le monde réel.

Une implémentation d'ORYA a été réalisée afin de valider notre approche dans le cadre d'une expérimentation industrielle en vraie grandeur..

Mots clés : Déploiement, Procédés, Environnement, Automatisation, Cycle de vie, Modèles à composants, Gestion de configuration, Langages.

Abstract

Software life cycle encompasses many activities: analysis, design, production, test and deployment. This last activity is itself a complex process composed of sub-activities like configuration, installation and update. Complexity and importance of deployment are both increased by network evolution and component based applications. It is possible to imagine some solutions allowing an automated deployment by providing to each user the most consistent and adapted software version, with respect to deployment strategies of the enterprise and user needs. There are many approaches and tools for deployment however few allow to cover all the deployment life cycle usually imposing some strong constraints.

This thesis proposes a deployment environment named ORYA: an automated support for deployment life cycle activities. ORYA is based on the reuse and integration of existing deployment tools. We propose an abstraction of the various deployment actors and entities as well as an infrastructure allowing the interoperation of heterogeneous tools. This work focuses more particularly on the selection activity using a generic component model and a framework based on an annotation system and some rules. The second activity studied in this thesis is the installation, which is based on a process language allowing the process description and its realization in the real world.

The ORYA prototype has been validated in the context of an industrial experimentation.

Keywords: Deployment, Process, Environment, Automation, Life Cycle, Component Models, Configuration Management, Languages.

Table des matières

Chapitre I Introduction 17
1. Introduction.. 18
2. Le domaine de la thèse : le déploiement à large échelle.................................... 19
   2.1. Applications industrielles 19
   2.2. Déploiement vers des entreprises 19
3. Les objectifs de la thèse 20
4. Organisation de la thèse 20

Chapitre II Le déploiement 23
1. Introduction au déploiement 24
2. Scénario de motivation 24
3. Le cycle de vie du déploiement 25
   3.1. Introduction 25
   3.2. Les différentes activités 26
   3.3. Déploiement versus adaptation dynamique 29
   3.4. Conclusion 30
4. Les différentes problématiques liées au déploiement 30
   4.1. La gestion du cycle de vie du déploiement.. 30
   4.2. La gestion de configuration 30
   4.3. L'évolution de l'environnement 30
   4.4. Les relations entre les applications 31
   4.5. Les réseaux 31
   4.6. L'hétérogénéité des plate-formes 31
   4.7. La sécurité... 32
   4.8. Conclusion 32
5. Architecture à 3 niveaux 32
   5.1. Introduction 32
   5.2. Niveau producteur 33
   5.3. Niveau entreprise 34
   5.4. Niveau utilisateur 35
   5.5. Conclusion 35
6. Synthèse 35

Chapitre III Etat de l'art et de la pratique 37
1. Introduction... 38
2. la gestion de configuration.. 38
   2.1. Introduction 38
   2.2. Revision Control System 39
   2.3. Adele 39
   2.4. Conclusion 40
3. Les procédés logiciels 40
   3.1. Introduction 40
   3.2. La modélisation de procédé 40
   3.3. Les langages de modélisation des procédés 42
   3.4. Les environnements de procédés 42
   3.5. Conclusion 43

4. Les modèles à composant ... 43
4.1. Introduction ... 43
4.2. Les EJBs ... 44
4.3. Corba Component Model ... 48
4.4. Dot Net ... 50
4.5. Bilan du déploiement des composants ... 52
5. Les environnements de déploiement ... 52
5.1. Introduction ... 52
5.2. SoftwareDock ... 52
5.3. Tivoli ... 55
5.4. InstallShield ... 57
5.5. JavaWebStart ... 59
5.6. Bilan sur l'état de la pratique ... 60
6. Conclusion ... 61

Chapitre IV Notre approche : ORYA ... 63
1. Motivations ... 64
2. La gestion du cycle de vie ... 65
2.1. La couverture du cycle de vie ... 65
2.2. L'automatisation du déploiement ... 65
2.3. La modification de procédés ... 66
2.4. La coordination ... 66
2.5. Conclusion ... 66
3. Une approche basée sur l'abstraction ... 66
3.1. Le principe ... 67
3.2. Conclusion ... 67
4. La réutilisation ... 68
4.1. La problématique ... 68
4.2. Le problème de l'interopérabilité ... 68
4.3. Le problème de dépendance vis à vis des outils ... 69
4.4. Le problème du comportement des outils ... 69
4.5. Le problème de l'exécution à distance ... 70
4.6. Le problème du partage des ressources ... 70
4.7. Conclusion ... 70
5. Notre environnement : ORYA ... 71
5.1. Architecture ... 71
5.2. Les entités ... 71
5.3. La technologie des fédérations ... 72
5.4. Conclusion ... 74
6. Conclusion sur l'approche ... 75

Chapitre V La sélection ... 77
1. Introduction ... 78
2. La sélection ... 78
2.1. Définitions ... 78
2.2. Le choix des modèles de composant ... 79
2.3. Le choix des annotations ... 80
3. Le modèle de composant ... 80

3.1. Les objectifs de ce modèle 80
3.2. Principes 80
3.3. Le méta-modèle de composant 81
3.4. Conclusion sur le modèle 88
4. Le framework d'annotation 89
4.1. Principes 89
4.2. Les dépendances entre les annotations 89
4.3. Le modèle d'annotation 90
4.4. Exemple 92
4.5. Conclusion sur le framework d'annotation 94
5. La construction des configurations 94
5.1. Principes 94
5.2. La requête 94
5.3. La qualification des attributs 95
5.4. Les contraintes de sélection 96
5.5. L'algorithme de sélection 96
5.6. Exemple d'une configuration 98
6. Le résultat 102
6.1. Principes 102
6.2. Le modèle d'application 103
6.3. Le descripteur d'application 103
6.4. Les "packages" 103
7. Résumé 104

Chapitre VI L'installation 105
1. Introduction 106
2. Le problème de l'installation 106
2.1. Principe 106
2.2. Les solutions au niveau réel 107
2.3. Les solutions au niveau modèle 107
2.4. Le framework d'installation 108
3. Le monde de l'environnement de déploiement 110
3.1. Intérêt 110
3.2. Le méta-modèle de l'environnement 111
3.3. Les sites 112
3.4. Les serveurs d'applications 114
3.5. Le serveur de déploiement 114
3.6. L'entreprise 116
4. Le monde des procédés 117
4.1. Introduction 117
4.2. Meta Modèle des procédés 117
4.3. Le moteur de procédés 118
4.4. Le langage de description d'un procédé 119
5. La réalisation de la solution 121
5.1. Le principe 121
5.2. Les actions 121
5.3. L'utilisation des rôles 122
5.4. Les manipulations sur les produits 122
5.5. Les manipulations sur les procédés 124

5.6. Conclusion sur ce langage 125
6. Utilisation : Création & modification de procédé........ 126
6.1. La création de procédé........ 126
6.2. La modification dynamique de procédé 129
6.3. Conclusion sur l'utilisation de procédés........ 130
7. Résumé de l'activité d'installation 131

Chapitre VII L'implémentation 133
1. Introduction........ 134
2. L'architecture d'ORYA 134
2.1. Introduction 134
2.2. Les serveurs d'applications 134
2.3. Le serveur de déploiement........ 137
2.4. Les sites 138
2.5. Conclusion sur l'architecture 140
3. La création de configurations 140
3.1. Introduction 140
3.2. La gestion des composants dans la base de développement........ 141
3.3. La gestion des annotations........ 143
3.4. La construction des configurations........ 144
3.5. La description des applications........ 145
3.6. Conclusion sur la partie sélection 145
4. La gestion des procédés 145
4.1. Introduction 145
4.2. Le langage graphique de description 146
4.3. Le langage de réalisation 147
4.4. L'exécution des procédés 151
4.5. Conclusion 152
5. Un exemple de procédé d'installation 152
5.1. Introduction 152
5.2. Le descripteur d'installation........ 153
5.3. Le procédé d'installation........ 155
5.4. Conclusion sur le procédé d'installation 160
6. Les outils associés........ 160
6.1. L'éditeur de déploiement 160
6.2. L'outil de packaging........ 161
6.3. L'agenda........ 162
6.4. La gestion des traces du déploiement 162
7. Conclusion sur l'implémentation 164

Chapitre VIII Expérimentation et validation 165
1. Introduction........ 166
2. L'application Centr'Actoll........ 166
2.1. Description de l'application 166
2.2. La plate-forme de démonstration........ 167
3. Les exigences liées au déploiement........ 168
3.1. Introduction 168
3.2. La configuration 168

3.3. La distribution 168
3.4. L'automatisation 169
3.5. Le monitoring 169
3.6. Le "roll back" 169
3.7. La sécurité 169
3.8. Les tests et vérifications 169
3.9. Conclusion 169
4. L'expérimentation 170
4.1. Introduction 170
4.2. Analyse des besoins 170
4.3. Un outil de déploiement spécifique 171
4.4. Un outil de déploiement générique 172
4.5. Conclusion 172
5. La validation 173
5.1. Introduction 173
5.2. Déploiement d'un composant sur une machine 173
5.3. Déploiement multiple d'un composant 174
5.4. Déploiement d'un composant et d'une dépendance 176
5.5. Déploiement via Internet 176
5.6. Conclusion 178
6. Bilan 178

Chapitre IX Conclusion et perspectives 179
1. Synthèse de l'approche 180
1.1. Identification de l'architecture de déploiement 180
1.2. Proposition d'un environnement basé sur la réutilisation 181
1.3. Proposition d'automatisation du déploiement 181
1.4. Conclusion sur l'approche 181
2. Les Perspectives 182
2.1. Les autres activités du déploiement 182
2.2. La gestion du procédé de déploiement 182
2.3. Les politiques de déploiement 182
2.4. L'industrialisation d'ORYA 182
3. Conclusion 183

Chapitre X Bibilographie 185

Chapitre XI Annexes 193
1. Exemple du langage de description des procédés 194
2. Le modèle de description d'un serveur d'applications 196
3. Le modèle de description d'un site 196
4. Description d'un manifeste 197
5. Template de l'activité GetFile 197
6. Template de l'activité Transfert 198
7. Template de l'activité PutFile 199
8. Le manifeste d'installation 199
9. Un exemple de manifeste d'installation 200

Liste des figures

Figure 1 : Exemple de la structuration d'une entreprise 24
Figure 2 : Cycle de vie du déploiement logiciel 26
Figure 3 : les sous activités de l'activité de configuration et de sélection 26
Figure 4 : les sous-activités de l'activité d'installation 27
Figure 5 : Architecture à 3 niveaux 33
Figure 6 : Les concepts de base de la modélisation de procédés 41
Figure 7 : L'architecture conceptuelle d'un PSEE 43
Figure 8 : Les différents rôles du cycle de vie des applications EJB 45
Figure 9: l'architecture physique d'ORYA 71
Figure 10 : Exemple de description graphique d'un procédé 73
Figure 11 : description de la connexion d'un outil dans une fédération 74
Figure 12 : Modèle simplifié du modèle de composant 81
Figure 13 : Représentation d'un rôle 82
Figure 14 : Représentation d'une implémentation native 82
Figure 15 : Représentation externe d'une implémentation composite 82
Figure 16 :Exemple de la représentation interne d'une implémentation composite 83
Figure 17 : Exemple d'implémentations 84
Figure 18 : Exemple de raffinements 85
Figure 19 : Exemple de composants et de relations 86
Figure 20 : Exemple de relations d'occurrence 88
Figure 21 : Interdépendance entre annotation 89
Figure 22 : Le modèle d'annotation 90
Figure 23 : Interdépendance entre attributs 91
Figure 24 : Exemple d'annotation 92
Figure 25 : Exemple d'exécution de stratégie de vérification 93
Figure 26 : Exemple d'application des stratégies sur la relation d'occurrence 93
Figure 27 : exemple de requête 98
Figure 28 : Exemple d'annotations et de contraintes de sélection 99
Figure 29 : Exemple d'annotations 100
Figure 30 : Visualisation du parcours de sélection (partie 1) 101
Figure 31 : Visualisation du parcours de sélection (partie 2) 102
Figure 32 : Modèle d'application 103
Figure 33 : le niveau modèle 109
Figure 34 : le niveau du monde réel 109
Figure 35 : Architecture du framework 110
Figure 36 : Le méta-modèle de l'environnement de déploiement 111
Figure 37 : Le modèle de site 113
Figure 38 : Le modèle du serveur d'application 114
Figure 39 : Le modèle du serveur de déploiement 115
Figure 40 : Le modèle d'entreprise 116
Figure 41 : Le méta-modèle des procédés 117
Figure 42 : Exemple de procédé 120
Figure 43 : Exemple d'activité 123
Figure 44 : Exemple d'une activité de transfert 126
Figure 45 : Exemple d'une activité d'installation 127
Figure 46 : Exemple de réutilisation 127
Figure 47 : Exemple de modification 128

Figure 48 : Exemple de l’ajout d’une sous-activité 129
Figure 49 : Interface graphique d'un serveur d'applications 136
Figure 50 : Interface graphique du serveur de déploiement 138
Figure 51 : Interface graphique d'un site 140
Figure 52 : Interface de création d'une implémentation 142
Figure 53 : Interface d'exploration des composants 143
Figure 54 : Interfaces pour la sélection 144
Figure 55 : Description de l'activité de transfert 147
Figure 56 : Un exemple d'activité qui boucle 153
Figure 57 : Le procédé global d'installation 156
Figure 58 : L'activité composite d'installation 158
Figure 59 : La partie extraction de l'installation 159
Figure 60 : Les différentes étapes d'installation 159
Figure 61 : L'interface graphique de l'éditeur de déploiement 161
Figure 62 : L'outil de monitoring 162
Figure 63 : Exemple de traces d'un déploiement 163
Figure 64 : Outil de visualisation des traces 164
Figure 65 : Schéma général de l'application Centr'Actoll 167
Figure 66 : Architecture physique de la plate-forme 168
Figure 67 : Exemple de diagramme de séquence pour l'installation du Scar 171
Figure 68 : Architecture du scénario 1 174
Figure 69 : Architecture du scénario 2 175
Figure 70 : Architecture du scénario 4 177

Chapitre I

Introduction

1. Introduction

Depuis ces dernières années, l'intérêt pour le domaine du déploiement logiciel s'est fortement accru à la fois dans le monde de la recherche et dans celui de l'industrie. Mais avant de nous intéresser aux raisons de cette évolution, définissons d'abord ce que signifie le déploiement. Le déploiement logiciel concerne toutes les personnes disposant d'un ordinateur, en effet quand on installe une application sur sa machine, on réalise un déploiement. Mais contrairement aux idées reçues, le déploiement est une activité beaucoup plus complexe que celle qui consiste à installer une application en insérant un CD ou en la téléchargeant via Internet.

Il existe de nombreuses définitions du déploiement mais généralement on peut dire que le déploiement regroupe toutes les activités réalisées après le développement d'une application. Autrement dit, le déploiement couvre toutes les étapes depuis la validation de l'application par le producteur jusqu'à son installation puis sa désinstallation en passant par sa maintenance sur la machine cible (du client). L'ensemble de ces étapes forme ce que l'on appelle le cycle de vie du déploiement logiciel.

Cette définition du déploiement est déjà complexe alors qu'elle est incomplète, car le déploiement ne concerne pas seulement le déploiement d'une application sur une machine mais aussi la gestion d'un ensemble d'applications à déployer sur un ensemble de machines. C'est pourquoi le déploiement à la différence de l'installation fait intervenir non seulement les deux acteurs traditionnels que sont le *producteur* de logiciel et le *client*, mais aussi un acteur intermédiaire : *l'entreprise*. Chacun de ces acteurs a ses propres attentes vis à vis du déploiement. L'un des objectifs du déploiement est de satisfaire au mieux ces attentes en sachant qu'elles sont souvent différentes voir contradictoires :

- le client, c'est à dire l'utilisateur de la machine cible, désire avoir les applications les plus récentes, celles qui exploiterons le mieux les capacités intrinsèques de sa machine, c'est à dire les caractéristiques matérielles, celles qui correspondront le mieux à ses besoins et à son rôle dans l'entreprise. En effet, un chef de projet a des attentes en terme d'application sûrement différentes de celles d'un développeur par exemple,
- le producteur souhaite quand à lui répondre pour des raisons économiques à l'attente de ses clients et réduire au maximum le coût du déploiement. Pour simplifier le déploiement, une solution consisterait à développer une seule configuration par application, ce qui est en opposition avec le souhait de ses clients, qui souhaite quasiment une configuration personnalisée à chaque déploiement,
- le troisième acteur a lui aussi des attentes différentes des autres. L'entreprise souhaite que les activités de déploiement soient les plus transparentes possibles et les moins coûteuses possible à la fois en terme d'argent mais aussi en terme de perte d'activités et de ressources (matérielles et humaines). D'un autre côté, elle souhaite que chacun de ses employés dispose des configurations les plus pertinentes. De plus chaque machine doit disposer seulement des applications nécessaires à son utilisateur, ce qui permet de diminuer le nombre de licences.

Il est difficile de trouver une solution qui convient à tout le monde. Si le client gère lui-même son déploiement, il diminue sa productivité et si le producteur propose un service de mise à jour automatique, l'entreprise doit accepter de perdre le contrôle des activités de déploiement, ce qu'elle refuse pour des raisons évidentes de sécurité, de coût, de stratégies, etc..

Une des raisons majeures qui explique l'évolution de l'intérêt vis à vis du déploiement, c'est qu'avec l'évolution des réseaux et de la programmation à base de composants, les possibilités de créer et de distribuer (de façon plus ou moins automatisée) une configuration personnalisée de chaque application pour chaque machine deviennent de plus en plus grandes. Il est maintenant envisageable d'imaginer des solutions permettant le déploiement automatisé d'applications dans le parc informatique d'une entreprise. Il existe d'ailleurs déjà de telles solutions. Cette thèse se place dans ce contexte, c'est à dire celui du déploiement à large échelle.

2. Le domaine de la thèse : le déploiement à large échelle

Comme on vient de le dire, le contexte de cette thèse est celui du déploiement à large échelle, c'est à dire que nous nous intéressons au déploiement réalisé au sein d'une entreprise de grande taille composée de milliers de machines hétérogènes. L'un de nos objectifs est de déployer pour chaque machine les configurations les plus pertinentes pour chaque application. Pour cela, nous avons choisi de traiter le cas du déploiement des applications dites "industrielles". Ce type d'applications et la taille des entreprises sont deux des facteurs qui justifient la notion de déploiement à large échelle.

2.1. Applications industrielles

L'un des objectifs du déploiement consiste à assurer que la configuration logicielle de chaque machine de l'environnement soit la plus pertinente possible; pertinence par rapport aux caractéristiques de la machine, par rapport à la position (et les choix) de l'utilisateur de celle-ci, par rapport aux exigences (stratégies) de déploiement de l'entreprise concernée.

Dans ce contexte particulier, nous avons choisi d'étudier le déploiement dans le cadre des applications dites "industrielles". Une application de ce type est une application qui a les caractéristiques suivantes :

- versionnée. Il existe de nombreuses versions de l'application, chacune correspondant à des exigences ou fonctionnalités précises,
- évolutive. L'application est mise à jour régulièrement, que ce soit pour des raisons de corrections de "bugs" ou pour l'ajout de fonctionnalités,
- de grande taille. Chaque application est constituée de milliers de "composants".

Ces applications permettent d'aborder l'ensemble des problématiques liées au déploiement, comme par exemple, la gestion de configuration ou bien les problèmes de mise à jour de logiciels déployés.

2.2. Déploiement vers des entreprises

L'autre facteur qui justifie ce terme de large échelle, c'est l'environnement dans lequel le déploiement est réalisé. Nous nous plaçons dans le contexte du déploiement dans un environnement distribué à l'échelle des grandes "entreprises" possédant un nombre important de stations de travail de natures, de localisations et de caractéristiques variées. Au sein de telles organisations, le déploiement est un processus vital. C'est pourquoi la plupart des

grandes entreprises cherchent à se doter de nouvelles structures prenant en charge le processus de déploiement.

3. Les objectifs de la thèse

L'objectif de cette thèse est de proposer un environnement de déploiement, c'est à dire une plate-forme permettant d'offrir un support automatisé aux activités du cycle de vie du déploiement (sélection, installation, mise à jour, désinstallation, etc..).

L'originalité de l'approche consiste en l'utilisation des outils de déploiement existants. Nous proposons une infrastructure générique dans laquelle il est possible d'intégrer (ou d'utiliser) toutes sortes d'outils liés ou non au déploiement, comme des outils simples de transfert ou d'introspection, ou bien des outils plus complexes comme des installateurs type "InstallShield". Nous avons choisi de proposer cette approche à la suite du constat, selon lequel :

- aucun outil ou environnement de déploiement ne permettait de couvrir tous le cycle de vie du déploiement logiciel,
- par contre il existe de nombreux outils qui réalisent parfaitement des activités partielles de déploiement.

Pour cela nous proposons un environnement permettant l'intégration et la coopération de ces outils, indépendants et hétérogènes en fournissant :

- une infrastructure de communication et de coopération
- une infrastructure d'exécution à distance de ces outils
- une modélisation des différents acteurs et entités manipulés lors du déploiement

Notre approche permet aussi l'automatisation du cycle de vie du déploiement, ainsi que des activités de ce cycle. Pour cela, nous avons développé divers langages de description permettant d'une part de décrire les procédés en terme d'ordonnancement et d'autre part en terme de réalisation (décrire quels outils vont être utilisés, quand et comment). Nous proposons aussi une solution interprétant ces langages et réalisant réellement les activités de déploiement dans le monde réel

4. Organisation de la thèse

Outre ce chapitre d'introduction, le document est divisé en trois grandes parties.

La première partie propose un survol de l'état de l'art et de la pratique. Deux chapitres composent cette partie :

1. Le chapitre 2 introduit d'abord le déploiement à travers un scénario de motivation. A partir de celui-ci, nous détaillons le cycle de vie du déploiement logiciel. Puis nous abordons certaines des problématiques du génie logiciel ayant un lien plus ou moins fort avec celle du déploiement. Ce chapitre se termine par une description de l'architecture à trois niveaux dans laquelle nous avons choisi de situer le déploiement (contexte dû au déploiement à large échelle).

2. Le chapitre 3 propose un survol de l'état de l'art et de la pratique. Le déploiement dans le cadre des modèles à composant [Dot Net, CCM, EJB] est abordé. Nous nous intéressons à deux domaines importants pour la réalisation du déploiement que sont la gestion de configuration et les procédés. Quelques environnements ou outils de déploiement industriels sont aussi évoqués.

La seconde partie du document concerne l'approche proposée dans cette thèse, déjà évoquée précédemment. Elle est composée de trois chapitres :

1. Le chapitre 4 présente l'environnement de déploiement ORYA. Nous décrivons ses trois caractéristiques principales que sont la gestion du cycle de vie, l'abstraction de l'approche et la réutilisation (ou intégration) de l'existant dans l'environnement. L'architecture d'ORYA est aussi détaillée.
2. Le chapitre 5 concerne la première activité du cycle de vie du déploiement : la sélection. Nous présentons le modèle de composant qui a servi de point de départ à notre étude. Le framework d'annotation (des composants) et les règles (ou contraintes) de sélection sont ensuite détaillées. Ce chapitre se termine un exemple d'algorithme de sélection ainsi que par une description du résultat.
3. Le chapitre 6 présente l'autre activité de déploiement étudiée dans le cadre de cette thèse : l'installation. Nous décrivons la problématique de l'installation ainsi que la solution choisie, basée sur l'automatisation via les procédés. Des langages de description et de réalisation ont été définis afin notamment de pouvoir réaliser réellement les activités de déploiement ou de modifier dynamiquement les procédés.

La troisième et dernière partie du document concerne l'implémentation de l'environnement et son expérimentation à travers une collaboration industrielle. Elle est composée de trois chapitres :

1. Le chapitre 7 concerne la description de l'implémentation d'ORYA. Nous décrivons d'abord l'architecture développée. Puis nous présentons l'implémentation des deux activités (sélection et installation) détaillées dans la partie précédente. Nous décrivons notamment le langage de réalisation et la technologie utilisée pour exécuter et piloter (à distance) les procédés d'installation. Pour l'activité de sélection, nous présentons la base de composants développée ainsi que le framework permettant de sélectionner de façon cohérente la meilleure configuration d'une application (celle respectant au mieux les contraintes).
2. Le chapitre 8 décrit les différentes étapes de l'expérimentation d'ORYA réalisée avec la société Actoll. Elle détaille aussi les différents scénarios mis au point pour valider l'approche présentée dans ce document.
3. Enfin, le chapitre 9 propose une synthèse de l'approche ainsi que des perspectives sur l'environnement de déploiement, comme par exemple une industrialisation prévue d'ORYA.

Chapitre II

Le déploiement

1. Introduction au déploiement

Dans ce chapitre nous allons définir ce que nous entendons par la notion de déploiement logiciel, c'est à dire l'ensemble des activités permettant d'installer une application sur une machine et ensuite de la faire évoluer dans le temps.

Pour cela nous commençons tout d'abord par introduire le déploiement à l'aide d'un exemple (ou scénario) de déploiement. A la suite de cet exemple, nous insistons plus sur certains aspects (ou problématiques) liés au déploiement et nous terminons ce chapitre en décrivant l'architecture à 3 niveaux du déploiement qui permettra de réaliser le déploiement dans le contexte du "large échelle" [Hall99, Les00].

2. Scénario de motivation

L'exemple de déploiement se déroule dans une entreprise de taille moyenne. Son parc informatique est constitué de milliers de machines hétérogènes (PC, stations de travail, des Macs, des ordinateurs portables, des serveurs, etc..) réparties géographiquement sur une dizaine de sites. Chaque site étant constitué d'un serveur auquel des centaines de machines sont reliées, le tout formant une branche. Une entreprise est constituée d'un ensemble de branches réunies autour d'un serveur principal (Figure 1).

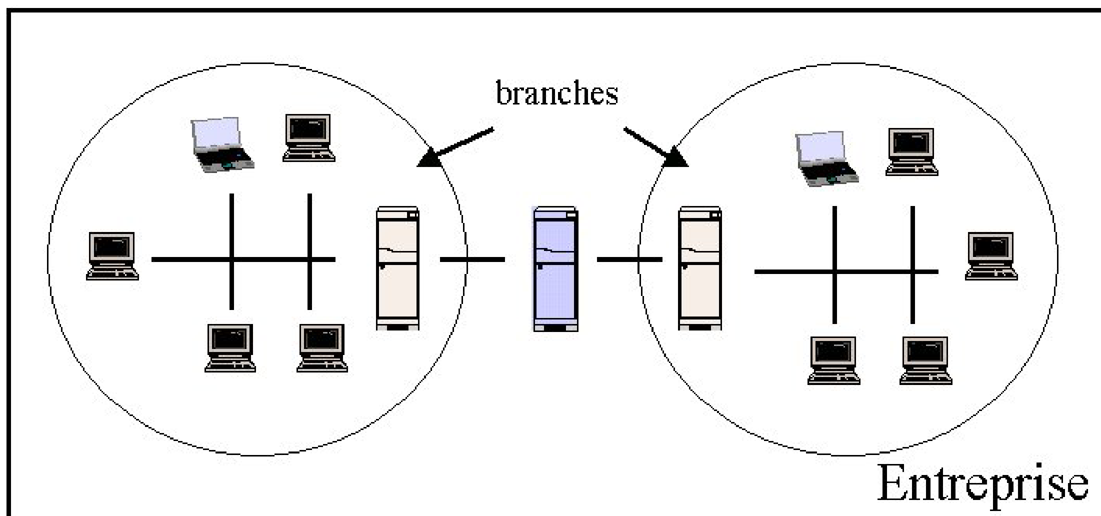

Figure 1 : Exemple de la structuration d'une entreprise

La gestion de l'environnement logiciel de l'entreprise est réalisée de la manière suivante. Le serveur principal se charge de fournir l'ensemble des applications (sous la forme de versions) pouvant être installées sur une machine. Il est interdit à un utilisateur d'installer (ou de modifier) une application en se connectant directement sur le site du producteur ou en utilisant un CD d'installation fourni par le producteur. De cette façon l'entreprise s'assure que les applications qui sont déployées, sont cohérentes (normalement le contenu des "packages" a été vérifié) et que leur déploiement est sécurisé (car il se déroule à l'intérieur de l'entreprise).

Etant donné l'hétérogénéité de son parc informatique, les applications déployées doivent exister en plusieurs versions. Ainsi il est possible d'installer une application sous l'environnement Linux et sous celui de Macintosh. De plus, afin de diminuer le coût des

licences, seules les applications nécessaires (à l'utilisateur de la machine) sont déployées sur chaque machine. Ainsi un chef de projet ne disposera pas des mêmes applications qu'un graphiste par exemple, vu que leurs besoins sont différents.

Afin de diminuer les risques de sécurité inhérents au déploiement, les producteurs de logiciels fournissent à l'entreprise un ensemble de "packages", l'entreprise se chargeant ensuite de déployer ces différents "packages". De cette façon, l'entreprise garde le contrôle total des opérations de déploiement. Ces "packages" servent à l'installation ou à la mise à jour.

L'un des intérêts de centraliser le déploiement au niveau de l'entreprise, c'est de pouvoir le planifier. Ainsi l'entreprise, pour des raisons de coûts a choisi de réaliser les déploiements de façon automatisée et à distance uniquement le soir ou le week-end, sauf dans le cas de mises à jour critiques (pour corriger un bug important ou pour résoudre un trou de sécurité). De cette façon, l'entreprise minimise le coût du déploiement, en terme de ressources (bande passante) ou de baisse de fonctionnalités.

L'automatisation du déploiement (réalisée grâce à la centralisation des connaissances concernant les machines, les utilisateurs et les politiques de déploiement et via le réseau) permet :

- de diminuer les coûts du déploiement (on n'a plus besoin d'avoir une personne par déploiement),
- d'installer les bonnes versions sur chaque machine.

Comme on peut l'apercevoir à l'aide de cet exemple, le déploiement est une tâche importante dans la vie d'une entreprise et l'objectif est d'essayer de réduire son coût et les risques (pour la sécurité et l'intégrité des données) pour l'entreprise.

3. Le cycle de vie du déploiement

3.1. Introduction

Le déploiement logiciel est une activité complexe, qui couvre toutes les étapes depuis la validation du logiciel par le producteur jusqu'à son installation puis sa désinstallation sur les sites utilisateurs. Ces étapes, auxquelles il faut ajouter celles liées à la maintenance de l'application une fois installée (reconfiguration et mise à jour), représentent le cycle de vie du déploiement logiciel (Figure 2) [CFH+98]. Ce cycle permet de décrire toutes les différentes activités, ainsi que leurs interactions, ayant lieu lors du déploiement.

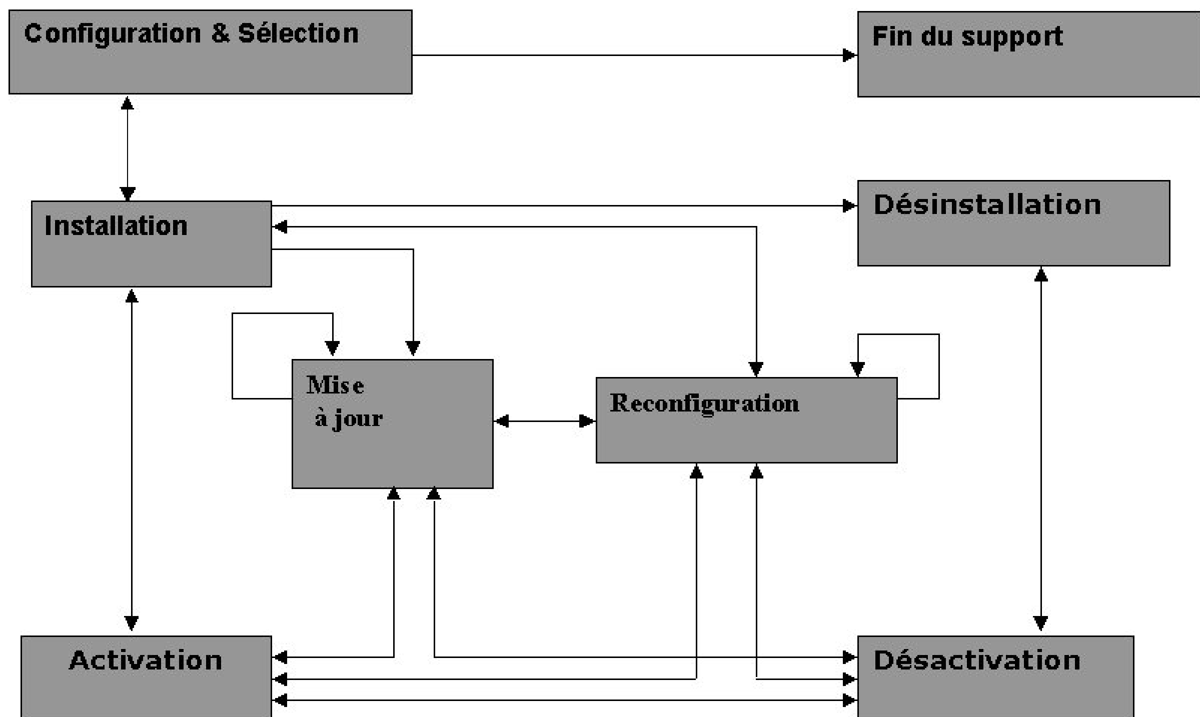

Figure 2 : Cycle de vie du déploiement logiciel

Ce cycle de vie (Figure 2), dont nous allons détailler chacune des activités, on se rend compte que le déploiement est une activité beaucoup plus complexe que le simple fait d'insérer un CD d'installation puis de répondre à quelques questions visant à paramétrer l'application.

3.2. Les différentes activités

3.2.1. Configuration et sélection

La première activité que l'on associe au déploiement concerne la *configuration et la sélection* des applications qui seront ensuite installées sur les machines des entreprises clientes.

L'activité est composée elle-même de sous activités [CE00]. Son objectif est de mettre à disposition des clients du producteur une application (ou du moins l'une des versions de l'application). Pour cela, il faut d'abord configurer l'application, c'est à dire spécifier les fonctionnalités attendues et les contraintes à respecter (comme par exemple, que la version fonctionne sous l'environnement Windows).

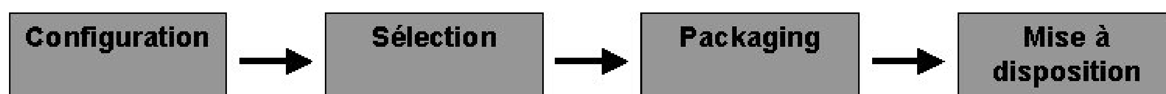

Figure 3 : les sous activités de l'activité de configuration et de sélection

A partir de cette configuration, on sélectionne les "composants" qui réunis forment la version de l'application qui correspond le mieux aux contraintes et souhaits exprimés par la configuration. Ensuite, il faut packager (mettre sous forme de "package") l'application, c'est à dire réunir les différents composants et aussi fournir la description de l'application. Une fois

l'application packagée, elle est mise à disposition des clients, en attente d'être transférée chez les clients pour y être installée.

Ces "packages" servent soit à installer une application ou bien à modifier (dans le cadre d'une *mise à jour*) une application existante.

3.2.2. Installation

L'activité d'*installation*, comme son nom l'indique, a pour but d'installer sur une machine une application packagée. L'activité précédente mettant à disposition des "packages", l'activité d'installation commence par le transfert d'un "package" (celui de l'application à installer), puis il y a éventuellement une activité de configuration et enfin l'installation proprement dite. La figure ci-dessous donne un aperçu de l'activité d'*installation*.

Figure 4 : les sous-activités de l'activité d'installation

L'étape de transfert peut être initiée par le producteur, on parle alors de déploiement en mode "push". Dans ce cas, on impose l'installation d'un "package" particulier, dont on aura ou non vérifié au préalable la compatibilité avec la machine. A l'inverse si c'est l'utilisateur qui demande l'installation, c'est à sa charge de trouver la configuration qu'il souhaite installer, on parle alors de déploiement en mode "pull".

L'étape de configuration lors de l'installation consiste à compléter la configuration en précisant les dernières possibilités offertes avant l'installation comme celle du choix du répertoire d'installation par exemple. On peut aussi faire une configuration plus complète, on parle alors d'installation personnalisée, où l'on peut par exemple sélectionner les fonctionnalités proposées par l'application.

Une fois l'application installée, c'est à dire une fois que l'on a installé les composants de l'application et résolu les éventuelles dépendances ainsi que les problèmes liés aux composants (ou fichiers) partagés, on peut passer à l'activité d'*activation*.

3.2.3. Activation

On regroupe dans l'activité d'activation toutes les opérations nécessaires pour que l'application (que l'on vient d'installer) puisse être exécutée [HAN99]. Il s'agit par exemple de créer des instances d'une base de données, de lancer un Web-Service ou de lancer un service windows. Bref, cela consiste à préparer l'environnement d'exécution de l'application (environnement constitué de la machine ou éventuellement d'un ensemble de machines dans le cas d'applications client/serveur par exemple). A la fin de cette activité, l'application installée est prête à être exécutée.

3.2.4. Mise à jour

L'activité de *mise à jour* consiste à modifier une application déjà installée. Elle est réalisée à l'initiative de l'utilisateur ou du producteur (par exemple, dans le cadre d'un service de mise à jour). A la différence de l'activité suivante, celle de *reconfiguration*, la mise à jour nécessite de retourner chez le producteur pour récupérer les éléments (sous forme de "package" ou de "patch") nécessaires à la modification.

Dans la plupart des cas, la *mise à jour* (comme la *reconfiguration*) nécessite de faire appel à l'activité de *désactivation*, puis de transférer le "package" correspondant à la mise à jour proposée (ou imposée) par le producteur, d'extraire ensuite les données du "package", de les installer et enfin de faire appel à l'activité *d'activation*.

Comme on le voit l'activité de *mise à jour* est quasiment identique à celle de l'*installation* à la différence près, que la plupart des ressources nécessaires à l'application sont déjà présentes sur la machine. On peut considérer que l'activité de *mise à jour* est un cas particulier de celle de l'*installation* [CFH+98].

3.2.5. Reconfiguration

Tout comme l'activité de *mise à jour*, celle de *reconfiguration* a pour objectif de modifier une application déjà installée sur une machine. La différence vient du fait que premièrement l'initiative de la reconfiguration vient de l'utilisateur et non du producteur et que deuxièmement la reconfiguration se fait localement sur la machine sans faire appel à d'autres "packages" du producteur.

L'activité de *reconfiguration* a lieu soit à la suite d'une modification de la configuration matérielle et/ou logicielle de la machine, soit à la demande de l'utilisateur. L'objectif étant d'avoir toujours la version la plus cohérente de l'application déployée. Par exemple, si l'on enlève la carte vidéo de la machine et que l'application risque de ne plus fonctionner correctement, il faut alors la reconfigurer.

La *reconfiguration* ne nécessite pas l'utilisation d'autres "packages" du producteur, par conséquent cela signifie que toutes les données (et les ressources) nécessaires à la modification se trouve dans le "package" ayant servi lors de l'installation. Cela signifie aussi qu'il faut avoir gardé le "package" à la fin de l'installation.

3.2.6. Désactivation

La *désactivation* correspond à l'activité inverse de celle d'*activation*. Elle est utilisée avant les activités de *reconfiguration*, de *mise à jour* et de *désinstallation*. Son rôle est de désactiver les composants de l'application, afin que l'on puisse la modifier ou la supprimer. L'activité de désactivation est utilisée lors des modifications statiques : celles qui ne concernent pas les applications critiques (une application critique ne devant jamais être interrompue même dans le cas de modification). Dans la section 3.3 nous reviendrons plus en détails sur les différences entre l'adaptation dynamique et le déploiement.

3.2.7. Désinstallation

L'activité de *désinstallation* est la dernière (du point de vue temporel) activité du cycle de vie du déploiement logiciel. Elle consiste tout simplement à désinstaller une application existante. Lors de la désinstallation, il faut tenir compte des dépendances et des fichiers (ou composants) partagés.

3.2.8. Fin de support

Cette activité de *fin de support* concerne la fin du service "après vente" fourni par le producteur d'une application. En d'autre termes, il peut s'agir d'un service on-line de dépannage, de la mise à disposition de corrections (pour corriger des bugs par exemple), de la mise à disposition de nouvelles versions, bref tout ce qui concerne l'évolution de l'application.

Cette activité signifie qu'une application ne pourra plus être mise à jour, ni même installée. Par contre elle peut rester sur les machines où elle est déjà installée.

3.3. Déploiement versus adaptation dynamique

Une application est une configuration qui a été développée en respectant un certain nombre de contraintes et ou de préférences. Si l'une de ces contraintes ou préférences change une fois l'application installée, il se peut que l'on doive faire évoluer (c'est à dire modifier) la configuration de l'application. Il existe globalement deux façons de réaliser cette évolution. La première solution consiste à stopper l'application, réaliser les modifications nécessaires et ensuite relancer l'application. On parle alors de modification statique de l'application. La seconde approche consiste à modifier dynamiquement l'application, c'est à dire que les modifications sont réalisées sans interrompre l'exécution de l'application [PBJ97]. On parle dans ce cas d'adaptation dynamique.

Il existe un certain nombre d'applications pour lesquelles l'adaptation dynamique est préférable à l'adaptation statique [KBC02] :

- le cas des applications critiques comme celles qui gèrent les centrales nucléaires et dont l'arrêt est impossible (pour des raisons évidentes),
- le cas des applications dont l'environnement d'exécution change constamment, ce qui rend inadapté l'arrêt de l'application à chaque modification,
- les applications dont l'arrêt est coûteux (en termes de ressources, de fonctionnalités, etc..) pour les entreprises.

L'objectif de ce paragraphe n'est pas d'étudier l'adaptation dynamique mais de voir quels sont ses liens avec le déploiement. Il existe de nombreux travaux sur ce thème [PB97, Pal01]. On peut dire que l'adaptation dynamique se décompose en deux grandes étapes : la préparation et ensuite la réalisation de l'adaptation. L'étape de préparation consiste (lors de la phase de développement ou juste après) à rajouter les informations (ou fonctionnalités) nécessaire pour réaliser l'adaptation dynamique. L'étape suivante consiste "simplement" à lancer l'adaptation dynamique.

Les activités de mise à jour et de reconfiguration sont proches des objectifs réalisés par l'adaptation, c'est à dire la modification de l'application. Pour autant on ne pas dire que

l'adaptation dynamique fasse partie intégrante du déploiement. C'est "simplement" une technologie utilisée lors du cycle de vie du déploiement.

3.4. Conclusion

Comme on vient de le voir avec l'étude du cycle de vie, le déploiement est une activité complexe faisant intervenir de nombreuses activités liées les unes aux autres. Le déploiement fait intervenir (ou plutôt implique) de nombreux domaines de recherches comme la gestion de configuration, l'évolution des applications, l'étude des réseaux, etc..

L'objectif de cette thèse est de proposer une approche permettant de couvrir tout le cycle de vie du déploiement logiciel, mais pour cela il faut pouvoir résoudre (ou du moins tenir compte) les différentes problématiques liées au déploiement [CFH+98]. Nous allons en détailler quelques unes dans la section suivante.

4. Les différentes problématiques liées au déploiement

4.1. La gestion du cycle de vie du déploiement

La complexité du cycle de vie du déploiement logiciel est due notamment au nombre important de relations entre les différentes activités du déploiement. Traiter le déploiement implique d'être capable de couvrir toutes les activités, de gérer les relations entre activités mais aussi d'être capable de coordonner les différents déploiements en cours d'exécution dans l'entreprise. L'utilisation de la technologie des procédés ou des workflows est une façon de pouvoir correctement gérer le cycle de vie. Du point de vue de l'entreprise, cela signifie que les différents déploiements ne doivent pas se gêner entre eux et surtout qu'ils ne doivent pas pénaliser le fonctionnement de l'entreprise (ce qui serait le cas, si par exemple toutes les ressources de l'entreprise se retrouvaient utilisées par le déploiement à la suite d'une mauvaise gestion du cycle de vie).

4.2. La gestion de configuration

Choisir de déployer pour une machine donnée la version la plus cohérente possible (par rapport à ses caractéristiques, aux politiques de déploiement de l'entreprise, aux souhaits de l'utilisateur de la machine, etc..) implique que l'on soit capable lors du déploiement de "construire" la meilleure configuration. Par conséquent, le déploiement implique de résoudre (ou de traiter) les problèmes liés à la gestion de configuration comme la résolution des dépendances, le versionnement, la compatibilité ascendante/descendante, etc..

4.3. L'évolution de l'environnement

Les applications déployées doivent être les plus cohérentes possibles par rapport aux configurations des machines sur lesquelles elles seront installées : cohérence par rapport aux caractéristiques logicielles et matérielles. Par exemple, prenons le cas d'une application existant sous 2 versions : l'une avec une interface graphique et l'autre avec une interface textuelle. Une stratégie de déploiement peut consister à déployer la version graphique sur les machines disposant d'une carte graphique assez puissante et sur les autres la version textuelle. Maintenant imaginons qu'une fois l'application déployée sur une machine, celle-ci change de configuration matérielle (en ajoutant ou supprimant la carte graphique), il est nécessaire de

faire évoluer l'application pour prendre en compte ce changement, que ce soit par l'activité de mise à jour ou celle d'adaptation.

Ceci implique que le déploiement soit capable de modifier une application en exécution (dans le cas d'applications critiques), de traiter le problème des "composants" partagés (éviter de supprimer un "composant" à la suite d'une modification si celui-ci est utilisé par d'autres applications), etc.…

4.4. Les relations entre les applications

Les applications sont de plus liées les unes aux autres, que ce soit en termes de dépendance ou d'incompatibilité. Par exemple, installer une application nécessite de vérifier la présence des entités dont elle dépend et le cas échéant de les installer. Dans le cas des applications à base de composants ou tout simplement dans le cas des librairies dynamiques de Windows (DLL), l'installation d'une nouvelle application implique parfois la gestion de "composants" partagés. Par exemple, si on installe une nouvelle application contenant une DLL et que cette DLL est déjà installée mais sous une version différente, il faudra choisir entre garder l'ancienne (au risque de rendre incohérente la nouvelle application) ou la remplacer par la nouvelle (avec ce que cela implique pour les autres applications utilisant ce composant) ou être capable d'installer et de gérer les deux versions [And00]. Il faut aussi lors de la désinstallation, traiter le cas des applications dépendantes : doit on désinstaller aussi ces applications ou bien doit on les laisser ? Dans le cas des composants partagés, il est nécessaire de connaître l'ensemble des applications les utilisant afin de savoir si l'on peut détruire un composant partagé (uniquement dans le cas où la seule application l'utilisant est celle que l'on désinstalle).

4.5. Les réseaux

L'importance de plus en plus grande des réseaux (notamment d'Internet) a multiplié les possibilités de déploiement. Il est maintenant possible de déployer simplement une application à partir d'une machine du réseau sur une autre machine.

De ce fait, le déploiement doit tenir compte des problèmes inhérents aux réseaux que sont par exemple la gestion de la bande passante ou bien la gestion des transferts de données (garantir la qualité et la sécurité des transferts).

Dans le cas plus spécifique de l'Internet, les relations entre les producteurs et leurs clients ont totalement changé, car maintenant il est relativement aisé en cas de problèmes d'aller sur le site du producteur pour trouver de l'information ou bien pour télécharger des mises à jour. Cette nouvelle dimension doit être prise en compte par le déploiement en résolvant le problème de l'intégration d'Internet dans le déploiement, en cohérence avec la politique de l'entreprise.

4.6. L'hétérogénéité des plate-formes

Le déploiement fait intervenir de nombreux acteurs (producteurs, entreprises et machines). Chacun de ses acteurs peut correspondre à une plate-forme différente et cependant il faut pour réussir le déploiement être capable de traiter ces différents cas. Par exemple, si l'on automatise le déploiement d'une application, il faut que celui-ci puisse se réaliser sur Windows ainsi que sur Linux. On peut résoudre le problème en spécifiant une solution (façon d'automatiser le

déploiement) par plate-forme ou bien essayer de généraliser la solution en tenant compte de la problématique due à l'hétérogénéité des plate-formes.

4.7. La sécurité

Avec l'avènement des technologies des réseaux et notamment de l'Internet, le déploiement est de plus en plus automatisé et évolué mais paradoxalement il est potentiellement de moins en moins sécurisé. En effet, l'automatisation des activités de déploiement comme par exemple le service de mise à jour automatique d'une application implique que de nouveaux "composants" sont installés quasiment tous les jours sur les machines. Bien que ce service soit sécurisé, il n'en reste pas moins que l'entreprise cliente laisse un trou de sécurité via une connexion directe avec le producteur et elle dépend de celui-ci en cas d'attaque ou de malhonnêté.
Un environnement de déploiement permet de déployer, c'est à dire de modifier physiquement n'importe laquelle des machines de l'entreprise. Il faut donc éviter que n'importe qui dans l'entreprise puisse faire "n'importe quoi", le déploiement devant être réservé uniquement à certaines personnes de l'entreprise. En effet, prenons le cas, de données sensibles, elles ne doivent pas (pour des raisons de confidentialité) être à la disposition de tous les employés de l'entreprise et à fortiori de ceux des producteurs. Un système d'authentification et de droits doit impérativement être mis en place.

4.8. Conclusion

Cette liste (non exhaustive) de problématiques sur le déploiement montre que celui-ci est non seulement une problématique en soi mais qu'il en fait intervenir d'autres issues de domaines différents. Ceci complique la problématique du déploiement mais permet aussi de cataloguer les problèmes afin d'appliquer les solutions (ou les approches) existantes (si elles existent).

Nous allons maintenant détailler l'architecture à 3 niveaux sur laquelle nous allons nous baser pour étudier le déploiement à grande échelle.

5. Architecture à 3 niveaux

5.1. Introduction

Le déploiement est une activité complexe comme le montre le cycle de vie du déploiement (Figure 2). Dans le contexte du déploiement à large échelle (déploiement de logiciels de grande taille dans des entreprises) de nombreuses applications sous des configurations diverses vont être déployées. Dans ces conditions, il n'est pas envisageable de rester sur le schéma classique "producteur utilisateur" [LBC01]. Il est pertinent de rajouter entre le niveau producteur (celui où sont développées les applications) et le niveau utilisateur (celui où sont déployées les applications) un autre niveau, celui de l'entreprise (Figure 5).

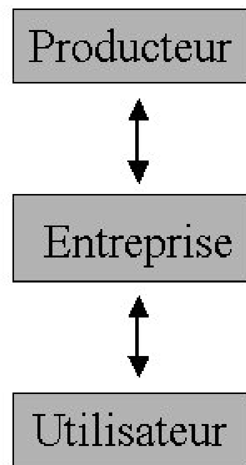

Figure 5 : Architecture à 3 niveaux

Afin de mieux comprendre le rôle de chacun de ses niveaux dans le déploiement, nous allons présenter par niveau les différentes activités liées au déploiement.

5.2. Niveau producteur

Le but des activités de déploiement au niveau producteur est de fournir les applications à déployer et d'assurer ensuite l'évolution de ces applications.

5.2.1. Les applications à déployer :

La plupart du temps, les applications fournies par les producteurs ne sont pas encore exécutables. Elles existent sous la forme de "packages", qui vont servir à l'installation. L'installation peut aller de la simple copie de fichiers sur la machine cible à une opération de configuration directement sur la machine. C'est le cas par exemple, lorsque l'on installe une application à partir d'un exécutable style InstallShield [InstallShield]; il est proposé lors de l'installation de choisir soit une installation standard ou une installation personnalisée (où l'application sera configurée selon les vœux de l'utilisateur final).

Pour réaliser ces configurations, le "package" doit contenir la description de l'application c'est à dire au minimum les informations suivantes (sous la forme de manifestes, de descripteurs, etc.) :

- l'architecture de l'application,
- les contraintes logicielles et matérielles,
- les dépendances (ou incompatibilité) vis à vis d'autres applications,
- la compatibilité entre les versions des "composants".

Cette description est appelée modèle d'application.

5.2.2. Le support des applications

L'autre activité au niveau producteur est la gestion du support aux activités. Ce support concerne toutes les activités qui touchent de près ou de loin au le cycle de vie de l'application une fois installée. Il comporte la notification de la disponibilité d'une nouvelle application (ou

d'une nouvelle version d'une application). Il peut s'agir aussi d'un service automatisé de téléchargement des nouvelles versions (ou des patchs de mise à jour). L'aide en ligne aux clients fait aussi partie de ce que l'on appelle le support.

5.2.3. Le transfert des applications

Le transfert des applications peut se faire de différentes façons :

- à l'initiative du client (utilisateur ou entreprise), on parle alors de transfert en mode pull, le producteur mettant ses produits à disposition de ces clients.
- à l'initiative du producteur. On parle alors de mode push. Dans ce cas, le transfert ne fait pas partie de l'activité d'installation.

5.3. Niveau entreprise

Une fois l'application transférée au niveau entreprise, elle n'est pas tout de suite installée, car l'entreprise joue le rôle d'intermédiaire (avec transformation) entre le producteur et les différentes machines du niveau utilisateur.

Le rôle du niveau entreprise est multiple, il sert à :

- modifier l'application avant de l'installer, c'est le cas par exemple de l'application CATIA [CATIA, Les00] de Dassault Systèmes Les entreprises clientes ajoutent leurs propres extensions avant d'installer l'application sur les machines. Cette activité est appelée activité d'*assemblage*,
- préparer le déploiement, c'est à dire à fixer où, quoi, quand et comment le déploiement de l'application va avoir lieu.

Dans le cadre du déploiement à grande échelle, les entreprises ne déploient pas toutes les applications sur toutes leurs machines. En effet, si l'on reprend notre exemple de CATIA, on peut imaginer que seule une partie des employés des entreprises clientes de Dassault Systèmes l'utiliseront, les autres n'en ayant pas besoin. Le premier rôle du niveau entreprise est donc de spécifier parmi les machines de l'entreprise celles qui seront concernées par le déploiement.

Une fois la liste des machines connues, il va falloir analyser les configurations de ces machines afin (éventuellement) de configurer l'application, c'est à dire de préparer pour chaque machine la configuration qu'elle recevra. Cette étape peut être (et c'est préférable) réalisée dynamiquement lors de chaque installation physique. L'analyse des machines est réalisée à l'aide d'outils d'introspection et/ou en utilisant un modèle de site (regroupant et stockant toutes les informations nécessaires au déploiement concernant chaque machine).

Une fois la réponse aux questions "où" et "quoi" connue, il reste à fixer la date du déploiement et la manière de le réaliser. Ceci implique de connaître l'organisation de l'entreprise et ses politiques en termes de déploiement [HHW99b]. Ces politiques (ou stratégies) sont en fait un ensemble de contraintes (par exemple une contrainte peut spécifier que l' équipe de développement doit recevoir les nouvelles versions un mois avant l'équipe de test), de politiques de déploiement (par exemple, l'entreprise peut imposer un déploiement en mode "big-bang" au cours duquel une application est déployée sur toutes les machines en même temps), etc.

La connaissance de l'organisation est aussi primordiale, car elle permet de tenir compte lors du déploiement de la position des utilisateurs des machines dans l'entreprise, de leurs rôles (par exemple programmeur ou chef de projet). Toutes ces informations peuvent être utilisées pour permettre de mieux répondre aux besoins des employés de l'entreprise.

Ces informations ainsi que les politiques sont contenues dans un modèle appelé *modèle d'entreprise*. A partir de celui-ci, on peut spécifier quand le déploiement aura lieu pour chacune des machines concernées et de quelle manière il aura lieu.

5.4. Niveau utilisateur

Le niveau utilisateur correspond à celui des machines associées à leurs utilisateurs potentiels. C'est à ce niveau que la plupart des activités du cycle de vie du déploiement ont lieu. A la suite de la préparation du déploiement en amont (au niveau entreprise), les applications sous la forme de "packages" sont prêtes à être installées physiquement.

Une fois l'application installée, l'application peut évoluer, être modifiée dynamiquement ou être désinstallée.

Comme on l'a vu le niveau utilisateur est symbolisé par un modèle de site qui permet de fournir aux différentes activités du cycle de vie, les informations nécessaires à la fois sur la configuration matérielle et sur la configuration logicielle (utile en particulier pour les résolutions de dépendances).

5.5. Conclusion

A la différence notable avec les approches traditionnelles du déploiement, qui consistent à ne considérer le déploiement qu'à travers les relations producteurs/clients, les approches faisant intervenir le niveau entreprise permettent d'envisager de traiter le déploiement à grande échelle. Il faut remarquer que le niveau entreprise peut se décomposer en sous niveaux avec pour chacun ses propres politiques (stratégies) de déploiement.

6. Synthèse

Ce chapitre a permis de mettre en évidence et de préciser les concepts liés au déploiement. Déployer une application ne consiste pas seulement à l'installer mais il faut au préalable préparer cette installation et ensuite gérer l'application une fois installée. Le déploiement est une activité complexe faisant intervenir divers domaines de recherche comme la gestion de configuration, les procédés, la modélisation, etc..

Le déploiement dans le cadre d'une entreprise et d'applications de grande taille ne peut pas (ou plus) être réalisé manuellement. Il faut obligatoirement passer par l'automatisation.

Avant de détailler notre approche, c'est à dire un environnement de déploiement, nous allons dans le chapitre suivant étudier les environnements existants de déploiement, certains outils de déploiements, les nouvelles approches à base de composants (qui de plus en plus prennent en compte explicitement le déploiement). Nous étudions aussi différentes technologies ou approches qui nous ont permis d'appréhender, de traiter et d'intégrer les différents aspects et étapes du déploiement.

Chapitre III

Etat de l'art et de la pratique

1. Introduction

Le domaine de recherche du déploiement est très vaste. Il existe de nombreuses technologies (ou domaines de recherche) qui peuvent avoir plus ou moins de liens avec le déploiement. L'objectif de ce chapitre consiste à présenter certains de ces domaines de recherche ou technologies en insistant plus particulièrement sur leurs liens avec le déploiement ou sur leur utilisation dans le contexte du déploiement.

Ce chapitre est structuré de la manière suivante. Tout d'abord nous présentons la gestion de configuration (section 2) et plus particulièrement la sélection et la construction de configurations. Dans le même état d'esprit, nous présentons ensuite le domaine de recherche des procédés (section 3) avec deux grandes parties : les environnements de procédés et les langages de modélisation. Ces deux premières parties (section 2 & 3) n'ont pas de liens direct avec le déploiement mais les concepts et les approches proposés peuvent être réutilisés pour l'activité de sélection du déploiement (proche de la gestion de configuration) et pour l'automatisation des différentes activités du déploiement (les procédés).

Les sections suivantes présentent des approches et des technologies plus dédiées au déploiement. Il s'agit tout d'abord de l'étude de modèles à composants : ceux de CORBA [CCM], de Microsoft [DotNet] et de SUN [EJB]. Dans la section 4 nous présentons ces trois modèles en insistant sur la partie déploiement. L'objectif de cette section est de présenter la façon dont le déploiement est pris en compte et non de présenter les modèles eux-mêmes. La dernière partie (section 5) de cet état de l'art et de la pratique concerne les outils ou environnement de déploiement existants. Il existe dans le commerce de nombreux outils dédiés au déploiement. Nous n'avons pas voulu faire une étude exhaustive, nous avons préféré présenter des outils représentatifs de leur domaine. La section 5 présente tout d'abord un environnement de déploiement issu de la recherche universitaire (SoftwareDock [SoftwareDock]), un autre du monde industriel (Tivoli [Tivoli]), puis l'un des outils de déploiement les plus utilisés (InstallShield [InstallShield]) et enfin l'outil de déploiement de Sun pour les applications côté client (Java Web Start [JWS]).

2. la gestion de configuration

2.1. Introduction

Lorsque l'on construit des configurations, l'un des problèmes majeurs est de savoir (et pouvoir) contrôler les versions, c'est à dire de conserver tous les changements effectués sur une application lors de son développement [Dart91, WMC01, CW97]. Au final, une famille d'applications constituée de plusieurs versions est obtenue.

La problématique de construction de configurations cohérentes d'applications fait partie du domaine de recherche de la gestion de configuration. Il existe un certain nombre d'outils qui permettent de faire du contrôle de version [CW96, CW98]. Nous allons présenter (sans rentrer dans les détails techniques) deux de ces outils : Revision Control System (RCS) [RCS] et Adèle [Est85, EC94]. Il existe d'autres outils du même genre (comme celui de Shape [ML88]) mais notre objectif est de présenter le domaine de recherche afin d'en retirer des enseignements et non de faire une étude exhaustive (et comparative) des outils de contrôle de version.

2.2. Revision Control System

RCS est un système de contrôle de version (ou plutôt de révision) proposé par le GNU [GNU]. Il a été développé principalement par W. Tichy [Tichy85]. Il permet d'extraire des fichiers, de les modifier, de soumettre des modifications et de restaurer n'importe quelle révision d'un fichier.

RCS ne conserve pas une copie entière de chaque révision, il stocke uniquement les *deltas*, c'est à dire les différences entre les révisions successives. RCS est basé sur une base d'objets dans laquelle chaque fichier est représenté sous la forme d'un arbre de révisions. RCS permet d'ajouter des informations sur chaque révision, en particulier des informations sur l'état qui peut être stable, en développement, en test, etc.…

Du point de vue du déploiement, des outils comme celui de RCS proposent une solution simple permettant de "sélectionner" une révision (ou groupe de révisions) parmi les différentes révisions disponibles. RCS propose pour cela un certain nombre de fonctions de sélection. On peut sélectionner les dernières révisions de tous les fichiers de la base (le repository), on peut aussi réaliser la sélection sur l'état des révisions (stable, en test, etc..), ou sur la date de création, etc..

En conclusion, RCS (ou tout autre outil du même type, comme par exemple SSCS "Source Code Control System" [BB95]) n'est pas réellement conçu pour le déploiement, il offre surtout une aide à la gestion de version. La notion de modèle d'application n'apparaît pas explicitement et la notion de *delta* implique que les fichiers soient de type ASCII. Par contre, les concepts et fonctions de sélection sont très intéressants du point de vue de l'activité de sélection du procédé de déploiement.

2.3. Adele

Adele [EC94] est un système de gestion de configuration. Il permet la modélisation des produits (les données versionnées). Il introduit la notion de famille de produits et la construction automatique de configurations [TCG93, TGC95].

Les objets versionnés sont (comme dans RCS) associés à des attributs qui permettent de les décrire. L'une des différences avec RCS c'est que les objets ont une relation (autre que celle qui lie deux révisions entre elles) de dépendance. De cette façon, on peut créer explicitement, pour chaque élément de la base, un lien avec ses dépendances. Ce lien sera ensuite utilisé lors de la construction automatique des configurations. Celles-ci sont exprimées en termes d'attributs qui seront lors de la construction comparés/évalués avec ceux des objets de la base.

Adele propose aussi un système de règles de sélection basées sur un système de valeurs, de contraintes et de préférence. Grâce à ses règles, Adele peut détecter les configurations incomplètes ou incohérentes.

Dans le cadre du déploiement, Adele offre un modèle de produit (non détaillé ici) plus élaboré que celui de RCS (limité au concept de fichier). Cette modélisation permet de décrire les relations et les différences entre les différents éléments d'une application. Ces informations peuvent être utilisées lors du déploiement pour l'activité de sélection et pour celle de mise à jour.

2.4. Conclusion

Les systèmes de gestion de configuration, comme ceux de RCS ou celui d'Adele, ne sont pas des outils de déploiement et ne peuvent pas être utilisés comme tel. Par contre, les concepts de gestion de version font partie intégrante de la phase de déploiement [HHHW97b, HHW95, Hoek00]. En effet, le déploiement consiste à installer puis à gérer des applications; pour cela il faut être capable de construire les applications (et leurs différentes versions) de manière cohérente. Ces outils de gestion de configuration, comme ceux de CVS [Ced03] (qui utilise RCS en ajoutant la partie travail collaboratif), d'Aide de Camp [ADC], d'Adele et autres, sont très utilisés dans la phase de développement des applications. Il faut donc être capable d'utiliser certaines de leurs fonctionnalités, soit pour réaliser directement la phase de configuration ou pour construire nous mêmes les configurations à l'aide des informations (attributs et dépendances) contenues dans les bases de données de ces outils.

3. Les procédés logiciels

3.1. Introduction

La technologie des procédés logiciels a pour but de supporter le procédé de production logiciel en fournissant les moyens de modéliser, d'analyser, d'améliorer, de mesurer et d'automatiser les activités de production [DAW98]. Depuis quelques années, cette technologie a été utilisée comme support pour d'autres activités que la production de logiciels. Il est intéressant de s'intéresser à cette technologie afin d'étudier comment ses concepts et approches peuvent être réutilisés dans le cadre du déploiement. Pour cela, nous nous sommes intéressés à certains aspects de cette technologie : la modélisation de procédé (section 3.2), les différents langages de modélisation (section 3.3) et enfin les environnement de procédés (section 3.4). Nous allons maintenant détailler ces différents éléments.

3.2. La modélisation de procédé

Un procédé peut être vu comme un ensemble d'activités ou d'opérations liées les unes ou autres afin de réaliser un objectif (qui peut être décomposé en sous objectifs). Les procédés logiciels font intervenir un certain nombre d'éléments. La Figure 6 [DAW98, Der94] permet de décrire les différentes interactions ayant lieu entre ces éléments.

Une *activité* est soit une opération atomique ou composite, soit une étape d'un procédé (un sous-procédé). Le but de ces activités est de manipuler (création/modification/suppression) un ensemble de *produits* (parfois appelés artefacts). Une *activité* a besoin d'utiliser des ressources pour être réalisée. On distingue deux types de ressources : les *outils* et les *acteurs*. Ces ressources correspondent à des agents humains (dans le cas des *acteurs*) ou à des agents informatisés (dans le cas des *outils*). Dans le modèle de la Figure 6 les *outils* sont directement reliés aux *activités* alors que les acteurs utilisent un élément intermédiaire nommé *rôle*. Un *rôle* permet d'abstraire les *acteurs* en un ensemble de responsabilités et d'obligations.

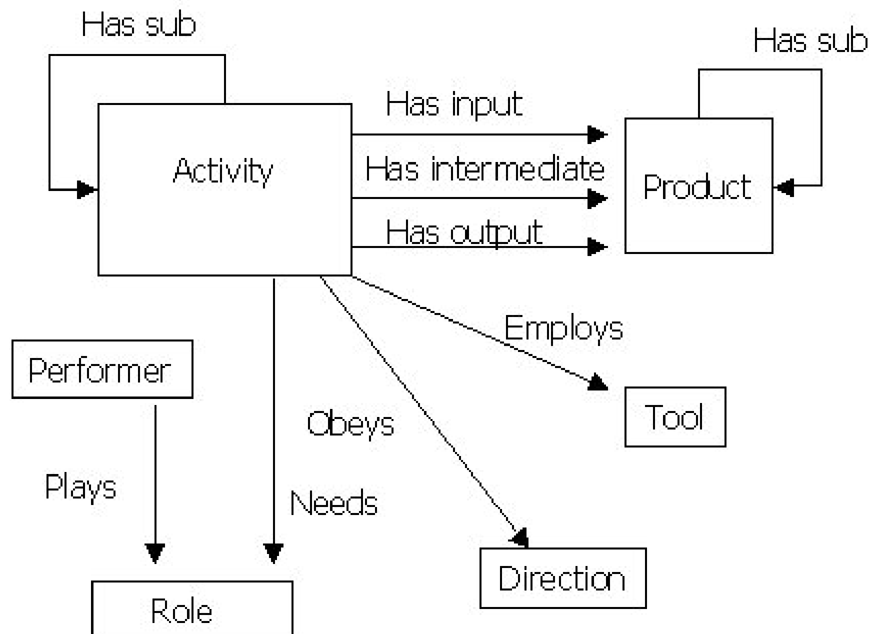

Figure 6 : Les concepts de base de la modélisation de procédés

Chacun de ces éléments peut être aussi l'objet d'une modélisation. Ainsi on peut citer les modèles suivants :

- le modèle d'activité, dont le but est de décrire en particulier les relations entre les différentes activités d'un procédé,
- le modèle de produit qui permet de décrire les types, les structures et les propriétés des entités manipulées par les activités,
- le modèle de ressource, qui permet de décrire les différents acteurs intervenant dans un procédé,
- le modèle de rôle, qui permet de décrire le type d'acteur (humain selon le modèle de la Figure 6 ou non selon d'autres modèles) que l'on souhaite pour réaliser une activité.

Ces différents modèles correspondent à des vues. Selon l'intérêt que l'on porte au procédé logiciel, la vue sera différente. Par exemple, un responsable des ressources humaines utilise la vision proposée par le modèle de rôle, alors qu'un chef de projet utilise la vue issue du modèle d'activité afin de planifier la production du logiciel.

La technologie des procédés logiciels a pour objectif d'intégrer ces modèles et ses vues dans des environnements de travail appelés PSEE [SW94, Garg95] ("Process-sensitive Software Engineering Environnement"). La section 3.4 propose une description rapide des objectifs et fonctionnalités de ces environnements.

Il existe de nombreux modèles associés à la technologie des procédés. De la même façon, il existe de nombreux langages développés dans le cadre de cette technologie. Il existe plusieurs taxonomies de ces langages par rapport aux éléments du procédé (activités, rôles, produits, etc..) ou bien par rapport aux différentes phases du méta-procédé [CFF94]. Ce méta-procédé correspond au procédé qui gère le procédé logiciel (création et évolution). Nous ne détaillons pas ici ce méta-procédé, mais on peut dire que la création d'un logiciel correspond à la succession de plusieurs phases comme celles de l'analyse, de la conception, de l'implémentation, de l'interprétation ou de l'estimation.

3.3. Les langages de modélisation des procédés

Comme on vient de le dire, il existe de nombreux langages utilisés par la technologie des procédés, ces langages sont regroupés sous le terme de langage de modélisation de procédés, les PMLs ("Process Modeling Languages").

Chaque étape (ou phase) du méta-procédé est elle-même un procédé et fait appel à un modèle de procédé. Comme un PML a pour fonction d'être utilisé lors de ces différentes étapes, il doit répondre à un certain nombre d'exigences parfois différentes (voir contradictoires) selon les phases. Par exemple, le langage doit offrir une description suffisamment détaillée du procédé pour que celui-ci soit exécuté ou bien il doit être suffisamment compréhensible par un humain lors de la phase de conception, etc..

Il existe un débat au sein de la communauté procédé sur le fait que l'on doive utiliser un ou plusieurs PMLs, c'est à dire doit-on utiliser un langage par phase ou bien concevoir un langage "universel" utilisable par toutes les phases [CL95]. Il apparaît cependant difficile de concevoir un tel langage sachant que les caractéristiques suivantes (issues des besoins des différents acteurs du procédé) doivent être respectées : plusieurs niveaux de formalisme, d'expression, de compréhension ou d'exécution. Selon la phase dans laquelle on se trouve, le niveau de l'une (ou plusieurs) de ces caractéristiques varie. Par exemple, dans la phase d'exécution le langage doit être opérationnel alors que dans la phase de conception le langage doit plutôt être compréhensible.

Si l'on étudie les langages actuels, on s'aperçoit que chaque langage a été développé pour une (voir deux ou trois) phases du méta-procédé. Par conséquent, plusieurs approches ont été utilisées afin de répondre au mieux aux besoins de la phase en question. Par exemple, le langages Adele et Scale [Oque95] ont été développés selon l'approche dite réactive (basée sur des "triggers") afin de pouvoir réaliser la phase d'exécution, alors que le langage ALF [CBD+94] s'appuie lui sur une approche basée sur les règles. Une taxonomie de ces approches a été faite. Parmi ces approches on peut citer l'approche réactive, l'approche réflexive, celle basée sur les multi-agents, celle basée sur les règles, l'approche procédurale, ou bien celle orientée graphe, etc.. Toutes partagent un point commun : celui de fournir des modèles dont les instances sont exécutables ("enactables").

Nous allons maintenant décrire les environnements dans lesquels les langages de modélisation des procédés sont utilisés.

3.4. Les environnements de procédés

Un environnement de procédés a pour rôle de fournir le support à la technologie des procédés, c'est à dire de fournir les moyens pour définir, modéliser, analyser et exécuter les procédés. La Figure 7 propose une description de l'architecture d'un PSEE.

Un PSEE est contrôlé par un moteur de procédés ("Process Engine") qui a pour rôle de d'interpréter le (ou les) langage(s) de modélisation de procédés utilisés par l'environnement. Autrement dit, le moteur de procédés a pour but de contrôler les flots de données (d'informations) qui transitent durant le procédé entre les différents acteurs (représenté dans l'architecture soit par les machines ou par les utilisateurs de ces machines). Ces échanges d'information sont réalisés en utilisant l'infrastructure de communication intégrée dans le

PSEE. Celui-ci offre aussi une infrastructure (base de données ou autre) permettant de stocker les modèles, les produits utilisés ainsi que leurs instances. Ces données sont manipulées par les acteurs pendant l'exécution des activités dans des espaces de travail ("Workspace"). Ceux-ci peuvent être partagés par plusieurs acteurs, c'est à dire que les produits et modèles d'un espace de travail peuvent être échangés, manipulés par plusieurs acteurs.

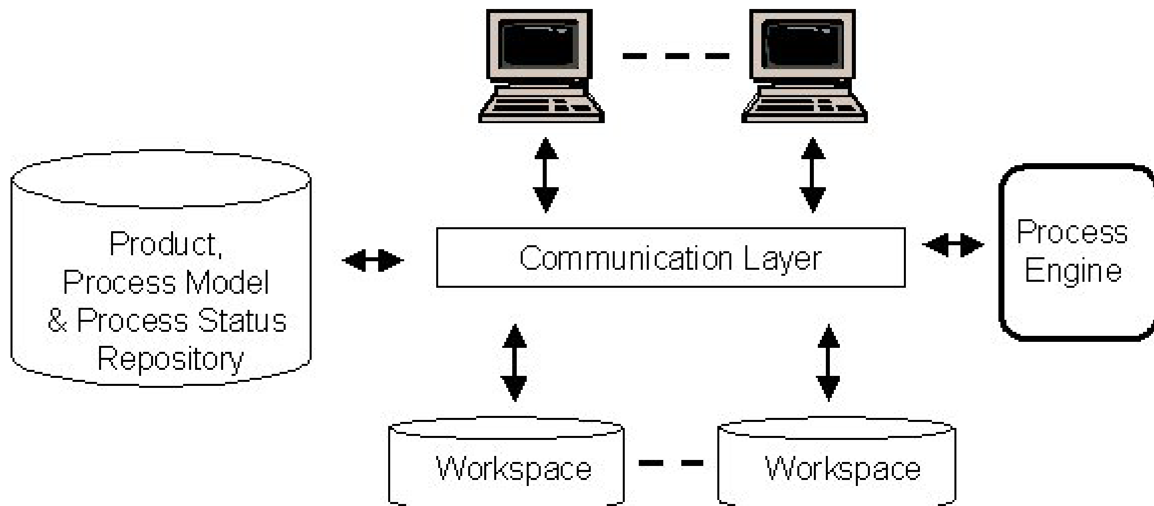

Figure 7 : L'architecture conceptuelle d'un PSEE

3.5. Conclusion

La technologie des procédés logiciels permet de décrire les différentes phases de la production d'un logiciel. Divers langages basés sur des approches différentes ont été développés dans ce cadre et des environnements de support (les PSEEs) ont été construits afin d'offrir un support automatisé à ces différentes phases.

Dans le cadre du déploiement, nous souhaitons appliquer la technologie des procédés et développer un environnement permettant la gestion et l'automatisation du cycle de vie du déploiement. La technologie des procédés propose une solution permettant à la fois de décrire un procédé et ensuite de l'exécuter. Un environnement de déploiement peut être ainsi bâti à partir d'un PSEE.

4. Les modèles à composant

4.1. Introduction

L'un des problèmes du déploiement est que les applications dites "traditionnelles" ne fournissent aucune aide pour le déploiement. Les personnes qui souhaitent installer une application doivent calculer ou extraire des applications les informations comme les dépendances, la répartition géographique (pour les applications réparties), etc..

Avec l'approche de la programmation à base de composants, un effort important a été fait pour faciliter le déploiement des applications à base de composants, l'objectif étant de prévoir la phase de déploiement pendant celle du développement. C'est à dire que les modèles prévoient explicitement la description des composants et de leurs dépendances. Ils permettent aussi dans certains cas de résoudre à la base le problème de versionnement (exemple de l'enfer des DLLs de Windows détaillé dans la section 4.4.1).

Nous allons dans cette section de l'état de l'art décrire le déploiement dans trois modèles à composant :

- "Enterprise Java Beans" (EJB) (section 4.2),
- "Corba Component Model" (CCM) (section 4.3),
- "Dot Net" (section 4.4).

EJB et Dot Net sont parmi les modèles les plus utilisés et CCM est l'un des plus complets du point de vue du déploiement. Dans le cadre de cette thèse, nous n'allons pas décrire entièrement ces modèles, nous allons nous intéresser aux concepts liés au déploiement (comme les descripteurs, les procédés de déploiement, etc..)

4.2. Les EJBs

4.2.1. Introduction

L'architecture des Enterprise JavaBeans (EJB) est une architecture à base de composants (appelé "beans") pour le développement et le déploiement d'applications distribuées. La technologie des EJB [EJB, Mon01] propose un modèle de composant qui a pour objectif de simplifier le développement d'applications de type "middleware" en offrant un support pour les services non-fonctionnels comme les transactions ou la sécurité. Cette technologie a aussi comme but de créer des composants (les beans) indépendants des plate-formes d'exécution. Cela signifie qu'un bean une fois développé doit pouvoir être déployé sur toutes les plate-formes EJB sans nécessiter de recompilation ou de modification de code. La nouvelle architecture (EJB 2.1) a comme objectif de proposer les mêmes services pour les "Web Services" que pour les beans.

Pour résumer, l'architecture des EJBs doit s'occuper des phases de développement, de déploiement et d'exécution du cycle de vie des applications. Nous allons maintenant détailler les différents aspects des EJBs qui concernent le déploiement (cycle de vie, packaging et descripteur de déploiement).

4.2.2. Le cycle de vie du développement et du déploiement

L'architecture des EJBs définit six rôles distincts dans le cycle de vie du développement et du déploiement d'une application EJB. Ce cycle de vie regroupe toutes les activités qui existent à partir du développement des classes Java et qui se terminent lors de l'exécution de l'application. Il s'agit :

- du producteur de beans,
- de l'assembleur d'applications,
- de l'installeur (appelé aussi déployeur),
- du fournisseur de conteneur EJB,
- du fournisseur de serveur EJB
- de l'administrateur système.

La Figure 8 décrit les relations entre ces différents rôles.

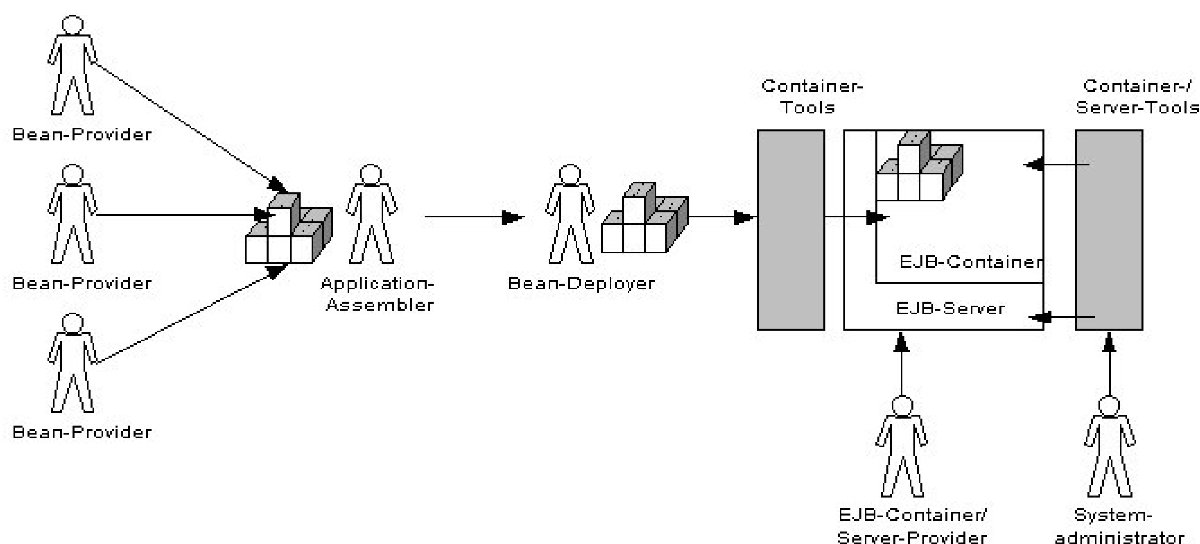

Figure 8 : Les différents rôles du cycle de vie des applications EJB

Nous allons maintenant décrire le rôle de chacun de ses acteurs lors des phases de développement et de déploiement.

4.2.2.1. Le producteur de beans

Le producteur de beans, comme son nom l'indique, a pour rôle de développer les beans, de les décrire et ensuite de les archiver en vue de leur déploiement ou de leur utilisation dans des assemblages (voir le rôle d'assembleur). Une archive (en fait il s'agit d'un fichier de type jar) est constitué d'un bean (ou de plusieurs beans) et d'un descripteur de déploiement. Un bean est formé par la compilation d'un ensemble de classes (et interfaces) java : les classes java réalisant son implémentation, le "home" du bean, celui qui fournit les méthodes de création/suppression des instances du bean et enfin l'interface "remote". Le descripteur de déploiement a pour fonction de décrire les informations structurelles du bean ainsi que ses dépendances logicielles. Le descripteur sera décrit plus en détail dans la section 4.2.4 de ce chapitre. L'ensemble, c'est à dire le bean (les classes compilées) et son descripteur sont ensuite packagés dans une archive.

Une telle archive est directement "installable" et exécutable dans une plate-forme d'exécution d'EJB, mais elle peut aussi être utilisée dans une application de plus grande taille composée d'un ensemble de beans : c'est le rôle de l'assembleur de créer de telles applications.

4.2.2.2. L'assembleur d'applications

Comme on vient de le dire une application peut être constituée d'un ensemble de beans, on parle alors d'un assemblage. Le rôle d'un assembleur (d'applications) est de réunir les différents beans (issus d'une ou plusieurs archives) dans une unité de déploiement. Le fait de réunir et donc de connecter des beans implique que les descripteurs de déploiement peuvent être modifiés et/ou complétés. Ces modifications permettront d'assembler l'application lors de l'installation.

4.2.2.3. L'installeur

L'installeur (ou déployeur) a pour fonction de récupérer les différentes archives constituant l'assemblage, puis de les déployer dans une plate-forme d'exécution d'EJB. Son rôle consiste aussi à résoudre les dépendances spécifiées dans les descripteurs de déploiement. En pratique cela revient à s'assurer que ces dépendances sont présentes sur la plate-forme. Une fois les dépendances "résolues", l'installeur applique les différentes consignes d'installations présentes dans les descripteurs. Il a pour cela, à sa disposition les outils de déploiement fournis par la plate-forme.

4.2.2.4. Le fournisseur de conteneur EJB

Le fournisseur de conteneur EJB a comme rôle de fournir les outils nécessaires pour les activités de déploiement et d'exécution des beans. Le principe étant de placer les beans dans des "enveloppes" afin qu'ils bénéficient de services non-fonctionnels comme la persistance, les transactions ou la gestion des ressources. De cette manière, le producteur de beans a juste à spécifier (de façon descriptive dans le descripteur de déploiement) quel types d'aspects non-fonctionnels il souhaite pour le bean. Le rôle du fournisseur de conteneur étant d'assurer lors de l'exécution que ces aspects seront bien ajoutés au bean.

4.2.2.5. Le fournisseur de serveur EJB

Le rôle du fournisseur de serveur EJB est de fournir une implémentation d'un serveur EJB. Dans l'approche des EJBs, les beans (du côté client) sont indépendants de l'implémentation du serveur grâce aux conteneurs. Les conteneurs et les serveurs implémentant les mécanismes de bas niveau utilisés par les applications : transaction, persistance, etc..

4.2.2.6. L'administrateur système

Le dernier rôle concerne plus particulièrement l'administration système, c'est à dire la gestion des ressources informatiques et des réseaux utilisés par les applications EJB.

On vient de voir que ces différents rôles manipulent et s'échangent des archives et des descripteurs de déploiement. Nous allons maintenant décrire plus précisément chacun de ses éléments.

4.2.3. Les archives

Dans l'approche EJB, l'archive (aussi appelée "package") est l'unité de déploiement pour les applications à base de beans. Une archive peut contenir un ou plusieurs beans. Il s'agit d'un standard, qui sera ensuite utilisé par les différents outils de déploiement (et d'exécution) des différentes implémentations des serveurs d'EJB. Ceci permettant (en partie) d'affirmer qu'une archive (et donc un bean) est exécutable sur n'importe quel serveur EJB. Une archive est composée :

- des classes Java (celles qui implémentent les beans), ainsi que des "homes" et des interfaces pour les Web-services,
- d'un descripteur de déploiement.

Dans certains cas, l'archive ne contient pas directement les classes Java mais uniquement des références vers ces classes. Nous allons maintenant détailler le descripteur de déploiement et le genre d'informations qu'il contient.

4.2.4. Le descripteur de déploiement

Comme on vient de le voir, chaque archive contient un descripteur de déploiement. Une archive est créée par le producteur de bean et peut être modifiée par l'assembleur d'applications, par conséquent le descripteur de déploiement contient des informations à la fois sur le bean lui-même et sur sa place dans une application donnée (uniquement dans le cas où le bean fait partie d'un assemblage). Il y a deux types d'informations dans le descripteur de déploiement :

- les informations structurelles,
- les informations concernant l'assemblage.

Les informations structurelles décrivent la structure d'un bean ainsi que ses dépendances externes. Ces informations sont fournies par le producteur du bean. Parmi ce type d'information, on trouve le nom du bean ainsi que celui de la classe l'implémentant, son type (session, entité ou "message-driven") ainsi que des informations plus spécifiques selon son type (comme le type de persistance dans le cas d'un bean entité ou le type de session (statefull or stateless) dans le cas d'un bean session).

Si un bean est utilisé lors d'un assemblage, cela signifie que l'assembleur va modifier et/ou compléter le descripteur de déploiement. Les modifications concernent plus particulièrement des changements dans le but d'éviter des conflits de noms entre les différents descripteurs (par exemple deux beans peuvent avoir des homes de même nom). Les ajouts dans le descripteur de déploiement permettent de spécifier la façon de lier les différents beans de l'application, de définir les droits d'accès pour les différentes méthodes des beans, etc..

Comme on l'a vu, un assemblage peut avoir en entrée une ou plusieurs archives (et donc un ou plusieurs descripteurs de déploiement). Le résultat peut aussi être sous la forme d'une ou plusieurs archives. Plus précisément une archive (en entrée) peut être décomposée (en sortie) en plusieurs archives et plusieurs archives peuvent être réunies (en sortie) en une seule archive.

4.2.5. Conclusion

Du point de vue du déploiement, la spécification des EJB apporte une plus grande facilité, notamment dans le fait qu'une archive est indépendante d'une implémentation (de serveur d'EJB) donnée, ce qui, en théorie, évite de créer une unité de déploiement d'une application (et à fortiori d'un bean) par type de plate-forme d'exécution. Ce bénéfice est résumé par la phrase suivante : "Write Once, Run Anywhere" [Sun]. Cependant, dans la réalité tout n'est pas aussi simple [CA01], les archives (ainsi que les descripteurs) étant créées par des outils de déploiement spécifique à une implémentation donnée (comme par exemple : BEA [BEA], WebSphere [WebSphere] ou Jboss [Jboss]), elles ne sont pas aussi indépendantes que cela. Le deuxième avantage fournit par les beans pour le déploiement est que certaines informations primordiales comme les dépendances sont explicitement définies lors du développement (rôle du producteur de bean) ou lors de l'assemblage (rôle de l'assembleur).

4.3. Corba Component Model

4.3.1. Introduction

Corba Component Model (CCM) [CCM] correspond à la vision "composant" de l'OMG [OMG] basée sur les concepts répartis de Corba [COR]. Dans la définition initiale de l'architecture Corba, les aspects liés au déploiement et à la distribution des objets n'avaient pas été traités, la spécification traitant principalement de la problématique de l'interopérabilité. Par contre le modèle à composant de Corba [MM01, OMG02] prend en compte ces aspects. Il offre en plus la possibilité d'assembler les composants.

4.3.2. Les composants CORBA

Un composant CORBA est composé d'une interface et d'une ou plusieurs implémentations de cette interface. Un composant CORBA est une unité de déploiement, c'est à dire qu'il s'agit de l'élément de base du déploiement. Cette unité de déploiement est réalisée par un fichier d'archive du composant, dans la pratique une archive de composant est un fichier "zip" contenant la description du composant, les fichiers réalisant la ou les implémentations et un fichier permettant de gérer des propriétés.

4.3.3. Le descripteur de composant CORBA

Il décrit les caractéristiques du composant nécessaires aux phases de conception et de déploiement. Plus précisément, le descripteur (un fichier .ccd pour "Corba Component Description") décrit la version abstraite du composant en terme de ports (facettes, réceptacles, puits d'événements,..). Il spécifie la répartition logique du code (ces informations sont générées automatiquement lors du développement) ainsi que les caractéristiques non-fonctionnelles qu'il utilise.

Lors du déploiement, le descripteur de composant est utilisé premièrement pour déterminer le type de conteneur dans lequel le composant a besoin d'être installé et deuxièmement pour fournir au conteneur l'information sur le composant. Le descripteur fournit des informations sur les services non-fonctionnels qu'il attend du conteneur (transaction, persistance, multi-thread,…).

Chaque implémentation d'un composant CORBA dispose de son descripteur. Nous allons maintenant voir comment est décrit un "package" logiciel, c'est à dire un ensemble d'implémentations d'un même composant

4.3.4. Le descripteur de package logiciel

Il comporte des informations d'ordre général sur le composant, comme par exemple l'identifiant, le numéro de version, celui de la licence ou des informations sur le producteur du composant. En plus de ces informations générales, le descripteur contient des informations plus spécifiques pour chacune des implémentations. Dans ces informations (pour chacune des implémentations) on trouve la liste des dépendances, des informations sur les contraintes de la future plate-forme d'exécution (comme le nom et la version de système d'exploitation). On y trouve aussi le nom du descripteur de composant, qui décrit l'implémentation. Le descripteur est un fichier d'extension .csd pour "Corba Software Description".

Chaque composant CORBA est donc décrit à l'aide de plusieurs descripteurs : celui du composant lui-même et ceux de ces différentes implémentations. CCM permet d'assembler des composants en formant un assemblage. Il faut noter qu'un assemblage ne peut pas contenir lui-même des assemblages, mais uniquement des composants.

4.3.5. Les assemblages CORBA

Un assemblage de composants CORBA est un ensemble de composants interconnectés et logiquement répartis sur un ensemble de machines. Lors du déploiement, l'assemblage sera installé physiquement sur une configuration donnée et les connexions entre les composants seront établies. Pour réaliser et décrire l'assemblage, CCM utilise un descripteur (fichier .cad pour "Component Assembly Descriptor") et introduit la notion de "package" d'assemblage qui contient le descripteur d'assemblage et un ensemble de "packages" de composant (contenant les composants participant à l'assemblage).

4.3.6. Le descripteur d'assemblage

Le descripteur d'assemblage décrit quels sont les composants qui forment l'assemblage, la façon dont ils sont répartis et la façon dont ils sont interconnectés. Ainsi, on peut dire que CCM permet de déployer des composants déjà interconnectés, le travail ayant été réalisé lors de l'assemblage. C'est à dire qu'il sert de guide (voir de patron [GHJK95]) pour l'instanciation et l'exécution des composants.

4.3.7. La phase de déploiement

La phase de déploiement (c'est à dire le déploiement des composants et des assemblages sur les différentes cibles) est réalisé par des outils de déploiement fournis par l'ORB. Ces outils ayant pour rôle de réaliser entre autres : le transfert, l'installation, la composition, l'instanciation puis la configuration des composants sur les cibles. CCM spécifie cependant un certain nombre d'étapes à réaliser lors du procédé de déploiement :

- définition et choix des sites du déploiement
- installation des implémentations à l'aide des informations contenues dans le descripteur de "package" logiciel. L'outil de déploiement doit vérifier qu'il n'y a pas déjà une implémentation installée.
- instanciation des composants
- connexion des composants

4.3.8. Conclusion

CCM en décrivant plusieurs implémentations d'un même composant laisse la possibilité lors du déploiement du choix de l'implémentation à installer, c'est à dire que l'on peut configurer l'application à déployer lors de son installation et selon certaines caractéristiques matérielles et logicielles du site. Cependant, le descripteur de "package" logiciel est basé sur celui d'OSD [OSD] et cela implique que seules les configurations décrites peuvent être installées.

4.4. Dot Net

4.4.1. L'enfer des Dlls

"L'enfer des Dlls" est le problème le plus connu des développeurs d'application Windows [And00]. Une DLL ("Dynamic Link Librairies") est l'un des éléments principaux de Windows. Il s'agit d'une bibliothèque de fonctions, qui seront appelées par un programme exécutable au cours de son exécution.

Une DLL peut être vue comme un composant réutilisable par toutes les applications Windows, si elles le souhaitent. Windows met à disposition des développeurs un certain nombre de ces DLLs pour gérer les périphériques, la communication réseau, etc.. On parle alors de DLLs publiques, c'est à dire disponible pour tous.

Une application Windows comporte donc un certain nombre de DLLs privées (celles qui ont été développées pour l'application) et elle utilise un certain nombre de DLLs publiques. Et c'est là que commence l'enfer des DLLs. La plupart du temps, une application contient non seulement ses DLLs privées mais aussi les DLLs publiques qu'elle utilise. Cela permet au déployeur d'assurer que l'application fonctionnera correctement, alors que si elle utilisait les DLLs déjà installées, le déployeur ne pourrait pas assurer la cohérence de l'ensemble (problématique de la gestion de version).

Le problème vient du fait, qu'en installant l'application, chaque DLL publique est automatiquement installée en écrasant la version existante de la DLL correspondante si elle était déjà installée. Or cette version de la DLL était utilisée par d'autres applications, qui maintenant devront fonctionner avec une nouvelle version de la DLL, qui a des chances de pas être entièrement compatible. En installant une nouvelle application, on a peut être rendu instable l'environnement Windows.

Pour résoudre ce problème Microsoft, ne pouvant pas empêcher les développeurs de fournir à chaque fois avec leurs applications les DLLs publiques, a résolu ce problème dans sa nouvelle plate-forme Dot NET en rendant le développement d'application indépendant de la plate-forme (donc de Windows).

4.4.2. Le déploiement d'applications Dot Net

Microsoft affirme que le déploiement d'applications Dot NET est devenu simple : il suffit de copier les applications dans leur répertoire et c'est tout [Scott00]. Plus besoin de modifier les registres Windows et ensuite de redémarrer l'ordinateur. Pour réussir cela, le framework Dot NET introduit les concepts d'assemblage et de manifeste afin de résoudre les problèmes de versionnement en utilisant la technologie du déploiement dit "side by side".

4.4.2.1. Le concept d'assemblage

L'assemblage est la brique de base dans le framework Dot NET. Il s'agit d'une collection de fonctionnalités. Un assemblage est composé par du code (les DLLs), des ressources et enfin un descripteur appelé manifeste. Ce descripteur (détaillé dans la section suivante) permet de rendre l'assemblage auto-descriptif. Il existe deux types d'assemblage : les assemblages publics et les assemblages privés, les assemblages publics pouvant être utilisés par d'autres

assemblages. Dans cette approche, il faut remarquer que la construction d'une DLL ou d'un exécutable implique la création d'un assemblage. Nous allons maintenant détailler le descripteur associé à chaque assemblage et la façon dont Microsoft a résolu les problèmes de versionnement et ce qu'il a appelé le "side by side".

4.4.2.2. Le concept de manifeste

Un manifeste réalise les fonctions suivantes :

- établir l'identité de l'assemblage, sous la forme de son nom, de sa version et éventuellement de sa signature (dans le cas des assemblages publics),
- définir quels sont les fichiers qui composent l'implémentation de l'assemblage,
- détailler les dépendances envers les autres assemblages,
- spécifier les types et les ressources qui composent l'assemblage,
- spécifier la liste des permissions nécessaires pour que l'assemblage fonctionne correctement.

Ces informations sont utilisées, à l'exécution, pour résoudre les dépendances, pour appliquer les politiques de versionnement et pour valider l'intégrité de l'assemblage (à l'aide des signatures). Le manifeste contient aussi le numéro de version de l'assemblage, nous allons maintenant voir comment Microsoft l'utilise pour gérer les versions dans son framework.

4.4.2.3. La gestion de version

A l'aide des informations sur les versions contenues dans les manifestes, Dot NET garantit l'application des stratégies de gestion de version ainsi que la pérennité des applications même si l'on installe des versions plus récentes et incompatibles de DLL partagées. Pour cela il utilise un numéro de version composé de quatre parties :

<majeur><mineur><construction><révision>

Ce numéro est utilisé par Dot NET lors de l'exécution pour savoir quelle version de l'assemblage doit être chargée. Il existe plusieurs politiques de gestion de version; voici la stratégie standard proposée par Dot NET : une fois l'application installée Dot NET prend automatiquement les versions les plus récentes des assemblages référencés par l'application si ces versions concordent avec les numéros majeur et mineur. La gestion de version sous Dot NET ne concerne que les assemblages partagés.

4.4.2.4. Le "side by side"

On a vu que le principal problème de Microsoft lors du déploiement est du aux DLLs. Une DLL étant considérée comme un assemblage, le problème était encore présent avec les applications Dot Net. Pour le résoudre, Microsoft a introduit le concept de "side by side" qui consiste à rendre possible l'exécution simultanée de plusieurs versions différentes d'un même assemblage (donc d'une même DLL) sur la même machine [Ang02].

4.4.3. Conclusion

Microsoft avec cette nouvelle architecture a essayé (et est parvenue en grande partie) à résoudre ses problèmes de déploiement, symbolisés par la notion d'enfer des DLLs.

Cependant, cela implique de respecter les contraintes suivantes pour toutes les applications Dot NET :

- les assemblages doivent être descriptifs, ainsi il n'y a pas d'impact sur les registres lors des installations. Ceci permet de simplifier les activités d'installation et de désinstallation,
- les assemblages doivent respecter une politique de numérotation des versions,
- le framework doit supporter les composants "side by side" permettant à plusieurs versions d'un composant de s'exécuter sur la même machine,
- l'application doit être isolée le plus possible : un assemblage ne doit être accessible que par une seule application. Les assemblages peuvent être partagés mais ils doivent alors respecter des contraintes plus fortes comme avoir un nom global et unique.

4.5. Bilan du déploiement des composants

En spécifiant explicitement les contraintes à respecter pour le déploiement, on améliore les activités de déploiement (en terme de dépendance, de localisation physique des composants…). Ainsi certains problèmes de déploiement sont pris en compte dès la phase de développement de l'application et non lors de l'installation elle-même. Un autre avantage est l'effort de standardisation des descriptions.

5. Les environnements de déploiement

5.1. Introduction

Un environnement de déploiement a pour but de traiter tout ou partie des différentes activités du cycle de vie du déploiement logiciel. Installer et gérer les applications est devenu l'un des thèmes majeurs pour les entreprises et pour les producteurs de logiciels qui souhaitent pouvoir, pour les uns, gérer le plus simplement possible l'ensemble des logiciels déployés et, pour les autres, faciliter l'installation et l'évolution de leurs produits.

Ce besoin croissant explique le nombre important de solutions (industrielles pour la plupart) dans le domaine du déploiement. Ces outils ou environnements de déploiement sont très variés en terme de couverture du cycle de vie du déploiement ou d'efficacité.

Nous avons étudié un certain nombre de ces outils. Nous avons choisi de présenter un environnement issu de la recherche universitaire "SoftwareDock" (section 5.2) et un autre issu du monde de l'industrie "Tivoli" (section 5.3). Nous présentons ensuite deux outils plus spécialisés que sont "InstallShield" (section 5.4) et "Java Web Start" (section 5.5). Un outil spécialisé se distingue d'un environnement de déploiement dans le fait qu'il ne couvre qu'une partie du déploiement (installation et mise à jour par exemple) et/ou parce qu'il ne prend pas en compte la dimension de l'entreprise.

5.2. SoftwareDock

SoftwareDock (SD) [Hall99, HHW99d] permet d'installer et de gérer des applications. Il est le résultat d'un travail de recherche de l'université du Colorado. SD introduit la notion de famille de logiciels, représentant l'ensemble des configurations (ou versions) d'un logiciel. SD introduit aussi le cycle de vie du déploiement [HHC+98]. Nous allons décrire l'architecture de

cet environnement de déploiement et enfin nous terminerons par l'étude du procédé de déploiement proposé par SoftwareDock.

5.2.1. L'architecture de SoftwareDock

SoftwareDock a une architecture basée sur le concept de "Dock" [HHHW97a]. Un "dock" étant un client, un serveur ou une entreprise. Le nom de "dock" prend tout son sens sur le fait que l'approche de SD est basée sur un système multi-agents. Les agents, chargés de réaliser les différentes tâches du déploiement, s'arrimant aux docks pour remplir leurs fonctions.

5.2.1.1. FieldDock

Il réside sur le site consommateur (celui sur lequel les applications seront déployées). Il s'agit en fait d'un modèle de site dans lequel on retrouve des informations sur le site lui-même (caractéristiques matérielles) et sur les applications déjà déployées. Chaque FieldDock (il y en a un par site) est chargé de gérer (via des agents) les modifications imposées par l'environnement, comme l'installation ou la suppression d'une application.

5.2.1.2. ReleaseDock

Il réside sur le site du producteur. Il permet à l'environnement d'avoir connaissance des différentes applications disponibles pour le déploiement. C'est lui qui est chargé de réaliser les activités d'installation et de mise à jour, via ses agents d'installation. A chaque fois que l'on décide de déployer une application, le ReleaseDock envoie l'un de ses agents pour préparer le déploiement (récupérer les caractéristiques de la cible afin de construire la configuration la plus cohérente). Ensuite, un agent d'installation se chargera d'aller installer (ou de mettre à jour) l'application.

5.2.1.3. InterDock

Il réside dans une entreprise regroupant un ensemble de sites client. Il sert d'intermédiaire entre ces derniers et les producteurs. Il comporte un certain nombre d'informations permettant d'avoir une vue globale de l'entreprise.

5.2.2. Les fichiers de description

Pour réaliser le déploiement, SoftwareDock s'appuie sur un certain nombre de descripteurs [HHW99a, HHW98]. Ceux-ci permettent de modéliser les différentes entités de l'architecture : les producteurs et les clients. SoftwareDock offre la possibilité de configurer les applications afin de ne déployer que les versions les plus cohérentes (celles qui fonctionneront le mieux et qui respecteront le plus les souhaits des utilisateurs). Pour cela, SD propose aussi une modélisation des applications.

5.2.2.1. Le producteur

Le descripteur du producteur décrit la localisation des différentes applications disponibles pour le déploiement. Il s'agit en fait de la localisation des différents descripteurs d'applications.

5.2.2.2. Le client

Le descripteur du client permet de décrire :

- les caractéristiques matérielles telles que l'architecture ou la mémoire
- les caractéristiques logicielles, comme l'OS ou bien les différentes applications déjà installées sur le client.

Les applications une fois installées sont décrites de façon très détaillée. SoftwareDock fait le choix de garder la trace de chaque fichier de chaque application installée. De cette façon, les activités de mise à jour et de cohérence (où l'on vérifie que l'application n'est pas corrompue) sont simplifiées.

5.2.2.3. Les applications

Le descripteur d'application est basé sur une extension de celui d'OSD [OSD] ("Open Software Description"). Une application est décrite en termes de propriétés, de contraintes et d'artefacts. Les propriétés permettent de décrire l'application elle-même (numéro de version, nom de l'application, etc.). Les contraintes correspondent aux exigences (matérielles et logicielles) de l'application afin de pouvoir fonctionner correctement. Il existe deux types de contraintes : celles qui sont résolvables et les autres [HHW99c]. On peut lors du déploiement essayer de résoudre une contrainte résolvable (comme dans le cas d'une dépendance par exemple). Par contre pour les autres, si elles ne sont pas vérifiées le procédé de déploiement est arrêté. Le choix de l'OS est un bon exemple de ce type de contrainte : une application développée pour Windows ne peut pas fonctionner sous Linux, le déploiement est donc inutile dans le cas d'un client fonctionnant sous Linux. Les artefacts représentent les fichiers de l'applications, le descripteur spécifie pour chacun son nom et sa localisation.

5.2.3. Le procédé de déploiement

Le procédé de déploiement mis en œuvre dans SoftwareDock permet de réaliser les activités suivantes : l'installation, la mise à jour, la cohérence, la reconfiguration et la désinstallation. Le tout formant le cycle de vie du déploiement. Nous allons maintenant décrire rapidement les différentes activités de ce procédé.

5.2.3.1. L'installation

L'installation d'une application sur le site d'un client est initiée par le producteur (en mode push). Avant de transférer l'application, on doit configurer l'application à partir des informations issues du descripteur de site et d'un certain nombre de choix de la part du client (dans le cas du mode pull) ou du producteur. Une fois la configuration construite, elle est transférée chez le client et l'installation peut commencer. Les modifications réalisées sur le site (ajout de fichiers) sont prises en compte et le modèle de site est mis à jour. Lors de l'installation, les dépendances doivent elles-aussi être installées (si elles ne le sont pas déjà) sinon le déploiement échoue et le client retrouve son état initial.

5.2.3.2. La mise à jour

Lorsque une application est installée, le site peut souscrire auprès du producteur à un service de mise à jour, c'est à dire qu'il sera notifié à chaque fois qu'une nouvelle version sera

disponible. A l'inverse de l'activité d'installation, la mise à jour peut être déclenchée soit par le site, soit par le producteur. Une mise à jour consiste à modifier l'application installée.

5.2.3.3. La cohérence

Cette activité permet tout au long de la vie de l'application chez le client de s'assurer qu'elle fonctionne correctement. Cela consiste simplement à vérifier la présence de fichiers considérés comme nécessaires à l'exécution de l'application. Si l'un de ces fichiers est absent, SoftwareDock permet de le remettre en place. Cela nécessite de garder sur le site, l'ensemble des artefacts de chaque application

5.2.3.4. La reconfiguration

La reconfiguration consiste à réinstaller une application mais sous une nouvelle configuration. Par contre la version de l'application ne change pas, c'est à dire que la nouvelle configuration est issue du même "package" que l'ancienne. Il n'y a donc pas d'échange avec le producteur, tout se faisant en local chez le client. Le modèle de site est modifié en conséquence.

5.2.3.5. La désinstallation

Cette activité consiste à effacer physiquement l'application et à modifier le modèle de site. SoftwareDock prend en compte la gestion des dépendances, c'est à dire que l'on ne peut pas désinstaller une application qui est nécessaire au bon fonctionnement d'une autre application du client.

5.2.4. Conclusion sur SoftwareDock

SoftwareDock est un environnement de déploiement assez complet mais il impose un certain nombre de contraintes assez fortes. Ses points forts sont l'utilisation d'un modèle d'application extensible permettant de configurer les applications lors du déploiement, la couverture assez complète du cycle de vie du déploiement logiciel. Par contre, SoftwareDock propose un procédé de déploiement figé et non modifiable, ce qui empêche de l'adapter aux différents contextes de déploiement ou bien de réaliser des déploiement multiples. Enfin, il ne permet pas l'installation en mode pull, c'est à dire à l'initiative du client.

5.3. Tivoli

5.3.1. Introduction

Tivoli [tivoli] est un environnement de déploiement (ou d'administration) proposant des solutions (entre autres) pour la gestion de réseaux et d'applications. Parmi l'ensemble des outils, le gestionnaire de configuration ("Tivoli Configuration Manager") propose une solution pour déployer et gérer les applications dans les entreprises [TCM]. Le gestionnaire de configuration est constitué de deux composants principaux :

- l'outil d'inventaire [TI],
- l'outil de distribution logiciel [TSD].

L'outil d'inventaire fournit les informations sur l'environnement (machines, les applications installées, etc..) en assurant la mise à jour de ces données par rapport à ce qui est déployé

réellement. L'outil de distribution permet d'installer, de configurer et de mettre à jour à distance les applications dans l'entreprise.

Nous allons maintenant détailler ces deux outils (ou composants).

5.3.2. L'outil d'inventaire

Lors du déploiement, Tivoli a besoin de connaître pour chaque machine de l'entreprise les informations sur l'inventaire matériel et logiciel. Cela implique de pouvoir collecter ces informations, puis de mettre à jour la connaissance du gestionnaire de configuration de Tivoli. C'est le rôle de l'outil d'inventaire.

La recherche des informations est basée sur l'introspection des machines. Pour cela, Tivoli utilise des profils qui permettent de définir l'information recherchée (type, localisation, etc.). Ces informations sont ensuite décrites en format MIF (Management Information Format) qui est standardisé par le DMTF (Distributed Management Task Force) [DMTF]. L'ensemble des informations est ensuite centralisé dans le gestionnaire de données d'inventaire.

Ce gestionnaire permet aux utilisateurs (les déployeurs) d'exécuter des requêtes afin de préparer le déploiement. En effet, la connaissance de l'environnement de l'entreprise est une aide non négligeable pour le déploiement. Par exemple, si l'on décide de mettre à jour une application, il suffit de lancer une requête et on récupère la liste des machines contenant l'application.

L'outil d'inventaire peut être utilisé sur plusieurs plate-formes. Il reconnaît un certain nombre de propriétés matérielles comme l'architecture, le système d'exploitation.

Dans le cadre du déploiement, l'outil d'inventaire est une aide pour :

- déterminer la base logicielle de l'entreprise (utile pour planifier le déploiement),
- vérifier les contraintes logicielles et matérielles lors du déploiement,
- confirmer une configuration logicielle (utile pour vérifier la cohérence des applications installées).

5.3.3. L'outil de distribution logiciel

Avec l'outil de distribution logiciel, on peut installer, configurer et mettre à jour les logiciels et ceci à distance à partir d'une source unique. Il permet avant le déploiement (lors de la création du "package") :

- de définir les dépendances logicielles,
- de choisir un certain nombre d'actions à exécuter lors du déploiement comme par exemple la vérification de l'application une fois installée,
- de générer automatiquement le "package" en utilisant des techniques de "snapshot" (qui consiste à prendre une "image" d'une machine avant le déploiement, puis de comparer cette image avec une autre prise après le déploiement et d'en déduire l'impact du déploiement sur l'environnement de la machine).

Lors du déploiement, cet outil permet de :

- configurer l'application (modification des registres, création de répertoires, de raccourcis, etc..),
- vérifier les versions des composants dépendants,
- exécuter les actions (ou opérations) spécifiées lors de la création du "package".

Les "packages" sont (en théorie) déployables sur n'importe laquelle des plate-formes supportées par Tivoli. Pour cela on utilise des variables qui seront instanciées lors du déploiement. Ce qui permet de décrire les "packages" indépendamment des plate-formes. L'outil de distribution offre en plus de la construction des "packages", la possibilité de gérer la bande passante, de réaliser des déploiements multi plate-formes, d'interagir avec l'outil d'inventaire pour déterminer les configurations logicielles et matérielles des cibles (pour résoudre les dépendances par exemple). Il offre aussi un service de reprise sur erreur lors de l'installation.

5.3.4. Bilan de Tivoli

Tivoli présente une solution de gestion du déploiement assez complète. L'un des points fort de cette approche est la vision entreprise du déploiement avec la notion d'organisation inhérente à tous ses outils ainsi que l'existence de certaines politiques (comme l'utilisation de l'outil d'inventaire pour planifier le déploiement).

Cependant, cette solution est limitée par plusieurs choix ou contraintes comme le fait que le procédé de déploiement soit fixé (inventaire puis distribution). Tivoli utilise aussi un modèle d'application (basé sur le modèle CIM de la DMTF). Ce modèle ne permet pas de décrire dynamiquement les applications, c'est à dire que toute application doit être au préalable décrite entièrement avant d'être déployée. Enfin, l'opération de déploiement est entièrement centralisée

5.4. InstallShield

5.4.1. Introduction

InstallShield [InstallShield] est l'outil industriel le plus connu de déploiement pour les applications Windows. Cela tient surtout au fait qu'une grande partie des applications que l'on installe sous Windows utilise InstallShield. InstallShield s'appuie fortement sur les services d'installation proposés par Windows.

Nous allons dans cette section présenter tout d'abord le service d'installation de Windows (section 5.4.2), puis les différentes activités du procédé de déploiement mis en place par InstallShield : la création (section 5.4.3), la distribution (section 5.4.4) et enfin la maintenance (section 5.4.5).

Il faut noter que ce paragraphe discute de la version 8 d' "InstallShield Developer" et qu'elle concerne le déploiement d'applications sur les plate-formes Windows 2000/XP/Me. Elle ne concerne pas le déploiement des applications basées sur le modèle à composant Dot Net.

5.4.2. Le service Windows Installer

InstallShield a comme objectif annoncé de faciliter le déploiement d'application Windows. Pour cela il se base sur les fonctionnalités offertes par Microsoft en terme d'installation avec le service "Windows Installer" [WIS].

Ce service :

- offre des moyens automatiques de réparer les fichiers corrompus d'une application, permet d'installer et de désinstaller correctement les applications (prise en charge de la gestion des registres, des raccourcis, etc..),
- offre un service de reprise("roll back") pour retourner à l'état initial en cas de problème lors de l'installation.

Pour cela il faut utiliser les notions de produit et de "package" associés à Windows Installer. Un produit ne contient pas directement ses propres ressources, il s'agit d'un "package" (fichier .msi). Chaque produit a un identifiant global unique connu sous le nom de "Product code", ainsi le service d'installation peut identifier le produit (il garde pour cela la liste de tous les produits) et peut savoir si le produit est déjà installé.

Nous allons maintenant voir comment InstallShield gère le déploiement en se basant sur le service Windows Installer.

5.4.3. La création

La première étape a réaliser pour déployer une application consiste à créer le "package" qui sera ensuite distribué puis exécuté. Afin d'être utilisable par le service d'installation de Windows, InstallShield décrit ses applications sous la forme d'un fichier .msi (voir la section précédente). Ce fichier contient :

- des informations sur le "package" (comme le nom du vendeur ou la date de création),
- des instructions d'installations (la liste des fichiers, des registres qui seront implémentés et sous quelles conditions),
- les références vers les ressources de l'application.

L'activité de création spécifie les fichiers de l'application, les registres et résoud les dépendances. On obtient comme résultat un "package" Windows Installer. Il faut maintenant le distribuer et ensuite l'installer.

5.4.4. La distribution

L'activité de distribution fait référence au transport du "package" du site du producteur vers celui du client chez qui l'application va être installée. Différents moyens de transport sont offerts comme l'utilisation d'Internet ou la création d'un CD d'installation. Une fois le "package" physiquement sur la machine du client, il suffit de l'exécuter et il est pris en charge par le service d'installation de Windows.

5.4.5. La maintenance

Une fois l'application installée, InstallShield propose une activité de mise à jour des applications déployées via son outil de déploiement. Pour cela, il propose la création de "patchs". Cela évite au déployeur d'avoir à recréer tout le "package". Le "patch" faisant référence au "package" qu'il est censé mettre à jour.

L'autre activité liée à la maintenance consiste à assurer la cohérence des applications installées. Elle est réalisée par le service Windows Installer.

5.4.6. Conclusion

InstallShield est un outil d'installation très efficace mais qui a l'inconvénient d'être dédié uniquement aux applications Windows (du fait de son lien étroit avec les services Windows).

5.5. JavaWebStart

5.5.1. Principe

La technologie Java Web Start (JWS) [JWS] permet de déployer des applications côté-client. Basée sur la spécification JNLP ("Java Network Launch Protocol") [JNLP], JWS offre un mécanisme d'installation et de mise à jour pour ces applications [Zuk02]. L'application est hébergée sur un serveur. Le client (qui désire installer l'application) se connecte au serveur, télécharge l'application et ensuite l'exécute. Pour l'instant il n'y a rien de nouveau mais l'intérêt de Java Web Start est :

- de permettre de n'exécuter que les versions les plus récentes des applications et ceci de façon transparente au client,
- d'isoler les applications totalement de l'environnement du client, ce qui augmente le niveau de sécurité lorsque l'on installe une application téléchargée sur Internet.

Nous allons maintenant voir comment on crée des applications déployables via Java Web Start et quel est le procédé de déploiement utilisé.

5.5.2. Le "packaging"

L'application à déployer est sous la forme d'une archive jar. Afin de rendre l'archive déployable, on y ajoute un descripteur de déploiement, qui spécifie comment démarrer l'application. Ce descripteur (dont la spécification est décrite dans la spécification de JNLP) est un fichier XML basé sur une DTD [Xml01]. Il permet de spécifier les informations suivantes : la localisation des ressources de l'application (le répertoire où se trouve l'archive de l'application), des informations apparaissant lors du déploiement (comme le nom de l'application, celui de son auteur ou bien une description rapide de l'application par exemple) et le nom de l'archive de l'application (cette information est associée à celle de la localisation).

Ce descripteur (un fichier .jnlp) est ensuite inséré dans l'archive Jar de l'application. Il faut noter que pour être déployée l'archive doit avoir avoir une signature (constituée d'une clé et d'un mot de passe).

5.5.3. Le procédé de déploiement

Java Web Start impose à toutes les applications de respecter un procédé figé de déploiement. Ce procédé a comme objectif d'installer et de gérer l'application une fois installée.

5.5.3.1. L'installation

JWS propose plusieurs façons d'installer une application : soit en utilisant JWS ou bien en cliquant simplement sur un lien Web de téléchargement. Le seul pré-requis pour utiliser cette technologie est d'avoir au préalable téléchargé et installé Java Web Start. Si l'on utilise un lien Internet, celui-ci doit contenir les instructions permettant au navigateur Web du client d'invoquer JWS. Une fois invoqué (directement ou non), JWS interroge le client pour déterminer si toutes les ressources nécessaires pour exécuter l'application sont déjà présentes. Dans ce cas on passe à l'étape suivante du procédé de déploiement : la gestion de l'application. Dans l'autre cas, c'est à dire dans le cas où l'application n'est pas déjà installée, Java Web Start télécharge les ressources nécessaires. De cette façon seules les versions les plus récentes sont installées. Cela suppose aussi qu'il n'y a pas de configuration possible des applications.

5.5.3.2. La gestion de l'application

Si l'application que l'on souhaite installer est déjà présente chez le client Java Web Start va vérifier que la version installée est la plus récente. Si oui, alors l'application est exécutée, sinon la mise à jour est réalisée automatiquement. Une fois l'application installée, Java Web Start avant chaque exécution vérifie qu'il s'agit de la dernière version disponible afin de réaliser la mise à jour le cas échéant.

5.5.4. Conclusion sur Java Web Start

L'approche préconisée par Sun pour les applications côté client, a l'avantage de permettre (de façon transparente) l'exécution des versions les plus récentes des applications. Ces applications sont déployées chez le client de façon très simple (grâce à la technologie Internet), la seule contrainte étant d'installer chez chaque client Java Web Start. De plus, les applications étant isolées des autres applications du client, le risque (inhérents aux applications disponibles via le Web) en est d'autant diminué.

Par contre, cette approche ne permet que le déploiement d'applications java et implique que chaque configuration soit au préalable préparée et packagée.

5.6. Bilan sur l'état de la pratique

Comme on vient de le voir, il existe de nombreux outils dédiés au déploiement. Ces outils proposent des solutions variées à un certain nombre de problèmes de déploiement. Dans certains cas, ils sont très efficaces, mais ils ne couvrent pas tout le procédé de déploiement. C'est le cas de l'outil "InstallShield". D'autres comme "Java Web Start" propose une solution globale mais qui ne s'applique que dans des cas particuliers (applications java côté client par exemple). Les environnements de déploiement ont vocation à couvrir et surtout automatiser le procédé de déploiement. Mais de la même manière que pour les outils de déploiement, les solutions proposées sont souvent ad-hoc, c'est à dire des solutions de déploiement dans un

environnement bien particulier qu'il est difficile (voir impossible) de réutiliser dans un cadre différent.

6. Conclusion

Nous avons essayé dans ce chapitre sur l'état de l'art de présenter un panel d'outils (et d'environnements) de déploiement représentatif des approches existantes en terme de déploiement. Le second objectif était de présenter le déploiement dans le cadre des modèles à composant. En effet, il apparaît assez clairement que l'approche basée sur la composition et la réutilisation est et sera de plus en plus utilisée. Il est donc nécessaire pour tous les environnements de déploiement de tirer parti des avantages en terme de déploiement qu'offrent ces modèles. Le dernier objectif de cet état de l'art avait pour but de présenter des domaines de recherches différents de celui du déploiement mais dont les concepts et objectifs coïncident parfois avec ceux du déploiement.

En conclusion de ce chapitre, nous pouvons dire que certaines des problématiques du déploiement sont de plus en plus traitées soit par des outils de déploiement soit par les applications elles-même. Cependant aucune approche ou outil ne permet de résoudre (ou même de traiter) l'ensemble des problématiques (et problèmes) liées au déploiement. Nous avons cependant choisi dans notre approche (visant à créer un environnement de déploiement) de nous baser sur la réutilisation de ces solutions. Ceci est l'objet des chapitres suivants qui présentent notre approche.

Chapitre IV

Notre approche : ORYA

1. Motivations

De nos jours, on installe de plus en plus de logiciels sur nos machines. Et de plus en plus souvent on a la possibilité de configurer ces logiciels soit de façon manuelle (cela concerne les installations dites personnalisées) ou de façon automatisée (l'outil d'installation configurant de lui-même le logiciel à installer). De moins en moins souvent, on installe une version identique à celle de tout le monde.

Si l'on reporte ceci au niveau des entreprises, on peut légitiment penser, qu'elles souhaiteraient avoir le même service pour leurs parcs de machines, c'est à dire déployer sur chaque machine la version la plus cohérente possible : cohérence avec les contraintes imposées par l'entreprise (que nous nommerons les politiques de déploiement), les contraintes imposées par les machines et enfin les souhaits des utilisateurs des machines.

Un environnement de déploiement doit donc être capable de fournir ce genre de services, c'est à dire des déploiement personnalisés à chaque fois. Cependant, dans le contexte d'une entreprise la personnalisation des logiciels ne peut pas se faire manuellement, car cela signifierait qu'il faudrait "placer" une personne derrière chaque déploiement, ce qui est irréalisable à moins que chaque utilisateur gère son propre déploiement. Ce qui est l'inverse de l'objectif recherché.

Nous avons décrit dans le chapitre 2, le cycle de vie du déploiement d'un logiciel. Ce cycle de vie est complexe notamment à cause des relations multiples entre les activités, ce qui rend difficile de traiter ces activités indépendamment les unes des autres. Notre environnement doit donc offrir toute l'infrastructure permettant de supporter et de réaliser toutes les activités de ce cycle de vie (installation, mise à jour, etc.).

Si l'on analyse sommairement le déploiement à un instant donné d'une application, on remarque tout de suite qu'il est unique, c'est à dire qu'il est différent du déploiement des autres applications et même des autres déploiements de la même application. Par conséquent si l'on veut être capable de déployer tout type d'application dans n'importe quelle circonstance, il faut automatiser le déploiement et aussi abstraire l'environnement de déploiement afin de proposer des solutions les plus génériques possibles (ce qui diminue d'autant le nombre de déploiements à décrire).

Dans le chapitre sur l'état de l'art, nous avons conclu sur le fait qu'il existait de nombreux outils dédiés au déploiement, mais qu'aucun (ou alors dans des conditions ad-hoc) ne couvrait tout le cycle de vie du déploiement. Nous avons fait le choix de réutiliser ces outils afin de construire notre environnement. Nous avons pris garde de pas devenir dépendant d'un ensemble d'outils et pour cela nous avons basé notre approche sur un environnement ouvert et évolutif (prise en compte de nouveaux outils, mise à jour des outils, etc.).

Ce chapitre est structuré de la façon suivante : dans la section 2 nous décrivons la gestion du cycle de vie par notre environnement, la section 3 décriera l'abstraction réalisée sur l'environnement (ainsi que sur les solutions de déploiement). La section 4 justifie et explicite le choix de réutiliser les outils de déploiement existants et dans la section suivante nous présentons ORYA (pour « Open enviRonment to deploY Applications") notre environnement. Et enfin nous terminons par un récapitulatif des caractéristiques d'ORYA [ORYA].

2. La gestion du cycle de vie

Si l'on reprend la description du cycle de vie du déploiement d'un logiciel (figure 2 du chapitre 2), on remarque le nombre important de relations entre les différentes activités. Cela montre bien la forte dépendance d'une activité envers les autres. Par exemple, l'activité de mise à jour dépend de celle d'installation, de sélection et d'adaptation. Il est donc important (voir primordial) que notre environnement puisse gérer le cycle de vie, c'est à dire qu'il :

- couvre tout le spectre des activités du déploiement
- gère les relations entre ses activités
- assure aussi la coordination entre les différents procédés de déploiement en exécution

Assurer la coordination prend tout son sens lorsque l'on doit réaliser plusieurs déploiement en même temps, ce qui est le but d'un environnement de déploiement en entreprise. La gestion du cycle de vie concerne la gestion des activités elles-mêmes (installation, mise à jour, etc.). C'est à dire qu'ORYA doit aussi :

- automatiser les activités du déploiement en assurant leur enchaînement et leur bon déroulement.
- permettre la modification des procédés de déploiement.

Pour ce faire ORYA s'appuie notamment sur la technologie des procédés et sur celle des fédérations qui ont été développées au sein de l'équipe Adèle. Nous allons maintenant détailler un peu plus chacune de ses caractéristiques.

2.1. La couverture du cycle de vie

Les outils existants de déploiement ont pour la plupart un niveau de couverture partiel, c'est à dire qu'ils ne peuvent réaliser qu'une ou deux des activités de déploiement. Par contre ORYA cherche à couvrir tout le cycle de vie, car de plus en plus le déploiement des applications devient complexe. En effet, les nombreuses relations entre les différentes activités du déploiement montre bien qu'il est de plus en plus difficile d'isoler une activité par rapport aux autres. C'est le cas notamment le cas de l'activité de mise à jour qui nécessite une connaissance précise du résultat de l'activité d'installation initiale et des activités successives de mise à jour ayant déjà eu lieu.

Couvrir tout le cycle de vie impose de traiter toutes les activités et surtout de traiter les relations entre les activités.

2.2. L'automatisation du déploiement

Automatiser le déploiement signifie que l'on doit être capable de décrire le scénario de déploiement et ensuite de l'exécuter (à distance) et sans intervention humaine (sauf demande explicite de la part de l'administrateur de l'entreprise par exemple pour des raisons de sécurité).

Il est peu crédible de proposer une approche dans laquelle on doit décrire en amont tous les scénarios de déploiement avant de les exécuter. L'automatisation a pour but de diminuer l'étape de description et de spécification, par contre le nombre de scénarios réellement

exécutables ne va pas diminuer (sinon cela signifie que l'on installe toujours un logiciel de la même façon). Notre environnement doit donc offrir des solutions de déploiement (sorte de scénarios génériques) qui seront ensuite "instanciées" pour prendre en compte le moment et les caractéristiques du déploiement. Ce qui nous amène à la caractéristique suivante d'ORYA : la modification de procédés (ou plutôt de scénarios).

2.3. La modification de procédés

Comme on vient de le voir, ORYA permet la description des scénarios de déploiement sous forme de procédés. Ces procédés peuvent être modifiés lors de leur instanciation; ceci afin de prendre en compte par exemple les politiques de déploiement fixées par l'entreprise.

Dans le chapitre 6, nous présenterons de façon détaillée la manière que nous avons choisi pour réaliser ces modifications. Nous pouvons cependant dire qu'ORYA est capable de prendre en compte les modifications "statiques" (juste avant le début de l'exécution du procédé) et aussi les modifications "dynamiques" en cours d'exécution (ceci afin de pouvoir par exemple intervenir en cas de problèmes).

2.4. La coordination

Nous avons vu qu'ORYA pouvait traiter l'ensemble des activités du cycle de vie du déploiement. Ce cycle de vie ne concerne que le déploiement d'une application. Dans le cadre de déploiement dans des entreprises, on peut dire que de façon constante il se déroule plusieurs déploiements simultanément. Ceci n'est pas sans conséquence sur chaque déploiement. Par exemple, la gestion de la bande passante peut être un élément de suspension d'un déploiement (par exemple dans le cas où le débit est trop faible).

ORYA doit donc avoir une vision globale de tous ces déploiements afin de pouvoir les coordonner. Cette vision globale a aussi l'avantage qu'elle permet à ORYA de fournir des solutions pour les déploiements multiples : déploiement de la même application sur plusieurs machines en même temps ou le déploiement de plusieurs applications sur des machines différentes. ORYA offre un support pour la coordination des procédés de déploiement.

2.5. Conclusion

ORYA est un environnement de déploiement permettant de réaliser de façon automatisée et coordonnée les différentes activités du cycle de vie du déploiement. Nous allons maintenant voir comment ORYA permet de décrire des scénarios génériques de déploiement.

3. Une approche basée sur l'abstraction

"Une activité de déploiement peut être vue comme une procédure pour contrôler l'exécution et l'allocation de ressources" [CFH+98]. Les ressources dans cette définition correspondent aux différentes entités manipulées lors du déploiement; on y trouve les "packages", les applications, les outils de déploiement, etc.

Afin d'aider à la compréhension, nous utiliserons le terme de scénario de déploiement pour représenter une activité donnée du cycle de vie du déploiement pour une application donnée.

La mise à jour de l'application Word sur une machine lamda correspond à un scénario. La même mise à jour sur une autre machine correspond à un autre scénario.

Comme nous l'avons dit précédemment, nous ne pouvons pas décrire en amont tous les scénarios possibles. Nous avons donc choisi de baser notre approche sur l'abstraction.

3.1. Le principe

Tout d'abord nous avons modélisé (abstrait) les entités de notre environnement [Les02, LBC01]. Ceci pour plusieurs raisons. La première est, que si l'on veut simplifier la description des scénarios de déploiement, l'abstraction de l'environnement permet d'être indépendant des différences entre entités de même type. De même, une application à base de composants est différente d'une application plus classique, par contre les différences peuvent ne pas influencer le déploiement. Par conséquent abstraire les applications évite de devoir faire un scénario par application ou même type d'application.

Un autre avantage concerne plus particulièrement le cas du déploiement à large échelle. En caricaturant ce type de déploiement, on peut dire qu'il a pour objectif de déployer des applications sur des sites (les machines) en respectant des politiques de déploiement. Par conséquent si l'on veut calculer le nombre de scénarios de déploiement possibles, cela reviendrait (de façon simpliste) à multiplier le nombre d'applications par le nombre de machines puis par le nombre de politiques. On arrive vite à un nombre tellement important, qu'il est impensable de pouvoir décrire tous les scénarios potentiels. Pourtant, notre environnement doit être capable à tout moment de déployer une de ces applications sur une machine donnée en suivant une (ou plusieurs) politiques de déploiement.

Abstraire l'environnement va donc permettre de décrire les différents scénarios indépendamment du type de site, d'application et de politique. L'intérêt de l'abstraction apparaît tout de suite évident dans ce cas, car ainsi nous sommes capables de diminuer de façon significative le nombre de scénarios de déploiement à décrire.

Un scénario dépend de l'application à déployer, de la machine sur laquelle elle va être déployée, de l'entreprise, de l'état du réseau, bref de l'état des entités de l'environnement de déploiement. Comme nous l'expliquerons plus en détail dans le chapitre 6 de cette thèse, l'abstraction de l'environnement permet de réutiliser un même scénario dans des cas différents.

De cette façon, nous pouvons décrire un certain nombre de scénarios pour réaliser les activités du cycle de vie et ensuite les "instancier" pour prendre en compte les différentes contraintes spécifiques à l'application et à son contexte de déploiement.

3.2. Conclusion

En utilisant une approche basée sur l'abstraction, autrement dit en décrivant le déploiement au niveau modèle, nous sommes capables de décrire la plupart des scénarios qui seront ensuite "instanciés" et modifiés pour prendre en compte le contexte de déploiement. Nous allons maintenant décrire comment ORYA permet la réutilisation d'outils existants de déploiement pour exécuter les différents scénarios.

4. La réutilisation

4.1. La problématique

Si l'on regarde plus précisément l'exécution des différentes activités du cycle de vie du déploiement, on s'aperçoit qu'elles correspondent à l'exécution d'un ou plusieurs outils de déploiement. Dans le chapitre sur l'état de l'art, nous avons vu qu'il existait de nombreux outils dédiés au déploiement. Il est inutile (et coûteux) de refaire (et la plupart du temps moins bien) nous-mêmes ces outils. Nous avons choisi de les réutiliser, c'est à dire de baser notre environnement de déploiement sur l'utilisation d'outils existants.

Malheureusement, la majorité de ces outils ne couvre pas entièrement le cycle de vie et dans le cas contraire, les solutions proposées sont ad-hoc, c'est à dire que le déploiement doit respecter des contraintes fortes sur les applications et sur les stratégies mises en place.

Nous avons basé notre approche, sur la solution suivante : exprimer les procédés et les scénarios de déploiement à l'aide de ces outils. Par conséquent, la problématique du déploiement va (en partie) se ramener aux questions suivantes :

- Comment faire interopérer des outils hétérogènes ne se connaissant pas et ceci sans les modifier ?
- Comment rendre les scénarios de déploiement indépendant des outils et comment choisir un outil parmi plusieurs ?
- Comment contrôler le comportement de ces outils ?
- Comment exécuter à distance ces outils ?
- Comment partager un ensemble de ressources entre ces outils ?

Nous allons maintenant aborder chacune de ces questions (ou problématiques).

4.2. Le problème de l'interopérabilité

Réutiliser et faire coopérer (fonctionner) des outils existants entraîne des problèmes d'interopérabilité [Pol01]. Ce terme définit la notion de coopération entre applications. Dans le cas d'outils homogènes, il est relativement aisé de les faire coopérer. Mais, dans notre cas les outils sont hétérogènes, c'est à dire qu'ils ont été construits indépendamment les uns des autres et ont donc peu de chance de pouvoir fonctionner ensemble.

Il existe de nombreuses raisons pour que des applications ne puissent pas fonctionner ensemble. Par exemple si :

- elles ont été conçues pour des plate-formes différentes,
- elles sont dans l'incapacité de communiquer entre elles (protocoles réseaux différents),
- elles utilisent des bases de données hétérogènes, etc.…

Ils existent plusieurs façon de traiter le problème d'interopérabilité. Une solution consiste tout simplement à modifier les outils afin qu'ils puissent fonctionner ensemble. Malheureusement, les outils qui nous concernent sont des outils propriétaires et par conséquent non modifiables.

Résoudre les problèmes d'interopérabilité, cela signifie résoudre pour chaque outil la façon de coopérer avec les autres outils. Mais cette solution est difficile à mettre en place du fait du nombre et de la complexité des outils en question.

Une autre solution, utilisée entre autres par CORBA et les architectures à base de composants (CCM, .NET, EJB…), consiste à décrire chaque outil à l'aide d'un langage commun à tous. Par exemple, CORBA utilise le langage IDL [COR] pour décrire l'interface des différents outils, ce qui permet ensuite de décrire les différentes interactions au niveau de ces interfaces.

ORYA propose une solution basée sur l'utilisation d'un langage commun pour décrire les différents outils et sur l'utilisation d'un langage de procédés pour décrire les différentes interactions entre les outils. Pour cela ORYA s'est appuyée sur la technologie des fédérations (cf section 5.3). Dans le chapitre 6, nous décrivons plus en détail l'utilisation des procédés dans ORYA pour résoudre (notamment) les problèmes d'interopérabilité.

4.3. Le problème de dépendance vis à vis des outils

Le choix d'utiliser des outils existants pour réaliser nos activités de déploiement pose le problème de la dépendance de notre approche vis à vis de ces outils. En effet, que se passe-t-il lorsque l'un de ces outils ne peut pas être utilisé sur une machine par exemple ? Ou bien que se passe-t-il pour les scénarios de déploiement en cas d'évolution des outils ?

Pour éviter ce genre de problèmes ou plutôt pour essayer de diminuer le risque, nous avons choisi d'abstraire les outils [Mer02], c'est à dire de décrire chaque outil sous la forme d'une entité nommée "rôle". Un rôle permet d'abstraire un outil en fournissant l'interface fonctionnelle de celui-ci. Un rôle s'exprime en termes de services que chaque outil "jouant" ce rôle s'engage à fournir. Un rôle peut être joué par plusieurs outils et un outil peut jouer plusieurs rôles.

L'intérêt de cette approche est double :

- Elle permet de disposer éventuellement du choix de l'outil pour jouer un rôle. De cette façon notre environnement pourra choisir l'outil disponible le plus cohérent, le plus efficace, le moins gourmand en ressources, etc..
- Et surtout, nos solutions de déploiement deviennent indépendantes de l'évolution des outils, car elles ne s'expriment plus en termes d'outils mais de rôles.

Pour utiliser cette approche, notre environnement devra donc être capable de faire le lien entre les outils et les rôles. La solution choisie sera décrite dans la partie implémentation de cette thèse.

4.4. Le problème du comportement des outils

Utiliser des outils existants implique que l'on soit capable de tenir compte du comportement des outils. En effet, ces outils gardent leur propre fonctionnement autonome voire incontrôlé. C'est principalement le cas des outils faisant intervenir des êtres humains.

Il est impossible d'empêcher un outil d'avoir son propre comportement, car nous avons fait le choix de ne pas les modifier. Par contre, nous devons être capables d'espionner le comportement de ces outils. En effet, dans notre approche nous avons décrit nos solutions en

termes d'actions d'outils sur des ressources; la moindre modification (non imposée par l'exécution de la solution) doit être connue, car elle risque d'influencer la suite de l'exécution. Il faut donc que l'environnement gère et synchronise sa connaissance de l'état de l'environnement de déploiement. Nous avons choisi d'utiliser pour chaque outil un connecteur, celui-ci ayant la charge d'avertir l'environnement de déploiement des différentes actions de l'outil sur l'environnement.

4.5. Le problème de l'exécution à distance

L'intérêt d'automatiser le déploiement croît considérablement si l'on est capable de réaliser ce déploiement à distance, c'est à dire sans intervenir directement à partir de la machine cible. Ceci implique que l'on soit capable d'exécuter les outils impliqués dans la solution à partir soit du site du producteur de l'application à déployer ou bien à partir d'une machine de l'entreprise ceci pour des raisons de sécurité.

L'exécution à distance d'outils pose le problème de la sécurité : les entreprises ne souhaitant pas la plupart du temps que l'on déploie des applications sans qu'elles en aient connaissance et donné l'autorisation.

Pour traiter ce problème, ORYA offre la possibilité de choisir le niveau d'automatisation allant du tout automatique à la validation de chaque étape.

Un autre problème est posé par la présence de firewalls protégeant les entreprises. Notre environnement doit donc apporter des solutions acceptables du point de vue de la sécurité.

4.6. Le problème du partage des ressources

Dans notre approche, les scénarios de déploiement s'expriment en termes d'exécutions d'outils manipulant des ressources. Ceci entraîne des problèmes de synchronisation entre la connaissance de l'état de ses ressources par l'environnement et la réalité. De plus, comme on vient de le voir, les outils ont leur propre comportement, ils peuvent donc intervenir sur les ressources quand ils ne sont plus sous le contrôle de l'environnement. Il faut donc que l'environnement soit capable de connaître et contrôler l'état des ressources afin de pouvoir mettre à jour sa propre représentation.

Nous avons choisi de distinguer :

- les ressources partagées par les outils, c'est à dire en fait les ressources que l'on va manipuler/utiliser dans les scénarios de déploiement
- les ressources propres aux outils (même si elles sont utilisées par plusieurs outils), la gestion de ces ressources étant à la charge des outils eux-mêmes.

ORYA permet donc le partage de ressources et plus précisément assure la cohérence de l'état de ces ressources afin de les utiliser lors du déploiement.

4.7. Conclusion

Le choix de réutiliser des outils existants et surtout de les modéliser permet de décrire totalement le déploiement au niveau modèle. De cette façon ORYA peut évoluer (à travers l'évolution des outils) et reste plus ou moins indépendant des outils. On augmente même les

possibilités de déploiement en offrant le choix des outils. Par exemple, on peut décrire un scénario de déploiement et ensuite le réaliser pour un environnement Windows (avec des outils fonctionnant sous Windows) et aussi pour un environnement Linux (avec cette fois des outils fonctionnant sous Linux).

5. Notre environnement : ORYA

5.1. Architecture

Nous avons dans le chapitre 2 décrit une architecture de déploiement à 3 niveaux. : celui du producteur, de l'entreprise et des sites. On retrouve plus ou moins directement cette architecture dans ORYA (figure 1). Le niveau du producteur est représenté par le serveur d'applications. Les sites représentent les machines sur lesquelles le déploiement va se réaliser, c'est à dire le niveau client. Le niveau entreprise est représenté par le serveur d'entreprise, il est chargé de fournir une vue globale de l'entreprise et de fixer les politiques de déploiement de l'entreprise.

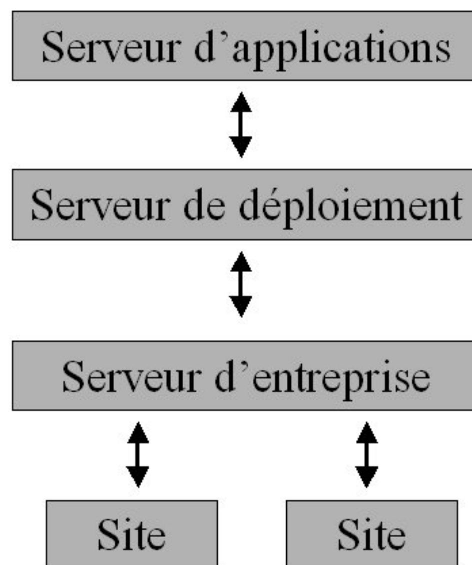

Figure 9: l'architecture physique d'ORYA

Nous avons rajouté un quatrième élément dans notre architecture, il s'agit du serveur de déploiement. Il a pour rôle de piloter, de gérer le déploiement et de servir d'intermédiaire entre les entreprises et les producteurs. Nous allons maintenant décrire rapidement chacune de ces entités; elles seront décrites plus en détails dans les chapitres suivants.

5.2. Les entités

5.2.1. Le serveur d'applications

Le serveur d'applications, comme son nom l'indique, a pour fonction de stocker les applications à déployer et de les mettre à disposition de l'environnement de déploiement. Le serveur d'applications, dans notre environnement, sert de représentant à un ou plusieurs

producteurs. Il permet ainsi, dans le cas des applications à base de composants de servir de support pour l'activité de sélection (chapitre 5).

5.2.2. Le serveur d'entreprise

Son rôle consiste à établir et maintenir une connaissance globale de l'entreprise. Cette information est vitale pour le déploiement, car elle permet de préparer en amont les différentes activités du déploiement. Par exemple, si l'entreprise souhaite mettre à jour une des applications installée sur certaines de ses machines, le rôle du serveur d'entreprise est de fournir la liste des machines sur lesquelles cette application est installée et qui sont concernées par cette mise à jour (celles qui ont une ancienne version d'installée).

L'autre rôle du serveur d'entreprise est de supporter les politiques de déploiement associées à l'entreprise. Choisir de déployer une application sur toutes les machines en même temps (technique du big-bang) est un exemple d'une telle politique.

Le serveur d'entreprise peut aussi servir de répertoire temporaire pour les "packages" à déployer, si l'on doit par exemple les déployer sur plusieurs machines. Cela limite les transferts de données entre les producteurs et les sites et augmente entre autres le niveau de sécurité. Le serveur d'entreprise sert aussi d'intermédiaire entre les producteurs et les sites.

5.2.3. Le serveur de déploiement

Son rôle consiste à gérer les différentes activités du cycle de vie du déploiement. Il peut résider chez le producteur de l'application à déployer ou bien directement dans l'entreprise si celle-ci veut diminuer les risques liés au déploiement. Notamment en cas d'exécution à distance d'outils sur les machines.

Dans le cas où le serveur de déploiement se trouve dans l'entreprise, il ne pourra gérer que les déploiements concernant l'entreprise, mais par contre il servira d'intermédiaire avec l'ensemble des producteurs. A l'inverse s'il reste chez le producteur il ne s'occupera que des déploiements concernant les produits de ce producteur mais, potentiellement, pour l'ensemble des entreprises.

5.2.4. Le site

Il s'agit du représentant des machines. Son rôle consiste à servir d'interface pour la machine qu'il représente. Cette interface doit fournir l'état de la machine, c'est à dire les caractéristiques matérielles (les ressources) et les caractéristiques logicielles (la description des applications déjà déployées).

5.3. La technologie des fédérations

Pour réaliser ORYA nous nous sommes basés sur l'approche des fédérations [LEV03, EVL+03, ELV03] développée dans notre équipe. Elle permet de faire interopérer des outils hétérogènes, de les exécuter à distance (en réglant les problèmes de communications liés aux firewalls par exemple). Elle propose aussi une façon de coordonner et contrôler le comportement des outils. Dans cette section, nous donnerons seulement un rapide aperçu des possibilités des fédérations et plus particulièrement celles qui nous ont permis de résoudre certains des problèmes évoqués dans la section 4.

5.3.1. L'infrastructure de communication

En s'appuyant sur la technologie des fédérations [Vil03], c'est à dire en faisant d'ORYA une fédération de déploiement, nous bénéficions de l'infrastructure de communication propre aux fédérations. Cette infrastructure se caractérise par :

- la connaissance de l'état de chacun des outils en exécution dans l'environnement. Ceci facilite le travail de l'environnement sur la connaissance de son propre état.
- l'utilisation d'un outil de démarrage : le « starter ». Il a pour fonction de lancer à distance l'exécution d'outils sur les machines de l'environnement, ce qui va nous permettre de réaliser les scénarios de déploiement à distance sans se préoccuper de la façon d'exécuter les outils de déploiement.
- des capacités de synchronisation en cas de problèmes. En cas de panne, par exemple en cas de déconnexion impromptue d'une machine du réseau, la fédération est capable de rétablir l'état de la machine (et de ses outils) avant la panne. Dans le contexte du déploiement à grande échelle un tel comportement est obligatoire

5.3.2. L'environnement de procédé

Comme on l'a vu précédemment dans la section 4.2, ORYA utilise les procédés pour décrire l'enchaînement des différents outils de déploiement. La fédération fournit un environnement de conception et d'exécution de procédés : "APEL" [DEA98]. Il est composé principalement de 3 outils :

- un éditeur de procédés permettant de décrire graphiquement les procédés,
- un moteur de procédé, chargé d'interpréter les procédés,
- un agenda permettant de visualiser et de réaliser les activités d'un procédé (dans le cas d'une activité manuelle).

L'éditeur permet de décrire graphiquement un procédé et plus particulièrement l'enchaînement logique des différentes activités constituant le procédé. Pour cela APEL utilise une syntaxe graphique basée sur un ensemble restreint de concepts : celui d'activité, de ports, de flots de données, de poste de travail et de produits. Nous reviendrons sur ces différents concepts dans le chapitre 6. La figure donne un exemple de la description graphique utilisée par APEL :

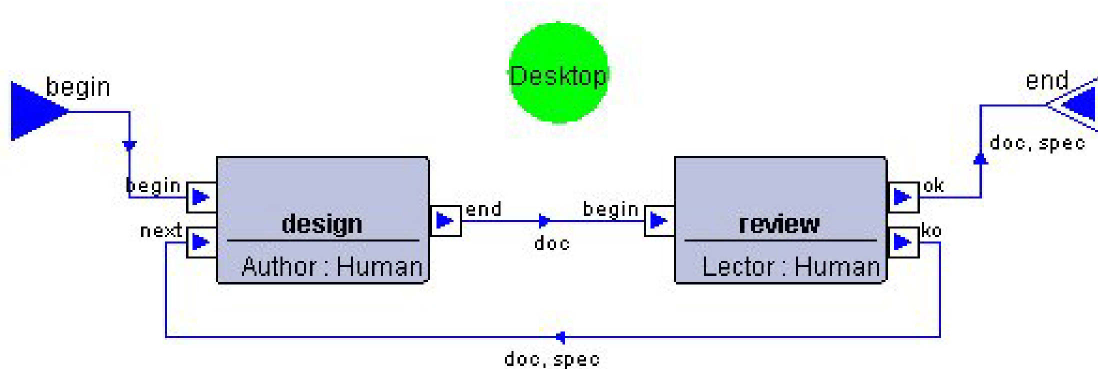

Figure 10 : Exemple de description graphique d'un procédé

Une fois les procédés décrits, il faut les exécuter. Pour cela APEL propose un moteur de procédé dont la fonction est d'interpréter ce langage graphique afin de piloter et de gérer le procédé en exécution.
Ce moteur permet de :

- gérer le cycle de vie des activités
- gérer les flots de données
- gérer le cycle de vie des produits

Le dernier outil proposé par APEL est un outil permettant à la fois d'observer le déroulement des procédés en exécution et aussi d'intervenir en cas d'activités manuelle par exemple. L'agenda (c'est le nom de l'outil) est associé de façon explicite à un utilisateur (dans le cas du déploiement, il peut s'agir d'un administrateur réseau, d'un responsable de déploiement, voire de l'utilisateur d'une machine sur laquelle le déploiement a lieu). Le lien entre les utilisateurs et les procédés se fait au niveau des activités; chaque activité étant affectée à un utilisateur (ou type d'utilisateur).

5.3.3. L'exécution à distance

Comme on l'a vu avec l'infrastructure de communication, la fédération offre un service de démarrage à distance d'outils. Pour cela, la technique utilisée consiste à écrire pour chaque outil un mandataire et un connecteur (figure 2).

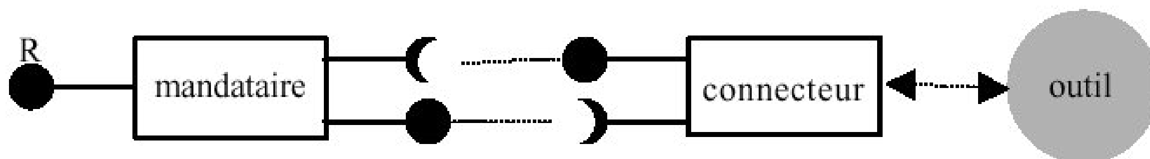

Figure 11 : description de la connexion d'un outil dans une fédération

Le noyau de la fédération (R), l'élément central d'une fédération, communique avec un outil par l'intermédiaire du mandataire de l'outil qui se charge (à l'aide de l'infrastructure de communication) de répercuter les demandes de la fédération à l'outil via son connecteur. Le connecteur est aussi chargé d'écouter les événements déclenchés par l'outil afin de les répercuter au noyau. De cette façon, la fédération peut éventuellement contrôler les outils.

5.4. Conclusion

Comme on le voit, construire notre environnement de déploiement en s'appuyant sur une fédération de déploiement permet de résoudre bon nombre de problèmes techniques inhérents au déploiement comme la communication, le retour sur panne, l'exécution à distance, les firewalls, etc..

Ainsi nous allons pouvoir traiter les différentes activités du cycle de vie du déploiement ainsi que sa gestion sans avoir à nous préoccuper ni prendre en compte les problèmes cités précédemment.

6. Conclusion sur l'approche

Nous venons de donner dans ce chapitre une vision globale de notre approche pour réaliser un environnement de déploiement couvrant toutes les activités du cycle de vie du déploiement.

ORYA est un environnement ouvert basé sur l'utilisation d'outils de déploiement existants. Le choix de décrire les scénarios de déploiement à un niveau plus abstrait que celui de la réalité permet de diminuer le nombre de scénarios à écrire. ORYA propose en parallèle des techniques pour "instancier" ces scénarios en tenant compte de l'état de l'environnement, des politiques de déploiement des entreprises. De cette façon, nous sommes capables de réaliser notre objectif de ne déployer que les configurations les plus cohérentes sur chacune des machines de l'entreprise. Et ceci de façon automatisée et contrôlée.

Dans les chapitres suivants, nous allons décrire plus particulièrement les activités du cycle de vie traitées dans le cadre de cette thèse : la sélection et l'installation. Dans ces chapitres nous reviendrons plus en détails sur certains choix ou solutions proposées dans le cadre d'ORYA.

Chapitre V

La sélection

1. Introduction

Dans ce chapitre, nous nous sommes intéressés à la première activité du cycle de vie du déploiement : la sélection. L'objectif de cette activité est de construire les "packages" qui seront ensuite installés sur les sites.

On a vu dans le chapitre précédent que l'un des objectifs de notre environnement de déploiement (ORYA) était d'installer sur chaque machine une configuration particulière des applications. Cet objectif ne s'applique qu'aux applications pouvant exister en plusieurs versions.

Ces constructions peuvent avoir lieu en amont de l'installation (ou de la mise à jour). Dans ce cas, l'activité de sélection consiste à construire des applications qui seront ultérieurement déployées. La sélection peut aussi intervenir pendant le déroulement de ces activités au cas où aucune des versions proposées (déjà construites) ne soit déployable. Dans ce cas on essayera de construire une nouvelle version de l'application à déployer.

Dans notre approche, nous avons choisi de traiter la sélection à travers des applications basées sur un modèle de composant (section 3). La sélection va donc se ramener à la sélection de composants afin de construire la version souhaitée [Katz90]. Cette sélection nécessite de connaître les composants, c'est à dire de connaître leurs fonctionnalités et leurs contraintes de sélections. Pour cela nous utilisons un framework d'annotation (section 4) [VBC02].

Ce chapitre se termine par la description du résultat (section 5) de la sélection en vue de son utilisation dans les activités suivantes du cycle de vie du déploiement (installation, mise à jour…).

2. La sélection

2.1. Définitions

2.1.1. Application/logiciel

Dans cette section, nous définissons ce que signifie une application du point de vue du déploiement. Une application est un ensemble d'artefacts (ou de ressources). Un artefact correspond à un exécutable, à de la documentation, à des données, à des librairies, bref à tout ce qu'un fichier peut contenir. Cette définition n'est cependant pas suffisante du point de vue du déploiement car elle ne tient pas compte des relations que possède une application.

Nous avons identifié différents type de relations :

- des relations internes (à l'application) de type architectural avec les connexions entre les artefacts d'une même application,
- des relations externes comme les dépendances avec d'autres applications,
- des relations avec l'environnement dans lequel l'application sera déployée.

On peut donc donner la définition suivante pour une application : un ensemble d'artefacts (ou de ressources) ainsi qu'un ensemble de relations.

2.1.2. Version

L'un des intérêts d'un déploiement automatisé est de pouvoir déployer sur chaque machine la version la plus cohérente de chaque application. Pour cela il faut disposer d'applications versionnées (c'est à dire existant sous plusieurs versions) afin de pouvoir choisir celle qui correspondra le mieux.

En quoi deux versions d'une même application sont elles différentes ? De façon simple, elles peuvent avoir :

- une architecture différente (des artefacts différents),
- des versions différentes pour certains des artefacts.

Ces deux différences sont implicitement liées, car comme on l'a vu les artefacts ont des dépendances entre eux. Ces dépendances peuvent être différentes selon les versions des artefacts, d'où les différences en terme d'architecture.

La problématique de la sélection va être de savoir, comment sélectionner les bons artefacts afin de créer une application cohérente et exécutable.

2.1.3. Famille de logiciel

Nous venons de voir qu'un logiciel peut exister sous plusieurs versions. Une famille de logiciel correspond à l'ensemble des versions d'un même logiciel. Nous pouvons maintenant dire que l'objectif de l'activité de sélection est de "créer" une application issue d'une famille de logiciel.

A partir d'une famille, on peut construire un ensemble d'applications. On a vu que celles-ci se différenciaient à travers les versions des artefacts composant ce logiciel. Comment se différencient les versions d'un même artefact ? Il existe plusieurs facteurs qui imposent le versionnement des artefacts (ils sont valables aussi pour les logiciels) :

- les fonctionnalités,
- les contraintes.

Un artefact offre un certain nombre de fonctionnalités et impose des contraintes (par exemple le système d'exploitation sous lequel les applications contenant cet artefact doivent s'exécuter). Deux versions d'un même artefact différent le plus souvent sur les contraintes et parfois sur les fonctionnalités.

2.2. Le choix des modèles de composant

Nous avons choisi de baser notre étude de l'activité de sélection sur des applications à base de composants, et plus particulièrement sur des applications issues de notre modèle de composant (qui se veut générique et capable de recouvrir les modèles à composants existants).

L'intérêt d'utiliser un modèle de composant est multiple. Tout d'abord les applications sont versionnées (les implémentations peuvent être vues comme autant de versions). Ensuite, il est relativement aisé de réaliser l'activité de sélection sur ce type d'application, car on dispose d'une architecture déjà pré-établie. La sélection consistant à choisir parmi les différentes

implémentations, celle(s) qui correspondent le mieux à nos exigences (souhaits et contraintes).

La section 3 détaille ce modèle de composant.

2.3. Le choix des annotations

Le choix des implémentations doit correspondre à un choix basé à la fois sur les fonctionnalités proposées par les implémentations et sur les contraintes imposées par celles-ci. C'est à dire basé sur la description des composants. Nous avons donc choisi d'associer à chaque composant une annotation permettant de le décrire. Nous reviendrons plus en détail sur ces annotations lors de la description du framework d'annotation (section 4).

3. Le modèle de composant

3.1. Les objectifs de ce modèle

Le modèle de composant, qui nous a servi de support pour la réalisation de l'activité de sélection du déploiement, a été développé par notre équipe. Il a servi de base à plusieurs travaux [Duc02]. L'objectif de ce modèle est d'étendre et d'ordonner les concepts (connexions, communications, etc.) des modèles de composant actuels.

Nous n'allons pas décrire précisément toutes les caractéristiques de ce méta-modèle, mais nous allons plutôt nous intéresser à celles qui concernent le déploiement.

3.2. Principes

Notre modèle de composant est fortement inspiré de CCM [CCM]. Nous retrouvons donc certains des concepts de CCM comme les rôles et les connexions entre les composants via les ports.

Nous avons étendu le modèle de CCM en ajoutant le concept de composant composite [Sal01], c'est à dire qu'un composant peut être composé d'autres composants (plus précisément de références vers d'autres composants) et pourra ensuite lui-même être utilisé lors d'une composition.

Nous avons ajouté dans le modèle les états intermédiaires, c'est à dire les composants non entièrement définis. Par exemple, nous pouvons avoir un composant composite composé avec un ou plusieurs rôles (c'est à dire une interface sans son implémentation).

En conséquence, nous avons ajouté une opération dite de "raffinement" qui permet de définir, un peu plus, un composant se trouvant dans un état intermédiaire. Cette opération consiste à choisir, pour l'un au moins des rôles du composant (que l'on raffine), l'une de ses implémentations. Autrement dit, les niveaux intermédiaires faisant référence à des rôles peuvent être plus précisément définis, en remplaçant les références vers des rôles par des références vers des implémentations. Nous développons plus précisément cette opération dans la suite de ce chapitre.

Nous allons maintenant décrire notre méta-modèle (Figure 12) de composant (version simplifiée) [BCLS01].

3.3. Le méta-modèle de composant

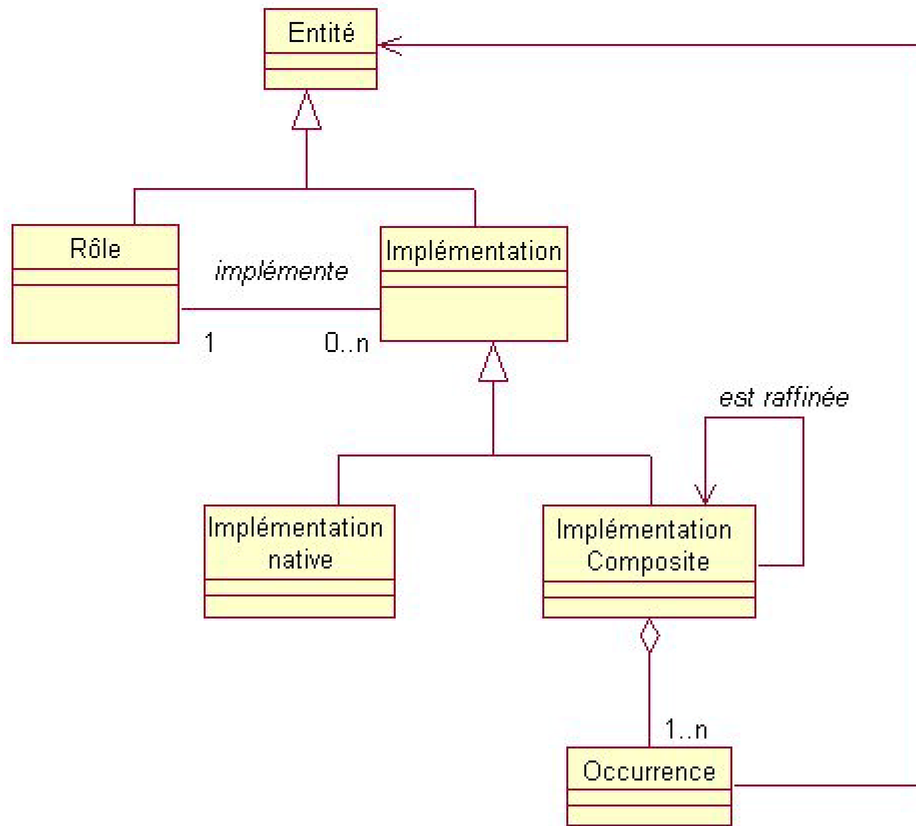

Figure 12 : Modèle simplifié du modèle de composant

Nous avons utilisé la représentation UML [UML97, RJB99] pour décrire ce modèle ainsi que les autres modèles présentés dans ce document de thèse. Dans la suite du document, nous utilisons parfois le terme de composant (appelé entité dans notre méta-modèle) pour parler des rôles ou des implémentations.

Dans cette version simplifiée, qui ne fait pas apparaître les connexions entre les composants, deux concepts apparaissent : les entités (ou composants) et les relations.

3.3.1. Les entités

3.3.1.1. Les rôles

Il s'agit de la vue abstraite d'un composant. Elle est équivalente à celle spécifiée dans CCM. Un composant possède un ensemble de "ports" (non représentés dans la Figure 12) composé de facettes, de réceptacles, de sources d'événements et de puits d'événements. La Figure 13 donne une représentation d'un rôle.

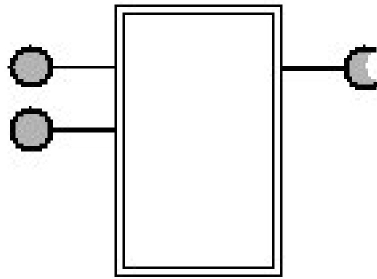

Figure 13 : Représentation d'un rôle

3.3.1.2. Les implémentations

On dit qu'une implémentation joue un rôle lorsqu'elle possède (au niveau externe) les mêmes ports que ce rôle. Autrement dit, une implémentation réalise un rôle, c'est à dire qu'elle réalise les fonctionnalités "promises" par le rôle. Dans cette version simplifiée du modèle, nous n'avons pas permis qu'une implémentation puisse jouer plusieurs rôles. Il existe plusieurs types d'implémentations : les implémentations natives et les implémentations composites.

Une implémentation native correspond à l'encapsulation d'un objet basique (écrit en langage natif). La Figure 14 représente une implémentation native.

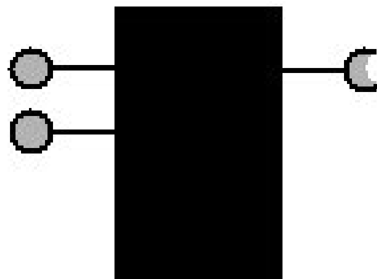

Figure 14 : Représentation d'une implémentation native

Une implémentation composite (aussi appelée assemblage) correspond à la composition de rôles et d'implémentations. Pour faciliter la compréhension du modèle, nous avons introduit le concept d'entité (appelé aussi composant). Une entité correspond au concept de la classe Objet dans un langage orienté objet. La Figure 15 donne la représentation externe d'une implémentation composite (les constituants du composite n'apparaissant pas dans cette représentation).

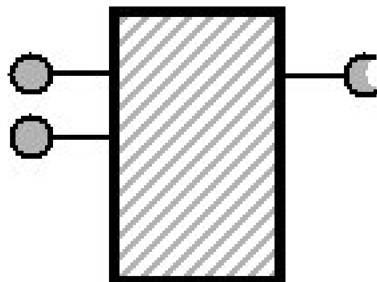

Figure 15 : Représentation externe d'une implémentation composite

Dans notre méta-modèle de composant, une composition est constituée de composants (utilisation du pattern composite [GHJV95]), qui sont en fait des références de composant (voir la section 3.3.3.1). De cette façon, notre modèle permet la réutilisation des composants. En effet, si l'on réalisait nos compositions avec les composants directement, ceux-ci ne pourraient faire parti que d'une seule composition (ce qui en limiterait très fortement la réutilisation). La Figure 16 donne un exemple d'un composite constitué de 2 constituants représentant des occurrences d'un composite et d'un rôle.

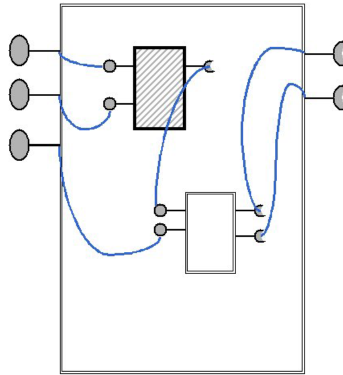

Figure 16 :Exemple de la représentation interne d'une implémentation composite

Nous allons maintenant expliciter les différentes relations permettant de lier ces différentes entités. Nous distinguons deux types de relations :

- les relations de création,
- les relations de parcours.

3.3.2. Les relations de création

3.3.2.1. Définition

Une relation est dite de création lorsqu'elle aboutit à la création d'un nouveau composant. Dans notre cas, nous avons deux relations de création : celle d'implémentation et celle de raffinement. La relation d'implémentation permet de créer des implémentations (composite et native) et la relation de raffinement permet elle de créer des implémentations composites.

3.3.2.2. La relation d'implémentation

Elle permet de lier un rôle à ses implémentations. Chacune de ses implémentations doit satisfaire le contrat défini par le rôle via ses ports. Cette relation peut être comparée à la relation de versionnement du domaine de la gestion de configuration. Chaque implémentation correspondant alors à une version différente. Ces versions ne sont pas toutes exécutables, car elles peuvent correspondre à un niveau intermédiaire (cas d'un composite dont au moins une des occurrences fait référence à un rôle).

Comme ces implémentations (non définies) peuvent être raffinées en d'autres implémentations, l'ensemble des versions de l'application (représentée par le rôle) correspond

à l'ensemble des implémentations liées directement au rôle (via la relation d'implémentation) et à l'ensemble des raffinements issus de ces implémentations.

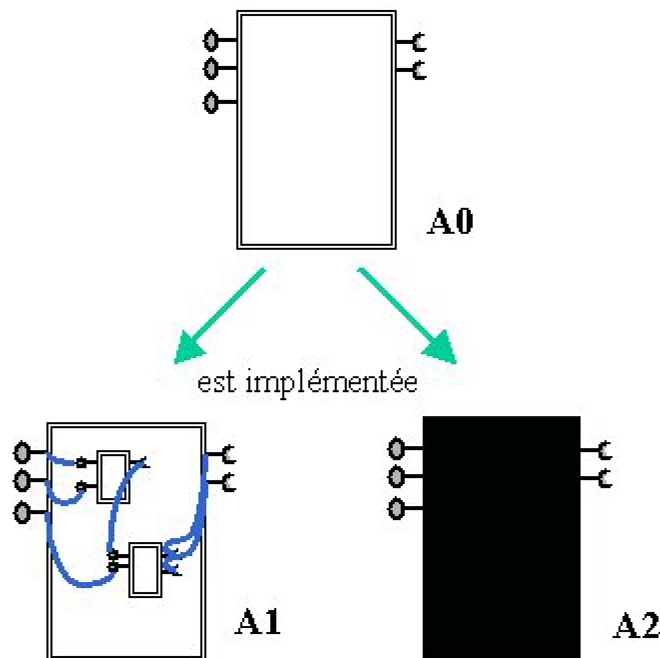

Figure 17 : Exemple d'implémentations

L'exemple de la Figure 17 donne un exemple du concept d'implémentation. L'entité A0 est implémentée par deux implémentations : une implémentation composite (A1) et une implémentation native (A2).

3.3.2.3. La relation de raffinement.

Comme nous avons commencé à l'expliquer précédemment, la relation de raffinement permet de mieux définir une implémentation non définie. Cela consiste à choisir, pour l'une au moins des occurrence référençant un rôle, une de ses implémentations. Cette implémentation peut être native ou composite.

L'exemple de la Figure 18 permet d'illustrer le concept de raffinement. L'entité A1 représente une implémentation composite comprenant deux occurrences de rôle. A1 est ensuite raffiné par A1.1 qui ne contient plus que des occurrences natives. A1.2 est aussi un raffinement de A1 mais elle reste indéfinie car l'une de ses occurrences correspond encore à un rôle. Elle pourra elle-même être raffinée.

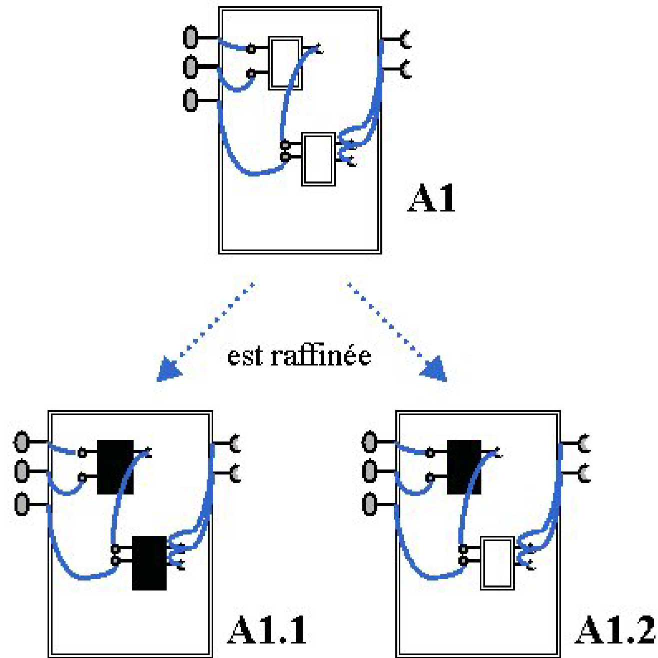

Figure 18 : Exemple de raffinements

3.3.2.4. Exemple de relations de création

Nous allons maintenant décrire un exemple, que nous réutiliserons tout au long de ce chapitre pour illustrer les différentes concepts utilisés dans l'activité de sélection.

Nous nous intéressons à 3 composants (A, B et C). Chacun de ces composant possède un rôle et une ou plusieurs implémentations. Par exemple, A (Figure 19) possède un rôle *A0* et six implémentations (*A1, A2, A1.1, A1.2, A1.2.1* et *A1.2.2*). Le composant *A2* est une implémentation native liée au rôle *A0* par la relation d'implémentation. Cela signifie, dans la réalité, que du code natif (écrit en java, c++ ou autre) correspondait au contrat (les fonctionnalités) décrit par *A0* et qu'il a été encapsulé afin de devenir une implémentation native.

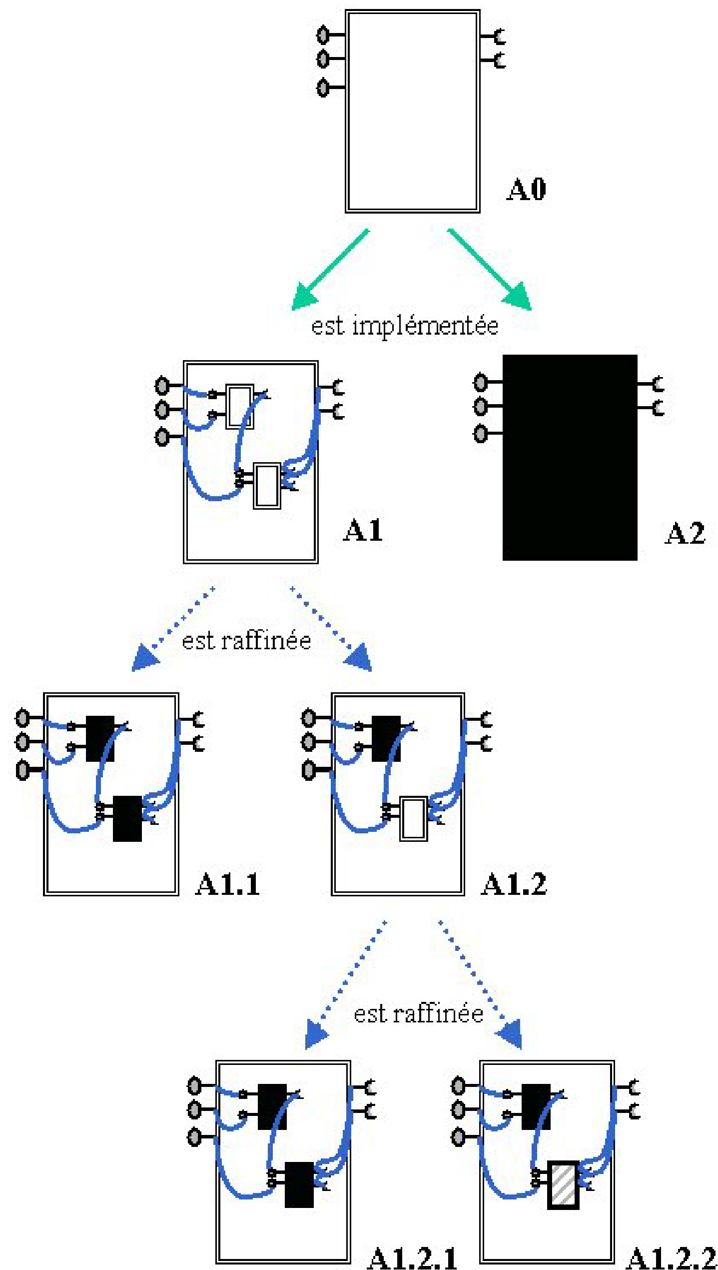

Figure 19 : Exemple de composants et de relations

Le composite *A1* est lui aussi lié à *A0* par la relation d'implémentation. Il n'est pas exécutable, car ses deux constituants sont des occurrences de rôles. Dans la pratique ce genre de composant correspond à une étape intermédiaire de développement. Sa présence en tant qu'implémentation signifie qu'il s'agit d'un nœud et que des versions différentes peuvent être développées à partir de lui. Et c'est le cas dans notre exemple, *A1* est raffiné par *A1.1* et *A1.2*. *A1* en fait correspond par exemple à une implémentation dont le choix de la plate-forme d'exécution n'a pas encore été fait et *A1.1* a été développé pour fonctionner sous Windows (ce qui explique qu'il soit entièrement défini) alors que *A1.2* fonctionne sous Linux mais le choix de la distribution n'a pas encore été fait (choix symbolisé par l'occurrence de rôle constituant *A1.2*). Il faut donc raffiner *A1.2*. Dans notre exemple, il existe deux raffinements de ce composite : *A1.2.1* (distribution Mandrake) et *A1.2.2* (distribution RedHat).

3.3.3. Les relations de parcours

3.3.3.1. La base de développement de composants

Tout d'abord il faut préciser que nous avons choisi d'utiliser une base de données contenant des composants (les rôles et les implémentations). Cette base est dite de développement, car elle contient aussi les implémentation non entièrement définies (celles qui peuvent être raffinées). La base de données est relationnelle, car les composants sont reliés entre eux via les différents types de relations : les relations de création et de parcours. Le modèle de la Figure 12 ne contient qu'une seule relation de parcours : la relation d'occurrence.

3.3.3.2. La relation d'occurrence

La relation d'occurrence permet de lier un membre d'une composition à un élément de la base de développement. Dans l'exemple de la Figure 18, l'implémentation composite A1 est constituée de deux occurrences de rôles. Chacune de ses occurrences de rôle est une référence vers un rôle de la base. La section suivante détaille plus précisément la relation d'occurrence à travers un exemple.

La relation d'occurrence peut être vue comme une relation de dépendance. Une implémentation composite ayant alors deux types de dépendance : des dépendances externes (symbolisées par les réceptacles) et des dépendances internes (les occurrences). Une occurrence peut représenter un rôle ou une implémentation. Dans le cas d'un rôle, le composite père pourra être raffiné, il suffira de remplacer le rôle par l'une de ses implémentations

3.3.3.3. Exemple de relations de parcours

Nous allons maintenant expliciter plus en détails la relation d'occurrence vue dans la section 3.3.2.3. Reprenons le composite *A1.2* et étudions ses occurrences nommées *O1* et *O2* (Figure 20). *O1* est une occurrence de l'implémentation native *C1* (qui implémente le rôle *C0*) et *O2* est une occurrence du rôle *B0*. Dans notre approche, *O1* est lié à *C1* par la relation d'occurrence.

Dans notre exemple, le composite *A1.2* est raffiné par les composites *A1.2.1* et *A1.2.2*. A1.2 est composé d'une occurrence de rôle et d'une implémentation native. Par conséquent, d'après notre modèle de composant, une implémentation raffinant A1.2 ne peut que le faire en choisissant une occurrence d'une des implémentations de ce rôle. Dans notre exemple, le rôle *O2* est une occurrence du rôle *B0*, qui est implémentée par *C1* (un natif) et *C2* (un composite). Raffiner *A1.2* consiste donc à remplacer l'occurrence de *B0* par soit une occurrence de *C1* (c'est le choix fait pour le composite *A1.2.1*) soit par une occurrence de *C2* (choix de *A1.2.2*).

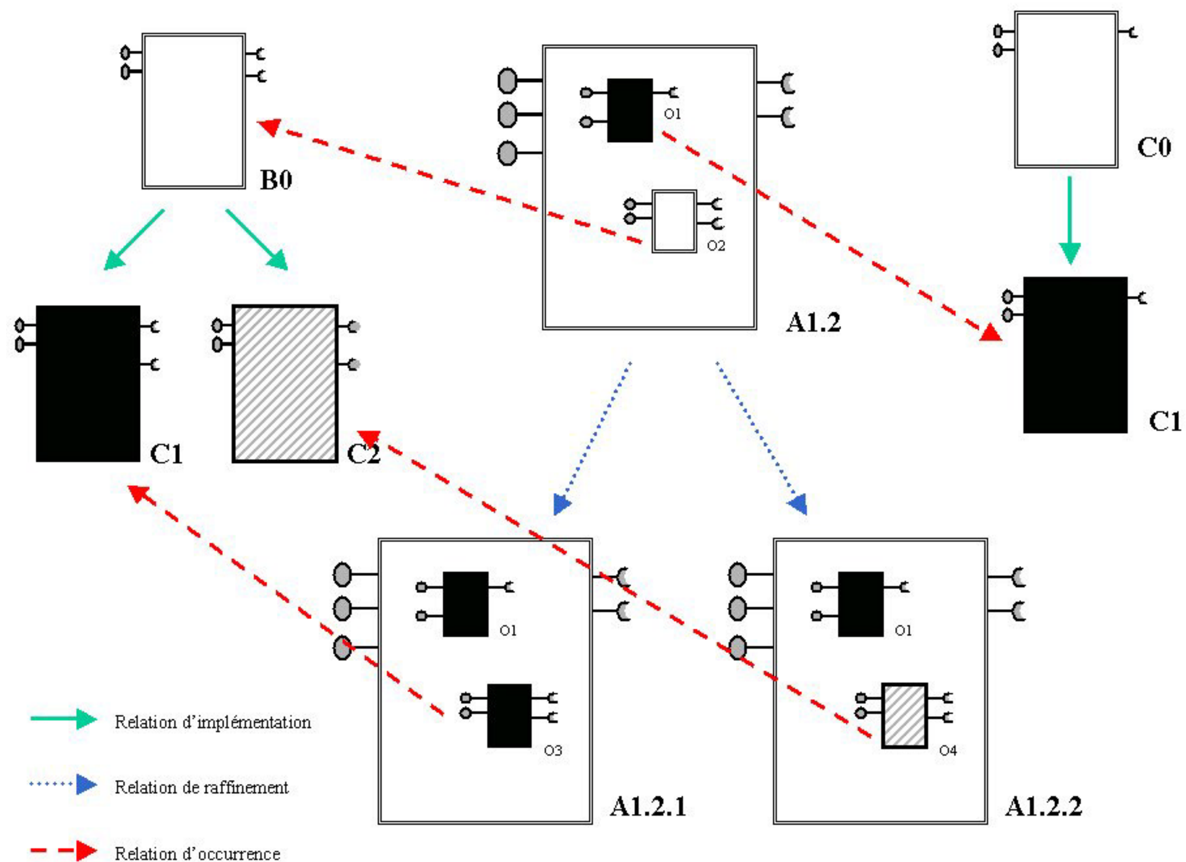

Figure 20 : Exemple de relations d'occurrence

Remarque : dans la Figure 20 nous avons pour des raisons visuelles enlever les connexions internes dans les composites.

3.4. Conclusion sur le modèle

Pour réaliser le déploiement, nous avons besoin de l'architecture de l'application. Nous disposons maintenant d'un modèle de composant permettant de décrire l'architecture d'une application et les liens entre les différents éléments de l'application (un rôle est implémenté par un composite, lui même est raffiné par un autre composite composé d'une occurrence de rôle, elle même implémentée….).

L'activité de sélection (dans le cas d'applications basées sur notre modèle) a comme point de départ un rôle. La sélection va consister à choisir une implémentation parmi les implémentations de ce rôle. Et ensuite éventuellement de choisir une implémentation raffinant la précédente. Dans le cas d'occurrences de rôles ou de composites, on applique le même procédé avec chacune de ces occurrences.

Nous allons maintenant nous intéresser à la façon de décrire les composants, description qui va permettre de choisir le bon composant par rapport à des critères de sélection. Pour cela, nous avons réalisé un framework d'annotation.

4. Le framework d'annotation

4.1. Principes

Afin de respecter notre objectif de fournir un environnement de déploiement le plus ouvert (et extensible) possible, nous avons fait le choix de définir un système d'annotation non exclusivement lié à notre modèle de composant. Pour cela nous nous sommes appuyés sur la définition suivante :

« Une application est composée d'entités liées entre elles par des relations de création et de parcours. Une application pouvant elle-même être utilisée dans une autre composition » [BCLS01].

La sélection va consister à parcourir la base de développement décrite dans la section 3.3.3.1 (le point de départ étant un rôle) et ensuite à "construire" la future application en choisissant à chaque fois la meilleure solution possible par rapport à la configuration souhaitée.

Nous allons maintenant décrire notre système d'annotation en insistant plus particulièrement sur la cohérence à travers la propagation des annotations. En effet, il apparaît important que la description d'une implémentation par exemple ne soit pas en contradiction avec la description de son rôle. Bien sur, la description d'un composant doit être totalement cohérente avec ce que le composant réalise en terme de fonctionnalités.

4.2. Les dépendances entre les annotations

On a vu que les composants sont liés par des relations de création (implémentation, raffinement). Lors de l'ajout d'un composant dans la base de données (par exemple lorsque l'on ajoute une implémentation à un rôle) on doit s'assurer que l'annotation que l'on va lui associer est cohérente avec celle du composant auquel il est relié. On doit aussi vérifier que cette annotation est en accord avec celles des occurrences internes dans le cas d'une implémentation composite.

Nous avons développé un système de propagation d'annotation. Nous avons fait l'hypothèse que chaque composant que l'on ajoute dans la base doit être cohérent avec l'annotation qui lui est associée.

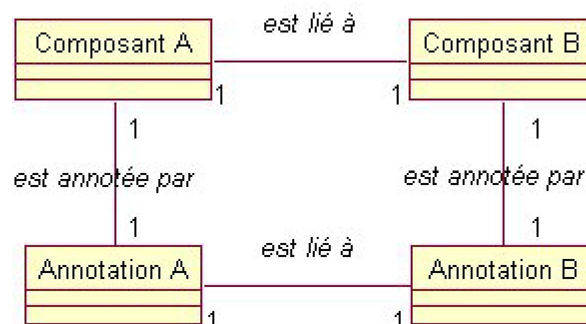

Figure 21 : Interdépendance entre annotation

La Figure 21 décrit l'interdépendance entre les annotations. On dispose d'un composant A qui est annoté par l'annotation A. Ce composant est lié au composant B (par exemple par une relation d'implémentation si A est un rôle et B une implémentation). B est annoté par l'annotation B.

Les composants A et B sont liés entre eux par une relation. On retrouve cette liaison au niveau des annotations. En effet, si l'on considère B comme une version de A, les caractéristiques de B doivent être influencées par celles de A.

Dans le reste de ce chapitre, nous utilisons parfois le terme de composant père et de composant fils. Dans notre exemple, A est le père de B, qui est donc le fils de A.

Nous allons maintenant détailler notre modèle d'annotation permettant d'annoter de façon cohérente les composants.

4.3. Le modèle d'annotation

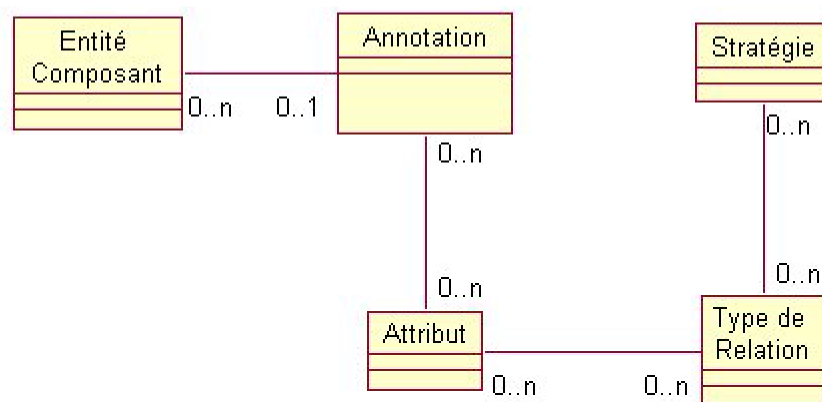

Figure 22 : Le modèle d'annotation

4.3.1. Les annotations

Une annotation permet de décrire une entité composant. On rappelle qu'un composant dans notre modèle peut être un rôle ou une implémentation (native ou composite). Le fait que les composants soient liés entre eux par des relations (de création en particulier) implique que les annotations partagent souvent le même type d'information. Au lieu de typer les annotations, nous avons choisi de définir une annotation comme un ensemble d'attributs. De cette façon notre système d'annotation permet une grande flexibilité (ajout ou suppression d'attributs par rapport à l'annotation du composant père). Par contre, le contrôle de la cohérence entre l'annotation du fils et celle du père est plus difficile.

4.3.2. Les attributs

Un attribut se décompose en deux parties : sa valeur (qui est typée) et sa logique de propagation selon les différentes façons qu'un composant a de se rattacher au composant père. En effet, nous avons choisi d'imposer qu'un composant (par exemple une implémentation) ne puisse se rattacher à un autre composant (par exemple un rôle) que si son annotation respecte entièrement la logique de propagation de l'annotation de son père. Par exemple, supposons que l'annotation d'un rôle comporte un attribut A, dont la valeur doit être commune à toutes les implémentations issues du rôle. On ne pourra ajouter une implémentation que si son

annotation contient un attribut A avec la même valeur. Si l'on reprend notre exemple de tout à l'heure, on obtient maintenant la Figure 23. Elle décrit la dépendance entre les attributs des annotations de composants liés par une relation. Maintenant la dépendance n'est plus exprimée au niveau des annotations mais au niveau des attributs de ces annotations. Concrètement, chaque attribut de l'annotation A peut être lié à un attribut de l'annotation B. Ce lien est sémantiquement fort, car il impose de fait des contraintes obligatoires sur la valeur de l'attribut de B ainsi lié.

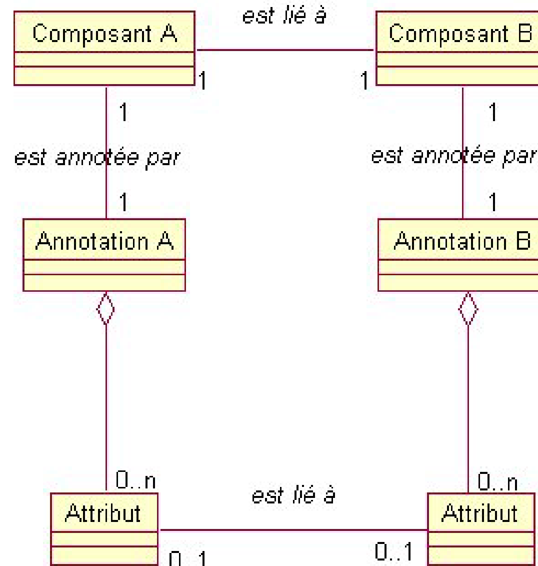

Figure 23 : Interdépendance entre attributs

Dans le but de créer des annotations cohérentes, nous avons donné la possibilité d'associer à un attribut une logique de propagation (une pour chaque relation possible à partir du composant).

4.3.3. Les types de relations

L'entité appelée "type de relation" (voir la Figure 22) permet d'associer à un attribut donné chacune des relations possibles que le composant annoté peut avoir. Par exemple, un rôle peut avoir des implémentations via la relation d'implémentation. Notre framework permet de prévoir la logique de propagation de chaque attribut d'une annotation selon les différentes relations. Dans le cas d'un rôle, on pourra associer aux attributs de son annotation une logique de propagation à utiliser lors de l'ajout d'une nouvelle implémentation du rôle.

Il faut noter que lors de l'ajout d'un composant, une seule relation de création peut être prise en compte (sinon on risque d'avoir des incohérences).

Dans le cas des relations de parcours (comme celle d'occurrence), la logique associée n'est plus une logique de propagation de valeur mais plutôt de vérification de valeur. Par exemple, dans le cas d'une implémentation composite, on devra vérifier que les annotations des occurrences internes sont compatibles avec l'annotation du nouveau composite. Par exemple, il ne faut pas qu'un composite qui doit fonctionner sous Windows se retrouve avec des occurrences (ses constituants) ne fonctionnant que sous Linux.

Nous avons donc associé à chaque attribut des logiques de propagation pour les relations de création (une seule peut être prise en compte) et des logiques de vérifications pour les relations de parcours.

4.3.4. Les stratégies

Les stratégies permettent de décrire les logiques de propagation et de vérification associées aux attributs. Nous avons distingué deux types de stratégies :

- les stratégies liées à la propagation des attributs : elles permettent de définir la façon de propager la valeur d'un attribut,
- les stratégies de vérification : elles permettent de maintenir la cohérence des valeurs d'attributs suivant des règles définies par l'utilisateur.

La copie d'une valeur de l'annotation mère sur l'annotation fille est un exemple de stratégie de propagation. Cela veut dire que si l'on associe cette stratégie à l'un des attributs d'un rôle, toutes les annotations des implémentations de ce rôle devront avoir la même valeur pour cet attribut.

4.4. Exemple

Reprenons notre exemple et utilisons le pour illustrer le framework d'annotation. Pour simplifier nous allons supposer que tous les composants possèdent le même type d'annotation avec les mêmes attributs : le système d'exploitation (OS) et le langage de programmation (langage). Nous représentons les annotations de la façon suivante (Figure 24) :

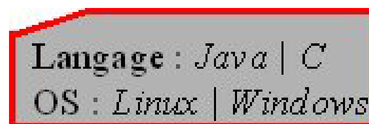

Figure 24 : Exemple d'annotation

L'annotation ci-dessus est celle du rôle *A0*. Elle signifie que le rôle peut fonctionner soit sous Linux ou sous Windows et que le langage de programmation sera Java ou C. Ces informations permettent de dire ce que l'on peut avoir éventuellement comme configuration. Cela signifie aussi que si l'on souhaite une configuration fonctionnant sous Mac, le rôle *A0* ainsi que toutes ses implémentations ne pourront pas être sélectionnés.

A partir de *A0* on peut créer des composants avec la relation d'implémentation. Nous allons pour les deux attributs de l'annotation de *A0* associer des stratégies selon la relation d'implémentation. Nous avons choisi d'utiliser, dans les deux cas, une stratégie de vérification qui a pour but de vérifier que la valeur de l'attribut OS (des futures implémentations) sera soit Linux soit Windows ou les deux. Dans le cas contraire l'implémentation ainsi annotée (avec par exemple la valeur Mac pour l'attribut OS) ne pourra pas être reliée à A0 par la relation d'implémentation. La Figure 25 illustre ce propos.

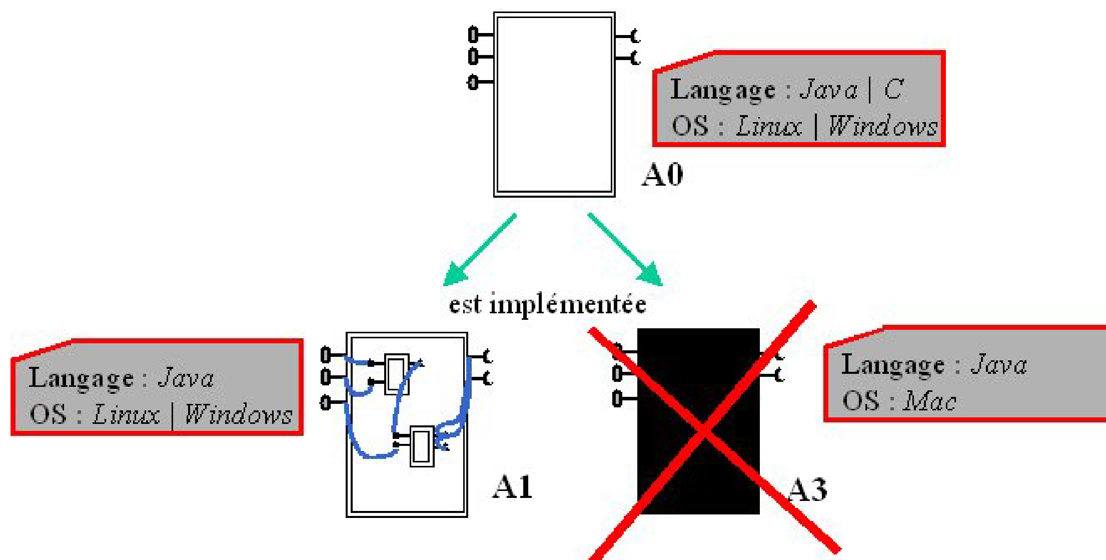

Figure 25 : Exemple d'exécution de stratégie de vérification

Dans cet exemple, on voit que le composant *A3* (fonctionnant sous Mac) ne peut pas être une implémentation de *A0*, cette interdiction étant provoquée par l'application (négative) de la stratégie de vérification de l'attribut OS de l'annotation de *A3*. Par contre l'annotation de *A1* est cohérente avec les stratégies de vérification de *A0*.

Comme on le voit l'annotation de A1 indique que son langage de programmation est Java par conséquent, nous avons choisi d'associer à l'attribut Langage de l'annotation de *A1* une stratégie de propagation permettant d'imposer aux attributs langage des annotations de ses raffinements (il s'agit dans notre exemple de la seule relation de création possible à partir de *A1*) de garder la valeur Java. De cette façon nous assurons que tous les composants issus de A1 sont exécutables dans un environnement Java.

Nous allons maintenant illustrer le cas des relations de parcours, c'est à dire pour nous le cas de la relation d'occurrence.

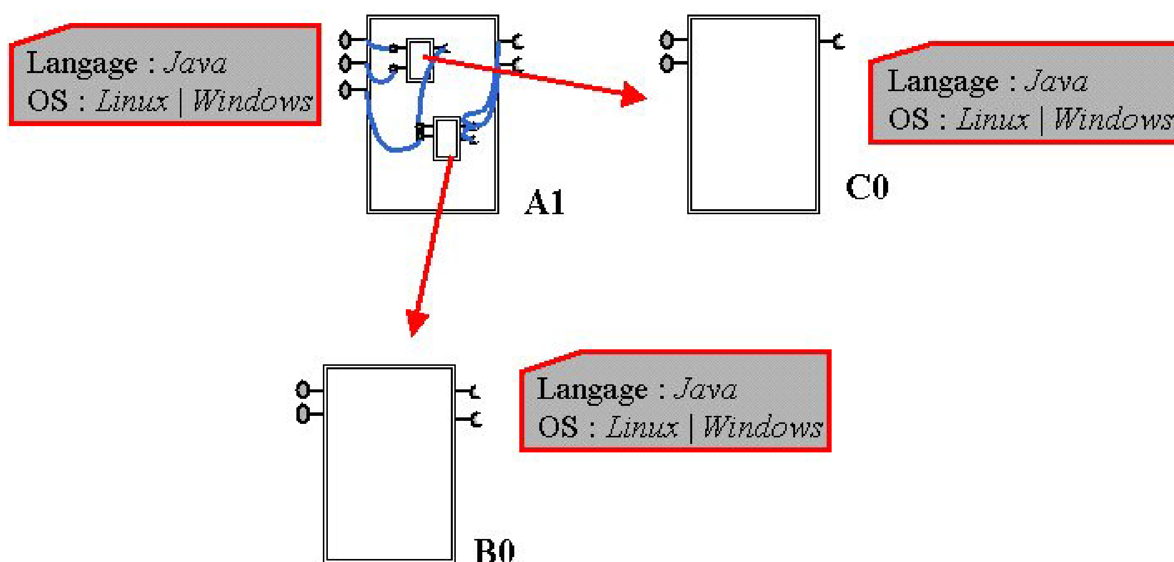

Figure 26 : Exemple d'application des stratégies sur la relation d'occurrence

La Figure 26 permet d'illustrer un exemple de stratégie de vérification associée aux attributs (OS et langage) de l'annotation de *A0* pour la relation d'occurrence. Le composant *A1* n'a pu être ajouté à *A0* que parce que les annotations de ces constituants (les rôles *B0* et *CO*) vérifiaient la stratégie de vérification (nous avons repris la même que dans l'exemple ci-dessus) de chacun des attributs.

4.5. Conclusion sur le framework d'annotation

L'objectif de ce framework d'annotation est de fournir un système assurant la cohérence des annotations et indépendant (le plus possible) d'un modèle à composant donné.

Nous disposons donc d'une base de données contenant l'ensemble des composants (rôles et implémentations). Ces composants sont liés entre eux via des relations de création (implémente et raffine) et de parcours (occurrence). Le framework d'annotation permet d'annoter chacun de ses composants en assurant la cohérence des annotations. Par exemple un rôle, dont l'annotation spécifie qu'il doit être écrit en Java, ne pourra avoir que des implémentations écrites en Java.

En étant basé sur les concepts de relations et d'entités composant, le framework devient indépendant du modèle de composant présenté dans la section précédente. En effet, il est possible d'ajouter de nouvelles relations de création et de parcours (voir de supprimer celles de notre modèle) et les entités annotées n'ont comme contrainte que d'avoir des relations entre elles.

Nous allons maintenant décrire comment, une fois les composants de la base de données annotés, nous réalisons la sélection, c'est à dire comment nous parcourons et sélectionnons les éléments de la base de développement.

5. La construction des configurations

5.1. Principes

Le framework décrit précédemment permet de qualifier les éléments de la base de développement en assurant le calcul et la vérification de cohérence des annotations.

La construction d'une configuration d'une application, basée sur un modèle de composant, consiste à sélectionner un ensemble de composants satisfaisant à une configuration souhaitée, exprimée sous la forme d'une requête. L'ensemble de ces composants doit bien sur constituer une configuration valide, c'est à dire capable de fonctionner.

5.2. La requête

Une configuration correspond à un ensemble de fonctionnalités souhaitées et de contraintes. Ces informations peuvent provenir de plusieurs sources :

- de l'administrateur de la future cible,
- des contraintes associées à la cible (elles seront détaillées dans le chapitre suivant avec la description du modèle de site),

- des stratégies de déploiement associées à l'entreprise contenant la cible…

Nous avons choisi d'exprimer ces contraintes ou souhaits de sélection à l'aide d'une requête exprimée en terme de couples attribut/valeurs et de contraintes de sélection.

Les attributs de la requête correspondent à ceux de notre système d'annotation. De façon simple, on peut dire qu'un composant peut être sélectionné si les valeurs de ses attributs sont en accord avec ceux de la requête.

5.3. La qualification des attributs

Afin de simplifier le parcours de la base de données, nous avons rajouté aux attributs le concept de portée. C'est à dire que nous avons de façon explicite qualifié la portée des valeurs des attributs.

Nous avons identifié trois niveaux de portée d'attribut :

- local,
- gelé,
- contextuel.

Nous allons maintenant décrire chacun de ses niveaux ainsi que leurs conséquences sur la propagation de la valeur de l'attribut ainsi qualifié.

5.3.1. Local

La valeur de l'attribut n'est valable que pour le composant ainsi annoté. Les éventuels composants fils pourront avoir des valeurs différentes pour le même attribut. Un attribut donnant la date de création du composant est un bon exemple d'une telle portée, en effet chaque composant a sa propre date de création

5.3.2. Gelé

Si la valeur d'un attribut est gelée, cela signifie que tous les fils (et descendants) du composant annoté auront la même valeur pour le même attribut. Par exemple, un composant écrit en Java aura tous ses fils écrit en Java (contrainte) si la valeur est gelée au niveau du composant.

5.3.3. Contextuel

La dernière qualification correspond à une valeur dite contextuelle. C'est à dire que la valeur des fils dépendra des logiques de propagation associées à l'attribut. Cette qualification est différente de celle de local, car elle impose des contraintes sur la façon de calculer la valeur des attributs fils (les stratégies de propagation) alors que celle de local n'impose rien du tout ni sur le calcul ni sur les vérifications.

5.3.4. Intérêt

Qualifier la portée des attributs apporte des informations supplémentaires pour la sélection. En effet, si, par exemple, lors du parcours pour sélectionner une configuration Linux on

étudie un composant dont l'annotation spécifie qu'il ne fonctionne que sous Windows et que cette valeur est gelée, alors il ne sert à rien de continuer à explorer cette branche car aucun des composants la constituant ne pourra être sélectionné. Ainsi nous avons ajouté, au niveau des annotations, des informations sur les annotations des éventuels composants fils.

5.4. Les contraintes de sélection

Notre système d'annotation ne permet pas d'exprimer toute les informations sur l'architecture d'une application. Nous avons introduit un langage d'expression de contrainte de sélection. L'exemple suivant permet de montrer le mécanisme mis en place pour exprimer et traiter les contraintes de sélection.

Supposons que l'on s'aperçoive après le développement d'un composant, qu'il ne fonctionne pas (lui et ses descendants) dans des conditions particulières (non testées lors du développement). Le but des contraintes de sélection est d'exprimer ce constat afin que le composant ne soit pas sélectionné dans une configuration recréant ces conditions particulières. On écrit donc :

Si <conditions> EXCLUDE

On ne peut pas directement exclure un composant car on ne peut pas savoir exactement le nom de tous les composants à exclure. Par contre on connaît les valeurs de certains attributs que l'on trouve dans les annotations des composants à exclure. Par exemple, on peut vouloir exclure tous les composants (issus du composant sur lequel la règle est ajoutée) dont l'annotation comporte la valeur 128 pour l'attribut mémoire, ce qui donne :

Si "Memoire==128" EXCLUDE

Dans la section 5.6 nous donnons un exemple plus explicite d'une contrainte de sélection. On peut imaginer d'autres type de contraintes comme l'inclusion par exemple.

Ces contraintes s'ajoutent sur un composant et s'appliquent sur le composant ainsi que tous ces descendants. Nous avons aussi ajouté le concept de contrainte globale, c'est à dire une contrainte qui s'applique à tous les composants de la base.

5.5. L'algorithme de sélection

5.5.1. Les hypothèses

Nous présentons ici l'algorithme utilisant les relations d'implémentation, de raffinement et d'occurrence.

Tout d'abord l'algorithme de sélection a pour rôle de parcourir la base de données en suivant les différentes relations (implémentation, raffinement et occurrence). Le but de ce parcours est d'élaguer les branches non conformes à la requête initiale. De cette façon nous obtenons à la fin du procédé de sélection, un ensemble de composants et leurs relations à partir desquels nous sommes capables de créer au moins une version de l'application.

Pour réaliser cet objectif, nous avons fait plusieurs hypothèses. Tout d'abord nous ne construisons que des versions exécutables, c'est à dire qu'à la fin du procédé de sélection,

chaque rôle doit au moins être relié à une implémentation, sinon le rôle ne sera pas retenu lors de la sélection.

Pour simplifier l'algorithme, nous avons décidé que le rejet d'un composant implique que tous ses descendants seront rejetés de la sélection. Par exemple, le rejet d'un rôle implique le rejet de toutes ses implémentations.

Et enfin, nous nous sommes donnés la possibilité lors de la sélection d'essayer de construire, pour les états intermédiaires (composite non défini) non raffinés, des versions définies. Autrement dit, nous avons autorisé la construction (indirecte) d'implémentations raffinant un état intermédiaire qui n'a pas déjà été raffiné.

5.5.2. L'algorithme

Le but de cet algorithme est de parcourir la base de données. Pour cela nous allons nous appuyer sur les relations liant les composants entre eux. Chaque type de composant dispose de sa propre logique de parcours. Par exemple, un rôle est lié avec ses descendants (ses implémentations) par la relation d'implémentation. Une implémentation composite dispose elle de deux relations possibles : celle de raffinement et celle d'occurrence.

Il faut noter qu'à chaque fois que l'on précise que l'on compare la requête à une annotation, on applique au préalable les contraintes de sélection décrites précédemment

Nous appliquons les règles suivantes dans notre algorithme de sélection selon le type de composant.

5.5.2.1. Le cas des implémentations natives

Il s'agit du cas le plus simple. On dispose de la requête initiale et on doit la comparer à l'annotation de l'implémentation native. Si elle est cohérente alors le composant est ajouté, sinon (et aussi si le composant n'est pas annoté) le composant est exclu de la sélection.

5.5.2.2. Le cas des implémentations composites

Tout d'abord, on vérifie que le composite est annoté et que son annotation est cohérente avec la requête initiale. Dans le cas négatif, on applique l'une des hypothèse de départ, qui dit que le composite ne sera pas sélectionné ainsi que ses descendants éventuels. Dans le cas positif, nous avons distingué deux cas de figure selon que le composite a été raffiné ou pas.

Si le composite ne possède pas de raffinements, on doit analyser l'arbre correspondant à chacune des occurrences. Si l'une au moins n'est pas cohérente, le composite ne sera pas sélectionné. Pour chacune des occurrences, on applique l'algorithme de sélection correspondant à son type (rôle, composite ou natif).

Si le composite est raffiné, alors on passe directement à l'analyse des implémentations le raffinant (en leur appliquant le même algorithme). Le composite sera sélectionné uniquement si l'une au moins de ses implémentations (le raffinant) est sélectionnée.

5.5.2.3. Le cas des rôles

Comme précédemment, on vérifie d'abord que le rôle est annoté et que son annotation est cohérente. Si l'analyse est négative, le rôle n'est pas sélectionné. Sinon, nous avons encore distingué deux cas possibles : le rôle a des implémentations ou n'en a pas. Dans ce deuxième cas, le rôle n'est pas sélectionné (voir les hypothèses). Dans le cas où il existe des implémentations, le rôle ne sera sélectionné que si l'une au moins de ses implémentations est sélectionné (voir les cas précédents des implémentations composite ou native).

5.6. Exemple d'une configuration

Essayons de réaliser une configuration à partir de notre exemple. On commence par créer une requête. Pour simplifier nous présenterons la requête sous la forme d'une annotation. Nous avons choisi de sélectionner une configuration fonctionnant sous linux, écrite en Java et disponible avec la distribution Redhat de linux. Ce qui donne la requête suivante (Figure 27):

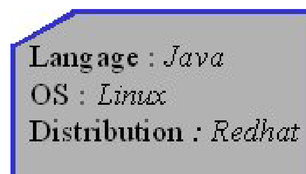

Figure 27 : exemple de requête

Nous avons aussi ajouté un attribut (Distribution) à certaines de nos annotations. On dispose maintenant de composants annotés (Figure 28).

Pour illustrer la portée des attributs, nous avons qualifié de "**gelé**" l'attribut Langage de l'annotation de *A1*. De cette façon, nous sommes sûrs lors de la sélection de ne trouver que des composants écrits en Java à partir du composant *A1*. Par contre l'attribut OS de la même annotation est qualifié de "contextuel", c'est à dire que la valeur de l'attribut OS de l'annotation de *A1.1* dépendra des stratégies associés.

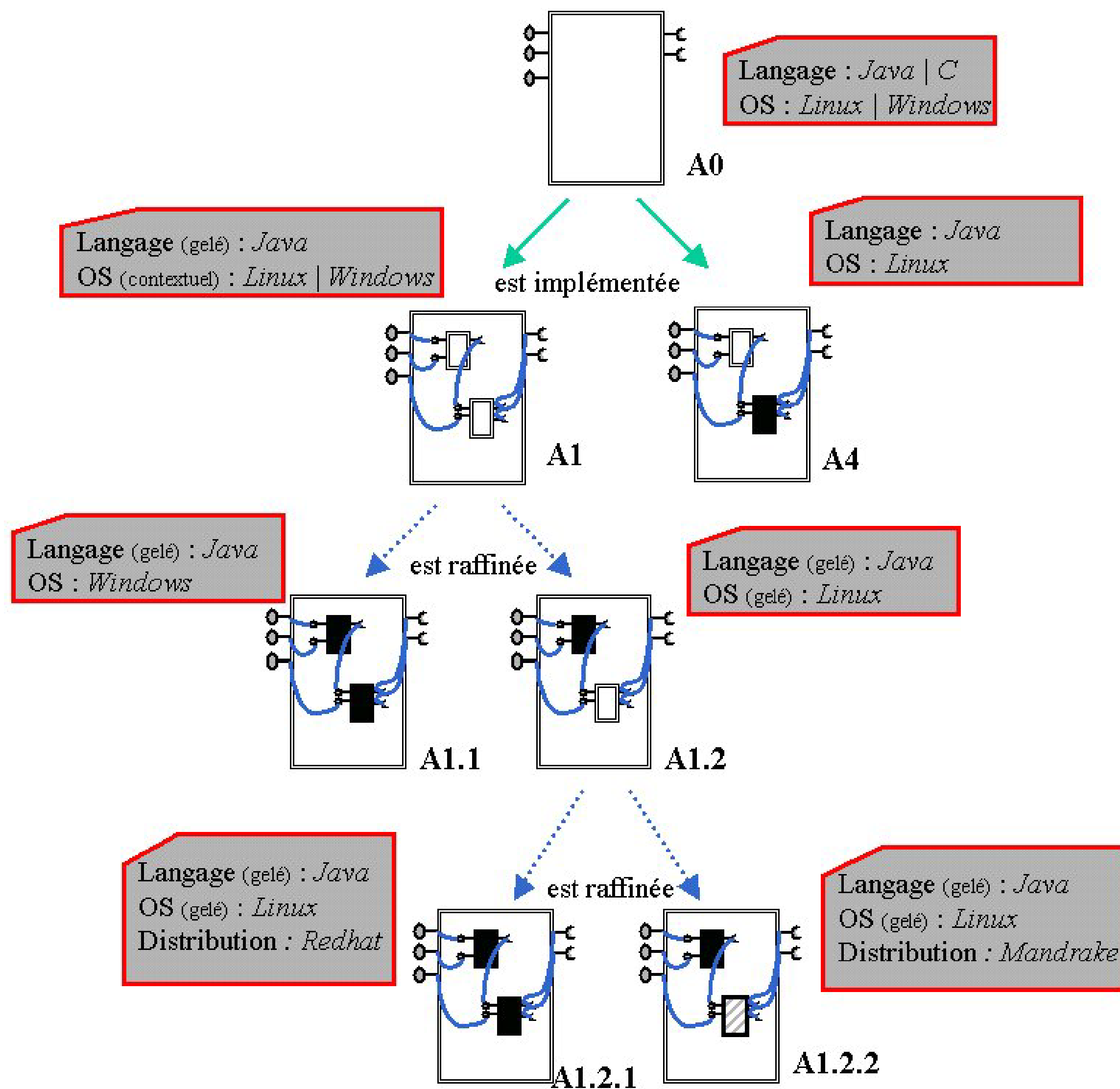

Figure 28 : Exemple d'annotations et de contraintes de sélection

Nous allons maintenant donner un exemple d'une contrainte de sélection. Supposons que l'on sache que le composant *A4* ne fonctionnera pas avec des occurrences dont l'annotation comporte l'attribut mémoire avec une valeur <256 Mo et ceci dans le cas d'une configuration fonctionnant sous le processeur Intel.

Il faut rappeler que lors de la sélection, on peut par exemple si A4 est compatible chercher à le décrire entièrement, c'est à dire trouver une implémentation (de l'occurrence de rôle *D0* dans *A4*) qui sera cohérente avec notre requête. On dispose par exemple des annotations suivantes (Figure 29).

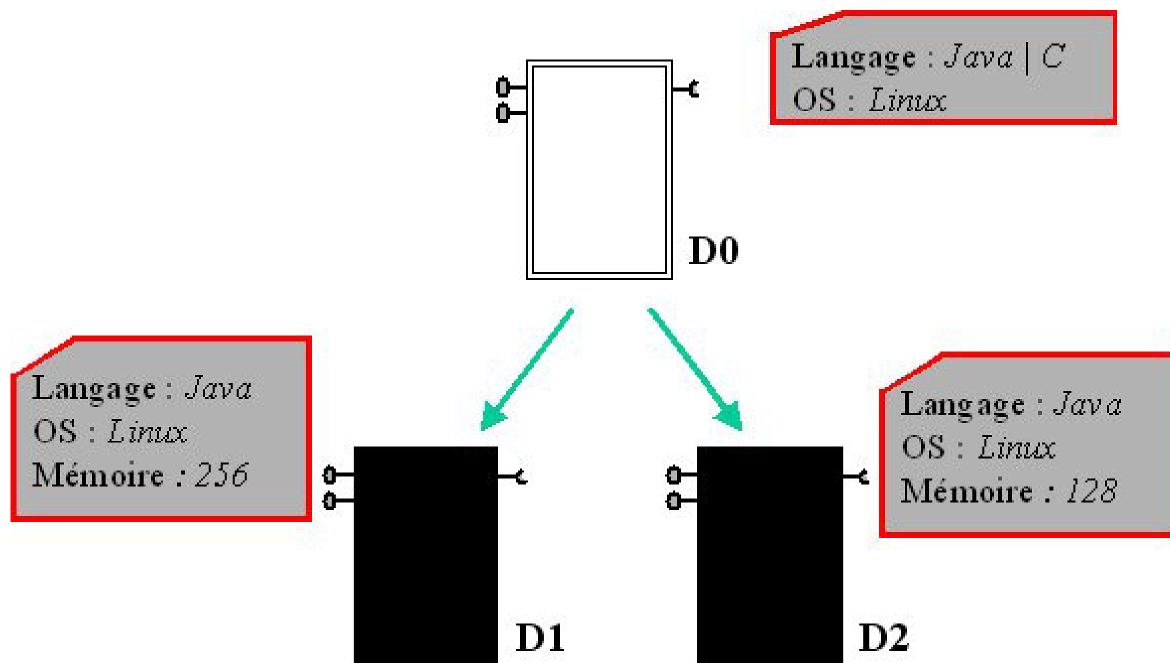

Figure 29 : Exemple d'annotations

Nous avons complété notre requête, en lui ajoutant une contrainte sur le processeur qui devra être Intel. Sans une contrainte de sélection sur A4, nous pouvons avoir une configuration avec comme raffinement de *A4* une occurrence de *D2* qui ne pourra pas fonctionner car la configuration faite pour Intel demande au moins 256 pour Mémoire.

Nous avons donc rajouté sur A4, la contrainte exprimée de la façon suivante :

Si "processeur == Intel et Mémoire < 256" EXCLUDE

Elle signifie que si la requête possède l'attribut processeur initialisé avec la valeur Intel alors on doit exclure tous les composants (parcourus à partir de A4) dont l'annotation comporte l'attribut Mémoire avec une valeur inférieure à 256. Dans notre exemple, *D2* sera exclu de la sélection alors que *D1* ne le sera pas.

Nous pouvons maintenant lancer la sélection. En respectant l'algorithme de sélection de la section 5.5 nous obtenons le résultat, dont une visualisation graphique du parcours est la suivante : Figure 30 et Figure 31.

Dans l'étape 1, on met *A0* à "sélectionnable", car son annotation est compatible avec la requête, il deviendra "sélectionné" si l'une de ses implémentations est "sélectionnée". On va ensuite examiner ses implémentations *A1* puis *A4*. A1 est sélectionnable (étape 2), on va donc examiner ses fils (*A1.1* et *A1.2*). Comme *A1* est raffinée, on applique l'hypothèse selon laquelle on ne cherchera pas à trouver des implémentations pour les rôles constituant *A1*. Si ni *A1.1* et *A1.2* ne sont sélectionné alors *A1* ne le sera pas même si on aurait pu trouver nous même un raffinement permettant de le sélectionner.

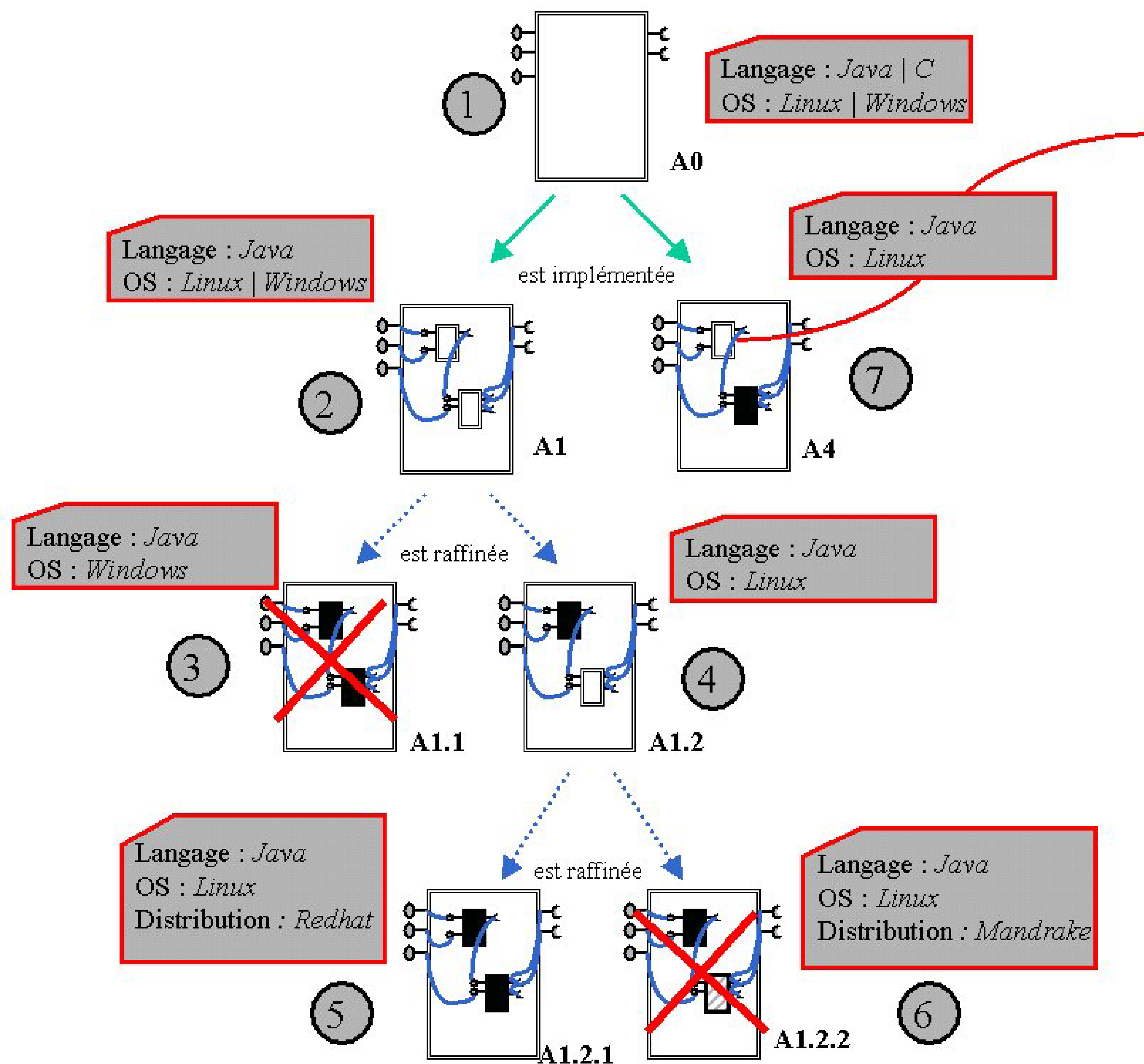

Figure 30 : Visualisation du parcours de sélection (partie 1)

A1.1 est non sélectionnable (étape 3), car son annotation comporte la valeur Windows pour l'attribut OS alors que l'on souhaite avoir des composants fonctionnant sous Linux. On passe à *A1.2* (étape 4), elle est sélectionnable. Parmi ses raffinements seul *A1.2.1* est "sélectionné" (étape 5) alors que *A1.2.2* ne l'est pas (étape 6). Par conséquent *A1.2* devient "sélectionné", de même pour *A1* et *A0*. On passe maintenant à A4. Il est sélectionnable (étape 7) mais il n'est pas exécutable et il n'est pas raffiné. Nous allons donc essayer de le rendre sélectionné en cherchant parmi les implémentations de l'occurrence de rôle qui le compose.

On passe maintenant à la Figure 31, *D0* correspond au rôle dont on cherche à trouver une implémentation. Elle est sélectionnée d'office (ceci est rendu possible par le fait que notre base de composants est cohérente). N'oublions pas la contrainte de sélection que nous avons ajoutée à *A4*. Elle rend impossible la sélection d'un composant dont l'annotation comporte l'attribut mémoire avec une valeur inférieure à 256, règle active si dans la requête on a imposer d'avoir un processeur Intel. C'est le cas de notre exemple. *D1* sera sélectionné (étape 8) et par conséquent *A4* le sera aussi. Par contre *D2* ne le sera pas (étape 9) à cause de la contrainte de sélection sur *A4*.

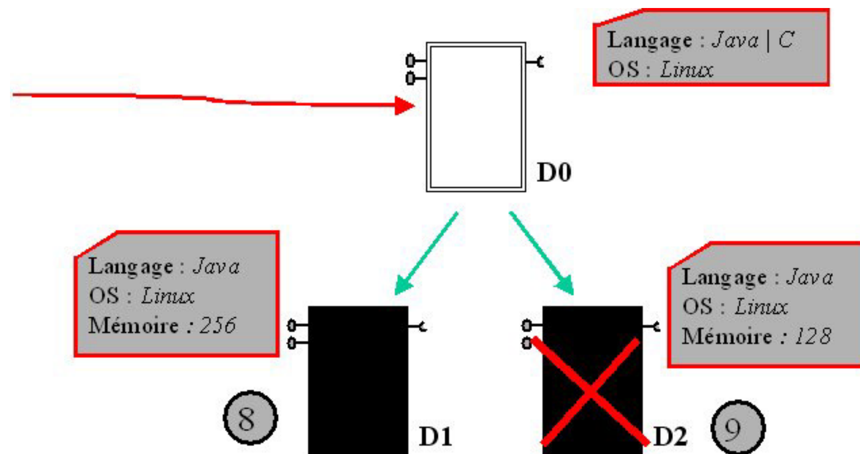

Figure 31 : Visualisation du parcours de sélection (partie 2)

Le résultat de notre sélection sera donc la sélection des composants suivants : {A0, A1, A1.2, A1.2.1, A4, Do, D1}.

6. Le résultat

6.1. Principes

Le résultat de l'activité de sélection correspond à une application à base de composants. Cette application a ensuite vocation à être installée sur des sites. Pour cela, nous devons expliciter la façon de décrire l'application (information nécessaire pour choisir et vérifier que l'application fonctionnera sur la machine cible). Comme le reste du cycle de vie du déploiement concerne tous les types d'applications, nous avons défini un modèle d'application, qui sera utilisé par toutes les autres activités du cycle de vie du déploiement. Il s'agit du modèle commun d'application utilisé par notre environnement de déploiement (ORYA). Ce modèle est très générique et notre modèle de composant correspond au type d'application décrite par ce nouveau modèle.

Lors des différentes activités de déploiement, nous avons besoin d'avoir une description de l'application (en termes de dépendances, de fonctionnalités, etc.). C'est pourquoi nous avons défini un descripteur d'application (appelé manifeste).

Enfin, les applications ont besoin d'être packagées afin d'être utilisées lors des différentes étapes du déploiement (en particulier la phase de transfert de l'installation).

6.2. Le modèle d'application

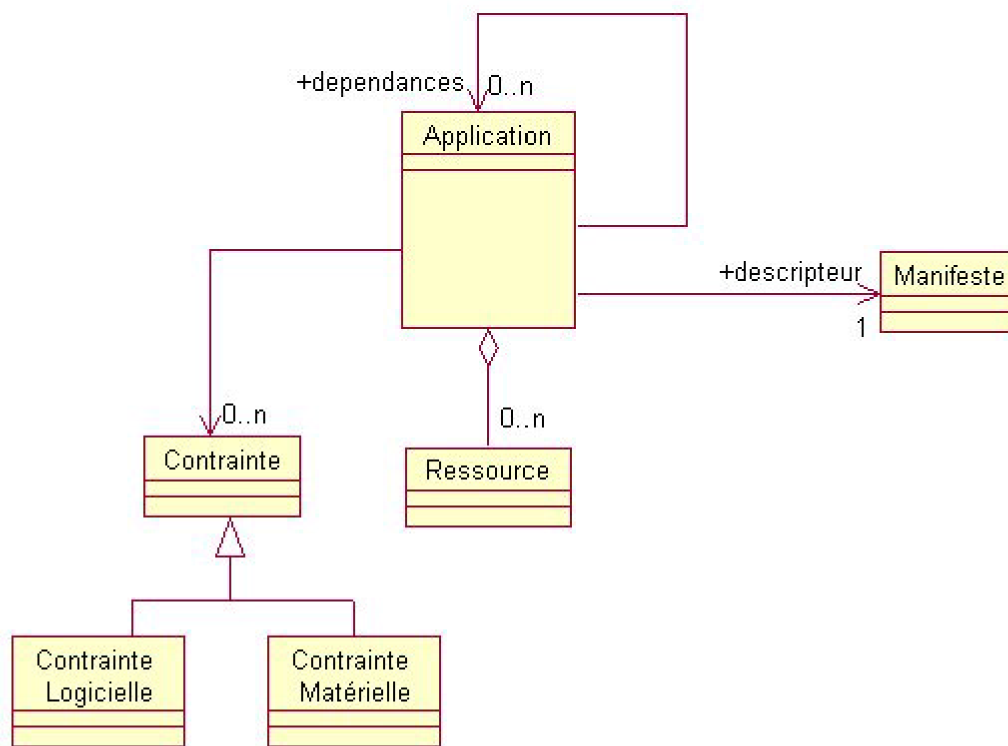

Figure 32 : Modèle d'application

Ce modèle d'application (Figure 32) est volontairement très générique et simplifié afin de prendre en compte tous les types d'applications existants [Les00]. Il ne contient aucune information sur l'architecture de l'application. Une application est considérée comme un ensemble de ressources ayant éventuellement des dépendances envers d'autres applications. Du point de vue du déploiement, les informations importantes sont : la liste des dépendances de l'application, la liste des ressources à installer et enfin les contraintes que le site doit satisfaire. On distingue généralement deux types de contraintes : les contraintes logicielles et les contraintes matérielles. Les contraintes logicielles concernent les applications incompatibles avec le bon fonctionnement de l'application et les contraintes matérielles représentent la liste des matériels (espace mémoire, disque, présence d'une carte vidéo…) que doit satisfaire le site cible.

6.3. Le descripteur d'application

Toutes ces informations (dépendances, ressources, contraintes) plus celles concernant l'application elle-même (version, nom producteur, etc.) sont regroupées dans un descripteur appelé manifeste. Ce manifeste sera utilisé lors de différentes étapes du déploiement (comme la recherche du "package" le plus cohérent, la résolution des dépendances, l'installation…).

6.4. Les "packages"

Un "package" (par exemple un fichier zip) doit contenir l'ensemble des ressources (ou du moins les informations nécessaire pour accéder à ces ressources) et le manifeste décrivant l'application ainsi packagée.

7. Résumé

L'activité de sélection a pour objectif de fournir les "packages" à l'environnement de déploiement qui ensuite les déploiera sur les diverses machines clientes.

Nous avons dans ce chapitre présenté un framework d'annotation permettant de décrire (du point de vue du déploiement) des composants. Ces descriptions respectent des règles de cohérences établies selon le type de relations liant un composant à un autre. De cette façon, nous disposons d'une base de composants à partir desquels nous pouvons "construire" des versions d'applications. Pour cela, il suffit de décrire les fonctionnalités et les contraintes que la version doit fournir ou satisfaire.

L'intérêt d'utiliser un tel framework, est d'être indépendant d'un modèle de composant particulier : les concepts d'entité (ou de composant) et de relations sont suffisamment de haut niveau pour pouvoir être appliqués à la plupart des modèles de composant existants.

Chapitre VI

L'installation

1. Introduction

Dans la chapitre précédent, nous avons étudié l'activité de sélection et de configuration du cycle de vie du déploiement. Nous avons terminé en décrivant le résultat de cette activité, c'est à dire les "packages". Nous allons maintenant, dans ce chapitre, montrer comment nous décrivons l'installation de ces "packages" sur les machines cibles du déploiement.

L'activité d'installation, comme son nom l'indique, a pour objectif de réaliser l'installation d'une application sur une ou plusieurs machines. On parle d'installation unitaire lorsqu'une seule machine est concernée et d'installation multiple en cas de plusieurs machines. Dans un but d'automatisation, l'installation doit pouvoir être lancée et exécutée à distance et sans aucune intervention à partir de la machine cible du déploiement.

Intéressons nous pour l'instant au cas des installations unitaires. L'un des objectifs d'ORYA en matière de déploiement consiste à installer sur les machines les versions les plus cohérentes. Cohérence avec les caractéristiques matérielles de la machine (type de carte vidéo, espace disque disponible…), avec les logiciels déjà déployés (pour éviter de déployer deux applications incompatibles sur la même machine par exemple) ou avec le profil et les souhaits du (ou des) utilisateur(s) de la machine. De plus, il faut lors de l'installation tenir compte des politiques de déploiement de l'entreprise, ainsi que celles associées à la machine cible.

L'activité d'installation est plus complexe que le "simple" fait d'exécuter un script d'installation du type "InstallShield" [InstallShield]. Une installation doit tout d'abord trouver le "package" le plus cohérent (voir lancer le processus de sélection pour le créer), éventuellement analyser l'installation avant de la lancer (analyse des dépendances par exemple), résoudre les dépendances, transférer le "package" sur la machine cible et enfin réaliser le déploiement proprement dit. Cette description n'est qu'une façon de faire parmi d'autres.

Ce chapitre est structuré de la façon suivante : dans la section 2, nous décrivons le problème et ses solutions, dans la section 3 et 4, nous présentons les différents mondes intervenant dans la solution choisie, puis dans la section 5, nous discutons de la réalisation de la solution, ensuite de la modification de procédé, puis nous terminons par quelles techniques de création et de modification de procédé.

2. Le problème de l'installation

2.1. Principe

De manière générale nous pouvons dire qu'un problème se situe dans un monde, et qu'une solution décrit la façon de changer l'état de ce monde pour arriver à une situation où le problème a été résolu [Vil03].

Appliquons cette définition au problème de l'installation dans le cycle de vie du déploiement. Le monde correspond à l'environnement de déploiement, plus précisément à l'ensemble des machines, des "packages", des applications, etc..

L’état initial (avant l’installation) correspond à l’état de chacun des éléments de notre monde (le nombre de "packages" disponibles sur un serveur d’applications, etc.). De façon très simplifiée, on peut dire que le changement d’état dû à l’installation correspond à un changement d’état de la machine cible (au moins une nouvelle application lui a été ajoutée).

A chaque installation, le problème reste le même mais la solution peut être différente. En effet, on peut choisir de réaliser par exemple le déploiement en mode pull ou bien en mode push, dans ce cas les solutions sont différentes.

2.2. Les solutions au niveau réel

Le niveau réel correspond à la réalité "physique" de notre environnement, on y parle de machines physiques, de connections, de routeurs, etc.…

Les solutions du problème doivent s’exprimer en termes de réseaux, de transferts de fichiers, d’introspection de machines, etc.. Pour réaliser l’installation, nous disposons d’un ensemble d’outils dédiés ou non au déploiement, comme l’outil de transfert ou celui d’introspection.

Chacun de ces outils est capable de modifier partiellement l’état du monde. Les solutions dans le monde réel consistent donc à exécuter de façon ordonnée un ensemble défini d’outils. Cet ordonnancement pouvant être décrit comme un programme.

Décrire directement les solutions dans le monde réel a comme inconvénient de nécessiter une connaissance pointue du fonctionnement de chacun des outils. Il faut aussi écrire une solution pour chaque installation et la réutilisation des précédentes solutions (écrites pour d’autres installations) est très difficile. Et enfin, les solutions sont dépendantes de l’évolution des outils.

Pour résumer, on peut dire qu’exprimer les solutions directement dans le monde réel rend difficile l’évolution et la réutilisation, ce qui est en contradiction avec les objectifs de notre approche.

2.3. Les solutions au niveau modèle

Le niveau modèle correspond à une abstraction du niveau précédent. En d’autres termes, nous avons abstrait l’environnement de déploiement et analysé ce que signifie une solution dans ce niveau et quels en sont les avantages.

On a vu que les solutions exprimées dans le niveau réel étaient difficilement réutilisables et leurs évolutions difficiles. De plus une connaissance approfondie des outils était nécessaire.

L’avantage d’abstraire le monde (l’environnement de déploiement) est que l’on peut ainsi abstraire le problème et ses solutions. L’abstractions des outils est aussi intéressante car cela permet de résoudre le problème de la connaissance approfondie des outils. Dans le reste de ce document, nous appellerons un "rôle" l’abstraction d’un outil. Un rôle correspond sommairement à un ensemble de fonctionnalités offertes par l’outil. Par exemple le rôle de l’outil de transfert peut se résumer à deux fonctionnalités : une de récupération d’un fichier et une autre d’envoi d’un fichier. Il est intéressant de noter qu’un rôle peut être joué par plusieurs outils, ce qui augmente les possibilités de déploiement.

Dans le monde réel la même installation entraîne l'écriture de plusieurs solutions : une par machine où sera installée l'application. En effet, chaque différence entre les machines peut impliquer des solutions différentes (par exemple, la présence ou non d'un outil particulier peut entraîner des solutions radicalement différentes).

Comment éviter de devoir écrire une solution pour chaque installation ? Tout d'abord, on peut décrire les solutions en termes d'éléments de l'abstraction de notre monde et en termes de rôles. Ce qui réduit fortement le nombre de solutions à écrire : la même solution pouvant être utilisée pour installer la même application sur plusieurs machines. Bien sur il n'existe pas une solution universelle pour l'installation mais l'abstraction permet de réduire le nombre de cas.

La description d'une solution au niveau réel permet de décrire concrètement la manière dont l'installation sera réellement réalisée. La description au niveau modèle permet elle de décrire le *comment*.

Il existe de nombreuses façons pour décrire ces solutions (au niveau modèle) comme l'utilisation de règles ou de procédés. Nous avons choisi d'utiliser l'approche des procédés.

2.4. Le framework d'installation

Nous devons décrire comment une installation doit se dérouler, ce qui correspond à la description de la solution à l'aide d'un procédé. Afin d'abstraire (c'est à dire diminuer la dépendance envers le monde réel) nous avons modélisé l'environnement (section 3) et nous avons exprimé nos solutions à l'aide d'éléments du modèle de procédé (section 4).

Nous allons donc avoir deux mondes :

- le monde de l'environnement,
- le monde des procédés.

Le monde des procédés se caractérise par des activités qui s'enchaînent, des rôles qui réalisent ces activités et des produits, qui sont créés, manipulés, transmis et détruits.

Ces deux mondes partagent en partie le concept de produit car certains des produits manipulés par les procédés correspondent à des éléments de l'environnement. Cependant leur vision (notamment du cycle de vie) des produits n'est pas identique et il faudra les réconcilier. Par exemple, la destruction d'un produit de type machine dans le monde des procédés à la fin d'une activité car il n'est plus utilisé ne correspond pas à sa disparition dans le monde de l'environnement que ce soit au niveau modèle ou au niveau réel.

Essayons de schématiser notre approche. Au niveau modèle nous avons donc le modèle d'environnement et celui des procédés. De plus nous avons aussi celui des produits (produits utilisés par les procédés). Le modèle de procédé a comme objectif de modifier l'état du modèle d'environnement via des rôles. Ce qui donne la schématisation suivante :

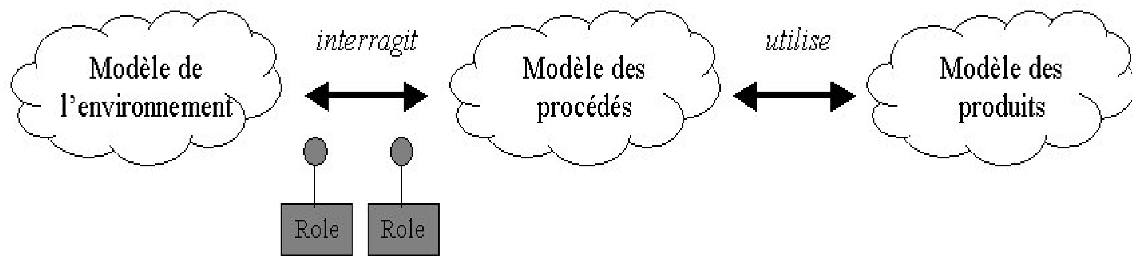

Figure 33 : le niveau modèle

Chaque installation d'une application correspond à plusieurs solutions (une en mode push et une autre en mode pull par exemple). Une solution correspond à un modèle décrivant l'utilisation des rôles. A chaque fois que l'on installe une application, cela correspond à une instanciation du modèle de procédé en un procédé manipulant lui des outils. Cela donne le schéma suivant :

Figure 34 : le niveau du monde réel

Etant donné nos objectifs de réutilisation et de cohérence nous sommes conduits à modéliser le niveau modèle en utilisant des méta-modèles : celui de l'environnement, des procédés et des produits.

Un modèle (de procédé, d'environnement ou de produit) est une instance du méta-modèle correspondant. En pratique comme le montre les schémas précédents le modèle de procédé contient/utilise le concept de produit et de type de produit. Par conséquent dans notre approche, pour des raisons de simplification, le méta-modèle de procédé contient le méta-modèle des produits.

Nous disposons donc d'un framework d'installation avec lequel nous sommes capables de décrire les différentes installations au niveau modèle en respectant une certaine cohérence entre les différents modèles de procédés grâce au niveau méta-modèle. Ces modèles seront ensuite instanciés pour donner un procédé spécifique à l'installation d'une application donnée, pour une machine donnée et en respectant des politiques de déploiement données.

Si l'on essaye de représenter ces différents niveaux et en tenant compte du fait que le monde des procédés contient celui des produits, l'architecture de notre framework est la suivante :

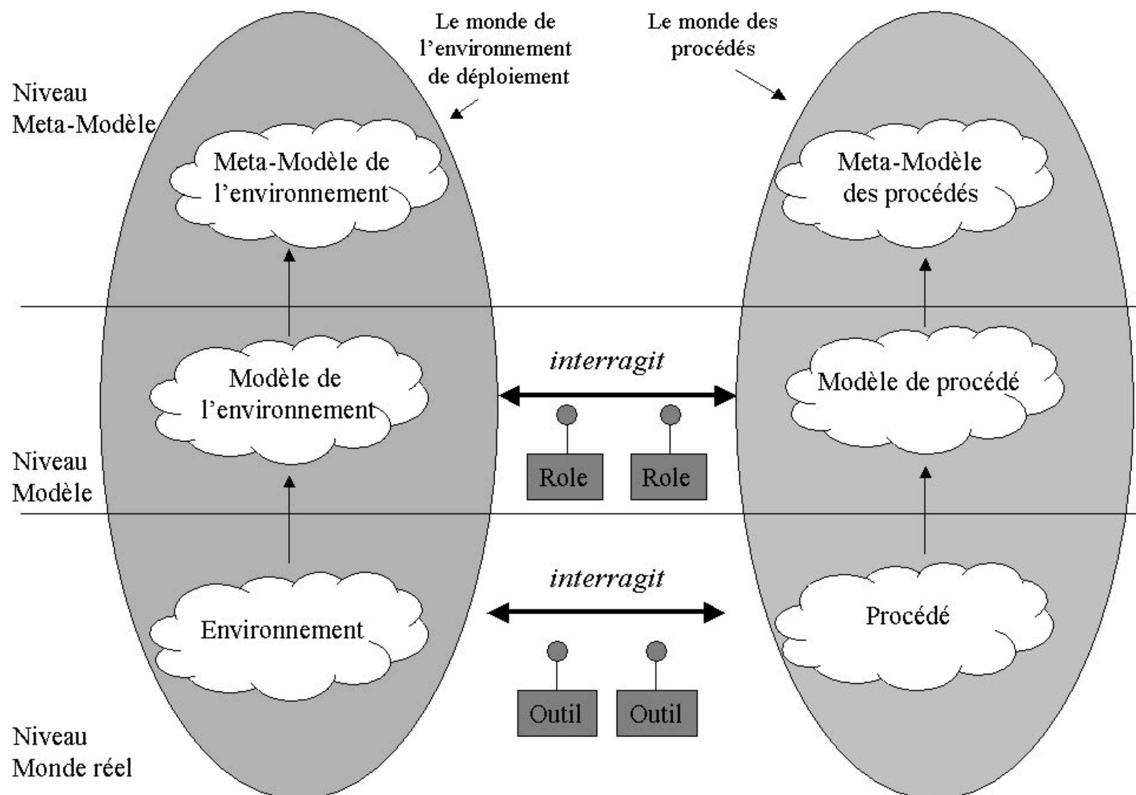

Figure 35 : Architecture du framework

Pour installer une application sur une machine donnée, notre framework propose les fonctionnalités suivantes :

- la création de modèles de procédés permettant de décrire les étapes communes de chaque installation de l'application,
- la création du procédé spécifique à cette installation. Cette instanciation du modèle précédemment créé tient compte des spécificité (par rapport au déploiement) de la machine.

Nous avons aussi ajouté au niveau du framework la possibilité d'intervenir en cas de problèmes lors du déploiement (par exemple pour modifier "dynamiquement" le procédé en cours).

Nous allons maintenant décrire plus en détail les différents mondes, puis le langage de réalisation et d'évolution qui permet de réaliser réellement l'installation.

3. Le monde de l'environnement de déploiement

3.1. Intérêt

L'installation d'un logiciel est une activité complexe faisant intervenir de nombreux acteurs comme les sites ou les entreprises. Ils interviennent à plusieurs endroits du procédé d'installation comme :

- lors du choix de la version de l'application,
- lors de la résolution des dépendances logicielles,
- pour appliquer les politiques de déploiement.

Le procédé d'installation nécessite de connaître des informations sur les sites, les applications, les entreprises et les politiques de déploiement. Modéliser ces acteurs permet de définir une abstraction qui sera utilisée par les modèles de procédés de déploiement. Nous allons maintenant décrire du point de vue du déploiement (plus précisément de l'installation) les différentes entités de cet environnement.

3.2. Le méta-modèle de l'environnement

Tout d'abord, voici une version très simplifiée du méta-modèle de l'environnement :

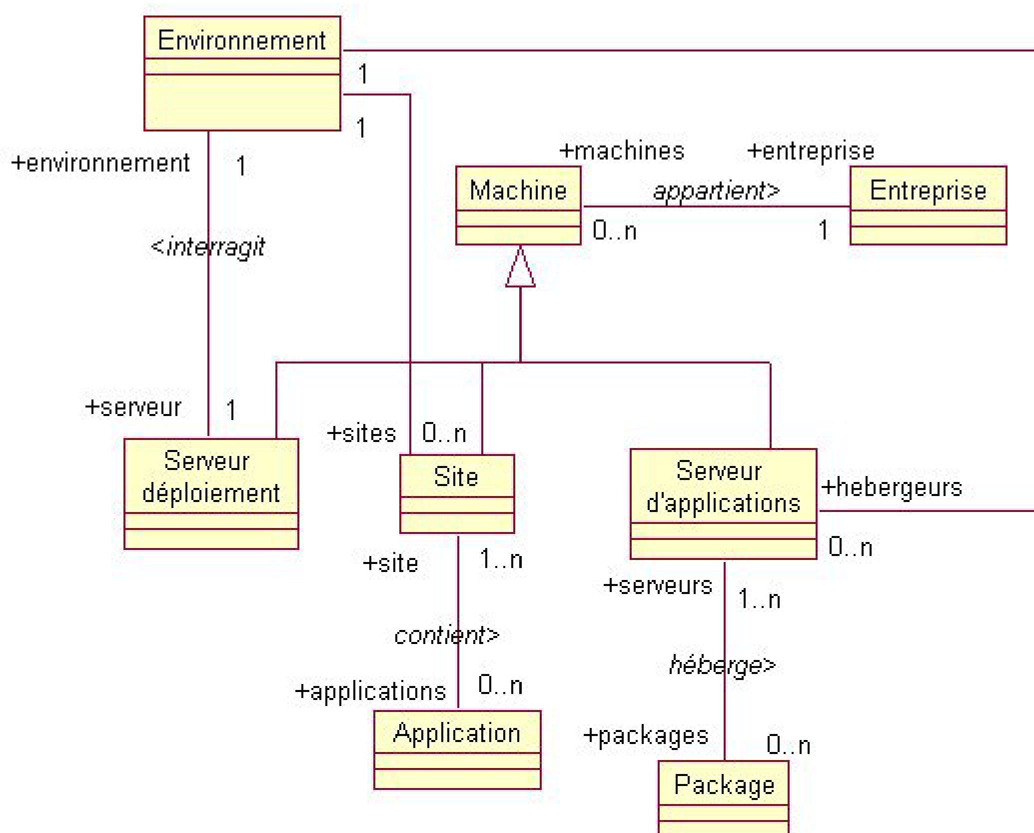

Figure 36 : Le méta-modèle de l'environnement de déploiement

Une installation unitaire (un seul "package" sur un unique site) consiste à récupérer le "package" sur un serveur d'applications puis de la transférer vers la cible du déploiement (le site) et ensuite de défaire le "package" pour lancer l'installation proprement dite. Nous allons maintenant détailler les différentes entités de ce méta-modèle en décrivant leur rôle vis à vis de l'installation et l'information qu'ils doivent posséder. Nous nous intéresseront plus particulièrement aux sites. Les applications et les "packages" ont déjà été étudiés dans le chapitre précédent (la sélection).

3.3. Les sites

3.3.1. La notion de site

Un modèle de site permet de décrire de façon abstraite les ressources et la configuration d'un site [Gom00]. Un tel modèle doit contenir des informations comme le nombre de disques et pour chacun leur capacité, comme le type de système d'opération, ou la liste des logiciels déjà installés sur le site.

Nous avons identifié plusieurs types d'informations susceptibles d'être utilisées lors du déploiement :

- les caractéristiques matérielles,
- les caractéristiques logicielles,
- la description des utilisateurs du site.

De cette façon, un procédé d'installation pourra s'appuyer sur ces informations. Nous allons dans les paragraphes suivants reprendre chacune de ses données afin de définir le modèle de site.

3.3.2. La description matérielle

Nous distinguons deux types d'utilisation de la description matérielle d'un site lors du déploiement (et plus particulièrement lors de l'activité d'installation). Tout d'abord, elle est utilisée pour vérifier qu'une application peut être installée sur un site. En effet, toute application nécessite une configuration matérielle minimale pour pouvoir s'installer ou s'exécuter sur un site. Cette configuration sera comparée avec celle du site pour valider le déploiement. Cette utilisation de la description matérielle apparaît lorsque l'environnement de déploiement veut déployer une application particulière sur un site (on parle alors de mode *push*). Par contre, on peut être dans une politique où le choix de la configuration de l'application est établi (entre autre) à partir de la configuration du site. La description matérielle sera donc utilisée soit pour établir la configuration souhaitée, soit pour valider la compatibilité d'une application avec un site.

La description matérielle d'un site correspond aux informations associées aux caractéristiques matérielles du site. Une carte vidéo ou la taille d'un disque sont des exemples de ce type d'information. Si une version d'une application nécessite une carte vidéo, il faut s'assurer avant le déploiement que la machine cible possède ce type de matériel. Ce genre d'information sera utilisé pour choisir la configuration la plus cohérente de chaque application à déployer sur la machine.

3.3.3. La description logicielle

La description logicielle est utilisée le plus souvent pour vérifier les contraintes logicielles associées à une application à déployer. Une contrainte logicielle peut prendre plusieurs formes :

- cas 1 : une application peut avoir des conflits avec d'autre applications, autrement dit, elles ne peuvent pas co-exister sur le même site. Dans ce cas, si l'application avec

laquelle il y a un conflit est déjà installée sur le site, l'application à déployer ne pourra pas (et ne devra pas) être installée,

- cas 2 : une application est dépendante envers d'autres applications. Dans ce cas, il existe plusieurs politiques de déploiement. On peut vouloir que les dépendances soient installées avant de déployer l'application ou bien installer l'application et résoudre les dépendances ensuite.

Dans le cadre du déploiement, il est nécessaire d'avoir une connaissance des applications déjà déployées. L'information associée à chaque application installée doit contenir diverses informations comme le nom, la version, les dépendances, etc.

On doit même connaître en détail chaque composant de l'application installée afin de pouvoir plus tard assurer la cohérence de l'application ou la mise à jour. Cette connaissance est aussi utile pour traiter le cas des composants ou librairies partagées entre plusieurs applications. Dans cette thèse nous traitons les applications à déployer comme des éléments atomiques, c'est à dire que nous ne nous intéressons pas au niveau des composants ou éléments de chaque application.

3.3.4. La description des utilisateurs

Un site peut avoir potentiellement un ou plusieurs utilisateurs. Certains sites n'ont, de par leur type, aucun utilisateur comme les automates par exemple. L'intérêt de nous intéresser à la description des utilisateurs est que cela nous permet d'ajouter un niveau supplémentaire dans le choix de la version la plus cohérente des applications à installer. En effet, la position d'un utilisateur dans une entreprise peut influer sur le choix de la version. Par exemple, un développeur n'aura pas les mêmes besoins qu'un chef de projet par exemple et lors de l'installation cette information pourra être utilisée pour configurer l'application à installer.

3.3.5. Le modèle du site

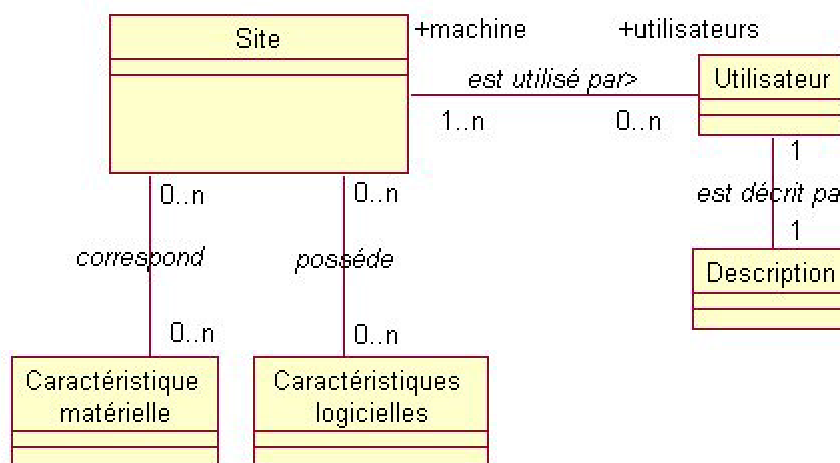

Figure 37 : Le modèle de site

3.4. Les serveurs d'applications

3.4.1. La notion de serveur d'application

Un serveur d'application a pour rôle de gérer le cycle de vie des différents "packages" qu'il héberge. Utiliser une abstraction nous permet d'utiliser dans ORYA plusieurs types de stockage de données (les "packages") au niveau du monde réel. On pourra avoir des bases de données, des gestionnaires de configuration ou un simple système de fichiers.

3.4.2. Gestionnaire de packages

De façon très simple, un serveur d'application au niveau modèle doit héberger des "packages". Il doit aussi fournir au serveur de déploiement la description de chacun des "packages" qu'il héberge.

3.4.3. Le modèle de serveur d'applications

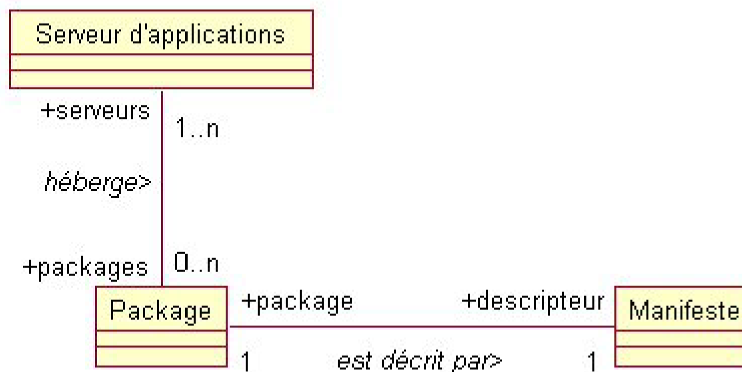

Figure 38 : Le modèle du serveur d'application

3.5. Le serveur de déploiement

3.5.1. La notion de serveur de déploiement

Le serveur de déploiement correspond au noyau de notre environnement de déploiement (ORYA). Il a principalement deux rôles : assurer la connaissance en temps réel de l'état de l'environnement et gérer les différents déploiements ayant cours dans l'environnement.

3.5.2. La connaissance de l'environnement

Pour réaliser les différentes étapes du déploiement, nous avons besoin de connaître à tout moment l'état de celui-ci. En effet, si par exemple, on planifie l'installation d'une application le soir, il faudra s'assurer que la machine est bien connectée au moment du déploiement. De même, on peut avoir besoin lors de l'installation d'installer des dépendances, il faudra donc connaître la liste des serveurs d'applications ainsi que leurs "packages". On voit bien que la connaissance de l'état de l'environnement est un aspect critique du déploiement.

Le serveur de déploiement doit donc "écouter" les événements issus de l'environnement (connexion, déconnexion d'une machine par exemple).

3.5.3. Gérer et piloter les différents déploiements

L'autre rôle du serveur de déploiement consiste à gérer les différents déploiements. Cela signifie plusieurs choses :

- préparer les installations,
- piloter chaque installation,
- appliquer les différentes politiques de déploiement.

L'un des rôles du serveur, c'est de gérer les relations entre les différentes installations en cours (gestion des ressources matérielles par exemple) et aussi de planifier les scénarios d'installation en cas de déploiement multiple (plusieurs cibles en même temps) ou de déploiement unitaire programmé.

Lorsque l'on décide d'installer une application sur une machine, le serveur de déploiement doit choisir (selon des politiques de déploiement) la version de l'application à installer. Il peut aussi analyser l'installation (examen de la résolution des dépendances par exemple) et il doit choisir le modèle de procédé à utiliser. Les politiques seront décrites dans la section 3.6.2.

Piloter une installation cela signifie que le serveur de déploiement dispose d'un moteur de procédés afin d'exécuter réellement l'installation. Il est important de préciser que le type de déploiement qui nous intéresse est en mode push donc toute la logique de l'installation est gérée par le serveur de déploiement et non par les cibles du déploiement.

3.5.4. Le modèle du serveur de déploiement

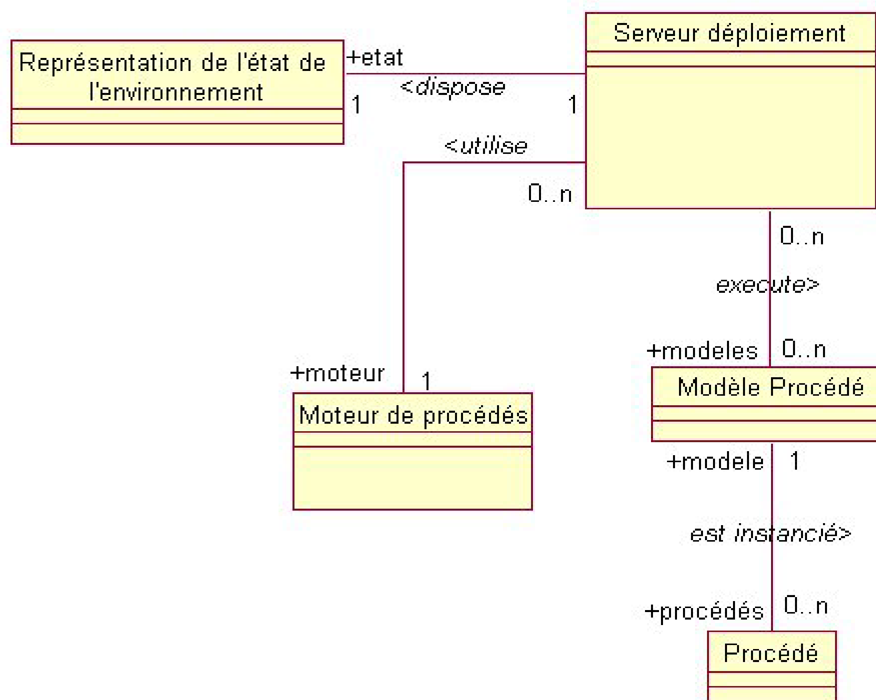

Figure 39 : Le modèle du serveur de déploiement

3.6. L'entreprise

3.6.1. La notion d'entreprise

On retrouve dans notre monde de l'environnement, le niveau entreprise présenté dans le chapitre 2 de cette thèse. L'intérêt d'un tel niveau pour l'activité d'installation est multiple :

- il permet d'appliquer des politiques globale de déploiement valables pour toutes les installations réalisées sur les machines de l'entreprise. Chaque machine ayant ses propres politiques de déploiement mais elles ne peuvent pas être contradictoires avec celles de l'entreprise,
- il permet aussi de fournir une connaissance globale de l'état de l'environnement (nombre de machines, listes des applications installées, etc.). Cette information est utilisée par le serveur de déploiement pour procéder aux diverses installations.

Nous allons maintenant décrire ce que nous entendons par politique de déploiement.

3.6.2. Les politiques de déploiement

Une politique de déploiement comme son nom l'indique sert à fixer les règles (ou des contraintes) dont l'on devra tenir compte lors de l'installation, mais aussi lors des autres activités de déploiement. Nous avons distingué plusieurs types de politiques dont voici quelques exemples :

- les politiques liées à la planification des taches de déploiement (par exemple, forcer le déploiement uniquement pendant la nuit ou bien déployer toutes les machines en même temps, etc.),
- les politiques liées à la sécurité (imposer un niveau de sécurité particulier allant de l'automatisation totale à la nécessité de valider chaque étape…),
- les politiques liées à la gestion des sites (en spécifiant par exemple, l'endroit où les "packages" doivent être installés).

Ces politiques peuvent être locales, c'est à dire spécifique à un site ou globales (au niveau de l'entreprise). Elles peuvent servir lors de l'installation pour choisir la date de l'installation, le niveau d'automatisation (ce qui aura une influence sur le procédé d'installation), etc.

3.6.3. Le modèle d'entreprise

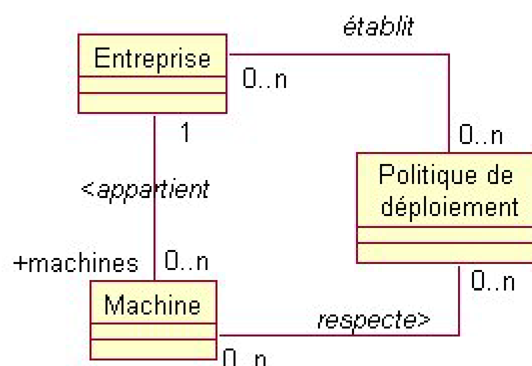

Figure 40 : Le modèle d'entreprise

4. Le monde des procédés

4.1. Introduction

Nous disposons maintenant de la modélisation des différentes entités du monde de l'environnement (entreprise, site, application, serveur d'applications, serveur de déploiement). Il faut maintenant décrire comment nous allons intéragir avec cet environnement afin de le modifier.

Pour commencer nous devons décrire le méta-modèle des procédés, à partir duquel nous allons créer nos modèles de déploiement (installation). Ces modèles vont nous permettre de décrire l'enchaînement des différentes manipulations sur l'environnement.

4.2. Meta Modèle des procédés

Le méta modèle que nous avons choisi d'utiliser pour automatiser nos activités est un modèle basé sur un ensemble limité de concepts.

"An activity is an atomic or composite operation, or a step in a process that contributes to the realization of an objective" [Promoter96]. On peut voir une activité comme la transformation d'un ensemble de données fournies en entrée (via un port d'entrée) vers un ensemble de données de sortie représentant le résultat de la tâche de l'activité. Une activité est dite composite lorsqu'elle contient d'autres activités appelées sous activités. Les relations entre l'activité et ses sous activités sont définies par les flots de données.

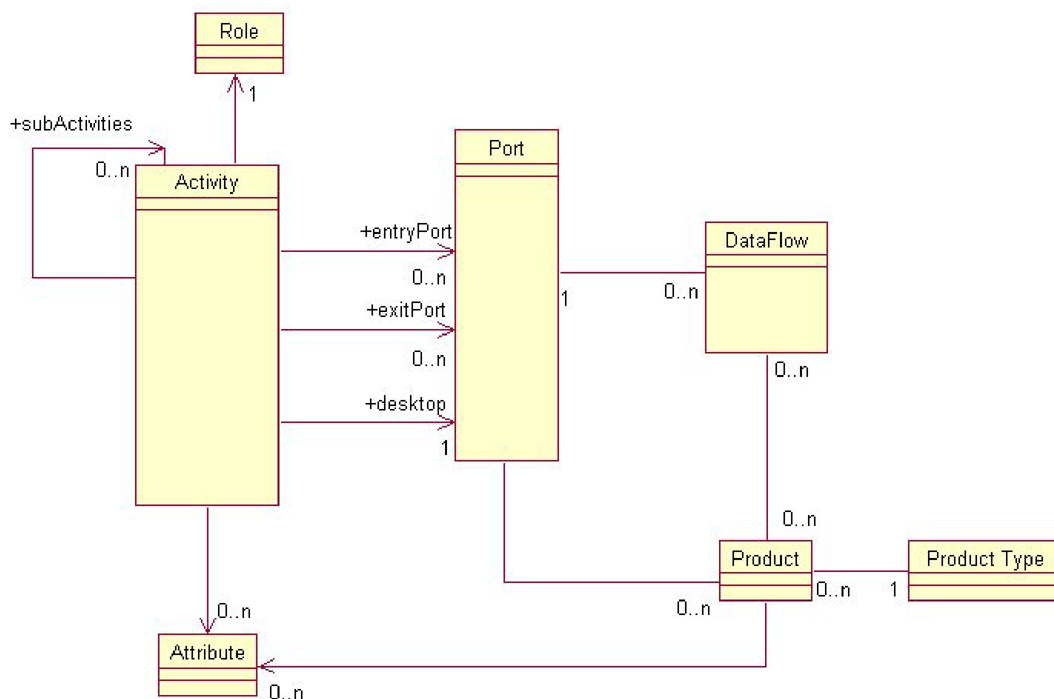

Figure 41 : Le méta-modèle des procédés

Nous allons maintenant détailler rapidement les différents éléments de ce modèle.

4.2.1. Les activités

Une activité est un pas dans un procédé qui contribue à la réalisation d'un objectif. Une activité, du point de vue fonctionnel, peut être considérée comme une transformation d'un ensemble de données fournies en entrée vers un ensemble de données de sortie, qui représentent le résultat de la tâche. Les activités peuvent être simples ou composites.

4.2.2. Les produits

Les produits sont les objets (ou du moins leurs représentants) que les activités doivent manipuler. Chaque produit est caractérisé à l'aide d'attributs. Par exemple, un produit de type fichier pourra avoir un attribut contenant son nom et un autre contenant son adresse. Les produits sont créés, transformés et détruits par les activités qui les utilisent. Les produits sont typés. Rappelons que ces produits peuvent correspondre à des éléments de l'environnement.

4.2.3. Les ports

Les ports, en général, définissent les possibles points d'entrée et de sortie des activités. Chaque port est caractérisé par l'ensemble des produits attendus. Une activité peut commencer, si dans un des ports d'entrée il y a tous les produits attendus. Une activité peut terminer, si dans un des ports de sortie il y a tous les produits attendus.

4.2.4. Les flots de données ("dataflows")

Les flots de données montrent comment les produits circulent entre les ports des activités. Les flots de données peuvent être considérés comme les arcs du graphe défini par le procédé.

4.2.5. Le poste de travail ("desktop")

Chaque activité a un poste de travail, où le responsable de la tâche va trouver les produits nécessaires pour pouvoir la réaliser. Dans notre cas, le responsable correspond à la réalisation automatisée de l'activité par des outils et les produits dans les ports d'entrée sont transférés automatiquement dans le poste de travail.

4.3. Le moteur de procédés

Un moteur de procédés doit au moins réaliser les fonctionnalités suivantes :

- Gérer les flot de données
- Gérer le cycle de vie des activités
- Gérer le cycle de vie des produits

Il faut aussi noter que le moteur de procédés doit garantir la reprise sur panne en cas d'erreur en assurant le retour à la version avant la panne de l'instance du modèle de procédés.

4.3.1. Gestion des flots de données

Gérer les flot de données signifie :

- gérer les transferts de données entre les activités (depuis un port de sortie vers un port d'entrée),
- gérer les transfert de données à l'intérieur de chaque activité depuis le port d'entrée vers le poste de travail ou vers un port de sortie, à condition qu'il soit attendu à destination.

Dans certains cas, en particulier dans le cas d'instances multiples du même produit, le produit restera en attente dans le port d'entrée tant que l'activité de ce port n'est pas terminée (sinon le produit sera détruit). Le produit sera transféré dès que l'autre instance aura quitté le port de sortie.

4.3.2. Gestion du cycle de vie des activités

Une activité possède un état. Son état initial est *init*. A partir d'*init*, l'activité passe à l'état *ready* lorsque le desktop contient les produits d'entrée (tous ceux issus d'un port d'entrée). Ensuite elle passe à *active* lorsqu'elle est assignée à un responsable. Lorsque l'activité devient *active* alors les sous activités sont créées. Et enfin elle passe à *terminate* dès que l'ensemble des produits attendus dans un port de sortie est transmis à l'activité suivante au moyen d'un flot de données.

Lorsque l'activité est terminée, toutes les éventuelles sous-activités sont détruites.

4.3.3. Gestion du cycle de vie des produits

Les produits sont créés par les activités, ils sont ensuite en général transmis par les flot de données à une autre activité, qui les manipulera et/ou les détruira. Le moteur de procédés doit gérer le cycle de vie des produits : création, transfert, manipulation et destruction. Il doit par exemple bloquer le procédé tant que les produits attendus n'ont pas été créés ou bien il doit détruire tous les produits d'une activité qui se termine et qui n'ont pas été transféré vers une autre activité.

4.4. Le langage de description d'un procédé

4.4.1. Principes

Nous allons maintenant décrire les procédés. Pour cela nous avons utilisé un langage de description de procédé. La description d'un procédé doit décrire :

- les activités et la façon dont elles s'enchaînent,
- les produits utilisés par les activités,
- les flots de données et les ports,
- les responsables de chaque activités.

Pour décrire toutes ces informations, nous avons choisi d'utiliser un langage de description graphique [DEA98]. La figure 8 donne un exemple de description graphique d'un procédé.

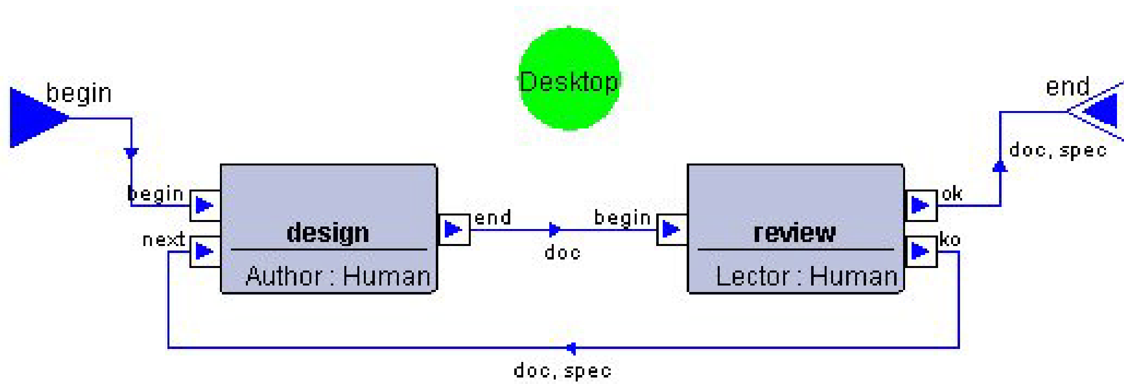

Figure 42 : Exemple de procédé

Pour simplifier la compréhension de notre langage graphique, nous allons décrire le langage textuel correspondant à l'aide d'une description style BNF du langage textuel correspondant. Un exemple d'une implémentation de ce langage en XML se trouve en annexe 1. Il correspond à la description de l'exemple de la figure 8.

4.4.2. Le langage de description

Dans cette partie, nous allons décrire la structure des informations nécessaires pour décrire chaque éléments des procédés (procédé, activité, produit, port, etc.).

<process> ::= <name, <u>root</u> : activity, {product}>

Un procédé a un nom (celui de son instance), il comporte une activité racine (celle qui sera exécutée la première) et un ensemble de produits utilisés.

<product> ::= <nom, product-type>

Un produit est identifié par son nom et son type (product-type). Le type d'un produit permet d'associer des attributs aux produits.

<activity> ::= atomic-activity | composite-activity

Une activité peut être atomique ou composite (elle posséde des sous-activités).

<atomic-activity> ::= <name, responsible, desktop, {entry-port}, {exit-port}>

Une activité atomique est caractérisée par son nom, le nom de son responsable, un poste de travail et un ensemble de ports d'entrée et de sortie.

<composite-activity> ::= <atomic-activity, <u>sub-activities</u> : {activity}>

Une activité est une activité atomique qui possède des sous activités. Ces sous activités sont liées entre elles via des flot de données qui seront décrit lors de la description des ports des activités.

<port> ::= desktop | entry-port | exit-port

Dans notre langage de description, nous avons assimilé le poste de travail à un port particulier.

<port> ::= <name, {product-name}, in : {dataflow}, out : {dataflow}>

Un port est décrit par son nom, un ensemble de produits attendus arrivant via des flots de données en entrée et sortant via des flots de données en sortie.

<dataflow> ::= <from : port, to : port, {product}>

Un flot de données a un sens de parcours, les produits arrivant par un port et sortant par un autre.

4.4.3. Conclusion sur le langage de description

A ce stade une activité est totalement abstraite. La réalisation signifie implanter concrètement sa sémantique, c'est à dire la création/manipulation/destruction de produits qu'elle doit réaliser. Par ailleurs il faut aussi assurer la cohérence des produits du monde des procédés avec les éléments du monde l'environnement.

5. La réalisation de la solution

5.1. Le principe

Nous allons maintenant nous intéresser à la réalisation de la solution, c'est à dire à la concrétisation des actions que chaque activité du procédé est censée réaliser. Pour l'instant notre solution est exprimée en termes d'activités, de produits et d'enchaînement des activités. Nous allons maintenant décrire comment les produits sont manipulés par les différentes activités. Pour cela nous avons introduit un langage de réalisation, que nous allons spécifier dans les sections suivantes.

Avant de détailler plus en détail les entités de notre langage, il faut préciser que toutes les opérations décrites ci-dessous ont lieu explicitement dans le poste de travail ("desktop") de l'activité dont on décrit la réalisation.

5.2. Les actions

Une action correspond à la sémantique de la réalisation, c'est à dire comment on réalise l'activité. Il existe deux moments où une activité est amenée à exécuter des actions, c'est à dire à modifier/manipuler des produits :

- lorsque l'activité devient active,
- lorsque l'une de ses sous activités (dans le cas des activités composites) vient de se terminer.

Le premier (le plus fréquent) correspond au traitement des produits dans une activité. Le second prend son sens lorsque l'on doit manipuler des produits uniquement après l'exécution d'une sous activité. Nous allons maintenant détailler l'utilisation des rôles et des produits.

5.3. L'utilisation des rôles

Lors de son exécution une activité modifie l'état de l'environnement. Pour modéliser cela on peut utiliser un rôle. On a besoin de préciser le nom du rôle (correspond au type d'une variable dans un langage de programmation), le nom de l'instance de rôle (correspond à la variable) et le nom de la machine sur laquelle le rôle (enfin plus précisément l'outil jouant ce rôle) sera exécuté. Un rôle étant associé à une machine donnée, il faut déclarer autant d'instances que de machines sur lesquelles le rôle sera exécuté.

On dispose de la déclaration suivante pour les rôles :

nomInstanceRôle : nomRôle at nomMachine;

Nous allons maintenant décrire les opérations disponibles sur les produits d'un modèle de procédé.

5.4. Les manipulations sur les produits

Le concept de produit est très important, car comme expliqué précédemment certains produits doivent être reliés à des éléments du monde de l'environnement. Les manipulations de ces produits dans le monde des procédés peuvent avoir des implications sur les éléments de l'environnement auxquels ils sont liés. Par exemple, un produit qui dans un modèle de procédé représente une machine correspondra à un élément du monde de l'environnement.. Nous avons distingué 2 types de produits :

- les produits représentant les éléments du modèle de l'environnement,
- les autres produits.

Les autres produits que l'on nommera à partir de maintenant comme les produits normaux permettent de représenter tous les produits qui n'ont pas de correspondance directe avec les éléments du monde de l'environnement. Par exemple, un produit contenant le nom d'un fichier. Nous allons maintenant décrire les différentes manipulations que l'on peut réaliser sur un produit.

5.4.1. La création de produit

On ne peut créer un produit que si celui-ci est attendu dans le poste de travail, ce qui signifie que l'un au moins des flots de données partant de ce port fera circuler le produit. La classe *Product* est un élément du méta-modèle de procédé, un produit est une instance de cette classe. Donc créer un produit consiste à créer au niveau du modèle une instance de la classe.

La création d'un produit A lié à un élément B du monde de l'environnement ne signifie pas nécessairement que l'on va créer B. Les produits dans le monde des procédés servent à garder des informations utilisées par les activités qui les reçoivent via leurs ports. Prenons l'exemple d'une activité qui a pour objectif de chercher un serveur d'application hébergeant un "package" donné.

Figure 43 : Exemple d'activité

L'activité va envoyer, via son port de sortie, un produit serveur. Comme il n'existait pas auparavant, le procédé va créer une instance du produit dans le monde des procédés. Par contre le serveur d'applications existait déjà dans le monde de l'environnement, il ne faut donc pas le créer mais uniquement créer le lien entre lui et l'instance de produit nouvellement créé.

Par contre lors de l'installation, une nouvelle application va réellement être ajoutée à une machine et par conséquent il y a bien un lien direct de création, dans ce cas, entre le concept de création dans le monde des procédés et celui dans le monde de l'environnement.

La création réelle de l'application sera à la charge d'un rôle à déclencher qui interviendra directement dans le monde de l'environnement pour ajouter l'application à la machine. Ensuite, l'environnement sera mis à jour soit lors d'une prochaine introspection soit directement en utilisant un rôle.
Par conséquent les manipulations sur les produits n'ont pas de conséquence directe sur les éléments du monde de l'environnement auquel ils sont liés. Les manipulations étant assurée directement par les rôles.

Pour créer un produit on dispose de la méthode newProduct(). Il est nécessaire de préciser le nom du produit ainsi que de son type. Le résultat de la méthode correspondant à une instance du produit créé.

newProduct(nomProduit, nomTypeProduit) ;

Pour résumer, la création d'un produit lié à un élément du monde de l'environnement permet de réaliser ce lien et ainsi de donner aux activités manipulant ce produit l'accès à une représentation de l'élément et la possibilité d'agir réellement sur l'objet via des outils.

5.4.2. La destruction de produit

Détruire un produit, cela signifie que le produit ne sera plus utilisé par les autres activités du procédé. Si l'on reprend l'exemple précédent, on voit que le produit *nomPackage* ne sort pas de l'activité par le port de sortie, il doit donc être détruit lorsque l'activité se termine.

On dispose de la méthode destroyProduct(). On doit préciser le nom du produit ainsi que celui du port dans lequel il se trouve.

destroyProduct(nomProduit,nomPort) ;

Dans le cas des produits liés à l'environnement, la destruction de l'instance de produit dans le monde des procédés ne signifie pas la plupart du temps que l'élément lié va être détruit lui aussi. La destruction éventuelle (par exemple, si l'on désinstalle une application sur une

machine) est assurée par un rôle à déclencher et l'environnement (et donc l'élément du modèle d'environnement) est mis automatiquement à jour.

5.4.3. La modification de produit

Modifier un produit, cela signifie que l'on change la valeur d'un des attributs de l'instance du produit concerné. Dans le cas des produits liés à l'environnement, cela signifie que l'on change une de ses caractéristiques. Cette modification peut être réalisée soit par un outil, soit par un autre système, cela dépendra de l'implémentation choisie.

Pour réaliser une modification, on n'utilise pas directement une méthode, on fait directement une affectation de la valeur. La syntaxe suivante permet d'accéder directement à l'attribut (ainsi qu'à sa valeur) : $activity.port.produit.attribut. Pour la modifier il suffit d'écrire :

```
$activity.port.produit.attribut = "nouvelle valeur";
```

5.5. Les manipulations sur les procédés

Manipuler un procédé cela revient à ajouter/modifier/supprimer une activité. Cette partie de notre langage (dont nous allons décrire quelques fonctionnalités : la création d'activité et de flot de données) peut être utilisée pour :

- Spécialiser un procédé dont la description n'est pas suffisamment détaillée
- Modifier un procédé existant

Ces manipulations peuvent être statiques (spécifiées dans le template avant l'installation) ou dynamiques (voir la section 6.2 de ce chapitre).

5.5.1. La création d'activité

Dans notre langage, nous avons donné la possibilité de créer une activité. La création d'une activité ne signifie pas que l'on va créer entièrement une nouvelle activité mais que l'on va insérer une activité existante (c'est à dire qui a déjà été spécifiée). C'est pourquoi les implémentations de notre langage devront s'assurer de la cohérence de l'ajout (nom des ports, bons types des produits…).

On dispose de la méthode newActivity() pour créer une nouvelle sous-activité à l'activité. On a besoin du nom de la nouvelle sous activité ainsi que de son type. Le type permet d'avoir accès aux templates associés ainsi qu'à sa description.

```
newActivity(nomActivité, typeActivité) ;
```

Il faut noter aussi que les produits en entrée et en sortie doivent avoir le même nom que ceux de l'activité mère, ce qui implique soit d'utiliser directement les bons noms ou bien d'assurer à l'exécution l'activité de renommage présentée dans la section suivante.

Pouvoir créer des activités ne sert à rien, si l'on ne peut pas l'insérer dans un enchaînement d'activités. Nous avons donc ajouté la possibilité de créer des flot de données.

5.5.2. La création de flot de données

Pour la création de flot de données, on dispose de la méthode newdataFlow(). On a besoin de préciser le nom de l'activité d'où partira le flot de données (et son port de sortie), ainsi que celui de l'activité d'arrivée et de son port d'entrée.

```
newDataFlow(nomPortDepart, nomActivitéDépart,
            nomPortArrivé, nomActivitéArrivée) ;
```

En offrant dans notre langage la possibilité de créer des flot de données, nous avons du considérer le poste de travail des activités comme un port. De cette manière nous sommes capable de créer un flot de données partant du poste de travail vers une nouvelle activité par exemple.

5.5.3. La suppression d'activité

Pour la suppression d'une activité, nous disposons de la méthode destroyActivity(). On doit préciser le nom de l'activité à détruire.

```
destroyActivity(nomActivité) ;
```

La destruction d'une activité est nécessairement associée à la destruction/redirection des flots de données entrant et à la destructions des flots de données sortant. Il faudra aussi traiter le cas des produits qui devaient être transférés via ces flots de données sortant (car les produits sont attendus par d'autres activités).

5.5.4. La suppression de flot de données

La suppression d'un flot de données est possible grâce à la méthode destroyDataFlow(). On a besoin du nom du port de départ du flot de données et du nom du port d'arrivée du flot de données (ainsi que le noms des activités possédant ces ports).

```
destroyDataFlow(nomPortDepart, nomActivitéDépart,
                nomPortArrivé, nomActivitéArrivée) ;
```

Nous avons choisi de ne pas proposer de méthode permettant la redirection d'un flot de données (c'est à dire le changement du port de départ ou bien du port de sortie). Pour réaliser une redirection, il faut supprimer le flot de données initial puis créer le nouveau.

5.6. Conclusion sur ce langage

Ce langage développé dans le cadre du déploiement a été conçu pour être indépendant du déploiement. Il est actuellement utilisé pour piloter des Web-Services. Sans ce langage, nous disposions d'une machine de workflow permettant de décrire la solution d'un problème ainsi que d'un moteur de procédé qui lui gère la sémantique de la machine. Ce qui est suffisant dans le cas où les activités sont exécutées manuellement par des utilisateurs. Dans notre cas, il a fallu rajouter la partie matérialisant (ou concrétisant) chaque activité du procédé, ce qui permet de réaliser la solution.

6. Utilisation : Création & modification de procédé

Maintenant on dispose de tous les éléments pour créer et modifier des processus de déploiement : la description des procédés et le langage permettant d'écrire les templates associés à chaque activité. Un template correspond au fichier contenant la partie réalisation d'une activité et chaque activité a son propre template.

6.1. La création de procédé

6.1.1. Principe

Notre objectif est toujours d'exprimer la solution d'un problème d'installation à l'aide d'un procédé exécutable et automatisé. Ce qui signifie décrire le procédé avec le langage de description et associer à chaque activité un template décrivant la réalisation de l'activité. Nous allons proposer plusieurs approches (non exclusives) pour la réalisation de cet objectif : la création d'un procédé. Nous avons identifié plusieurs types de création :

- la création d'un procédé de A à Z,
- la création d'un procédé en composant des procédés existants,
- la création d'un procédé en modifiant un procédé existant.

Pour réaliser ces méthodes de création, nous nous sommes appuyés sur le langage graphique de description et sur le langage de réalisation (à travers les templates).

6.1.2. La création de procédé de A à Z

La première approche est la plus basique, elle consiste à écrire entièrement le procédé et ensuite à écrire chaque template associé à une activité.

Dans le cas de procédés de petite taille (quelques activités) on peut utiliser cette approche. Il faut remarquer que quelle que soit l'approche choisie il y aura toujours un moment où l'on devra décrire à la main le procédé et ses templates.

Cette façon de faire permet :

- de créer entièrement un procédé,
- de créer des briques de base du déploiement.

En effet, notre approche prend tout son intérêt dans le fait que l'on va pouvoir réutiliser des procédés en tant qu'unité de composition. Prenons comme exemple, l'activité (ou le procédé) de transfert. On obtient le schéma suivant :

Figure 44 : Exemple d'une activité de transfert

Associé avec cette activité on dispose d'un template (écrit à la main ou généré par un outil) qui réalise le transfert proprement dit.

6.1.3. La réutilisation de procédés

On dispose d'une base de procédés. On peut les voir comme des briques de base du déploiement (ou d'un autre domaine). Notre procédé de transfert en est un bon exemple.

Pour réutiliser un procédé, il faut le considérer comme un procédé atomique même si ce n'est pas le cas.

Un point important lors de la réutilisation, c'est la phase de renommage. En effet, les procédés que l'on réutilise ont leurs propres produits en entrée(s) et en sortie(s). Il faut donc lors de la réutilisation s'assurer d'abord que l'on fournit bien au procédé des produits de même type que ceux attendus.

Reprenons l'activité de transfert précédente et essayons de la composer avec une autre activité de base : l'installation.

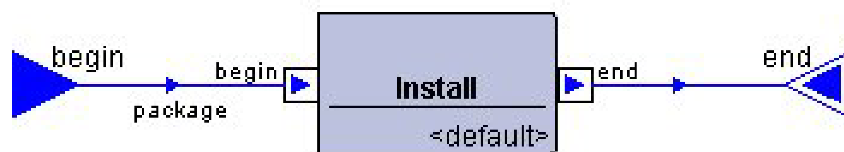

Figure 45 : Exemple d'une activité d'installation

Les produits package et file sont de même type, on peut donc créer un flot de données entre le port « end » de Transfer et celui de « begin » de l'activité Install. Par contre il faut résoudre le problème de renommage. Il va falloir renommer au moins l'un des 2 produits. Dans notre cas, on va supposer que l'on change file en package pour obtenir :

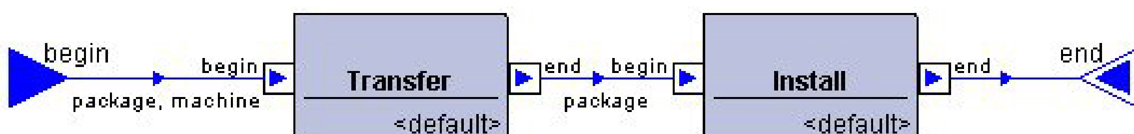

Figure 46 : Exemple de réutilisation

En plus de renommer le produit file en package au niveau de la description, il va falloir aussi renommer le produit file dans le (ou les) fichiers de template associé(s) à l'activité Transfer. En effet le produit file n'existe plus au niveau de cette composition, il faut le remplacer à chaque fois.

Analysons la composition au niveau des descriptions et templates. L'activité Transfer avait un fichier de description détaillant sa propre structure (sous activités, flot de données, produits) et elle avait un certain nombre de fichiers template. De même pour l'activité Install. Le

résultat de cette composition correspond à un fichier de description et des templates. Le fichier de description correspond (dans notre cas) au fichier de description de Transfer (fichier modifié pour tenir compte du renommage de file en package) plus celui de Install et enfin la description de la composition des 2 activités. Les templates correspondent aux templates de Transfer (modifiés), plus ceux de Install et éventuellement celui associé à la composition.

Maintenant nous disposons d'une activité réalisant le transfert puis l'installation d'un "package". Cette activité pourra être elle aussi réutilisée par une autre composition.

6.1.4. La modification statique de procédé

Dans cette partie, nous ne considérons que les modifications (ajout/suppression d'activités et de flots de données) réalisées directement à partir du langage graphique de description. Les autres cas (se basant sur les fonctions de modification de procédé offertes par notre langage) concernent ce que nous appelons les modifications dynamique de procédé (section 6.2).

Dans les cas précédents, nous ne nous intéressions pas à la structure (composite ou atomique) des activités que nous réutilisions. Dans le cas de modification de procédé, c'est le contraire.

Supposons que nous souhaitions reprendre la composition précédente et ensuite la modifier pour obtenir le nouveau procédé suivant (figure 13) :

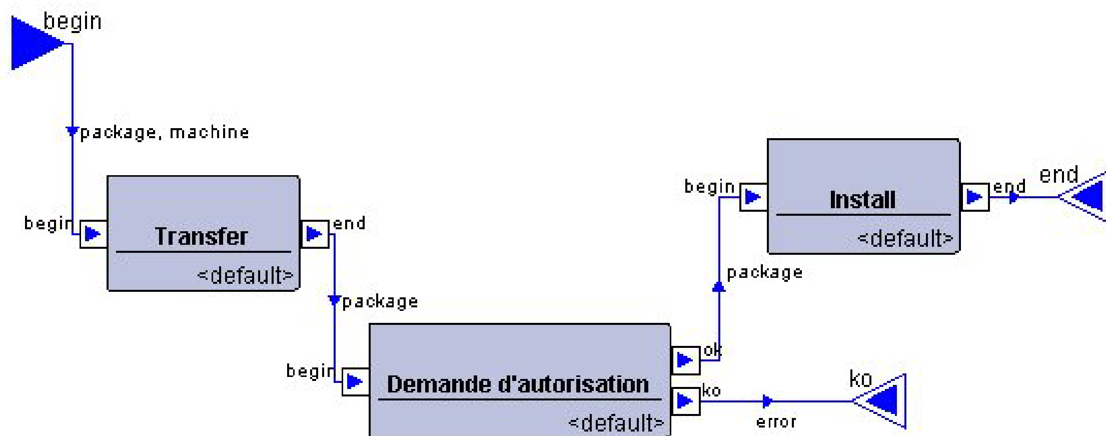

Figure 47 : Exemple de modification

En terme de modification, la plupart des choses sont possibles, mais certaines sont plus compliquées que d'autres. Ici on a rajouté une activité (« Demande autorisation »), modifié un flot de données et ajouté deux nouveaux flots de données; enfin nous avons rajouté un port de sortie. Ce que l'on ne peut pas faire, c'est de supprimer un produit en entrée d'une activité (par exemple le produit package de Install).

6.2. La modification dynamique de procédé

6.2.1. Le principe

Nous disposons maintenant de procédés exécutables. Nous avons vu dans la section 5 de ce chapitre, que notre langage offre des possibilités de manipuler "dynamiquement" les procédés.

Les modifications dynamiques permettent d'intervenir directement sur un procédé en cours d'exécution. Par exemple, en cas de problème lors de l'exécution, il peut être intéressant d'offrir la possibilité au déployeur de "débuguer" le procédé en le modifiant dynamiquement. On peut aussi vouloir spécialiser le procédé mais dynamiquement cette fois. Pour cela nous avons ajouté dans notre langage le concept de point d'arrêt permettant l'intervention dynamique (section 6.2.3).

Nous allons maintenant voir quelques exemples de modification.

6.2.2. L'appel distant à une sous-activité

Dans le cas d'un problème, le déployeur peut vouloir lancer une activité de "débugage" pour analyser la situation. On utilise la méthode newProcess() dont la signature est la suivante :

newProcess(nomProcess, nomPortEntrée, listeProduit);

Pour pouvoir exécuter un process, on a besoin de préciser le nom du modèle de procédé, le nom du port d'entrée que l'on veut utiliser et la liste des instances de produit attendus dans le port d'entrée. La cohérence des paramètres de la méthode avec la réalité de l'activité (type des produits d'entrée et de sortie…) sera à vérifier avant de réaliser l'ajout.

6.2.3. L'ajout d'une sous-activité

L'ajout d'activité dans un procédé existant peut être utilisé pour spécialiser ou pour modifier le procédé. L'exemple suivant décrit graphiquement le résultat de l'ajout d'une sous-activité.

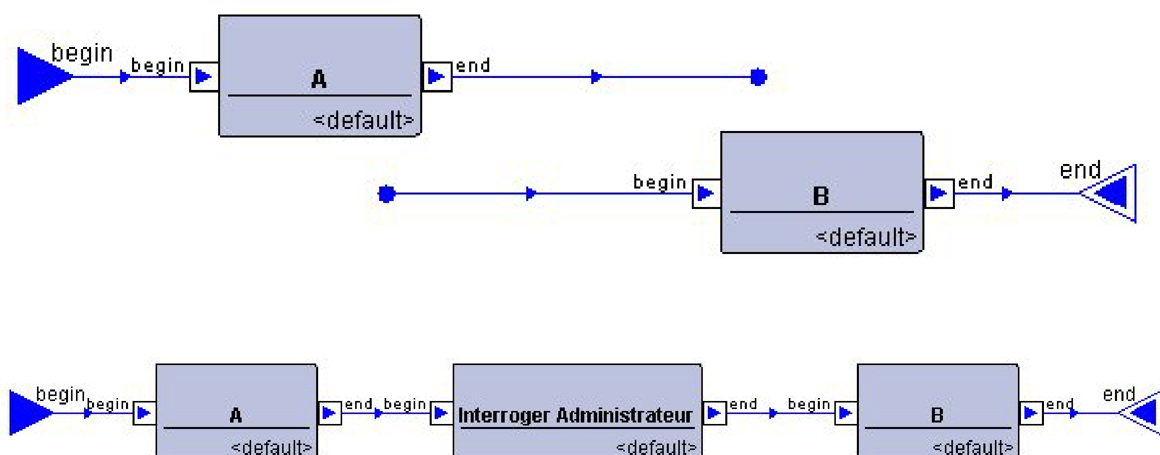

Figure 48 : Exemple de l'ajout d'une sous-activité

Dans cet exemple, le procédé initial ne décrivait pas comment on passait de l'activité A et celle de B. Pour être exécutable, on doit préciser à l'aide de notre langage comment on réalise le passage de A à B. Dans notre exemple, nous avons choisi de passer par une sous-activité intermédiaire. Cet ajout peut s'écrire de la façon suivante :

```
// On utilise une variable activity de type Activity
$activity = newActivity("Interroger Administrateur", InterractionActivity) ;
newDataFlow($A.end, $A, $activity.begin, $activity);
newDataFlow($activity.end, $activity, $B.begin, $B);
```

On commence par créer la nouvelle activité en précisant son nom et son type. Puis on réalise les connexions (via les flot de données) concernant les ports de cette nouvelle activité.

Ces lignes de codes peuvent être ajoutées lors de la création du template associé à l'activité mère, c'est à dire de manière statique ou bien dynamiquement lors de l'exécution. Pour cela, on aura du au préalable ajouter la ligne suivante dans le template :

```
newBreakPoint(deployeur) ;
```

Cela signifie que le déployeur prend la main à ce moment de l'exécution du procédé.

Il reste maintenant à préciser (ceci est valable dans les 2 cas) quand l'ajout doit être réalisé. Pour cela nous utilisons la ligne de code suivante :

```
case $(A).isTerminated() && $(A).getFinalExitPort() == $A.end :
{
//code
$activity = newActivity("Interroger Administrateur", InterractionActivity) ;
newDataFlow($A.end, $A, $activity.begin, $activity);
newDataFlow($activity.end, $activity, $B.begin, $B);

//ou selon la manière choisie
newBreakPoint(deployeur) ;
}
```

Cela signifie que le code sera exécuté lorsque l'activité A sera terminée par le port de sortie nommé "end".

6.3. Conclusion sur l'utilisation de procédés

Nous venons de voir dans ce chapitre les différentes façons offertes par notre environnement pour créer des procédés. Ces créations utilisent les avantages de la réutilisation de procédés existants soit pour les modifier ou les composer pour créer d'autres procédés.

Une fois ces procédés créés, on peut les exécuter et là notre environnement (via les fonctions de modification de procédé de notre langage de réalisation) permet la modification dynamique des procédés. Ceci rend possible l'intervention d'un humain (ou d'un outil) directement sur le procédé en cas de problème par exemple ou en cas de modification en directe du procédé.

7. Résumé de l'activité d'installation

Nous avons vu dans ce chapitre comment décrire un procédé à la fois son architecture (la description de l'ordonnancement de ses sous-activités et celle des flots de données), la description de la manière de réaliser les activités (comment lier une activité à des rôles, modifier les produits…) et comment décrire les modifications dynamique.

De cette façon, nous sommes capable de décrire des installations et de prévoir des modifications en cas de problèmes. Par exemple, on peut lors d'une installation choisir de continuer l'installation même en cas de problèmes car ceux-ci sont résolvables plus tard.

Nous avons terminé ce chapitre par quelques manières offertes par notre environnement de déploiement ORYA pour la création de modèles de procédés en insistant plus particulièrement sur les capacités de réutilisation et d'évolution de l'existant.

Nous allons maintenant dans les chapitres suivants décrire notre implémentation d'ORYA et aussi décrire celle du langage de réalisation et d'évolution. Nous présenterons aussi des exemples de procédés développés dans le cadre de notre collaboration avec la société Actoll [Actoll].

Chapitre VII

L'implémentation

1. Introduction

Les chapitres précédents ont exposés successivement les objectifs de notre environnement de déploiement ORYA, puis deux des activités du déploiement (la sélection et l'installation) étudiées dans le cadre de cette thèse. L'objectif de ce chapitre est maintenant de décrire l'implémentation d'ORYA. La section 2 décrit l'implémentation de l'architecture d'ORYA en insistant plus particulièrement sur les entités de l'environnement (serveur de déploiement, d'applications et les sites). La section 3 décrit l'implémentation du framework de sélection basé sur le modèle de composant décrit dans le chapitre 5. Les sections 4 et 5 décrivent l'activité d'installation (langage de réalisation, description et exécution d'un procédé d'installation). Puis la section 6 présente des outils de déploiement comme l'outil de packaging. Et enfin la section 7 propose un bilan de l'implémentation d'ORYA en sachant que le chapitre suivant de cette thèse fait le bilan de l'évaluation d'ORYA dans le cadre d'une expérimentation industrielle.

2. L'architecture d'ORYA

2.1. Introduction

Comme indiqué dans le chapitre 4, ORYA est une fédération de déploiement. Le but de ce chapitre n'est pas de présenter une implémentation de l'approche des fédérations, ni l'infrastructure de communication de notre environnement, ni le moteur de procédé utilisé, car ils font partis des outils fournis par la fédération. Nous présentons uniquement les implémentations des outils et des langages développé dans le cadre du déploiement.

Il faut noter aussi que la plupart des outils et entités ont été développés dans le cadre des activités d'installation et de sélection. Cela signifie qu'ils seront potentiellement modifiés dans le futur afin de pouvoir être utilisés durant les autres activités du cycle de vie du déploiement. C'est le cas particulièrement du modèle de site et du modèle d'application.

L'architecture d'ORYA est composée d'un ensemble de composants : le serveur d'application, le serveur de déploiement et le site. Nous allons dans cette section décrire l'implémentation réalisée pour chacun de ces composants.

2.2. Les serveurs d'applications

2.2.1. Introduction

Le rôle d'un serveur d'application est de servir d'hébergeur à des "packages" et de gérer leur cycle de vie. Dans ORYA, un serveur d'application a principalement les fonctionnalités suivantes :

- fournir la liste de ses "packages" : cette information est utilisée par le serveur de déploiement pour sa connaissance de l'environnement,
- retourner le fichier de description d'un "package",
- retourner un "package" selon son nom (et son identifiant).

Toutes ces fonctionnalités sont utilisées par le serveur de déploiement à la fois pour mettre à jour sa connaissance de l'environnement (la liste des "packages" de chaque serveur

d'applications) et lors des différents déploiements (recherche du "package" le plus cohérent à déployer sur une machine).

En plus de ces fonctionnalités, le serveur d'application doit assurer la persistance de ses informations. Pour cela nous nous sommes appuyés sur les schémas de description XML dont nous allons maintenant donner le modèle ainsi qu'un exemple.

2.2.2. Le modèle de description

Nous avons identifié deux types d'informations concernant un serveur d'applications qui doivent être persistantes :

- les informations sur les "packages" hébergés,
- les propriétés associées au serveur d'application.

Une propriété est une information, qui peut être soit commune à tous les serveurs d'applications de l'environnement ou bien spécifique à un ou plusieurs serveurs. Dans le premier cas, l'information peut être utilisée par l'environnement de déploiement. Le répertoire où se trouve tous les "packages" est un bon exemple de ce type d'information.

L'information sur les "packages" fournit la liste des "packages" disponibles et pour chacun d'eux des informations comme le nom de l'application, l'identifiant, le répertoire où se trouve le "package" et la liste des dépendances. Ces informations sur les "packages" permettent de réaliser une première sélection/vérification sans avoir besoin d'extraire les informations contenues dans le "package".

Le schéma du modèle de description d'un serveur d'application se trouve en annexe 2. Nous allons maintenant présenter un exemple puis présenter l'interface graphique d'un serveur d'applications.

2.2.3. Un exemple

Dans cet exemple, le serveur d'applications contient deux "packages". Le premier correspond à l'application *oracle* dont le "package" se trouve à l'adresse suivante:
d:\fede\fedexec\client\fedeClient\data\packages\oracle.zip. Le second "package" correspond à l'application *smonB* (il s'agit de l'une des applications développées par Actoll, voir le chapitre 8), dont le "package" se trouve à la même adresse que le "package" précédent (il y a juste le nom du "package" qui change : *smonB.zip*). L'application *smonB* dépend de l'application *oracle*.

```
<?xml version="1.0" ?>
<ApplicationServerModel xmlns="http://castor.exolab.org/">
        <property>
                <name>package.home</name>
                <value>d:\fede\fedexec\client\fedeClient\data\packages</value>
        </property>
        <package>
                <name>oracle</name>
                <id>nc</id>
                <version>1</version>
                <url>oracle.zip</url>
        </package>
```

```
        <package>
                <name>smonB</name>
                <id>nc</id>
                <version>1</version>
                <url> smonB.zip</url>
                <dependency>oracle</dependency>
        </package>
</ApplicationServerModel>
```

Ce serveur d'applications a aussi une propriété nommée *package.home* dont la fonction est de fournir à l'environnement de déploiement la racine à partir de laquelle tous les "packages" se trouvent. Dans notre implémentation, nous avons choisi d'utiliser un système de fichiers pour conserver les "packages".

2.2.4. L'interface graphique

La partie graphique du serveur de déploiement (voir Figure 49) permet d'avoir la liste des "packages" disponibles ainsi que la description (bouton "Info") de chacun de ses "packages". L'administrateur du site peut ajouter ou supprimer un "package" à l'aide de simple boutons ("Add" et "Remove"). A chaque ajout/suppression, la description du serveur est mise à jour et un événement est émis vers le serveur de déploiement afin qu'il se synchronise.

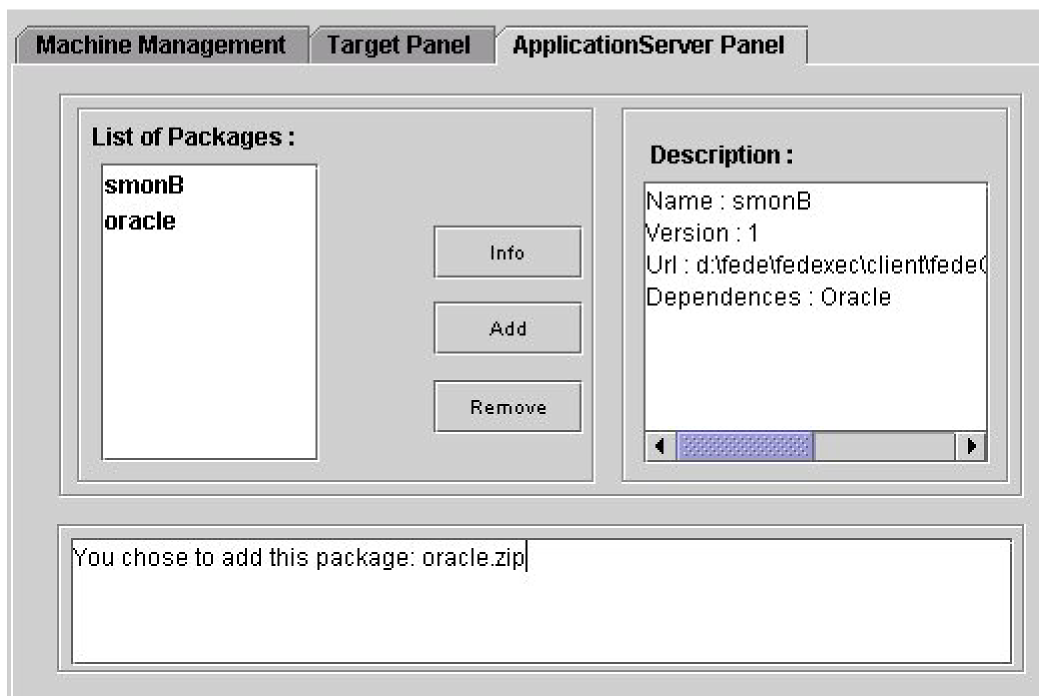

Figure 49 : Interface graphique d'un serveur d'applications

2.2.5. Conclusion

Un seul type de serveur d'applications a été développé dans le cadre d'ORYA. Basé sur le système de fichiers de Windows, il assure la persistance des données essentielles au déploiement ("packages" et propriétés) et offre aussi à son administrateur une interface graphique de gestion.

2.3. Le serveur de déploiement

2.3.1. Introduction

Le rôle du serveur de déploiement est de gérer les différents déploiements et d'assurer la connaissance en temps réel de l'état de l'environnement (comme la connaissance des "packages" disponibles). Notre outil offre :

- une interface pour visualiser l'état de l'environnement,
- une interface pour démarrer les différents déploiements.

Cet outil s'appuie sur le moteur de procédé (Apel) de la fédération ainsi que sur d'autres outils comme ceux détaillés dans la section 6 [Vil02].

2.3.2. La gestion des installations

Le serveur de déploiement permet de lancer une installation sur une ou plusieurs machines, selon que l'on souhaite un déploiement unitaire ou multiple. Pour cela, on choisit la (ou les) machine(s) cibles parmi la liste des machines connectées. A l'heure actuelle la connaissance de l'environnement n'étant pas persistante, l'interface de gestion ne permet que le lancement de déploiement en temps réel et pas encore la planification des déploiements.

Intéressons nous au déploiement unitaire. Une fois l'application et la machine choisies on peut lancer l'installation. Cela consiste à exécuter un procédé d'installation (il est décrit dans la section 5 de ce chapitre). Une fois trouvé le bon "package", celui-ci est transféré sur le serveur de déploiement. Nous avons choisi pour cette version d'ORYA d'utiliser un modèle de procédé d'installation unique qui est instancié selon chaque application. La section 5.3 décrit plus précisément ce que signifie cette instantiation. Signalons que l'instance de procédé est unique pour chaque installation.

Comme nous gérons plusieurs installations en même temps, nous avons séparé sur le serveur de déploiement les informations (les descriptions, l'instance de procédé ainsi que son évolution) propres à chaque installation. Ainsi nous sommes capables de gérer les différentes installations sans risque de mélanger les informations.

2.3.3. L'interface graphique

L'interface graphique (Figure 50) est divisée en deux parties. Celle de gauche permet de visualiser l'état de l'environnement (les machines et leurs applications déjà installées ainsi que les serveurs d'applications et leurs "packages"). Cette information est mise à jour en temps réel pour prendre en compte la connexion/déconnexion des machines ou l'installation d'une nouvelle application par exemple. La partie de droite contient un ensemble de vues dont une interface de gestion des installations (celle qui correspond à la partie de droite de la Figure 50) ainsi qu'un certains nombre de vues utilisées pour la gestion de traces du déploiement (voir la section 6.4).

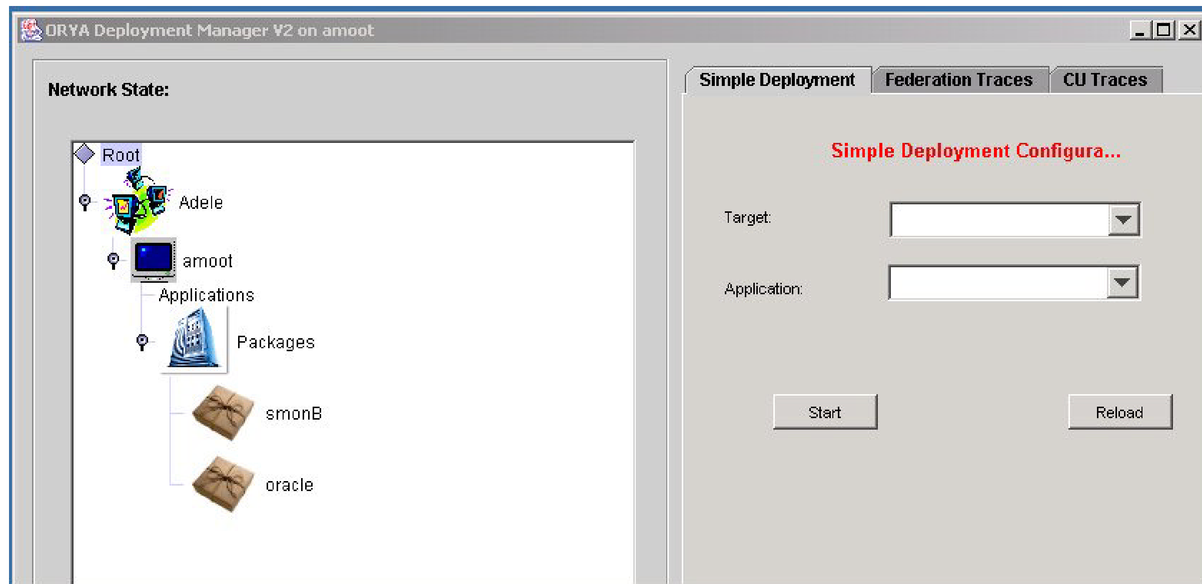

Figure 50 : Interface graphique du serveur de déploiement

2.3.4. Conclusion

Le serveur de déploiement permet au déployeur d'avoir une vision globale de son environnement ainsi qu'une interface de gestion des installations. Cet outil permet de lancer les installations. D'autres outils comme le moteur de procédé prennent ensuite le relais et offrent au déployeur un certain nombre de fonctionnalités comme le monitoring, un service de trace, etc..

2.4. Les sites

2.4.1. Introduction

Le rôle du modèle de site est de fournir des informations sur les applications déjà installées, sur les caractéristiques matérielles du site ainsi que sur les politiques de déploiement associées au site (ou à son utilisateur). L'implémentation actuelle est assez simple et ne propose pas encore toutes ces fonctionnalités. Elle permet :

- d'introspecter l'environnement Windows pour connaître l'ensemble des applications installées indépendamment d'ORYA,
- de lancer une installation en mode "pull",
- de supprimer une application installée (en gérant le problème des dépendances)

De la même façon (et pour les mêmes raisons) que pour les serveurs d'applications, un certain nombre d'informations sur les sites sont persistantes.

2.4.2. Le modèle de description de site

Le modèle de description est similaire à celui des serveurs d'applications. Un site est décrit en termes d'applications et de propriétés. La description d'une application est constituée de son nom, de l'endroit où est stocké son "package", de son identifiant ainsi que de ses dépendances éventuelles.

Le choix de garder physiquement le "package" une fois l'application installée a été fait en prévision des étapes futures du déploiement, en particulier celle de reconfiguration qui nécessite la présence du "package". L'information sur les dépendances est utilisée lors de la désinstallation (on a choisit d'interdire la désinstallation d'une application dont dépend une autre application).

Le schéma de cette description est donné en annexe 3. La section suivante présente un exemple de description d'un site.

2.4.3. Un exemple

Dans cet exemple, le site contient une propriété *TMP.SOURCE* qui indique au serveur de déploiement quel est le répertoire d'extraction des "packages". Ce site contient une application, plus précisément l'application *oracle*, qui a été installée via ORYA.

```
<?xml version="1.0" ?>
<SiteModel xmlns="http://castor.exolab.org/">
        <property>
                <name>TMP.SOURCE</name>
                <value>d:\tmp</value>
        </property>
        <property>
                <name>installed.home</name>
                <value>d:\fede\fedexec\client\fedeClient\data\installed</value>
        </property>
        <application>
                <name>oracle</name>
                <id>nc</id>
                <version>1</version>
                <url> d:\fede\fedexec\client\fedeClient\data\installed\oracle.zip</url>
        </application>
</SiteModel>
```

2.4.4. L'interface graphique

L'interface graphique permet à l'utilisateur de connaître les applications installées via ORYA ou par d'autres outils de déploiement (la fonctionnalité a été implémentée uniquement sous Windows, le bouton *Introspect* permet de lancer l'introspection sur le site afin d'afficher la liste des applications installées). L'utilisateur peut accéder à l'information concernant chaque "package". Il peut aussi désinstaller une application installée via ORYA (bouton *Remove*). Pour cela, on fait appel au procédé d'installation contenu dans le "package" de l'application en question (que l'on a gardé). Dans l'exemple de la Figure 51, seule l'application *oracle* a été installée via ORYA, la liste de la partie gauche de la fenêtre correspond aux autres applications du site (non installées via ORYA). On verra dans la section 5 comment à partir du procédé d'installation on peut obtenir celui de la désinstallation.

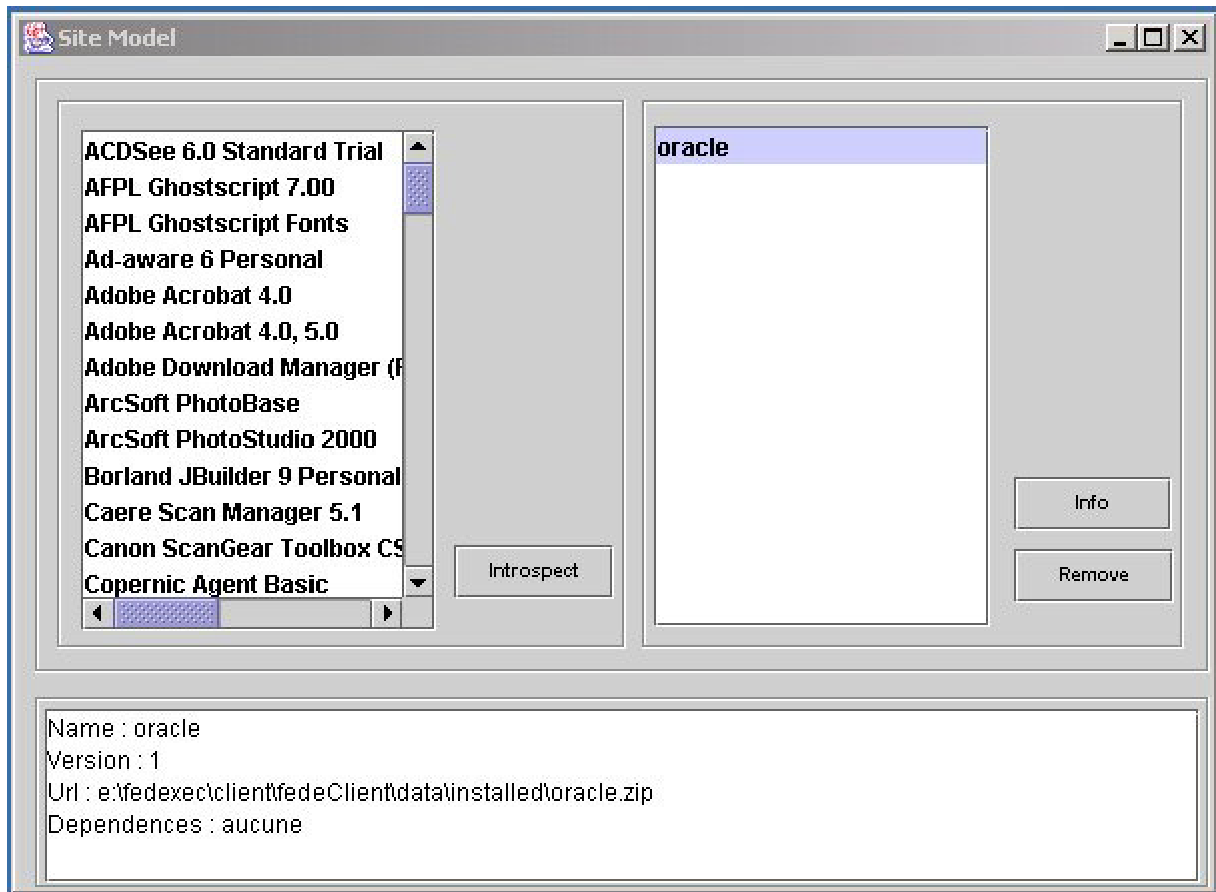

Figure 51 : Interface graphique d'un site

2.4.5. Conclusion

L'implémentation du modèle de site est actuellement centrée activité d'installation, c'est à dire que la représentation d'une application installée est très simple et ne permet pas pour l'instant la réalisation des activités comme la mise à jour.

2.5. Conclusion sur l'architecture

Nous venons de voir comment sont implémentées les différentes entités de l'environnement ORYA. Nous avons mis l'accent sur les notions de persistance et d'interface graphique, car elles sont nécessaires pour toutes les implémentations. En effet, elles permettent une meilleure ergonomie (la partie graphique) et une utilisation plus réaliste (la persistance) d'un prototype. Nous allons maintenant voir l'implémentation des activités de sélection et d'installation

3. La création de configurations

3.1. Introduction

La partie de l'implémentation qui réalise l'activité de sélection et de configuration a été réalisée à un moment où le choix de se baser sur les fédérations n'avait pas été encore pris. Cependant, il est relativement aisé de transformer ce prototype afin de l'utiliser dans ORYA.

Ce prototype gère les aspects suivants :

- la gestion des composants de notre modèle dans une base de données orientée object,
- la gestion des annotations de ces composants,
- la construction des configurations.

Tous ces aspects ont été réalisés dans un framework de sélection basé sur un modèle MVC [BMR96]. Nous allons maintenant détailler ces 3 aspects ainsi que l'interface graphique développée.

3.2. La gestion des composants dans la base de développement

3.2.1. Principe

La gestion des composants dans la base de développement est basée sur les fonctionnalités suivantes :

- l'ajout et la suppression des composants à l'aide d'une interface graphique,
- la gestion de la cohérence de la base,
- l'interface graphique d'exploration de la base.

Nous allons maintenant les détailler.

3.2.2. L'ajout et la suppression de composants

La création d'une implémentation (composite ou native) est réalisée à l'aide d'une l'interface graphique (voir Figure 52). L'interface graphique contient deux parties, la première fournit la liste des interfaces (ou rôles) présentes dans la base de développement et la seconde la liste des implémentations de chaque interface. Dans l'exemple de la figure ci-dessous, l'interface *Compilateur* possède une implémentation composite *CompilateurV1*. La création (ou l'ajout) d'un composant se fait à l'aide de menus contextuels. Pour créer une implémentation (qui implémente directement une interface), il suffit de cliquer sur le nom de l'interface et à l'aide du bouton droit de la souris de sélectionner la méthode "*Implémenter*". Dans le cas de la création d'une implémentation composite raffinant une implémentation (existant dans la base), il suffit de se positionner sur l'implémentation composite à raffiner et ensuite de la même façon que précédemment de sélectionner cette fois la méthode "*Raffiner*". Dans l'exemple ci-dessous, on est en train de créer une implémentation composite qui raffine l'implémentation *CompilateurV1*.

L'intérêt de ces menus contextuel est double :

- ils permettent de ne proposer que des actions possibles sur chaque élément de la base. Il est impossible de créer une implémentation native avec la relation de raffinement par exemple,
- ils permettent aussi de rester indépendant du type de composant et de relation. En effet, l'interface graphique ne dépend pas des types de données.

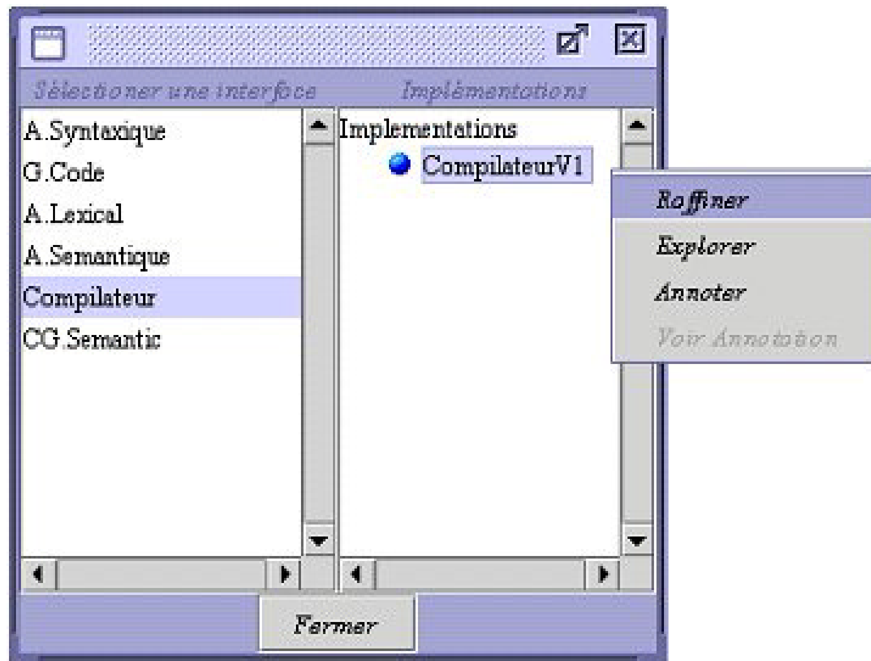

Figure 52 : Interface de création d'une implémentation

Dans le cas des implémentations composites, on utilise une autre interface qui permet de ne sélectionner que des éléments de la base comme occurrences. Et dans le cas des raffinements, on ne peut raffiner une occurrence que si celle-ci est une interface et on ne peut choisir que des implémentations de cette interface comme occurrence. De cette manière, nous assurons la cohérence de la base. Cette cohérence est complétée en même temps par le système de gestion des annotations, décrit dans la section 3.3.

3.2.3. L'exploration de la base

Nous offrons dans ce prototype la possibilité d'explorer la base de développement en suivant les différentes relations. La Figure 53 donne un exemple d'exploration des occurrences de l'implémentation composite *CompilateurV1*. A partir de cette vue, on est capable de voir par exemple, quelles sont les implémentations possibles que l'on peut utiliser pour raffiner les occurrences de *CompilateurV1* (du moins celle du type interface).

L'outil d'exploration permet de sélectionner les relations de création et de parcours qui nous intéressent. Dans l'exemple de la figure ci-dessous, on a choisit de ne pas visualiser la relation de raffinement.

Chacun des éléments de cette vue peut être visualisé indépendamment des autres, en double-cliquant dessus. Dans ce cas une fenêtre du même type s'ouvre avec comme racine l'élément double-cliqué.

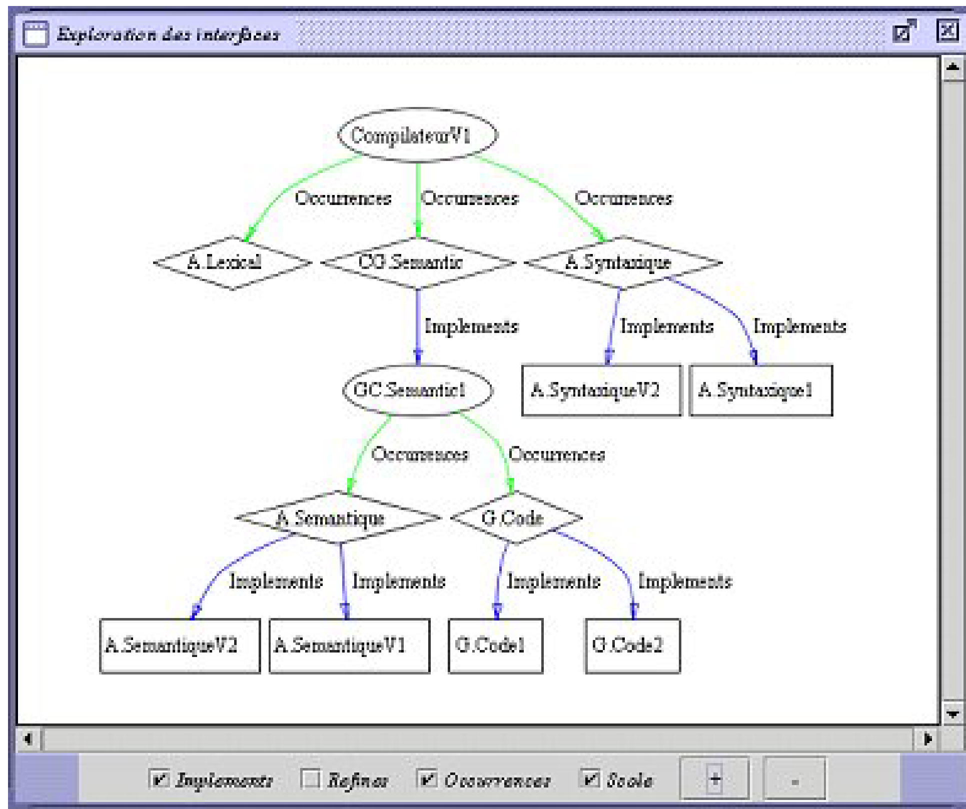

Figure 53 : Interface d'exploration des composants

3.2.4. Conclusion

On vient de voir un aperçu des possibilités de la partie gestion des composants du prototype de sélection. Ce qu'il faut retenir, c'est qu'elle est basée sur le modèle MVC et que l'on peut facilement rajouter de nouveaux éléments dans le modèle de composant et aussi rajouter de nouvelles relations de création et de parcours.

3.3. La gestion des annotations

Tout d'abord une annotation est un ensemble d'attributs (dont les valeurs sont uniquement des chaînes de caractères et des nombres).

Notre framework de sélection propose les fonctionnalités suivantes pour gérer les annotations des composants.

Dès qu'un composant est ajouté dans la base, il n'est validé que lorsqu'on l'a annoté. Dans notre implémentation, il est impossible d'ajouter un composant si son annotation est incohérente avec celle de son père (règle valable uniquement pour les implémentations et pas pour les interfaces).

Lorsque l'on crée une annotation, on choisit le type de l'annotation parmi un choix restreint (seuls les types d'annotations qui permettent de prendre en compte les contraintes de sélection associées à l'annotation du père sont prises en compte). L'utilisateur remplit ensuite les valeurs des attributs, sauf pour les attributs dont la valeur a déjà été calculée par une règle de

sélection. Dans certains cas, l'utilisateur doit choisir parmi un certain nombre de valeurs et dans d'autres il peut rentrer la valeur qu'il veut.

Une fois toutes les valeurs des attributs précisées, l'annotation est analysée pour vérifier sa cohérence avec l'ensemble des règles de sélection de l'annotation du père. S'il y a une incohérence, le composant est exclu de la base de développement. Sinon, l'utilisateur précise pour chacun des attributs de la nouvelle annotation les règles de sélection qu'il désire (ou qui lui sont imposées par l'annotation du père) et ceci pour chacune des relations de parcours et de création possibles à partir du type de composant que l'on vient d'annoter.

3.4. La construction des configurations

Pour construire une configuration, on commence d'abord par construire la requête à l'aide de l'interface de la Figure 54 (fenêtre de droite). Pour cela, on propose au configurateur de choisir parmi l'ensemble des attributs utilisés dans la base, ceux qui l'intéressent. Pour ces attributs, on lui propose un ensemble de valeurs possibles et il choisit parmi ces valeurs celle(s) qu'il désire.

La deuxième étape consiste à choisir l'application à configurer. Une application est représentée par une interface dans notre prototype. On lui propose donc (fenêtre de gauche) la liste des interfaces présentes dans la base (sauf celles qui n'ont pas d'implémentations).

Nous avons implémenté une solution basée sur l'algorithme de sélection présenté dans le chapitre 4. On récupère donc la liste complète de tous les composants qui correspondent à la requête et qui ont été atteints à travers les relations de parcours et de création. Nous avons ensuite une fonctionnalité qui permet de construire une configuration en faisant des choix parmi les différents composants sélectionnés.

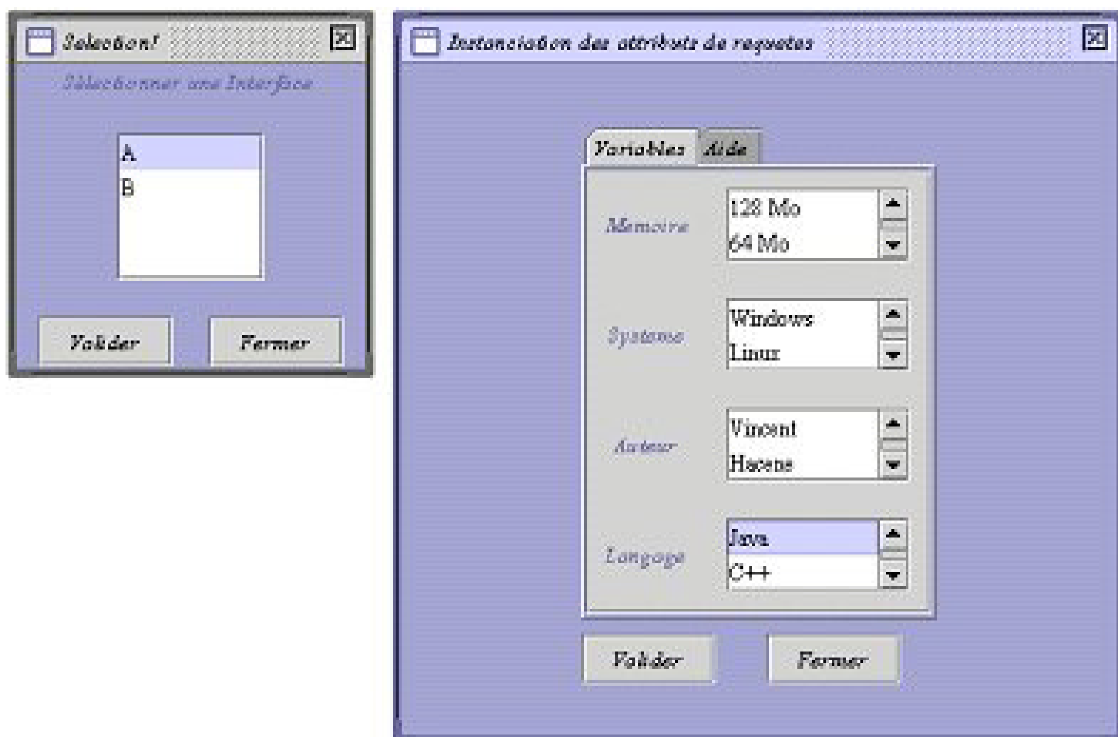

Figure 54 : Interfaces pour la sélection

Nous allons maintenant détailler les applications issues de l'activité de sélection (celle présentée ici ou une autre).

3.5. La description des applications

La description d'une application va être utilisée dans les différentes activités du cycle de vie du déploiement. Elle doit être adaptée à tout type d'application. Par conséquent elle est très générique et ne contient que des informations d'ordre général comme :

- le nom de l'application,
- l'identifiant,
- le numéro de version,
- les dépendances logicielles.

Il ne s'agit pas d'un modèle d'application mais d'une sorte de manifeste permettant de décrire une application contenue dans un "package". Cette description est ensuite ajoutée au "package". Cette description est basée sur un schéma XML [XML01] décrit en annexe 4. L'exemple ci-dessous correspond à la description d'une application nommée *scar*, dont l'identifiant est 8857, le numéro de version est 1.0.1 et qui n'a pas de dépendances logicielles.

```
<Manifest xmlns="http://castor.exolab.org/">
        <package>
                <name>scar</name>
                <id>8857</id>
                <version>1.0.1</version>
        </package>
</Manifest>
```

3.6. Conclusion sur la partie sélection

L'implémentation de l'activité de sélection a été réalisée dans le cas d'applications issues de notre modèle de composants. Cependant l'implémentation en étant basée sur le modèle MVC garantit que même si le modèle change (ajout/suppression d'éléments ou de relations), le prototype fonctionnera encore en tenant compte de ces modifications.

Il faut cependant noter que l'algorithme de sélection implémenté est assez rudimentaire et que de nombreux outils de gestion de configuration sont beaucoup plus évolués. Notre implémentation a cependant permis de valider notre approche en terme de sélection.

Nous allons maintenant décrire l'implémentation de l'activité d'installation en commençant par indiquer comment nous avons d'abord implémenté la gestion des procédés.

4. La gestion des procédés

4.1. Introduction

ORYA est un environnement de déploiement ayant comme fonction d'automatiser les activités de déploiement. Une partie de cette automatisation est réalisée à l'aide de la technologie des procédés. On a décrit dans le chapitre 6 de cette thèse (à travers l'activité

d'installation) notre vision des procédés basée principalement sur un méta-modèle et deux langages :

- le langage de description des procédés,
- le langage de réalisation des activités utilisées dans les procédés.

L'objectif de ce paragraphe est de présenter l'implémentation de ces langages (et plus particulièrement celui de réalisation) et de montrer comment à partir de ces descriptions on réalise réellement l'installation. Pour cela nous avons choisi de nous appuyer sur un exemple simple d'activité de déploiement : celui de l'activité de "transfert" appartenant au procédé décrit dans la section 5.

L'objectif de cette activité est de réaliser le transfert d'un "package" d'un serveur d'application vers un site. Le "package" doit être vérifié (vérification de la signature du "package", recherche de virus, etc.). Pour cela, le "package" doit être transféré sur le serveur de déploiement afin d'être contrôlé puis ensuite seulement transféré vers le site. Il y a donc 3 étapes à réaliser dans cette activité de transfert :

- le transfert du serveur d'application vers le serveur de déploiement,
- les vérifications sur le "package",
- le transfert du serveur de déploiement vers le site.

Nous allons maintenant décrire les implémentations de ces langages.

4.2. Le langage graphique de description

ORYA est une fédération de déploiement et la technologie des fédérations fournit un langage de description, un éditeur graphique et un moteur de procédé (APEL). Nous avons tout simplement utilisé ces outils pour générer (et exécuter) nos procédés de déploiement.

L'éditeur graphique permet de créer graphiquement les procédés (dont la Figure 55 est un exemple). Il offre les fonctionnalités de création et de suppression sur les éléments suivants :

- les activités,
- les ports,
- les flots de données,
- les produits.

Toutes ces manipulations sont gérées de façon cohérente, c'est à dire que le procédé final doit être cohérent. Par exemple, l'outil vérifie qu'un produit issu d'un flot de données et bien attendu dans l'activité de destination du flot de données (plus précisément dans l'un des ports d'entrée de l'activité). L'ensemble de ces vérifications permet d'assurer que les procédés ainsi créés sont corrects.

Reprenons l'exemple de l'activité de transfert. La Figure 55 propose une manière de la réaliser. La solution consiste à décomposer l'activité de transfert en deux sous activités. La première ("*GetFile*") a comme objectif de transférer le "package" du serveur d'application vers le serveur de déploiement. Elle dispose pour cela du nom du "package" et du nom du serveur d'applications. Ces informations lui sont fournies sous forme de deux produits ("*serveur Application*" et "*nomPackage*"). La deuxième sous-activité ("*PutFile*") a comme

objectif de transférer le "package" du serveur de déploiement vers la machine cible (le site). On aurait pu utiliser une solution basée sur trois sous activités, mais d'illustrer toutes les possibilités du langage.

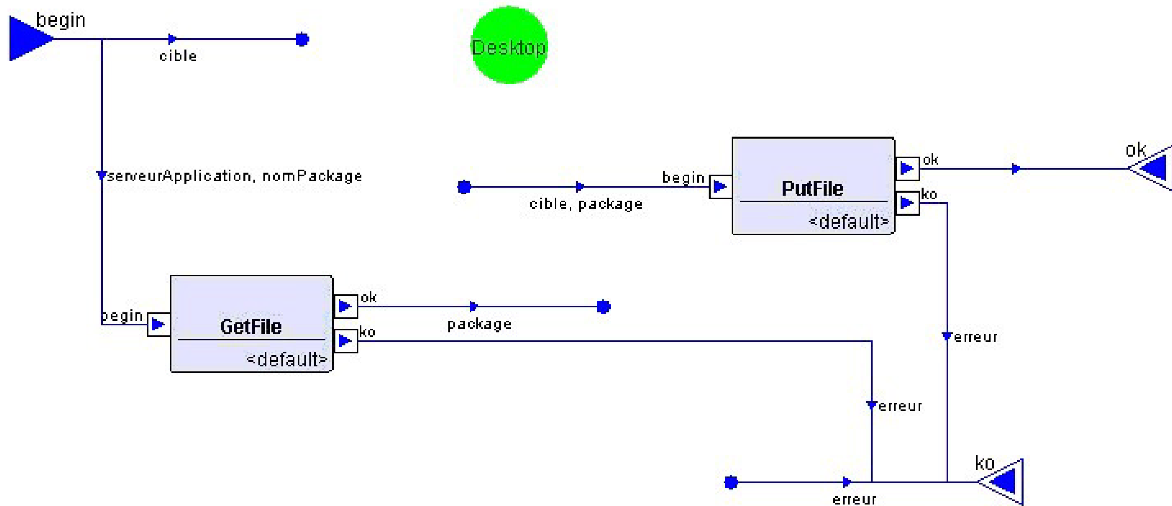

Figure 55 : Description de l'activité de transfert

Dans notre procédé, il n'y a pas de flot de données direct entre les deux sous activités. Le résultat de l'activité "*GetFile*" est envoyé vers le poste de travail de l'activité de transfert (considéré et géré comme un port dans Apel). Le "package" est ensuite envoyé (via un autre flot de données) vers la sous-activité "*PutFile*". Entre temps (conformément à la spécification de notre exemple) le "package" a été vérifié et analysé. En cas de problème ou dans le cas d'un "package" invalide, un produit nommé *erreur* est envoyé vers le port de sortie *ko* de l'activité de transfert.

A partir de cette description graphique, un fichier de description est généré. Il décrit l'ensemble des activités, des ports, des produits et des flots de données. Ce fichier sera utilisé par le moteur de procédés lors de l'exécution pour gérer l'enchaînement des activités et les flots de données.

Une fois l'activité décrite, il reste encore à décrire la partie réalisation de l'activité, c'est à dire comment les deux sous activités réalisent les transferts et comment l'activité globale de transfert vérifie le "package". Pour cela nous utilisons le langage de réalisation présenté dans la section 5 du chapitre 6. Nous allons maintenant décrire l'implémentation réalisée pour ce langage.

4.3. Le langage de réalisation

4.3.1. Introduction

Dans notre approche, chaque activité peut être associée à un fichier de réalisation. Ce fichier que l'on nomme *template* (extension .tpl) permet de décrire l'ensemble des actions à réaliser pour une activité.

La section suivante présente la structure d'un *template* et les conventions de nommage utilisées. Le langage et ses possibilités seront décrites à travers l'exemple de la réalisation de l'activité de transfert.

4.3.2. La structure d'un fichier .tpl

L'implémentation du langage de réalisation est basée sur la syntaxe de Java. Un fichier template est structuré de la manière suivante :

```
activity <activityName>
{
        components
        {
        }
        case <exp>:
        {
                (<inst>)*
        }
}
```

Chaque template correspond à la description de la réalisation d'une activité précise. On précise donc le nom de l'activité concernée : *activityName*. La réalisation peut nécessiter l'utilisation d'outils de notre environnement. On les déclare dans la partie *components* du template de la manière suivante :

```
<nom de l'instance>  : <nom du rôle> at <nom de la machine>
```

Par exemple l'instruction suivante signifie qu'un outil de type machine va être exécuté sur le serveur de déploiement (représenté par la variable globale $LOCAL_HOST).

```
deployServer : machine at $LOCAL_HOST;
```

Comme il est possible dans le même template d'exécuter deux outils de même type (rôle) sur deux machines distinctes, nous avons ajouté le concept de nom pour chaque instance. De cette façon, il est plus aisé d'utiliser l'outil par son nom et cela permet aussi de distinguer les outils entre eux.

La réalisation d'une activité peut se faire en plusieurs étapes selon le cycle de vie de l'activité et certaines des instructions ne doivent être exécutées que dans certaines conditions. Nous avons introduit le concept de blocs d'instructions et de triggers pour répondre à ces contraintes. Un template peut donc contenir plusieurs blocs d'instructions. Un bloc étant exécuté uniquement lorsque certaines conditions exprimées en termes d'état d'activité et de ports de sortie sont vérifiées. Par exemple, l'instruction suivante signifie que le bloc d'instruction A sera exécuté si l'activité associée au template passe à l'état "*Ready*" :

```
case $(this).isReady() :
{
        //bloc d'instruction A
}
```

Nous avons étendu le langage en implémentant un certain nombre de méthodes facilitant l'écriture des instructions. Par exemple, la méthode *terminate* permet de terminer une activité par un port donné en assurant le transfert éventuel des produits attendus par ce port de sortie.

Dans la spécification du langage de réalisation, il est indiqué que l'on peut manipuler des activités, des ports, des produits et des attributs. Afin de distinguer ces différents éléments, nous utilisons les conventions de nommage suivantes :

Instruction	Ce qu'elle représente
$this	L'activité courante (celle associée au template)
$activityName	La sous activité de l'activité courante appelée "activityName"
$activityName.portName	Le port "portName" de l'activité "activityName"
$activityName.portName.productName	Le produit "productName" du port "portName" de l'activité "activityName"
$activityName.portName.productName.attributeName	L'attribut "attributeName" du produit "productName" du port "portName" de l'activité "activityName"

Nous allons maintenant illustrer ce langage à l'aide de l'exemple de l'activité de transfert.

4.3.3. Le fichier GetFile.tpl

Dans cette activité, la réalisation nécessite l'exécution de deux rôles sur des machines différentes : celui de *transfert* sur le serveur d'applications et celui de *deployServer* sur le serveur de déploiement. L'information du nom de la machine sur laquelle se trouve le serveur d'application est obtenue à l'aide du produit *serverApplication* (qui se trouve dans le desktop) et de son attribut *name*. Il faut noter que nous avons considéré le poste de travail comme un port. Cela donne les instructions suivantes :

```
components
{
        transfer : transfer at $this.desktop.serveurApplication.name;
        deployServer : deployServer at $LOCAL_HOST;
}
```

La réalisation doit être exécutée lorsque tous les produits attendus par l'activité sont arrivés. Cela correspond au passage de l'activité à l'état Ready. On écrit donc :

```
case $(this).isReady() :
```

On remarque ici, que l'information sur l'état de l'activité est obtenue par l'appel de la méthode *isReady()* de la classe *Activity* du moteur de procédé. *$(this)* permet de faire référence à l'instance de la classe *Activity* représentant notre activité dans le moteur de procédés.

L'activité proprement dite est réalisée en utilisant l'une des méthodes de l'outil de transfert sur le serveur d'applications. Cette méthode *getFile*() nécessite comme paramètres le nom de l'application à transférer ainsi que son nouveau nom sur le serveur de déploiement. Ces informations sont obtenues soit à l'aide des attributs des produits du poste de travail :

```
$this.desktop.nomPackage
```

ou bien l'information est directement calculée :

```
String processInstanceName = $(this).getProcessInstance().getProcessId();
String tmpName = processInstanceName+".zip";
```

On récupère le résultat de l'exécution de l'outil. La suite de la réalisation dépend de ce résultat. En cas de problème lors du transfert, on doit créer un produit de type "*error*" en précisant à l'aide de l'un de ses attributs *msg* la cause de l'échec du transfert :

```
$this.desktop.error = $newProduct;
$this.desktop.error.msg = "pb pendant le transfert vers la fede";
```

On termine ensuite l'activité par le port "ko" de l'activité :

```
terminate($this,$this.ko);
```

Si le transfert a réussi, on crée un produit de type *package* (afin de représenter le "package" que l'on va manipuler dans la suite du procédé), on instancie aussi certains de ses attributs (*url* et *absoluteName*) de la manière suivante :

```
$this.desktop.package = $newProduct;
String url = deployServer.getSiteProperty("tmp.folder"+"\\"+processInstanceName);
$this.desktop.package.url = url;
$this.desktop.package.absoluteName = newTmpDir + "\\" +tmpName;
```

Et on termine ensuite l'activité par le port "ok" :

```
terminate($this,$this.ok);
```

La description complète du fichier GetFile.tpl se trouve dans l'annexe 5.

4.3.4. Le fichier Transfert.tpl

L'activité transfert est composé de deux sous activité : *GetFile* et *PutFile*. La partie réalisation d'une activité composite s'écrit de la même manière que celle d'une activité basique.

Les instructions sont semblables à celles du template précédent. Le template de l'activité PutFile n'est pas décrit dans ce chapitre. Les templates complets des activités Transfert et PutFile se trouvent en annexe 6 et 7.

On peut cependant expliquer la partie suivante du template de l'activité de transfert :

```
case $(this).isReady() && $(GetFile).isTerminated() && $(GetFile).getFinalExitPort() == $GetFile.ok :
{
        // bloc d'instructions
}
```

Cela signifie que le bloc d'instructions est exécuté si les conditions suivantes sont vérifiées :

- l'activité *transfert* est à l'état *Ready,*
- la sous activité *GetFile* a été terminée par son port "*ok*"

Il faut noter que dans la spécification du moteur de procédé utilisé actuellement l'activité *GetFile* ne peut passer de l'état *Init* puis *Ready* et enfin *Terminate*, que si l'état de l'activité transfert est *Ready*. L'expression précédente est donc superflue, mais elle permet d'illustrer

avec notre exemple, comment on a implémenter cette fonctionnalité du langage. Cette fonctionnalité est utilisée pour exprimer des conditions sur les états de sous activités.

4.3.5. Conclusion

Nous venons de voir à travers l'exemple de l'activité de transfert un aperçu des possibilités du langage de réalisation. Nous sommes donc capables maintenant de décrire graphiquement le procédé et de spécifier ensuite pour chaque activité du procédé comment la réalisation doit être faite. La section suivante décrit comment à l'aide de ses informations, le procédé est réellement exécuté.

4.4. L'exécution des procédés

4.4.1. Le principe

La description d'un procédé comporte donc les éléments de description suivants : un fichier de description du procédé et un ensemble de *templates*. On distingue deux parties (inter-connectées) lors de l'exécution d'un procédé :

- la gestion de l'enchaînement des activités et du transfert des produits,
- la réalisation de chacune des activités.

La gestion du procédé, c'est à dire la gestion du cycle de vie des activités et la gestion des flots de données est réalisée par un moteur de procédés fourni par la fédération. Nous n'allons pas dans ce chapitre détailler comment le moteur de procédés fonctionne, nous allons plutôt nous intéresser à la deuxième partie, c'est à dire comment à partir d'un template on arrive à interagir avec l'environnement de déploiement.

Il y a deux choses à résoudre pour pouvoir réaliser un procédé. La première consiste à interpréter le langage et la seconde consiste à être capable d'exécuter les bonnes instructions au bon moment.

La fédération utilise le concept d'aspect. Un aspect permet de décrire comment utiliser un outil de la fédération. On peut aussi spécifier textuellement quand on doit exécuter l'aspect. Afin de réutiliser toute l'infrastructure mise en place par la fédération pour la gestion des aspects, nous avons développé un traducteur de notre langage permettant de générer des aspects. Nous n'allons pas détailler ce traducteur mais la section suivante décrit sommairement les aspects et la façon dont on les utilise dans le cadre du déploiement.

4.4.2. Les aspects

Sans rentrer dans les détails de la technologie des aspects, il faut savoir qu'un aspect est associé à une méthode d'une classe de l'univers commun de la fédération. On rappelle que notre univers commun contient l'abstraction de notre environnement et en particulier les instances du modèle de procédé. Dans notre cas, nous avons associé tous nos aspects à la méthode *makeTransition*() de la classe *Activity*. Cette méthode est utilisée par le moteur de procédé pour changer l'état des instances de la classe Activity. Par exemple, elle est appelée lorsque l'activité tranfert passe de l'état *init* à l'état *ready*.

Comme les aspects sont associés à une classe et non à une instance de classe, l'ensemble des aspects peut être déclenché à chaque fois qu'une instance d'activité change d'état. Pour éviter cela, nous utilisons le nom de l'instance en question afin de n'exécuter que ses aspects. Dans l'exemple suivant, l'aspect n'est exécuté que lorsque l'activité Transfer change d'état.

```
when instance.getName( ).equals("Transfer")
{
        //Corps de l'aspect
}
```

Cette information n'est cependant pas suffisante, car la partie réalisation d'une activité n'est déclenchée que lorsque certaines conditions exprimées en terme de ports et d'état d'activité sont vérifiées. Si l'on reprend l'exemple de l'activité de transfert, on a la contrainte suivante :

```
case $(this).isReady() && $(GetFile).isTerminated() && $(GetFile).getFinalExitPort() == $GetFile.ok :
```

qui devient, dans le langage d'aspect :

```
when instance.getName( ).equals("Transfer") && instance.isReady()
        && ((ActivityComposite)instance).findActivity("GetFile").isTerminated()
        && ((ActivityComposite)instance).findActivity("GetFile").getFinalExitPort()
                == ((ActivityComposite)instance).findActivity("GetFile").getPort("ok") ;
```

Pour traduire nos *templates* en aspects nous avons développé un traducteur du langage de réalisation vers le langage d'aspect, puis utilisé le traducteur d'aspect (fourni par la fédération) pour transformer les aspects en classes Java (qui seront ensuite utilisées par la fédération).

4.5. Conclusion

ORYA propose donc toute une panoplie d'outils et d'interpréteurs (pour la plupart issus des fédérations) pour décrire un procédé en termes d'enchaînement d'activités, pour décrire ensuite la réalisation du procédé (via le langage de réalisation) et enfin pour exécuter réellement les procédés.

A l'heure actuelle, ORYA est en cours d'évolution pour implémenter et tester en grandeur réelle la création et l'exécution de ces descripteurs. Par conséquent nous allons dans le chapitre suivant, décrire la version stable d'ORYA qui propose une approche assez voisine pour la partie réalisation mais sans utiliser le langage. Nous allons décrire le procédé d'installation développé dans le cadre de notre coopération avec la société Actoll et la manière dont nous avons implémenté la réalisation.

5. Un exemple de procédé d'installation

5.1. Introduction

Nous allons dans cette section décrire le procédé de déploiement développé dans le cadre de notre coopération avec la société Actoll. Après l'analyse des différentes installations réalisées par cette société, nous en avons déduit un scénario d'installation commun à toutes ces installations. A partir de ce scénario, nous avons développé un procédé d'installation générique (capable d'installer au minimum toutes les applications d'Actoll). Ce procédé est décrit dans la section 5.3.3 de ce chapitre. Avec ce procédé nous avons imposé une sorte de

canevas pour l'installation. Cela signifie que toutes les installations doivent respecter le même scénario, qui consiste en une étape d'extraction, un ensemble d'étapes d'installation et une étape de fin de procédé.

Qu'est ce qui distingue une installation d'une autre ? En supposant que les installations respectent notre scénario, la différence entre les installations se fait essentiellement au niveau du nombre des étapes d'installations. Comme notre procédé doit fonctionner pour toutes les applications, nous avons utilisé le concept de boucle. La Figure 56 illustre ce concept.

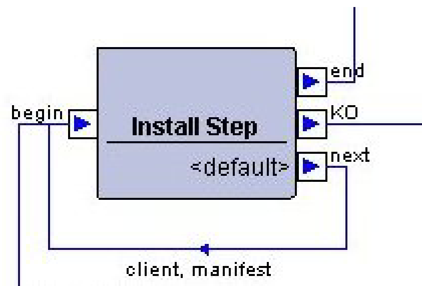

Figure 56 : Un exemple d'activité qui boucle

De cette manière, nous sommes capables dans notre procédé d'exprimer (simplement) que l'installation est constituée de une ou plusieurs étapes d'installation. Il faut maintenant décrire quelque part (pour chaque application) le nombre d'étapes d'installation, ainsi que la réalisation associée. Pour cela nous avons introduit un fichier de description d'installation (voir la section 5.2). Nous avons aussi remarqué (et imposé de fait) qu'une étape d'installation peut être vue comme l'exécution d'un outil suivie d'un certain nombre de vérifications associées. Nous allons maintenant décrire ce descripteur d'installation (section 5.2) et ensuite le procédé d'installation lui-même (section 5.3.3).

5.2. Le descripteur d'installation

5.2.1. Le format de description

Comme on vient de le dire, une installation doit suivre un canevas précis. On retrouve donc cette contrainte dans la structure même du descripteur d'installation, qui comporte les éléments suivants :

- une étape d'extraction,
- un ensemble d'étapes d'installation,
- une étape de désinstallation.

Ces différentes étapes sont toutes du même type : "*RoleType*". Cela correspond en fait à l'exécution d'un outil de la fédération ou plus précisément d'une méthode d'un outil de la fédération. Pour cela, on précise pour chaque étape les informations suivantes :

- le nom du rôle,
- le nom de la méthode,
- les paramètres de la méthode.

Les paramètres sont instanciés soit directement lors de la description, soit dynamiquement lors du déploiement. Pour cela on utilise le concept de propriété. Une propriété correspond à une variable dont la valeur est calculée lors de l'installation.

On peut associer à chacune de ces étapes un ensemble de vérifications. Ces vérifications sont du même type que les étapes ci-dessus; cela signifie que l'on utilise des outils pour réaliser ces vérifications. Le schéma du manifeste se trouve en annexe 8. Nous allons maintenant revenir sur ces différents éléments à travers un exemple simple.

5.2.2. Un exemple de description

Cet exemple a été écrit pour installer une application nommée *scarPatch320* (il s'agit en fait d'un petit script de mise à jour d'une application d'Actoll). L'installation est très simple, elle consiste à exécuter un script nommé *deploy.cmd*.

Notre installation va donc se dérouler en 3 étapes : l'extraction, l'exécution du script et l'étape de terminaison du procédé. Nous allons décrire chacune de ses étapes (en insistant plus particulièrement sur celle de l'extraction).

L'étape d'extraction consiste à utiliser un rôle de la fédération nommé *installer*, qui offre une méthode d'extraction *unzip*. Cette méthode nécessite deux paramètres : le nom du fichier zip et le répertoire d'extraction. Le nom du fichier est connu dès l'écriture du manifeste, on peut donc le préciser explicitement :

```
<param calculated="false">
        <value>scarPatch320.zip</value>
        <type>java.lang.String</type>
</param>
```

Par contre le répertoire d'installation n'est connu que lors du déploiement (il dépend de la machine cible). On utilise donc la propriété *SCAR.INSTALL_TEMP* :

```
<param calculated="true">
        <value>%SCAR.INSTALL_TEMP</value>
        <type>java.lang.String</type>
</param>
```

La valeur de la propriété est instanciée lors du déploiement. Pour cela on ajoute dans le manifeste la description suivante :

```
<property calculated="true">
        <name>SCAR.INSTALL_TEMP</name>
        <value>getTmpDir()</ value>
</property>
```

Cela signifie que la valeur de la propriété *SCAR.INSTALL_TEMP* correspond au résultat de la méthode *getTmpDir()* du modèle de site de la machine cible. Cette méthode est appelée au début du déploiement.

On peut associer à l'étape d'extraction des vérifications. Ces vérifications s'écrivent de la même façon que les étapes d'installations : il s'agit aussi de l'exécution d'une méthode d'un rôle de la fédération. Dans notre exemple, on utilise un outil qui vérifie l'existence d'un

fichier: on vérifie que le script à exécuter (*deploy.cmd*) a bien été extrait du "package". On remarque que l'on peut exprimer un paramètre en combinant des propriétés et des valeurs. Par exemple, le nom absolu de notre fichier est obtenu en concaténant le répertoire d'extraction (*%SCAR.INSTALL_TEMP*) avec le nom du fichier (*deploy.cmd*).

On obtient donc pour l'étape d'extraction la description suivante :

```
<extract stepName="Extraction">
        <do>
                <roleName>installer</roleName>
                <methodName>unzip</methodName>
                <param calculated="false">
                        <value>scarPatch320.zip</value>
                        <type>java.lang.String</type>
                </param>
                <param calculated="true">
                        <value>%SCAR.INSTALL_TEMP</value>
                        <type>java.lang.String</type>
                </param>
        </do>
        <verification>
                <roleName>fileVerification</roleName>
                <methodName>existFile</methodName>
                <param calculated="true">
                        <value>%SCAR.INSTALL_TEMP+\\+deploy.cmd</value>
                        <type>java.lang.String</type>
                </param>
        </verification>
</extract>
```

Les autres étapes, celle de l'exécution du script et celle de terminaison sont décrites selon le même schéma. Leurs descriptions complètes se trouvent en annexe 9.

5.3. Le procédé d'installation

5.3.1. Avant Propos

Dans les descriptions qui vont suivre, nous allons décrire les activités de notre procédé d'installation [LB03]. Afin de simplifier la description, nous ne précisons pas à chaque fois les produits transférés via les flots de données, cette information étant explicitement décrite dans les figures du procédé. De le même façon, une activité qui réussie se termine (sauf indication contraire) par le port "OK" et si elle échoue par le "KO".

5.3.2. Les produits utilisés

Comme on l'a vu dans le chapitre 6, le méta-modèle des procédés contient le concept de produit dont certains types représentent des éléments de l'environnement de déploiement.

On retrouve, dans notre exemple de procédé d'installation, des types de produits liés à l'environnement :

- le type "machine" qui permet de représenter les machines (serveurs d'applications et sites) de l'environnement,
- le type "package" qui représente les "packages" à déployer,

- le type "manifeste" qui représente le descripteur d'installation

Il y a aussi des types de produits, qui sont propres au procédé comme :

- le type "error" qui permet de décrire les différents problèmes ayant lieu lors de l'installation
- le type "data" qui permet de fournir aux activités des données comme par exemple le nom de l'application à déployer

Chacun de ces produits disposent d'attributs qui seront utilisés lors de la réalisation des activités.

5.3.3. Le procédé global d'installation

5.3.3.1. La vision globale de l'installation

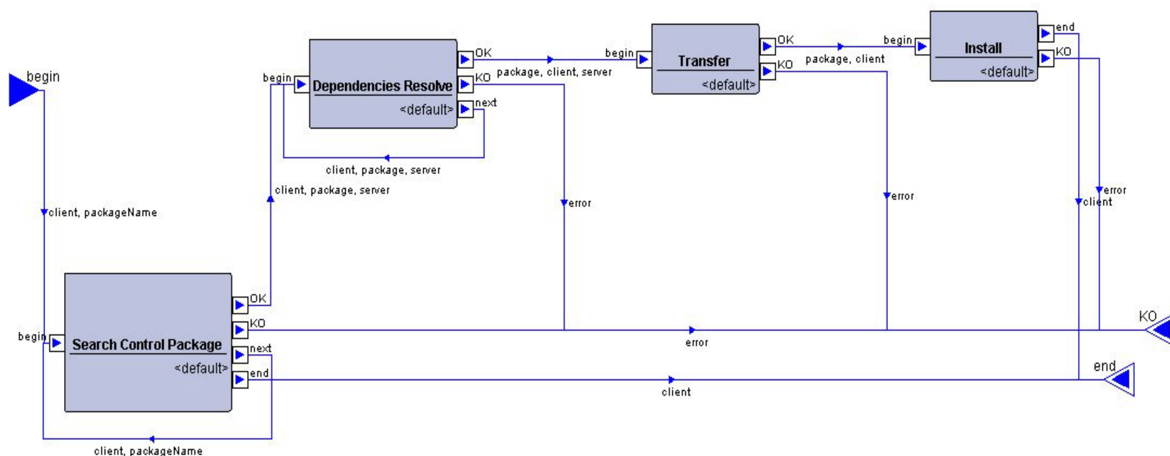

Figure 57 : Le procédé global d'installation

5.3.3.2. L'activité "Search Control Package"

Cette activité reçoit en entrée le nom de l'application à installer ainsi que la référence du site (produit "client" de type machine) sur lequel l'installation doit se dérouler. Son rôle consiste tout d'abord à vérifier que l'application n'est pas déjà déployée sur le site (sinon le procédé d'installation se termine par le port "end"). Dans ce cas, le serveur de déploiement recherche parmi les serveurs d'application connectés ceux qui disposent d'un "package" contenant l'application. Ensuite chacun de ces "packages" est vérifié, c'est à dire que l'on vérifie que la configuration et les contraintes associées sont cohérentes avec le site. Une fois qu'un "package" est conforme, on passe à l'activité suivante ("Dependencies Resolve") en terminant l'activité avec le port "OK". Si aucun serveur ne contient un "package" qui convient, l'activité se termine par le port "KO" et le procédé d'installation se termine aussi.

5.3.3.3. L'activité "Dependencies Resolve"

L'objectif de cette activité est de traiter les dépendances du "package" à déployer (celui fournit par l'activité précédente). L'information sur ces dépendances est extraite du fichier de description contenu dans le "package". Pour chacune de ses dépendances, on instancie et

exécute le procédé d'installation (celui que l'on est en train de décrire). Nous avons choisi de résoudre les dépendances une à une. Dès que l'installation d'une des dépendances échoue, le procédé d'installation se termine (port "KO"). On verra plus tard comment nous assurons le retour à l'état initial du site (c'est à dire la désinstallation des dépendances que l'on vient d'installer). Une fois le déploiement des dépendances réussi, on termine l'activité par le port "OK".

5.3.3.4. L'activité "Transfer"

On retrouve ici l'activité de transfert que nous avons décrite précédemment. Elle a pour rôle de transférer le "package" du serveur d'application vers le site.

5.3.3.5. L'activité "Install"

Cette activité est plus "complexe" que les précédentes. Dans le cadre de ce procédé nous avons choisi de décomposer l'installation (la partie physique) en un ensemble de sous-installations (nommées ici "Install Step") et d'étapes de désinstallations ("Undo Step").

Une étape d'installation consiste en l'exécution d'un script suivi d'un ensemble de vérifications. Un script peut être un fichier exécutable de type InstallShield ou bien l'exécution d'un outil de déploiement (comme un manipulateur de fichiers). Après chaque exécution, on peut exécuter un ensemble de vérifications pour s'assurer que l'exécution du script a bien fonctionné.

Si l'une des étapes d'installation échoue, le procédé permet un retour à l'état initial (celui avant le début de l'installation). L'activité de désinstallation n'est pas triviale car elle doit tenir compte de l'endroit où le déploiement a échoué afin de ne "désinstaller" que ce qui a été installé. Nous allons maintenant détailler l'activité composite d'installation.

5.3.4. L'activité composite d'installation

5.3.5. Le modèle de l'activité d'installation

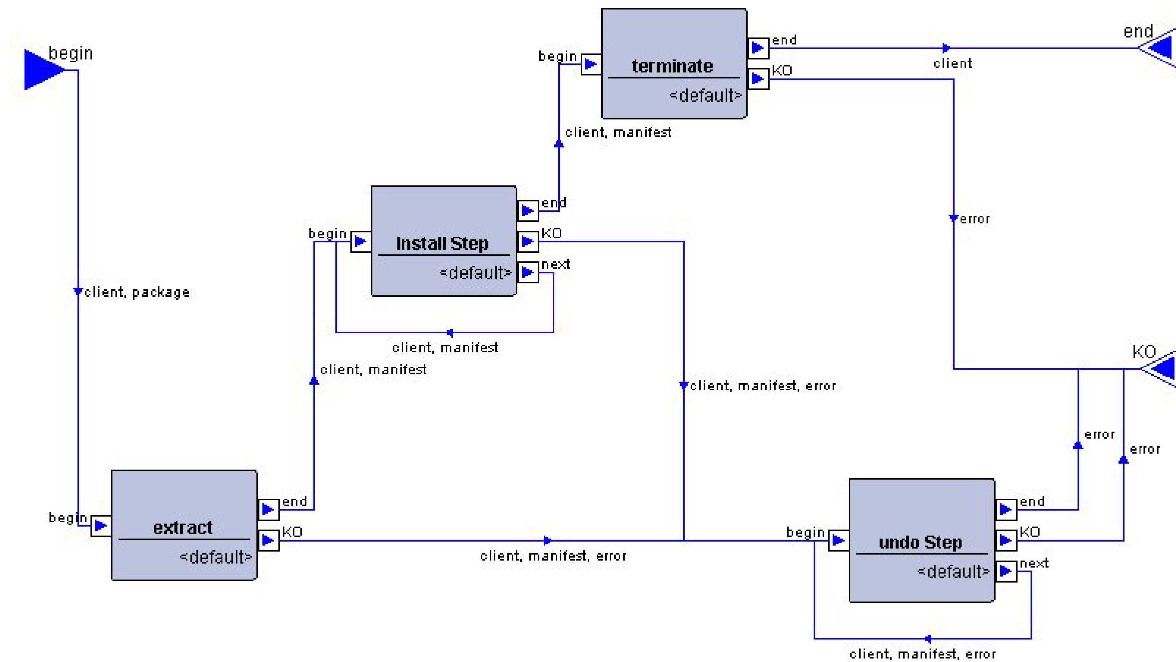

Figure 58 : L'activité composite d'installation

5.3.5.1. L'activité "Extract"

La Figure 59 décrit les différentes étapes de l'extraction. Cela consiste tout d'abord à extraire du "package" le manifeste (fichier décrivant les différentes étapes de l'installation). Cette sous-activité a lieu physiquement sur le serveur de déploiement, car le déploiement est géré et piloté à distance (depuis le serveur de déploiement et non depuis le site). Cela implique qu'une copie du "package" soit disponible sur le serveur de déploiement. Une fois le manifeste extrait, le "package" est extrait entièrement sur le site cette fois-ci. Cela consiste à dépackager le "package" dans un répertoire temporaire dont le chemin est calculé au moment du déploiement à l'aide du modèle de site. Il est possible ensuite de vérifier que l'extraction a bien réussi (vérification de l'existence de fichiers par exemple). Ces vérifications sont aussi extraite du manifeste.

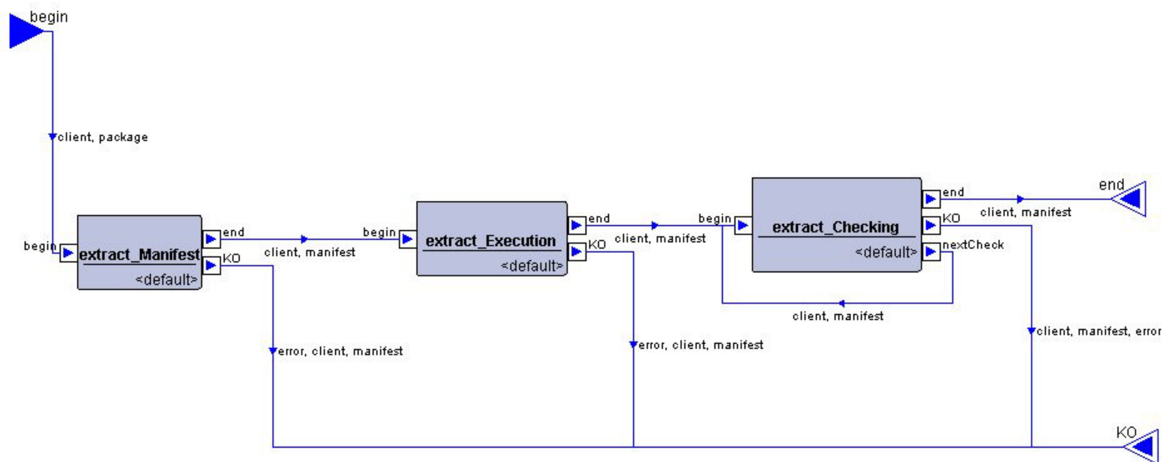

Figure 59 : La partie extraction de l'installation

5.3.5.2. L'activité "Install Step"

Comme on l'a dit précédemment, une installation est décomposée en plusieurs étapes. Chacune de ces étapes consiste en l'exécution d'un script ou d'un outil suivie (si nécessaire) d'un ensemble de vérifications. Sans rentrer dans les détails, on peut dire qu'après chaque vérification, on peut :

- soit passer à la vérification suivante (s'il y en a une de prévue) par le port "nextCheck",
- soit passer à l'étape d'installation suivante (s'il y en a encore une) par le port "nextStep",
- soit terminer l'installation par le port "end" si toutes les étapes ont été réalisées avec succès,
- soit arrêter le procédé d'installation et passer à l'activité de désinstallation "Undo Step".

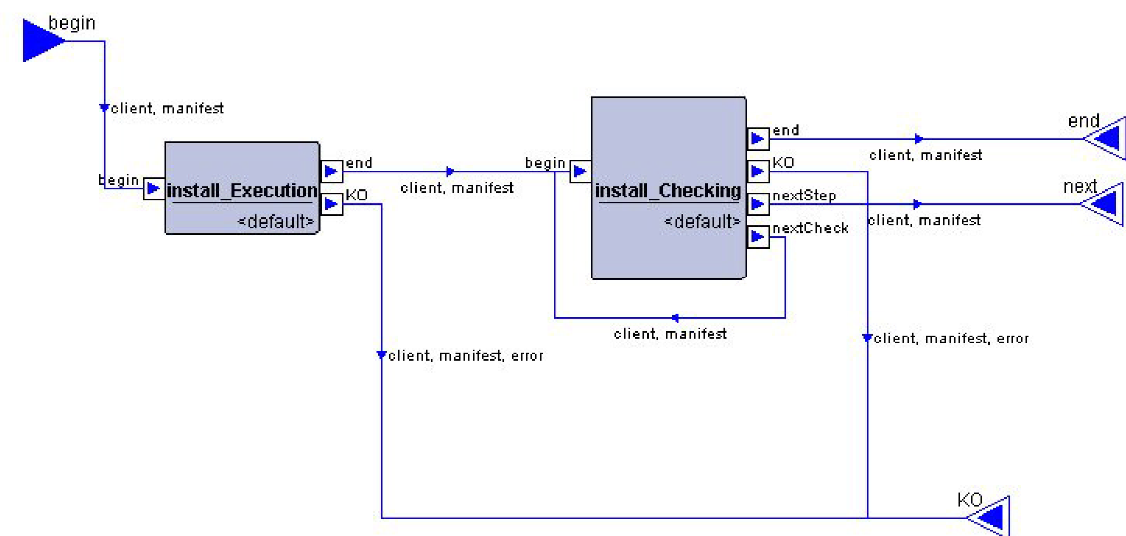

Figure 60 : Les différentes étapes d'installation

5.3.5.3. L'activité "Terminate"

L'activité "terminate" est exécutée lorsque l'installation de l'application a réussi. Elle consiste à sauvegarder le "package" (en vue des reconfigurations éventuelles) dans un répertoire spécifié par le modèle de site, puis à détruire physiquement du répertoire d'extraction le contenu du "package" et enfin à lancer la mise à jour du modèle de site.

5.3.5.4. L'activité "Undo Step"

L'activité de désinstallation consiste à enlever toutes les modifications qui ont réalisées lors du procédé d'installation (création de fichiers, modifications de registres, etc.). Elle a lieu lorsqu'une des étapes d'installation a échoué. Il est important de connaître le moment où l'installation a échoué car cela permet de n'enlever que ce qui a été ajouté.

5.4. Conclusion sur le procédé d'installation

Ce procédé d'installation a comme avantage de simplifier le travail du déployeur, car il n'a pas à créer explicitement le procédé d'installation. Il doit simplement spécifier le nombre d'étapes d'installation et pour chacune de ces étapes le rôle à exécuter ainsi que les vérifications associées. Comme l'écriture du descripteur d'installation est assez simple mais source d'erreurs (vu le nombre d'informations à préciser), nous avons développé un éditeur graphique d'aide à la conception des descripteurs. Cet outil est décrit dans la section 6.1.

Par contre il a un certain nombre d'inconvénients. Il ne fonctionne que dans le cas où l'on est capable de structurer l'installation sous formes d'étapes d'installation et il est difficile de faire évoluer le procédé (et surtout l'implémentation associée qui se charge d'interpréter le descripteur et d'exécuter les outils). C'est pourquoi nous avons développé le langage de réalisation qui va nous permettre de créer/modifier simplement les procédés (et la réalisation de ces procédés).

6. Les outils associés

6.1. L'éditeur de déploiement

Comme on l'a vu précédemment, le descripteur de déploiement devient vite source d'erreurs et sa rédaction se complique fortement lorsqu'il y a beaucoup d'activités. A titre d'exemple, le descripteur d'une des applications d'Actoll contenait :

- 5 propriétés,
- 5 étapes d'installation + celles d'extraction et de terminaison,
- 7 vérifications.

Bien que relativement simple comme installation a décrire, elle a été source d'erreurs. Nous avons pour éviter ce genre de problèmes développé un éditeur de déploiement (voir la Figure 61).

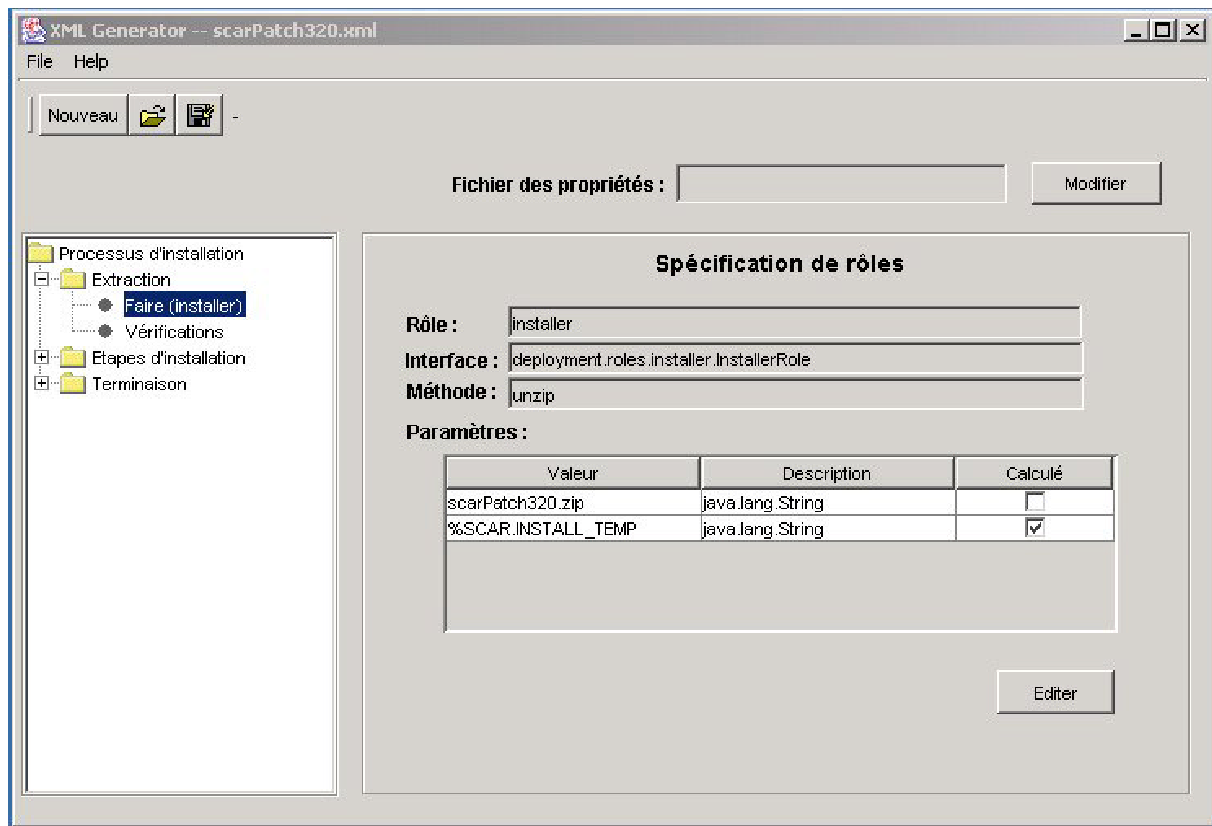

Figure 61 : L'interface graphique de l'éditeur de déploiement

Cet éditeur a les fonctionnalités suivantes :

- il permet de créer graphiquement le descripteur d'installation,
- il guide la création en fournissant la liste des rôles disponibles dans ORYA, leurs méthodes et aussi les paramètres exigés. Toutes ces informations sont documentées afin que l'utilisateur sache par exemple ce que fait réellement chacune des méthodes des outils.
- il vérifie la cohérence des informations spécifiées par l'utilisateur.

De cette façon, nous assurons que le descripteur ainsi créé est cohérent en termes de rôles et d'utilisation de rôles.

6.2. L'outil de packaging

L'outil de packaging offre les fonctionnalités suivantes :

- la capacité de packager un ensemble de fichiers dans un "package",
- la vérification de la présence de fichiers comme le descripteur d'installation ou le manifeste de l'application,
- la vérification de la cohérence entre le descripteur et les éléments du "package". On vérifie par exemple, que les scripts spécifiés dans le descripteur se trouvent bien dans le "package".

Cet outil est en cours de développement, l'objectif étant de fournir un outil complet permettent de lancer la création/modification du manifeste ou du descripteur d'installation et de réaliser les vérifications de cohérences du "package".

6.3. L'agenda

L'agenda est un outil de monitoring qui fait partie des outils associés au moteur de procédé Apel. L'agenda a essentiellement deux fonctionnalités :

- la réalisation manuelle des activités,
- le monitoring des instances de procédés en exécution.

Dans le cadre d'ORYA nous avons utilisé la deuxième de ces fonctionnalités : le monitoring. Il est basé sur un outil graphique (voir la Figure 62).

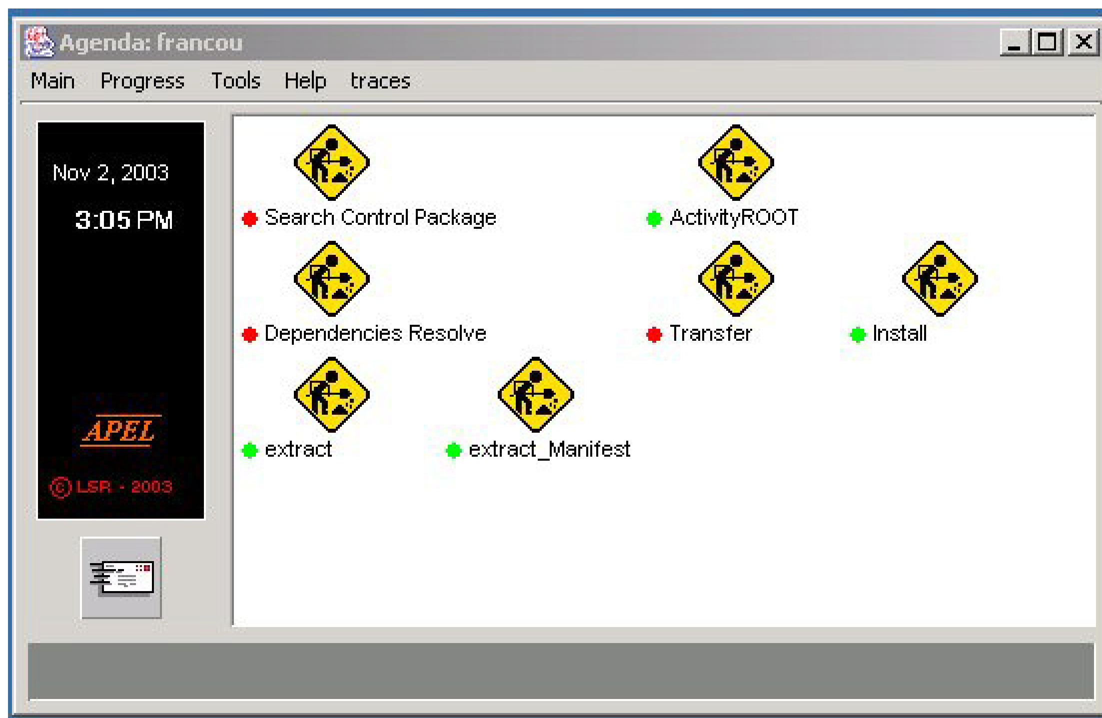

Figure 62 : L'outil de monitoring

A l'aide de cet outil, le déployeur est capable de visualiser en temps réel le (ou les) procédés d'installation en cours d'exécution. L'outil peut être aussi utilisé pour réaliser par exemple, les procédés en mode pas à pas (il suffit de déclarer les ports de sortie en mode non-automatique et ensuite le déployeur va valider chaque fin d'activité une à une). Dans le futur on pourra aussi utiliser l'agenda pour supporter la modification dynamique des procédés.

6.4. La gestion des traces du déploiement

La gestion des traces du déploiement dans ORYA intervient à plusieurs niveaux.

Il existe les traces dites de "débugage" de l'environnement (elle sont utilisées par le gestionnaire de déploiement). Elles apparaissent dans plusieurs des vues de l'outil de serveur de déploiement. Actuellement, il y a 2 vues qui ont été implémentées : celles qui concernent l'univers commun de notre fédération (il s'agit en fait d'une représentation de la connaissance de l'environnement de déploiement) et celles qui concernent l'utilisation des différents outils de déploiement d'ORYA dans tout l'environnement.

Un autre type de trace concerne l'exécution des procédés de déploiement. Nous avons implémenté plusieurs niveaux de vision parmi ces traces :

- une vision centrée sur le déploiement d'une instance. On y retrouve tous les commentaires décrivant l'exécution (succession des activités, création des produits) et les messages d'erreurs éventuels. La Figure 63 montre un exemple de ces traces.

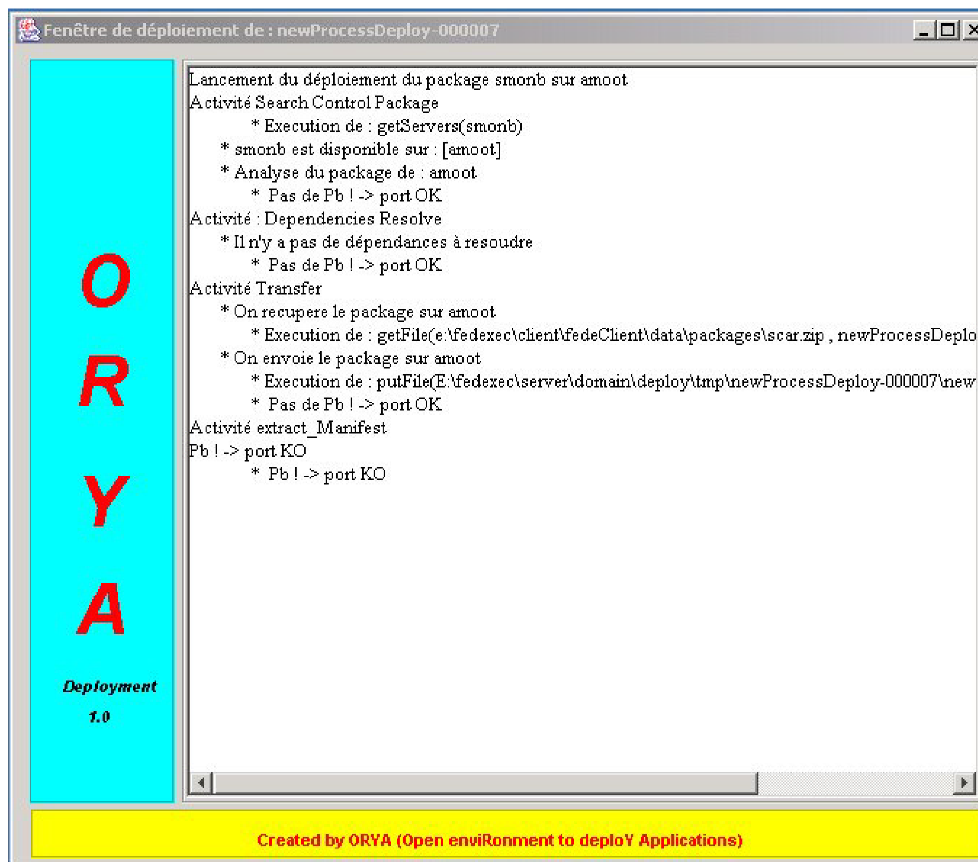

Figure 63 : Exemple de traces d'un déploiement

- une vision centrée sur les outils de déploiement. Chaque outil de déploiement a son propre gestionnaire de trace. Nous avons implémenté un outil (voir la Figure 64) permettant au déployeur de visualiser simplement (par un simple clic) les traces de n'importe quel outil exécuté dans l'environnement (les fichiers de traces étant transférés physiquement sur le serveur de déploiement).

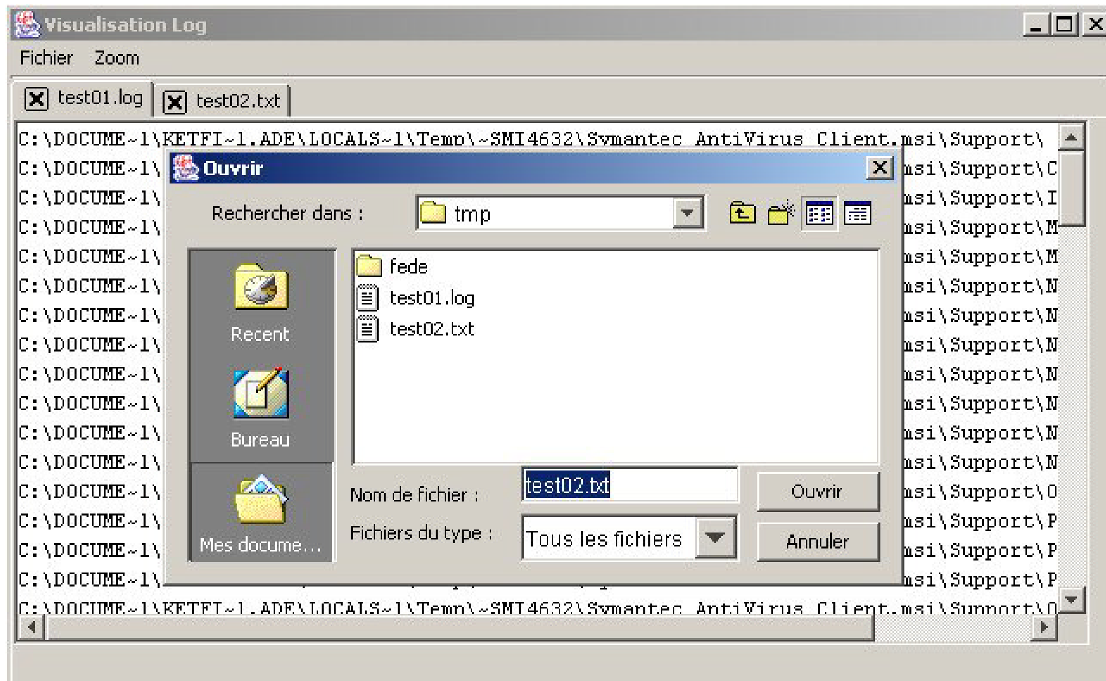

Figure 64 : Outil de visualisation des traces

7. Conclusion sur l'implémentation

Nous avons dans ce chapitre décrit l'implémentation d'ORYA concernant les activités de sélection et d'installation. Cette dernière activité est en cours d'évolution, afin d'utiliser pleinement les avantages du langage de réalisation. L'objectif est de pouvoir continuer à utiliser le procédé d'installation décrit dans la section 5.3 mais en utilisant les templates et non plus le fichier de description et son interpréteur. De cette façon, nous serons capables aussi de faire évoluer simplement ce procédé.

Nous allons dans le chapitre suivant décrire l'expérimentation réalisée dans le cadre de notre coopération avec la société Actoll et en déduire une première évaluation de notre approche dans un contexte industriel.

Chapitre VIII

Expérimentation et validation

1. Introduction

Nous venons dans les précédents chapitres de présenter notre approche à travers l'environnement de déploiement ORYA [ORYA]. Cet environnement de déploiement a servi de support à l'étude du déploiement dans le cadre du projet Centr'Actoll [Centractoll]. Centr'Actoll est un système informatique dédié à la gestion des péages des réseaux de transport urbain et interurbain. Ce système fournit à l'exploitant du réseau de transport les fonctionnalités suivantes :

- la représentation des composantes du réseau : topologie, tarifs,…
- la gestion clientèle : gestion des comptes clients et les services d'information à l'usager,
- la fonction de compensation financière entre l'usager et les banques,
- des moyens d'information et de transactions utilisables par les clients.

Actuellement, les applications dédiées au monde des transports et des péages évoluent rapidement en raison de l'apparition de nouveaux services, de l'évolution technologique, etc.. L'un des objectifs de ce projet (qui nous concerne le plus) est de proposer une solution pour le déploiement de logiciels sur des réseaux étendus et complexes. C'est en partie dans le cadre de ce projet, que nous avons expérimenté notre environnement de déploiement.

Ce chapitre se structure de la manière suivante. La section 2 décrit l'application Centr'Actoll. La section 3 décrit les différentes exigences en terme de déploiement. La section 4 décrit les différentes étapes de l'expérimentation réalisée en collaboration avec la société Actoll. La section 5 décrit les différents scénarios de déploiement proposés par Actoll et qui servent à valider ORYA. Nous terminons ce chapitre par un bilan de l'expérimentation et de la validation de l'environnement de déploiement.

2. L'application Centr'Actoll

2.1. Description de l'application

L'application Centr'Actoll est une application composée d'un ensemble de composants [LMKB02]. Ces composants sont des applications de type client/serveur. En plus de ces composants, l'application dispose d'une infrastructure de communication.

L'application peut être installée sous plusieurs configurations possibles. Chacune de ces configurations est composée de l'ensemble des composants de l'application ou bien seulement d'un sous-ensemble. Une version de l'application est donc formée de 1 à 7 composants, parmi les composants suivants :

- le référentiel (SREF),
- le serveur de configuration (SCONF),
- le serveur pour la gestion clientèle (SCAR),
- le serveur pour la personnalisation (SPER),
- le serveur monétique (SMON),
- le serveur d'analyse des données (SAND),
- le serveur d'inter modalité (SIM).

Chacun de ces composants existe sous une ou plusieurs versions. Par exemple le composant monétique (le Smon) a été implémenté en 3 versions différentes. On peut donc dire que l'application Centr'Actoll correspond à une famille de logiciels et que par conséquent elle va nous permettre de tester et de valider l'environnement de déploiement. La Figure 65 propose une schématisation des relations entre les différents composants de l'application. On peut remarquer que les composants ont des dépendances fortes entre eux.

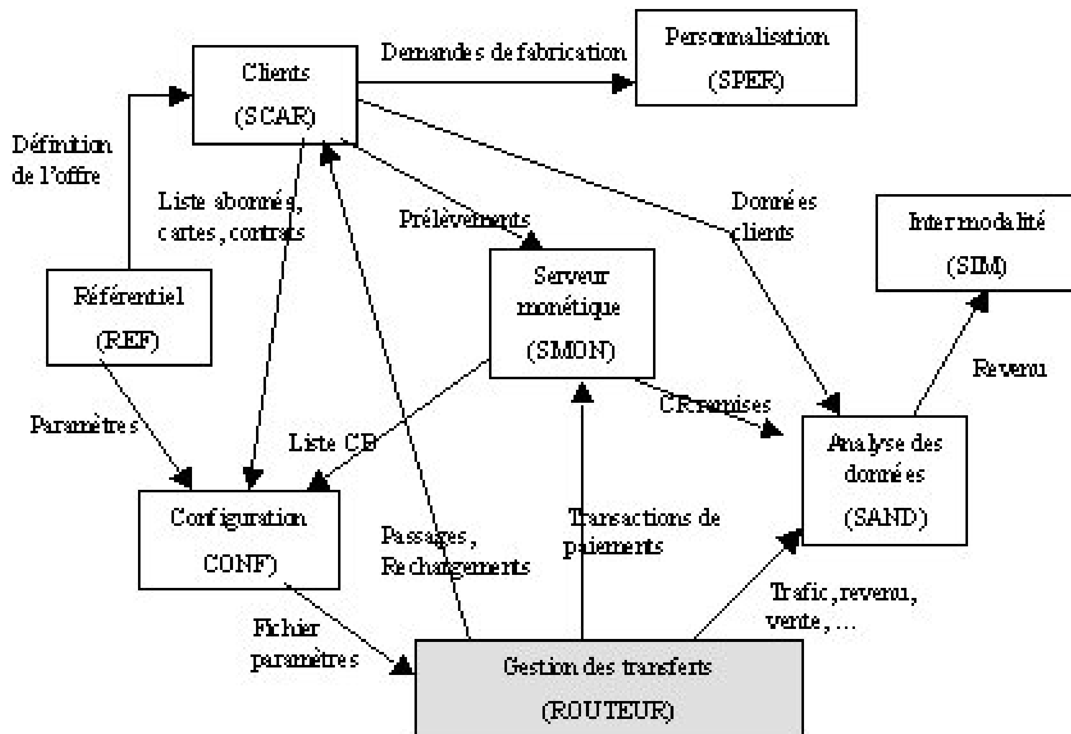

Figure 65 : Schéma général de l'application Centr'Actoll

Nous allons maintenant décrire l'infrastructure matérielle (la plate-forme) sur laquelle nous avons (en collaboration avec la société Actoll) expérimenté et validé l'environnement de déploiement.

2.2. La plate-forme de démonstration

Dans le cadre de ce projet, nous avons expérimenté le déploiement sur une plate-forme de démonstration construite au sein de la société Actoll. Elle est composée de six machines physiques (voir la Figure 66). Si l'on essaye de faire le lien avec notre approche, on peut dire que cette plate-forme comporte un serveur de déploiement (*ILLAMPU*) et cinq machines cibles (*CHARTREUSE*, *VERCORS*, *DEVOLUY*, *BEAUFORTIN* et *HERRON*). La machine nommée Vercors joue un rôle particulier, elle a pour fonction d'héberger les parties client de chacun des composants de l'application. Chacune de ses machines a ses propres contraintes matérielles et logicielles (non détaillées ici) qui doivent être prises en compte lors du déploiement.

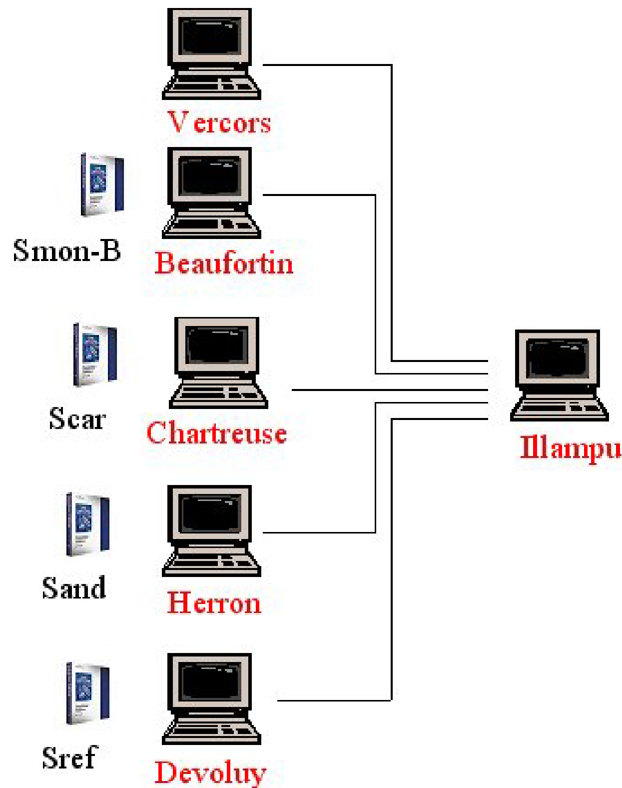

Figure 66 : Architecture physique de la plate-forme

Ces machines se trouvent sur le même réseau local : celui d'Actoll. Nous allons maintenant détailler les exigences en terme de déploiement imposées par Actoll.

3. Les exigences liées au déploiement

3.1. Introduction

Dans le cadre du déploiement la société Actoll a imposé des exigences de plusieurs types que nous avons essayées de classifier. Elles correspondent toutes aux problématiques introduites dans le chapitre 2 de cette thèse. Ces exigences (ou contraintes) sont issues soit des personnes d'Actoll ou bien de leurs clients. Nous allons maintenant détailler ces exigences.

3.2. La configuration

L'application doit pouvoir être déployée sous toutes ses configurations qu'elles soient logiques ou matérielles. L'application est composée principalement d'un ensemble de serveurs répartis sur un ensemble de machines. On doit donc être capable de déployer l'application de façon distribuée sur plusieurs machines.

3.3. La distribution

Le déploiement doit être possible soit via Internet (en tenant compte de la présence des pare-feux éventuels) ou soit via un réseau local.

3.4. L'automatisation

Chaque client a ses propres exigences en terme de sécurité (voir la section 3.7); cela a des répercussions sur le niveau d'automatisation du procédé de déploiement. En effet, un client peut exiger de valider chaque étape du déploiement et dans ce cas l'automatisation à 100% du procédé de déploiement n'est pas envisageable. On doit donc avoir plusieurs niveaux d'automatisation.

3.5. Le monitoring

Lors du déploiement, toutes les actions doivent être enregistrées. C'est à dire que le procédé de déploiement doit garder la trace de toutes les actions qu'il a réalisées. De plus, le producteur doit être capable d'observer l'évolution de l'activité de déploiement afin de pouvoir éventuellement intervenir en cours d'exécution (en cas de problème par exemple). Et enfin, le producteur et le client doivent connaître l'état de l'environnement (où a été déployé l'application) après chaque activité liée au déploiement (installation, mise à jour, etc.).

3.6. Le "roll back"

Si le déploiement échoue et que l'application n'a pas pu être installée chez le client, la société Actoll (le producteur) doit assurer le retour à l'état initial de l'environnement du client. En d'autre termes, les machines du client doivent retrouver leur configuration logicielle d'avant le début du déploiement.

3.7. La sécurité

Déployer une application implique d'une part de respecter les contraintes de sécurité exigées par les clients (ceux chez qui l'application va être installée) et d'autre part d'assurer la confidentialité des données de l'application. Ce dernier cas impose de sécuriser les étapes de transfert du producteur vers le client pendant les différentes activités du déploiement.

3.8. Les tests et vérifications

Dans le cadre du déploiement un certain nombre de tests doivent être effectués avant le déploiement. En effet, le déploiement d'un serveur prend en général plusieurs heures (durée due à la création de base de données Oracle), et il est gênant de devoir arrêter le déploiement en cours car il manque un fichier ou bien parce qu'une information (utilisée lors du déploiement) est fausse. Pour éviter cela, le déploiement doit être testé auparavant. Des vérifications doivent être effectuées pendant et après le déploiement pour vérifier soit le bon déroulement du déploiement, soit le résultat final de l'activité réalisée.

3.9. Conclusion

La plupart de ces exigences correspondent aux problématiques liées au déploiement introduites dans le chapitre 2. Elles permettent de confirmer qu'elles sont réellement au cœur des préoccupations des professionnels, notamment pour ce qui est de la sécurité et de la volonté du producteur de "gêner" le moins possible ses clients ("Roll back", monitoring, etc..).

4. L'expérimentation

4.1. Introduction

La phase d'expérimentation a duré plus d'un an (elle doit s'arrêter en décembre 2003). Elle a servi de support aux différentes étapes de développement qui ont abouti à l'environnement de déploiement ORYA. Nous allons dans cette section décrire les différentes étapes de cette expérimentation. Chacune de ces étapes correspond à une avancée importante dans notre étude du déploiement.

La première étape (voir la section 4.2) a consisté en l'analyse des besoins en terme de déploiement de la société Actoll. En conclusion de cette analyse, nous avons proposé une approche permettant de déployer de façon automatique certains des composants de l'application Centr'Actoll.

Dans la seconde étape nous avons implémenté et testé (en collaboration avec Actoll) l'approche proposée dans l'étape précédente. A la fin de cette seconde étape, un bilan a été dressé et de nouveau objectifs de déploiement ont été fixés.

Dans l'étape 3, nous avons expérimenté ces nouveaux objectifs à l'aide de la première version d'ORYA. Cette version utilise le procédé d'installation décrit dans le chapitre précédent. Nous sommes ensuite passés à la dernière étape qui consiste à implémenter puis expérimenter principalement le langage de réalisation décrit dans le chapitre 6. Cette phase étant en cours de développement, elle n'a pas pu être encore ni expérimentée, ni validée, c'est pourquoi nous n'en parlons pas dans ce chapitre.

Nous allons maintenant détailler les 3 étapes réalisées de l'expérimentation.

4.2. Analyse des besoins

4.2.1. Objectifs

La première étape de l'expérimentation a consisté à analyser les besoins pour déployer l'application Centr'Actoll. Pour cela nous avons étudié l'application et son déploiement (réalisé de façon manuelle). L'objectif de cette étude était de définir les différentes activités à réaliser pour installer ces 2 composants.

4.2.2. Réalisation

Nous avons commencé par étudier l'application Centr'Actoll et ses composants. Nous en avons tiré une modélisation de ces composants et proposé un modèle d'application permettant de décrire chacun des composants.

Ensuite, nous avons étudié le déploiement manuel des composants à travers deux de ces composants : le *Scar* et le *Smon.* Nous en avons tiré un certain nombre d'enseignements sur la manière d'installer l'application. Pour chacun de ces 2 composants, nous avons décrit le scénario d'installation à l'aide de diagrammes de séquences comme celui de la Figure 67 qui correspond à celui du *Scar*.

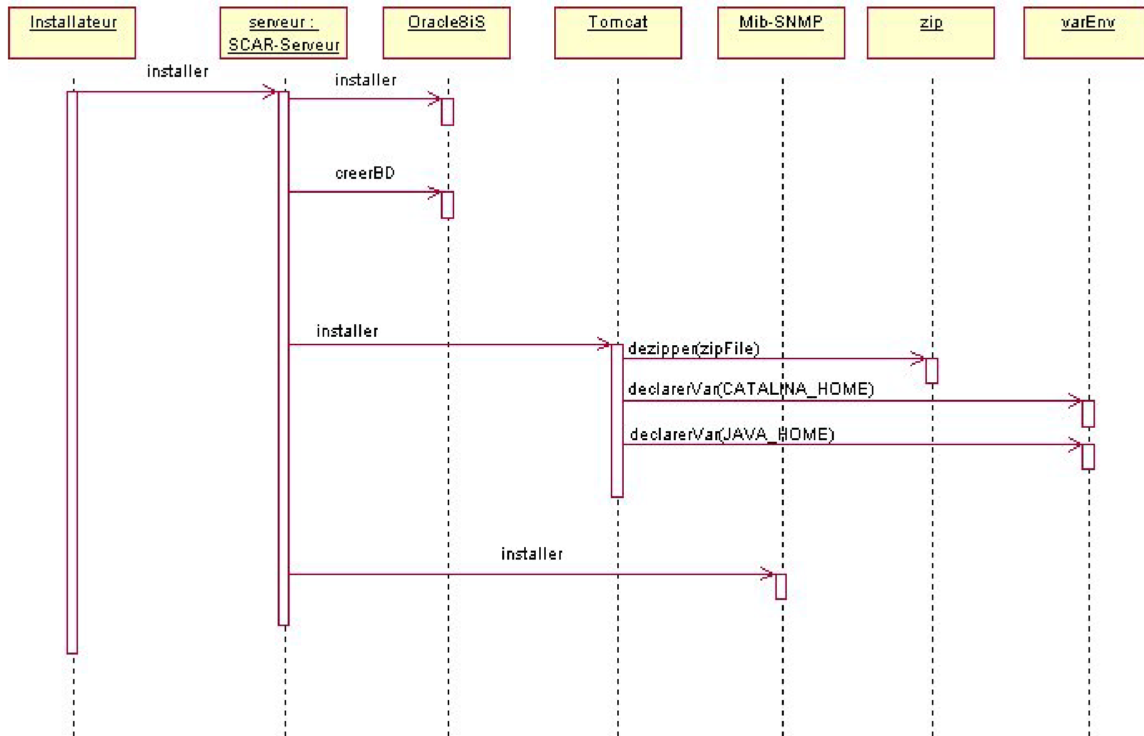

Figure 67 : Exemple de diagramme de séquence pour l'installation du Scar

4.2.3. Bilan

A la fin de cette phase d'expérimentation, nous avons conclu sur le fait que l'installation d'un composant pouvait être réalisée de façon automatique et à distance à l'aide de la technologie des procédés.

4.3. Un outil de déploiement spécifique

4.3.1. Objectifs

L'objectif de cette phase était de réaliser un prototype permettant d'installer à distance et de façon automatisée, deux des composants de l'application Centr'Actoll. Ce prototype devait permettre de vérifier que toutes les étapes d'installation identifiées lors de la phase d'analyse des besoins étaient automatisables.

4.3.2. Réalisation

Nous avons commencé, à l'aide de l'éditeur de procédé d'Apel, par décrire chaque installation en termes de procédés. L'approche a consisté à écrire un procédé d'installation par composant. Ensuite, nous avons réalisé le prototype permettant d'exécuter ces 2 procédés. Dans cette phase, nous avons implémenté la plupart des outils qui font maintenant partie d'ORYA (outil de transfert, de manipulation de registres/fichiers, etc..).

4.3.3. Bilan

Le résultat a été dans l'ensemble très positif, mis à part quelques petits problèmes liés pour la plupart aux variables d'environnement de Windows (qui ont été résolus lors des phases suivantes).

Une fois démontré, que l'automatisation est possible et surtout qu'elle apporte un gain de temps (plus besoin de se déplacer chez le client pour installer l'application) et de fiabilité (le procédé est décrit une fois pour toute) non négligeable à Actoll, nous avons décidé de spécifier un procédé d'installation capable de déployer n'importe lequel des serveurs.

4.4. Un outil de déploiement générique

4.4.1. Objectifs

L'objectif de cette phase était de réaliser un prototype basé sur l'exécution d'un seul procédé d'installation, capable (sans modification) d'installer tous les composants de l'application.

4.4.2. Réalisation

Pour réaliser cet objectif, nous avons étudié les installations des autres composants de l'application afin d'essayer de trouver un schéma commun. Nous avons identifié qu'une installation pouvait être décomposée en plusieurs étapes, chaque étape correspondant basiquement à l'exécution d'un script (ou outil) suivie de vérifications. A partir de ce constat, nous avons spécifié un procédé d'installation commun à tous les composants. Ce procédé est décrit dans le chapitre 7.

La deuxième partie de cette phase a consisté à implémenter ce procédé d'installation. Pour cela nous nous sommes basé sur la technologie des fédérations et nous avons développé la première version d'ORYA.

4.4.3. Bilan

Le bilan de cette phase de l'expérimentation est intéressante car cela a permis de valider le procédé d'installation ainsi que son implémentation dans un environnement de déploiement. Cet environnement a été complété en parallèle par un modèle de site, un modèle d'application et autres éléments et outils détaillés dans les chapitres 4 et 7.

4.5. Conclusion

Cette expérimentation nous a permis d'aboutir à l'implémentation de l'environnement de déploiement. Celui à l'avantage de fournir une solution concrète pour des problèmes réels de déploiement. Cependant cet avantage a aussi un défaut, c'est que la solution proposée risque d'être trop liée au contexte de l'expérimentation (le type d'application, la plate-forme, etc..). C'est pourquoi à la suite de cette expérimentation, nous avons essayé de diminuer ce lien en développant par exemple un langage de réalisation permettant de séparer (et de faire évoluer) le procédé d'installation utilisé par ORYA.

Cette évolution dans ORYA est actuellement en cours d'implémentation et sera suivie d'une phase d'expérimentation avec l'application Centr'Actoll. En parallèle à ces évolutions, nous

sommes passés à l'étape suivante qui consiste à valider l'approche (et ORYA). Pour cela, Actoll a proposé un certain nombre de scénarios de déploiement que nous allons décrire dans la section suivante.

5. La validation

5.1. Introduction

Cette section a pour but de décrire les différentes étapes proposées par la société Actoll pour valider l'environnement de déploiement ORYA. Cela consiste à réaliser un certain nombre de scénarios de déploiement [Pay03]. Chaque scénario correspond à une situation réelle à laquelle un producteur de logiciel est confronté tous les jours. Chacun de ces scénarios permet de vérifier (ou de valider) certaines des exigences de déploiement présentées dans la section 3. Ces exigences sont présentées ici sous la forme de contraintes.

Nous avons choisi de présenter les scénarios qui permettent de valider l'activité d'installation sous ORYA. Pour chaque scénario, nous décrivons le (ou les) objectif(s) et les contraintes à appliquer. Dans cette section nous utilisons le terme de composant pour désigner les serveurs de l'application Centr'Actoll. Il faut noter que tous ces scénarios se déroulent dans l'environnement de déploiement, c'est à dire que toutes les machines sont connectées au serveur de déploiement.

5.2. Déploiement d'un composant sur une machine

5.2.1. L'objectif

Pour réaliser ce scénario simple, on dispose d'un serveur d'application, d'un client et du serveur de déploiement symbolisant ORYA. La Figure 68 décrit l'architecture utilisée. Ces trois machines se trouvent sur le même réseau local. On ne traite donc pas les problèmes liées à la distribution via Internet par exemple.

L'objectif est d'installer sur le client (la machine) un des composants de l'application Centr'Actoll. Ce composant se trouve sur une des machines (le serveur d'applications) du réseau local. L'installation doit être totalement automatisée sans aucune intervention humaine.

L'objectif de ce scénario est de vérifier qu'ORYA permet dans des conditions particulières (réseau local, déploiement à distance) de réaliser l'installation avec au moins les mêmes performances qu'une installation manuelle exécutée directement de la machine du client (qui dispose pour cela d'un accès aux données du serveur d'applications). Pour cela un certain nombre de contraintes ont été imposées dans le scénario.

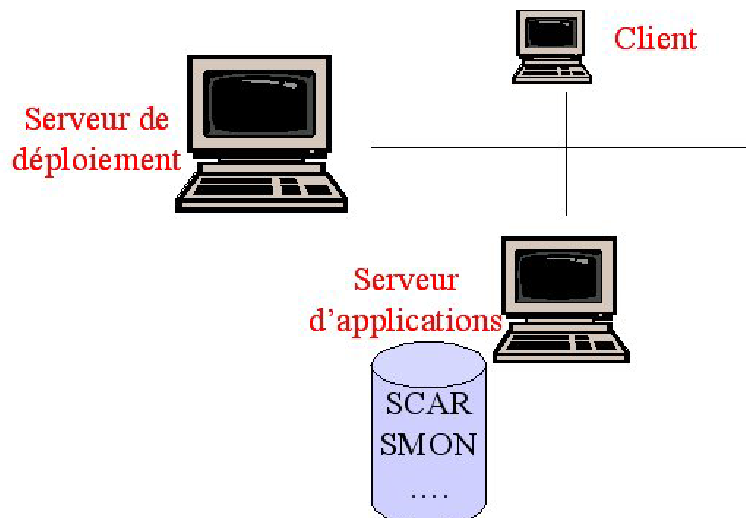

Figure 68 : Architecture du scénario 1

5.2.2. Les contraintes testées

Ces contraintes (de débit, de reprise et de disponibilité) ont été choisies par Actoll pour vérifier que le passage à un déploiement automatisé et réalisé à distance a été pris en compte par ORYA et qu'il n'influe pas sur les performances.

Nom de la contrainte	Commentaire
Contrainte de débit	En réseau local, le temps supplémentaire dû à l'automatisation de l'installation doit être faible par rapport à une installation manuelle
Contrainte de reprise	En cas de panne lors de l'installation, le système doit être capable de reprendre l'installation (sans recommencer à zéro) ou sinon d'assurer que la machine retrouve son état initial
Contrainte de disponibilité	La machine doit continuer de fonctionner normalement pendant l'installation du nouveau composant

5.2.3. Résultat

La durée de l'installation d'un composant est sensiblement la même en mode manuel ou en utilisant ORYA. La grande différence est qu'on n'a plus besoin d'avoir une personne devant la machine cible pour exécuter les différentes étapes et vérifications. En cas de panne, le procédé d'installation d'ORYA permet d'exécuter les différentes actions de reprise (ou de retour à l'état initial) spécifiées par le personnel d'Actoll, à condition qu'elles se présentent sous la forme d'exécutions d'outils. Lors de l'installation, la machine continue à fonctionner normalement mais l'installation (en mode manuel ou automatique) est coûteuse en terme de ressources.

5.3. Déploiement multiple d'un composant

5.3.1. L'objectif

Ce scénario se déroule toujours sur un réseau local. On dispose cette fois de deux machines clientes (*A* et *B*). La Figure 69 décrit l'architecture dans laquelle se déroule le scénario 2.

L'objectif de ce scénario est de tester le déploiement d'un même composant sur plusieurs machines en même temps ou l'un à la suite de l'autre.

L'intérêt de ce scénario est de vérifier qu'ORYA est capable d'exécuter plusieurs déploiements en même temps, sans qu'il y ait des interactions. Cela permet de vérifier que chaque exécution de procédé de déploiement est gérée de façon indépendante des autres exécutions.

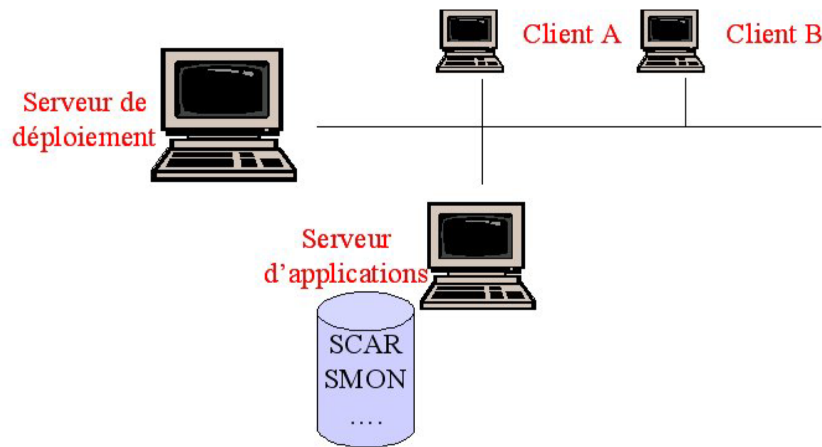

Figure 69 : Architecture du scénario 2

5.3.2. Les contraintes testées

Les contraintes imposées ici permettent de vérifier que le fait d'exécuter plusieurs installations en même temps est d'une part possible (débit) et d'autre part que les performances (et les fonctionnalités) des déploiements unitaires ne sont pas affectées lorsque le déploiement devient multiple.

Nom de la contrainte	Commentaire
Contrainte de plate-forme	On peut choisir au dernier moment le composant à déployer et sur les machines cibles.
Contrainte de débit	Les différentes installations doivent pouvoir être réalisées en parallèle.
Contrainte de cohérence	Si l'une des installations échoue, on doit pouvoir choisir entre désinstaller le composant sur toutes les autres machines ou bien continuer (si besoin) l'installation sur les autres machines

5.3.3. Résultat

ORYA permet de traiter le déploiement multiple, c'est à dire qu'il est possible d'installer plusieurs applications sur la même machine en même temps et/ou d'installer une ou plusieurs applications sur plusieurs machines en même temps. On peut donc réaliser actuellement l'installation d'une même application sur plusieurs machines en même temps. Par contre, nous n'avons (pas encore) implémenté la notion de planification du déploiement permettant de lier entre-eux plusieurs déploiement. Actuellement, les différents déploiement sont totalement autonomes (sauf dans le cas des dépendances), si l'une des installations échoue, les autres continuent.

5.4. Déploiement d'un composant et d'une dépendance

5.4.1. L'objectif

L'objectif de ce scénario est de tester le déploiement du composant *C1* sur une machine *A*. Cette application a une dépendance (le composant *C2*) qui doit être installée sur la machine *B*. L'architecture est donc la même que celle du scénario précédent (voir la Figure 69).

L'objectif de ce scénario est principalement de tester comment ORYA gère les dépendances (descriptions des applications) dans le cas d'applications distribuées.

5.4.2. Les contraintes testées

Les contraintes imposées ici ont pour but de vérifier qu'ORYA peut proposer différents scénarios dans le cadre de l'installation d'une application distribuée. Notamment lorsque l'un des déploiements échoue, que ce soit celui du composant principal ou celui de la dépendance.

Nom de la contrainte	Commentaire
Contrainte de cohérence	En cas d'échec de l'installation de la dépendance, l'installation du composant doit être arrêtée et la machine remise en état. En cas d'échec de l'installation du composant, on doit pouvoir choisir entre continuer à installer la dépendance, arrêter l'installation ou désinstaller la dépendance

5.4.3. Résultat

Dans la version d'ORYA qui utilise le procédé d'installation décrit dans le chapitre 8, en cas d'échec de l'installation d'une dépendance (du composant), l'installation du composant est arrêtée et la machine est remise en état. Dans la version future (celle basée sur le langage de réalisation), il sera possible de préciser la politique à suivre en cas d'échec.

5.5. Déploiement via Internet

5.5.1. L'objectif

L'objectif de ce dernier scénario est de tester l'installation lorsque le réseau utilisé n'est plus local mais Internet. L'objectif consiste à installer un composant sur une machine d'une entreprise cliente. On accède à cette machine via Internet. La Figure 70 décrit l'architecture de ce scénario.

L'intérêt de ce scénario est de vérifier d'une part qu'ORYA permet le déploiement à grande échelle (dont la taille du réseau est l'une des caractéristiques) et d'autre part de vérifier que l'approche proposée gère les problèmes posés par la présence de pare-feux.

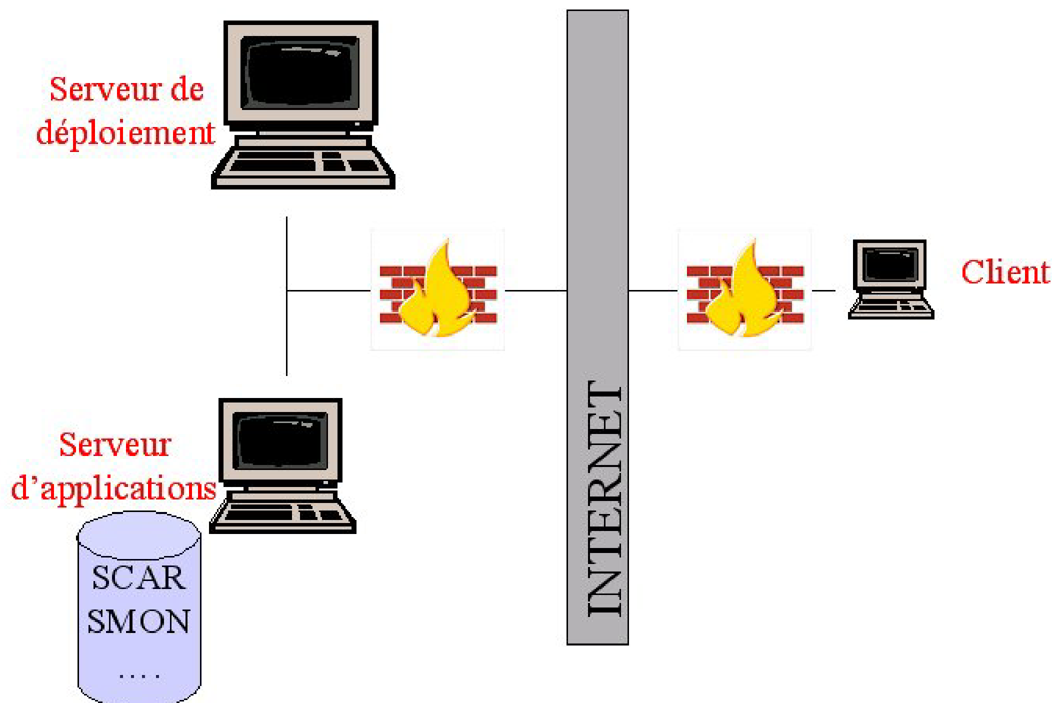

Figure 70 : Architecture du scénario 4

5.5.2. Les contraintes testées

Les contraintes imposées par Actoll dans ce scénario ont pour but de vérifier de quelle manière les performances (*débit*, *reprise*) d'ORYA en termes d'installation sont impactées lorsque l'on passe d'un réseau local à Internet. On doit vérifier aussi comment la *sécurité* des données est gérée.

Nom de la contrainte	Commentaire
Contrainte de sécurité	Le déploiement doit être réalisé malgré le filtrage d'adresses IP
Contrainte de débit	Le temps de déploiement ne doit pas être supérieur à celui lors du déploiement manuel (via Internet)
Contrainte de reprise	Cette contrainte est d'autant plus importante que le débit pouvant être faible, la communication peut être longue et le risque de coupure élevé.

5.5.3. Résultat

Pour réaliser ce scénario, nous nous sommes appuyés sur la technologie des fédérations, plus particulièrement sur l'infrastructure de communication. Elle permet d'exécuter à distance (via Internet) des outils sur des machines distantes même si celles-ci se trouvent derrière un pare-feu. Pour ce qui concerne la contrainte de débit, la solution choisie consiste à transférer en une fois toutes les données (et ressources) nécessaires pour l'installation. Par rapport à une installation manuelle, on doit ajouter le temps nécessaire pour le transfert mais ensuite on bénéficie des avantages de l'automatisation. Celle-ci est gérée à distance, mais elle est peu coûteuse en temps, car uniquement des informations en format texte sont transmises du serveur de déploiement vers la machine cible. En cas de coupure réseau, la fédération, qui assure la persistance des données du moteur de procédé, permet la reprise du procédé d'installation à l'endroit où il a été interrompu.

5.6. Conclusion

Comme on vient de le voir, cette phase de validation a pour objectif de tester l'environnement de déploiement dans plusieurs cas de figures. L'intérêt étant de vérifier que les exigences décrites dans la section 3 ont été prises en contre dans la réalisation d'ORYA.

Certains de ces scénarios ont pu être testés, d'autres sont en cours de validation (principalement le test basé sur Internet). Le résultat actuel de cette validation est qu'ORYA peut être utilisé pour installer les composants de l'application de façon plus fiable que l'installation manuelle à condition que les descriptions de ces installations soient bien sûr correctes. C'est pourquoi un gros travail de vérification et de validation a été réalisé au niveau des outils de création de ces descriptions.

6. Bilan

Cette collaboration avec la société Actoll sur la partie déploiement nous a permis pendant plus d'un an de nous confronter à un cas industriel de déploiement d'application (surtout de la partie installation du déploiement). Cet échange a permis de mettre en place un processus de développement basé sur l'analyse, la réalisation puis la validation d'un certain nombre d'idées. De cet échange sont issues la plupart des idées présentées dans les chapitres 4 et 6 et dont l'implémentation est décrite dans le chapitre 7.

Chapitre IX

Conclusion et perspectives

1. Synthèse de l'approche

Le domaine de cette thèse est celui du déploiement d'applications industrielles dans des entreprises de grande taille. L'étude des approches et outils existants nous a permis d'arriver aux conclusions suivantes :

- les modèles d'applications à base de "composants" paraissent les plus adaptés pour prendre en compte les exigences des clients en terme de configuration et ils prennent de plus en plus en compte la problèmatique du déploiement lors de la phase de développement,
- les outils (ou environnements) de déploiement existants sont généralement spécialisés dans la réalisation d'une activité du cycle de vie du déploiement et souvent imposent des contraintes fortes sur le type de déploiement à réaliser.

L'objectif de cette thèse est de proposer un environnement de déploiement, c'est à dire une plate-forme permettant de couvrir toutes les activités du cycle de vie du déploiement logiciel. Dans le contexte du "large échelle" chaque producteur d'application dispose (ou utilise) des outils de déploiement et applique ses propres stratégies (ou politiques) de déploiement. Par exemple, un producteur peut utiliser l'outil "InstallShield" pour déployer en mode "push" ses applications, alors qu'un autre producteur peut utiliser "Java Web Start" pour permettre le déploiement en mode "pull" de ses applications. De la même manière, les entreprises appliquent des politiques de déploiement et imposent des contraintes (de sécurité par exemple) différentes. Nous pensons qu'un environnement de déploiement ne sera utilisé, du point de vue industriel, que s'il permet l'utilisation de ces outils de déploiement et la mise en pratique des différentes stratégies de déploiement.

Nous résumons les différentes contributions de notre travail.

1.1. Identification de l'architecture de déploiement

En analysant les différents scénarios de déploiement proposés par la société Actoll et en étudiant les environnements de déploiement existants (comme celui de Tivoli) nous nous sommes rendus compte qu'un certain nombre d'acteurs ou d'informations étaient nécessaires pour réaliser le déploiement.Dans le contexte du déploiement à large échelle, nous avons identifié une architecture à 3 niveaux :

- le niveau producteur chargé de fournir les applications à déployer et d'assurer ensuite l'évolution de ces applications,
- le niveau entreprise chargé de préparer le déploiement, c'est à dire de fixer où, quoi, quand et comment le déploiement de l'application va avoir lieu,
- le niveau utilisateur sur lequel les applications sont déployées.

Pour chacun de ces niveaux, nous avons proposé une modélisation. En plus de ces 3 niveaux, le déploiement fait intervenir un certain nombre d'élements manipulés/échangés par les acteurs du déploiement (le producteur, l'entreeprise et le client) par l'intermédiaire des outils de déploiement. Parmi ces éléments, on trouve les applications et les "packages".

1.2. Proposition d'un environnement basé sur la réutilisation

Cette thèse propose un environnement permettant aux acteurs du déploiement (entreprises, producteurs et les clients) de continuer à utiliser leurs outils de déploiement et d'appliquer leurs politiques de déploiement. Comme on l'a vu dans le chapitre sur l'état de l'art, il n'existe (à notre connaissance) aucun outil de déploiement capable de couvrir tout le cycle de vie du déploiement. Il faut donc que l'environnement soit capable, d'une part d'utiliser ces outils et d'autre part de les faire coopérer entre eux. En plus de cette contrainte, l'objectif d'un environnement de déploiement est de simplifier en l'automatisant le déploiement, il faut donc être capable d'exécuter à distance un outil et de le contrôler ensuite. Pour réaliser cela, nous proposons dans ORYA :

- une infrastructure de communication et de coopération,
- une infrastructure d'exécution à distance des outils.

Nous sommes capables maintenant d'utiliser et d'exécuter (assez simplement) à distance n'importe lequel des outils de déploiement

1.3. Proposition d'automatisation du déploiement

Le déploiement peut être vu comme l'enchaînement contrôlé d'un ensemble d'actions. Automatiser le déploiement, consiste à décrire, interpréter puis exécuter ce processus. Pour cela nous avons choisi de nous baser sur la technologie des procédés. Comme présenté dans le chapitre 6, chaque installation (et à fortiori chaque activité du déploiement) est différente. Cette différence s'explique par de nombreuses raisons : les politiques de déploiement, l'état de l'environnement au moment de l'installation, etc.. Pour tenir compte de ces différences, nous proposons un langage permettant :

- de décrire les différents actions à réaliser,
- de décrire l'enchaînement de ces actions,
- de modifier dynamiquement le procédé.

Pour réaliser les deux premières fonctionnalités, nous avons développé un "environnement" de conception et d'exécution de procédés. La modification dynamique de procédé est un aspect important du langage, car il permet au moment du déploiement de prendre en compte l'état de l'environnement et/ou les politiques de déploiement des différents acteurs et de modifier en conséquence le procédé de déploiement.

1.4. Conclusion sur l'approche

Pour conclure sur l'approche, ORYA est un environnement de déploiement, qui grâce à l'utilisation des fonctionnalités offertes par les outils, est capable de couvrir tout le cycle de vie du déploiement. L'intégration de ces outils est avantageuse en terme d'efficacité (les outils sont très performants), de correction, de maintenance et d'utilisation. L'autre point fort d'ORYA consiste en la mise en place du support (via le langage de réalisation) aux politiques de déploiement.

L'approche a été validée dans le cadre d'une collaboration industrielle avec la société Actoll. Lors de cette expériementation, un effort important a été réalisé pour rendre facile la création

et l'utilisation d'ORYA par des personnes non spécialisées dans le déploiement. Le résultat est très satisafaisant, car il a prouvé qu'ORYA pouvait déployer de manière automatisée et à distance l'application Centr'actoll et ceci en respectant le cahier des charges (dont une partie a été décrite dans le chapitre 8).

2. Les Perspectives

Les bases de l'environnement de déploiement étant posées et validées, plusieurs voies de recherche ou de développement méritent, selon nous, d'être à présent explorées. Nous identifions plusieurs perspectives à court et à plus long terme. Certaines d'entre elles font d'ailleurs déjà l'objet de travaux de thèse ou de masters.

2.1. Les autres activités du déploiement

Nos travaux ont porté essentiellement sur les activités de sélection et d'installation. Il reste maintenant à étudier les autres activités du déploiement en particulier les activités de mise à jour et de reconfiguration. Pour cela, il faut largement développer les travaux autour des modèles de site et d'application. Il faut aussi définir la frontière exacte (si c'est possible) entre l'adaptation dynamique et le déploiement. Et enfin, nous avons montré que les procédés étaient une bonne solution pour automatiser l'activité d'installation, il est intéressant de vérifier que c'est aussi le cas pour les autres activités du déploiement (en particulier pour celles de mise à jour et de reconfiguration).

2.2. La gestion du procédé de déploiement

Nous avons proposé d'utiliser les procédés comme technologie pour réaliser les activités de déploiement. Il apparaît intéressant d'étudier s'il est possible de faire de même pour gérer le niveau au-dessus : celui de la gestion du déploiement au niveau entreprise, c'est à dire la gestion des différentes phases du cycle de vie entre-elles et la gestion des différents déploiements entre-eux (politiques de déploiements).

2.3. Les politiques de déploiement

La solution proposée pour l'activité d'installation peut être vue comme la solution réalisant une (ou plusieurs) politique(s) de déploiement. Les politiques de déploiement étant différentes selon les entreprises, il faut pouvoir adapter les procédés en conséquence. Le langage de réalisation nous apparaît comme un bon début, car il offre un support pour l'expression de ces politiques (sous forme d'activités) et pour la modification dynamique (par exemple, pour modifier un procédé en exécution en accord avec la politique de déploiement de l'entreprise). Il nous semble particuliérement important de poursuivre les travaux dans cet axe et d'aborder une formalisation des politiques de déploiement.

2.4. L'industrialisation d'ORYA

Dans le cadre de notre collaboration avec la société Actoll, une industrialisation de l'environnement de déploiement est envisagée afin de l'intégrer dans l'application Centr'actoll comme support à son déploiement. Ce projet nécessite de stabiliser et sécuriser ORYA, autrement dit de passer du niveau prototype au niveau application. Par exemple, il faut

développer des outils ergonomiques et facile d'utilisation permettant la réutilisation d'activités pour la conception de procédés de déploiement ou d'aide à l'écriture des templates.

3. Conclusion

L'environnement et par conséquent l'approche proposée ont été validées par une expérimentation réalisée dans un contexte industriel. Nous avons défini un procédé d'installation et implémenté des outils permettant d'installer les applications de la société actoll. Bien que nous soyons au début des tests en grandeur réel, nous considérons que les résultats sont très satisfaisants.

Nous pensons que le travail présenté dans cette thèse est un bon support pour le déploiement. En effet en basant notre approche sur la réutilisation des outils existants et sur une modélisation des applications suffisamment générique pour correspondre à la plupart des types d'application existants, nous sommes capable de déployer ces applications.

Cependant, il est nécessaire de "prendre du recul" et d'engager une réflexion et une formalisation de l'ensemble des concepts manipulés afin d'être à même de fournir un support au déploiement garantissant (avec preuve à l'appui) la qualité de service qu'est en droit d'attendre chaque utilisateur ou en entreprise.

Chapitre X

Bibilographie

[Actoll] Société Actoll site web : http://www.actoll.com/

[ADC] Software Maintenance & Development Systems, Inc., Aide-De-Camp Software Management System, Product Overview, Concord, MA, 1989.

[And00] R. Anderson. "The End of DLL Hell". Technical Article of Microsoft Corporation. January 2000. Disponible à : http://msdn.microsoft.com/library/

[Ang02] D. Angeline. "Side-by-Side Execution of .Net Framework". Technical Article of Microsoft Corporation. January 2000. Disponible à : http://msdn.microsoft.com/library/

[BB95] D. Bolinger and T. Bronson. "Applying RCS and SCCS". O'Reilly & Associates, 1995.

[BCLS01] N.Belkhatir, P.Y. Cunin, V. Lestideau and H. Sali. "An OO framework for configuration of deployable large component based Software Products". Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, Florida, October 2001.

[BEA] BEA's WebLogic. http://www.bea.com

[CA01] J. Caple and M. H. Altarace. "The art of EJB Deployment". Technical Article. August 2001. Disponible à : http://www.javaworld.com/javaworld/jw-08-2001/jw-0803-ejb_p.html

[CATIA] Dassault Systèmes. CATIA : http://www.3ds.com/en/brands/catia_ipf.asp

[CBD+94] G. Canals, N. Boudjlida, J.-C Derniame, C. Godart and J. Lonchamp. "ALF: A Framework for Building Process-Centered Software Engineering Environments". chapter 7, pages 153-187. Research Studies Press. In ``Software Process Modeling and Technology", A. Finkelstein, J. Kramer and B.A. Nuseibeh, editors. 1994

[CCM] OMG, "CORBA Components – Version 3.0". Juin 2002. Disponible à : http://www.omg.org/technology/documents/formal/components.htm

[CE00] T. Coupaye and J. Estublier. "Foundations of Enterprise Software Deployment". Proceedings of the 4th European Conference on Software Maintenance and Reengineering, Zurick, February 2000, IEEE Computer Society, 2000 pages 65-74.

[Ced03] P. Cederqvist. "Version Management with CVS". Disponible à : http://www.cvshome.org/docs/manual/

[Centractoll] Centr'Actoll Project. http://www-adele.imag.fr/Les.Groupes/centractoll/

[CFF94] R. Conradi, C. Fernstrom, and A. Fuggetta. "Concepts for evolving software processes". In A. Finklestein, J. Kramer, and B. Nuseibeh, editors, Software Process Modelling and Technology, pages 9-31. Research Studies Press, 1994.

[CFH+98] A.Carzaniga, A. Fuggetta, R. S. Hall, A. van der Hoek, D. Heimbigner, A. L. Wolf. "A Characterization Framework for Software Deployment Technologies," Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado, April 1998.

[CL95] R. Conradi and C. Liu. "Process Modelling Languages: One or Many?". In Proceedings of the 4th European Workshop on Software Process Technology, Lecture Notes in Computer Science 913, Springer, Noordwijkerhout, The Netherlands, April 1995.

[COR] Object Management Group. CORBA Home Page. http://www.corba.org

[CW96] R. Conradi and B. Westfechtel. "Configuring Versioned Software Products" Proceedings of the 6th International Workshop on Configuration Management, Berlin, Germany, March 1996.

[CW97] R. Conradi and B. Westfechtel. "Towards a Uniform Version Model for Software Configuration Management". Proceedings of the 7th International Workshop on Software Configuration Management, number 1235 in Lecture Notes in Computer Science, pages 1-17, Springer 1997, New-York 1997.

[CW98] R. Conradi and B. Westfechtel. "Version Models for Software Configuration Management". ACM Computing Surveys, pages 232-282. 1998.

[Dart91] S. Dart. "Concepts in Configuration Management Systems". In P.H. Feiler, editor, Proceedings of the Third International Workshop on Software Configuration Management, pages 1-18, Trondheim, Norway, June 1997. ACM SIGSOFT, ACM Press, New York.

[DAW98] J.C. Derniame, B. Ali Kaba and D. Wastell. "Software Process: Principles, Methodology, Technology". Lecture Notes in Computer Science 1500 Springer 1999.

[DEA98] S. Dami, J. Estublier, and M. Amiour. "APEL: a Graphical Yet Executable Formalism for Process Modeling". Kuwler Academic Publisher, pp. 60-96, Boston, January 1998

[Der94] J.-C. Derniame and all. "Life Cycle Process Support in PCIS". Proceedings of PCTE'94, San Francisco, November 1994.

[DMTF] Distributed Management Task Force homepage : http://www.dmtf.org/home

[DotNet] Microsoft Dot Net homepage : http://www.microsoft.com/net/

[Duc02] F. Duclos. "Environnement de Gestion de Services Non Fonctionnels dans les applications à Composant". Thèse en informatique à l'Université Joseph Fourier, Octobre 2002.

[EC94] J. Estublier and R. Casallas. "The Adele Software Configuration Manager". chapter 4, pages 99–139. Trends in Software. J. Wiley and Sons, Baffins Lane, Chichester West Sussex, PO19 1UD, England, 1994.

[EJB] Enterprise Java Beans 2.1 Specification Proposed Final Draft. Disponible à : http://java.sun.com/products/ejb/docs.html

[Est85] J. Estublier. "A Configuration Manager: The Adele Data Base of Programs", Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large, June 1985, pp. 140-147.

[EVL+03] J. Estublier, J. Villalobos, A.-T. Le, S. Sanlaville and G. Vega. "An approach and Framework for Extensible Process Support System". Proceedings of the 9th Workshop on Software Process Technology, Lecture Note in Computer Science 2786, Springer

2003, pages 46-61, Helsinki, September 2003.

[EVL03] J. Estublier, J. Villalobos and A.-T. Le. "Using Federations for Flexible SCM Systems". Proceedings in the 11th International Workshop on Software Configuration Management. Lecture Note in Computer Science, Springer 2003, Portland, May 2003.

[Garg95] P. K. Garg and M. Jazayeri, editors. "Process-Centered Software Engineering Environnements". IEEE Computer Society Press, 1995.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "Design Patterns". Addison Wesley, 1995.

[GNU] GNU home page : http://www.gnu.org/

[Gom00] L. Gomez. " Modèle de Sites pour le déploiement de logiciels". Rapport de DEA, Université Joseph Fourier, Juin 2000.

[Hall99] R. S. Hall "Agent-based Software Configuration and Deployment". Phd Thesis of University of Colorado, 1999.

[Han99] W. J. Hansen, "Deployment Descriptions in a World of COTS and Open Source" Proceedings in the International Symposium on System Configuration Management (SCM-9), Lecture Notes in Computer Science 1675, Springer, Heidelberg, September 1999.

[HHC+98] A. van der Hoek, R. S. Hall, A. Carzaniga, D. Heimbigner and A. L. Wolf. "Extending Configuration Management Support into the Field". Crosstalk, The Journal of Defense Software Engineering, volume 11, number 2, February 1998

[HHHW97a] R.S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. "An Architecture for Post-Development Configuration Management in a Wide-Area Network". Proceedings of the 17th International Conference on Distributed Computing Systems, Baltimore, USA, May 1997.

[HHHW97b] A. van der Hoek, R.S. Hall, D. Heimbigner, and A. L. Wolf. "Software Release Management". Proceedings of the 6th European Software Engineering Conference, Zurich, Switzerland, September 1997.

[HHW95] A. van der Hoek, D. Heimbigner, and A. L. Wolf. " Does Configuration Management Research Have a Future?". Proceedings of the 5th International Software Configuration Management Workshop, Seattle, USA, April 1995.

[HHW98] R. S. Hall, D. Heimbigner, and A. L. Wolf. "Evaluating Software Deployment Languages and Schema," Proceedings of the 1998 International Conference on Software Maintenance, IEEE Computing Society, Nov. 1998.

[HHW99a] R. S. Hall, D. Heimbigner, and A. L. Wolf. "Specifying the deployable software description format in XML". Technical Report CU-SERL-207-99, University of Colorado Software Engineering Research Laboratory, Mar. 1999

[HHW99b] R. S. Hall, D. Heimbigner, and A. L. Wolf. "Enterprise Software Deployment: It's the Control, Stupid". Proceedings in the ICSE 99 Workshop "Software Engineering over the Internet, Calgary, May 1999.

[HHW99c] D. Heimbigner, R.S. Hall, and A.L. Wolf, "A Framework for Analyzing

Configurations of Deployable Software Systems," Proceedings of the 5th IEEE Int'l Conference on Engineering of Complex Computer Systems, pages 32-42, Las Vegas, October 1999.

[HHW99d] R. Hall, D. Heimbigner, and A.L. Wolf, "A Cooperative Approach to Support Software Deployment Using the Software Dock," Proceedings of the Internationale Conference on Software Engineering, IEEE Computer Science Press, 1999.

[Hoek00] A. van der Hoek, "A Reusable, Distributed Repository for Configuration Management Policy Programming," Ph. D. Thesis, January 21, 2000, Department of Computer Science, University of Colorado.

[InstallShield] InstallShield homepage : http://www.installshield.com/

[Java] Java Sun homepage : http://java.sun.com/

[Jboss] Jboss : Homepage : http://www.JBoss.org

[JNLP] Java Network Launching Protocol Specification 1.0.1. May 2001. Disponible à : http://java.sun.com/products/javawebstart/download-spec.html

[JWS] Java Web Start homepage : http://java.sun.com/products/javawebstart/

[Katz90] R.H. Katz, "Toward a Unified Framework for Version Modeling in Engineering Databases," ACM Computing Surveys, vol. 22, no. 4, pages 375-408, Dec. 1990.

[KBC02] M. Ketfi, N. Belkhatir and P.Y. Cunin. "Automatic Adaptation of Component-based Software: Issues and Experiences". Proceedings in the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, June 2002.

[LB03] V. Lestideau and N. Belkhatir. "Providing Highly automated and generic means for software process deployment". Proceedings of the 9th Workshop on Software Process Technology, Lecture Note in Computer Science 2786, Springer 2003, pages 128-142, Helsinki, September 2003.

[LBC01] V. Lestideau, N. Belkhatir and P.-Y. Cunin. " Un environnement de déploiement automatisé d'applications d'entreprise". Revue génie logiciel, pages 23-31, mars 2001.

[LBC02] V. Lestideau, N. Belkhatir and P.-Y. Cunin. "Towards automated software component configuration and deployment". Proceedings of the 3rd International Workshop on Process support for Distributed Team-based Software Development, Orlando, July 2002.

[Les00] V. Lestideau. " Modèle d'application pour le déploiement de logiciel". Rapport de DEA, Université de Savoie, Juin 2000.

[Les02] V. Lestideau. "Un environnement de déploiement automatique pour les applications à base de composants". Proceedings of the 15th International Conference Software & Systems Engineering and their Applications, Paris, December 2002.

[LEV03] A.-T Le, J. Estublier and J. Villalobos. "Multi-Level Composition for Software Federations". Proceedings in the Software Composition 2003, ENTCS Vol. 82, N°5, Elsevier, Warsaw, April 2003.

[LMKB02] V. Lestideau, N. Merle, M. Ketfi and N. Belkhatir. "Spécification du déploiement de la plate-forme Centr'ACTOLL". Document Technique Equipe Adèle, Novembre 2002.

[Mer02] N. Merle. " Etude des principes et concepts à la base des architectures pour les systèmes de déploiement". Rapport de DEA, Université Joseph Fourier, Juin 2002.

[ML88] A. Mahler and A. Lampen. "Shape --- a software configuration management tool". Proceedings of the International Workshop on Software Version and Configuration Control, Grassau, W. Germany, January 1988.

[MM01] R. Marvie and P. Merle. "CORBA Component Model: Discussion and Use with Open CCM". Informatica - International Journal of Computing and Informatics, 2001.

[Mon01] R. Monson-Haefel. "Enterprise JavaBeans". O'Reilly & Associates, 3rd edition, october 2001.

[ObjectWeb] Object Web Consortium homepage : http://www.objectweb.org/index.html

[OMG] Object Management Group homepage : http://www.omg.org/

[OMG02] OMG CCM Implementers Group. "CORBA Component Model Tutorial". OMG meeting, Yokohama, Japon, le 24 avril 2001, disponible à : http://www.omg.org/cgi-bin/doc?ccm/2002-04-01

[Oque95] F. Oquendo. "SCALE: Process Modelling Formalism and Environment Framework for Goal-Directed Cooperative Processes". In Proceedings of 7th International Conference on Software Engineering Environments". IEEE Computer Science Press, Noordwijkerhout, The Netherlands, April 1995.

[ORYA] Open enviRonment to deploY Applications homepage : http://www-adele.imag.fr/ORYA/

[OSD] Open Software Description Format Specification. Disponible à : http://www.w3.org/TR/NOTE-OSD

[Pal01] N. De Palma. "Services d'administration d'applications réparties". Thèse de doctorat, Université Joseph Fourier, Grenoble, Novembre 2001.

[Pay03] F. Payraudeau. "Cahier des charges : Utilisation d'ORYA". Document technique de la société Actoll. Juillet 2003.

[PBJ97] F. Plasil, D. Balek and R. Janecek, "DCUP: Dynamic Component Updating in Java/CORBA Environment", Technical Report No. 97/10, Dep. Of SW Engineering, Charles University, Prague, 1997.

[Pol01] J. T. Pollock, "The big Issue: Interoperability vs. Integration", EAI Journal, pages 48-52, October 2001.

[Promoter96] Promoter Group. Editor "Software Process Technology". 1996

[RCS] Revision Control System. Disponible à : http://www.gnu.org/software/rcs/rcs.html

[RJB99] J. Rumbaugh, I. Jacobson and G. Booch. "The Unified Modeling Language Reference Manual". Object Technology Series. Addison Wesley, 1999.

[Sali01] H. Sali. "Framework Orienté Objet pour le déploiement de composants logiciels hiérarchiques". Rapport de DEA, Université Joseph Fourier, Juin 2001.

[Scott00] S. Scott. "Structuring a .NET Application For Easy Deployment". Technical Article of Microsoft Corporation. February 2000. Disponible à : http://msdn.microsoft.com/library/

[SoftwareDock] Software Dock. Disponible à : http://www.cs.colorado.edu/serl/cm/dock.html

[SW94] R. A. Snowdon and B. C. Warboys. "An Introduction to Process-Centred Environments". In A. Finklestein, J. Kramer, and B. Nuseibeh, editors, Software Process Modelling and Technology, pages 1-8. Research Studies Press, 1994.

[TCG93] E. Tryggeseth, R. Conradi and B. Gulla. "Software Configuration Management in PROTEUS". Proceedings of the 4th International Workshop on Software Configuration Management, 1993.

[TCM] Tivoli Configuration Manager homepage : http://www-3.ibm.com/software/tivoli/products/config-mgr/

[TGC95] E. Tryggeseth, B. Gulla, and R. Conradi. "Modelling systems with variability using the PROTEUS configuration language". In Software Conguration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers, number 1005 in Lecture Notes in Computer Science, pages 216-240, New York, Springer-Verlag, New York, 1995.

[TI] Tivoli Inventory homepage : http://www-3.ibm.com/software/tivoli/products/inventory/

[Tichy85] W. F. Tichy. "RCS - A system for version control". Software-Practice and Experience, vol. 15, no. 7, pages 637-654, July 1985.

[Tivoli] Tivoli : http://www-3.ibm.com/software/tivoli/

[TSD] Tivoli Software Distribution whitePaper : http://www-3.ibm.com/software/tivoli/resource-center/security/wp-software-dist.jsp

[UML97] Object Modeling Group. "Unified Modeling Language version 1.0". January 1997.

[Vil02] J. Villalobos. "APEL: Spécification formelle du moteur". Rapport Technique Equipe Adèle. Mars 2002.

[Vil03] J. Villalobos. "Fédération de composants : une Architecture Logicielle pour la Composition par Coordination". Thèse de doctorat, Université Joseph Fourier, Grenoble, Juillet 2003. Disponible à : http://www-adele.imag.fr/Les.Publications/reports/PHD2003Vil.pdf

[WebSphere] IBM's WebSphere homepage : http://www-3.ibm.com/software/info1/websphere

[WIS] The Windows Installer Service. White paper. Disponible à : http://www.microsoft.com

[WMC01] B. Westfechtel, B.P. Munch, and R. Conradi. "A Layered Architecture for Uniform Version Management". IEEE Transactions on Software Engineering, pages 1111-1133, 2001.

[Xml01] W3C. XML Schema. W3C Recommendation, May 2001. Disponible à : http://www.w3.org/TR/xmlschema-0/

[Zuk02] J. Zukowski. "Deploying Software with JNLP and Java Web Start". Technical Article. August 2002. Disponible à : http://developer.java.sun.com/developer/technicalArticles/Programming/jnlp/

Chapitre XI

Annexes

1. Exemple du langage de description des procédés

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<process name="transfert">
	<description>
		null
	</description>
	<activity name="ActivityROOT" instances="1" type="composite">
		<port name="begin" type="in-sync">
			<product name="nomPackage"/>
			<product name="serveurApplication"/>
			<product name="cible"/>
			<dataflow from-port="begin" to-port="Desktop">
				<product name="cible"/>
			</dataflow>
			<dataflow to-activity="GetFile" from-port="begin" to-port="begin">
				<product name="serveurApplication"/>
				<product name="nomPackage"/>
			</dataflow>
		</port>
		<port name="ok" type="out-sync">
		</port>
		<port name="ko" type="out-sync">
			<product name="erreur"/>
			<dataflow from-port="Desktop" to-port="ko">
				<product name="erreur"/>
			</dataflow>
		</port>
		<desktop>
			<product name="erreur"/>
			<product name="package"/>
			<product name="cible"/>
		</desktop>
		<subactivity name="GetFile"/>
		<subactivity name="PutFile"/>
	</activity>
	<activity name="GetFile" instances="1" type="simple" responsable="default">
		<port name="begin" type="in-sync">
			<product name="nomPackage"/>
			<product name="serveurApplication"/>
			<dataflow from-port="begin" to-port="Desktop">
				<product name="serveurApplication"/>
				<product name="nomPackage"/>
			</dataflow>
		</port>
		<port name="ok" type="out-sync">
			<product name="package"/>
			<dataflow from-port="Desktop" to-port="ok">
				<product name="package"/>
			</dataflow>
			<dataflow from-activity="GetFile" from-port="ok" to-port="Desktop">
				<product name="package"/>
			</dataflow>
		</port>
		<port name="ko" type="out-sync">
			<product name="erreur"/>
			<dataflow from-activity="GetFile" from-port="ko" to-port="ko">
				<product name="erreur"/>
```

```xml
				</dataflow>
				<dataflow from-port="Desktop" to-port="ko">
					<product name="erreur"/>
				</dataflow>
			</port>
			<desktop>
				<product name="nomPackage"/>
				<product name="erreur"/>
				<product name="package"/>
				<product name="serveurApplication"/>
			</desktop>
		</activity>
		<activity name="PutFile" instances="1" type="simple" responsable="default">
			<port name="begin" type="in-sync">
				<product name="package"/>
				<product name="cible"/>
				<dataflow from-port="begin" to-port="Desktop">
					<product name="cible"/>
					<product name="package"/>
				</dataflow>
				<dataflow to-activity="PutFile" from-port="Desktop" to-port="begin">
					<product name="cible"/>
					<product name="package"/>
				</dataflow>
			</port>
			<port name="ok" type="out-sync">
				<dataflow from-port="Desktop" to-port="ok">
				</dataflow>
				<dataflow from-activity="PutFile" from-port="ok" to-port="ok">
				</dataflow>
			</port>
			<port name="ko" type="out-sync">
				<product name="erreur"/>
				<dataflow from-activity="PutFile" from-port="ko" to-port="ko">
					<product name="erreur"/>
				</dataflow>
				<dataflow from-port="Desktop" to-port="ko">
					<product name="erreur"/>
				</dataflow>
			</port>
			<desktop>
				<product name="erreur"/>
				<product name="package"/>
				<product name="cible"/>
			</desktop>
		</activity>
		<product name="nomPackage" type="data"/>
		<product name="serveurApplication" type="machine"/>
		<product name="cible" type="machine"/>
		<product name="erreur" type="error"/>
		<product name="package" type="package"/>
</process>
```

2. Le modèle de description d'un serveur d'applications

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://castor.exolab.org/" version="0.9.4">
        <xsd:complexType name="Property">
                <xsd:sequence>
                        <xsd:element name="name" type="xsd:string"/>
                        <xsd:element name="value" type="xsd:string"/>
                </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="Package">
                <xsd:sequence>
                        <xsd:element name="name" type="xsd:string"/>
                        <xsd:element name="id" type="xsd:string"/>
                        <xsd:element name="version" type="xsd:string"/>
                        <xsd:element name="url" type="xsd:string"/>
                        <xsd:element name="dependency"
                                type="xsd:string" minOccurs="0"
                                maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>

        <xsd:element name="ApplicationServerModel">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element name="property" type="Property"
                                        minOccurs="0" maxOccurs="unbounded"/>
                                <xsd:element name="package"
                                         type="Package" minOccurs="0"
                                        maxOccurs="unbounded"/>
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
</xsd:schema>
```

3. Le modèle de description d'un site

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://castor.exolab.org/" version="0.9.4">
        <xsd:complexType name="Property">
                <xsd:sequence>
                        <xsd:element name="name" type="xsd:string"/>
                        <xsd:element name="value" type="xsd:string"/>
                </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="Application">
                <xsd:sequence>
                        <xsd:element name="name" type="xsd:string"/>
                        <xsd:element name="id" type="xsd:string"/>
                        <xsd:element name="version" type="xsd:string"/>
                        <xsd:element name="url" type="xsd:string"/>
                        <xsd:element name="dependency" type="xsd:string"
                                minOccurs="0" maxOccurs="unbounded"/>
```

```
		</xsd:sequence>
	</xsd:complexType>

	<xsd:element name="SiteModel">
		<xsd:complexType>
			<xsd:sequence>
				<xsd:element name="property" type="Property"
					minOccurs="0" maxOccurs="unbounded"/>
				<xsd:element name="application" type="Application"
					minOccurs="0" maxOccurs="unbounded"/>
			</xsd:sequence>
		</xsd:complexType>
	</xsd:element>
</xsd:schema>
```

4. Description d'un manifeste

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
	targetNamespace="http://castor.exolab.org/" version="0.9.4">
	 <xsd:complexType name="Package">
		<xsd:sequence>
			<xsd:element name="name" type="xsd:string"/>
			 <xsd:element name="id" type="xsd:string"/>
			<xsd:element name="version" type="xsd:string"/>
			<xsd:element name="url" type="xsd:string"/>
		<xsd:element name="dependency" type="xsd:string"
				minOccurs="0" maxOccurs="unbounded"/>
		</xsd:sequence>
	</xsd:complexType>
	<xsd:element name="Manifest">
		<xsd:complexType>
			<xsd:sequence>
				<xsd:element name="package" type="Package"/>
			</xsd:sequence>
		</xsd:complexType>
	</xsd:element>
</xsd:schema>
```

5. Template de l'activité GetFile

```
import java.util.*;
import apel.motor.model.*;
import java.io.*;

activity GetFile
{
components
{
	transfer : transfer at $this.desktop.serverApplication.name;
	deployServer : deployServer at $LOCAL_HOST;
}

case $(this).isReady() :
{
	String processInstanceName = $(this).getProcessInstance().getProcessId();
	String tmpName = processInstanceName+".zip";
		Boolean res = transfer.getFile($this.desktop.nomPackage, tmpName);
```

```
            if (!res)
            {
                        $this.desktop.error = $newProduct;
                        $this.desktop.error.msg = "pb pendant le transfert vers la fede";
                        terminate($this,$this.ko);
            }
            else
            {
                        $this.desktop.package = $newProduct;
                        String url = deploySer ver.getSiteProperty("tmp.folder"
                                                +"\\"+processInstanceName);
                        $this.desktop.package.url = url;
                        $this.desktop.package.absoluteName = newTmpDir
                                                + "\\" +tmpName;
                        terminate($this,$this.ok);
            }
}
}
```

6. Template de l'activité Transfert

```
import java.util.*;
import apel.motor.model.*;

activity Transfer
{

components
{
            verif : verificationTool at $LOCAL_HOST;
}

case $(this).isReady()
                        && $(GetFile).isTerminated()
                        && $(GetFile).getFinalExitPort() == $GetFile.ok :
{
            boolean res = verif.analyse($this.desktop.package.absoluteName);
            if (res )
            {
                        terminate($this,$PutFile.begin);
            }
            else
            {
                        $this.desktop.error = $newProduct;
                        $this.desktop.error.msg = "le package n'est pas validé";
                        terminate($this,$this.ko);
            }
}
}
```

7. Template de l'activité PutFile

```
import java.util.*;
import apel.motor.model.*;
import java.io.*;

activity GetFile
{
	components
	{
		transfer : transfer at $this.desktop.server.machine;
	}

	case $(this).isReady() :
	{
		boolean res = transfer.putFile($this.desktop.package.absoluteName);
		if (!res)
		{
			$this.desktop.error = $newProduct;
			$this.desktop.error.msg = "pb pendant le transfert vers la cible";
			terminate($this,$this.ko);
		}
		else
		{
			terminate($this,$this.ok);
		}
	}
}
```

8. Le manifeste d'installation

```
	<?xml version="1.0" encoding="UTF-8"?>
	<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
		targetNamespace="http://castor.exolab.org/" version="0.9.4">
		<xsd:complexType name="Param">
			<xsd:sequence>
				<xsd:element name="value" type="xsd:string"/>
				<xsd:element name="type" type="xsd:string"/>
			</xsd:sequence>
			<xsd:attribute name="calculated" type="xsd:boolean" />
		</xsd:complexType>
		<xsd:complexType name="Property">
			<xsd:sequence>
				<xsd:element name="name" type="xsd:string"/>
				<xsd:element name="value" type="xsd:string"/>
			</xsd:sequence>
			<xsd:attribute name="calculated" type="xsd:boolean" />
		</xsd:complexType>
		<xsd:complexType name="RoleType">
			<xsd:sequence>
				<xsd:element name="roleName" type="xsd:string"/>
				<xsd:element name="interfaceName" type="xsd:string"/>
				<xsd:element name="methodName" type="xsd:string"/>
				<xsd:element name="param" type="Param"
					minOccurs="0" maxOccurs="unbounded"/>
				<xsd:element name="comment" type="xsd:string"
					minOccurs="0"  maxOccurs="1"/>
```

```
            </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="Step">
            <xsd:sequence>
                <xsd:element name="do" type="RoleType" />
                <xsd:element name="verification" type="RoleType"
                    minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="undo" type="RoleType"
                    minOccurs="0" maxOccurs="1" />
                <xsd:element name="comment" type="xsd:string"
                    minOccurs="0"  maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="stepName" type="xsd:string" />
        </xsd:complexType>
        <xsd:element name="Install">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="property" type="Property"
                        minOccurs="0" maxOccurs="unbounded" />
                    <xsd:element name="extract" type="Step"/>
                    <xsd:element name="installStep" type="Step"
                        minOccurs="1" maxOccurs="unbounded" />
                    <xsd:element name="terminate" type="Step" />
                    <xsd:element name="comment" type="xsd:string"
                         minOccurs="0"  maxOccurs="1"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:schema>
```

9. Un exemple de manifeste d'installation

```
    <Install xmlns="http://castor.exolab.org/">
        <property calculated="true" xmlns:="http://castor.exolab.org/">
            <name>SCAR.INSTALL_TEMP</name>
            <value>%TMP.SOURCE</ value>
        </property>
        <extract stepName="Extraction">
            <do>
                <roleName>installer</roleName>
                <interfaceName>
                    deployment.roles.installer.InstallerRole
                </interfaceName>
                <methodName>unzip</methodName>
                <param calculated="false">
                    <value>scarPatch320.zip</value>
                    <type>java.lang.String</type>
                </param>
                <param calculated="true">
                    <value>%SCAR.INSTALL_TEMP</value>
                    <type>java.lang.String</type>
                </param>
            </do>
            <verification>
                <roleName>fileVerification</roleName>
                <methodName>existFile</methodName>
                <param calculated="true">
                    <value>%SCAR.INSTALL_TEMP+\\+deploy.cmd</value>
                    <type>java.lang.String</type>
```

```
                </param>
            </verification>
        </extract>
        <installStep stepName="Etape-1">
            <do>
                <roleName>executer</roleName>
                <interfaceName>
                    deployment.roles.batManager.BatManagerRole
                </interfaceName>
                <methodName>execute</methodName>
                <param calculated="true">
                    <value>
                        %SCAR.INSTALL_TEMP+\+Deploy.cmd
                    </value>
                    <type>java.lang.String</type>
                </param>
            </do>
        </installStep>
        <terminate stepName="Terminaison">
            <do>
                <roleName>fileManager</roleName>
                <interfaceName>
                    deployment.roles.fileManager.FileRole
                </interfaceName>
                <methodName>deleteDirectory</methodName>
                <param calculated="true">
                    <value>%SCAR.INSTALL_TEMP</value>
                    <type>java.lang.String</type>
                </param>
            </do>
        </terminate>
    </Install>
```