Déploiement d'Applications à Services sur des Environnements d'Exécution à Services : Une Approche Dirigée par les Modèles.

Antonin Chazalet

UNIVERSITE JOSEPH FOURIER – GRENOBLE 1

THESE

pour obtenir le grade de

DOCTEUR de l'Université Joseph Fourier de Grenoble

(Arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

Discipline : Informatique

présentée et soutenue publiquement par

Antonin Maël CHAZALET

Le 21 novembre 2008

Déploiement d'Applications à Services sur des Environnements d'Exécution à Services : Une Approche Dirigée par les Modèles.

Directeur de thèse :
Pr. Philippe LALANDA

JURY :

Pr. Jean-Pierre GIRAUDIN, Professeur à l'Université Pierre Mendès France	(Président)
Pr. Carlo MONTANGERO, Professeur à l'Université de PISE (Italie)	(Rapporteur)
Pr. Pierre-Alain MULLER, Professeur à l'Université de Haute-Alsace	(Rapporteur)
Dr. Catherine HAMON, Chef de projet à France Télécom R&D	(Examinatrice)
Pr. Philippe LALANDA, Professeur à l'Université Joseph Fourier	(Directeur)

Thèse préparée au sein du Laboratoire LIG dans l'équipe ADELE

Si l'on sait exactement ce qu'on va faire, à quoi bon le faire ?
Picasso - Pablo Ruiz

L'imagination est plus importante que le savoir.
Albert Einstein

L'homme de science le sait bien, lui, que seule la science, a pu, au fil des siècles, lui apporter l'horloge pointeuse et le parcmètre automatique sans lesquels il n'est pas de bonheur terrestre possible.
Pierre Desproges

À Françoise et Frédéric

À Carmen

Remerciements

L'aide et le soutien de plusieurs personnes ont été importants pour aboutir à ce travail de thèse. Je tiens sincèrement à toutes les remercier et plus particulièrement :

Les membres du jury qui m'ont fait l'honneur de participer à ma soutenance de thèse et tout spécialement à mes rapporteurs Mr. Carlo MONTANGERO et Mr. Pierre-Alain MULLER pour avoir accepté de rapporter ma thèse et pour leurs commentaires, ainsi que Mr. Jean-Pierre GIRAUDIN et Mme. Catherine HAMON pour avoir été, respectivement, le président et l'examinatrice de ma soutenance de thèse.

Mon directeur de thèse Mr. Philippe LALANDA pour sa confiance et son soutien tout au long de cette thèse, ainsi que pour son ouverture d'esprit, ses commentaires et ses conseils sur les recherches réalisées.

Je tiens aussi à remercier Mr. Jacky ESTUBLIER, Mr. Pierre-Yves CUNIN, Mr. Philippe LALANDA et Mr. Didier DONSEZ pour m'avoir accueilli dans leur équipe. Je remercie aussi, d'une manière générale, tous les membres de l'équipe ; avec une mention spéciale pour le docteur Cristina MARIN (ma colocataire du bureau 204) qui a survécu et qui se remet, semble-t-il, bien, et pour Mr. Jean-Marie FAVRE pour avoir répondu à plusieurs de mes interrogations sur les modèles, les méta- et les méga- et pour donner une autre vision de la recherche.

Je tiens aussi à remercier mes amis, sans ou avec chapeaux à pin's, pour m'avoir apporté amitiés et soirées déstressantes, ainsi que l'aïkido et les émissions "le dessous des cartes", "rendez-vous avec X", "tracks" et "GRD" pour une autre vision du monde.

Je veux remercier, avec une attention toute particulière, Françoise et Frédéric, sans lesquels je ne serais pas là où j'en suis. Bisous maman, bisous papa.

Enfin, plein de bisssssous (avec la prononciation à la mexicaine) pour ma petite Pitufin-ita de Carmen pour être heureux tous les deux chaque jour.

Résumé

Notre objectif est la fourniture de services Internet basés sur des architectures "n-tiers". Dans notre contexte, ces architectures sont composées de cinq tiers : "équipements", "passerelles", "médiation", "serveur Internet" et "clients finaux". L'appréhension du tiers "passerelle" nécessite l'introduction d'applications à services qui sont destinées à répondre à des requêtes du tiers "médiation" et à collecter et transformer des données provenant du tiers "équipements" qui seront ensuite utilisées par le tiers "médiation" puis manipulées par le tiers "serveur Internet" qui est chargé de leur présentation aux clients finaux. L'introduction de telles applications à services destinées à fonctionner dans des environnements d'exécution à services nécessite la résolution de problématiques propres :

- au développement d'applications à services métiers [ML07],
- à leur déploiement dans des environnements d'exécution à services (ce qui est l'objectif de cette thèse) [CL07a]
- à la gestion dynamique des interactions entre les applications à services et les équipements (dé)-branchés à la passerelle (qui sont exposées comme des services) [EBL+08].

Le cycle de vie logiciel est composé de plusieurs phases. Les principales sont l'analyse, la conception, l'implémentation, le test et le déploiement. Cette dernière est une phase complexe qui est composée d'activités et de sous-activités comme l'installation, l'activation, la désactivation, les mises à jour statique et dynamique, l'évolution, le dynamisme et la désinstallation.

L'introduction et l'utilisation de l'approche à service augmente encore la complexité de la phase de déploiement, en particulier lorsqu'il s'agit de déployer des applications à services sur des environnements d'exécution à services. Les enjeux sont, ici :

- d'exécuter des activités de déploiement sur les applications tout en respectant l'architecture de chaque application (c'est-à-dire le modèle de chaque application),
- de maîtriser l'état de déploiement des (implémentations de) services réalisant ces applications,
- d'exécuter des activités de déploiement sur des services appartenant à ces applications en ne touchant que le service ciblé et donc sans impacter entièrement les applications dans leur ensemble.

Nous appelons ce dernier enjeu : "déploiement au plus près". Il impose que les unités de développement (c'est-à-dire les composants orientés service contraints) utilisées pour développer les applications et les unités de déploiement (c'est-à-dire les implémentations de services) utilisées pour réaliser les applications aient le même grain.

Cette thèse propose un prototype, nommé DMSA (pour *Deployment Manager for Services Applications*), automatisant le déploiement d'applications à services sur des environnements à services en suivant une approche dirigée par les modèles. L'utilisation de l'approche dirigée par les modèles est une réponse à la constante montée en abstraction du déploiement et de son point d'entrée. Le découplage total entre les niveaux modèles et "réalité" a été choisi afin de permettre la réutilisation des implémentations de services entre modèles de services. Cette thèse explicite notre approche, le méta-modèle pour les applications à services, la définition du déploiement pour les applications à services, l'architecture du DMSA et nos propositions. Ces dernières ciblent le déploiement d'un, puis de plusieurs modèles d'application à services sur un environnement d'exécution à services, le déploiement "transactionnel", l'appréhension de notre contexte embarqué/réactif et l'exécution de listes ordonnées d'activités de déploiement sur des environnements d'exécutions à services.

Mots clés : Génie Logiciel, Déploiement, Approche à service, Informatique orientée service, Architecture orientée service, Service, Application à services, Approche dirigée par les modèles, Ingénierie dirigée par les modèles, Déploiement dirigé par les modèles.

Services Applications Deployment on Services Execution Environments: A Model Driven Approach.

Abstract

Our goal is to provide Internet services on top of "n-tier" architectures. In our context, these architectures are made of five tiers: devices, gateways, mediation, Internet servers and presentation. Apprehending the gateways tier needs the introduction of services applications to answer the requests of the mediation tier and to collect and transform data from the "devices" tier, these data are then used by the "mediation" tier, next they are manipulated by the "Internet server" tier which is in charge of the displaying to the end users. The introduction of such services applications intended to run in services execution environments raises issues related to:

- the development of business-specific services applications [ML07],
- their deployment in services execution environments (this is the focus of this thesis) [CL07a]
- the dynamic management of interactions between services applications and the devices (un)-plugged to the gateways (these devices are exposed as services in the environments) [EBL+08].

Software lifecycle is made of several phases. The main ones are analysis, conception, implementation, test and deployment. The latter is a complex one; it is made of activities and sub-activities like installation, activation, deactivation, static and dynamic updates, evolution, dynamism and deinstallation.

The introduction and use of the service approach still raises the deployment phase complexity, in particular when the goal is the deployment of services applications on services execution environments. Stakes, here, are:

- to execute deployment activities targeting services applications while respecting each application's architecture (*i.e.* each application's model),
- to control the deployment state of the services (implementations) realizing these applications,
- to execute deployment activities on services belonging to these applications while acting only the targeted service and avoiding to impact the whole applications.

We called this last stake: "closely deployment" or "deployment in the small". It imposes that development units (*i.e.* constraints service-oriented components) used for developing applications and deployment units (*i.e.* services implementations) used for realizing applications have the same (fine) grain.

This thesis proposes a software prototype, called DMSA (for Deployment Manager for Services Applications), automating the deployment of services applications on services execution environments while following a model-driven approach. Using a model-driven approach is an answer for the constant rise in abstraction of the deployment phase and of its entry point. The full decoupling between model level and "reality" level has been chosen in order to reuse services implementations between services models. This document clarifies our approach, the services applications meta-model, the services applications deployment definition, the DMSA's software architecture and our propositions. These latter target deployment of several applications models on several services execution environments, "transactional" deployment, our embedded context apprehension and execution of deployment activities ordered lists on services execution environments.

Keywords: Software Engineering, Deployment, Service Approach Service-oriented computing (SOC), Service-oriented architecture (SOA), Service, Services applications, Model-driven approach, Model-driven engineering, Model-driven deployment.

Table des Matières

Table des Figures

Table des tableaux

INTRODUCTION

INTRODUCTION

1. CONTEXTE

La communauté du génie logiciel s'efforce d'appréhender un nombre significatif de besoins comme l'amélioration de la productivité et de la qualité et le besoin de flexibilité à l'exécution. En réponse, l'informatique orientée-service (*Service-oriented Computing*, SOC) a été récemment introduite. Le SOC promeut l'utilisation d'unités de compositions bien définies et découplées les unes des autres - les services - afin de supporter une ingénierie rapide et performante des logiciels. L'objectif majeur du SOC est la réduction du couplage entre unités de compositions, où une telle unité est typiquement un groupe de fonctionnalités accédé par un tiers. La réduction des ces dépendances permet aux différentes unités d'évoluer séparément, rendant les logiciels utilisant le paradigme SOC plus flexibles que, par exemple, les logiciels monolithiques [Pap03].

En ce qui concerne les services, les applications à services et le déploiement, quatre constats peuvent être faits :

- Les visions actuelles de l'approche à services ne sont pas satisfaisantes concernant l'expression et la manipulation d'applications à services.
- Il existe plusieurs définitions différentes du déploiement dont la cohérence et la complétude de chacune peuvent être critiquées.
- Le problème actuel du déploiement n'est plus de déployer un logiciel empaqueté sous la forme d'une unique unité de déploiement (dont le grain est celui du logiciel) ou sous la forme de plusieurs unités de déploiement dont le découpage a été imposé par la distribution/répartition du logiciel ; le problème actuel est le déploiement de logiciels au plus près. L'objectif est de pouvoir exécuter des activités de déploiement sur une partie d'un logiciel sans pour autant toucher entièrement le logiciel. Par exemple, dans le cadre du déploiement au plus près, la mise à jour d'une unité de développement doit pouvoir être réalisée en ne mettant à jour que l'unité de développement cible et donc sans devoir désinstaller et ré-installer le logiciel entier. Cela nécessite que tout ou partie des unités de déploiement aient le même grain que celui des unités de développement qui composent les logiciels.
- Enfin, la phase de déploiement logiciel et plus particulièrement son point d'entrée n'ont jamais cessé de monter en abstraction.

De manière plus concrète, notre travail de thèse se place dans le contexte des offres de services Internet de haut niveau. Ce type d'offre devient de plus en plus important aujourd'hui. C'est en particulier le cas pour les services Internet basés sur des données continuellement collectées à partir d'environnements physiques installés dans des usines, des centres commerciaux ou des réseaux domestiques. Dans l'industrie de la distribution électrique, par exemple, les fabricants conçoivent, développent et déploient de tels services Internet afin d'aider leurs clients à gérer leurs équipements électriques et afin de leur offrir (et facturer) des solutions pour optimiser leur consommation électrique.

Pour ce faire, les fabricants doivent créer des suites logicielles complexes et doivent, en outre, mettre en place et appréhender des environnements eux-aussi complexes. L'infrastructure ainsi mise en place doit, qui plus est, passer à l'échelle et être flexible, elle doit aussi appréhender les changements à venir de manière transparente et enfin, elle doit assurer que les experts électriques (et non informaticiens) pourront gérer les services Internet offerts aussi facilement que possible [Lal05].

La plupart des systèmes offrant des services Internet utilisant des données environnementales sont basés sur des architectures n-tiers. De telles architectures mettent en jeu une application web qui fonctionne sur un serveur Internet et qui est connectée à des passerelles pervasives au-dessus de protocoles basés sur IP (tel, par exemple, HTTP). Ces passerelles pervasives sont connectées à des équipements physiques via des bus (CAN, par exemple). Elles collectent, transforment et envoient régulièrement des données au serveur Internet. Une fois reçues par le serveur, ces données sont stockées et utilisées pour offrir les services Internet. La figure ci-dessous présente un exemple d'architecture n-tiers, en l'occurrence, il s'agit ici d'une architecture 5-tiers:

- le premier tiers (le plus en haut) correspond aux clients finaux,
- le tiers suivant est celui du serveur Internet,
- vient ensuite le tiers des serveurs de médiation,
- le quatrième tiers correspond aux passerelles pervasives (qui sont, par exemple, enfouies dans des usines et qui embarquent des environnements d'exécution)
- enfin, le cinquième et dernier tiers est celui des équipements.

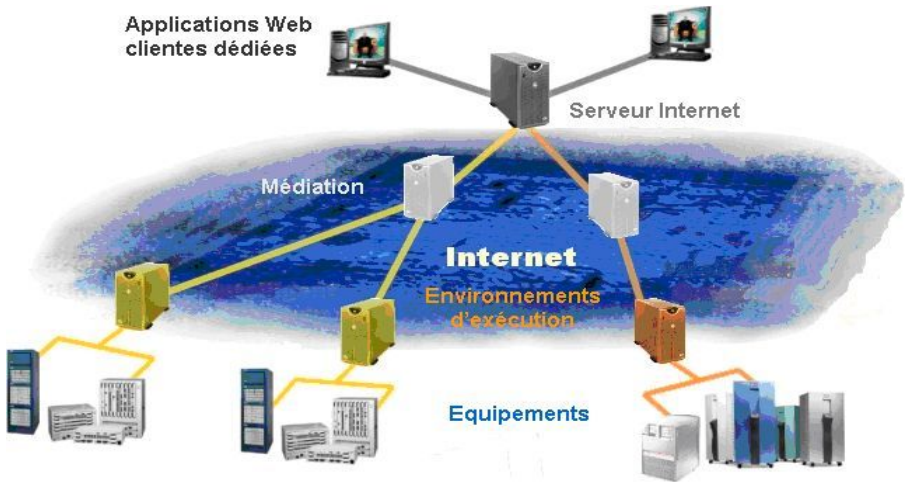


Figure 1. Exemple d'architecture n-tiers.

Dans notre contexte, nous nous sommes focalisés sur des passerelles pervasives qui embarquent, chacune, un environnement d'exécution. Ces environnements d'exécution sont des environnements d'exécution à services puisque les interactions entre services sont établies en suivant le patron SOA. Ces environnements d'exécution exécutent des (implémentations de) services, rendent accessibles les équipements connectés à la passerelle, orchestrent les actions liées aux équipements et gèrent les interactions avec le *IT* (c'est-à-dire l'informatique de gestion). Les environnements d'exécution à services ciblés définissent un modèle minimal de composant orienté service, un environnement basic pour l'administration (et donc le déploiement) de ces composants et un ensemble de services standards. L'utilisation d'un tel environnement d'exécution permet aux développeurs d'implémenter des services et des moyens d'interactions entre services hautement flexibles où équipements, services et interactions peuvent changer au fil du temps sans pour autant nécessiter le redémarrage (c'est-à-dire la désactivation et la réactivation) de l'environnement. Malheureusement, développer et déployer des services au-dessus de tels environnements est complexe et nécessite un haut niveau d'expertise. Typiquement, le déploiement de services, d'ensembles de services, et dans notre cas, d'applications à services est particulièrement difficile.

Mes travaux se basent aussi sur des environnements d'exécution à services car la dynamicité apportée par ces environnements est primordiale dans notre contexte. Certes, cette dynamicité est

combattue à l'intérieur d'une même application (pour assurer le respect de l'architecture de l'application et son état de déploiement), cependant à l'extérieur de l'application (et plus particulièrement au niveau des demandeurs promus au niveau de l'application), cette dynamicité est nécessaire puisqu'elle permet de brancher et débrancher des équipements à la passerelle, alors même que l'environnement embarqué dans la passerelle et les implémentations de services contenues dans l'environnement sont en cours d'exécution. Par exemple, pour l'application CEPC (présentée dans la partie validation de ce document), si l'équipement Voltmètre est branché à la passerelle, alors, CEPC (active) peut calculer la puissance électrique consommée, cependant, si le voltmètre est débranché, alors CEPC (active) doit retourner que le calcul est impossible puisque aucun voltmètre n'est branché ; ce qui est bien différent de : CEPC ne trouvant plus de fournisseur "voltmètre" s'est désactivée et donc ne répond plus aux requêtes "puissance électrique consommée". Dans le premier, le responsable de la non-fourniture du service Internet de haut niveau au client final sera le responsable chargé des branchements, dans le second cas, le responsable sera l'administrateur qui aura présenté CEPC comme active alors que dans les faits elle n'est plus active. L'exploitation de cette dynamicité est traitée par mon équipe de recherche Adèle dans [EBL+08].

Dans le cadre de cette thèse, nous ciblons une architecture n-tiers et en particulier, nous distinguons un tiers correspondant aux passerelles pervasives et un tiers de plus haut niveau dans l'architecture qui peut être, par exemple, le tiers serveur Internet ou tout du moins un tiers de niveau équivalent. L'idée est de permettre à un administrateur, situé devant un serveur Internet (ou équivalent), de commander des actions de déploiement destinées à être réalisées automatiquement par notre architecture de déploiement dans le tiers passerelles/environnements d'exécution.

Enfin, il est important de noter que, dans cette thèse, nous nous intéressons au déploiement d'applications à services. Des travaux relatifs aux interactions entre les tiers peuvent être trouvés dans [BDE+06] et [EBL+08].

2. PROBLÉMATIQUE

Notre problématique porte sur le déploiement d'applications à services sur des environnements d'exécution à services. Nous cherchons à rendre possible le déploiement d'applications à services et donc à concevoir et fournir une architecture et un système de déploiement utilisables par un administrateur souhaitant déployer des applications à services sur les environnements d'exécution qu'il administre.

L'état des lieux, le contexte et cette problématique font apparaître plusieurs défis :

- Le premier défi vise la définition d'une notion d'application à services permettant le "déploiement au plus près", la production d'un méta-modèle correspondant à cette définition et l'expression effective de telles applications (c'est-à-dire la création de modèles d'application à services dissociés de toute implémentation).
- Le second défi cible l'adaptation de la définition du déploiement que nous estimons être la plus complète et la plus cohérente (c'est-à-dire celle de l'université du Colorado [HHC+98]) avec pour but avoué de préciser la définition du déploiement pour l'approche à service et pour les services et applications à services.
- Le troisième défi vise l'intégration notre système de déploiement dans le contexte global présenté.
- Le quatrième défi cible le déploiement de modèles d'application à services. Nous verrons que nous nous sommes focalisés sur les activités d'installation, d'activation, de désactivation, de désinstallation d'applications à services et la mise à jour statique d'une manière générale.
- Le cinquième défi vise l'étude et la gestion du partage d'implémentations de services, de l'isolation entre applications et du partage d'instances d'implémentations de services.
- L'avant dernier défi cible la minimisation de la taille des applications à services déployées (nous parlons ici en espace disque) et du nombre d'implémentations de services packagées (c'est-à-dire d'unités de déploiement) déployées dans tout environnement.
- Enfin, le septième et dernier défi porte sur le déploiement de modèles d'application à services sur des environnements d'exécution à services et sur l'exécution d'un ensemble d'activités de déploiement concernant des modèles d'application à services (c'est-à-dire sur l'exécution de plusieurs activités de déploiement les unes à la suite des autres). Notre objectif est la proposition d'un algorithme d'exécution plus performant que l'algorithme d'exécution séquentiel trivial.

3. PROPOSITION

Dans le cadre de cette thèse, nous proposons un prototype appelé *Deployment Manager for Services Applications* (DMSA). Notre prototype met en œuvre l'approche dirigée par les modèles que nous avons choisie pour permettre et automatiser le déploiement (de modèles) d'applications à services sur des environnements d'exécutions à services. Pour effectivement procéder à cette mise en œuvre, il nous a été nécessaire de :

- Définir un méta modèle pour les applications à services dans notre contexte de "déploiement au plus près" (c'est-à-dire à définir une vue déploiement pour les applications à services satisfaisant notre contexte et le "déploiement au plus près"),
- Utiliser ce méta-modèle pour modéliser les applications, les implémentations de services (c'est-à-dire les unités de déploiement) et les environnements d'exécution à services.
- Définir le déploiement pour les applications à services dans notre contexte.

Il nous a aussi été nécessaire :

- D'appréhender ce déploiement, c'est-à-dire permettre l'installation, l'activation, la désactivation, la mise à jour statique et la désinstallation de modèles d'application sur un environnement d'exécution à services en agissant sur ces dits modèles et sur le modèle global de l'environnement.
- De gérer le partage d'implémentations de services et d'instances, ainsi que l'isolation entre les modèles d'application déployés.
- D'appréhender le contexte embarqué/réactif des environnements d'exécution à services que nous ciblons, c'est-à-dire maximiser le partage des implémentations de services et des instances d'implémentations de services dès que cela est possible.
- Et de permettre le déploiement sur plusieurs environnements, ainsi que l'exécution de listes d'activités de déploiement.

Ainsi, dans ce document de thèse, nous détaillons l'approche dirigée par les modèles que nous avons suivie, notre méta-modèle pour les applications à services, la définition du déploiement pour les applications à services issue de notre adaptation de la définition de l'université du Colorado, ainsi que l'architecture de notre système de déploiement. Nous détaillons aussi nos propositions autour du déploiement d'un, puis de modèles d'application sur un environnement, ainsi que notre appréhension du contexte embarqué des environnements et nos propositions concernant l'exécution (de listes) d'activités de déploiement sur des environnements.

4. ORGANISATION DU DOCUMENT

Ce document est organisé comme suit. Après cette introduction, le manuscrit est divisé en trois parties : l'état de l'art, la contribution et la réalisation.

La première partie présente l'état de l'art. Les travaux de cette thèse se situent à la convergence de l'approche à service et du déploiement logiciel. Pour cela, l'état de l'art est découpé en deux chapitres :

- Le chapitre 1 présente l'approche à services. Nous détaillons son positionnement historique, ses éléments fondamentaux, puis les deux visions majeures de l'approche à services actuellement promues via les trois technologies orientées services majeures (les services Web, OSGi et SCA), ainsi que via les technologies JINI et UPnP.
- Le chapitre 2 détaille le déploiement logiciel. Nous le détaillons au travers des trois définitions majeures (celle de C. Szyperski, celle de l'OMG et celle de l'université du Colorado), ensuite, nous présentons les points de caractérisation du déploiement sur lesquels nous nous sommes focalisés, puis nous les appliquons au déploiement de composants (sur les technologies CCM, JEE EJB et .Net) et au déploiement de services (via les technologies OSGi et SCA).

La seconde partie concerne notre contribution. Elle est découpée en deux chapitres :

- Le chapitre 3 se focalise sur la problématique de cette thèse. Il rappelle les quatre constats de la partie état de l'art, précise le contexte de nos travaux et explicite la problématique.
- Le chapitre 4 présente notre proposition. Il explicite notre approche, le méta-modèle pour les applications à services, la définition du déploiement pour les applications à services, l'architecture du DMSA et nos propositions. Ces dernières ciblent le déploiement d'un, puis de plusieurs modèles d'application à services sur un environnement, le déploiement "transactionnel", l'appréhension de notre contexte embarqué/réactif et l'exécution de listes ordonnées d'activités de déploiement sur des environnements d'exécutions à services.

La troisième et dernière partie présente notre réalisation. Elle est composée de deux chapitres :

- Le chapitre 5 présente la réalisation de nos propositions, le prototype DMSA (*Deployment Manager for Services Applications*).
- Le chapitre 6, quant à lui, présente la validation de nos propositions. Pour ce faire, nous déroulons le déploiement d'un modèle d'application à services "type" : l'application CEPC (*Current Electrical Power Consumed*), dans un et plusieurs environnements d'exécution à services.

Enfin, le chapitre conclusion est le dernier chapitre de cette thèse. Il présente la synthèse des principales idées de notre proposition. A cette occasion, nous soulignons les principales contributions de cette thèse et nous identifions les questions ouvertes et les perspectives de ce travail.

PREMIÈRE PARTIE : ETAT DE L'ART

I. L'APPROCHE À SERVICE

Dans ce second chapitre, nous présentons un état de l'art sur l'approche à service et les applications à services. L'approche à service est un nouveau paradigme informatique dont le but est l'augmentation de la modularité des applications composées d'entités logicielles et la réduction du couplage entre ces mêmes entités logicielles.

Nous allons voir que plusieurs visions de l'approche à service coexistent. C'est pourquoi, nous commençons tout d'abord par expliciter quels sont les points consensuels de cette approche. Ensuite, nous verrons quelles sont les trois visions majeures de l'approche à service. La première vision est celle proposée par le *World Wide Web Consortium* (W3C), où un service décrit ce qu'il propose, mais où aucun mécanisme de composition de services n'est envisagé. La seconde vision est celle promue par OSGi via le concept de composant orienté service qui introduit un mécanisme de composition de service. Enfin, la troisième et dernière vision est celle proposée par SCA, qui, en plus d'utiliser le concept de composant orienté service, introduit le concept de composite orienté service.

Ce chapitre comprend huit sections. La première section positionne historiquement l'approche à service par rapport aux approches objet, à composant et également par rapport à l'approche (dirigée) par les modèles. La seconde détaille les points que nous considérons comme consensuels dans l'approche à service. Nous définirons ainsi le concept de service, les accords de niveau de service, l'architecture orientée service et les six propriétés remarquables de l'approche à service. La troisième section présente la vision de l'approche à service du point de vue du consortium W3C. Les quatrième et, respectivement, cinquième sections présentent, quant à elles, la vision de l'approche à service du point de vue des technologies OSGi et, respectivement, SCA. La sixième section présente d'autres technologies à services. La septième section synthétise les différentes visions présentées par le biais, entre autre, d'un tableau et détaille les avantages et limites de ces trois visions. Enfin, la huitième section conclut cet état de l'art sur l'approche à service et présente les défis actuels.

1. POSITIONNEMENT HISTORIQUE

Dans cette première section, nous positionnons l'approche à services par rapport aux approches objet et à composant. Nous évoquons également l'approche (dirigée) par les modèles.

1.1 L'approche objet

Dans les années 1980, l'informatique a vu l'introduction de l'approche objet pour la modélisation, la conception et la programmation logicielle [Mey99]. Selon cette approche, un objet est défini comme une entité logicielle à part entière lors des phases de développement et d'exécution. Un objet possède un grain fin, est réutilisable, encapsule des attributs et propose des méthodes d'accès. De plus, un objet supporte les concepts de polymorphisme, d'héritage, de surcharge et de redéfinition. Ces propriétés sont les apports principaux de l'approche objet. Cependant, bien qu'ayant rencontré un succès réel, cette approche a aussi plusieurs limites. Ainsi, une limite importante de l'approche objet réside dans la trop grande finesse du grain des objets qui nuit, voire empêche, la réutilisation. Une autre limite concerne l'absence d'un niveau application à part entière. En effet, l'aspect architectural n'apparaît pas dans la plupart des méthodes de conception et de développement orientées objet (la notion de composant logiciel n'est apparue réellement que dans UML 2.0, par exemple).

1.2 L'approche à composant

Une dizaine d'années plus tard, l'approche à composant est introduite. Son objectif est de pallier les limites de l'approche objet, tout en conservant ses principaux avantages. Ainsi, un composant est défini comme étant une entité logicielle à part entière tout au long du cycle logiciel, qui se veut réutilisable. Il présente ses fonctionnalités via des interfaces fournies et requiert des fonctionnalités par le biais d'interfaces requises [Szy03]. En conséquence, le grain logiciel est fixé, non plus par la portée de l'objet représenté, mais par la portée des fonctionnalités offertes et requises par le composant. De fait, le grain des composants est plus gros que celui des objets, améliorant ainsi la réutilisabilité des composants. L'exemple le plus notable concerne la réutilisation des composants dans les lignes de produits. En effet, le développement d'un produit au sein d'une ligne de produits se fait en (ré-) utilisant un socle de composants communs à toute la ligne [Cle02]. Contrairement à l'approche objet, l'approche à composant possède clairement un niveau application. Cela est démontré par la multitude des langages de description d'architecture (ADL), tels 4+1 Rational, Wright, C2, ACME [MT00] ou des descripteurs d'applications qui peuvent être trouvés dans diverses technologies à composant (.Net, EJB, Fractal, etc.).

Cependant, bien que palliant certaines limites de l'approche objet, l'approche à composant souffre elle aussi de plusieurs limites. En premier lieu, nombre de technologies telles celles des *Enterprise Java Bean* (EJB) par exemple n'appréhendent les composants comme entités logicielles à part entière que lors des phases de développement et d'exécution. Ainsi, les composants constituant les applications à composants apparaissent clairement lors des phases de développement et d'exécution, mais disparaissent bien souvent dans le monolithe formé par le package de l'application qui sert de point d'entrée à la phase de déploiement. L'exemple le plus explicite à ce propos est celui des applications basées sur les EJB. En effet, une application EJB présente clairement le grain EJB lors de son développement et lors de son exécution. Cependant, ce grain disparait lors du déploiement de l'application, puisque cette dernière est packagée sous la forme d'un fichier archive monolithique. La disparition du grain composant limite les possibilités de déploiement de l'application : seules les activités d'installation, d'activation, de désactivation et de désinstallation de l'application (entière) sont possibles. L'activité de maintenance (composée des sous-activités de mise à jour statique, de mise à jour dynamique, d'évolution, de dynamisme) est, elle, rendue délicate. Nous reviendrons sur ces différents points liés au déploiement dans le prochain chapitre.

Une autre limite des composants est que, bien que ces derniers soient des entités logicielles *a priori* spécialisées (en fonction des fonctionnalités qu'ils offrent), rien dans l'approche à composant ne les forcent à l'être. En effet, un même composant peut tout à fait fournir plusieurs interfaces couvrant des

préoccupations distinctes tout en les implémentant, par exemple, via un seul et même bloc algorithmique. De ce fait, la réutilisation des fonctionnalités d'une interface d'un composant force la réutilisation du composant dans son ensemble, alors que seules les fonctionnalités d'une interface sont réellement la cible de la réutilisation. De plus, l'usage des différentes interfaces d'un même composant ne peut pas être garanti, puisque les algorithmes réalisant ces interfaces peuvent interagir entre eux et donc modifier (de manière volontaire ou non, connue ou non, explicite ou non) le résultat d'une fonctionnalité offerte par une interface suite à l'utilisation de fonctionnalités d'une autre interface.

1.3 L'approche à service

Les années 2000 ont vu l'introduction de l'approche à service ou informatique orientée service (*Service-oriented Computing*, SOC). Un service est défini comme une entité logicielle à part entière tout au long du cycle de vie logiciel, dont les fonctionnalités doivent être spécifiées dans un contrat et dont l'utilisation doit se faire au sein d'une architecture à service (*Service-oriented Architecture*, SOA) [PTD+06]. Nous pensons que cette approche permet de pallier, d'une part, le fort couplage qui peut exister entre les composants d'une application et qui restreint notamment les activités de déploiement et, d'autre part, les conséquences du manque de spécialisation des composants qui nuit à leur réutilisabilité. L'approche à service met en avant les propriétés de transparence de la substituabilité, de couplage faible et de liaison retardée et impose l'existence et l'utilisation d'un contrat unique, mais négociable, pour chaque service. La description d'un service par un contrat permet de passer de composants fournissant plusieurs interfaces à des services ne fournissant qu'un contrat, augmentant de ce fait la spécialisation des services et donc leur réutilisabilité. L'introduction des contrats permet aussi de passer de dépendances vers/entre composants à des dépendances vers des contrats, diminuant de ce fait le couplage entre services par rapport au couplage entre composants.

Les sections suivantes de ce chapitre proposent un état de l'art détaillé de l'approche à service.

1.4 L'approche dirigée par les modèles

Enfin, depuis 2000, une nouvelle approche a été introduite. Elle est dite approche (dirigée) par les modèles ; elle est aussi dénommée "Ingénierie Dirigée par les Modèles" (IDM) ou *Model Driven Engineering* (MDE) en anglais. Elle a été introduite pour faire face à la compléxité et au besoin d'évolution croissants des applications [KWB03] [MSU+04]. Elle vise à favoriser une démarche qui soit plus proche des métiers ; pour se faire, les applications sont appréhendées selon des points de vue exprimés séparément : des modèles. La composition et la mise en cohérence de ces modèles sont des préoccupations qui font aussi parties des problématiques de l'IDM [Fav04][Sei03]. L'IDM veut être une approche productive puisqu'elle cible, aussi, la prise en charge des outils servant à valider et transformer des modèles, de même, que les outils de génération de code à partir de modèles.

Alors que l'approche orientée objet est fondée sur deux relations essentielles, "InstanceDe" et "HériteDe", l'ingénierie dirigée par les modèles est basée sur un autre jeu de concepts et de relations. Un consensus est en train de se former autour de deux relations : la relation "RepréstationDe" (μ) menant à la notion de modèle et la relation "ConformeA" (χ) menant à la notion de méta-modèle [BGM+03]. De plus, le fait qu'un méta-modèle soit un modèle d'un langage de modélisation est également un fait admis.

La notion de transformation est un autre concept central pour l'IDM. Décrire comment transformer un modèle est une condition indispensable pour le rendre productif. Bien que dans [Fav04] une troisième relation nommée "TransforméEn" (τ) soit introduite, il n'existe pas à ce jour de définition ou de terminologie consensuelle concernant les transformations. Les termes modèles de transformations, langages de transformation, correspondances, etc. sont souvent utilisés de manière peu cohérente.

De façon synthétique, l'approche dirigée par les modèles doit être vue comme une intégration, un prolongement et un renforcement d'approches déjà connues. Ses éléments premiers sont les concepts de modèles, de langages, de méta-modèles, de transformations et d'interprétations [FEF06].

L'approche par les modèles propose que la résolution informatique d'un problème se fasse en s'abstrayant du niveau implémentation pour ne raisonner qu'à un niveau où seules les informations

relatives au problème interviennent. Le fruit du raisonnement est, ensuite, spécifié sous la forme d'un méta-modèle qui contient les informations relatives au problème, ainsi que celles nécessaires à la solution retenue. Puis, la solution retenue est projetée vers le niveau implémentation afin de la réaliser concrètement. Cette projection se fait via le développement d'un logiciel qui prend en entrée tout modèle conforme au méta-modèle et qui produit le résultat correspondant à la solution retenue en sortie. La vision la plus connue de l'IDM est celle proposée par l'*Object Management Group* (OMG) via sa spécification *Model Driven Architecture* [OMG03]. L'intérêt de l'approche par les modèles est qu'elle permet un découplage total de la solution du problème par rapport au niveau implémentation (qu'il soit objet, composant, service, etc.). En conséquence, la solution retenue devient indépendante d'une technologie cible particulière et il est ainsi possible de l'appliquer sur toute autre technologie dès lors que celle-ci supporte les hypothèses de la solution et qu'il est possible de développer un *wrapper* technologique.

De plus amples informations à propos de L'IDM peuvent être trouvées dans [AK03] [Fav04a] et [Sei03].

Aujourd'hui, l'IDM peut être structurée en six sous-domaines :

- "Le génie logiciel et l'IDM" qui s'intéresse au cadre méthodologique et technologique que l'IDM peut offrir au génie logiciel, avec un attrait tout particulier pour la proposition d'un cadre méthodologique unifié/unificateur. [INRIA05] [Bos98] [LSJ00] [MB02] et [NE00] sont relatifs à ce sous-domaine.
- "Les plates-formes d'exécution et l'IDM" qui se focalise principalement sur ce qu'est une plate-forme, quels sont les concepts qui lui sont associés et sur la proposition d'une définition unificatrice [BBM04] [BCG+04] [DRD99] [Hud98] et [OMG01].
- "Les langages et l'IDM" qui investigue les travaux de recherche issus de la compilation des langages de programmation pour identifier les principes méthodologiques et les fondations théoriques réutilisables dans la démarche IDM. En effet, l'IDM et les langages partagent les mêmes préoccupations, or ces derniers ont accumulé, en quatre décennies d'efforts, une somme considérable de résultats scientifiques et techniques applicables directement ou indirectement à l'IDM [AJ74] [FBP+03] [KWB03] et [Sei03].
- "Les bases de données et l'IDM" qui investigue aussi les travaux issus des bases de données. En effet, les concepts de modèle, de méta-modèle, de niveaux d'abstraction multiples et de transformations sont au coeur des problématiques des bases de données. En conséquences, les résultats et les succès engrangés par les efforts menés sur le domaine des bases de données sont très largement réutilisables dans le cadre de l'IDM [BDC00] [Che76] et [Ull89].
- "Le temps réel embarqué et l'IDM" qui s'intéresse à l'identification des liens entre le domaine du temps réel et l'IDM et aux apports fournis par l'utilisation de l'IDM dans le domaine du temps réel (en particulier concernant la spécification et le prototypage d'applications temps réelles embarquées) [CNRS88] [GMT+04] [Sta88] et [TGM03].
- et "l'évolution, la rétro-ingénierie et l'IDM" qui se focalise sur les points communs entre l'évolution, la rétro-ingénierie et l'IDM, ainsi que sur les apports de l'IDM pour l'évolution et la rétro-ingénierie et réciproquement [BP01] [CC90] [EVI05] et [Fav96].

De façon plus générale, l'IDM doit, aujourd'hui, apporter des solutions à plusieurs grandes problématiques :

- La gestion des modèles et des associations entre modèles (méga-modélisation) est une des problématiques. Celle-ci englobe, en outre, la composition de modèles, la gestion de la synchronisation entre les modèles et la gestion des erreurs.
- L'évolution des modèles
- La validation des modèles au regard du problème abordé, ainsi que la validation des interprétations et transformations.
- L'instrumentation des approches dirigées par les modèles, avec une préoccupation toute particulière concernant l'ergonomie des outils produits.
- Et enfin, les modèles à l'exécution.

Enfin, en guise de remarque, il est important de distinguer le MDA qui est une collection de standards industriels promue par l'OMG, de l'IDM qui est une approche intégrative générale.

L'enchainement historique des quatre approches que nous venons de décrire succinctement se retrouve dans le diagramme étendu des vagues de Racoon représenté dans la figure ci-dessous.

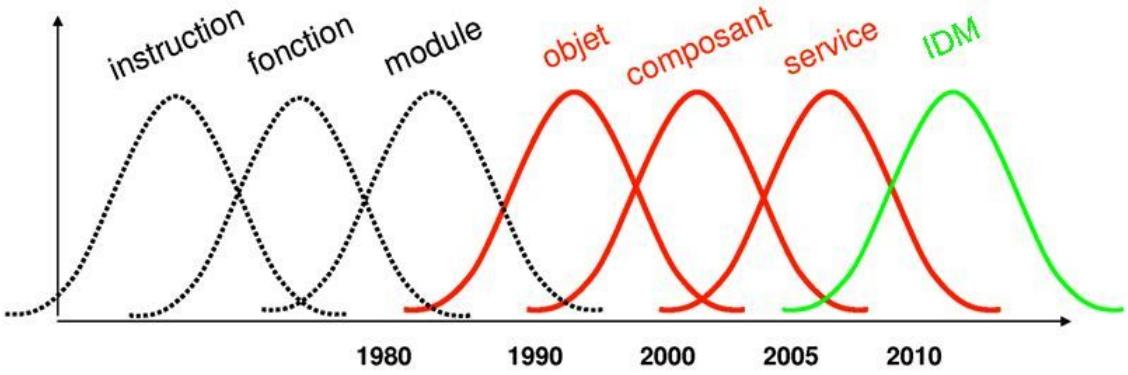


Figure 2. Diagramme étendu des vagues de Racoon [Don06].

Nous verrons dans cette thèse que l'approche dirigée par les modèles est une approche novatrice et originale qui peut être utile dans le cadre du déploiement de services et d'applications à services.

2. LES FONDAMENTAUX

Dans cette seconde section, nous présentons certains éléments que nous considérons comme consensuels dans l'approche à service. Tout d'abord, nous nous focalisons sur le concept de service. Puis, nous présentons le concept d'accord de niveau de service (*Service Level Agreement*, SLA) qui entre en jeu au niveau du contrat proposé par un service et qui est négocié entre demandeurs et fournisseurs de service. Après cela, nous nous intéressons à l'architecture orientée service (*Service-oriented Architecture*, SOA) et au patron architectural qui doit être utilisé par le demandeur et le fournisseur d'un service afin d'établir une interaction au niveau service. Enfin, nous développons six propriétés consensuelles et remarquables qui caractérisent l'approche à service.

2.1 Service

Le concept de service a donné lieu à plusieurs définitions. Les quatre définitions que nous présentons ici sont, dans l'ordre, celle de M. Papazoglou [PTD+07], d'A. Arsanjani [Ars04], de M. Bichier et K.-J. Lin [BL06] et, enfin, de M. Huhns et M. P. Singh [HS05].

Dans [PTD+07], les services sont définis comme des entités logicielles autonomes et indépendantes de toute plateforme. Les services peuvent être décrits, publiés, découverts et couplés de manière lâche. La publication et la découverte de services se fait via le patron SOA (*Service-oriented Architecture*). Le patron SOA met en jeu trois entités : le fournisseur de service, le demandeur de service et un registre de contrat de service. Le fournisseur de service enregistre son contrat de service dans le registre de service. Le demandeur de service recherche le contrat de service qu'il souhaite utiliser dans le registre de contrat de service. Si un fournisseur correspondant au contrat est trouvé, le demandeur initie l'établissement d'une liaison avec le fournisseur. Enfin, le demandeur et le fournisseur d'un service doivent être tous deux d'accord sur la description du service et sur la sémantique qui régira leur interaction.

Dans [Ars04], un service est défini comme une ressource logicielle découvrable possédant une description. Cette description permet à un service d'être recherché puis lié et invoqué par un consommateur de service. Elle est effectivement réalisée par un fournisseur de service par le biais de son implémentation. C'est aussi le fournisseur du service qui spécifie les requis de qualité du service et les transmet au consommateur du service.

Dans [BL06], le concept de service est défini à travers les rôles de demandeur et de fournisseur de service. De plus, un service peut décrire ses capacités (fonctionnelles et non-fonctionnelles) en utilisant un modèle sémantique. Ensuite, les services doivent être découvrables afin de permettre leur composition. Enfin, les services logiciels doivent être accessibles au moyen de protocoles de communication standards.

Enfin, dans [HS05], les services sont vus comme des entités offrant un haut niveau d'abstraction. Ils peuvent encapsuler le comportement de composants à plusieurs niveaux, tout en les décrivant de la même manière, facilitant par là-même la composition de ces composants. L'architecture pour les applications orientées service est constituée de trois acteurs principaux, à savoir, le fournisseur, le demandeur et le registre de service. Le fournisseur de service publie ses services dans un/des registres de service. Le demandeur de service, quant à lui, recherche un service dans ce/ces registre(s) et invoque celui ou ceux qu'il aura sélectionnés.

Au final, plusieurs éléments consensuels au niveau du concept de service peuvent être retenus. Ainsi, un service peut être défini comme étant :

> Une entité logicielle à part entière tout au long du cycle de vie logiciel, spécifiée par un contrat de service (décrivant généralement au moins les fonctionnalités de l'entité, c'est-à-dire ce qu'elle fournit) publiable et découvrable qui permettra ensuite l'établissement d'une liaison et l'interaction entre l'entité et le ou les consommateurs/demandeurs du contrat de service. Enfin la publication et la découverte des contrats de services et l'établissement de liaisons s'effectuent en suivant le patron SOA.

Les objectifs de l'introduction du concept de service et du SOC sont la réduction du couplage entre entités logicielles, ainsi que l'augmentation de leur spécialisation afin d'assurer la modularité des applications basées sur ces entités tout au long du cycle logiciel et d'augmenter la réutilisabilité de chaque entité. La finalité du SOC, d'une manière générale, est de minimiser les coûts engendrés par une application tout au long de son cycle logiciel et cela tout en ne restreignant pas les types d'applications possibles (c'est-à-dire en permettant la création d'applications distribuées, interopérables et évolutives).

Enfin, la portée fonctionnelle des services (et donc de leur contrat) peut varier d'un service à l'autre. En effet, un service peut offrir des fonctionnalités allant de la simple réponse à une requête basique à l'exécution de fonctionnalités métiers avancées requérant l'usage de plusieurs autres services (le contrat de service spécifiera alors, au moins, ce qui est fourni et ce qui est requis). La construction d'un logiciel basé sur le concept de service doit se faire en maximisant, autant que faire se peut, la réutilisation des services existants. Tout morceau de code existant peut être transformé en un service. Pour cela, il suffit de définir le contrat correspondant au code ciblé par la transformation et de packager l'ensemble formé par le code et le contrat de façon à ce qu'il puisse être exécuté dans un environnement d'exécution à services et cela en accord avec le patron SOA (Architecture Orientée Service ou *Service-oriented Architecture*). A titre d'exemple, le contrat de service d'un web service décrit un fournisseur de service et le moyen d'interagir avec lui ; le contrat de service d'un composant orienté service peut être de portée, soit similaire à celui d'un web service, soit couvrant ce qui est fourni par le composant, le composant lui-même et ce qui est requis par le composant.

Dans cadre de cette thèse, chaque modèle d'application à services que nous manipulons peut être vu comme un contrat de service (orienté déploiement) de l'application ; pour effectivement déployer ces contrats nous nous basons sur des contrats de service de composants orientés service contraints (COSC) qui correspondent à des implémentations (de service) et nous nous basons aussi sur des environnements d'exécutions à services qui exécutent des composants orientés service ; une fois activé, un composant orienté service enregistre le contrat de service de chacun de ses fournisseurs dans le registre de l'environnement (les demandeurs de services des composants peuvent alors les rechercher dans le registre, établir une liaison et interagir avec les fournisseurs enregistrés).

2.2 Accord de niveau de service

Un accord de niveau de service est le résultat de la négociation entre un fournisseur et un demandeur de service. Cette négociation est basée sur le contrat de service fourni par le fournisseur et sur celui requis par le demandeur du service. Afin de représenter cette étape de négociation, ainsi que l'expression des contrats de service, le terme *Service Level Agreement* (SLA) a été introduit. Le SLA concerne le niveau du contrat de service proposé par le service, le niveau de la négociation qui aura lieu entre demandeur et fournisseur de service et le niveau de l'utilisation du service. En effet, comme indiqué dans [SDE04], l'utilité et parfois même le fonctionnement d'un service dépend, non seulement des fonctionnalités rendues, mais aussi de la qualité du service qui est rendu. A titre d'exemple, la qualité d'un service peut porter sur la disponibilité du service ou sur la fiabilité des données qu'il fournit. Ces qualités dépendent, certes, du comportement du fournisseur du service, mais aussi de celui du client. C'est pourquoi, un contrat entre le fournisseur d'un service et le client du service doit exister et doit contenir les clauses régissant leurs responsabilités individuelles et mutuelles à l'égard des fonctions et qualités du

service. C'est dans le cadre de ce contrat qu'interviennent les accords de niveau de service. A ce propos, dans [BJP+99], quatre niveaux d'accord de niveau de service sont distingués :

- le niveau syntaxique qui est constitué des signatures des méthodes offertes par l'entité logicielle correspondant au contrat, c'est-à-dire le nom des méthodes, le type des paramètres d'entrée et de sortie et le type des exceptions qui peuvent être levées. Un contrat de niveau syntaxique peut être, par exemple, une interface programmatique.
- le niveau comportemental qui étend le premier niveau en introduisant l'usage de pré/post-conditions et d'invariants. Ainsi chaque méthode définie dans le niveau syntaxique se retrouve décorée par des pré-conditions, des post-conditions et des invariants.
- le niveau synchronisation qui concerne l'enchaînement des appels entre les méthodes d'un contrat de second niveau. Ce niveau de contrat précise comment l'entité logicielle qui propose le contrat offre ses fonctionnalités au client.
- enfin, le niveau qualité de service qui ajoute des facteurs, des critères et des métriques de qualité à un contrat de niveau trois. Les facteurs ajoutés peuvent porter sur la conformité, l'efficacité, la disponibilité des fonctionnalités offertes via le contrat. Les critères associés aux facteurs peuvent, par exemple, être la consistance, la complétude, le temps d'exécution, l'accessibilité. Ces critères sont ensuite quantifiés sous la forme de métriques, par exemple, la précision de la réponse à une demande, ainsi que le temps maximal (ou moyen) de réponse, le pourcentage de demandes perdues.

2.3 Architecture orientée service

La mise en œuvre des services est régie par une architecture orienté service (*Service-oriented Architecture*, SOA) qui définit un patron (ou *pattern*) architectural pour la publication, la recherche (ou découverte) et l'établissement de liaison entre un fournisseur et un demandeur de service. Ce patron met en jeu trois entités qui sont un registre de service, un fournisseur de service et un demandeur de service (comme il est possible de le voir sur la figure ci-dessous). Son but est de permettre la composition d'un fournisseur et d'un demandeur de service, aussi appelée composition de service [BDD+04] [TBB03].

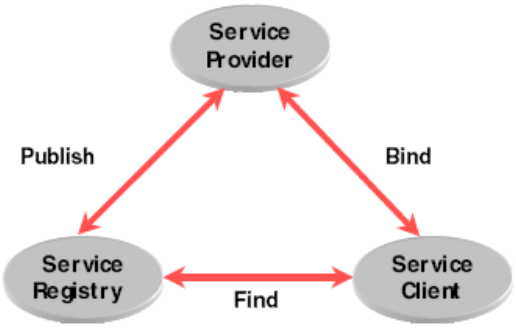


Figure 3. L'architecture orientée service [Pap03]

Le patron de base est relativement simple. Un fournisseur de service publie son contrat de service dans un registre de service et peut être amené à négocier un accord de service avec un demandeur pour effectivement offrir/vendre son service. Un demandeur de service, quant à lui, cherche à satisfaire son requis de service. Pour ce faire, il recherche un fournisseur (correspondant à son requis de service) dans un registre de service, négocie un accord de service avec le fournisseur et établit une liaison avec ce fournisseur afin de l'utiliser. Le résultat de la négociation entre un fournisseur et un demandeur est un accord de service qui explicite l'utilisation du service. Il est à noter que toute entité logicielle peut être simultanément fournisseur et demandeur de service.

Rappelons que dans [PTD+07] et [HS05], le patron mentionné ci-dessus est un élément à part entière de la définition du concept de service.

Plusieurs types d'architectures à service ont été définis. Ils se distinguent par les supports fournis à la composition et par l'étendue des compositions possibles (par exemple, distribuée ou non, hétérogène ou non). En particulier, il existe deux approches pour créer effectivement une composition de service. Ces approches sont l'orchestration et la chorégraphie [BDD+04] :

- Une orchestration décrit le comportement implémenté par un fournisseur de service pour offrir un service. Elle se focalise sur un service particulier. La logique de contrôle est centralisée au niveau du fournisseur de service qui implante le comportement. Les interactions avec des tiers sont décrites, ainsi que les actions internes que doit exécuter le fournisseur de service. Cette orchestration doit ensuite être exécutée par un moteur d'orchestration. BPEL4WS (*Business Process Execution Language for Web Services*) est un exemple de langage typique pour la description des interactions entre un fournisseur de service et des tiers dans le cadre d'une orchestration [IBM07] [WVD+02].
- Une chorégraphie décrit une collaboration entre des services afin d'aboutir à un objectif commun. De ce fait, elle ne se focalise pas sur un service, mais plutôt sur l'objectif. La logique de contrôle est donc distribuée sur l'ensemble des services impliqués, lorsque les services interagissent entre eux. Pour décrire une chorégraphie, il faut tout d'abord définir les interactions entre services puis les liens entre ces interactions. Les actions réalisées en interne par les services ne sont pas décrites. WS-CDL (*Web Services Choreography Description Language*) est un exemple de langage de description de chorégraphie [W3C05] [BDO05].

L'orchestration est très utilisée dans le cadre des services web. Dans la prochaine section, nous présenterons en détail la vision de l'approche à service promue par les services web et le W3C et leur mise en œuvre du patron SOA. De son côté, la chorégraphie est utilisée principalement dans le cadre des composants orientés service tels SCA et OSGi. Les prochaines sections présenteront également leur vision de l'approche à service et leur mise en œuvre du patron SOA.

2.4 Propriétés remarquables de l'approche à service

Pour finir, nous présentons ci-dessous six propriétés consensuelles et remarquables qui sont mises en avant par l'approche à service. Ces propriétés sont :

- Le couplage faible. Il permet à une entité logicielle de spécifier une dépendance, non pas vers une autre entité, mais, vers un contrat de service. Ce faisant, il devient possible de choisir quelle entité logicielle va effectivement fournir le contrat à l'entité qui a spécifié la dépendance. Ce choix peut revenir, soit à l'utilisateur de l'entité qui spécifie la dépendance, soit à l'entité qui spécifie la dépendance elle-même, soit à une tierce partie.
- La liaison retardée (*Late Binding*). Elle consiste à lier un fournisseur et un demandeur uniquement lorsque ces derniers sont activés. C'est la forme d'établissement de liaison la plus retardée qui existe. En guise de comparaison, une liaison peut être créée lors du développement (via des aspects par exemple), lors de la compilation, lors du packaging, etc.
- La substituabilité. Elle consiste à assurer la transparence du remplacement d'une entité logicielle par une autre entité logicielle de contrat équivalent. La portée de la propriété de substituabilité dépend principalement de la portée du contrat.
- Les accords de niveaux de service (SLA). Cette propriété assure que toute entité logicielle peut être contractuellement définie. Sachant, bien sûr, que plusieurs niveaux de contrats existent. De même, la portée des contrats peut varier.
- Les propriétés de courtage et négociation. Elles permettent d'assurer la libre négociation au niveau des services. Ce courtage et cette négociation pouvant porter sur le choix des parties de l'interaction, sur le choix de l'interaction et sur le choix du maintien de l'interaction établie.
- Et enfin la propriété concernant les domaines d'administration. Cette dernière assure qu'un demandeur de service et un fournisseur de service appartenant à des domaines d'administrations disjoints vont pouvoir établir une liaison et interagir.

Ces propriétés nous serviront, notamment, pour caractériser les technologies à service présentées dans les prochaines sections.

3. LES SERVICES WEB

Dans cette section, nous présentons la vision de l'approche à service du *World Wide Web Consortium* (W3C). Le noyau central de cette vision est défini dans la spécification *Web Services Description Language Version 2.0* (WSDL V2.0) [W3C07]. Au cours de cette présentation, nous allons voir comment la définition du concept de service, le SLA, le SOA et les six propriétés remarquables du SOC sont appréhendées par le W3C. Nous verrons également si la vision du W3C est dépendante d'un langage de programmation et si la construction d'applications à services récursives (ou non) est possible.

3.1 Définition

De façon très générale, un service web est une entité logicielle à part entière qui peut être utilisée, à distance, par un client et dont les fonctionnalités sont spécifiées par l'intermédiaire d'une interface jouant le rôle de définition abstraite. Plus précisément, le W3C définit les services web comme étant un moyen standard permettant l'interopération d'applications logicielles s'exécutant dans des plateformes variées. Ils reposent sur des technologies standards de l'Internet, telles XML, HTTP/SOAP et sont caractérisés par leur grande interopérabilité et extensivité, ainsi que par leurs descriptions compréhensibles par les machines. Ils peuvent, en outre, être combinés afin de réaliser des opérations complexes, mais tout en restant faiblement couplés [W3C02].

Le W3C a aussi défini la spécification WSDL V2.0. Le but de cette spécification est l'établissement d'un langage de description des services web. Pour ce faire, la spécification WSDL V2.0 définit un modèle (cf. figure ci-dessous) et un formatage XML, c'est-à-dire un langage permettant de décrire, d'une part, un niveau abstrait qui spécifie la fonctionnalité offerte par un service et, d'autre part, un niveau concret où sont spécifiés les détails concrets de la description du service, tels que "comment" et "où" la fonctionnalité abstraite est offerte. Le but de ce découpage en deux niveaux est, d'une part, de séparer les préoccupations liées à la description abstraite du service de celles liées à son implémentation concrète et, d'autre part, de permettre la réutilisation de la description abstraite. Enfin, la spécification définit des critères de conformité pour les documents écrits en WSDL V2.0.

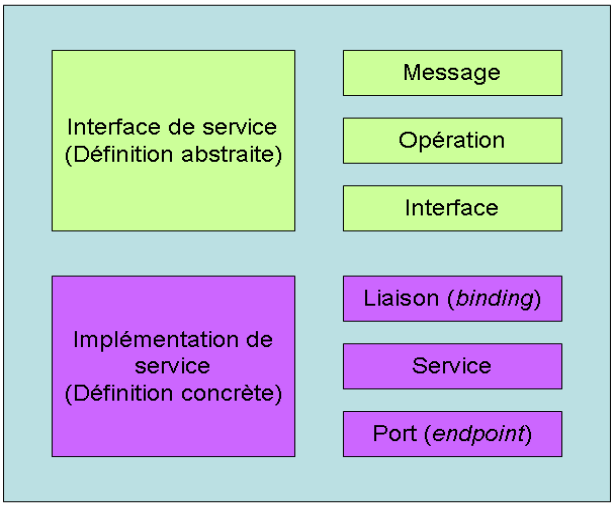


Figure 4. Modèle conceptuel d'un service web en WSDL V2.0.

Au niveau abstrait, un service web est décrit en fonction des messages qu'il envoie et qu'il reçoit. Les messages sont spécifiés en utilisant un système de type (concrètement un schéma XML) afin de les rendre indépendants de toute technologie de communication. Ensuite, le WSDL V2.0 décrit les opérations associées aux messages précédemment définis. Chaque opération associe un patron d'échange de messages avec un ou plusieurs messages. Puis, chaque patron d'échange de messages identifie la séquence

(c'est-à-dire l'ordre) et la cardinalité des messages envoyés et/ou reçus, ainsi que leurs émetteurs et récepteurs logiques. Enfin, les opérations sélectionnées sont regroupées dans une même interface. La définition abstraite du service est formée par cette interface.

Au niveau concret, une liaison (*binding*) spécifie la technologie de communication à employer pour une opération associée à une interface. Un port (*endPoint*) associe une adresse réseau à chaque liaison. Tous les ports qui sont associés à une même interface sont regroupés dans un même service.

Le but d'une description WSDL V2.0 d'un service est d'indiquer comment les clients potentiels sont destinés à interagir avec le service associé à la description. Ainsi, la description d'un service est une assertion quant au fait que le service décrit est conforme et implémente entièrement sa description. Par exemple, si une description spécifie une extension optionnelle, cela signifie que, d'une part, le client peut l'utiliser et que, d'autre part, le service doit effectivement l'implémenter. L'interface d'un service web décrit les interactions potentielles de ce service web et non les interactions requises par ce service web. La déclaration d'une opération dans une interface n'est pas une assertion concernant le fait que l'interaction décrite par l'opération va effectivement se produire. Cette déclaration d'opération est une assertion sur le fait que si l'interaction correspondante est initiée, alors, l'opération déclarée décrit comment l'interaction est sensée se dérouler.

3.2 Spécification d'un service web

Nous allons maintenant détailler les étapes qui doivent être suivies afin de spécifier un service web. Il convient de commencer par la définition du *target namespace* du service web. Ce *target namespace* identifie de manière unique un service web. Cet identifiant est une URI (*Uniform Resource Identifier*). La figure ci-dessous précise la portée d'une URI, par rapport à une URL (*Uniform Resource Locator*) ou à une URN (*Uniform Resource Name*).

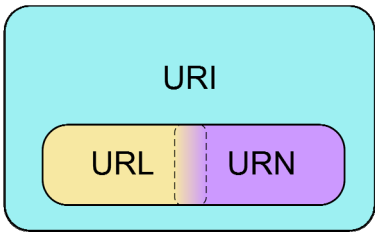


Figure 5. URI versus URL et UNR.

Ensuite, il faut définir les types de messages que le service web sera amené à envoyer ou recevoir. Chacun de ces types a un identifiant unique dans le WSDL. Une fois que cela est fait, la partie abstraite du service web doit être fournie. Elle comprend un ensemble d'interfaces, chaque interface ayant son propre identifiant unique dans le WSDL et étant constituée d'un ensemble d'opérations abstraites. Chacune de ces opérations a elle aussi un identifiant unique dans l'interface qui la déclare et représente une interaction entre le client et le service web. De plus, chaque opération spécifie l'identifiant des types de messages que le service peut envoyer ou recevoir. Ces identifiants correspondent aux types définis dans l'étape précédente. Chaque opération spécifie également un patron d'échange de messages qui indique la séquence suivant laquelle les messages associés à l'opération doivent être transmis entre le client et le service web.

La figure ci-dessous présente un exemple de définition abstraite d'un service web, dans lequel il est possible de retrouver des types des messages, ainsi qu'une interface nommée *reservationInterface* et l'opération, nommée *opCheckAvailability*, associée à cette interface.

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/ns/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
    xmlns:wsoap= "http://www.w3.org/ns/wsdl/soap"
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsdlx= "http://www.w3.org/ns/wsdl-extensions">

  <documentation>
    This document describes the GreatH Web service.  Additional
    application-level requirements for use of this service --
    beyond what WSDL 2.0 is able to describe -- are available
    at http://greath.example.com/2004/reservation-documentation.html
  </documentation>

  <types>
    <xs:schema
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://greath.example.com/2004/schemas/resSvc"
        xmlns="http://greath.example.com/2004/schemas/resSvc">

      <xs:element name="checkAvailability" type="tCheckAvailability"/>
      <xs:complexType name="tCheckAvailability">
        <xs:sequence>
          <xs:element  name="checkInDate" type="xs:date"/>
          <xs:element  name="checkOutDate" type="xs:date"/>
          <xs:element  name="roomType" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>

      <xs:element name="checkAvailabilityResponse" type="xs:double"/>

      <xs:element name="invalidDataError" type="xs:string"/>

    </xs:schema>
  </types>

  <interface  name = "reservationInterface" >

    <fault name = "invalidDataFault"
            element = "ghns:invalidDataError"/>

    <operation name="opCheckAvailability"
            pattern="http://www.w3.org/ns/wsdl/in-out"
            style="http://www.w3.org/ns/wsdl/style/iri"
            wsdlx:safe = "true">
        <input messageLabel="In"
              element="ghns:checkAvailability" />
        <output messageLabel="Out"
              element="ghns:checkAvailabilityResponse" />
        <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
    </operation>

  </interface>

</description>
```

Figure 6. Exemple de définition abstraite d'un service web de réservation de chambres d'hôtel.

Une fois la spécification des messages abstraits pouvant être échangés avec le service web finie (le *quoi*), il est alors nécessaire de spécifier la partie concrète du WSDL, c'est-à-dire la façon dont ces messages peuvent être échangés (le *comment*). C'est ici qu'intervient la spécification de l'ensemble des liaisons (*bindings*) du service web. Toute liaison a un identifiant unique dans le WSDL. Le but d'une liaison est de spécifier le formatage concret des messages, ainsi que les détails du protocole de transmission pour (toutes les opérations et erreurs d') une interface donnée.

La figure ci-dessous présente un exemple de liaison associée au service web dont la définition abstraite est donnée ci-dessus. La liaison présentée ci-dessous est définie pour l'interface *reservationInterface* et pour l'opération *opCheckAvailability*.

```
<?xml version="1.0" encoding="utf-8" ?>
<description ...>

<binding name="reservationSOAPBinding"
          interface="tns:reservationInterface"
          type="http://www.w3.org/ns/wsdl/soap"
          wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">

    <fault ref="tns:invalidDataFault" wsoap:code="soap:Sender"/>

    <operation ref="tns:opCheckAvailability"
      wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>

  </binding>

</description>
```

Figure 7. Exemple de liaison (*binding*) d'un service web.

Dès lors que le *comment* est spécifié, il convient de spécifier *où* le service peut être accédé. Un WSDL lie le service avec l'interface qu'il supporte et avec un ensemble de ports (*endPoints*) par lesquels le service peut être accédé. Chaque port doit, quant à lui, référencer une des liaisons définies précédemment afin de préciser le formatage et le protocole qui doivent être utilisés pour ce port. Chaque port est défini dans le cadre d'un service, il a donc un identifiant unique dans le cadre du service qui le définit. Il est important de remarquer qu'un service ne peut avoir qu'une et une seule interface. Par contre, aucune précision n'est donnée sur le fait qu'un WSDL ne contienne qu'un unique service ou plusieurs services, typiquement, des WSDL contenant plusieurs services peuvent être trouvés facilement sur le web (parmi les WSDL d'Amazon, par exemple).

La figure ci-dessous est un exemple de port (*endPoint*) référençant l'exemple de liaison présenté ci-dessus.

```
<?xml version="1.0" encoding="utf-8" ?>
<description ...>

<service name="reservationService" interface="tns:reservationInterface">

  <endpoint name="reservationEndpoint"
    binding="tns:reservationSOAPBinding"
    address="http://greath.example.com/2004/reservation"/>

  </service>

</description>
```

Figure 8. Exemple de port d'accès d'un service web.

Ensuite, nous avons vu que la définition abstraite d'un service précise non seulement le type des messages reçus et envoyés par le service web, mais précise aussi les patrons de communication que doivent suivre les envois et réceptions de messages. Ainsi, du point de vue du SLA, la définition abstraite d'un service est de niveau 1 (syntaxique) et de niveau 3 (enchainement des messages).

3.3 Universal Description Discovery and Integration

La spécification WSDL V2.0 n'appréhende pas la notion de registre et aucune précision n'est donnée quant au rôle de demandeur/client de service web. Ce qui est compréhensible puisque, comme nous l'avons écrit précédemment, la spécification WSDL V2.0 ne se focalise que sur la description du service web. Cela implique donc que le demandeur de service qui souhaite interagir avec un service web doit, soit connaitre le WSDL du service web, soit savoir où il peut le récupérer. Cependant, dans la pratique, les fournisseurs et demandeurs de services ont introduit un registre de service de manière à

pouvoir profiter des avantages du patron SOA. Le registre le plus habituellement utilisé est *Universal Description Discovery and Integration* (UDDI). C'est un annuaire de contrats de service basé sur XML plus particulièrement destiné aux services web.

La figure ci-dessous présente le mécanisme d'établissement d'une liaison entre un client/demandeur d'un service web et le service web en utilisant UDDI. Le préambule à ce mécanisme est l'enregistrement du service web dans l'annuaire UDDI.

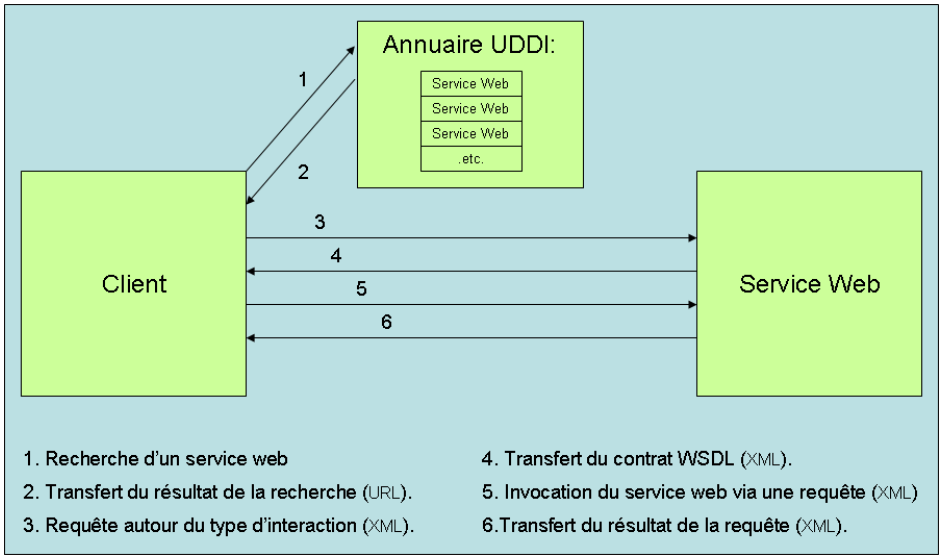


Figure 9. Établissement d'une liaison entre un client et un service web via UDDI.

Si le demandeur du service Web n'utilise pas de registre de service, alors, le mécanisme d'établissement de la liaison se limite soit aux étapes 3, 4, 5 et 6, soit aux étapes 5 et 6 si le client connait déjà le WSDL du service web.

3.4 Synthèse

La vision du W3C respecte les six propriétés mentionnées en section 2. Ainsi, comme nous l'avons vu :

- Un service web est contractuellement défini.
- L'utilisation de contrats assure les propriétés de faible couplage et de substituabilité.
- La liaison entre un client et un service web est créée de manière retardée.
- Le courtage et la négociation d'une liaison sont possibles mais laissés à la charge du client (le W3C ne fait aucune proposition).
- Enfin, un client et un service web peuvent appartenir à des domaines d'administration disjoints.

La spécification WSDL V2.0 et le W3C ne permettent pas de spécifier ou construire un service contenant/encapsulant au moins un autre service. Concrètement, le concept de service composite n'est pas supporté. De même, aucune mention ou proposition n'est faite autour du concept d'application à services et cela que cette dernière soit composite ou non.

Au niveau de la dépendance de la vision du W3C envers un langage de programmation, la spécification WSDL V2.0 définit son propre langage qui est destiné à servir de pont entre un client et un service web et cela que ces derniers soient basés sur le même langage de programmation ou non. De fait, la description d'un service faite via WSDL V2.0 est indépendante de tout langage.

Ensuite, afin de compléter notre propos, il faut noter que le consortium W3C n'est pas le seul à s'intéresser aux services web. Ainsi, bien que nous ne les détaillions pas dans l'état de l'art de cette thèse, les entreprises IBM et Microsoft proposent aussi leur vision des services web. IBM définit un service web comme une entité logicielle modulaire auto-contenue qui peut être décrite, publiée, trouvée et invoquée via un réseau, généralement le réseau Internet [IBM00]. De même, Microsoft définit un service web comme une logique applicative programmable, accessible en utilisant les protocoles Internet standards et offrant une interface ou un contrat bien défini qui décrit le service fourni [Mic08]. Au final, les définitions des services web du W3C, d'IBM et de Microsoft appréhendent toutes les trois les services de façon similaire.

Pour conclure, il est important de noter que les services web sont très populaires aujourd'hui. Ces derniers sont particulièrement utilisés pour l'intégration d'applications [IBM08]. L'utilisation des services web dans le cadre de la mise en ligne de services est, quant à elle, peu répandue.

Enfin, un tableau de synthèse concernant la vision de l'approche à service du W3C est présenté dans la section "Synthèse" de ce chapitre.

4. OSGI R4.1

Dans cette section, nous présentons la vision de l'Alliance OSGi™. Au cours de cette présentation, nous allons voir comment le concept de service, le SLA, le SOA et les six propriétés remarquables du SOC sont appréhendés par OSGi. Nous verrons aussi, si la vision OSGi est dépendante d'un langage de programmation et si la construction d'applications à services récursives (ou non) est possible.

4.1 Présentation

L'Alliance OSGi™ a été fondée en mars 1999. Elle propose des standards pour les services internet à destination des maisons, voitures, téléphones mobiles, ordinateurs de bureau, PMI-PME et d'autres environnements. Ses propositions sont concrétisées sous la forme de spécifications ouvertes pour la livraison de services sur des réseaux locaux, ces services peuvent être gérés et délivrés à distance via l'utilisation de tout type de réseau. Ces spécifications définissent différentes versions de la plateforme OSGi (par exemple, les versions R2.0, R3.0, R4.1).

La spécification de la plateforme OSGi propose une architecture commune et ouverte pour les fournisseurs de services, les développeurs, les éditeurs de logiciels, les opérateurs de passerelles (par exemple, les décodeurs satellites) et les fabricants de telles passerelles, afin que tous puissent développer, déployer et gérer des services de manière cohérente.

Dans cet état de l'art, nous nous focalisons sur la version 4.1, dite *Release 4.1* ou R4.1, de la spécification de la plateforme OSGi. Elle a été publiée en avril 2007 [OSGi07].

L'environnement d'exécution (c'est-à-dire le *framework*) est le noyau de la spécification de la plateforme OSGi. Cet environnement d'exécution est destiné à un usage général. Il a pour caractéristiques d'être basé sur la technologie Java, ainsi que d'être géré et sécurisé. Il supporte aussi le déploiement (local et à distance) de *bundles*.

Les *bundles* peuvent avoir deux types de dépendances, des dépendances au niveau package (au sens package Java) et des dépendances de services via la spécification d'un ou plusieurs services requis (par le biais d'une ou plusieurs interfaces Java). La plateforme OSGi offre des mécanismes qui permettent de connaitre l'état de résolution (satisfaction) de ces dépendances. De plus, la spécification OSGi présente les *bundles* comme des applications extensibles.

La plateforme OSGi offre, au développeur de *bundle*, les ressources et les points d'entrée suffisants pour tirer parti, d'une part, de l'indépendance qui est donnée vis-à-vis de la plateforme Java (JVM) et, d'autre part, de la capacité de chargement dynamique de code (c'est-à-dire de classes) de la plateforme OSGi. Ces deux propriétés permettent de développer des services pour des passerelles relevant du domaine de l'embarqué/réactif. De telles passerelles sont massivement répandues, par exemple, les décodeurs satellites dans les maisons, ou les ordinateurs de bords dans des voitures, typiquement, le 4x4 X5 de BMW embarque une plateforme OSGi.

Les fonctionnalités de la plateforme OSGi sont réparties selon plusieurs couches :

- la couche sécurité (*Security Layer*),
- la couche *bundle* (*Module Layer*),
- la couche concernant le cycle de vie des *bundles* (*Life Cycle Layer*)
- et enfin, la couche service (*Service Layer*).

Une vue globale de la plateforme à service OSGi sur une passerelle générique est présentée dans la figure ci-dessous.

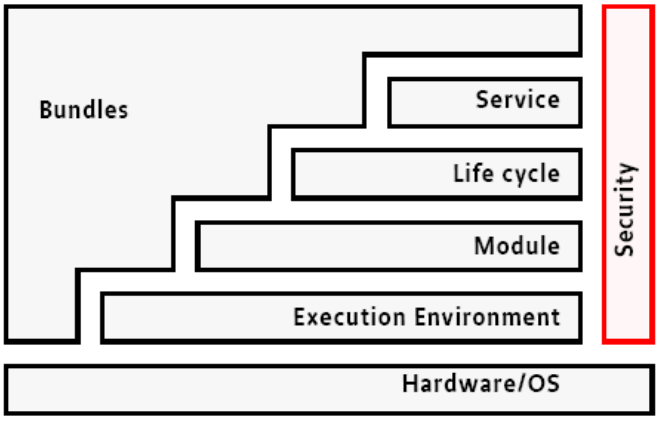


Figure 10. Les différentes couches de la plateforme OSGi [OSGi07].

4.2 La plateforme OSGi

La couche sécurité est une couche qui est transverse aux différentes couches de la plateforme OSGi et qui est optionnelle. Elle est basée sur l'architecture *Java 2 Security*. Elle offre une infrastructure pour déployer et gérer des *bundles* OSGi qui doivent s'exécuter dans des environnements sécurisés à grain-fin. Les possibilités concernant la sécurité vont jusqu'au niveau applicatif.

La couche *bundle* définit son propre modèle de modularisation au-dessus de la technologie Java. Cette couche possède des règles strictes concernant le partage de packages Java entre *bundles* et la mise à disposition de packages Java internes à un *bundle*.

La couche cycle de vie définit le cycle de vie des *bundles*. Elle offre aussi une API permettant de gérer effectivement le cycle de vie de chaque *bundle*. Cette API définit un modèle d'exécution pour les *bundles*. Ce modèle d'exécution définit le déploiement des *bundles*. Les activités de déploiement qui sont couvertes sont l'installation, l'activation, la désactivation, la mise à jour statique et la désinstallation. Ces activités correspondent respectivement aux commandes : *install*, *start*, *stop*, *update* et *uninstall*. Néanmoins, comme nous le verrons dans le chapitre suivant concernant l'état de l'art autour du déploiement, le modèle d'exécution OSGi ne couvre pas l'ensemble des activités du déploiement.

Enfin, la couche service d'OSGi fournit les mécanismes liés à l'approche à service. Elle définit un modèle de programmation dynamique pour les développeurs de *bundle*. Ce modèle simplifie le développement et le déploiement de *bundles* orientés service en découplant le contrat des services de leur implémentation. En OSGi, le contrat d'un service est un contrat de niveau 1 (concrètement, c'est une interface Java), l'implémentation d'un service est, quant à elle, enfouie dans le *bundle*. Ce modèle permet aux développeurs de *bundles* de définir des fournisseurs de services et des demandeurs de services. Les demandeurs définissent implicitement des dépendances sur des contrats de services. Lors de son activation, chaque *bundle* cherche à résoudre ces dépendances de contrats de services, cette résolution se fait dans le cadre fixé par le patron SOA. Une fois qu'une dépendance est résolue, le demandeur et le fournisseur du (contrat de) service entrent en interaction. Il est intéressant de noter qu'un demandeur de service qui s'est lié avec un fournisseur de service pourra tout à fait en changer tout au long de son exécution et cela selon sa propre politique, par exemple, lors de l'apparition d'un nouveau fournisseur de service ou encore si le fournisseur actuellement choisit vient à ne plus être disponible. Enfin, il faut noter que lorsqu'un *bundle* requérant un service vient à être activé et qu'il cherche un fournisseur pour ce service (afin de satisfaire son requis de service), il ne sait pas par avance avec quel fournisseur de service (c'est-à-dire *bundle*, ou plus précisément, implémentation de service) il va interagir. En effet, les dépendances de service d'un *bundle* sont définies dès son développement, mais le réel ensemble de *bundles* interagissant n'apparait que lors de l'exécution effective de chaque *bundle*, en fonction des *bundles* déployés dans la même plateforme à service et de la stratégie de sélection qu'il leur applique.

L'environnement d'exécution OSGi autorise un *bundle* à choisir les implémentations de services correspondant à ses requis de services et cela aussi bien au cours de son activation, qu'à l'exécution (c'est-à-dire une fois que le *bundle* est activé). Ce choix s'effectue en suivant le patron SOA, c'est-à-dire en utilisant le registre de service de l'environnement d'exécution OSGi. Ainsi, les *bundles* activés peuvent enregistrer leurs propres fournisseurs de services, recevoir des notifications à propos de tous les fournisseurs de services présents dans le registre de services. Ils peuvent aussi chercher des fournisseurs de services dans le registre de service, récupérer ceux qui leur conviennent et relâcher ceux qui ne leur conviennent plus. Cet aspect de l'environnement d'exécution OSGi permet aux *bundles* déployés de s'adapter au contexte dans lequel ils se trouvent. De plus, cette faculté d'adaptation ne nécessite pas la réactivation (c'est-à-dire la désactivation puis l'activation), ni de l'environnement d'exécution, ni du *bundle* embarquant les demandeurs de services. Par opposition, la mise à jour statique d'un *bundle* nécessite, quant à elle, la désactivation du *bundle*, sa désinstallation, l'installation et optionnellement l'activation du "nouveau" *bundle* (ces quatre activités peuvent être enchainées dans l'ordre présenté ici, mais aussi selon d'autres ordres).

Les interactions entre les différentes couches de la plateforme à service OSGi sont schématisées dans la figure ci-dessous.

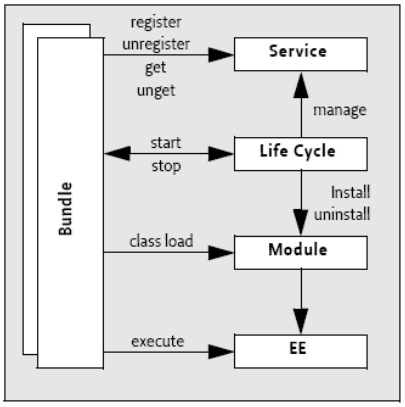


Figure 11. Interactions entre les différentes couches de la plateforme OSGi [OSGi07].

4.3 OSGi et service

Comme nous l'avons vu, OSGi introduit le concept de *bundle*. Un *bundle* est une entité logicielle qui contient des classes, mais aussi des ressources (par exemple, des fichiers images, des sons), ainsi qu'un ensemble de métadonnées sous la forme d'un fichier nommé manifest.mf. Un *bundle* peut être vu comme un composant. Un *bundle* peut requérir zéro, un ou plusieurs packages, comme il peut proposer zéro, un ou plusieurs packages. Pour cela, il doit spécifier les packages fournis et requis dans ses métadonnées. En effet, c'est la plateforme à service OSGi, elle-même, qui se charge de résoudre les dépendances de packages des *bundles*. La résolution ou non de ces dépendances de packages influe sur le fait qu'un *bundle* puisse devenir actif ou non, typiquement, un *bundle* possédant une dépendance de package non résolue ne peut être activé.

De même, un *bundle* peut être le fournisseur de zéro, un ou plusieurs services et le demandeur de zéro, un ou plusieurs services. Ainsi, un *bundle* peut être considéré comme un composant orienté service. Concernant le niveau service, les versions 1.0, 2.0 et 3.0 de la spécification OSGi impose la spécification, dans le manifeste du *bundle*, des demandeurs et fournisseurs de services qui sont embarqués dans le *bundle*. Néanmoins spécifier les fournisseurs de services (c'est-à-dire les services fournis) et les demandeurs de services (c'est-à-dire les services requis) d'un *bundle* dans son fichier manifest.mf n'est qu'indicatif ! En effet, il est possible de spécifier les services fournis A et B dans le manifest.mf d'un *bundle*, alors que dans les faits, le *bundle* n'aura qu'un service requis C. La version 4.1 de la spécification OSGi note les tags *Import-Service* et *Export-Service* comme étant obsolètes (*deprecated*). Ces tags représentaient respectivement les services requis et fournis. Cela implique qu'il est tout à fait possible de

déployer un *bundle* dont le déployeur connaitra les dépendances de packages (qui sont gérées par la plateforme OSGi) et ne connaitra pas les dépendances de services (dont la résolution est, qui plus est, laissées à la charge du *bundle* lui-même).
Il est aussi important de noter que le *bundle* est l'unité de déploiement en OSGi.

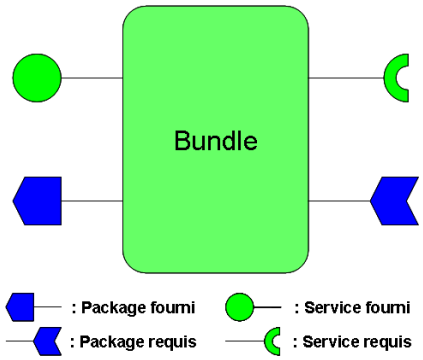


Figure 12. Schéma d'un *bundle* OSGi avec ses packages et services.

Dans la figure ci-après, nous présentons le méta-modèle des *bundles* OSGi du point de vue des entités logicielles et non du point de vue descriptif (c'est-à-dire du point de vue des contrats de services, par exemple).

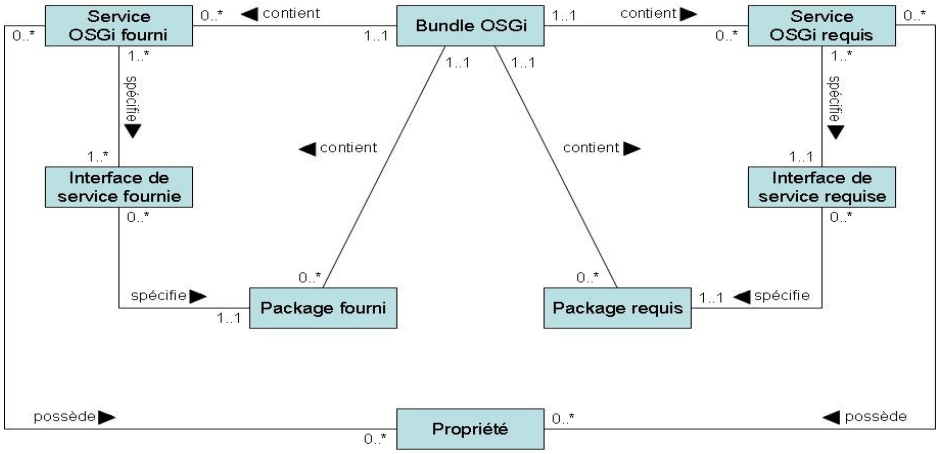


Figure 13. Méta-Modèle des *Bundles* OSGi (du point de vue entité logicielle).

Un *bundle* doit contenir un fichier manifest.mf. Ce fichier contient diverses métadonnées concernant le *bundle*. Ensuite, si le développeur du *bundle* s'en tient à la spécification OSGi Core R4.1, un *bundle* doit posséder une classe *Activator* (pour rappel, cette classe sert à l'activation et à la désactivation du *bundle*). Néanmoins, si le *bundle* utilise la spécification *Declarative Service* qui est définie dans le compendium associé à la spécification OSGi Core R4.1 (chapitre 112), alors, un *bundle* n'a pas besoin d'une classe *Activator* à proprement parlé. En effet, *Declarative Services* prend en charge les activités d'activation et de désactivation [OSGi07a]. Enfin, un *bundle* peut aussi contenir zéro, une ou plusieurs ressources internes (par exemple, des classes, des fichiers de configuration, des images, des sons).

Nous allons aussi apporter une précision quant à la cardinalité associée à la classe "Interface de service fournie" dans l'association entre les classes "Service OSGi fourni" et "Interface de service fournie". Elle est (1..*), car, en OSGi, un même service peut être enregistré dans le registre de service sous une ou plusieurs interfaces (comme cela est précisé dans la section "5.2.3: Registering Services" de [OSGi07]).

De plus, le registre de service OSGi peut contenir plusieurs "Services OSGi fournis" implémentant la même "Interface de service fournie". Ainsi, plusieurs *bundles* OSGi peuvent offrir la même interface par le biais de services qui leur sont propres.

La cardinalité (1..1) associée à la classe "Bundle OSGi" dans l'association entre "Bundle OSGi" et "Package fourni" (respectivement "Package requis") permet de spécifier que l'ensemble des classes réalisant le package fourni (respectivement requis) est contenu par un et un seul *bundle*. Cependant, des copies de ce même ensemble de classes peuvent tout à fait se trouver dans plusieurs *bundles*. Ainsi plusieurs *bundles* peuvent chacun fournir ou requérir un ou des packages de même identifiant (un package est identifié par son nom canonique JAVA, par exemple, "fr.imag.exemple").

Enfin, la spécification OSGi ne fait ni remarque, ni proposition autour de l'idée de service ou de composant orienté service qui soit hiérarchique, composite. Elle ne mentionne, ni ne précise rien non plus quand au concept d'application à services composée d'ensembles de fournisseurs et demandeurs de services interagissants, que ces services soient des entités logicielles en eux-mêmes, ou des composants orientés service.

En définitive, l'introduction et l'utilisation de *bundle* fait qu'OSGi appréhende l'approche à service non pas suivant la vision service du W3C, mais selon une vision composant orienté service.

5. SERVICE COMPONENT ARCHITECTURE V1.00

5.1 Présentation

La collaboration *Open Service-oriented Architecture* (OSOA) représente un groupe informel d'industriels composé, entre autres, de Bea Systems Inc., IBM Corporation, Oracle Corporation, Red Hat Inc., SAP AG, Siemens AG, Sun Microsystems [OSOA08]. Ce groupe cherche à définir un modèle de programmation indépendant de tout langage de programmation particulier et répondant aux attentes du développement de logiciels exploitant les caractéristiques et bénéfices de l'architecture orientée service (*service-oriented architecture*, SOA). La collaboration OSOA n'est pas un organisme de standardisation ; c'est un ensemble d'industriels qui souhaitent innover via le développement de ce modèle de programmation et livrer librement ses spécifications à la communauté informatique afin que des implémentations compatibles puissent être produites. Une fois à maturité, l'objectif de la collaboration OSOA est de donner ces spécifications à un organisme de standardisation, afin qu'elles deviennent des normes.

Le projet *Service Component Architecture* (SCA) est l'un des projets de la collaboration OSOA [SCA07]. Les spécifications de ce projet sont le sujet de la cinquième section de cet état de l'art. Il est à noter que nous nous basons sur la spécification SCA version 1.00 qui est la dernière version en date (publiée en mars 2007).

Le projet SCA cherche à fournir un modèle pour la création de composants orientés service indépendants de tout langage de programmation particulier et pour assembler ces composants orientés service dans une solution *business* - un tel modèle est nécessaire lors de la construction d'applications utilisant l'architecture orientée service. Pour ce faire, le projet SCA a défini un ensemble de spécifications qui décrivent un modèle pour la construction d'applications et de systèmes utilisant le patron SOA. Cet ensemble de spécifications est construit au-dessus de standards libres, comme les services web.

SCA encourage la création d'applications à base de composants implémentant la logique métier de l'application et qui, d'une part, offrent leurs capacités par le biais d'interfaces orientées service appelées *services* et, d'autre part, requièrent les fonctionnalités d'autres composants, toujours à travers des interfaces orientées service nommées *references*. SCA divise la construction d'une application orientée service en deux grandes étapes :

- La conception et le développement des implémentations de composants qui fournissent des services et requièrent d'autres services.
- La seconde étape est la création de l'assemblage de composants et la spécification des liens (*wires*) entre les références et les services des composants.

Le modèle de programmation SCA cherche à englober un vaste panel de technologies via ses composants orientés service et ses mécanismes de communication. Pour les composants, cela inclut non seulement des langages de programmation, mais aussi des environnements et modèles qui leurs sont communément associés, comme JAVA, C++, PHP, BPEL, XSLT, SQL, ainsi que les environnements et modèles Java EE, Spring. Au niveau des communications, les compositions SCA permettent l'utilisation de plusieurs technologies telles que les services web, les systèmes à messages et les systèmes synchrones ou asynchrones. A ce propos, il est important de bien faire la distinction entre un lien (*wire*) et une liaison (*binding*). En effet, un lien apparait entre un service et une référence, alors qu'une liaison décrit soit le moyen d'utiliser un service (à l'image du terme *binding* des services web), soit le mécanisme qui va être employé par une référence afin d'utiliser un service.

5.2 Spécification

La spécification SCA est divisée en deux parties. Chaque partie se focalise sur un aspect de SCA. La première partie concerne la spécification du modèle d'assemblage (*Assembly model*) qui décrit les composants, l'agrégation de composants et les liens entre composants. Il faut noter que ces liens

mettent en jeu la notion de composite. Comme nous l'avons déjà précisé précédemment, le modèle d'assemblage spécifié par cette première partie est indépendant de tout langage d'implémentation. La seconde partie concerne, d'une part, l'implémentation du modèle d'assemblage SCA au-dessus d'une technologie particulière et, d'autre part, l'implémentation d'un client d'un système ou d'une application SCA au-dessus de cette même technologie. Cette seconde partie existe pour chaque technologie qui supporte SCA. Par exemple, le document de spécification nommé *SCA C++ Client and Implementation V1.00* décrit l'implémentation du modèle SCA au-dessus du langage C++.

Comme nous l'avons vu précédemment, SCA est un modèle de programmation conçu pour la construction d'applications et de systèmes basés sur le SOA. Le modèle SCA part de l'idée qu'une fonction *business* est fournie par une série de services, ces services étant assemblés afin de créer des solutions répondant à un ou plusieurs besoins particuliers. Ainsi, les applications composites peuvent contenir à la fois de nouveaux services créés spécialement pour telle ou telle application, mais aussi des fonctionnalités provenant de la réutilisation d'applications et de systèmes existants. SCA fournit un modèle permettant, d'une part, la composition de services (préexistants ou non) et, d'autre part, la création de composants orientés service.

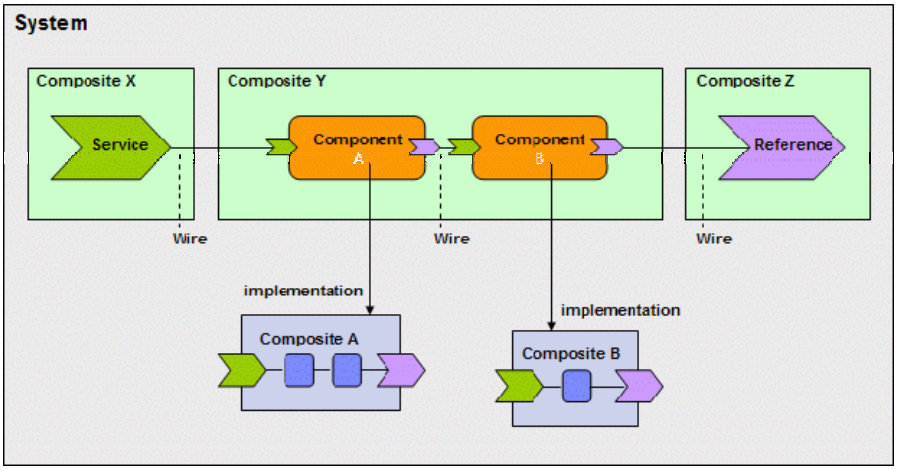


Figure 14. Exemple de système SCA composé de composites, composants et liaisons [SCA07].

Dans la spécification SCA, un composant est une entité logicielle qui contient zéro, un ou plusieurs services et/ou références. Un composant peut aussi avoir des propriétés. Enfin, un composant peut être, soit une implémentation en elle-même, soit un composite. La figure ci-dessous montre un exemple de composant SCA.

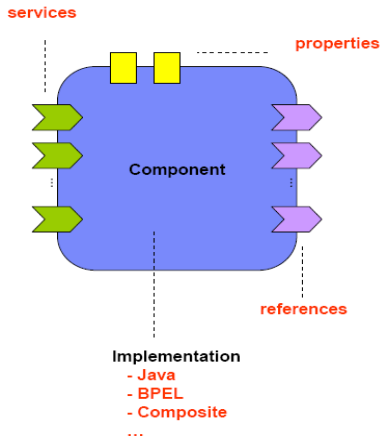


Figure 15. Composant SCA [SCA07].

Le modèle d'assemblage SCA définit la configuration d'une application ou d'un système SCA en termes de composants orientés service qui implémentent ou utilisent des services et des composites. Chaque composite décrit un assemblage de composants (assemblé selon des critères arbitraires), des liens entre les composants, des propriétés de configuration et comment les services et références sont exposés (en vue d'être utilisés).

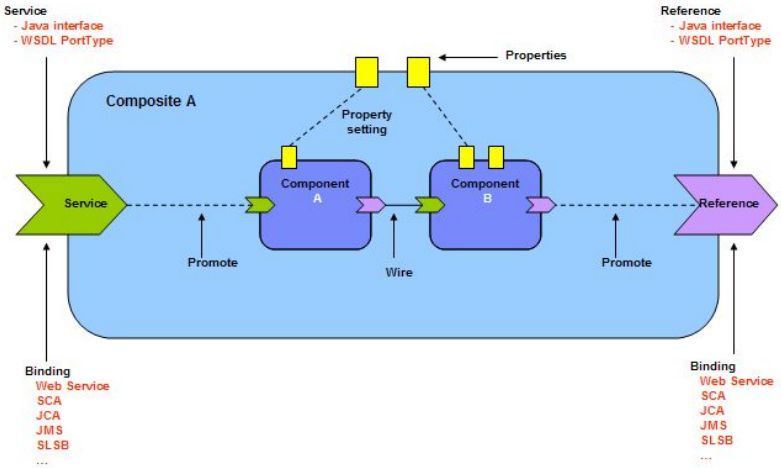


Figure 16. Composite SCA [SCA07].

Le modèle d'assemblage SCA introduit l'idée que les composites peuvent être utilisés comme implémentations de composants, d'une manière similaire aux classes JAVA, par exemple. De cette manière, le modèle d'assemblage SCA permet de créer une hiérarchie de composites, sans contrainte de profondeur. Ce type de structuration rend le modèle SCA hiérarchique/récursif. Les composites fournissent aussi les mécanismes de bases pour définir quels artefacts doivent être déployés ensemble (déploiement), quels artefacts devront s'exécuter dans le même processus (localité) et enfin quels artefacts pourront être vus par quels autres artefacts (visibilité).

Le modèle d'assemblage SCA fournit aussi plusieurs fonctionnalités pour les composites. La première de ces fonctionnalités est appelée *promotion*. Elle permet d'exposer des services et des références appartenant à des composants inclus dans un composite au niveau de ce même composite. Tout service (respectivement référence) exposé par un composite provient obligatoirement d'un service (respectivement d'une référence) d'un des composants qu'il inclut. Plusieurs références de composants peuvent être promues comme une seule référence au niveau du composite, mais pour cela, elles doivent être compatibles entre elles (la spécification utilise le terme compatible, sans pour autant le définir). De plus, dès que la référence au niveau du composite est liée à un service, alors, toutes les références de composants promues pour cette référence au niveau composite sont liées à ce même service. Les services et les références de niveau composant peuvent être liés entre eux à l'intérieur du composite, mais aussi être promus au niveau composite et liés, dans le même temps, avec des composants et/ou composites extérieurs au composite dont ils proviennent. La seconde de ces fonctionnalités est appelé Auto-Lien (*AutoWire*). C'est une fonctionnalité optionnelle qui permet de simplifier l'assemblage dans un composite. En effet, l'Auto-Lien permet de lier automatiquement des références à des services qui les satisfont et cela sans que le développeur ait besoin de spécifier explicitement le ou les liens. Quand la fonctionnalité Auto-Lien est utilisée, une référence qui n'est pas promue et qui n'est pas explicitement liée à un service dans un composite est alors automatiquement liée à un service appartenant à ce même composite. L'Auto-Lien fonctionne en cherchant, dans le composite où se trouve la référence, une interface de service qui correspond à l'interface de la référence qui a spécifié son attribut *AutoWire*. La fonctionnalité d'Auto-Lien n'est pas active par défaut, elle doit être spécifiée (il est nécessaire d'associer la valeur *true* à l'attribut *AutoWire* du composite). La troisième et dernière fonctionnalité n'a pas de nom particulier. Elle permet, dans les composites, de spécifier des liens qui seront créés, à l'exécution, par les implémentations des composants.

Nous avons vu précédemment que le modèle d'assemblage SCA est un modèle récursif. Récursif signifie que les composants et composites SCA peuvent être encapsulés par d'autres composites SCA qui eux-mêmes peuvent être encapsulés par d'autres composites SCA, etc. (c'est le patron composite).

Enfin, il est à noter que plusieurs extensions de SCA existent. L'une d'elles est SRML (*SENSORIA Reference Modelling Language*) [Sen05] [BHL+06]. SRML fournit des primitives pour modéliser les processus *business* (c'est-à-dire métier) en s'abstrayant de toute technologie. SRML est basé sur la notion de "module" qui sont vus comme des compositions de composants fortement couplés et de services faiblement couplés et découverts dynamiquement.

5.3 Synthèse

Le modèle SCA possède trois avantages. Il permet :

- une approche *Top-Down* lors du développement de l'application ou du système SCA,
- d'encapsuler les artefacts SCA, il devient donc possible de jouer sur la visibilité de tel ou tel composant ou composite
- et, enfin, puisqu'un composite peut être vu comme un composant, cela uniformise les attributs et les traitements applicables aux composants et composites (sans pour autant rendre ces traitements identiques).

Dans le cadre de cet état de l'art, nous avons modélisé la proposition de SCA. La figure ci-dessous présente le méta-modèle correspondant.

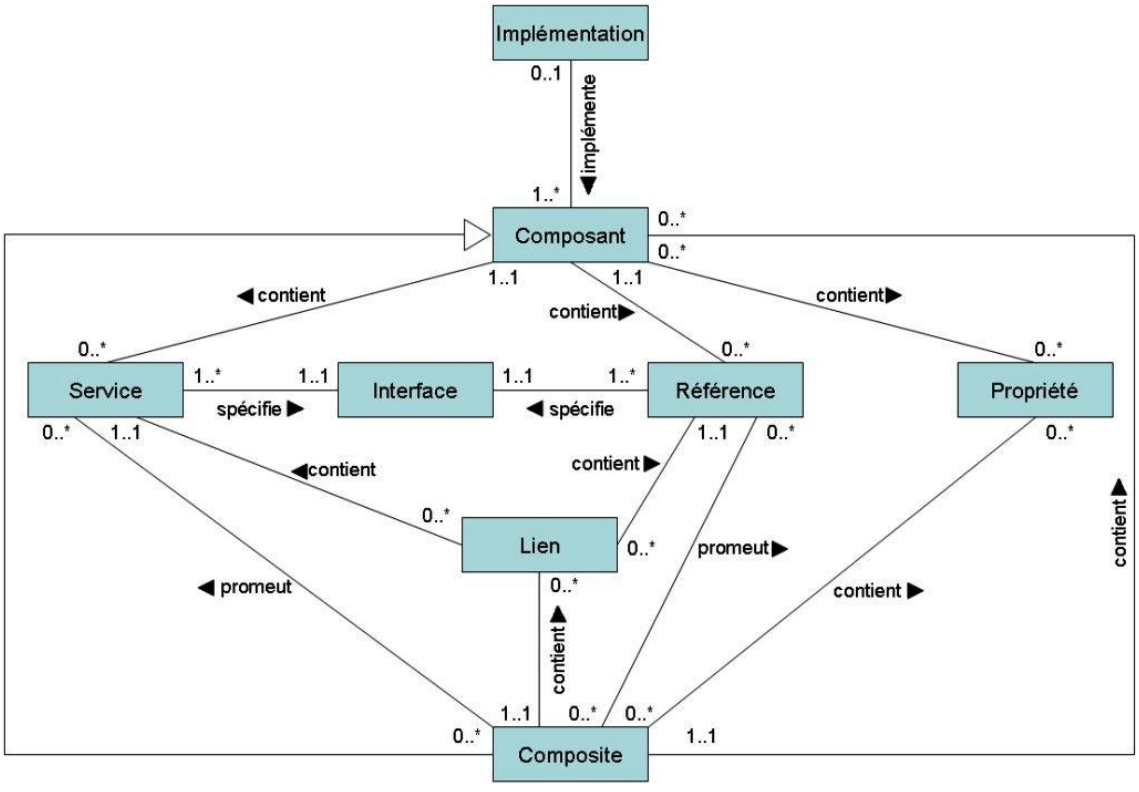


Figure 17. Méta-modèle SCA (niveau entité logicielle et non description).

Afin de présenter entièrement la vision SCA de l'approche à service, il est nécessaire de faire un certain nombre de remarques complétant ce méta-modèle.

Premièrement, un service (respectivement une référence) est toujours contenu par un et un seul composant, cela étant, ce service (respectivement cette référence) peut être promu au niveau du composite qui contient ce composant. Cela explique la cardinalité (1..1) qui est associée à la classe composant pour l'association entre les classes composant et service (respectivement référence) et cela explique aussi la cardinalité (0..*) associée à la classe composite dans l'association entre les classes composite et service (respectivement référence).

Deuxièmement, un service (respectivement une référence) est certes contenu par un et un seul composant, cependant, plusieurs services (respectivement références) identiques en tous points peuvent être contenus par des composants différents.

Ensuite, un composant peut être implémenté soit par une implémentation, soit par un composite qui jouera alors le rôle d'implémentation du composant.

La spécification SCA ne mentionne jamais l'architecture orientée service, bien qu'utilisant le concept de service et étant basée sur des composants orientés service. Le concept même de registre de service n'est pas mentionné. Par contre, le document concernant l'implémentation de SCA au-dessus de JAVA (ou de C++) précise que les services d'un composant peuvent être accédés en utilisant le contexte du composant (*component context*) [SCA07a]. Cela laisse entendre que tous les composants et composites possèdent leur propre registre de services. En conséquence, le client doit connaitre au préalable le composant ou le composite duquel il pourra prendre des services.

Enfin, il existe aujourd'hui plusieurs implémentations de SCA, par exemple, *WebSphere* d'IBM, *Aqualogic* de BEA et *Tuscany* de l'*Apache Software Foundation* [IBM05] [BEA08] [TUS08].Or, toutes ces implémentations sont incompatibles entre elles.

6. AUTRES TECHNOLOGIES

6.1 JINI

La technologie Jini™ est spécifiée par Sun Microsystems [JINI05]. Elle promeut une architecture orientée service qui définit un modèle de programmation qui exploite et étend la technologie Java™ [JINI07] [AOS+99]. Son but est de permettre la construction de systèmes orientés service distribués et sécurisés. Jini n'est pas dépendant d'un système d'exploitation. En effet, les périphériques et les logiciels sont considérés comme des services indépendants et communicants. Cependant, Jini nécessite une machine virtuelle Java (JVM). Les systèmes construits au-dessus de Jini consistent en des fédérations de services et de clients au-dessus de réseaux. De telles fédérations peuvent s'installer automatiquement et fonctionner. Jini peut être utilisé pour construire des systèmes distribués adaptatifs qui peuvent passer à l'échelle, qui sont évolutifs et flexibles, comme cela est requis dans les environnements informatiques dynamiques actuels. Jini peut traiter les problématiques liées à distribution, à l'évolution des systèmes, à la sécurité et à l'assemblage de services et supporte aussi les transactions.

Jini définit un service comme une entité logicielle qui peut être utilisée par une personne, un programme, ou un autre service. Un service peut fournir, par exemple, du calcul, du stockage, un canal de communication avec un autre utilisateur, un filtre logiciel, un point d'entrée vers un équipement. Un service est décrit par une interface Java et des propriétés.

Jini possède plusieurs capacités intéressantes, telles la publication de services, la découverte de services, la création d'une unique ou de plusieurs instances de services, la mobilité de code, c'est-à-dire le téléchargement de code à travers le réseau, (qui lui permet d'être, par exemple, indépendant envers les protocoles de communication). Jini fournit aussi un ensemble de notifications concernant les services permettant aux clients d'être informés des changements au niveau des services. Ensuite, Jini supporte de façon explicite les politiques de libération des instances de service via l'utilisation de bail. Pour interagir, les clients et les fournisseurs de services doivent commencer, tout d'abord, par trouver un registre. A ce propos, un registre peut être contacté en utilisant une adresse fixe ou bien il peut être découvert, via la diffusion d'une demande. Dans Jini, les services peuvent être organisés par groupes (par exemple un groupe de services d'impression). Un registre peut héberger un groupe de services particuliers. Cependant, bien que Jini supporte l'existence de registres multiples, il n'offre pas de mécanisme explicite permettant aux registres de déléguer une demande de service. En conséquence, les fournisseurs de services enregistrent, généralement, leurs services dans tous les registres qu'ils ont à disposition. Jini supporte l'existence d'un nombre indéfini de registres de services.

Pour conclure, Jini est une technologie dont la vision de l'approche à service est en ligne avec celle des services web. Les contrats de services sont de niveau 1. Jini respecte l'architecture orienté service. Enfin, les services Jini respectent toutes les propriétés remarquables de l'approche à service que nous avons identifiées, à savoir, le couplage faible, la liaison retardée, la substituabilité, les accords de niveau de service, le courtage et la négociation et la possibilité de domaines d'administration disjoints. Enfin, la responsabilité de Jini a été transférée, fin 2006, de Sun Microsystems à Apache (*Apache Software Foundation*) via la création du projet River [RIV08].

6.2 UPnP

Le Forum UPnP™ est un consortium industriel qui a été formé en 1999 [UPNP99] [UPNP00] [JW03]. Le terme UPnP signifie *Universal Plug and Play*, il est dérivé du terme *Plug and Play* qui est une technologie pour attacher dynamiquement des périphériques à un ordinateur. Le but du Forum UPnP est la définition de standards simplifiant la mise en réseaux de périphériques intelligents dans les maisons et dans les entreprises (*Small Office Home Office*, SOHO) et tout en les rendant robustes. Le Forum UPnP est constitué de plus de 800 industriels, incluant des leaders dans les domaines de l'informatique, de l'impression, des réseaux, ainsi que dans les domaines de l'électronique, de l'électroménager, de

l'automatisation, du contrôle et de la sécurité et des produits mobiles. En spécifiant et publiant des descriptions de périphériques UPnP et de services UPnP, le Forum UPnP cherche à promouvoir l'émergence de périphériques facilement connectables, à simplifier l'implémentation de logiciels pour les équipements réseaux, à rendre leur connexion transparente et enfin à conduire le développement d'un écosystème pour le support des périphériques UPnP.

L'architecture d'UPnP est constituée d'une plateforme distribuée de services dynamiques pour des périphériques réseau (comme, par exemple, les télévisions, les éclairages, les routeurs ADSL, les lecteurs de DVD, les volets déroulants) et de points de contrôle (comme, par exemple, des PDAs, des télévisions, des interrupteur muraux) connectés entre eux par un réseau ad-hoc. UPnP définit les protocoles réseau permettant la détection (et le retrait) dynamique des périphériques, l'utilisation des services qu'ils fournissent par des points de contrôle et la notification des changements de valeurs des variables d'état associées aux services. La première version des protocoles de l'UPnP s'appuyait sur XML et HTTP au dessus de TCP, UDP et UDP Multicast qui sont des standards d'Internet. La seconde version, appelée *Device Profile for Web Services* (DPWS) réoriente les protocoles vers les standards des services web, comme SOAP, WSDL, XML Schema, WS-Addressing, WS-Discovery, WS-Eventing.

Le modèle de services d'UPnP est relativement simple. Un *device* UPnP expose des services. Chaque service est constitué d'une liste de variables d'état qui peuvent être observée par plusieurs points de contrôle, et par une liste d'actions relatives à chaque variable d'état. L'invocation d'une action par un point de contrôle peut provoquer un changement dans l'état des variables d'état. En cas de changement, les points de contrôle qui observent ces variables sont notifiés du changement. Un *device* appelé racine (*root*) peut contenir d'autres *devices*.

Les *devices* et, respectivement, les services sont chacun décrits au moyen de document XML. Il existe un schéma XML pour les *devices* et un autre pour les services.

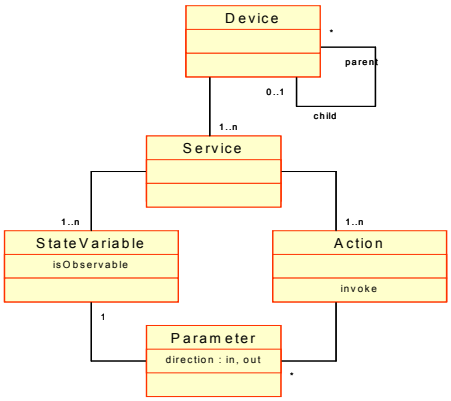


Figure 18. Modèle de données d'UPnP [UPNP99]

Une des activités importantes de l'UPnP Forum est la standardisation de périphériques [UPNP07]. Pour ce faire, les membres du Forum UPnP proposent des descriptions de périphériques standards (*Standardized Device Control Protocol*, SDCP). Les SDCP décrivent des *devices*, leurs services obligatoires et optionnels, leurs variables d'état obligatoires et optionnelles et les actions obligatoires et optionnelles. Les fabricants d'équipements UPnP sont libres d'ajouter des services, des variables d'état et des actions propriétaires aux descripteurs de leurs équipements. La liste des SDCP comporte, actuellement, 16 devices et une quarantaine de services standardisés pour les domaines d'application suivants, à savoir, l'audio et la vidéo (via les standards *MediaServer V2.0* et *MediaRenderer V2.0*), la maison, le réseau (via le standard *Internet Gateway V1.0*), les imprimantes, le contrôle (à distance) et les scanners.

UPnP ne suit pas réellement l'architecture orienté service. En effet, il n'existe pas de registre de services en tant que tel. Cependant, les points de contrôles situés sur le même réseau qu'un périphérique qui vient de se connecter sont informés, par le périphérique, des services qu'il propose. Parallèlement, lorsqu'un point de contrôle vient à être connecté à un réseau, il peut découvrir les périphériques et leurs

services associés. Ainsi, dans UPnP, comme dans le patron SOA, un fournisseur de service se publie et un demandeur de service peut rechercher et établir une interaction avec un fournisseur.

UPnP n'est pas lié à un langage de programmation particulier. En effet, plusieurs API (comme, par exemple, C, C#, C++, Java) existent sur de nombreuses plates-formes (comme, par exemple, Windows XP, Windows CE, Linux) et sont proposés par les principaux promoteurs d'UPnP, à savoir, Microsoft, Siemens (http://www.plug-n-play-technologies.com), Intel (http://www.intel.com/technology/UPnP/). Au niveau du langage Java, l'Alliance OSGi spécifie la manière de développer des périphériques UPnP et des points de contrôle UPnP au dessus d'OSGi (qui est une technologie basée sur le langage Java) [OSGi07b]. Le périphérique UPnP est ensuite conditionné dans un *bundle* qui est déployé au moyen du *Device Access* [OSGi07c].

Pour conclure, UPnP est une technologie dont la vision de l'approche à service est en ligne avec celle des services web. Les contrats de services sont de niveau 1. UPnP ne respecte pas l'architecture orientée service en tant que telle, mais un fournisseur de service se publie et un demandeur de service peut rechercher et établir une interaction avec un fournisseur. UPnP pallie donc l'absence de registre de services par des protocoles de publication et de découverte. Il est important de noter qu'une telle approche n'est viable que dans les réseaux locaux et qu'elle ne passe pas à l'échelle. Enfin, les services UPnP respectent toutes les propriétés remarquables de l'approche à service que nous avons identifiées, à savoir, le couplage faible, la liaison retardée, la substituabilité, les accords de niveau de service, le courtage et la négociation et la possibilité de domaines d'administration disjoints.

7. SYNTHÈSE AUTOUR DES APPROCHES À SERVICE

Cette section est structurée en trois sous-sections. La première sous-section propose un tableau qui fait la synthèse des visions de l'approche à service proposées par les spécifications WSDL V2.00 W3C, OSGi R4.1 et SCA V1.00. Les deux autres sous-sections se focalisent sur les apports et les limites de ces visions du point de vue des applications à services.

7.1 Tableau récapitulatif des visions du W3C, d'OSGi et de SCA

Ce tableau (ci-dessous) porte sur la définition du concept de service retenu par chacune des trois spécifications, ainsi que sur le contrat de service (SLA) et sur la mise en œuvre et l'utilisation du patron SOA, enfin, il porte aussi sur la présence des six propriétés "service", l'existence de la notion d'application à services, récursive ou non et sur la dépendance de la spécification envers un langage de programmation.

	W3C WSDL V2.0	OSGi R4.1	SCA V1.00
Concept de service.	Service, il possède une unique interface. Un service ne spécifie pas de requis ou de dépendances	Composant orienté service. Un service peut être publié sous plusieurs interfaces. Un composant peut avoir des requis de services.	Composant orienté service. Un composant peut avoir des requis de services.
Contrat de service.	Niveaux 1 et 3, les enchainements de messages sont spécifiés.	Niveau 1, interface JAVA.	A priori niveau 1, mais le niveau peut varier selon la technologie réalisant SCA.
Patron SOA, existence et mise en œuvre.	La notion de registre n'apparaît pas dans la spécification et aucune précision n'est donnée quant au demandeur/client. Donc, soit le demandeur sait où trouver le contrat, soit il le connaît. Cependant, dans la pratique, UDDI est utilisé comme registre de WSDL.	Chaque environnement d'exécution OSGi possède son propre registre de service et tout bundle OSGi peut être simultanément fournisseur ou demandeur de zéro, un ou plusieurs services.	La notion de registre n'apparaît pas dans la spécification. En JAVA et en C++, pas de référence concernant un quelconque registre. Par contre, les services d'un composant peuvent être accédés en utilisant le contexte du composant. Aucun exemple de client n'est donné
Couplage faible.	Oui, utilisation de contrats.	Oui, utilisation de contrats.	Oui, utilisation de contrats.
Liaison retardée.	Oui.	Oui.	Oui.
Substituabilité.	Oui.	Oui pour un service. Non pour un composant orienté service.	Oui pour un service. Non pour un composant orienté service.
Courtage, Négociation.	Possible, mais laissé à la charge du demandeur.	Possible, mais laissé à la charge du demandeur.	Possible, mais laissé à la charge du demandeur.
Domaines d'administration disjoints.	Oui.	Oui.	Probablement, cependant aucune précision n'est donnée et le fonctionnement des demandeurs de services n'est quasiment pas défini.
Notion d'application à services.	Non.	Oui et non, un bundle OSGi est une application extensible.	Oui, via les composants/composites.
Récursivité.	Non.	Non.	Oui.
Dépendance vers un langage.	Non.	Oui, JAVA.	Oui, (par exemple, JAVA, C++).

Tableau 1. Synthèse des différentes visions de l'approche à service.

7.2 La vision du W3C et des services web en général

Du point de vue du déploiement d'applications à services, la principale critique concernant la vision de l'approche à service définie par la spécification WSDL V2.00 du W3C porte sur le fait que les services ne sont vus que comme un couple comprenant l'interface du service et un ensemble de protocoles à suivre pour utiliser cette interface. Cette vision et son couplage avec UDDI permet, certes, aux demandeurs de services d'interagir avec des fournisseurs de services. Cependant, elle ne propose aucun mécanisme pour définir une entité logicielle fournissant et demandant des services. En effet, un service web ne peut pas spécifier de requis de services (donc il ne peut être demandeur de service), un service web est uniquement un fournisseur de service. Cette vision ne propose pas non plus de mécanisme ni pour composer entre elles des entités logicielles fournissant des services, ni pour spécifier l'architecture d'un tel ensemble. En définitive, la notion d'application à service n'est ni abordée, ni même mentionnée.

Enfin, cette vision de l'approche à service et plus particulièrement l'utilisation du WSDL prend tout son sens si l'objectif poursuivi est la mise à disposition des fonctionnalités offertes par une entité logicielle. Par exemple, la mise à disposition d'une base de connaissances, des cours de la bourse, des données d'un équipement peut tout à fait être faite via WSDL. De même, WSDL prend aussi sens dans le cas où un logiciel est formé d'une entité logicielle centrale et d'un ensemble d'entités logicielles utilisées par l'entité centrale ; dans ce cas, il est tout à fait pertinent d'utiliser WSDL pour définir les services fournis par chaque entité. Typiquement, cela peut être le cas lors de l'utilisation du patron *Enterprise Application Integration* (EAI), où l'EAI est l'entité centrale et les applications intégrées par l'EAI sont les entités utilisées.

7.3 Les visions composants orientés service d'OSGi et de SCA

Contrairement à la vision de l'approche à service du W3C et des services web en général discutée ci-dessus, les spécifications OSGi R4.1 et SCA V1.00 appréhendent les services en utilisant des composants orientés service. Cette seconde vision diffère de la première principalement parce qu'elle inclut un mécanisme qui permet à un composant orienté service de posséder des fournisseurs de services et des demandeurs de services. Ainsi, les spécifications OSGi et SCA permettent à un composant orienté service de proposer des fournisseurs de services, mais aussi de spécifier des requis (c'est-à-dire, des dépendances) vers des fournisseurs de services à travers la spécification de demandeurs de services.

Nous venons de voir que la vision à composant orienté service introduit un mécanisme de composition qui permet à un composant de posséder des fournisseurs de services et des demandeurs de services. Utiliser cette vision a l'avantage de permettre la réutilisation des concepts et logiciels définis autour des modèles à composants. En effet, ces derniers définissent des interfaces fournies et requises, qui, dans les spécifications d'OSGi et de SCA, sont vues comme des fournisseurs de services et respectivement des demandeurs de services. De même, l'introduction du patron SOA et son utilisation obligatoire pour établir des interactions entre les fournisseurs et les demandeurs de services renforce encore la translation d'interface à service. La spécification OSGi est cependant plus pertinente que la spécification SCA, puisqu'elle explicite l'utilisation du patron SOA. Au final, dans la vision composant orienté service, les interfaces des composants sont vues comme des contrats de services offerts par les fournisseurs ou requis par les demandeurs appartenant aux composants. Plus précisément, ces contrats sont de niveau "syntaxique" pour OSGi ; il en est de même pour les contrats SCA, cependant les contrats SCA peuvent aussi être de niveau "synchronisation", en effet, la spécification SCA impose que les contrats de services des composants orienté service SCA soit exprimables sous la forme de WSDL (WSDL permet la définition de contrat relevant du niveau syntaxique).

Cette seconde vision est tout à fait pertinente dans le cadre d'une entité logicielle possédant un ou plusieurs fournisseurs de services et un ou plusieurs demandeurs de services. Concrètement, dans le cadre d'OSGi, cette entité logicielle est un *bundle*, ce dernier est présenté comme une application extensible (par la spécification OSGi). Cependant, si l'objectif suivi est la mise en place d'un ou plusieurs services mettant en jeu, voire composant, un ensemble de services et si, qui plus est, cet ensemble de services doit respecter une architecture (par exemple, de simples interactions prédéfinies), alors, la vision

composant orienté service n'est pas satisfaisante. En effet, bien que les composants puissent fournir et demander des services, un service n'en reste pas moins décrit par un contrat qui ne spécifie que le service fourni ou requis et qui ne spécifie pas le composant auquel il appartient. De plus, dans le cas d'un fournisseur de service, son contrat ne spécifie pas non plus les demandeurs de services (appartenant au composant) qu'il utilise pour rendre son service. Or, dans le cadre d'ensembles de services et plus précisément d'applications à services, le contrat d'un service doit préciser, certes, ce que le service fournit, mais aussi ce que le service requiert. En effet, l'architecture de chaque service de l'application et l'architecture globale de l'application elle-même sont essentielles. Pour être tout à fait clair, deux applications à services offrant les mêmes fournisseurs de services et nécessitant les mêmes requis de services ne sont équivalentes que si leurs architectures sont équivalentes. Formulé autrement, deux applications à services sont équivalentes si et seulement si :

- chaque application possède des fournisseurs de services et des demandeurs de services équivalents (c'est-à-dire de contrats de services équivalents),
- chaque application contient (composants orientés) services équivalents, c'est-à-dire de contrats de services équivalents
- et si les interactions entre les (composants orientés) services composant chaque application sont équivalentes.

Qui plus est, en OSGi, un demandeur de service appartenant à un composant spécifie le contrat du fournisseur de service dont il a besoin, mais ne donne aucune précision quant au composant qui lui offrira le fournisseur de service.

En plus de cela, l'approche à service stipule que des services de contrats équivalents doivent être substituables les uns aux autres. Or, dans le cadre de la vision composant orienté service, deux services fournis chacun par un composant ne sont pas substituables dans le cas général et cela, même si leurs contrats de service respectifs sont équivalents. Par exemple, un service S1 fourni par un composant C1 et qui n'utilise pas les requis de service de son composant n'est pas substituable par un service S2 (de contrat équivalent) fourni par un composant C2 et qui, lui, utilise plusieurs requis de services de son composant (C2). En conséquence, aucune garantie ne peut être apportée quant au fait qu'un fournisseur de service de contrat X fourni par un composant orienté service puisse bien être substitué à un autre fournisseur de services de contrat X fourni par un autre composant orienté service. Cela est illustré par un exemple dans la figure ci-dessous. Dans cet exemple, le service fourni X réalisé par l'algorithme 1 n'est pas substituable par le service fourni X réalisé par l'algorithme 3 ou 5 ou 7 et 8 ou 9 et 10. En effet, dans le premier cas, le service fourni n'utilise aucun demandeur de service, dans les autres cas, un demandeur de service est mis en jeu et l'état de l'algorithme réalisant X peut être perturbé par les algorithmes réalisant le service Y.

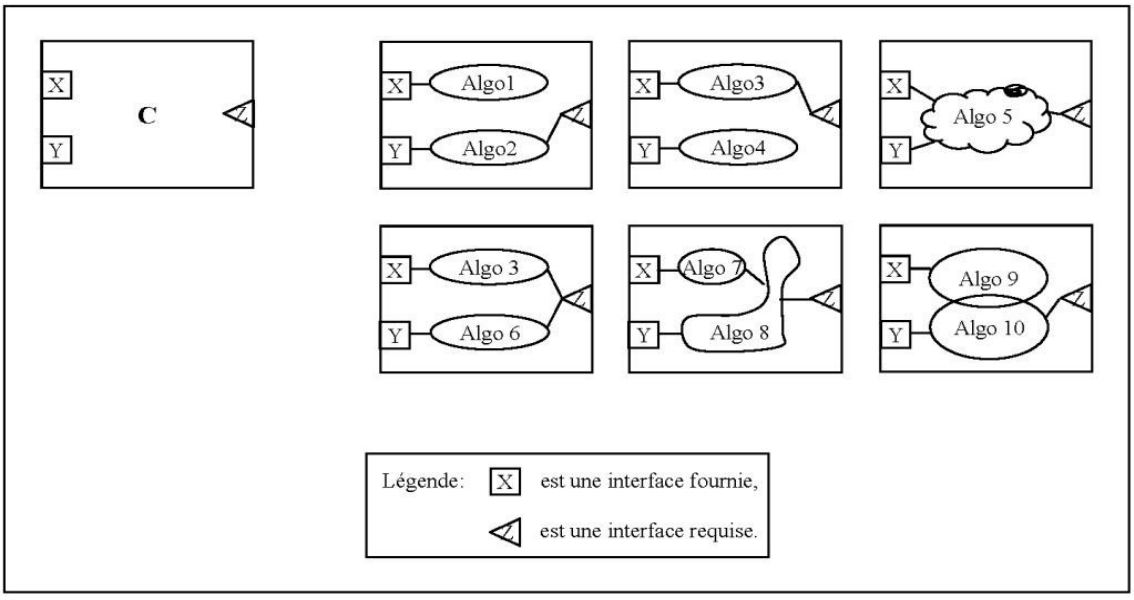


Figure 19. Non-substituabilité des fournisseurs de services de composants orientés service.

De manière générale, l'utilisation de la vision à composant orienté service nécessite de prendre conscience que les services fournis par les composants orientés service ne sont pas interchangeables dans le cas général. Pour être interchangeables, chaque fournisseur de service devrait préciser quelles sont les interactions (internes au composant) qu'il entretient avec les demandeurs de services du composant.

De façon synthétique, deux cadres sont à distinguer. Le premier cadre est celui des services. Dans ce cadre-ci, tout service peut être substitué par un service de contrat équivalent. Par exemple, si un client souhaite utiliser un service d'horloge qui donne l'heure universelle, alors, peu importe l'identité du fournisseur du service tant que l'heure obtenue est bien l'heure universelle (c'est-à-dire tant que le contrat est honoré). A l'inverse, dans le second cadre qui est celui des applications à services ; l'architecture de l'application, l'identité des entités formant l'application et leur architecture sont essentielles. Dans ce second cadre, deux services (fourni par le même composant ou des composants différents) ne sont, en général, pas substituables l'un à l'autre.

En ce qui concerne la définition ou l'utilisation de la notion d'application à services, la spécification OSGi n'est pas plus avancée que la spécification WSDL et les services web. Ainsi, les cas applicatifs du WSDL et des services web correspondent aussi bien aux composants orientés service OSGi. Bien qu'OSGi présentent ses *bundles* comme étant des applications extensibles, le patron architectural correspondant à ces applications extensibles reste une entité logicielle centralisatrice qui peut être étendue par d'autres entités logicielles lui offrant des services ; c'est typiquement le cas de l'IDE Eclipse [Ecl08] et de sa construction en plugins ou des serveurs d'applications JEE tels Jonas de Bull et l'Application Server 10g d'Oracle [Jon08] [Ora07]. Or, imposer un patron pour la construction d'application ne permet pas de décrire l'ensemble des applications à services possibles. Ce manque concernant une notion globale, générique, d'application à services dans OSGi a été relevée par Richard S. Hall dans une présentation autour de la version 2.0 de l'*OSGi Bundle Repository* (OBR2). Cette présentation a eut lieu lors de l'*OSGi Alliance Community Event* les 26 et 27 juin 2007 [Hal07].

La spécification SCA est, quant à elle, plus avancée que la spécification OSGi concernant la notion d'application à services. En effet, bien que se basant sur la vision composant orienté service, SCA inclut le patron composite et offre ainsi, un système récursif, hiérarchique, de composants et composites orientés service permettant la construction d'applications à services génériques. Ainsi, il est possible de spécifier les interactions de services entre des services fournis par des composants/composites. Néanmoins, comme pour OSGi, les fournisseurs et les demandeurs de services d'un même composant SCA n'explicitent pas les interactions qu'ils entretiennent au sein de ce même composant. En conséquence, la substitution d'un service d'un composant SCA par un autre service d'un autre composant SCA ne peut pas non plus être garantie.

En définitive, la mise en place et la spécification d'ensembles de services (respectant ou non une architecture) et d'applications à services (généralistes, c'est-à-dire ne suivant pas de patrons prédéfinis) n'est pas possible au-dessus de la vision à composant orienté service dans l'état actuel.

Enfin, La substitution d'un service d'un composant par un service d'un autre composant pourrait être assurée, si la vision composant orienté service était étendue avec les hypothèses suivantes, à savoir :

- les algorithmes réalisant la fourniture d'un service doivent être disjoints deux à deux,
- ils ne doivent pas interagir entre eux,
- les services fournis doivent préciser les services requis qu'ils utilisent
- et, lors d'une substitution, tous les algorithmes doivent conserver les liaisons qu'ils possèdent avec chaque interface.

8. CONCLUSION

Dans ce chapitre concernant l'approche à service, nous avons présenté une définition synthétique de la notion de service après avoir examiné les différentes propositions actuelles. Nous avons également présenté les visions de l'approche à services promues par le W3C (via WSDL), l'Alliance OSGi et la collaboration OSOA (via SCA), ainsi qu'un tableau de synthèse de ces visions technologiques. Nous avons discuté de leur pertinence dans le cadre de la définition d'applications à services.

Pour conclure, nous allons examiner les thèmes de recherche majeurs relatifs à l'approche à service. Tout d'abord, il faut se rappeler que cette approche est un domaine relativement récent et que son but est de permettre et de faciliter à la mise en place effective de services et de solutions orientées service. A notre sens, trois axes majeurs de recherche peuvent être dégagés. Ils portent sur les aspects fondamentaux de l'approche, les modèles économiques et sur l'architecture orientée service. Ces trois axes peuvent être retrouvés dans les conférences majeures sur les services telles, par exemple, *International Conference on Web Services* (ICWS) et *Services Computing Conference* (SCC).

Le premier axe regroupe les préoccupations ayant trait au génie logiciel, à la mise à disposition, au déploiement, à la maintenance, à l'autonomique, aux accords de niveau de service, aux patrons d'interaction entre services, aux applications à services et aux standards pour les services. Le second axe agrège les préoccupations concernant le contrôle des services *Business-to-Business* (B2B) intra ou inter entreprises et des services *Business-to-Customer* (B2C), les modèles économiques pour les services (les calculs de frais, de coûts pour les transactions) dans le cadre, par exemple, de la santé, de la finance, de l'aéronautique. Enfin, le troisième axe principal de recherche porte sur l'architecture orientée service et se focalise sur la mise en place et les déclinaisons du patron SOA, ainsi que sur les standards autour du SOA.

En complément, dans [PTD+07] M. Papazoglou fait un point sur les défis "Recherche" concernant l'approche à service (ou SOC). Comme présenté dans la figure ci-dessous, trois domaines de recherche sont considérés :

- Le premier domaine concerne les aspects fondamentaux du SOC. Il regroupe les défis ayant trait à la publication, découverte, sélection et liaison des services, ainsi que l'expression, la spécification de contrats de service.
- Le second comprend les problématiques liées la composition de services et des propriétés que doit apporter et garantir cette composition (typiquement la coordination, la conformité, les transactions, la qualité de service, etc.).
- Enfin, le troisième et dernier domaine de recherche concerne les fonctionnalités de plus haut niveau (c'est-à-dire les plus avancées). Ses défis sont l'expression de métriques de qualité de service, la gestion de l'état du service et du changement de (fournisseur de) service, ainsi que le *management* (dont la phase de déploiement fait partie) et la surveillance des services.

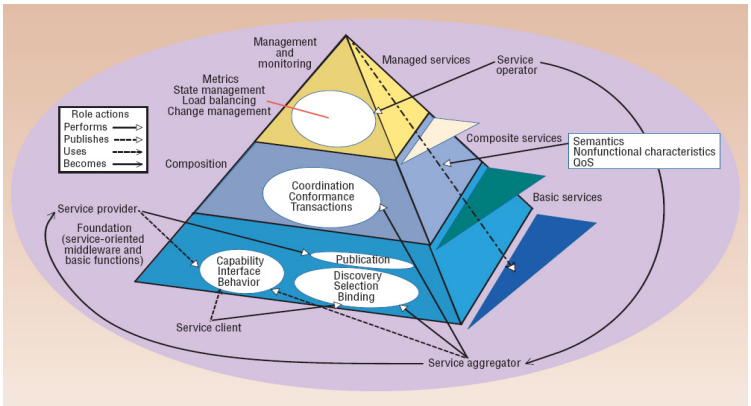


Figure 20. Les défis "Recherche" du SOC identifiés dans [PTD+07]

D'un point de vue général, l'ensemble des défis identifié dans [PTD+07] est cohérent et relativement complet au vue des recherches qui sont menées dans le domaine du SOC et des articles qui sont publiés. Ainsi, concernant les aspects fondamentaux du SOC, dans [PAB07], les auteurs s'intéressent à l'automatisation de la découverte de services en utilisant la technologie du web sémantique. Ils proposent une approche pour la découverte de services web qui combine ontologie et index sémantique. Dans [ZYC+07], les chercheurs se focalisent sur les approches décentralisées pour la découverte de services web, et plus particulièrement sur le patron pair-à-pair. Dans ce cadre, ils promeuvent un *framework* pair-à-pair utilisant un système de graphe nommé *Skip Graph*. Enfin, dans [ZMS07], les auteurs présentent un *framework* qui supporte la découverte de services fournissant des fonctionnalités données et qui doivent satisfaire des propriétés et des contraintes provenant de systèmes logiciels orientés service lors de la phase de conception de ces derniers.

De même, des recherches concernant la composition de services sont, elles-aussi, menées actuellement, par exemple, dans [ML07], [LYL07] et [BC07]. Dans [ML07], C. Marin et P. Lalanda présentent un environnement de développement multi-domaines qui automatise la composition de services, en se basant sur une approche dirigée par des modèles. Dans [LYL07], les auteurs proposent une approche pour analyser et valider la correction de compositions de services web sémantiques. Ensuite, dans [BC07], les chercheurs introduisent une méthodologie pour composer les accords de niveau de service (SLA) associés à une composition (c'est-à-dire dans leur cas, à un *workflow*) de services web.

Enfin, par exemple, dans [ABD+07], [CTA07] et [NB07], les auteurs présentent des travaux relatifs au troisième domaine de recherche défini par M. Papazoglou, c'est-à-dire à la gestion et à la surveillance des services. Ainsi, dans [ABD+07], les chercheurs proposent une méthodologie conceptuelle pour l'automatisation du processus de gestion de services IT. Dans [CTA07], les auteurs proposent une approche pour l'intégration et la gestion de processus métiers orientés service. Pour finir, dans [NB07], M. F. Nowlan et M. Brian Blake proposent une architecture et des protocoles de communications qui ciblent les solutions orientées service qui s'étendent sur plusieurs entreprises et/ou organisations et dont le but est la gestion de l'utilisation de services web.

Cependant, bien que le défi de la composition de services soit mis en avant, le concept d'application à services en tant que tel et les problématiques qu'il engendre ne sont pas relevés. En complément, certes, le défi autour de la gestion (*management*) et donc du déploiement d'un service est souligné, mais la gestion (et donc le déploiement) d'applications à services, quant à lui, ne l'est pas. Or les solutions qui doivent être apportées aussi bien pour la gestion des services que des applications à services sont loin d'être exemptes de défis.

Pour finir, concernant l'expression et la manipulation d'applications à services, l'ensemble de cet état de l'art nous permet de conclure que les visions service et composants orientés service, qui existent à l'heure actuelle, ne sont pas entièrement satisfaisantes. Nous verrons dans cette thèse que l'approche dirigée par les modèles est une approche novatrice et originale qui est pertinente et qui peut être utile dans le cadre du déploiement de services et d'applications à services. Une des contributions de cette thèse consiste donc à proposer un modèle de service qui permette de définir des applications à services, tout en respectant le SOC, l'utilisation du patron SOA et les six propriétés fondamentales des services. Ce modèle de service est la base de notre méta-modèle pour les applications à services. Ce dernier nous a permis de définir un méta-modèle minimal de spécification d'applications à services permettant leur déploiement. A ce propos, notre état de l'art sur le déploiement est présenté dans le prochain chapitre.

II. LE DÉPLOIEMENT

Dans le chapitre précédent, nous nous sommes focalisés sur l'approche à service. Dans ce chapitre-ci, nous allons nous concentrer sur la notion de déploiement logiciel. Nous commencerons par examiner les définitions majeures. Nous regarderons ensuite le déploiement d'un point de vue technologique au travers des spécifications et technologies *Software Dock*, puis CCM v4.0, EJB v3.0, .Net et enfin OSGi R4.1 et SCA v1.00.

D'après le "Petit Robert", le verbe déployer a pour définition : "développer dans toute son extension (une chose qui était pliée)". Parmi ses synonymes, il est possible de trouver les verbes : déplier, ouvrir et mettre en œuvre. En informatique et en génie logiciel en particulier, le déploiement logiciel consiste donc à développer (dans le sens de déplier) dans toute son extension une entité logicielle pliée (c'est-à-dire packagée). Le terme unité de déploiement est généralement utilisé pour désigner une entité logicielle packagée.

Ce chapitre comprend sept sections. La première section présente des éléments de base liés à la phase de déploiement, ainsi que la place de cette phase dans le cycle de vie logiciel. La seconde détaille les trois définitions majeures du déploiement, à savoir, celles de C. Szyperski, de l'*Object Management Group* et de A. van der Hoek et R. S. Hall [HHC+98] que nous appelons définition de l'Université du Colorado. La troisième section présente notre système de caractérisation du déploiement et une première application de ce système sur un prototype de "Recherche" reconnu : *Software Dock*. La quatrième section présente trois cas d'études autour du déploiement de composants, à savoir, Corba Component Model V4.0 (CCM), Enterprise Java Beans (EJB 3.0), Microsoft .Net Framework. La cinquième section présente, quant à elle, deux cas d'études autour du déploiement de services. L'avant dernière section effectue la synthèse des six cas d'études présentés. Enfin, la septième et dernière section conclue cet état de l'art sur le déploiement.

1. INTRODUCTION

Dans cette première section, nous présentons des généralités qui s'appliquent à la phase de déploiement, ainsi que la place de la phase de déploiement dans le cycle de vie logiciel.

1.1 Généralités

La phase de déploiement en elle-même est constituée par un ensemble d'activités. Nous verrons, dès la prochaine section, qu'il n'existe pas à l'heure actuelle de consensus autour de l'ensemble des activités qui font partie de la phase de déploiement. La phase de déploiement d'un logiciel commence généralement une fois que l'administrateur a acquis le logiciel (et les licences correspondantes) par le biais de son entreprise. Du point de vue de l'entreprise, c'est l'administrateur qui est responsable des environnements d'exécutions de l'entreprise. Ces derniers sont la cible de la phase de déploiement à proprement parler.

Dans le cadre de la phase de déploiement, un administrateur spécifie l'ensemble des activités de déploiement devant être exécutées, ainsi que l'ensemble des logiciels et l'ensemble des environnements d'exécution cibles de ces activités. Il spécifie donc des instructions de déploiement qui sont ensuite transmises au déployeur. Ce dernier est en charge et est responsable de leur exécution. Une fois que les instructions de déploiement ont été exécutées, le déployeur transmet le résultat de l'exécution de chaque instruction à l'administrateur.

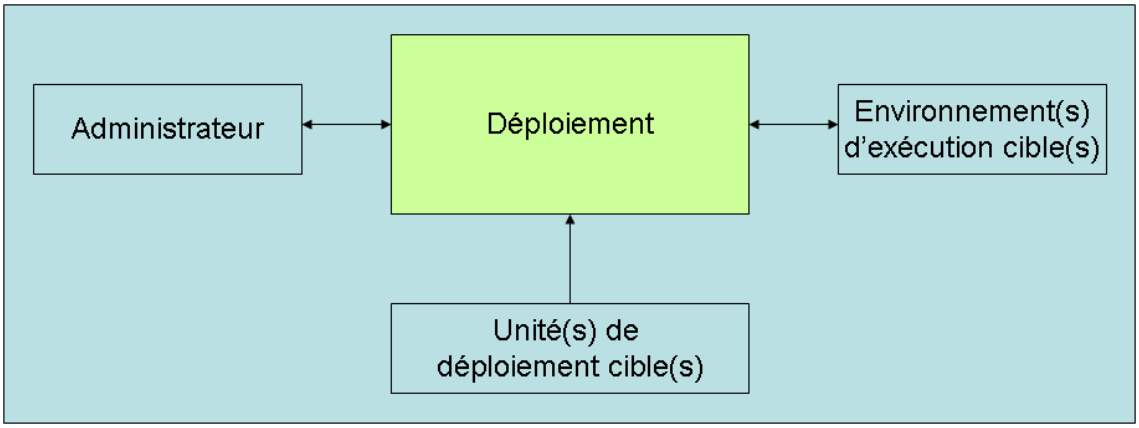


Figure 21. Diagramme de contexte du déploiement.

Par abus de langage, le terme déploiement (de) logiciel est très souvent utilisé et laisse à penser que l'unité de base de la phase de déploiement est le logiciel. Or dans les faits, les éléments qui sont effectivement déployés sont les unités de déploiement. Une unité de déploiement est une entité logicielle packagée de façon à pouvoir être déployée. Ainsi, le logiciel acquis peut être une unité de déploiement en lui-même ou être constitué d'un ensemble d'unités de déploiement.

> Une unité de déploiement est un délivrable qui est exécutable dans un environnement d'exécution, sans qu'un humain ait besoin d'intervenir pour le modifier afin de le rendre effectivement installable et prêt à être exécuté.

En complément, il peut exister plusieurs unités de déploiement pour un logiciel donné (voire même pour une même version). Typiquement, dans le cas du logiciel WinRar, il existe une unité de déploiement pour le système d'exploitation Windows, une pour Unix et une pour Mac OS. Ainsi, lors du déploiement de ce logiciel, le déployeur pourra être amené à sélectionner l'unité de déploiement à déployer et cela en fonction, bien évidemment, de la version demandée, mais aussi, par exemple, en fonction du système d'exploitation cible, ou de l'espace disque restant, ou afin d'homogénéiser la version

déployée sur un ensemble d'environnements d'exécution. Il est important de noter qu'une telle étape de sélection est obligatoire dans le cadre d'un déploiement dirigé par les modèles, puisque le point d'entrée de la phase de déploiement est un modèle du logiciel, et non un logiciel dont les unités de déploiement sont spécifiées.

Les phases de développement et de déploiement sont totalement disjointes. La phase de développement se termine lorsque les implémentations développées ont été packagées et sont devenues des unités de déploiement. Ainsi, ni l'administrateur, ni le déployeur n'interviennent dans le packaging des unités de déploiement. Cependant, les unités de déploiement récupérées par l'administrateur peuvent encore être configurées, soit par l'administrateur lui-même, soit par le déployeur. Ainsi, ils peuvent être amenés à préciser des valeurs par défaut, comme l'adresse IP de l'environnement d'exécution dans laquelle l'unité de déploiement va être déployée, les unités de déploiement avec lesquelles elle va pouvoir interagir, le système de mesures à utiliser, la langue.

Une unité de déploiement doit préciser ce qu'elle offre et ce qu'elle requiert. Dans le cas des composants, par exemple, ce qu'elle offre est défini par le biais d'interfaces fournies (*provides-interfaces*) et ses requis sont définis par le biais d'interfaces requises (*requires-interfaces*) [Szy03]. Dans le cas de la technologie OSGi, l'unité de déploiement est le *bundle*. Celui-ci spécifie les packages Java qu'il fournit et qu'il requiert par le biais de métadonnées (spécifiées dans son fichier manifest.mf), par contre il n'est pas obligé de préciser les services fournis et requis. D'une manière générale, une unité de déploiement exprime ce qu'elle fournit et ce qu'elle requiert afin de permettre son déploiement. Il existe cependant des exceptions ; le cas OSGi pour le niveau service l'illustre bien.

Dans le cadre d'un logiciel, les requis d'une unité de déploiement peuvent être satisfaits soit par d'autres unités de déploiement apparaissant dans le même logiciel, soit par des unités de déploiement apparaissant hors du logiciel (par exemple, par des unités de déploiement appartenant à d'autres logiciels déployés), soit par l'environnement d'exécution dans lequel le logiciel est déployé. Le terme ressource est utilisé communément pour désigner les artefacts qu'une unité de déploiement peut requérir. Satisfaire les requis (exprimés ou non) d'une unité de déploiement est généralement nécessaire à sa future exécution et donc à la future exécution du logiciel. Par exemple, le logiciel Eclipse doit spécifier qu'il nécessite un *Java Development Kit* (JDK) pour être déployé (l'installation d'Eclipse peut se faire sans la présence du JDK, mais son activation ne peut être faite sans JDK). Il existe certaines unités de déploiement qui pourront être activées, que leurs requis soient ou non satisfaits, néanmoins, leur fonctionnement sera vraisemblablement diminué si leurs requis ne sont pas satisfaits. Typiquement, exécuter Eclipse sans l'exécuteur de script Ant permettra certes d'éditer des programmes, mais ne permettra pas de les compiler.

Nous venons de donner quelques précisions quant aux unités de déploiement et à la satisfaction de leurs requis. Ces précisions ont une grande importance dans le cadre du déploiement d'assemblage d'unités de déploiement. En effet, un assemblage ne peut être considéré installé ou activé que lorsque toutes ses unités de déploiement sont elles aussi installées ou respectivement activées. Néanmoins, dans le cadre du déploiement d'assemblages, les précisions que nous avons données jusqu'ici sont nécessaires, mais insuffisantes. Il est, en effet, nécessaire de définir la notion d'ordonnancement. Ordonnancer, c'est trouver l'ordre dans lequel les différentes unités de déploiement d'un assemblage doivent être activées ou respectivement désactivées, afin de permettre l'activation ou respectivement la désactivation de l'assemblage. Prenons deux exemples concrets, si un ensemble d'EJB est regroupé dans une seule unité de déploiement, alors, le déployeur n'a pas à se préoccuper d'ordonnancer les différents EJB. Par contre, si les EJB sont livrés dans plusieurs unités de déploiement, le déployeur va devoir ordonnancer les différentes unités de déploiement. Concrètement, dans le premier cas, lors de l'activation, tous les EJB sont chargés par le serveur EJB en même temps et l'exécution de l'assemblage à proprement parler ne commence qu'une fois tous les EJB activés (c'est-à-dire lorsque l'assemblage devient lui-même actif). Dans le second cas, si une unité de déploiement est active (c'est-à-dire en exécution) avant que toutes les unités de déploiement qu'elle requiert ne soient elles aussi actives, alors, la première unité de déploiement tentera de localiser (*look-up*) des EJB qui n'apparaitront pas (encore) dans le registre JNDI. Cela entraîne la levée d'exceptions et un possible *crash* de la première unité de déploiement. La technologie OSGi rencontre elle aussi ce problème de levée d'exception et de *crash* lors de l'activation et de la désactivation d'assemblage d'unités de déploiement.

Dans le cadre de la phase de déploiement, deux types de logiciels peuvent être distingués : les logiciels distribués et les logiciels centralisés. Le traitement ou non de l'aspect distribué d'un logiciel est un facteur discriminant pour un système de déploiement. Si la phase de déploiement concerne un logiciel dont l'ensemble des unités de déploiement est centralisé dans un et un seul environnement d'exécution, il est alors question de déploiement centralisé. A l'inverse, si la phase de déploiement porte sur le déploiement d'un logiciel dont les unités de déploiement sont distribuées dans plusieurs environnements d'exécution, alors, il est question de déploiement distribué.

Un autre point discriminant au niveau de la phase déploiement est le fait que le déploiement soit effectué à distance ou en local. Il est question de déploiement à distance lors que le déployeur et l'environnement d'exécution ne sont pas localisés au même endroit. D'un autre côté, si le déployeur peut interagir physiquement avec l'environnement d'exécution et si il se sert de ce dernier lors de la phase de déploiement, il est alors question de déploiement en local (c'est, par exemple, le cas lorsque chacun de nous déploie un logiciel sur son ordinateur personnel, depuis ce même ordinateur personnel). Néanmoins, la distinction qui est faite entre déploiement local ou à distance peut prêter à confusion. Concrètement, il est tout à fait possible de déployer en local une unité de déploiement qui est située sur un site distant (par exemple lorsque l'on installe un logiciel situé dans un site web sur son ordinateur personnel). Il est aussi tout à fait possible de déployer à distance une unité de déploiement qui se trouve déjà dans l'environnement d'exécution. Par exemple, c'est le cas, lorsque qu'un déployeur français doit activer le logiciel qu'il a installé dans une grille canadienne quelques heures auparavant. Enfin, il est tout à fait possible de déployer une unité de déploiement dans un environnement d'exécution, en utilisant ce même environnement pour faire appel à une machine distante qui va concrètement effectuer le déploiement dans l'environnement ; cependant, dans ce dernier cas, l'utilisateur joue le rôle d'administrateur, la machine distante joue celui du déployeur, donc ce déploiement est un déploiement à distance. Ce qu'il faut retenir de la notion de déploiement local ou à distance c'est qu'elle sert à préciser l'existence ou non d'une distance entre le déployeur et l'environnement d'exécution.

Il convient aussi de distinguer si un déploiement est sensible au contexte de l'environnement d'exécution cible. Le terme contexte peut regrouper indifféremment plusieurs informations. Ces informations peuvent concerner, par exemple, le système d'exploitation, les paramètres de l'environnement d'exécution, ainsi que les logiciels et unités de déploiement qu'il contient, les équipements qui lui sont raccordés. La caractéristique concernant la prise en compte du contexte de l'environnement d'exécution peut paraître anodine, mais, dans les faits, cela est loin d'être le cas. En effet, la prise en compte du contexte d'un environnement d'exécution est le point d'entrée de tout gestionnaire de partage d'unités de déploiement entre logiciels. De même, tout gestionnaire d'isolation entre logiciels (partageant ou non des unités de déploiement) requiert lui aussi la prise entre compte du contexte de l'environnement d'exécution.

Nous avons déjà précisé que toute unité de déploiement ou assemblage d'unités de déploiement doit spécifier, entre autre ce qu'il fournit et ce qu'il requiert. Cependant, dans le cas d'un assemblage distribué, il devra aussi spécifier la distribution de ses unités de déploiement sur les différents environnements d'exécution cibles. De même, une unité de déploiement ou un assemblage pourra aussi spécifier d'autres propriétés, telles, par exemple, la version, l'identifiant de l'entreprise qui est à l'origine de la fabrication. Trois termes sont souvent utilisés pour faire référence à ces informations, les termes descripteur de déploiement, configuration et modèle ; et cela aussi bien pour les unités de déploiement que pour les assemblages d'unités de déploiement. Il est important que noter que les deux premiers termes désignent des informations qui sont extraites à partir des unités de déploiement. D'un autre côté, un modèle peut être construit, soit en utilisant des informations provenant des unités de déploiement, soit de manière plus abstraite, sans partir d'une unité de déploiement, soit en faisant un mélange entre les deux approches.

Enfin, le terme plan de déploiement est le dernier terme que nous préciserons ici. Pour expliciter ce terme, il faut rappeler que lors du déploiement d'unités de déploiement dans des environnements d'exécution, le déployeur s'appuie sur des mécanismes fournis par les environnements d'exécution. Par exemple, dans le cas du déploiement via des *installers*, ces derniers s'appuient sur les capacités de copie de fichiers offertes par les environnements d'exécution ; pour cette thèse, le déploiement d'assemblages d'unités de déploiement se base sur les capacités de déploiement d'unité de déploiement offertes par les

environnements d'exécution OSGi. Le déployeur doit donc transformer les instructions qu'il reçoit de l'administrateur vers un ensemble d'instructions basées sur les capacités de ses environnements d'exécutions cibles. L'ensemble d'instructions ainsi produit par le déployeur est appelé plan de déploiement.

1.2 Déploiement et cycle de vie

Cette sous-section est une introduction concernant la place de la phase de déploiement dans le cycle de vie logiciel. Plus précisément, nous allons nous intéresser aux cycles de vie définis d'une part par le glossaire standard IEEE concernant la terminologie du génie logiciel et d'autre part par l'entreprise Microsoft via sa plateforme *Microsoft Solutions Framework* (MSF) [IEEE90] [.Net07]. Le glossaire IEEE a été publié en 1990 et notre document de référence concernant le déploiement dans MSF a été publié pour la première fois en janvier 2003 et sa dernière mise à jour date de fin 2007.

Le cycle de vie logiciel défini par le glossaire IEEE est linéaire (voir la figure ci-dessous). Il est intéressant de remarquer que le terme de déploiement n'apparaît ni dans le cycle de vie logiciel proposé, ni dans le glossaire en lui-même.

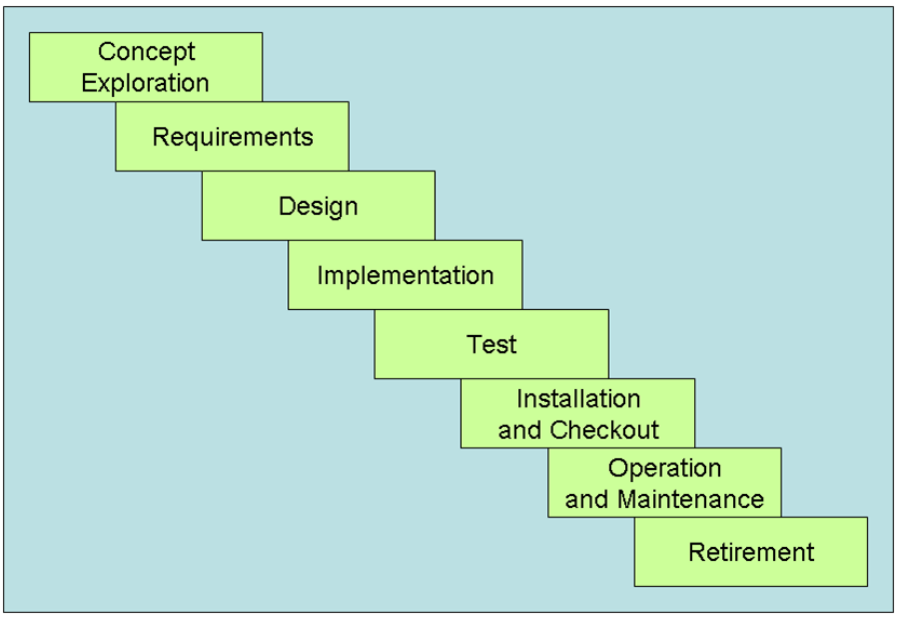


Figure 22. Cycle de vie logiciel défini dans [IEEE90].

Cependant, bien que le terme de déploiement n'apparaisse pas, en tant que tel dans le glossaire IEEE, force est de constater que les activités d'*Installation*, de *Maintenance* et de *Retirement* (c'est-à-dire de désinstallation) apparaissent clairement. De plus, les activités d'activation et de désactivation se retrouvent aussi dans ce cycle de vie, de manière indirecte, via la phase *Operation*. Nous verrons dans les prochaines sections qu'il existe plusieurs définitions de la phase de déploiement et que tout ou partie de ces activités est inclus dans la phase de déploiement suivant la définition choisie. En complément, le glossaire IEEE définit, d'une part, l'activité *Checkout* comme étant l'ensemble des tests conduits dans l'environnement opérationnel (c'est-à-dire dans l'environnement cible), afin d'assurer que le produit logiciel fonctionne correctement suite à son installation, d'autre part, l'activité *Operation* comme étant la période de temps pendant laquelle le produit logiciel est utilisé dans son environnement d'opération (c'est-à-dire, dans son environnement d'exécution) et pendant laquelle la qualité de ses performances est surveillée [IEEE90]. Enfin, il est important de remarquer que le glossaire IEEE situe l'ensemble des activités *Installation*, *Operation*, Maintenance et *Retirement* à la fin du cycle de vie logiciel.

La plateforme MSF définit, quant à elle, un cycle de vie logiciel en spirale [.Net07]. MSF présente ce cycle de vie comme étant un ensemble de bonnes pratiques, de bons principes et modèles qui peuvent être utilisés pour planifier, construire et déployer des logiciels. Comme il est possible de le voir dans la figure ci-dessous, ce cycle de vie contient cinq phases, à savoir, la prospection, la planification, le développement, la stabilisation et, pour finir, le déploiement. MSF précise que la phase de déploiement commence une fois que le logiciel a été publié et ne se termine que lorsqu'elle est jugée complète. Cependant, MSF ne précise pas quels sont les critères qui permettent de juger de la complétude de la phase de déploiement. Parallèlement à cela, MSF précise aussi que la phase de déploiement ne se finit pas forcement une fois l'installation du logiciel faite. Ainsi, la phase de déploiement de MSF peut aussi comprendre les activités d'activation et de désactivation, de même que, celles de mise à jour et de désinstallation.

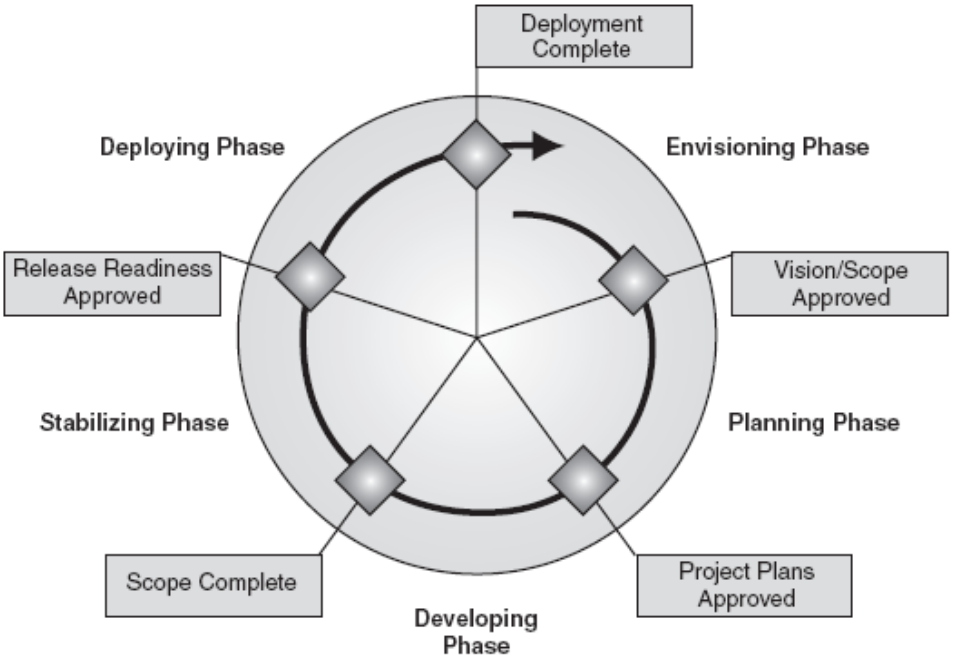


Figure 23. Cycle de vie logiciel (v3.1) défini par MSF [.Net07].

A notre sens, les informations importantes à retenir concernant la place de la phase de déploiement dans les cycles de vie logiciel présentés sont les suivantes :

- la phase de déploiement occupe la dernière phase du cycle de vie logiciel,
- elle commence par l'installation du logiciel,
- elle peut se poursuivre par sa possible exécution, sa possible maintenance et sa possible désinstallation.

2. DÉFINITIONS

Nous pourrions croire, à tort, que la phase de déploiement est aujourd'hui standardisée. Il n'en est rien. Ainsi, dans les faits, trois définitions majeures sont reconnues et utilisées.

Le but de cette section est de présenter ces trois grandes définitions. Nous commencerons par la définition de Clemens Szyperski [Szy03], puis nous poursuivrons par celle proposée par l'*Object Management Group* (OMG) [OMG08] et enfin, nous finirons par la définition de l'université du Colorado [HHC+98]. Ces trois définitions sont présentées en commençant par celle de plus faible scope pour finir par celle de plus large scope.

2.1 Définition de C. Szyperski

La définition du déploiement proposée par C. Szyperski vise explicitement le domaine des composants logiciels, sans pour autant préciser si elle se limite ou non à ce domaine. Le déploiement est défini comme étant l'étape de préparation d'un composant en vue de l'installer dans un environnement spécifique. C. Szyperski précise explicitement que le déploiement (c'est-à-dire, la préparation) revient à renseigner les paramètres d'un descripteur de déploiement [Szy03].

Nous allons maintenant présenter quelles sont les étapes qui précèdent et qui suivent cette phase de déploiement. Cela va nous permettre d'expliciter où cette phase de déploiement est située dans le cycle de vie logiciel et quelle est sa portée. L'étape qui précède le déploiement est l'acquisition. C'est l'étape d'obtention d'un composant logiciel. C. Szyperski précise que tout composant acquis est déployable. Ce qui signifie qu'un composant proposé à l'acquisition est un livrable exécutable par une machine physique ou virtuelle. Plus précisément, dans [Szy03], l'auteur met en avant le fait que tout composant doit être une unité de déploiement (page 686). Il définit ensuite une unité de déploiement comme un délivrable exécutable dans un environnement d'exécution, sans qu'un humain n'ait besoin d'intervenir pour modifier le composant afin de le rendre effectivement installable et prêt à être exécuté. L'installation est l'étape qui suit immédiatement le déploiement. Elle rend un composant disponible sur un site (*host*) particulier, dans un environnement particulier. Il est précisé que cette étape d'installation est souvent automatisée. Enfin, l'étape d'installation est suivie par le chargement (*Loading*) qui consiste à lancer l'exécution d'un composant dans un contexte d'exécution particulier.

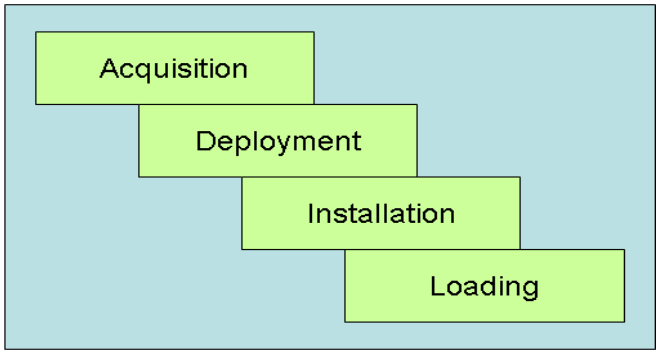


Figure 24. La phase de déploiement et son contexte selon [Szy03].

2.2 Synthèse

La proposition de C. Szyperski concernant la phase de déploiement possède, à notre sens, deux points positifs, à savoir :

- Les phases de développement et de déploiement sont clairement distinguées (via le concept d'unité de déploiement),
- L'acquisition est identifiée comme étant une pré-condition à la phase de déploiement.

Cependant, cette proposition nécessiterait quelques précisions supplémentaires, comme, par exemple :

- Aucune mention n'est faite à propos des rôles d'administrateur et de déployeur. Par exemple, lorsque l'étape d'acquisition est définie, aucune précision n'est donnée concernant l'acteur qui acquiert l'unité de déploiement.
- De même, il aurait été opportun de préciser les étapes (minimales) suivant le chargement.
- Aucune précision n'est donnée quant au fait que cette définition englobe aussi le déploiement d'assemblages d'unités de déploiement comme, par exemple, les applications à composants.
- Enfin, dans le cadre du déploiement d'assemblages de composants, aucune précision n'est non plus donnée quant à la nécessité ou l'existence d'un mécanisme chargé d'ordonnancer l'installation et/ou le chargement des composants (c'est-à-dire un mécanisme chargé d'établir un ordre entre les composants pour l'installation et/ou le chargement d'un assemblage de composants).

Il est essentiel de remarquer que cette définition du déploiement a une portée relativement faible, puisqu'elle consiste uniquement à préparer une unité de déploiement en vue de son installation.

Cette définition s'applique particulièrement bien dans le cadre du déploiement d'une unité de déploiement (composées d'un ou plusieurs composants) dans une grille via un logiciel automatisant une grande part du travail du déployeur. Ce dernier n'ayant alors qu'à préparer l'unité de déploiement en vue de sa future installation. Cela donne le déroulement suivant : le déployeur commence par préparer l'unité de déploiement, puis il la passe comme paramètre au logiciel de déploiement, ce dernier se charge d'installer l'unité de déploiement dans la grille puis de l'activer. L'unité de déploiement s'exécute alors jusqu'à finir la tâche pour laquelle elle a été activée. Une fois son exécution terminée, le résultat de tâche est retourné. L'unité de déploiement se trouve alors dans l'état désactivé. La désinstallation de l'unité est pour finir réalisée de manière indirecte via la remise à zéro de la grille, afin que cette dernière puisse servir à l'exécution de nouvelles unités de déploiement.

2.3 Définition de l'Object Management Group

Nous allons présenter, maintenant, la définition de la phase de déploiement proposée par l'*Object Management Group* (OMG) via sa spécification *Deployment and Configuration of Component-based Distributed Applications* (D&C). Cette présentation se base sur la version 4.0 de la spécification, qui a été publiée en avril 2006 et cible clairement les applications distribuées à base de composants [OMG06].

La spécification D&C voit la phase de déploiement comme un processus linéaire et séquentiel. Elle définit un ensemble de trois pré-conditions au processus de déploiement :

- La première pré-condition stipule que le fournisseur du logiciel doit packager le logiciel conformément à la spécification D&C. Ainsi, tout *package* doit contenir des métadonnées décrivant le logiciel et les binaires exécutables des différents éléments composant le logiciel. Plusieurs binaires exécutables peuvent être proposés pour un même élément. Le *package* doit, ensuite, être publié et mis à la disposition du déployeur via, par exemple, d'un CDROM, de l'URL d'un site FTP.
- La seconde pré-condition impose l'existence et la disponibilité d'un environnement cible. Cet environnement est une infrastructure composée de systèmes distribués (ordinateurs, réseaux, etc.) sur laquelle le logiciel est destiné à être exécuté.
- Enfin, la troisième et dernière pré-condition rend obligatoire la présence d'un dépôt (*repository*). Ce dépôt étant, au minimum, une zone dans laquelle le logiciel packagé est stocké avant le début du processus de déploiement.

Le processus de déploiement en lui-même débute lorsque le logiciel est acquis par un acteur qui, par cette action, devient le propriétaire du logiciel (*software owner*). Un logiciel ne peut être acquis qu'après avoir été développé, packagé, publié par un fournisseur de logiciel. Le logiciel est déployé par son propriétaire qui endosse, par la même, le rôle de déployeur.

La phase de déploiement est structurée en cinq étapes, à savoir, l'*Installation*, la *Configuration*, le *Planning*, la *Preparation* et le *Launch*. Ces cinq étapes sont enchainées les unes après les autres de manière linéaire et séquentielle.

La première étape est l'installation. Elle est définie comme la récupération, l'acquisition d'un package logiciel (*software package*) publié et son rapatriement dans un dépôt logiciel sous le contrôle du déployeur. Un dépôt peut se situer, ou non, sur la même machine ou sur le même système de fichier que celui dans lequel le logiciel va être déployé. C'est une zone qui permet au déployeur d'appliquer des politiques au logiciel, comme l'authentification ou la certification (du logiciel), avant de lancer toute étape concernant l'exécution du logiciel à proprement dit. Un dépôt ne doit pas nécessairement être persistant et ne doit pas, non plus, nécessairement stocker ou avoir une copie du logiciel ou des métadonnées du logiciel. L'OMG propose, aussi, sa propre définition de l'activité d'installation. Ainsi, l'installation n'est pas définie comme le déplacement d'un logiciel vers l'environnement d'exécution où il pourra ensuite être activé, mais simplement comme le déplacement du logiciel acquis vers le dépôt du déployeur.

La seconde étape du processus de déploiement est la configuration. Pour être configuré, un package logiciel doit être installé dans le dépôt du déployeur. Le déployeur est le seul à pouvoir effectivement le configurer. Les possibilités de configuration offertes dépendent de chaque package logiciel. Un package logiciel peut, par exemple, offrir des options de configuration concernant la langue à utiliser, le système de mesure, le délai d'attente entre chaque mesure ou encore la fréquence et le formatage des rapports de mesures générés. Plusieurs configurations peuvent être associées à un même package logiciel. Enfin, cette seconde étape ne concerne que la configuration des fonctions du package logiciel et n'est en aucun cas destinée à la prise de décisions concernant le déploiement comme, par exemple, le choix de l'implémentation à utiliser ou, le cas échéant, le choix de la distribution des différents éléments du package logiciel.

La troisième étape est la planification (*Planning*). Son but est de définir comment et où une configuration va être exécutée dans l'environnement cible. Pour rappel, la spécification définit un environnement comme étant une infrastructure composée d'ordinateurs et de réseaux. L'activité de planification prend en compte les requis de la configuration à déployer, ainsi que les ressources offertes par l'environnement cible et sélectionne les implémentations à utiliser. Elle décide également comment et où la configuration donnée sera exécutée dans l'environnement cible. Le résultat de l'étape de planification est un plan de déploiement spécifique (à la configuration en cours de déploiement, ainsi qu'à l'environnement cible). Le déploiement ainsi mis en œuvre est un déploiement sensible au contexte.

La planification met en avant deux modes, à savoir, la planification juste à temps (*Just-in-time planning*) et la planification par avance (*Advanced planning*). Le premier mode consiste à calculer le plan de déploiement d'une configuration donnée pour un environnement cible et à le faire exécuter immédiatement via l'étape de préparation. Nous verrons dans le paragraphe suivant que l'étape de préparation consiste à appliquer, exécuter, mettre en œuvre le plan de déploiement. Ce premier mode permet de ne pas faire d'hypothèses concernant l'évolution des ressources de l'environnement cible. En effet, les ressources d'un environnement peuvent évoluer au cours du temps et par conséquent invalider le plan de déploiement produit par la planification. Le second mode de planification est le mode par avance. Contrairement à l'approche précédente, ce second mode consiste certes toujours à produire un plan de déploiement pour une configuration donnée et un environnement cible, mais cette fois, le plan de déploiement produit n'est pas destiné à être mis en œuvre immédiatement (toujours via l'étape de préparation). Ainsi, la planification par avance permet, par exemple, d'éviter les interactions avec l'environnement cible si le système de déploiement en possède une image, un modèle. De plus, si une configuration est destinée à être déployée dans un ensemble d'environnements identiques, alors la planification par avance permet de ne calculer qu'un unique plan de déploiement qui pourra être appliqué à cet ensemble d'environnements cibles, permettant ainsi d'économiser le calcul d'étapes de planification. Cependant, ce second mode suppose que les ressources de l'environnement cible requises par le plan de déploiement du logiciel soient toujours disponibles lors de la mise en œuvre effective de ce dernier. La spécification ne précise rien quant à l'invalidation d'un plan déploiement (par exemple l'idée d'une date de péremption pourrait être introduite au niveau des plans de déploiement).

L'avant dernière étape est appelée préparation. Elle est définie comme la mise en œuvre effective des décisions prises lors de la planification. Ces décisions sont spécifiées dans le plan de déploiement. Le but de la préparation est de rendre le logiciel prêt à être exécuté. Cette mise en œuvre consiste, entre autre, à déplacer des binaires exécutables dans les ordinateurs qui leur ont été attribués. Comme pour la planification deux modes de préparation sont distingués. Le premier est le mode juste à temps, le second est le mode par avance. Le mode juste à temps implique que l'étape de préparation doit immédiatement être suivie par l'étape de lancement. Le mode par avance n'impose pas cette contrainte.

La cinquième et dernière étape est le lancement (*Launch*). Elle consiste à démarrer l'exécution du logiciel, en prenant, dans l'environnement, toutes les ressources requises par le ou les packages logiciels. Le lancement d'une application à base de composants implique l'instanciation des composants dans les ordinateurs de l'environnement cible, la configuration de ces instances de composants et la mise en place des interactions entre les différentes instances et enfin, le démarrage de l'exécution de l'application en elle-même. Dès que l'application s'exécute (c'est-à-dire une fois qu'elle est dans l'état activé) alors, soit l'application s'exécute tant que son exécution n'est pas complète, soit elle est arrêtée (c'est-à-dire désactivée) en utilisant la même infrastructure de déploiement que celle par laquelle son lancement a été effectué.

Les cinq étapes que nous venons de présenter peuvent tout à fait être déroulées une par une séparément, ou comme une seule grande étape de déploiement entièrement automatisée. La figure ci-dessous présente ces cinq étapes, ainsi que les liens entre ces étapes et le vecteur (c'est-à-dire l'unité de déploiement) utilisé par chaque lien.

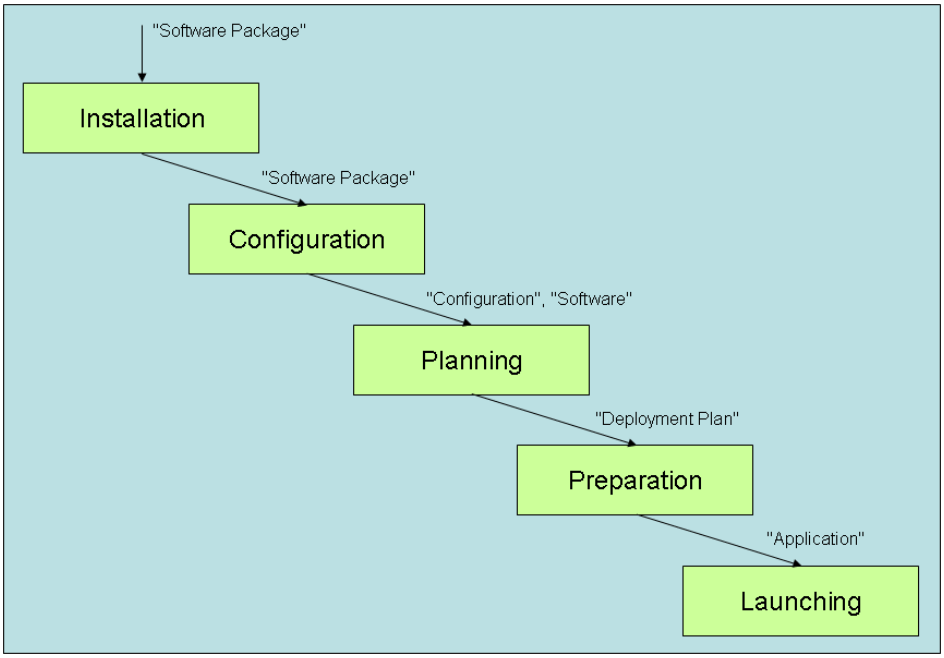


Figure 25. La phase de déploiement définie par l'OMG.

2.4 Synthèse

L'OMG, via sa spécification D&C, définit le déploiement comme étant un processus constitué de cinq étapes. Ce processus commence une fois que le logiciel a été acquis et se termine par l'arrêt de l'exécution du logiciel (c'est-à-dire sa désactivation), tout en passant par le déplacement du logiciel dans un dépôt, le choix des implémentations à utiliser pour réaliser le logiciel, le choix de la distribution du logiciel, l'insertion des implémentations dans l'environnement et enfin, le lancement de l'exécution du logiciel (c'est-à-dire son activation). Les termes unité de déploiement, environnement cible, plan de déploiement ainsi que le rôle de déployeur sont aussi définis.

Cette définition comporte un élément surprenant. En effet, l'OMG donne une définition de l'installation très particulière. Cette définition n'a aucun rapport avec la définition habituellement retenue (comme, par exemple, dans les deux autres définitions du déploiement que nous présentons dans cette section). En effet, la définition consensuelle stipule qu'installer un logiciel consiste à le déplacer dans l'environnement cible, ainsi qu'à distribuer les artéfacts du logiciel de façon à rendre possible son exécution (c'est-à-dire permettre son activation). Pour l'OMG, l'installation consiste à acquérir un logiciel publié et à le rapatrier dans un dépôt logiciel sous le contrôle du déployeur.

L'OMG a aussi fusionné les rôles d'administrateur (dans le cadre du déploiement) et de déployeur sous son propre rôle de déployeur. Ainsi, pour l'OMG, c'est le déployeur qui acquiert le logiciel (il joue le rôle de *Repository Administrator*), qui précise la distribution des artéfacts du logiciel sur l'environnement cible (il joue alors le rôle de *Planner*) et qui décide de l'occurrence des étapes de déploiement (c'est alors l'*Executor*). Or, généralement, ces trois tâches incombent à l'administrateur, puisque c'est ce dernier qui est en charge de la gestion des licences des logiciels, de la gestion de l'environnement et puisqu'il commande les activités de déploiement qui doivent être mises en œuvre. De son côté, le déployeur est en charge de l'exécution des activités de déploiement commandées.

Ensuite, il apparaît clairement que l'exécution du logiciel est l'unique but du déploiement pour l'OMG. Typiquement, l'activité de maintenance, pouvant mettre en jeu, par exemple, des mises à jour statiques ou dynamiques, l'évolution ou le dynamisme n'est pas mentionnée. Et de manière plus surprenante encore, bien qu'elle considère l'étape de préparation, l'OMG ne considère pas la désinstallation (du logiciel).

D'une manière générale, la définition du déploiement proposée par l'OMG à une portée plus grande que celle de C. Szyperski, cela dit, la définition de l'OMG reste tout de même moins complète que celle de l'université du Colorado que nous allons présenter ci-après. Enfin, à l'image de la définition de C. Szyperski qui prend tout son sens dans le cadre de l'utilisation d'un logiciel de déploiement prenant en entrée une configuration définie par le déployeur, ici, la définition de l'OMG prend son sens dans le cadre d'une infrastructure qui doit pouvoir être remise à zéro, afin d'assurer la désinstallation du ou des logiciels déployés (comme dans une grille, par exemple).

2.5 Définition de l'Université du Colorado

Nous allons présenter maintenant la définition de la phase de déploiement proposée dans [HHC+98].

La notion de déploiement logiciel est présentée, comme la livraison, l'assemblage et la maintenance d'une version spécifique d'un système logiciel sur un site afin de le rendre utilisable. Les auteurs précisent que leur travail s'est focalisé sur l'identification et la caractérisation des activités minimales devant appartenir à la phase de déploiement. Bien que cette dernière paraisse aisément appréhendable, dans les faits, c'est une phase du génie logiciel à part entière et particulièrement complexe. Pour les auteurs, la phase de déploiement est un processus, qui contrairement aux propositions de C. Szyperski et de l'OMG n'est ni linéaire, ni séquentiel. Il est formé de huit activités distinctes et de plusieurs transitions, le tout formant une phase de déploiement minimale qui peut être considérée comme générique. Bien que ses activités soient fixées, la phase de déploiement proposée n'en conserve pas moins une certaine part de flexibilité qui provient de la nature du système logiciel à déployer, ainsi que des caractéristiques et exigences de ses producteurs et consommateurs. Ainsi, la phase de déploiement peut être customisée et enrichie en fonction des besoins et contraintes du logiciel, de ses producteurs et de ses consommateurs.

Le producteur développe, package et informe de la mise à disposition du logiciel. Le consommateur, quant à lui, fait référence à l'utilisateur final du logiciel. Le terme en lui-même est principalement employé pour désigner l'environnement cible (*Consumer Site*).

Nous allons maintenant nous focaliser sur chacune des huit activités qui sont proposées. Ces huit activités sont la mise à disposition (*Release*), l'installation (*Install*), l'activation (*Activate*), la désactivation (*DeActivate*), la mise à jour (*Update*), l'adaptation (*Adapt*), la désinstallation (*DeInstall*) et la fin de support (*DeRelease*). Il est intéressant de remarquer que cette phase de déploiement est plutôt symétrique, ainsi, *DeRelease*, *DeInstall* et *DeActivate* sont, respectivement, l'inverse de *Release*, *Install* et *Activate*.

La première activité est la mise à disposition. Cette activité fait l'interface entre le processus de développement du logiciel et le processus de déploiement du logiciel. Elle englobe toutes les actions nécessaires pour préparer un système logiciel afin qu'il puisse être correctement assemblé au niveau du site du consommateur. Le résultat de cette activité est un package qui contient les composants du système, ses dépendances et ses contraintes, ainsi que les informations nécessaires aux autres activités de déploiement. Enfin, l'activité de mise à disposition est aussi chargée d'informer les consommateurs de la mise à disposition du système.

La seconde activité est l'installation. Cette activité permet d'insérer un système sur le site d'un consommateur. C'est l'activité la plus complexe du processus de déploiement puisqu'elle doit trouver et assembler toutes les ressources nécessaires au système. Concrètement, l'installation d'un système commence par deux étapes. La première étape est l'interprétation des informations fournies par le package du système. La seconde étape est l'inspection du site cible afin de configurer correctement le système logiciel. Ces deux étapes sont suivies par deux sous-activités. La première concerne le transfert du système, du site du producteur vers celui du consommateur. La seconde sous-activité réalise, quant à elle, la configuration du système afin de le rendre activable (c'est-à-dire prêt à s'exécuter sur le site du consommateur). Enfin, l'installation est l'activité qui, en général, est la mieux supportée par les logiciels de déploiement.

La troisième activité est l'activation. Cette activité consiste à démarrer les éléments d'un système afin de rendre le système utilisable (c'est-à-dire afin de lancer l'exécution du système). Pour un système simple, l'activation peut se résumer à l'appel d'une méthode de l'environnement cible. A l'inverse, pour un système complexe, il peut être nécessaire d'activer un certain nombre d'éléments de l'environnement, tels des serveurs et des démons, avant de pouvoir activer le système en lui-même. Ainsi, l'activation d'un système peut être tout à fait complexe. Toutes les activités de déploiement peuvent nécessiter l'activation d'autres systèmes. Ainsi, si un système a été packagé dans un fichier archive, alors, son installation nécessite un système de décompression afin d'extraire le système à installer de son archive. De plus, si aucun système de décompression n'est disponible dans l'environnement, alors, il est nécessaire d'en installer un et de l'activer. Les auteurs utilisent le terme déploiement récursif (*recursive deployment*) pour désigner cette situation.

La quatrième activité est la désactivation. Cette activité est le pendant de l'activation. Le but de la désactivation est d'arrêter l'exécution d'un système, cet arrêt se faisant via l'arrêt des différents éléments du système. L'activité de désactivation est souvent requise avant que d'autres activités comme la mise à jour statique puissent commencer.

La cinquième activité est la mise à jour. Les auteurs présentent cette activité comme étant un cas particulier d'installation. Cependant, cette activité est moins complexe que l'installation, puisque la plupart des ressources nécessaires au système ont déjà été obtenues durant l'installation du système. Deux cas sont distingués pour la mise à jour. Le premier cas implique la désactivation du système, puis sa mise à jour et enfin sa réactivation. Le second cas n'implique pas la désactivation du système. Certains systèmes peuvent, en effet, être mis à jour tout en restant actifs. Enfin, tout comme l'installation, la mise à jour inclut aussi les sous-activités de transfert de packages et de configuration des éléments.

La sixième activité est l'adaptation. Comme la mise à jour, elle permet de modifier un système logiciel précédemment installé. Cependant, l'activité d'adaptation diffère de l'activité de mise à jour puisque, contrairement à cette dernière qui est initiée par des événements distants (par exemple, par une commande du consommateur), l'adaptation est, quant à elle, initiée par des événements locaux (par exemple, par un changement dans l'environnement dans lequel le système est déployé). Une adaptation peut, par exemple, être initiée afin de maintenir la correction d'un système logiciel.

L'avant dernière activité de la phase de déploiement est la désinstallation. Elle consiste à enlever un système d'un environnement donné. Pour qu'un système puisse être désinstallé, il faut qu'il ait été préalablement désactivé. L'activité de désinstallation n'est pas nécessairement triviale. En effet, lors de la désinstallation d'un système, il convient de porter une attention toute particulière aux ressources que le système en cours de désinstallation partage avec d'autres systèmes. Les dépendances d'autres systèmes vers le système ciblé par la désinstallation doivent, elles-aussi, être prises en compte lors de la désinstallation du système.

Enfin, la dernière activité est la fin de support. Au final, un système peut être marqué comme obsolète, ce faisant, le producteur arrête de le maintenir. L'activité de fin de support n'est pas similaire à l'activité de désinstallation puisque, dans le cas d'une fin de support, le système devient indisponible pour de futures installations, mais n'est pas désinstallé des sites sur lesquels il est déjà déployé. De plus, l'activité de fin de support est aussi en charge d'informer tous les consommateurs du système de la fin du support.

La figure ci-dessous représente la définition de la phase de déploiement de l'université du Colorado.

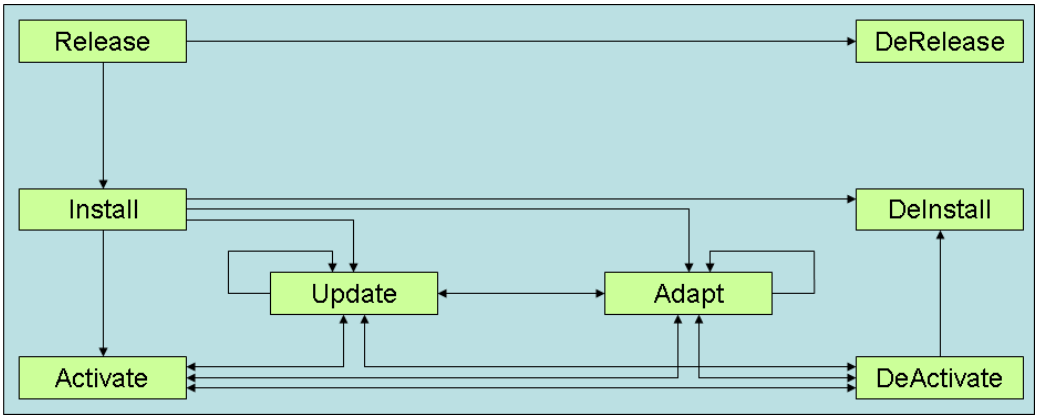


Figure 26. La phase de déploiement définie dans [HHC+98].

2.6 Synthèse

L'université du Colorado définit le déploiement comme étant un processus constitué de huit activités. Ce processus commence lors de la mise à disposition du système logiciel et se termine par la fin de son support, tout en passant par les activités d'installation, d'activation, de désactivation, de mise à jour, d'adaptation et de désinstallation. Un système mis à disposition est un système packagé. Dans cette définition, les auteurs introduisent les termes environnement (ou site), producteur, consommateur, cycle de vie du déploiement, la notion de ressources nécessaires à un système, et enfin, ils abordent le problème des dépendances entre systèmes.

Cette définition possède plusieurs points positifs. Tout d'abord, contrairement aux précédentes, elle appréhende le déploiement dans son ensemble, de façon cohérente et elle propose un cycle de vie générique pour le déploiement. A l'inverse, la définition de l'OMG ne fait qu'introduire et exécuter un logiciel dans un environnement et n'aborde ni sa maintenance, ni sa désinstallation. De son côté, la définition de C. Szyperski n'aborde ni l'installation, ni l'activation, ni la désactivation, ni la désinstallation du logiciel. Ensuite, il est intéressant de remarquer que contrairement à l'OMG pour qui le but du déploiement est l'exécution du logiciel, le déploiement est envisagé ici comme la livraison, l'assemblage et la maintenance d'une version spécifique d'un système logiciel dans un site. Ainsi, un logiciel peut être maintes fois installé, activé, désactivé, mise à jour, adapté, désinstallé.

Les auteurs insistent sur le fait qu'il faut prendre en compte les systèmes qui sont d'ores et déjà déployés dans le site cible. Par exemple, pour un système donné, il est tout à fait possible que son

installation requiert l'installation préalable d'autres systèmes. De même, lorsqu'un système est désinstallé, il est aussi nécessaire de prendre en compte la possible désinstallation des systèmes qu'il requérait. Qu'en est-il de la désactivation d'un système actuellement utilisé par d'autres, afin de pouvoir, par exemple, le mettre à jour (statiquement) ? Ainsi, les auteurs mettent en lumière le besoin d'une gestion du partage des systèmes nécessités par d'autres systèmes (c'est-à-dire, d'autres *packages*, d'autres unités de déploiement) tout au long de la phase de déploiement.

Voyons maintenant les limites de cette définition. Tout d'abord, l'introduction des rôles de producteur et consommateur n'apportent rien de nouveau par rapport aux rôles habituellement utilisés de développeur, d'administrateur et de déployeur. A notre avis, le rôle du consommateur ne fait que renforcer la confusion qui existe déjà entre les rôles d'administrateur et de déployeur.

Ensuite, les activités de mise à disposition (*Release*) et de fin de support (*DeRelease*) ne font pas partie de la phase de déploiement en elle-même. Dans le cas de la mise à disposition, il est évident que vouloir déployer un système qui n'a pas encore été publié n'a pas de sens. Cependant, même si le système est mis à disposition, si il n'a pas été acquis par l'administrateur de l'environnement d'exécution, alors, vouloir le déployer n'a pas non plus de sens. C'est pourquoi, il est plus logique de considérer la mise à disposition comme étant nécessaire avant toute acquisition par l'administrateur et l'acquisition comme étant une pré-condition aux consignes que l'administrateur donnera au niveau du déploiement. L'acquisition consiste, ici, en la récupération du package du système, ainsi que la possible licence correspondant à l'utilisation que l'administrateur souhaite faire du système. L'activité de fin de support est définie comme la fin du développement et de la publication de mises à jour pour un système logiciel. Tous les consommateurs du logiciel doivent être notifiés de la fin de support et un système qui n'est plus supporté ne peut plus être installé. Or cela est faux. Typiquement, si une entreprise possède une licence valide pour un logiciel qui n'est plus supporté, alors il est tout à fait possible que l'administrateur informatique de cette entreprise fasse installer ce logiciel sur des machines. Par exemple, il est possible d'installer le JDK v1.1 de Sun sur sa machine, alors que Sun ne le supporte plus.

Également, la présence de l'activité d'adaptation, dans cette définition du déploiement, renforce encore l'amalgame qui est fait entre les rôles d'administrateur et de déployeur. En effet, l'adaptation est définie comme étant la modification d'un système logiciel suite à un événement local à la différence d'une mise à jour qui est déclenchée par un événement distant. Cela signifie que le système logiciel et/ou l'environnement dans lequel le système est déployé peuvent modifier le système afin de l'adapter et cela sans que l'administrateur n'ait, *a priori*, à intervenir, sauf peut être en dernier recours. Or, l'activité d'adaptation relève, non pas de la phase de déploiement, mais de l'administration autonomique [KC03]. En effet, si le système et/ou l'environnement peuvent tenter, ou faire, des adaptations, ces adaptations sont des choix d'administration. Ces choix pourront, bien sûr, engendrer l'exécution d'un ensemble d'activités de déploiement ; mais les activités de déploiement ne seront que les conséquences des choix faits par l'acteur, l'entité, d'administration. En guise d'illustration simple, si un système communiquant au-dessus de TCP est adapté pour communiquer au-dessus d'UDP, alors cette adaptation peut donner lieu à la commande des activités de désactivation et de désinstallation du composant TCP du système, suivies par l'installation et l'activation d'un composant UDP au sein du même système. Cela dit, le déploiement en lui-même ne propose rien ni à propos du choix ou de la stratégie à suivre pour passer d'une communication au-dessus de TCP à une communication au-dessus d'UDP, ni à propos du diagnostic menant à cette adaptation.

Le cycle de vie du déploiement défini est générique. Cependant, il faut noter que l'unité de déploiement utilisée lors des différentes activités de déploiement est, soit le package du système et/ou le système lui-même (*Release*, *Install*, *Update*, *Adapt*, *DeInstall*, *DeRelease*), soit l'ensemble des composants du système (*Activate*, *DeActivate*). Ce changement d'unité de déploiement entre les activités de déploiement ne permet pas de dire si le cycle de vie défini vise un déploiement gros grain (c'est-à-dire où l'unité de déploiement est un système, un logiciel, une application, etc.) ou grain fin (c'est-à-dire où l'unité de déploiement est le composant, l'EJB, le service, etc.). Or, les contraintes que doit gérer le déploiement et les mécanismes qui doivent être mis en place dépendent du grain des unités de déploiement. Par exemple, le déploiement d'un logiciel via une unité de déploiement gros grain, pourra être réalisé de manière plutôt directe. Cependant, le déploiement du même logiciel via un ensemble d'unités de déploiement grain fin nécessitera, entre autres, l'ordonnancement des différentes unités de

déploiement lors de l'activation et de la désactivation du logiciel, ainsi que la gestion des unités de déploiement partagées et cela pour l'ensemble des activités de la phase de déploiement. De même, il sera nécessaire de gérer l'isolation entre les différentes unités de déploiement apparaissant dans l'environnement cible, etc. L'idée est, dans tous les cas, d'assurer que le logiciel déployé corresponde bien à ce qu'il serait si il était déployé seul ; cela revient à rendre la phase de déploiement la plus transparente possible.

L'activation et la désactivation d'un système sont faites au niveau des composants du système. Or, comme nous l'avons rapidement dit ci-dessus, l'activation et, respectivement, la désactivation d'un ensemble de composants nécessitent la mise en place d'un ordre sur les composants à activer et, respectivement, à désactiver. Cela est vrai, que ces composants soient gros grain ou grain fin. Ce problème se retrouve aussi, par exemple, dans les technologies OSGi et JEE EJB v3.0. mais aussi dans tous les systèmes distribués (où, par exemple, une base de donnée et un serveur, voire plus, doivent être activés avant que la partie client ne le soit, sous peine d'erreurs et/ou d'exceptions).

Enfin, nous avons vu que les auteurs mettent en avant la nécessité de se préoccuper du partage au niveau système / package / unité de déploiement lors de la phase de déploiement. Cependant, les auteurs ne proposent une prise en compte des éléments partagés qu'au niveau des activités d'installation et de désinstallation. Or il est aussi nécessaire de gérer les éléments partagés au niveau de l'activation et de la désactivation. De plus, la gestion de ce partage pour l'ensemble de la phase de déploiement n'est que la première étape pour assurer la correction du déploiement. La seconde étape est la gestion de l'isolation. Deux cas se distinguent alors : soit l'unité de déploiement est gros grain (et porte sur un système, une application, un logiciel, etc.), soit l'unité de déploiement a un grain fin (et porte sur un composant, un EJB, un service, etc.). Typiquement, il est évident qu'il est possible de décompresser simultanément deux archives en utilisant le même logiciel, or il est loin d'être évident de pouvoir utiliser le même composant simultanément dans deux logiciels différents. Les exemples les plus flagrants pour le niveau grain fin sont le composant de log, un composant qui compte le nombre de fois où il a été appelé, un composant qui conserve le contenu d'un panier virtuel.

Pour conclure, cette troisième définition du déploiement est celle qui, parmi les trois définitions du déploiement, a la plus grande portée. Elle est cohérente puisqu'elle englobe, à la fois, la mise à disposition et la fin de support, l'installation et la désinstallation, l'activation et la désactivation. C'est aussi la seule définition qui envisage le déploiement comme un cycle de vie complet, non linéaire et non séquentiel. Enfin, elle prend tout son sens dans le cadre du déploiement d'un ou plusieurs systèmes qui sont destinés à rester déployés dans un ou des environnements et qui peuvent être maintes fois activés, désactivés, mis à jour et adaptés.

2.7 Synthèse globale

Dans les sous-sections précédentes, nous avons présenté, détaillé et critiqué les trois définitions majeures de la notion de déploiement. En quelques mots, la définition de C. Szyperski illustre bien une définition minimaliste du déploiement. La définition de l'OMG donne une version plus cohérente et plus large, mais reste naïve. La définition de l'université du Colorado est la plus mature et la plus proche de la réalité. Cependant, sa partie maintenance demande à être approfondie et elle demande encore du travail dans le cadre de logiciels non monolithiques comme par exemple les applications à services. D'une manière générale, il n'existe pas, à l'heure actuelle, de définition fédératrice de la phase de déploiement. Les trois définitions que nous venons de voir mettent en avant le fait qu'il n'y a pas de définition claire des activités à inclure dans la phase de déploiement, pas plus que de consensus sur les acteurs (c'est-à-dire les rôles). Enfin, certaines activités elles-mêmes mériteraient d'être plus travaillées.

Le tableau ci-dessous propose une synthèse de ces trois définitions. Cette synthèse porte sur la définition de la phase de déploiement, l'unité de déploiement utilisée, le point d'entrée du déploiement et des activités qui le constituent, ainsi que la prise en compte du partage d'unités de déploiement, la gestion de ce partage et de l'isolation

	C. Szyperski	Object Management Group	Université du Colorado
Définition de la phase de déploiement.	**Processus linéaire et séquentiel :** *Configuration*.	**Processus linéaire et séquentiel :** *Installation*, *Configuration*, *Planification*, *Préparation*, *Launch*.	**Processus non linéaire et non séquentiel :** *Release*, *Install*, *Activate*, *DeActivate*, *Update*, *Adapt*, *DeActivate*, *DeInstall*, *DeRelease*.
Unité de déploiement	**Grain fin**: composant packagé.	**Gros grain**: logiciel packagé.	**Gros grain**: logiciel (toutes les activités mis à part *Activate* et *DeActivate*. **Grain fin**: composant (seulement pour *Activation*, *DeActivation*).
Point d'entrée de la phase de déploiement.	Composant packagé: **implémentation + modèle créé à partir de l'implémentation**.	Logiciel packagé: **implémentation + modèle créé à partir de l'implémentation**.	Logiciel packagé: **implémentation + modèle créé à partir de l'implémentation**.
Point(s) d'entrée des activités.	**composant packagé.**	**Dépend de l'activité**: *Software package*, *Configuration*, *Software*, *Deployment plan*, *Application*.	**logiciel packagé.**
Partage envisagé ?	**non.**	**non.**	**oui, mais** uniquement au niveau de logiciels entiers.
Gestion du partage ?	**non.**	**non.**	**oui, mis à part** pour les activités *Activate* et *DeActivate*.
Gestion de l'isolation ?	**non.**	**non.**	**non.**

Tableau 2. Synthèse concernant les définitions de la phase de déploiement présentées.

Nous verrons dans le prochain chapitre, que le premier travail de cette thèse a porté sur une mise à plat concernant la définition du déploiement logiciel. Cette mise à plat s'est traduite par la distinction stricte des rôles d'administrateur et de déployeur, par la définition d'un cycle de vie de la phase de déploiement logiciel qui soit cohérente, ainsi que par la proposition de l'utilisation d'un modèle (non créé à partir d'une ou d'un ensemble d'implémentations) comme point d'entrée du déploiement et enfin par l'uniformisation des points d'entrée des activités de déploiement. Nous verrons aussi, dans les sections suivantes, que la gestion du partage d'implémentations de services et d'instances d'implémentations de services entre applications à services et que la gestion de l'isolation au niveau des applications à services a fait partie de nos préoccupations. Notre but étant, bien évidemment, le déploiement des applications à services mais nous avons aussi souhaité rendre la phase de déploiement aussi transparente que possible, tout en garantissant sa correction.

3. CARACTÉRISATION DU DÉPLOIEMENT

3.1 Points de caractérisation

Les deux sections précédentes donnent un point de vue général sur la notion de déploiement, sans contexte ou hypothèses particuliers. Nous allons maintenant présenter un état de l'art portant sur le déploiement logiciel dans les solutions actuelles.

Nous considérons que les défis liés au déploiement dépendent de trois points, à savoir :

- la définition du déploiement retenue,
- l'unité de déploiement utilisée
- et les propriétés qui doivent être assurées (par le déploiement).

Par exemple, pour l'université du Colorado (et *Software Dock*), le déploiement est un cycle de vie qui comprend l'installation, l'activation, la désactivation, la désinstallation, etc. Les unités de déploiement manipulées sont des logiciels packagés (c'est-à-dire des unités de gros grain, des monolithes). Les propriétés à assurer sont la résolution des dépendances des unités de déploiement lors de leur installation et la correction du partage de ces mêmes unités de déploiement lors de leur installation et désinstallation. En d'autres termes, *Software Dock* gère les unités de déploiement qui sont nécessaires à l'activation d'une unité de déploiement (par contre, le niveau instance d'unité de déploiement n'est pas géré). Les défis sont donc le déploiement d'une unité de déploiement (c'est-à-dire d'un logiciel), en tant que tel, ainsi que la gestion des dépendances et la gestion du partage d'unités de déploiement.

Dans cet état de l'art, nous avons identifié huit points discriminants qui sont, à notre avis, les points fondamentaux pour caractériser le déploiement de logiciels centralisés. Ces points sont :

- la définition de la phase de déploiement,
- l'unité de développement,
- l'unité de déploiement,
- le point d'entrée de la phase de déploiement,
- la résolution de dépendances et la validation,
- le partage d'implémentation et sa gestion,
- le partage d'instance d'implémentation et sa gestion
- et la gestion de l'isolation.

Il est à noter que le déploiement de logiciels distribués, répartis fait apparaitre d'autres points discriminants comme, par exemple, le type de répartition supporté, la communication inter-environnements, l'homogénéité des environnements, l'homogénéité des versions d'implémentations (entre environnements).

Maintenant, dans le cadre de cette thèse, nous proposons et utilisons notre propre définition du déploiement qui reprend et étend la définition de l'université du Colorado (le chapitre contribution détaillera à la fois les éléments repris et les éléments étendus). Notre but est de compléter la définition qui nous semblait la plus pertinente pour satisfaire les besoins de notre étude. Les unités de déploiement que nous manipulons sont des implémentations de services. Nous avons cherché à automatiser le déploiement de modèles d'application à services dans des environnements d'exécution à services, tout en assurant l'isolation entre les modèles effectivement déployés (en effet, les environnements d'exécution que nous ciblons permettent le partage aussi bien d'implémentations de services que d'instances d'implémentations de services, mais ne le gèrent pas).

Cet état de l'art est composé de six cas d'études et est organisé en trois domaines. Le premier domaine cible les prototypes de "Recherche". Le cas d'étude correspondant est appelé *Software Dock* [HHW99]. Le second domaine concerne le déploiement pour les systèmes à composants. Les cas d'études sont CCM v4.0, EJB v3.0 et .Net [OMG06a] [SUN06] [.Net06]. Le dernier domaine cible le déploiement

dans l'approche à service. Nous nous focaliserons alors sur la technologie OSGi R4.1 et la spécification SCA V1.00 [OSGi07] [SCA07].

Pour remarque, concernant le dernier domaine, nous ne présentons pas la spécification du W3C et les services web. En effet, les services web se focalisent avant tout sur le contrat, l'interface du service et le W3C ne fait aucune proposition concernant le déploiement à proprement parlé.

3.2 Application à *Software Dock*

Software Dock (SD) est un prototype de recherche conçu pour le déploiement de logiciels monolithiques (il ne propose pas de modèle de programmation et n'est pas non plus lié à un quelconque modèle de programmation). Il a été développé par Richard S. Hall [HHC+98]. SD a été finalisé en 1999 et a donné lieu à une publication dans la conférence *International Conference of Software Engineering* cette même année [HHW99].

La phase de déploiement appréhendée par SD correspond à la phase de déploiement de l'université du Colorado comme présentée précédemment dans cet état de l'art. Pour rappel, cette phase comprend huit activités, à savoir, la *Release*, la *DeRelease*, l'*Installation*, l'*Activation*, l'*Update*, l'*Adapt*, la *DeActivation* et la *DeInstallation*. Les auteurs de SD utilisent aussi parfois les termes *Retire* pour désigner l'activité *DeRelease* ; *Remove* pour se référer à l'activité de *DeInstallation* et enfin *ReConfigure* et *Adapt* pour faire référence à l'activité *Adapt* définie dans [HHC+98].

SD appréhende les logiciels packagés sous la forme d'une unique unité de déploiement (c'est pourquoi nous parlons de logiciel monolithique). Cependant, bien que ces logiciels soient monolithiques, ils peuvent néanmoins exprimer des requis. Ces requis portent sur le fait que d'autres logiciels devront être déployés dans l'environnement cible et cela afin de permettre, par la suite, l'activation du logiciel en cours de déploiement. Le concepteur de SD précise que le nombre total de requis exprimés par un logiciel est généralement inférieur à dix. En guise d'exemple, le déploiement d'un conteneur EJB requiert un logiciel de compression et de décompression de fichiers, ce second logiciel requérant lui-même le logiciel *Windows Installer*. SD prend aussi en compte les logiciels qui sont d'ores et déjà déployés dans l'environnement cible. L'idée est de ne pas déployer un logiciel apparaissant déjà dans l'environnement cible. SD est donc sensible au contexte.

Le point d'entrée de la phase de déploiement est l'unité de déploiement que nous avons présenté ci-dessus. Le déployeur doit créer et renseigner via SD un descripteur correspondant à l'unité de déploiement. Ce descripteur est nommé *Deployable Software Description* (DSD). Tout DSD contient cinq parties distinctes. Ces parties sont nommées *Configuration*, *Assertion*, *Dependency*, *Artifact* et enfin *Activity*. La première partie se focalise sur l'expression de la configuration de l'unité de déploiement, ainsi que sur les ressources qu'elle fournit. La seconde partie définit des propriétés que l'environnement cible doit satisfaire, comme le type de système d'exploitation, la ou les machines virtuelles étant mises à disposition dans l'environnement. La troisième partie, nommée *Dependency*, regroupe les informations concernant des requis que SD devra satisfaire afin de réaliser le déploiement de l'unité de déploiement. Ces requis peuvent, par exemple, porter sur un logiciel nécessaire à l'activation de l'unité de déploiement ou sur la configuration d'une machine virtuelle. Ensuite, la quatrième partie décrit les éléments physiques que contient l'unité de déploiement, comme la liste des fichiers d'aide valides pour tel système d'exploitation ou telle langue, l'ensemble des fichiers et archives qu'elle contient. La cinquième et dernière partie, nommée *Activity*, décrit toute activité de déploiement non couverte par SD et spécifique à l'unité de déploiement. Néanmoins, les auteurs ne précisent pas le contenu de cette cinquième partie, ni comment son contenu est traité par SD. Enfin, il est très important de noter que le contenu des cinq parties que nous venons de présenter est exprimé sous la forme de propriétés (c'est-à-dire sous la forme de couples composés d'une clé et d'une valeur) [HHW99].
Au final, le couple constitué d'une unité de déploiement et d'un DSD est le point d'entrée de toutes les activités de déploiement.

Au niveau de la résolution des dépendances, nous avons vu précédemment que les unités de déploiement sont associées à une DSD et que cette DSD permet d'exprimer, entre autres choses, les

ressources fournies par l'unité de déploiement ainsi que les ressources dont elle a besoin. Ainsi, lors de l'installation d'un logiciel, SD va résoudre le ou les requis du logiciel en favorisant l'utilisation des logiciels d'ores et déjà déployés dans l'environnement cible (sensibilité au contexte).

En ce qui concerne la validation des dépendances entre unités de déploiement formant une application, elle n'est pas abordée par SD, puisque ce dernier manipule des logiciels packagés de façon monolithique.

SD permet le partage d'un logiciel (monolithique) entre d'autres logiciels monolithiques. Pour être plus précis, il est possible que plusieurs logiciels utilisent un même autre logiciel (par exemple, Jonas et Eclipse peuvent tous deux utiliser le même logiciel de compression et de décompression de fichiers). Par contre, SD n'aborde pas le partage d'implémentations à grain fin entre logiciels gros grain (c'est-à-dire le partage d'implémentations internes à un logiciel). Les logiciels ciblés sont monolithiques et, par conséquent, les éléments qui les composent sont inaccessibles en tant que tels. De même, SD n'aborde pas non plus le partage d'instances d'implémentations entre logiciels, typiquement, dans l'exemple du début de ce paragraphe, c'est le système d'exploitation qui décidera d'attribuer la même instance ou des instances distinctes du logiciel de compression (et décompression de fichiers) aux deux logiciels qui l'utilisent.

Nous avons vu que SD autorise le partage de l'utilisation d'un logiciel entre plusieurs logiciels. SD offre aussi un mécanisme de gestion de ce partage. Cela se traduit par le fait que SD ne permet pas la désinstallation d'un logiciel qui est encore nécessité par au moins un autre logiciel du même environnement cible. Cela est tout à fait logique, puisque désinstaller le premier logiciel empêcherait, corromprait le fonctionnement de tout logiciel le nécessitant. Il est important de noter que ce type de gestionnaire n'est réalisable que si le grain des unités de développement est le même que celui des unités de déploiement. Or, par exemple, des technologies comme OSGi ou EJB nous montrent que cela est loin d'être toujours le cas.

Enfin, SD n'aborde pas la gestion des activations et désactivations. Typiquement si un logiciel nécessite qu'un autre logiciel soit activé (afin de pouvoir lui-même fonctionner correctement), alors, c'est au premier logiciel d'activer cet autre logiciel avant d'exécuter sa propre activation. L'absence d'un tel mécanisme d'ordonnancement dans SD est surprenante, sachant que ce besoin est explicitement mis en lumière dans [HHC+98].

Pour finir, il faut noter que SD ne fait aucune proposition autour de la gestion de l'isolation. Pour rappel, nous avons vu que c'est le système d'exploitation qui a la charge du niveau instance d'implémentation (c'est-à-dire instance de logiciel).

	Software Dock
Définition de la phase de déploiement	Processus non linéaire et non séquentiel, ***Release***, ***Installation***, ***Activation***, ***DeActivation***, ***Update***, ***Adapt***, ***DeInstallation***, ***DeRelease***.
Unité de développement	*Software Dock* n'est pas lié à un modèle de programmation, il ne se préoccupe pas d'une quelconque unité de développement.
Unité de déploiement	***Package*** est une unité gros grain qui contient un logiciel entier. Ainsi, du point de vue du déploiement, les logiciels packagés sont des monolithes.
Point d'entrée de la phase de déploiement	**Un *Package*, (il contient l'implémentation du logiciel et des métadonnées** (DSD) **définies** via *Software Dock* **à partir de l'implémentation).**
Résolution de dépendances et validation	*Software Dock* propose **un mécanisme automatique de résolution des dépendances d'un *Package***, en prenant en compte les *Packages* d'ores et déjà déployés dans l'environnement cible. Le problème de la validation des dépendances entre unités de déploiement formant une application n'est pas pertinente ici.
Partage d'implémentation, Gestion de ce partage.	**Oui**, en effet, il est possible que plusieurs logiciels partagent l'utilisation d'un même logiciel. La possibilité de partage concerne un partage d'unités gros grain, aucun partage n'est permis au niveau *composant* (grain fin).
Partage d'instance d'implémentation, Gestion de ce partage.	**Non**, la problématique du partage d'instance n'est pas prise en compte par SD, la gestion du niveau instance est laissée au système d'exploitation de l'environnement cible.
Gestion de l'isolation	**Non.**
Remarque	Les termes *Retire* et, respectivement, *Remove* sont parfois utilisés par les auteurs pour désigner les activités de déploiement *DeRelease* et, respectivement, *DeInstallation*. De même, les termes *ReConfiguration* et *Adapt* sont eux aussi utilisés pour désigner l'activité *Adapt* définie par A. Carzaniga.

Tableau 3. Synthèse sur les capacités de déploiement de Software Dock.

4. LE DÉPLOIEMENT DE COMPOSANTS

4.1 Corba Component Model V4.0 (CCM)

Tout d'abord, il est intéressant de noter que les définitions de la phase de déploiement du modèle à composant CORBA (CCM) et de la spécification *Deployment&Configuration* (D&C) sont naturellement liées. En effet, ces spécifications sont, toutes deux, produites par l'OMG. La spécification CCM version 3 est, du reste, antérieure à la première version de la spécification D&C. La spécification CCM fut donc la première spécification de l'OMG à introduire un chapitre traitant du packaging et du déploiement de composants et d'assemblages de composants. Ce chapitre fut ensuite extrait de la spécification CCM pour donner lieu à une spécification en elle-même, à savoir, la spécification D&C de l'OMG. Finalement, la spécification CCM version 4 a construit son propre chapitre relatif au déploiement en appliquant la spécification D&C à son modèle de composant. Cet état de l'art se focalise sur cette dernière version de la spécification CCM (c'est-à-dire sur la version 4) [OMG06a].

Comme pour toutes les technologies présentées dans le cadre de cet état de l'art sur le déploiement, nous allons commencer par présenter la définition de la phase de déploiement retenue par la spécification CCM, ainsi que l'unité de développement, l'unité de déploiement, le point d'entrée de la phase de déploiement, l'existence et les possibilités de résolution de dépendances, de validation, de partage d'implémentation et d'instance, ainsi que les possibilités de gestion du partage et de l'isolation aux niveaux implémentation et instance.

Comme indiqué, la spécification CCM s'appuie sur la définition de la spécification D&C. De ce fait, la phase de déploiement CCM contient les activités d'installation, de configuration, de planification, de préparation et de lancement de l'exécution de l'unité de déploiement. Rappelons que, pour l'OMG, l'installation consiste à insérer une unité de déploiement dans un dépôt (et non dans l'environnement d'exécution comme cela est généralement entendu). Le lancement de l'exécution, quant à lui, consiste à activer l'unité de déploiement, mais il inclut aussi le fait que l'unité de déploiement puisse se désactiver elle-même (par exemple, lorsque sa tâche est finie) ou le fait qu'elle puisse être désactivée par l'infrastructure à l'origine de son activation. Outre le fait qu'elle se base sur la spécification D&C, la spécification CCM étend la définition de la phase de déploiement en permettant la désinstallation d'unités de déploiement du dépôt qui contient les unités de déploiement. Cependant, ni la spécification CCM, ni la spécification D&C n'appréhende l'activité de suppression d'une unité de déploiement de l'environnement dans lequel elle est déployée (c'est-à-dire l'activité de désinstallation telle quelle est entendue généralement).

Concernant l'unité de développement, il faut noter que la spécification CCM propose un modèle à composant construit au-dessus de Corba. Le but suivi est l'extension du modèle d'objet distribué Corba, via l'introduction d'un modèle à composant lui aussi distribué. Dans le modèle à composant proposé par CCM, un composant est l'entité logicielle de développement (figure ci-dessous). Ce composant a un ensemble d'attributs et un ensemble d'interfaces (il est question de *ports*). Chaque port possède un ensemble d'opérations et peut être utilisé par d'autres composants ou utiliser d'autres composants (il s'agit, respectivement, de ports fournis et de ports requis par le composant). La spécification CCM précise aussi qu'un ensemble de composants, interconnectés ou non, est appelé un *assemblage*. Enfin, le modèle à composant proposé par la spécification CCM n'est pas récursif. En effet, un assemblage n'est pas appréhendé comme un composant.

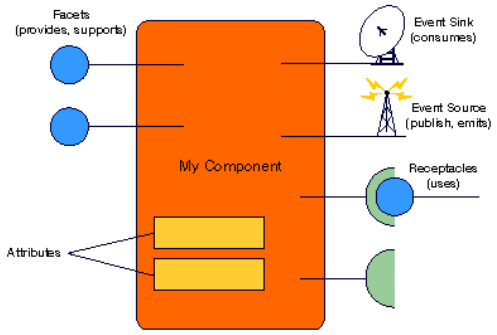


Figure 27. Le modèle à composant CORBA (CCM) [IBM01].

La spécification CCM définit une unité de déploiement nommée *Component Package*. Un *Component Package* contient soit un seul composant monolithique, soit un assemblage de plusieurs composants (*Assembly*). Dans tous les cas, c'est un fichier archive (.car) contenant :

- une description du composant via quatre fichiers (*Software Package Descriptor*, *Corba Component Descriptor*, *Component Assembly Descriptor* et *Property File Descriptor*),
- une ou plusieurs implémentations par composant (selon, par exemple, l'OS cible, la VM cible)
- et un fichier de configuration contenant l'état initial des propriétés des composants du *Component Package*.

Un *Component Package* peut aussi spécifier des ressources qui devront être présentes dans l'environnement où il va être déployé, au moment de son activation. Enfin, il faut noter que chaque composant appartenant à un *Component Package* possède sa propre fabrique[1]. Cette fabrique est nommée génériquement *Component Home* ou *Home*. Elle est en charge de la création, de l'activation, de la désactivation et de la destruction des instances du composant. Enfin, il est intéressant de remarquer que la spécification CCM définit, d'un côté, une unité de développement à grain fin et d'un autre côté, une unité de déploiement qui est gros grain, permettant ainsi un déploiement à grain fin (c'est-à-dire, où chaque composant est une unité de déploiement), mais aussi un déploiement gros grain (c'est-à-dire, où un assemblage de composants est packagé dans une unique unité de déploiement).

Les points d'entrée selon l'activité de déploiement sont les mêmes pour la spécification D&C de l'OMG et la spécification CCM. Ainsi, l'installation et la configuration prennent en entrée l'unité de déploiement définie, puis le planning, quant à lui, utilise une configuration et l'unité de déploiement correspondante, ensuite la préparation applique le plan de déploiement défini par le planning et enfin, l'activité *launch* est lancée en utilisant l'identifiant du logiciel correspondant à l'unité de déploiement.

En ce qui concerne la résolution de dépendances, la spécification CCM ne décrit ni ne propose de mécanisme particulier.

Au niveau de la validation des dépendances entre unités de déploiement formant une application, elle n'est pas abordée par CCM, pas plus que la validation de la description des *Component Package* (bien que la grammaire de l'*Interface Description Language* (IDL) servant à les écrire soit définie).

La spécification CCM ne permet pas le partage d'implémentations de composant entre assemblages de composants et cela, que ces derniers soient ou non dans le même *Component Package*. De même, le partage d'instance d'implémentation de composant entre composants n'est pas non plus

[1] La fabrique (*factory*) est un patron de conception créationnel utilisé en programmation orientée objet. Comme les autres modèles créationnels, la fabrique a pour rôle l'instanciation d'objets dont le type n'est pas prédéfini : les objets sont créés dynamiquement en fonction des paramètres passés à la fabrique.

autorisé. En conséquence, la spécification CCM ne fait aucune proposition autour d'un quelconque gestionnaire de partage. D'ailleurs, il est important de noter que, dans le monde CCM, tout composant CCM est rattaché à un *Component Home* qui est en charge de gérer les instances des implémentations. L'absence de partage, aussi bien au niveau implémentation de composant qu'au niveau instance d'implémentation de composant, permet à la spécification CCM d'assurer par défaut l'isolation entre les composants d'un même *Component Package*. Et cela sans que la mise en place d'un gestionnaire d'isolation soit nécessaire.

Le tableau ci-dessous présente la synthèse des différents points que nous avons abordés dans cette sous-section sur la spécification CCM V4.0.

	CCM V4.0
Définition de la phase de déploiement	Processus linéaire et séquentiel, ***Installation***, ***Configuration***, ***Planification***, ***Preparation***, ***Launch***, ***DeInstallation***. Cependant, les activités de déploiement *Installation* et *DeInstallation* correspondent à l'insertion et à la suppression d'une unité de déploiement dans le dépôt d'unités de déploiement.
Unité de développement	**Composant** et **assemblage de composants** (grain fin).
Unité de déploiement	***Component Package*** (gros grain) contenant, d'une part, soit un composant monolithique, soit un assemblage de composants et, d'autre part, un ensemble de fichiers, quatre décrivant le *Component Package* et un contenant sa configuration.
Point d'entrée de la phase de déploiement	***Component Package*** (une ou plusieurs implémentations des composants + des fichiers décrivant le composant ou l'assemblage de composants).
Résolution de dépendances et validation	**Les dépendances des composants doivent être gérées à la main** et décrites dans les métadonnées du *Component Package*. La validation des dépendances entre Component Packages formant une application n'est pas abordée par CCM.
Partage d'implémentation, Gestion de ce partage.	**Non.** **Un tel gestionnaire n'est pas nécessaire.**
Partage d'instance d'implémentation, Gestion de ce partage.	**Non.** **Un tel gestionnaire n'est pas nécessaire.**
Gestion de l'isolation	**Non, un tel gestionnaire n'est pas nécessaire, chaque composant ayant sa propre instance** attribuée par le "Component Home" auquel il est rattaché.
Remarque	CCM n'utilise pas le même point d'entrée pour toutes ses activités (*Component Package*, *Configuration*, *Deployment Plan*, *Software*). Cette démarche est assez inhabituelle, en comparaison aux technologies OSGi, .Net, EJB et aux démarches de recherche telle Software Dock.

Tableau 4. Synthèse sur les capacités de déploiement de CCM.

4.2 Enterprise Java Beans (EJB 3.0)

Depuis les années 2000, l'entreprise Sun Microsystems conçoit et publie la spécification *Enterprise Java Bean* (EJB). Cette spécification définit un standard pour la conception et le développement de *beans*. Les *beans* sont les briques de bases pour la programmation d'applications destinées à être déployées du côté serveur. La première spécification EJB (version 1.1) a été publiée en 1999 [SUN99]. Ensuite, la seconde (version 2.1) et la troisième (version 3.0) ont été publiées respectivement en 2003 et 2006 [SUN03] [SUN06]. Dans cet état de l'art, nous nous focalisons sur la version 3.0 qui est la dernière version en date.

La spécification *Enterprise Java Bean* (EJB) a été conçue en vue de simplifier la gestion des applications. Il faut noter que la gestion comprend, entre autres, la phase de déploiement. Pour la spécification EJB, la phase de déploiement est seulement constituée par les activités d'installation et d'activation. Cependant, elle définit aussi une phase d'*un-deployment* qui est constituée des activités de désactivation et de désinstallation. Ainsi, au final, la spécification EJB doit être considérée comme couvrant l'ensemble des activités d'installation, d'activation, de désactivation et de désinstallation. Elle ne couvre néanmoins pas la totalité de la phase de déploiement puisqu'elle n'apporte aucune précision ni sur les mises à jour statiques ou dynamiques, ni sur l'évolution et le dynamisme. La spécification EJB précise ensuite que les *beans* sont destinés à être déployés et exécutés à l'intérieur de conteneurs EJB, aussi appelés serveurs d'applications. Les principaux conteneurs EJB existants aujourd'hui sont Jonas, JBoss, Apache Geronimo, Oracle Application Server 10g et IBM WebSphere [Jon08] [Red07] [ASF08] [Ora07] [IBM08a]. Ils servent à abstraire l'environnement d'accueil et offrent une grande variété de propriétés fonctionnelles et non-fonctionnelles de haut niveau, comme la sécurité, la gestion du cycle de vie, la gestion des transactions, ainsi que plusieurs propriétés de bas niveau, telles la gestion de cache, la gestion de la mémoire et des instances.

Le concept de *bean* a été conçu pour simplifier le développement d'applications. Deux aspects sont à distinguer ici. Premièrement, l'idée des EJBs est de changer la granularité de programmation, en passant d'un modèle de programmation objet à un modèle de programmation à composant EJB (c'est-à-dire à *bean*). Tout *bean* est implémenté par au moins un objet Java (c'est son corps) et doit offrir et implémenter une interface @Remote et/ou une interface @Local. L'interface @Remote spécifie la signature des méthodes pouvant être appelées par des tiers se situant à l'extérieur du conteneur dans lequel se trouve le *bean* (typiquement, un client lourd distant). L'interface @Local, quant à elle, spécifie la signature des méthodes pouvant être appelées par des tiers se situant dans le même conteneur (appartenant, ou non, à la même application). Cependant, si un *bean* n'est pas destiné à être appelé à distance, alors, il peut ne pas avoir d'interface @Remote. De la même manière, si il n'est pas destiné à être appelé en local, l'interface @Local n'est pas nécessaire. De plus, tout *bean* ayant une dépendance vers un autre *bean* doit utiliser un registre (JNDI) afin de trouver le *bean* satisfaisant sa dépendance. Ainsi, tout *bean* peut s'enregistrer, rechercher et se retirer du registre. Enfin, il faut noter que ce registre est la propriété du conteneur EJB et qu'il n'existe qu'un unique registre par conteneur. Le second aspect simplifiant le développement d'applications porte sur le fait que les *beans* sont des entités logicielles spécialisées. Ainsi, d'un point de vue plus théorique, un *bean* peut être considéré comme une entité logicielle réalisant une seule interface dans laquelle toutes les méthodes sont décorées avec les annotations @Local et/ou @Remote. De plus, une dépendance n'est plus exprimée envers un *bean*, mais vers un couple composé d'une interface et d'une classe JAVA. Ainsi, le fait que les *beans* ne fournissent qu'une unique interface les spécialise. A l'inverse, les composants en général fournissent eux plusieurs interfaces, ces interfaces pouvant avoir des interactions non exprimées entre elles, ce qui peut, de fait, engendrer des erreurs, par exemple, lors de la réutilisation d'un composant, lors de sa substitution par un autre composant ou lors de l'utilisation ou de la réutilisation d'un ensemble partiel des interfaces qu'il fournit. Cette amélioration des possibilités de réutilisation des *beans* par rapport aux composants, contribue à simplifier le développement des applications (une partie de l'application peut être le fruit de la réutilisation de *beans*). Cette approche permet d'économiser les étapes de conception, de développement et de test correspondant aux *beans* réutilisés. La spécification EJB engage d'ailleurs à la réutilisation de mêmes *beans* dans différents contextes, dans différentes applications.

Nous avons vu précédemment que la spécification EJB définit l'installation, l'activation, la désactivation et la désinstallation de *beans*. Cependant, contrairement à ce qui pouvait être attendu, le *bean* n'est pas l'unité de déploiement qui a été retenue. En effet, le déploiement de *beans* ou d'applications basées sur des *beans* est défini comme devant se faire via un fichier archive (un .jar dans le cas d'un ensemble composé uniquement de *beans* et .ear dans le cas d'applications contenant à la fois des *beans*, mais aussi une partie IHM web). Ces archives peuvent contenir un fichier XML jouant le rôle de descripteur de déploiement. Ce descripteur contient typiquement les informations sur les requis de chaque *bean* en termes de transactions, de sécurité et de persistance, mais pas en terme de *beans* (!). Enfin, il faut noter que le format de l'archive et le contenu du descripteur de déploiement sont les éléments principaux du standard défini par la spécification EJB pour les unités de déploiement. Ce standard est détaillé par les chapitres 19 et 20 de la spécification *Core* EJB v3.0 [SUN06a].

Il est très important de remarquer que si une unité de déploiement EJB contient plusieurs *beans* alors, certes, ces *beans* vont avoir un comportement individuel et différent à l'exécution (*Stateless* versus *Statefull* versus *Entity*). Néanmoins, ils ne pourront être déployés qu'au travers de leur unité de déploiement, c'est-à-dire sous la forme d'un monolithe. Or cette contrainte restreint énormément les capacités de déploiement de telles applications. En effet, le fait qu'une application soit packagée sous la forme d'une unique unité de déploiement implique qu'elle ne pourra être déployée que d'un seul tenant. Ainsi, l'application sera installée, activée, désactivée, mise à jour, désinstallée d'un seul bloc. De plus, il ne peut être réellement question ni d'évolution, ni de dynamisme de l'application, puisque tout changement (au niveau de l'architecture) de l'application se traduit par un changement sur le monolithe (formé par l'unité de déploiement) et implique donc, au moins, la désinstallation du monolithe et l'installation du "nouveau" monolithe. Or si cette application est constituée de *beans* packagés individuellement, alors, certaines activités de déploiement au niveau de l'application peuvent ne se traduire qu'en des activités de déploiement au niveau d'un sous-ensemble des *beans* de l'application. Par exemple, une mise à jour de l'application portant sur un sous-ensemble de ses *beans* peut se traduire par leur désinstallation et l'installation des *beans* mis à jour. De même, un changement de l'architecture de l'application lorsque cette dernière est activée (c'est l'activité de dynamisme) ou lorsqu'elle ne l'est pas (c'est l'activité d'évolution) n'implique plus une désinstallation et une installation totale, seuls les *beans* concernés par l'activité seront effectivement touchés par le déploiement.

En complément, à l'heure actuelle, il est possible d'observer une transition au niveau des conteneurs EJB eux-mêmes. En effet, ces derniers étaient jusqu'à présent packagés sous la forme d'une unique unité de déploiement. Ainsi, tout changement au niveau du conteneur implique sa désactivation, sa désinstallation, sa "ré"-installation et sa "ré"-activation. Or la désactivation d'un conteneur peut être réellement couteuse suivant le contexte industriel dans lequel le conteneur évolue. Ainsi, par exemple, les entreprises Bull avec le conteneur JONAS, IBM avec le conteneur WebSphere, Red Hat avec JBOSS et ORACLE avec Oracle Application Server 10g sont en train ou ont modularisé leur conteneur, afin de pouvoir tirer parti des capacités de déploiement propres aux applications composées d'ensembles d'unités de déploiement et afin de permettre à leurs clients d'économiser de l'argent (l'application composée d'ensembles d'unités de déploiement étant, ici, le conteneur lui-même). Les problématiques liées à la modularisation des conteneurs EJB et une solution peuvent être trouvées dans [Des07].

Nous venons de présenter l'unité de déploiement standardisée définie par la spécification EJB. Elle est le point d'entrée de la phase de déploiement. Contrairement à la spécification *Deployment & Configuration* de l'OMG (par exemple), cette unité de déploiement est aussi le point d'entrée de toutes les activités de déploiement. Nous allons maintenant détailler le cycle de vie d'un *bean* (v3.0) et, par là même, le fonctionnement du conteneur EJB. Le cycle de vie d'un *bean*, tel qu'il est décrit dans la spécification, est constitué de quatre phases, à savoir, le développement, le déploiement, l'exécution (appelé aussi disponibilité) et le retrait (*un-deployment*). Dans la première phase, le *bean* est écrit puis packagé sous la forme d'une archive Java standardisée. Dans la seconde phase, l'assembleur de l'application ajuste les propriétés du possible descripteur de déploiement. Ces propriétés peuvent concerner la sécurité, les mécanismes de persistance et de transaction, ainsi que toute autre propriété intervenant dans le déploiement. Cette seconde phase se termine par l'installation de l'archive Java dans un répertoire du conteneur EJB. La troisième phase commence lorsque le conteneur active l'unité de déploiement, afin de pouvoir lancer l'exécution des *beans* qu'elle contient. Typiquement, pour le conteneur EJB v3.0 de JONAS (appelé EASYBEANS [OW208]), l'activation d'une unité de déploiement

est automatiquement lancée dès que cette dernière est placée dans un répertoire spécifique du conteneur (c'est-à-dire initialement, dans le répertoire *deploy*, puis dans les dernières version, dans le répertoire *ejb3s*). Cette troisième phase se poursuit en suivant quatre étapes de plus bas niveau. Ces quatre étapes sont *construct*, *activate*, *passivate* et *destruct*. Elles concernent, respectivement, la construction des instances des *beans*, leur activation, leur passivation (c'est-à-dire leur désactivation) et enfin la destruction des instances. La quatrième et dernière phase est la phase de retrait de l'unité de déploiement. Concrètement, elle consiste à enlever l'unité de déploiement du répertoire du conteneur dans lequel elle a été déposée. De cette manière, le conteneur, qui a une copie "personnelle" de l'unité de déploiement (*deep copy*), s'aperçoit que l'unité de déploiement doit être *un-deployed*. Il procède donc à la passivation puis à la destruction de toutes les instances des *beans* qui appartenaient à l'unité de déploiement, enfin, il supprime sa copie "personnelle" de l'unité de déploiement.

La spécification EJB ne fait aucune proposition au niveau de la résolution de dépendances. Typiquement, si un *bean* requiert un autre *bean* et si cet autre *bean* n'est pas présent (et n'est pas actif), alors, dans le cas général, le premier *bean* ne pourra pas fonctionner. Cependant, si celui-ci a été conçu pour rattraper la non satisfaction d'une de ses dépendances, alors, il est possible qu'il puisse tout de même fonctionner, mais d'une manière vraisemblablement partielle (voire incorrecte). Comme nous venons de le préciser, la spécification EJB ne gère pas la résolution de dépendances. Cependant, elle spécifie comment les *beans* se trouvent. Ainsi, les conteneurs EJB contiennent un registre (*Java Naming and Directory Interface*, JNDI). Ce registre permet l'enregistrement, la découverte et le désenregistrement des *beans*. Ainsi, ces derniers utilisent ce registre pour rechercher et trouver les *beans* qu'ils requièrent. La spécification EJB précise aussi le fonctionnement de ce registre. Les informations utilisées comme identifiant lors de l'enregistrement d'un *bean* sont le nom canonique Java du corps du *bean* et soit le nom canonique de son interface *Remote* augmentée de l'annotation @Remote, soit le nom canonique de son interface *Local* augmentée de l'annotation @Local. Il est important de noter qu'un *bean* peut être enregistré deux fois (c'est-à-dire, pour un accès local et pour un accès distant). Ce type d'identifiant permet ainsi un fonctionnement de type *naming* du registre, c'est-à-dire un fonctionnement basé sur des entités clairement identifiées (par opposition à un fonctionnement de type *trading* qui lui est basé sur une approche "libérale"). Grâce au fonctionnement *naming* du registre, tout *bean* sait avec quels autres *beans* il interagit, il connait le "nom" des *beans* qu'il requiert. Ce type de fonctionnement permet d'assurer que l'application développée est bien celle qui s'exécute. Le fonctionnement *trading*, basé par exemple sur des interfaces, ne permet pas d'offrir une telle garantie.

Nous avons vu précédemment, que les unités de déploiement EJB sont standardisées par la spécification EJB. Cette dernière précise que les descripteurs de déploiement des unités de déploiement doivent être validés avant toute installation. Néanmoins, cette validation ne porte que sur le respect du schéma XML des descripteurs de déploiement. La validation des dépendances entre *beans* packagés dans une unique unité de déploiement ou dans plusieurs unités de déploiement et formant une application n'est, quant à elle, pas abordée par la spécification EJB.

Un *bean* peut être utilisé par plusieurs autres *beans*. Cela revient à permettre le partage des *beans* entre plusieurs ensembles de *beans* définis ou non dans la même unité de déploiement (c'est-à-dire dans un ensemble de .jar versus dans une application .ear ou dans un ensemble d'applications .ear). Le partage d'implémentation est donc possible dans le monde EJB. Au niveau du partage d'instances d'implémentation (c'est-à-dire d'entités créées à partir d'une implémentation), trois cas sont à distinguer. Ces cas correspondent aux trois types de *beans*, à savoir, le type *Stateless*, le type *Statefull* et le type *Entity*.

Dans le cas du type *Stateless*, lorsqu'un client récupère une référence vers un tel *bean*, alors, dès qu'il va faire un appel de méthode sur ce *bean*, le conteneur va lui passer une instance nettoyée du *bean* cible. En effet, pour les *beans Stateless*, le conteneur EJB gère un *pool* qui contient des instances à usage unique, où toute instance ayant été utilisée (une fois) est nettoyée, puis remise en attente dans le *pool*. Ainsi, le partage d'instance d'implémentation n'est pas possible sur les *beans* de type *Stateless*. Pour les *beans* de type *Statefull*, lorsqu'un client récupère une référence vers un tel *bean*, alors, dès qu'il va faire un appel de méthode sur ce *bean*, le conteneur va toujours lui passer la même instance du *bean* cible (sauf, bien sur, si le client demande une nouvelle référence). Donc, puisque tout client doit initialement demander une référence du *bean* qu'il cible, il est impossible de partager des instances d'implémentation

de *beans Statefull*. Enfin, le troisième et dernier cas concerne les *beans* de type *Entity*. Un *bean Entity* représente et permet d'interagir avec une table dans une base de données. Pour être tout à fait précis, les attributs d'un tel *bean* représentent les colonnes de la table. Une référence qui pointe vers une instance de *bean Entity* représente, quant à elle, une ligne de la table. Au niveau du fonctionnement des *beans Entity*, le conteneur EJB les traite de la même manière que les *beans Statefull* au niveau de la gestion des instances. Ainsi, il est tout à fait possible que deux clients utilisent dans le même temps la même ligne dans une table et qu'ils aient tous deux la même référence (pointant vers la même instance de *bean Entity*) [SUN07]. Il est donc possible de partager des instances de *beans Entity*. Pour information, l'accès aux références des *beans Entity* est géré par l'*Entity Manager*. Ce dernier est en charge de la gestion des instances des *beans Entity* et de la persistance des valeurs associées aux instances dans la base de données (via la *Java Persistency API*).

En ce qui concerne la gestion du partage d'implémentations (c'est-à-dire des *beans* eux-mêmes), la spécification ne fait aucune proposition. Typiquement, lors de son installation, un *bean* peut écraser la référence d'un autre *bean* déjà présent dans le conteneur. Ainsi, si un ensemble composé des *beans* A, B et C (avec A utilisant B et B utilisant C) est déployé et si un second ensemble composé des *beans* D, B' et E (avec D utilisant B' et B' utilisant E) est ensuite déployé et si B et B' ont le même identifiant (c'est-à-dire les mêmes noms canoniques d'interface et de corps) alors le premier ensemble de *beans* passera des interactions (A, B) et (B, C) aux interactions (A, B') et (B', E). Ainsi, l'exécution d'une activité de déploiement dans un conteneur EJB peut perturber les *beans* déjà déployés dans ce même conteneur. Toujours au niveau de la gestion du partage d'implémentations, si une application utilise des *beans* d'une autre application et si cette dernière vient à être désactivée, voire désinstallée, alors la première application se trouvera avec des requis non satisfaits, ce qui aura pour conséquence d'empêcher ou de corrompre son fonctionnement.

La spécification EJB ne fait aucune proposition à propos de la gestion de l'isolation. Typiquement, si deux ensembles de *beans* interagissent (involontairement) avec le même *bean Entity*, alors les données de ces deux ensembles seront mélangées dans la même table. En conséquence, le contenu de la table sera vraisemblablement corrompu (par exemple, des données supplémentaires pourront être présentes, des données non-attendues, des écrasements de données). Ce qui fait que l'isolation des instances partagées de *beans Entity* n'est pas gérée.

Le tableau ci-dessous présente la synthèse des différents points que nous avons abordés dans cette sous-section sur la spécification EJB v3.0.

	EJB v3.0
Définition de la phase de déploiement	Processus linéaire et séquentiel, ***Installation***, ***Activation***, ***DeActivation***, ***DeInstallation***.
Unité de développement	**Enterprise Java Bean** (grain fin)
Unité de déploiement	**Archive** contenant soit un ou plusieurs *bean*s (dans un .jar), soit un ensemble de *beans* et une IHM web (dans un .ear). Dans le cas général, une archive est une unité de déploiement **gros grain**, cependant, si une archive est un .jar qui ne contient qu'un *bean*, alors elle doit être considérée comme étant de **grain fin**.
Point d'entrée de la phase de déploiement	**Archive** (implémentation des *beans* + un descripteur de déploiement optionnel décrivant les *beans* + une IHM web optionnelle).
Résolution de dépendances et validation	**Aucun mécanisme de résolution automatique de dépendances.** **Pas de mécanisme de validation** des dépendances entre *beans* formant une application.
Partage d'implémentation, Gestion de ce partage.	**Oui**, l'utilisation d'**un *bean* peut être partagée entre plusieurs *beans*.** **Non, aucun gestionnaire.**
Partage d'instance d'implémentation, Gestion de ce partage.	**Oui et non, le partage** de l'utilisation d'une même instance **n'est possible que pour l'instance unique créée pour un *Entity Bean***. C'est l'*Entity Manager* qui joue le rôle de gestionnaire.
Gestion de l'isolation	**Non**. En effet, deux *beans* partageant l'utilisation d'une même *bean Entity* peuvent écrire dans la même table (sur des lignes différentes ou sur la même ligne), ce qui peut mener à la corruption de la table.
Remarque	Tout *bean* possède une interface @Local et/ou une interface @Remote. Or d'un point de vue plus théorique, un *bean* peut être vu comme une entité logicielle ne possédant qu'une unique interface, dont chaque méthode serait annotée par un tag @Local et/ou @Remote, selon qu'elle soit accessible localement et/ou à distance.

Tableau 5. Synthèse sur les capacités de déploiement d'EJB V3.0.

4.3 Microsoft .Net Framework

Nous allons maintenant nous intéresser à la version 3.0 du *framework* .Net proposé par Microsoft [.Net06]. Le *framework* .Net version 3.0 a été publié en novembre 2006. Cette version 3.0 fait suite aux versions 1.1 et 2.0 ayant, toutes deux, été respectivement publiées en avril 2003 et janvier 2006. L'entreprise Microsoft présente elle-même son *framework* comme étant une plateforme de programmation orientée objet et non composant [.Net07]. Ce framework supporte, bien évidemment, une phase de déploiement.

La phase de déploiement retenue par le *framework* .Net est constituée par les activités de déploiement suivantes : l'installation, la maintenance et la désinstallation. Les activités d'activation et de désactivation sont supportées par le *framework*, mais ne sont pas présentées comme faisant partie du déploiement. Il faut noter que l'activité maintenance définie par le *framework* .Net contient les sous-activités de mise à jour statique et de réparation d'applications. Il est intéressant de noter que la mise à jour statique d'une application peut se faire sans contrainte concernant le fait que cette dernière soit activée ou désactivée. Cependant, si l'application est activée, alors, la mise à jour ne sera effective qu'une fois l'application désactivée puis (ré-) activée. C'est pour cela que cette mise à jour est considérée comme statique. Enfin, une installation peut être annulée, afin de restaurer l'état du *framework* .Net et du système d'exploitation (aux installations d'autres applications et aux modifications extérieures à l'application près).

Microsoft .Net Framework propose trois outils pour réaliser effectivement les activités de déploiement retenues. Ces outils sont *Windows Installer*, *ClickOnce* et *Windows Update*. Tout d'abord, *Windows Installer* est l'outil de déploiement en standard. Il couvre l'installation, la maintenance et la désinstallation d'applications. Le terme maintenance est utilisé par *Microsoft .Net Framework* pour désigner les sous-activités de mise à jour statique et de réparation qui sont proposées. L'outil *ClickOnce* est destiné aux administrateurs. Il couvre les activités d'installation, de mise à jour et de désinstallation. L'activité de mise à jour propose pas moins de sept stratégies différentes : *Checking for Updates After Application Startup*, *Checking for Updates Before Application Startup*, *Making Updates Required*, *Specifying Update Intervals*, *Providing a User Interface for Updates*, *Blocking Update Checking*, *Permission Elevation and Updates*. De plus, *ClickOnce* déploie uniquement des applications totalement isolées. En effet, ces applications se suffisent en elles-mêmes et n'ont aucune dépendance, ni aucun requis (mis à part la présence du système d'exploitation Windows et du *framework* .Net). Le troisième et dernier outil, appelé *Windows Update*, se focalise sur la sous-activité de mise à jour statique. Il propose aussi plusieurs stratégies, telles que la notification de l'existence d'une mise à jour, le couple comprenant le téléchargement et la notification de la mise à jour, ou encore le triplet comprenant le téléchargement, l'exécution et la notification de la mise à jour. Enfin, il est aussi possible de préprogrammer *Windows Update* afin qu'il fonctionne de manière autonome à une date précise. Dans *Windows Update*, toute notification est, suivant le contexte, soit une demande de confirmation ou de choix, soit une simple note informative. Enfin, *Windows Installer* permet la désinstallation des mises à jour installées.

Nous avons vu précédemment que le *framework* .Net est une plateforme de programmation orientée objet, ce qui signifie que l'objet est l'unité de développement dans le monde .Net. Il convient cependant d'être plus explicite concernant le développement et en particulier la compilation en .Net. Le *framework* .Net définit un langage commun de spécification (*Common Language Specification*, noté CLS). Tout langage de programmation respectant ce CLS est un langage .Net. Le *framework* .Net définit aussi un langage intermédiaire (*Intermediate Language* ou *Common Intermediate Language* ou encore *MicroSoft Intermediate Language*, notés respectivement IL, CIL et MSIL). Ce langage intermédiaire permet de compiler une application développée via un langage respectant le CLS, afin de la rendre exécutable dans tout *framework* .Net. Par défaut, Microsoft fournit dans Visual Studio .Net les compilateurs pour VB, C++, C#, J# et JScript. Cependant, des compilateurs viables, voire industriels, existent pour Cobol, Pascal, ML, etc. Enfin, l'application exprimée en IL sera à nouveau compilée sous la forme d'un code natif soit lors de son installation, soit lors de son activation. Le code natif ainsi produit étant spécifique au framework .Net de la machine cible, ainsi qu'à son environnement.

Nous venons de voir que l'objet est l'unité de développement en .Net. Nous allons maintenant nous intéresser à l'unité de déploiement. En .Net, l'unité de déploiement est l'*assembly* (c'est aussi l'unité de versionnement). Chaque *assembly* est identifié de manière unique via son nom. Le nom d'un *assembly* est un nom fort (*strong name*), il est constitué d'un nom simple, d'un numéro de version en quatre parties, d'informations concernant la culture (c'est-à-dire la langue, la région, le fuseau horaire, le système d'unités, la monnaie), d'une clé publique, ainsi que d'une signature digitale. Outre le fait d'assurer l'unicité de l'identifiant de tout *assembly*, l'utilisation de ce nom fort permet d'assurer qu'un *assembly* n'est pas corrompu et qu'aucune nouvelle version de cet *assembly* ne peut être produite, à moins de posséder, au préalable, le code source et la clé privée de l'*assembly* (qui deviendra alors l'*assembly* parent). Concrètement, un *assembly* est soit un fichier .exe, soit un fichier .dll. Tout *assembly* contient un manifeste, des méta-données concernant les types utilisés, le code MSIL et des ressources. Le manifeste décrit comment les éléments de l'*assembly* interagissent les uns avec les autres, il décrit aussi la portée de l'*assembly*, les ressources ainsi que les classes requises et les *assemblies* requis par l'*assembly*. Le manifeste peut être stocké avec l'*assembly* (c'est-à-dire dans le fichier .exe ou dans le fichier .dll) ; il peut aussi être stocké dans un fichier propre sous la forme d'un *Portable Executable* (PE). Les méta-données de type définissent les types contenus dans l'*assembly*. Le code MSIL implémente les types précédemment définis. C'est le code MSIL qui est le résultat des compilateurs des langages respectant CLS et c'est aussi le point d'entrée du compilateur qui va produire le code natif. Il est à noter que le *framework* .Net définit le *Common Language Runtime* (CLR). Ainsi, en plus d'exécuter le code natif, le CLR embarque le compilateur chargé de compiler du code MSIL en code natif. Ce dernier compilateur est nommé *Just-In-Time* compilateur (JIT). Enfin, les ressources définies par l'*assembly* pointent sur des ressources présentes, soit dans l'*assembly*, soit hors de l'*assembly* (par exemple des fichiers graphiques) [Dan05].

Un *assembly* peut être considéré à la fois comme une unité de déploiement gros grain et grain fin. En effet, il est tout à fait possible de packager une application entière dans un unique *assembly*, ce qui donne une unité de déploiement gros grain ; mais il est aussi tout à fait possible de packager une application via plusieurs *assemblies*, l'*assembly* devenant alors une unité de déploiement grain fin. Dans le cas où une application est packagée en utilisant plusieurs *assemblies*, c'est le fichier *Portable Executable* qui servira à décrire l'application, les *assemblies* qu'elle contient, ainsi que les interactions entre ces *assemblies*. De plus, ce fichier PE devra obligatoirement être un fichier à part. Il est important de noter que tel ou tel packaging va influencer les capacités de déploiement de l'application packagée. Ainsi, si des capacités de déploiement particulières sont attendues pour une application, il conviendra d'apporter un soin particulier à son packaging.

Enfin, les applications déployées avec *ClickOnce* possèdent deux manifestes. Le premier est nommé manifeste d'application. Il décrit l'application en elle-même, ses *assemblies*, ses dépendances, ses fichiers, les autorisations requises et l'emplacement dans lequel les mises à jour sont/seront disponibles (c'est un fichier PE séparé des *assemblies*). Le second manifeste est nommé manifeste de déploiement. Il décrit comment l'application est déployée, y compris l'emplacement du manifeste d'application et la version de l'application que les clients doivent exécuter [Tem04].

```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly xmlns="urn:schemas-microsoft-com:asm.v2" xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"
xmlns:asmv2="urn:schemas-microsoft-com:asm.v2" xmlns:dsig="http://www.w3.org/2000/09/xmldsig#"
manifestVersion="1.0" >
  <asmv1:assemblyIdentity name="Sample.application" version="1.0.0.0" publicKeyToken="3cb8aba78866dbbf"
language="neutral" processorArchitecture="msil" />
  <asmv1:description asmv2:publisher="Microsoft" asmv2:product="Sample" > This is a sample </asmv1:description>
  <deployment install="true">
    <subscription> <update> <beforeApplicationStartup /> </update> </subscription>
    <deploymentProvider codebase="http://myserver/Sample.application" />
  </deployment>
  <dependency>
    <dependentAssembly codebase="Sample_1.0.0.0\Sample.exe.manifest" size="3422">
      <assemblyIdentity name="Sample.exe" version="1.0.0.0" publicKeyToken="3cb8aba78866dbbf" language="neutral"
processorArchitecture="msil" />
      <hash>…</hash>
    </dependentAssembly>
  </dependency>
  <dsig:Signature Id="StrongNameSignature">…</dsig:Signature>
</asmv1:assembly>
```

Figure 28. Exemple de manifeste de déploiement pour *ClickOnce*.

Pour finir, il est important de noter que .Net distingue deux types d'*assembly*. Le premier type est nommé *private assembly*. Ce type d'*assembly* doit être utilisé si l'*assembly* est destiné à être utilisé au sein d'une et une seule application. Ce type d'*assembly* doit être déployé dans un répertoire et non dans le *Global Assembly Cache* (GAC). Une application basée uniquement sur ce type d'*assembly* est, ainsi, totalement isolée des autres applications. Il est aussi possible qu'une application n'ait seulement qu'un ou quelques *private assemblies*, alors seuls ces *private assemblies* seront isolés. Utiliser ce type d'*assembly* permet d'éviter les conflits liés à l'existence de plusieurs versions de la même DLL déployées via plusieurs applications, ce qui peut corrompre le fonctionnement d'une ou plusieurs applications. Cette situation est connue sous le terme d'enfer des DLLs. Les conséquences de cette situation étaient aggravées par le fait, qu'alors, l'exécution de DLLs de même identifiant, mais de versions différentes n'était pas autorisée (c'est-à-dire pas d'exécution *side-by-side*). Le second type est nommé *shared assembly*. L'idée est ici de permettre l'utilisation du même *assembly* par plusieurs applications. .Net fait trois propositions autour des *shared assemblies*. Ainsi, un *shared assembly* peut être déployé plusieurs fois (une copie pour chaque application le nécessitant). Il est aussi possible de déployer une seule fois le *shared assembly* dans un répertoire, les applications le nécessitant devront alors pointer directement vers ce répertoire. Enfin, il est possible de déployer le *shared assembly* directement dans le GAC, les applications le requérant se verront alors redirigées, par le CLR, vers l'instance (unique) du *shared assembly* créée via le GAC.

Nous venons de voir que l'*assembly* est l'unité de déploiement du monde .Net. Nous avons aussi vu indirectement que les points d'entrées du déploiement sont, soit un *assembly* incluant un fichier PE (*Portable Executable*), soit un couple contenant un ou plusieurs *assemblies* et un fichier PE externe aux *assemblies*.

Dans le *framework* .Net, la résolution des dépendances est réalisée lors de l'activation. Nous avons vu précédemment que tout *assembly* peut requérir des fonctionnalités (via des références) et peut aussi préciser quels *assemblies* doivent satisfaire ses requis. Ces deux types d'informations étant décrits dans le fichier PE associé à l'*assembly* ou à l'ensemble d'*assemblies* formant l'application. Ainsi, la résolution de dépendances proposée par .Net porte sur la résolution de références d'*assembly* et sur la localisation de ces *assemblies*. C'est le CLR qui est en charge de cette résolution. Cette résolution est composée de quatre étapes qui sont enchainées séquentiellement. Nous allons maintenant décrire ces quatre étapes [Mic08a]. Le processus de résolution et de localisation d'un *assembly* débute lorsque le CLR tente de résoudre une référence vers un autre *assembly*. Cette référence peut être statique ou dynamique. Le compilateur (langage de programmation vers MSIL) enregistre les références statiques dans les méta-données du manifeste d'*assembly* au moment de la génération. Les références dynamiques étant, quant à elles, construites à la volée en fonction des méthodes appelées au niveau de l'*assembly*, lors de son exécution. Il est important de noter que la meilleure façon de référencer un *assembly* est d'utiliser une

référence complète, comprenant le nom de l'*assembly*, sa version, sa culture et sa clé publique. Lors de l'activation, le CRL utilise ces informations pour résoudre une référence d'*assembly* et localiser l'*assembly* trouvé. Les quatre étapes de cette résolution sont :

1. Le CLR détermine la version correcte de l'*assembly*.
2. Il vérifie si le nom de l'*assembly* a déjà été lié (lors d'une précédente activation), si tel est le cas, c'est l'*assembly* précédemment chargé qui est utilisé.
3. Il vérifie le contenu du GAC. Si l'*assembly* s'y trouve, il est utilisé.
4. Et enfin, il tente de localiser l'*assembly* en procédant comme suit :
    - Si la configuration et la stratégie de l'éditeur n'affectent pas la référence d'origine et que la demande de liaison a été créée via une méthode particulière du *framework* .Net, alors le CLR vérifie la présence d'indications relatives à l'emplacement.
    - Si une base de code (c'est-à-dire un dépôt d'*assemblies*) est trouvée dans les fichiers de configuration, le CLR vérifie uniquement cet emplacement. Si cette tentative de localisation échoue, le CLR détermine que la demande de liaison a échoué et aucune autre tentative de localisation ne se produit.
    - Enfin, il tente de localiser l'*assembly* en utilisant les paramètres de localisation fournis. Si l'*assembly* n'est pas trouvé suite à cette tentative, alors le CLR demande à Windows Installer de fournir l'*assembly* (c'est une fonctionnalité d'installation à la demande).

Ensuite, la validation des dépendances entre *assemblies* formant une application n'est pas abordée par .Net et cela, bien que .Net permette la définition d'un fichier PE décrivant l'application.

Nous allons maintenant voir les possibilités qu'offre le *framework* .Net en termes de partage d'implémentation et d'instances d'implémentations entre applications. Ceci revient à présenter les possibilités de partage offertes, respectivement, aux niveaux *assembly* et instance d'*assembly*. Le *framework* .Net permet aussi bien le partage d'un *assembly* entre plusieurs applications, que le partage d'une instance d'*assembly* entre plusieurs applications. Pour ce faire (c'est-à-dire dans ces deux cas), l'*assembly* doit être du type *shared assembly*. Il doit aussi être déployé soit directement dans un répertoire, soit directement dans le GAC. Dans ces deux cas, c'est le CLR qui charge l'*assembly* et qui en crée une unique instance qui est partagée entre les différentes applications. Bien que les mécanismes de création d'instances soient similaires dans les deux cas ci-dessus, déployer un *assembly* dans un répertoire et le déployer dans le GAC ne leur donnent pas la même priorité lors de la résolution des dépendances. De plus, lorsqu'un *assembly* est déployé dans un répertoire et que plusieurs applications pointent sur cet *assembly* (dans ce répertoire), alors il n'est pas possible d'exécuter différentes versions de cet *assembly* côte à côte. Par contre, cela est possible si l'*assembly* est déployé dans le GAC. Or, l'exécution de différentes versions de la même .dll côte à côte (*side-by-side execution*) permet d'éviter les conflits de versions (c'est-à-dire permet d'éviter l'enfer des DLLs). Ensuite, le fait de n'avoir qu'un seul *assembly* déployé dans un répertoire évite aussi les problèmes de contrôle de versions, puisqu'une seule version de l'*assembly* est déployée. D'un autre côté, déployer un *assembly* dans le GAC permet d'avoir un point central pour gérer les *assemblies*, ce qui facilite le déploiement. Enfin, .Net annonce de meilleures performances lors du chargement d'*assembly* si ce dernier est déployé dans le GAC plutôt que dans un répertoire.

Au niveau de la gestion du partage d'*assembly*, le *framework* .Net propose un gestionnaire de partage d'*assembly*. Ce gestionnaire peut être mis en œuvre, soit en utilisant "gacutil.exe", mais dans ce cas, c'est au déployeur de préciser si l'application en cours de déploiement doit être notée comme nécessitant tel ou tel *assembly* ou non (via les paramètres "/ir" et "/ur"), soit en utilisant *Windows Installer* qui est cette fois un mécanisme automatique. L'information stockée afin de gérer le partage des *assemblies* est tout à fait rudimentaire, elle consiste en un compteur dont la valeur correspond au nombre de références faites sur l'*assembly*. Au niveau de la gestion du partage des instances d'*assemblies*, seule l'unique instance créée pour un *shared assembly* peut être partagée et c'est le CLR qui prend en charge le partage des instances de *shared assemblies*. Ainsi, tant qu'une instance d'*assembly* active possède une référence vers une instance d'un *shared assembly*, alors l'instance du *shared assembly* ne sera pas détruite (ce *shared assembly* restera actif).

Nous venons de voir que le *framework* .Net propose un gestionnaire de partage d'*assembly* et d'instances d'*assembly* entre plusieurs applications. Maintenant, au niveau de la gestion de l'isolation, le *framework* .Net ne propose aucun gestionnaire d'isolation entre les applications utilisant un même *assembly*. Ainsi, si le déployeur souhaite qu'une application soit isolée ou que certains des *assemblies* de cette application soit isolés, le *framework* .Net conseille de déployer les *assemblies* cibles comme des *private assemblies*, ce qui revient à déployer les *assemblies* directement dans un répertoire (le déploiement via le GAC est donc à bannir si un *assembly* doit être isolé). De plus, le fait que ces *assemblies* soient des *private assemblies* préviendra le fait que d'autres applications puissent venir pointer sur elles (c'est-à-dire les utiliser). Enfin, et pour être tout à fait explicite, si plusieurs applications partagent un même *assembly*, alors aucune isolation n'est envisageable.

Enfin, en .Net, le déploiement d'un *assembly* peut se faire soit via le GAC, soit par une simple copie dans un répertoire. Ensuite, grâce aux informations de sécurité et aux méta-données, le CLR prend en charge le chargement du bon *assembly*. Il sait même identifier les *assemblies* référencés par un *assembly* et les télécharger à la demande, ce qui permet de proposer des applications de faible taille, dont les *assemblies* sont chargés depuis un réseau aussi bien intranet, qu'internet, selon les besoins.

Le tableau ci-dessous présente la synthèse des différents points que nous avons abordés dans cette sous-section sur .Net V3.0.

	.Net V3.0
Définition de la phase de déploiement	Processus non linéaire et non séquentiel, **Installation**, **Activation**, **Désactivation**, **Mise à jour statique**, **Réparation**, **Désinstallation**. Outils à disposition : *Windows Installer*, *Click Once*, *Windows Update*. L'activation et la désactivation sont pris en charge directement par le *framework* .Net.
Unité de développement	**Objet** (grain fin), ses caractéristiques exactes dépendent du langage de programmation (VB, C++, C#, J#, ...), le code étant ensuite compilé en *Intermediate Language* (IL ou CIL ou MSIL).
Unité de déploiement	L'***Assembly*** qui peut contenir une application entière (**gros grain**) ou une partie (**grain fin**). Tout *assembly* doit contenir un **fichier *Portable Executable*** (PE) le décrivant. Le déploiement d'un **ensemble d'*assemblies*** associé à un unique fichier PE externe est possible ; ce dernier décrit les *assemblies*, leurs dépendances via des références vers d'autres *assemblies*, etc.
Point d'entrée de la phase de déploiement	Soit un ***Assembly*** qui contient du code IL et un fichier PE le décrivant, soit un **ensemble d'*Assemblies*** où chaque *Assembly* contient du code IL et un seul fichier PE externe pour l'ensemble. Dans tous les cas, **implémentation(s) + un descripteur créé à partir des *Assemblies*.**
Résolution de dépendances et validation	**Aucun mécanisme de résolution automatique de dépendances**, les *assemblies* destinés à satisfaire les dépendances d'un *assembly* sont spécifiés via le fichier PE de l'*assembly*. Cependant, les *assemblies* peuvent être déployés dans un répertoire, un dépôt ou dans le *Global Assembly Cache*. .Net propose donc des mécanismes pour sélectionner et localiser les *assemblies* correspondant au fichier PE. La validation n'est pas abordée.
Partage d'implémentation, Gestion de ce partage.	**Oui, mais** seulement pour les ***shared assemblies***, les *private assemblies* appartenant, quant à eux, à une unique application. **Par défaut, la gestion du partage d'*assemblies* n'est pas gérée, cependant,** lors de la phase de déploiement, il est possible d'utiliser ***Windows Installer*** ou **l'outil "gacutils.exe"** afin de mettre en place un compteur chargé de recenser le nombre d'utilisateur, *assembly* par *assembly*.
Partage d'instance d'implémentation, Gestion de ce partage.	Oui, mais **uniquement pour l'unique instance créée par *shared assembly*.** Le partage d'instances est géré automatiquement par le CLR.
Gestion de l'isolation	**Non**, aucun mécanisme n'est proposé pour gérer l'isolation lors du partage d'*assembly* ou d'instance d'*assembly*.
Remarque	.Net permet l'exécution d'*Assemblies side-by-side*. *Visual Studio .Net* couvre les phases de développement et déploiement.

Tableau 6. Synthèse sur les capacités de déploiement de .Net V3.0.

5. LE DÉPLOIEMENT DE SERVICES

Dans cette section, nous allons présenter deux cas d'études concernant le déploiement dans l'approche à service. Ces deux cas d'études sont OSGi R4.1 et SCA V1.00. Nous ne présentons pas les services web et la spécification du W3C. En effet, les services web se focalisent avant tout sur le contrat, l'interface du service et le W3C ne fait aucune proposition concernant le déploiement à proprement parler.

5.1 OSGi R4.1

Nous avons présenté, précédemment, l'Alliance OSGi et la spécification OSGi R4.1 du point de vue "service". Nous avons aussi présenté l'unité de développement du monde OSGi, à savoir le composant orienté service. Pour rappel, tout *bundle* OSGi peut avoir deux niveaux de dépendances. Le premier niveau de dépendance concerne les dépendances de code via les notions de package requis et fourni. Ces dépendances sont gérées automatiquement par l'environnement d'exécution OSGi. Le second niveau, quant à lui, est le niveau service où des services (c'est-à-dire des interfaces Java dans le monde OSGi) peuvent être fournis ou requis. Les dépendances de ce second niveau peuvent être gérées soit par l'environnement d'exécution OSGi, soit par le *bundle* lui-même (c'est-à-dire par le développeur), soit par un intervenant extérieur, typiquement, le déployeur.

Précédemment, nous avons aussi présenté la phase de déploiement retenue et couverte par la plateforme OSGi, à savoir l'installation, l'activation, la mise à jour statique, la désactivation et la désinstallation (de *bundle*). Ici, il est important de rappeler que l'environnement d'exécution OSGi est un environnement dynamique, c'est-à-dire qu'il est possible d'installer, d'activer, de désactiver et de désinstaller des *bundles* dans cet environnement sans pour autant qu'il soit nécessaire de désactiver puis de réactiver l'environnement. Il est aussi important de rappeler que la spécification OSGi définit un *bundle* comme étant une application extensible. Cela définit par là même un patron architectural d'application extensible. Concrètement, dans le monde OSGi, une application extensible est constituée d'une entité logicielle "cœur" qui peut être étendue et utilisée par d'autres entités logicielles. Ainsi, du point de vue du déploiement :

- l'entité logicielle "cœur" peut être active sans que les autres entités logicielles soient elles-mêmes actives (ces autres entités logicielles sont qualifiées d'entités extensives),
- si l'entité logicielle "cœur" est active et si une ou plusieurs autres entités logicielles deviennent actives à leur tour, alors l'entité logicielle "cœur" pourra se lier aux entités logicielles nouvellement activées, étendant par la-même l'entité "cœur" (d'où le terme d'application extensible)
- et troisièmement, si une entité logicielle à laquelle s'est liée l'entité "cœur" se retrouve désactivée, alors l'entité "cœur" peut rester active, mais elle perdra l'extension qu'offrait l'entité désactivée.

Ainsi, dans le cas du patron d'application extensible défini par OSGi, il est correct de désigner ces applications comme étant dynamiques. Cependant, le patron promu par OSGi ne gère pas l'état des entités extensives, ni la propagation ou la conservation de ces états, ni les interactions pouvant exister entre l'entité "cœur" et les entités extensives, ni l'état de ces interactions (en particulier, qu'advient il lorsqu'une entité extensive est désactivée pendant que l'entité "cœur" l'utilise ?). Or gérer ces différents problèmes et informations est nécessaire afin de garantir la correction de l'application, ainsi que la transparence des activations et désactivations des entités logicielles extensives. Au final, ce patron est, certes, un premier pas important vers les applications dynamiques, mais il demeure encore incomplet. De plus, il ne décrit qu'un cas particulier d'application qui est loin de couvrir l'ensemble (des patrons) des applications qui existent actuellement.

Nous avons déjà décrit l'unité de déploiement de l'environnement OSGi, à savoir le *bundle*. Un *bundle* contient un fichier manifest.mf précisant un certain nombre d'informations comme, par exemple, le nom du *bundle*, sa version, ses packages requis et fournis. Un *bundle* contient aussi les interfaces et classes Java nécessaires à la réalisation des services qu'il fournit et à l'utilisation des services qu'il

requiert. Il est important de noter que le compendium OSGi (qui est une extension de la spécification "cœur" d'OSGi) propose aussi une autre pseudo-unité de déploiement, à savoir le package de déploiement (*Deployment Package*) [OSGi07a]. Cette dernière est définie dans le chapitre 114 du compendium, via la spécification du *Deployment Admin*. Ainsi, un package de déploiement est défini comme un ensemble de ressources qui doivent être gérées d'un seul bloc. Ces ressources pouvant être, par exemple, un ensemble de *bundles* et leurs configurations. L'ensemble des ressources d'un package de déploiement ne doit pas être partagé avec des éléments extérieurs au package de déploiement. Le *Deployment Admin* ne gère que l'installation et la désinstallation des packages de déploiement, ce qui est loin de couvrir toutes les activités de déploiement définies par OSGi.

Le point d'entrée de la phase de déploiement couverte par OSGi est le *bundle*, c'est-à-dire un couple qui comprend, d'une part un fichier manifeste décrivant le corps du *bundle* et, d'autre part, le corps du *bundle* lui-même. Le fichier manifeste est créé en partant du corps du *bundle* auquel il sera associé. Ainsi, le point d'entrée de la phase de déploiement est un couple formé par une implémentation et un modèle créé à partir de cette implémentation. Cependant, il convient de préciser que le déploiement tel qu'il est appréhendé par la spécification OSGi ne peut être considéré comme un déploiement dirigé par les modèles.

Au niveau de la résolution des dépendances, nous avons déjà vu que les dépendances de packages des *bundles* sont gérées automatiquement par l'environnement d'exécution OSGi. C'est le rôle du *Module Layer*. Cependant, cette résolution au niveau package se fait en utilisant les *bundles* déployés dans l'environnement OSGi, le *Module Layer* ne permet pas la commande d'une quelconque activité de déploiement (comme la commande de l'installation d'un bundle offrant des packages requis non satisfait dans l'environnement). Du fait qu'OSGi se base sur l'approche à service, trois aspects concernant les dépendances de services sont à distinguer, à savoir :

- la résolution de dépendances de services,
- la validation de l'utilisation d'un service pour satisfaire une dépendance de service
- et enfin l'établissement d'un ordre pour l'activation (et la désactivation) des services faisant partie d'un ensemble de services et afin de permettre l'activation (et la désactivation) effective de cet ensemble de services. Nous parlons d'ordonnancement de l'activation et de la désactivation.

Pour bien comprendre le contenu des propositions de la spécification OSGi R4.1, il faut se rappeler que cette spécification est construite en trois parties. La première partie est le noyau (*core*) de la spécification [OSGi07]. La seconde partie est constituée du compendium [OSGi07a]. Enfin, la troisième et dernière partie concerne des propositions qui viendront ou non enrichir les futures versions de la spécification OSGi ; ces propositions prennent la forme d'un ensemble de *Request For Comments* (RFC) et de *Request For Proposal* (RFP). D'une manière générale, le noyau de la spécification ne fait aucune proposition concernant la résolution des dépendances de services et la validation de l'utilisation d'un service (bien qu'elle soulève ce problème [OSGi07d]). Donc, par défaut, c'est le développeur ou le déployeur qui doit gérer ces deux premiers aspects. Par contre, le noyau de la spécification fait une proposition autour de l'ordonnancement d'un ensemble de services, via le chapitre 8 *Start Service Level Specification* [OSGi07e].

De son côté, le compendium OSGi fait une proposition qui vise à régler, à la fois, le problème de la résolution de dépendances de services et le problème de la validation de l'utilisation d'un service. En effet, il définit un mécanisme appelé *Declarative Services* (DS). Ce mécanisme permet entre autres choses de résoudre à la volée les dépendances de services qu'un (ou des) composant embarqué dans un *bundle* peut avoir et cela en utilisant des services valides [OSGi07f]. Ainsi, lorsqu'un *bundle* est actif, si le DS peut résoudre, satisfaire, toutes les dépendances de services d'un composant embarqué dans ce *bundle*, alors, ce composant deviendra automatiquement actif, ce qui lui permettra de fournir ses propres services. De même, si une dépendance de service devient non satisfaite (c'est-à-dire si le service utilisé pour satisfaire cette dépendance n'est plus valide), alors, soit le DS réussira à trouver un autre service afin de satisfaire, à nouveau, cette dépendance et le composant restera actif, soit il ne réussira pas à trouver un autre service, alors le composant sera désactivé jusqu'à ce que le DS réussisse à trouver à nouveau un service résolvant cette dépendance de service. Attention, le DS recherche les services uniquement dans le

registre de services de l'environnement d'exécution OSGi, il ne permet pas de déployer des bundles afin d'introduire ou de retirer des services de la plateforme OSGi.

Pour trouver des services satisfaisants, le DS se base sur le mécanisme *trading* d'OSGi et sur un fichier XML propre au composant. Ce fichier XML est le descripteur du composant. Il permet de spécifier, entre autre, des informations de filtrage, de sélection, qui sont utilisées par le DS afin de trouver des services satisfaisants. Le compendium OSGi précise que la description du composant doit au minimum contenir une référence vers le service (c'est ce qui forme la dépendance de service) ; cette référence a un nom et spécifie le nom canonique de l'interface du service. Une référence peut aussi spécifier une cardinalité (soit 0..1, 0..n, 1..1, 1..n) précisant le nombre de services pouvant, devant, être mis à la disposition du composant. Une référence peut aussi spécifier une politique. Une politique prend la valeur *static* ou *dynamic*. Lorsque la politique *static* est choisie, alors, tout changement, dans les services qui satisfont les requis de service d'un composant, implique la désactivation, puis la réactivation du composant (le DS peut faire cela sans pour autant désactiver puis réactiver le *bundle*). A l'inverse, si la politique *dynamic* est choisie, alors, tant que chaque requis de service est satisfait par au moins un service, alors tout changement de service peut se faire sans que le DS n'ait à désactiver puis réactiver le composant. Enfin, une référence peut spécifier des propriétés sous la forme de couples clé-valeur (via les éléments *target*, *property* ou *properties*). Il faut avoir conscience que, bien que l'ensemble de ces informations permettent de filtrer les services pouvant satisfaire une référence particulière, le mécanisme de résolution de dépendances proposé par le DS n'en reste pas moins basé sur le mécanisme *trading* et, de plus, la résolution des services ne se fait qu'en fonction des services présents dans le registre de services de l'environnement d'exécution.

Au niveau de la résolution de dépendances de services, en plus du mécanisme DS, il existe une proposition nommée *OSGi Bundle Repository* (OBR). OBR est proposé via le RFC numéro 0112. Le but de l'OBR est de permettre de satisfaire, résoudre, les dépendances de services d'un *bundle* cible au moment de son installation. Cela rend ce mécanisme différent du mécanisme DS qui lui intervient au niveau de *bundles* d'ores et déjà déployés (plus précisément au niveau de *bundles* activés) et qui n'est pas capable de commander le déploiement de *bundles*. Pour effectuer la résolution des dépendances d'un *bundle*, OBR définit un dépôt de *bundles*. Ce dépôt consiste en un site web hébergé par l'Alliance OSGi qui contient des fichiers XML décrivant des *bundles*. Les *bundles* décrits pouvant être hébergés directement par le site web de l'Alliance OSGi ou par d'autres sites. Ensuite, l'OBR utilise ce dépôt (c'est-à-dire ce point d'entrée), afin de connaitre et résoudre les dépendances du *bundle* cible et en précisant (implicitement) un ordre (d'installation) sur les *bundles* à déployer. Il faut noter que le RFC de l'OBR, est particulièrement avare concernant les propriétés qu'offre la résolution, à peine précise-t-il que le mécanisme est récursif. En particulier, aucune précision n'est donnée quant au fait que la résolution prend en compte les *bundles* d'ores et déjà déployés ou quand au fait qu'elle gère la présence de cycle de dépendances. De même, le RFC précise, certes, qu'un ordre doit être établi concernant l'ensemble de *bundles* sélectionnés à déployer, cependant aucune information n'est donnée concernant cet ordre et son établissement.

En guise de remarque, pour faire sens, tout mécanisme de résolution de dépendances d'un *bundle* doit être un mécanisme récursif. En effet, lorsqu'un *bundle* est sélectionné pour satisfaire des dépendances, il est nécessaire que ses propres dépendances soient elles-mêmes satisfaites, sinon, il ne pourra être activé et, donc, il ne pourra satisfaire les dépendances pour lesquelles il a été sélectionné. Au final, bien que résolvant les dépendances, entre autres de services, d'un *bundle* lors de son installation, OBR ne permet pas d'assurer que l'ensemble des *bundles* sélectionnés interagiront bien suivant la résolution qu'il aura établie, voire même que les *bundles* interagiront bien ensemble. Par exemple, si nous souhaitons installer puis activer le *bundle* nommé A et si l'OBR propose l'installation des *bundles* C, B puis A (sachant que A est satisfait par B qui est lui-même est satisfait par C), alors, rien ne garantit que A utilisera bien les packages de B et que B utilisera bien les packages de C une fois l'installation réalisée. De même, une fois l'activation de C, B puis A réalisée, OBR ne peut pas non plus garantir que A utilisera bien les services de B et que B utilisera bien les services C. En effet, il est tout à fait possible que A utilise, par exemple, les *bundles* D, E et F qui se trouvaient déjà déployés dans l'environnement ou les *bundles* B et F.

Maintenant que nous avons examiné les propositions d'OSGi autour de la résolution de dépendances et de leurs limites, nous allons présenter les mécanismes proposés, toujours par OSGi, autour de la validation de l'utilisation d'un service.

Il est important ici de préciser pourquoi la validation de l'utilisation d'un service est nécessaire dans les environnements d'exécution dynamiques. Tout d'abord, il faut rappeler que le terme environnement d'exécution dynamique désigne un environnement d'exécution dans lequel des unités de déploiement peuvent être installées et désinstallées (voire activées, désactivées et maintenues), sans que l'environnement d'exécution n'ait besoin d'être désactivé puis réactivé. Le compendium OSGi précise que, dans le cas d'OSGi, la problématique de validation de l'utilisation d'un service porte plus précisément sur la validité de l'utilisation des objets, des instances, de services. En effet, du fait de la dynamicité de l'environnement d'exécution OSGi, il est possible qu'un *bundle* utilise un objet correspondant à un service qui a été dés-enregistré du registre de services, voire à un service d'un *bundle* qui a été désactivé et peut être même désinstallé. Cela a pour conséquence d'empêcher la suppression de l'objet correspondant au service et d'empêcher le déchargement ou la suppression des classes correspondantes à cet objet.

La spécification OSGi fait deux propositions autour de ce problème. La première est le mécanisme *Declarative Services* (DS), que nous avons présenté dans son ensemble ci-dessus. La seconde et dernière proposition est le mécanisme *Service Tracker* (ST), que nous présentons ci-dessous. Le mécanisme *Service Tracker* (ST) est défini dans le compendium OSGi. La spécification de ce mécanisme peut être trouvée dans le chapitre 701 du compendium [OSGi07g]. Le ST permet de surveiller les événements relatifs aux services, ces événements permettent de savoir si un service est valide ou non. Contrairement à DS, ST ne gère pas automatiquement la validation de l'utilisation d'un service. ST n'est qu'un moyen particulier qui peut être utilisé par le développeur pour développer du code capable de surveiller les événements liés au(x) service(s) qu'il utilise et cela afin de déterminer la validité de l'utilisation dudit service. De cette manière, le code développé pourra, si nécessaire, relâcher un service invalidé, permettant ainsi la suppression (de l'instance) du service, le déchargement, voire la suppression, de ses classes. Pour surveiller un service, le développeur doit expliciter le service à surveiller auprès du *Service Tracker*, puis il doit lancer la surveillance. Une fois ces deux étapes réalisées, un appel de méthode sur le ST suffit pour obtenir un service dont l'utilisation est valide ou pour vérifier si l'utilisation d'un service est valide.

Au final, la spécification OSGi R4.1 et les propositions qui l'accompagnent permettent de faire fonctionner un *bundle* (c'est-à-dire une application extensible), sans pour autant qu'il soit nécessaire de se préoccuper de l'identité des entités qui satisfont les dépendances du *bundle* cible. Cependant, l'environnement OSGi n'est pas satisfaisant dans le contexte que nous étudions. En effet, il ne permet pas d'assurer le déploiement des applications à services définies dans notre contexte (c'est-à-dire d'application à service formée par un ensemble de services pouvant interagir selon les spécifications établies par un développeur ou un architecte). Autrement dit, ni les mécanismes de résolution, ni les mécanismes de validation de l'utilisation de service proposés par OSGi (délégation au développeur, *Declarative Services*, *OSGi Bundle Repository*, *Service Tracker*) ne permettent d'assurer la correspondance entre une application à services définie par un développeur, par un architecte (c'est-à-dire une application constituée d'un ensemble de services pouvant interagir) et l'ensemble de services effectivement déployés pour réaliser cette application.

Nous venons de présenter l'ensemble des propositions d'OSGi concernant la validation de l'utilisation d'un service et de leurs limites, maintenant, nous allons présenter les mécanismes d'ordonnancement de l'activation et de la désactivation que propose la spécification OSGi.

La spécification OSGi propose explicitement un mécanisme permettant de mettre en œuvre un ordonnancement. Ce mécanisme est le *Start Level Service* (SLS) [OSGi07e]. En plus de cela, la notion d'ordonnancement apparait aussi de façon implicite dans les mécanismes DS et OBR. Tout d'abord, il faut noter que le mécanisme SLS spécifié par le noyau de la spécification OSGi est destiné à permettre le contrôle de l'ordre d'activation et de désactivation des *bundles* d'un environnement d'exécution OSGi que ce dernier soit en cours d'activation ou de désactivation ou simplement actif (c'est-à-dire en cours d'exécution). Ainsi, chaque *bundle* se voit assigner un niveau d'activation. Ce niveau d'activation peut par la suite être modifié. Concrètement, le niveau 0 correspond au niveau d'activation du *bundle System* (qui

représente l'environnement d'exécution lui-même). Les niveaux des autres *bundles* sont, quant à eux, tous strictement supérieurs au niveau 0. Il n'existe pas de limite supérieure. Maintenant, si un *bundle* a le niveau n, alors, il sera activé avant tout *bundle* de niveau strictement supérieur à n et il sera désactivé après que tous les *bundles* de niveaux strictement supérieurs à n aient eux-mêmes été désactivés. Ensuite, il est possible de définir un niveau d'activation courant pour un environnement. Ainsi, dès qu'un niveau courant est défini, l'environnement se charge d'activer les *bundles* dont le niveau est inférieur au niveau courant et de désactiver ceux dont le niveau est strictement supérieur au niveau courant.

Du point de vue des fonctionnalités, le SLS sert principalement à contrôler l'activation et la désactivation d'un environnement. Il sert aussi à modifier le niveau d'activation courant d'un environnement, ainsi qu'à gérer le niveau d'activation de chaque *bundle*. Ce mécanisme d'ordonnancement est utile pour permettre d'activer des *bundles* immédiatement après l'activation du *bundle System*, la spécification OSGI donne l'exemple d'un *bundle* chargé de dénombrer des événements ou d'un *bundle* de log. De plus, il permet aussi de débugger l'activation ou la désactivation d'un environnement. Ce mécanisme est nécessaire dans le cas général. En effet, l'activation d'un *bundle* A peut produire des erreurs ou échouer si il n'arrive pas à satisfaire tous ses requis de services. Or commencer par activer un *bundle* qui peut lui fournir ce service permettra d'assurer qu'ensuite l'activation du *bundle* A n'échouera pas (ou du moins, pas du fait de ce type de problème). De même, la désactivation d'un *bundle* dont un service est utilisé par un autre pourra, encore une fois dans le cas général, causer des erreurs, voire faire crasher le *bundle* utilisant ce service. Or ce second problème peut être, lui-aussi, rattrapé via l'ordonnancement des *bundles*. Enfin, il est important de noter que l'ordonnancement d'un ensemble de *bundles* n'est nécessaire que pour les activités d'activation et de désactivation.

Le mécanisme DS réalise implicitement un ordonnancement entre les composants contenus dans un ou des *bundles* et cela aussi bien lors de l'activation des composants que lors de leur désactivation. En guise de rappel, DS ne cherche à activer ou désactiver que des composants faisant partie de *bundles* qui sont eux d'ores et déjà activés. Concrètement, lorsqu'un *bundle* contenant un composant est activé, alors, DS va chercher à satisfaire les dépendances de ce composant afin qu'il puisse devenir actif. Par exemple, si un composant C n'a pas de dépendance de service, alors DS va l'activer immédiatement. Dans le même ordre d'idée, si un composant B requiert un service et si ce requis de service peut être satisfait par un service que fournit C, alors DS va regarder si C est actif, si ce n'est pas le cas, DS va chercher à l'activer et si C est déjà actif, alors, DS va ensuite activer B et lier le requis de service de B avec le service fourni par C. Donc, au final, pour activer B, DS aura implicitement ordonnancé l'ensemble B et C, en activant tout d'abord C, puis B. Nous venons de voir que DS ordonnance implicitement l'activation des composants de *bundles* activés. En complément, il faut noter que DS se concentre principalement sur l'activation et qu'il n'ordonnance pas la désactivation de composants.

Enfin, en ce qui concerne le mécanisme OBR, nous avons déjà précisé qu'aucune information n'est donnée quant à l'ordre établi par OBR lors d'une résolution, ni quant à la manière dont cet ordre est établi. De plus, OBR ne se préoccupe que de l'installation du *bundle* cible et des *bundles* nécessaires à la satisfaction de la chaine de dépendances ; or l'installation d'un ensemble de *bundles* (interdépendants ou non) ne nécessite ni l'établissement, ni l'utilisation d'un quelconque ordre du point de vue des dépendances de services.

En ce qui concerne la validation des dépendances entre bundles formant une application, elle n'est pas abordée par OSGi.

Au niveau du partage de services, la spécification OSGi permet effectivement de partager un service entre *bundles*. Un service est fourni par un *bundle* et un *bundle* utilise un service afin de satisfaire un de ses requis de services ; donc, deux *bundles* ayant des requis de services similaires pourront être amenés à partager le même service. Néanmoins, comme la spécification OSGi n'utilise pas le niveau application, il est inadéquat de parler de partage de service entre applications à services et cela bien que les *bundles* soient assimilés à des applications extensibles par la spécification OSGi.

Nous venons de voir que la spécification OSGi permet le partage de *bundles* entre plusieurs autres *bundles* via le niveau service. Nous avons aussi vu précédemment qu'un *bundle* a deux niveaux de dépendances, à savoir, le niveau package et le niveau service. Concernant la gestion du partage, la

spécification OSGi fait une proposition qui concerne uniquement le niveau package. Cette proposition est faite via la spécification du *Package Admin Service* (chapitre 7 du noyau de la spécification OSGi). Elle précise que ce dernier doit fournir une interface pour la gestion du partage des packages de *bundles* entre *bundles*, que cette interface est destinée à l'agent de gestion (*Management Agent*) de l'environnement d'exécution qui joue le rôle de décideur.
L'agent de gestion, en lui-même, est défini, en une page, dans le compendium. Sa définition précise essentiellement qu'il prend en charge l'installation, la désinstallation et la mise à jour statique des *bundles* et qu'il veille à ce que ces activités de déploiement respectent les contraintes de sécurité de l'environnement d'exécution. Pour être tout à fait exact, la spécification OSGi ne précise pas le fonctionnement du mécanisme qui permet la gestion du partage de packages de *bundles* entre *bundles*. Ainsi, la spécification OSGi fait une proposition (peu détaillée) autour de la gestion du partage de *bundles* entre *bundles* au niveau package. Par contre, aucune proposition n'est faite autour de la gestion du partage de services de *bundles* entre *bundles*.

Nous avons vu que des *bundles* peuvent être partagés entre plusieurs *bundles*. Or, cela pose des problèmes d'isolation. En effet, si deux *bundles* A et B utilisent un même *bundle* P, alors la spécification OSGi ne garantit pas que, dans ces conditions, A recevra de P les mêmes données que celles qu'il aurait reçu (de P) si il avait été le seul à l'utiliser. De plus, la spécification OSGi ne soulève pas la problématique de la gestion de l'isolation et ne fait pas non plus de proposition autour de cette problématique.

Concernant le déploiement à distance, la spécification OSGi fait mention d'un gestionnaire distant (*Remote Manager*), dont elle ne précise rien.

Le tableau ci-dessous synthétise la position et les propositions d'OSGi R4.1 concernant le déploiement.

	OSGI R4.1
Définition de la phase de déploiement	Processus non linéaire et non séquentiel, ***Install***, ***Activate***, ***DeActivate***, ***Static Update*** et ***DeInstall***.
Unité de développement	**Composant orienté service** (**grain fin**), ***Bundle*** c'est-à-dire application extensible (**gros grain**).
Unité de déploiement	***Bundle***, c'est une application extensible (=> **gros grain**), cependant, un *bundle* peut ne contenir qu'un composant (=> **grain fin**).
Point d'entrée de la phase de déploiement	***Bundle*** (implémentation + métadonnées extraites à partir de l'implémentation).
Résolution de dépendances et validation	**Deux types de dépendances: package et service.** **Les dépendances de package sont gérées automatiquement** par l'environnement OSGi via les *bundles* déployés. **Pour les dépendances de service**, il existe plusieurs mécanismes: ***Declarative Service*** résout les dépendances des composants orientés service en utilisant les *bundles* déployés, il ordonnance implicitement l'activation de ces composants et il valide les objets de service utilisés. ***OSGi Bundle Repository*** résout les dépendances d'un *bundle* cible lors de son installation, afin qu'il puisse être activé, cependant, OBR ne s'occupe que de l'installation et il ne garantit pas que les *bundles* installés pour satisfaire le *bundle* cible interagiront bien selon la résolution faite par OBR ou interagiront, au moins, ensemble. ***Service Tracker*** est un mécanisme de surveillance de services qui peut être utilisé comme base afin de programmer un système de validation des objets de services. La validation n'est pas abordée.
Partage d'implémentation, Gestion de ce partage.	**Oui**, tout service fourni peut être partagé entre plusieurs demandeurs de service (c'est-à-dire tout service fourni par un *bundle* peut être utilisé par plusieurs demandeurs de services appartenant à un ou plusieurs *bundles*). **Cependant, aucun gestionnaire n'est fourni.**
Partage d'instance d'implémentation, Gestion de ce partage.	**Oui**, toute instance de service peut être utilisée par plusieurs demandeurs de service. **Cependant, aucun gestionnaire n'est fourni.**
Gestion de l'isolation	**Non.**
Remarque	Le mécanisme *Start Service Level* permet de spécifier un ordre au niveau de l'activation et de la désactivation de tout ou partie des *bundles* déployés dans un environnement. Le mécanisme de sélection d'un service est basé sur l'approche *trading*.

Tableau 7. Synthèse sur les capacités de déploiement d'OSGi R4.1.

5.2 Service Component Architecture (SCA)

Nous avons déjà présenté la spécification issue du projet *Service Component Architecture* (SCA) dans l'état de l'art service de cette thèse. Nous nous focalisions, alors, sur la vision de l'approche à service proposée par cette spécification. Ici, nous allons présenter les aspects de cette spécification ayant trait à la phase de déploiement. Tout d'abord, il faut noter que la spécification SCA possède une partie dédiée uniquement au déploiement. Cette partie est intitulée *Packaging and Deployment*. Ainsi, les termes : *SCA Domain*, *Contribution* et *Composite* sont utilisés afin de préciser le packaging et le déploiement définis par la spécification SCA.

Le terme *SCA Domain* représente l'environnement cible. Cet environnement peut être distribué (il peut contenir un ou plusieurs nœuds). Un domaine SCA représente typiquement un ensemble de fonctionnalités *business* contrôlées par une même organisation. Il contient un composite virtuel dans lequel les composants sont exécutés, un ensemble de contributions (contenant des implémentations, interfaces et d'autres artéfacts nécessaires à l'exécution des composants) et un ensemble de services, respectivement de références, exposés par les composants contenus dans les contributions, respectivement requises par ces mêmes composants. Le terme *Contribution* désigne un composite packagé. A titre de comparaison, une contribution joue le même rôle pour le monde SCA qu'un *bundle* pour le monde OSGi. Une contribution peut posséder des métadonnées. Ces métadonnées explicitent l'ensemble des composites exécutables contenus dans la contribution, ainsi que les services requis pour l'exécution des composites et les services fournis lors de l'exécution des composites. Par défaut, une contribution est packagée sous la forme d'un .zip. Cependant, ce format n'est pas obligatoire, il doit juste être supporté par défaut. Ainsi, une contribution peut tout à fait être packagée sous la forme, par exemple, d'un .rar, d'un .jar, d'un .ear. Le packaging dépend, dans les faits, de la technologie utilisée pour réaliser la spécification SCA. Il est aussi précisé explicitement que, lors de la phase de déploiement, une contribution peut être installée, mise à jour et désinstallée d'un domaine SCA. Enfin, la spécification SCA précise encore un peu plus le terme *Composite*. En guise de rappel, un composite est l'unité de composition du monde SCA. Il fournit des services et requiert des références. Lors de la phase de déploiement, un composite appartient à une contribution. Il est donc installé, mis à jour et désinstallé via sa contribution. Cependant, il est aussi possible d'ajouter des composites à une contribution qui est déjà installée. De même, il est possible de mettre directement à jour un composite appartenant à une contribution installée.

Suite à cette introduction, deux points sont à remarquer. Premièrement, la spécification SCA définit un cycle de vie du déploiement qui comprend les activités d'installation, de mise à jour et de désinstallation. Elle précise explicitement qu'un service fourni par un composite n'est effectivement fourni que lorsque le composite est en cours d'exécution. Or, la spécification SCA ne donne aucune précision ni sur la mise en exécution (c'est-à-dire sur l'activité d'activation), ni sur une possible activité d'arrêt de l'exécution (c'est-à-dire sur l'activité de désactivation). Ainsi, le cycle de vie du déploiement proposé par SCA contient d'une part les activités d'installation, mise à jour, désinstallation qui sont détaillées et d'autre part, l'activité d'activation qui n'est pas même explicitement citée (ni ses dérivés, comme, *start*, *launch* ou *stop*), le terme *execute* apparaît très peu et ne précise rien, non plus, concernant la mise en exécution, l'exécution elle-même et l'arrêt de l'exécution. La spécification SCA ne précise pas non plus si l'activité de mise à jour est une mise à jour statique et/ou dynamique. Le second point remarquable concerne l'unité de déploiement ou plutôt les unités de déploiement utilisées par la spécification SCA. En effet, deux unités de déploiement peuvent être utilisées. La première est la contribution. Une contribution peut être installée, mise à jour, désinstallée (et peut être activée, voire désactivée ?). La seconde est le composite. Ce dernier devant faire partie d'une contribution déployée pour pouvoir être exécuté. Un composite peut être ajouté à une contribution (c'est-à-dire installé et ajouté au sein même d'une contribution), mis à jour et peut être activé et désactivé (?!., dans les faits, ce sont les composites qui sont exécutables). Un troisième point remarquable concerne le point d'entrée de la phase de déploiement. La spécification SCA utilise le concept de package (implémentation et modélisation de l'implémentation via des métadonnées) aussi bien pour les contributions que pour les composites. Dans ces packages, la présence des métadonnées est signalée comme étant optionnelle dans le cas général. Néanmoins, l'utilisation du mécanisme de résolution de dépendances définit par la spécification SCA requiert la présence de ces métadonnées.

En ce qui concerne la résolution de dépendances, la spécification SCA précise que seuls les composites valides peuvent être activés (c'est-à-dire ceux dont l'ensemble des références des composants et composites sont satisfaites, soit dans le composite lui-même, soit en étant déportée sur le composite). De plus, la spécification SCA propose son propre mécanisme de résolution des références qui ont été déportées sur le composite packagé (c'est-à-dire sur la contribution) et cela en mettant en jeu les services d'autres contributions. Néanmoins, la spécification SCA ne précise pas si ces autres contributions doivent faire partie du même domaine SCA. Enfin, la spécification SCA précise qu'il est préférable d'utiliser les mécanismes de résolution de dépendances propres aux technologies servant à réaliser l'environnement SCA, si ces derniers sont disponibles. Elle précise aussi que le mécanisme proposé est avant tout un mécanisme destiné à être utilisé comme un pont entre technologies (par exemple, entre des composants, composites basés sur OSGi, un code natif en Cobol et un composite basé sur le langage C++).

La validation des dépendances entre composites formant une contribution ou entre contributions n'est pas abordée par SCA.

Maintenant, nous allons nous intéresser aux possibilités de partage, ainsi qu'aux fonctionnalités de gestion de partage et d'isolation qu'offre la spécification SCA. La spécification SCA précise qu'un composant est une instance configurée d'une implémentation et qu'un composite peut être vu comme une implémentation. De plus, un composant appartient obligatoirement à un composite. La spécification SCA précise aussi qu'une instance d'implémentation tire sa logique *business* de l'implémentation sur laquelle elle est basée, mais que les valeurs des propriétés et les références sont tirées du composant qui configure l'implémentation. De plus, elle précise que toute instance d'une implémentation est le résultat d'une instanciation spécifique de cette implémentation faite lors de l'exécution. Ainsi, il ressort de ces différents éléments que chaque composant SCA possède sa propre instance configurée lors de l'exécution et que plusieurs composants peuvent tirer leur logique *business* de la même implémentation. La spécification SCA précise que tout composite déployé appartient à une contribution et que lorsqu'un composant appartient à une contribution, alors, l'implémentation correspondant à ce composant doit aussi être présente dans cette même contribution. Donc, seul le partage d'implémentation (entre composants d'une même contribution) est possible dans le monde SCA. Ensuite, la spécification SCA ne donne aucune précision concernant la gestion du partage d'implémentation ou ne fait mention d'aucun mécanisme dans ce sens. Cependant, vues les hypothèses prises concernant l'encapsulation des implémentations dans les contributions et vu que tout composant possède sa propre instance et sachant que l'activation des composites semble se faire via les contributions et que seules ces dernières peuvent être désinstallées, alors la stratégie de gestion du partage d'implémentations que la spécification SCA a retenu ne requiert pas de gestionnaire de partage d'implémentations en tant que tel.

En ce qui concerne l'isolation, il faut noter que la spécification SCA définit une stratégie d'isolation de manière implicite pour les composants. En effet, la spécification SCA précise que tout composant SCA possède sa propre instance configurée de l'implémentation sur laquelle il est basé. Or cette stratégie de gestion des instances garantit l'isolation au niveau composant. De plus, elle peut être mise en œuvre facilement dès lors qu'une fabrique d'instances est disponible pour toute implémentation. Néanmoins, la spécification SCA ne précise ni comment chaque composant obtient sa propre instance, ni comment est gérée la mise à jour d'un composite ou d'une contribution et quelles sont les conséquences au niveau instance.

Enfin, la spécification SCA permet de partager l'utilisation des services d'une contribution entre plusieurs contributions, tout en occultant le problème de la gestion du partage de contributions. Par exemple, aucune précision n'est donnée concernant la possibilité de mettre à jour et/ou désinstaller une contribution dont les services sont nécessités ou en cours d'utilisation par d'autres contributions. De même, la spécification SCA ne précise pas sa stratégie concernant la mise à jour d'un composant d'une contribution qui est partagée. Enfin, elle ne précise rien, non plus, autour d'une quelconque isolation entre deux contributions partageant une même contribution.

Finalement, il est précisé que le contenu d'une contribution, qui a été développée et packagée, ne doit nécessiter aucune nouvelle modification en vue d'être installée et utilisée dans un domaine SCA. Cet axiome est similaire à l'un de ceux que formule C. Szyperski pour le déploiement de composants (à la partie configuration près).

	SCA V1.00
Définition de la phase de déploiement	Processus non linéaire et non séquentiel, ***Installation***, ***Update***, ***DeInstallation***, ***Execution*** (la spécification ne précise rien autour de l'*Activation* et de la *DeActivation*).
Unité de développement	**Composant orienté service, composite orienté service** (grain fin).
Unité de déploiement	***Package*** contenant une/des ***Contribution*** (gros grain) ou un/des ***Composition*** (grain fin).
Point d'entrée de la phase de déploiement	***Package*** (implémentation + métadonnées extraites à partir de l'implémentation).
Résolution de dépendances et validation	Dans le cas général, **les dépendances de services des composants et des composites doivent être gérées à la main** et décrites dans les métadonnées du package. **Celles qui sont exposées au niveau contribution sont résolues automatiquement** par l'environnement SCA et cela en suivant les métadonnées des *Packages* et en utilisant les autres contributions. En complément, la spécification SCA spécifie aussi **le mécanisme *AutoWire***, qui **est chargé de satisfaire automatiquement les dépendances** (c'est-à-dire les références sans lien prédéfini) **sur demande.** La validation n'est pas abordée.
Partage d'implémentation, Gestion de ce partage.	**Oui**, toute implémentation peut être partagée entre plusieurs composants d'une même contribution. Pas de gestion au niveau de la phase de déploiement. Cependant, **tout composant possède sa propre instance** configurée de l'implémentation.
Partage d'instance d'implémentation, Gestion de ce partage.	**Non**, le partage d'instance d'implémentation n'est pas autorisé, (=> pas de gestionnaire).
Gestion de l'isolation	Pas de gestion au niveau de la phase de déploiement. Cependant, le fait que tout composant s'exécutant ait sa propre instance configurée garantit l'**isolation au niveau des composants/composites** et des instances.
Remarque	Toute contribution peut être partagée entre plusieurs autres contributions.

Tableau 8. Synthèse sur les capacités de déploiement de SCA.

6. SYNTHÈSE

Nous allons maintenant développer la synthèse correspondant aux six cas d'études que nous avons présentés dans les trois sections précédentes.

Tous les cas d'études, mis à part CCM V4.0, supportent au moins les quatre activités de déploiement les plus fondamentales, à savoir, l'installation, l'activation, la désactivation et la désinstallation (CCM ne supporte que les activités d'installation, d'activation et de désactivation). .Net V3.0, *Software Dock*, SCA V1.00 et OSGi R4.1 (d'une certaine manière) supportent tous les quatre l'activité de mise à jour statique. .Net propose même sept stratégies différentes pour la mise en œuvre d'une mise à jour statique. Il supporte aussi la réparation des unités de déploiement déployées. De son côté, *Software Dock* supporte l'adaptation des logiciels déployés.

Au niveau des unités de déploiement, les six cas d'études présentés proposent tous des unités de déploiement composées d'implémentations (c'est-à-dire d'unités de développement) et d'un descripteur. La présence d'un descripteur est obligatoire pour CCM, .Net, *Software Dock* et OSGi et est optionnelle pour EJB V3.0 et SCA. Les unités de déploiement proposées peuvent toutes encapsuler, empaqueter un ensemble d'unités de développement, voire même un logiciel entier dans le cas de *Software Dock* et de SCA. En conséquence, ces unités de déploiement sont considérées comme des unités de gros grain. Cependant, CCM, EJB, .Net, OSGi et SCA permettent aussi qu'une unité de déploiement ne contienne qu'une seule unité de développement. Dans ce second cas, ces unités de déploiement doivent être considérées comme des unités de grain fin (puisqu'elles ont le même grain que leur unité de développement respective). Maintenant, dans le cas du déploiement d'un logiciel composé de plusieurs unités de développement où chaque unité est packagée dans sa propre unité de déploiement, alors, seul .Net permet de spécifier un descripteur de déploiement externe aux différentes unités de déploiement, ce qui permet de décrire un logiciel dans son entier. Pour ce faire, il définit un fichier *Portable Executable* externe aux *assemblies*.

En ce qui concerne la résolution de dépendances, CCM, EJB et .Net ne proposent aucun mécanisme de résolution automatique des dépendances. Cela signifie que toutes les dépendances doivent être gérées à la main. Ainsi, pour CCM, les dépendances entre composants CCM doivent être décrites dans les méta-données du *Component-Package*. Pour EJB, les dépendances entre les *beans* ne sont pas même décrites dans le descripteur de déploiement, les requis d'un *bean* apparaissent dans son code (lors des *look-up*). Pour .Net, les dépendances d'un *assembly* doivent être spécifiées dans son fichier PE et dans le cas d'un ensemble d'*assemblies*, elles doivent être décrites dans un fichier PE externe aux *assemblies*.

De leur côté, *Software Dock*, OSGi et SCA proposent un, voire plusieurs mécanismes dans le cas d'OSGi. Le mécanisme de *Software Dock* propose une résolution automatique des dépendances d'un *Package*. Cette résolution se fait via l'usage du fichier DSD associé au *Package*, dans lequel sont exprimés les requis du *Package*. Le mécanisme de résolution proposé intervient lors de l'activité d'installation. Il est aussi important de noter que le mécanisme proposé est sensible au contexte. Cela signifie qu'il vérifie si l'environnement cible contient des *Packages* satisfaisant les requis d'un *Package* cible, avant l'installation de ce dernier. De plus, si tous les requis du *Package* cible ne sont pas satisfaits, *Software Dock* se charge d'installer les *Packages* manquants. OSGi, quant à lui, propose un mécanisme automatique de résolution des dépendances de packages entre bundles, mais, par défaut, ne propose pas de mécanisme pour la résolution des dépendances de services. Hors de ce fonctionnement par défaut, les mécanismes *Declarative Service*, OBR et *Service Tracker* sont proposés. Enfin, SCA propose son propre mécanisme de résolution des références qui ont été déportées sur le composite packagé (c'est-à-dire sur la contribution) et cela en mettant en jeu les services d'autres contributions. Ce mécanisme est nommé *AutoWire*. Cependant, SCA précise qu'il est préférable d'utiliser les mécanismes de résolution de dépendances propres aux technologies servant à réaliser l'environnement SCA, si ces derniers sont disponibles. *AutoWire* est avant tout destiné à être utilisé comme un pont entre technologies.

Au niveau de la validation des dépendances entre unités de déploiement formant une même application, aucun des cas d'études présentés ne l'aborde. Par exemple, EJB ne propose pas de mécanisme

de validation d'applications à *beans*, d'ensembles de *beans*, donc les *beans* déployés peuvent avoir des dépendances non-satisfaites. Il en est de même pour OSGi, et SCA, mais aussi pour .Net, et bien que ce dernier permette la description d'une application à *assemblies*.

En ce qui concerne le partage d'implémentation, EJB, .Net, *Software Dock*, OSGi et SCA le permettent. Cependant, EJB ne le gère absolument pas. Ainsi, tout *bean* peut être utilisé par d'autres *beans* appartenant à des logiciels, à des ensembles de *beans* différents. De son côté, .Net propose un gestionnaire de partage d'*assembly*. Ce gestionnaire peut être mis en œuvre, soit en utilisant "gacutil.exe", mais dans ce cas, c'est au déployeur de préciser si l'application en cours de déploiement doit être notée comme nécessitant tel ou tel *assembly* ou non (via les paramètres "/ir" et "/ur"), soit en utilisant *Windows Installer* qui, lui, est un mécanisme automatique. L'information stockée, par .Net, afin de gérer le partage des *assemblies* est tout à fait rudimentaire. Elle consiste en un compteur dont la valeur correspond au nombre de références faites sur l'*assembly*. De son côté, *Software Dock* propose un gestionnaire similaire, qui est, cependant, obligatoire et transparent. OSGi permet de partager l'utilisation d'un bundle entre plusieurs autres bundles, mais aucun gestionnaire n'est proposé. Enfin, SCA permet le partage d'une implémentation entre plusieurs composants, cependant, ces derniers doivent tous appartenir à la même contribution (dans ce dernier cas, aucun gestionnaire de partage n'est nécessaire, le partage se faisant à l'intérieur d'une unité de déploiement).

Au niveau du partage d'instance d'implémentation, CCM ne propose rien. Cela signifie que le partage d'instance d'implémentation n'est pas possible. En effet, chaque composant possède sa propre instance configurée qui lui est donnée par le *Component Home* auquel il est rattaché. Pour EJB, seuls les *Entity Beans*, dont une seule instance est créée et gérée via l'*Entity Manager*, permettent le partage de leur instance entre plusieurs *beans*. Il est important de comprendre qu'un *Entity Bean* représente une table dans une base de données et qu'en conséquence, le partage de l'instance d'un *Entity Bean* signifie le partage de la table qu'il représente. Pour la technologie .Net, seule l'unique instance créée pour un *shared assembly* peut être partagée. C'est le CLR qui prend en charge le partage des instances de *shared assemblies*. Ainsi, tant qu'une instance d'*assembly* active possède une référence vers une instance d'un *shared assembly*, alors l'instance du *shared assembly* ne sera pas détruite (ce *shared assembly* restera actif). *Software Dock*, quant à lui, ne prend pas en charge la gestion du niveau instance d'implémentation. Ensuite, SCA ne permet pas le partage d'instance d'implémentation entre composants. Enfin, en OSGi, toute instance d'implémentation de service peut être partagée. Cependant, OSGi ne gère pas ce partage.

En ce qui concerne la gestion de l'isolation, CCM n'a pas besoin de fournir un gestionnaire d'isolation en tant que tel, puisque chaque composant reçoit une instance qui lui est propre et qui lui est attribuée par le *Component Home* auquel il est rattaché. La technologie EJB, quant à elle, ne propose aucun mécanisme pour gérer l'isolation entre deux ensembles de *beans* déployés dans le même conteneur EJB. .Net ne propose pas non plus de mécanisme pour gérer l'isolation entre logiciels partageant un même *assembly*. Cependant, étant donné que .Net distingue deux types d'*assembly*, à savoir les *private assemblies* et les *shared assemblies*, c'est au développeur de choisir d'isoler tout ou partie de son logiciel. Ensuite, *Software Dock* ne propose pas non plus de gestionnaire d'isolation étant donné que la gestion du niveau instance est laissée à la charge du système d'exploitation de l'environnement cible. OSGi ne gère pas l'isolation. Enfin, le partage d'instance n'est pas permis par SCA, en conséquence, l'isolation est garantie et la présence d'un gestionnaire d'isolation n'est pas requise.

7. CONCLUSION

D'un point de vue général, le déploiement logiciel consiste à déplier, mettre en œuvre une unité de déploiement, un ensemble d'unités de déploiement ou un assemblage d'unités de déploiement (dans un ou plusieurs environnements d'exécution). Dans cet état de l'art, nous avons vu et souligné l'absence de consensus actuel autour de la définition du déploiement. Les activités d'installation, d'activation et de désactivation se retrouvent, néanmoins, dans la plupart des définitions. Cependant, afin de rendre la phase de déploiement cohérente, l'activité de désinstallation devrait aussi devenir une des activités consensuelles. En effet, après avoir introduit un logiciel dans un environnement, avoir lancé son exécution, puis avoir l'avoir arrêtée, il parait cohérent de pouvoir supprimer le logiciel (c'est-à-dire de pouvoir le désinstaller). En complément, des activités de déploiement comme la réparation, la mise à jour statique, l'adaptation sont aussi parfois mises en avant par certaines définitions.

A notre avis, le premier défi actuel concernant le déploiement logiciel porte sur la capacité, que doit posséder le déploiement, d'agir au plus près des unités de développements composant un logiciel. En effet, il n'est plus question aujourd'hui d'agir sur un logiciel (et en particulier sur une entité composant un logiciel) en enlevant, en touchant, en impactant l'ensemble de ce logiciel. Qui plus est, dans le cas de l'approche à service, les unités de développement se présentent comme substituables, il est donc important pour le déploiement de tirer partie de cette capacité de substitution. Un exemple caractéristique est la démarche actuelle de modularisation des conteneurs JEE au-dessus de technologies orientées service, dont le but est de permettre l'exécution d'activités de déploiement (concernant le conteneur) sans pour autant nécessiter le redémarrage dudit conteneur [Des07].

Le second défi porte sur la poursuite de la montée en abstraction du déploiement. Concrètement, à l'origine, un déployeur déployait une implémentation (typiquement, une archive .jar peut être utilisée comme "emballage" pour une implémentation [SUN03a]). A l'heure actuelle, il déploie des implémentations décorées de métadonnées [OSGi07] [CLM05] (mais aussi [SUN03a], si un .jar est manipulé avec son manifeste). Nous pensons que la prochaine étape consistera à déployer des logiciels en ne se basant que sur leur modèle (complété par l'identifiant de l'environnement cible du déploiement et l'activité de déploiement à exécuter ou l'état de déploiement final souhaité).

En complément, [Dea07] identifie la virtualisation, ainsi que le déploiement sur les grilles, les réseaux mobiles et les réseaux de capteurs comme des défis à venir pour le déploiement logiciel. Dans [And00], l'auteur souligne que la mise en œuvre effective des changements dans l'architecture d'une application est un problème à part entière du déploiement logiciel. De tels changements architecturaux sont connus sous les termes d'évolution (qui désigne tout changement dans l'architecture d'une application qui n'est pas active) et de dynamisme (qui désigne tout changement dans l'architecture d'une application qui est active) dans l'univers des langages de descriptions d'architectures (*Architectural Description Language*s) [MT00]. J. Andersson souligne aussi que le déploiement à grande échelle (*Large Scale Deployment*) est un autre défi à relever. Concernant ce dernier défi, T. Coupaye et J. Estublier précisent, dans un article sur les fondements du déploiement de logiciels d'entreprise, que le déploiement à grande échelle est particulièrement complexe [CE00] ; en effet :

- les applications peuvent contenir des centaines de composants,
- elles évoluent rapidement (c'est-à-dire qu'une nouvelle version est publiée tous les deux mois),
- elles peuvent dépendre d'autres applications, composants, services (comme, par exemple, des OS, des *middlewares*, etc.),
- elles peuvent être déployées dans des environnements hétérogènes,
- enfin, des contraintes comme le temps consommé, la synchronisation, la consistance (c'est-à-dire le respect des contraintes de compatibilité entre, par exemple, applications, composants), la criticité (par exemple, pour des logiciels nécessaires à la production) peuvent intervenir.

Les défis présentés ci-dessus font apparaitre ou renforcent un certain nombre d'enjeux pour le déploiement logiciel. Ainsi, il devient nécessaire :

- de pouvoir exprimer des applications packagées sous la forme d'unités de déploiement non-monolithiques (c'est-à-dire composées d'ensembles d'unités de déploiement),

- d'effectuer la convergence entre le grain de l'unité de développement et le grain de l'unité de déploiement (nous parlons de "déploiement au plus près"),
- de trouver une unité de déploiement ne nuisant pas à l'expressivité des applications et permettant de répondre au défi du déploiement au plus près,
- de pouvoir déployer de telles applications,
- d'être capable d'établir des liaisons, dans un environnement distribué (ou non), ce qui implique, par exemple, l'introduction du patron *service locator*, de l'*inversion of control* (ou encore de la *dependency injection*) et/ou de *middlewares* appropriés,
- de maintenir, tout au long du déploiement, les métadonnées associées à un logiciel et/ou le lien entre le modèle du logiciel et les unités de déploiement effectivement déployés,
- d'exprimer et de choisir (si nécessaire) la répartition d'un logiciel distribué,
- d'appréhender l'hétérogénéité des environnements cibles,
- d'appréhender le nombre des environnements cibles (1 versus 10 versus 100 versus 1000000)
- et enfin, d'être capable de mettre en place une architecture de déploiement ou d'utiliser l'architecture de déploiement existante.

Au niveau du déploiement d'applications à services, seul SCA permet réellement l'expression d'une application à services d'une manière générale (c'est-à-dire au-delà de la notion d'application extensible mise en avant par OSGi). Cependant, l'expression d'une telle application est uniquement possible dans le cadre d'un *package* (c'est-à-dire d'une unité de déploiement). Donc, du point de vue du déploiement, il n'existe pas, à l'heure actuelle, de mécanisme d'expression des applications à services non-monolithiques.

Dans cet état de l'art, nous avons vu que la vision service web n'est pas suffisante pour exprimer une application à services dans le cas général. De même, la vision composant orienté service n'est pas entièrement satisfaisante pour le déploiement des applications à services au plus près. Du fait de la non expression des compositions entre fournisseurs et demandeurs dans un même composant et de la non-substituabilité des services de composants orientés services et des composant orientés services eux-mêmes. Il est donc nécessaire de proposer une unité de développement/déploiement ne nuisant pas à l'expressivité des applications et permettant de répondre au défi du déploiement au plus près.

Concernant la mécanique du déploiement en elle-même, il est, à l'heure actuelle, nécessaire d'assurer qu'une application déployée correspond bien à sa description, à son modèle. Cela est d'autant plus nécessaire, dans le cadre d'un déploiement ne prenant en entrée, non plus un couple composé d'une implémentation et de métadonnées, mais un modèle de l'application à déployer, sans lien avec le niveau implémentation. Ainsi, offrir l'assurance discutée ci-dessus passe par la maitrise des niveaux implémentations, instances d'implémentations et liaisons entre instances (les entités appartenant à ces trois niveaux intervenant toutes dans la réalisation concrète d'une application à services). Cette maitrise vise à maintenir le lien entre le modèle de l'application et les entités effectivement mises en jeu.

La problématique de l'ordonnancement des activations et, respectivement, des désactivations d'implémentations lors de l'activation et, respectivement, de la désactivation d'une application doit, elle aussi, être appréhendée.

Le déploiement d'applications à services dans des environnements d'exécution à services qui permettent aussi bien le partage d'implémentations que d'instances, sans pour autant le gérer, offre l'opportunité d'investiguer le partage d'implémentations, le partage d'instance d'implémentations, l'isolation et leur gestion (lors du déploiement).

Dans le cadre de cette thèse, nous proposons un système automatisant le déploiement de modèles d'application à services (non-monolithiques du point de vue du déploiement), respectant la définition du déploiement que nous avons mise à plat et proposant une solution aux enjeux relatifs au déploiement d'applications à services présentés dans les paragraphes précédents.

SECONDE PARTIE : CONTRIBUTION

III. PROBLÉMATIQUE

Dans ce chapitre, nous allons présenter en détail la problématique de cette thèse, à savoir, le déploiement d'applications à services sur des environnements d'exécution à services. Nous commencerons par rappeler succinctement l'état des lieux issu de la partie précédente. Ensuite, nous nous intéresserons au contexte de cette thèse. Puis, nous détaillerons la problématique, ainsi que les défis soulevés par la conjonction de cette problématique, de ce contexte et de cet état des lieux.

1. RAPPELS

La première partie de cette thèse nous permet de formuler quatre constats, à savoir :

- Le premier constat porte sur le fait que les visions actuelles de l'approche à service, c'est-à-dire les visions service web et composant orienté service, ne sont pas satisfaisantes concernant l'expression et la manipulation d'applications à services. En effet, la vision service web ne permet pas la composition entre fournisseurs et demandeurs de services au sein d'une même entité logicielle (c'est-à-dire la composition entre des services fournis et des services requis au sein d'une même entité logicielle). Dit autrement, un service web exprime, certes, ce qu'il fournit, mais ne permet pas d'exprimer de possibles requis (via des demandeurs de services, liés ou non à ce qu'il fournit). La vision composant orienté service permet, quant à elle, cette composition via l'utilisation de composants. Cependant, cette composition n'est pas explicitée, ainsi le contrat d'un fournisseur porte sur ce qu'il fournit mais ne fait aucune mention des possibles demandeurs qu'il utilise au sein du composant (c'est aussi le cas pour les EJB, par exemple). Or, cette information est généralement nécessaire dès lors que les objectifs à atteindre sont l'expression de l'architecture d'une application et la garantie de la transparence de la substitution d'un service d'un composant par un autre service (de contrat équivalent) appartenant à un autre composant. Enfin, toujours dans le cadre des applications (à services), seul SCA propose un niveau "application" via l'introduction et l'utilisation de la notion de composite au-dessus des composants orientés services. Un tel niveau "application" est nécessaire, puisqu'il permet de réaliser et d'expliciter les compositions entre demandeurs et fournisseurs (typiquement entre un service requis d'un composant et un service fourni d'un autre composant) permettant ainsi l'expression de l'architecture de l'application.

- Le second constat se base sur l'existence de plusieurs définitions (différentes) du déploiement et sur le fait que la cohérence et la complétude de chacune de ces définitions peuvent être critiquées (de manière plus ou moins forte, plus ou moins évidente) pour souligner le besoin d'établissement d'une définition du déploiement qui soit la plus cohérente et la plus complète possible. L'idée est d'adapter une des définitions existantes afin d'augmenter sa cohérence et sa complétude sans pour autant exclure les apports des autres définitions (comme présenté dans la première partie de ce document, nous nous sommes focalisés sur trois définitions majeures, celle de C. Szyperski, celle de l'OMG et celle de l'université du Colorado). Nous verrons dans le chapitre proposition que nous avons choisi d'adapter la définition de l'université du Colorado.

- Le troisième constat porte sur le grain des unités de déploiement. A notre avis, trois grains principaux peuvent être observés. Le premier grain correspond au grain d'un logiciel, c'est le cas lorsqu'un logiciel est packagé sous la forme d'une seule unité de déploiement. Nous parlons alors d'unité de déploiement gros grain. Le second grain est lié à la distribution du logiciel, dans ce cas, les différentes entités (relatives à la distribution) composant le logiciel sont packagées chacune dans une unité de déploiement qui leur est propre (ainsi, par exemple, un logiciel distribué sur trois ordinateurs est packagé selon trois unités de déploiement, chaque unité de déploiement packageant l'entité logicielle relative à l'ordinateur cible). Le troisième et dernier grain correspond au grain de l'unité de développement utilisée dans le logiciel. Nous parlons d'unité de déploiement de grain fin. Ce dernier grain permet d'introduire et d'obtenir des propriétés de déploiement à l'intérieur d'un logiciel, au plus près de ses unités de développement. Nous parlons de déploiement au plus près. Deux exemples particulièrement illustratifs sont la transition qui s'effectue actuellement entre des conteneurs JEE packagés de façon monolithique et ces mêmes conteneurs modularisés et packagés suivant plusieurs unités de déploiement et l'apparition, l'avènement de plusieurs technologies, telles .Net, OSGi et SCA, qui permettent qu'une unité de déploiement ne contienne qu'une unité de développement.

- Enfin, le quatrième et dernier constat porte sur la montée en abstraction du point d'entrée de la phase de déploiement (et en définitive du/des point(s) d'entrée des activités de déploiement). En effet, le déploiement prenait initialement en entrée une unité de déploiement contenant une

unique implémentation. Actuellement, le déploiement prend généralement comme point d'entrée soit une unité de déploiement contenant une implémentation et des méta-données, soit un ensemble d'unités de déploiement où chaque unité est un couple formé d'une implémentation et de méta-données, soit des méta-données et un ensemble d'unités de déploiement où chaque unité contient une implémentation et peut contenir des méta-données. A notre avis, la prochaine étape dans la montée en abstraction du point d'entrée de la phase de déploiement sera l'utilisation de modèles découplés du niveau implémentation et vraisemblablement l'introduction de systèmes de déploiements dirigés par les modèles.

2. CONTEXTE

2.1 Contexte général

L'offre de services Internet de haut niveau devient de plus en plus importante aujourd'hui. C'est en particulier le cas pour les services basés sur des données continuellement collectées à partir d'environnements physiques installés dans des usines, des centres commerciaux ou des réseaux domestiques. Dans l'industrie de la distribution électrique, par exemple, les fabricants conçoivent, développent et déploient de tels services Internet afin d'aider leurs clients à gérer leurs équipements électriques et afin de leur offrir (et facturer) des solutions pour optimiser leur consommation électrique. Pour ce faire, les fabricants doivent créer des suites logicielles complexes et doivent, en outre, mettre en place et appréhender des environnements eux-aussi complexes. L'infrastructure ainsi mise en place doit, qui plus est, passer à l'échelle et être flexible, elle doit aussi appréhender les changements à venir de manière transparente et enfin, elle doit assurer que les experts électriques (et non informaticiens) pourront gérer les services Internet offerts aussi facilement que possible [Lal05].

La plupart des systèmes offrant des services Internet utilisant des données environnementales sont basés sur des architectures n-tiers. De telles architectures mettent en jeu une application web qui fonctionne sur un serveur Internet et qui est connectée à des passerelles pervasives au-dessus de protocoles basés sur IP (tel, par exemple, HTTP). Ces passerelles pervasives sont connectées à des équipements physiques via des bus (CAN, par exemple). Elles collectent, transforment et envoient régulièrement des données au serveur Internet. Une fois reçues par le serveur, ces données sont stockées et utilisées pour offrir les services Internet. La figure ci-dessous présente un exemple d'architecture n-tiers, en l'occurrence, il s'agit ici d'une architecture 5-tiers:

- le premier tiers (le plus en haut) correspond aux clients finaux,
- le tiers suivant est celui du serveur Internet,
- vient ensuite le tiers des serveurs de médiation,
- le quatrième tiers correspond aux passerelles pervasives (qui sont, par exemple, enfouies dans des usines et qui embarquent des environnements d'exécution)
- enfin, le cinquième et dernier tiers est celui des équipements.

Figure 29. Exemple d'architecture n-tiers.

L'informatique orientée service (ou *Service-oriented Computing*, SOC) fournit plusieurs des propriétés nécessaires aux architectures n-tiers. Les technologies UPnP, OSGi, SCA et/ou les services web ont été utilisés dans ces architectures afin d'intégrer le *IT* et les applications industrielles [JS05]. Pour rappel, l'acronyme *IT* désigne l'informatique de gestion. Dans la figure précédente, cela correspond aux

tiers applications web et serveur Internet. Le terme application industrielle correspond, quant à lui, à l'informatique qui apparait dans les usines et donc aux tiers passerelles pervasives (qui contiennent chacune un environnement d'exécution) et équipements.

Dans notre contexte, nous nous sommes focalisés sur des passerelles pervasives qui embarquent, chacune, un environnement d'exécution à services basé sur la technologie OSGi. Ces environnements d'exécution exécutent des (implémentations de) services, rendent accessibles les équipements connectés à la passerelle, orchestrent les actions liées aux équipements et gèrent les interactions avec le *IT* (c'est-à-dire l'informatique de gestion). Rappelons que l'acronyme OSGi signifie *Open Service Gateway Initiative* et que la technologie OSGi définit un modèle minimal de composant orienté service, implémente le patron SOA manipulant des contrats de services basés sur des interfaces Java, définit aussi un environnement basic pour l'administration (et donc pour le déploiement) de ces composants, ainsi qu'un ensemble de services standards. L'utilisation d'un tel environnement d'exécution permet aux développeurs d'implémenter des contrats de services et des moyens d'interactions entre services hautement flexibles où équipements, services et interactions peuvent changer au fil du temps sans pour autant nécessiter le redémarrage (c'est-à-dire la désactivation et la réactivation) de l'environnement. Malheureusement, développer et déployer des services au-dessus de tels environnements est complexe et nécessite un haut niveau d'expertise. Typiquement, le déploiement de services, d'ensembles de services, et dans notre cas, d'applications à services est particulièrement difficile.

Dans le cadre de cette thèse, nous ciblons une architecture n-tiers et en particulier, nous distinguons un tiers correspondant aux passerelles pervasives et un tiers de plus haut niveau dans l'architecture qui peut être, par exemple, le tiers serveur Internet ou tout du moins un tiers de niveau équivalent. L'idée est de permettre à un administrateur, situé devant un serveur Internet (ou équivalent), de commander des actions de déploiement (ciblant, dans notre cas, des applications à services) destinées à être réalisées automatiquement par notre architecture de déploiement dans le tiers passerelles/environnements d'exécution. Le déploiement de telles applications à services est une des étapes nécessaires à la fourniture de services Internet de haut niveau aux clients finaux.

Nous allons maintenant présenter le contexte dans lequel les applications à services doivent être déployées. Pour ce faire, nous allons réutiliser l'exemple d'architecture n-tiers présentée ci-dessus.

Ainsi trois de ses cinq tiers entrent en jeu, à savoir, les tiers "médiation", "passerelles" et "équipements". Le tiers "passerelles" est destiné à interagir avec les tiers "médiation" et "équipements". Il contient des passerelles qui embarquent, elles-mêmes, des environnements d'exécution sur lesquels sont déployées des unités de déploiement qui sont, dans notre cas, des implémentations de services (représentées par un "S" dans la figure ci-dessous) et sur lesquels les applications à services sont destinées à être déployées.

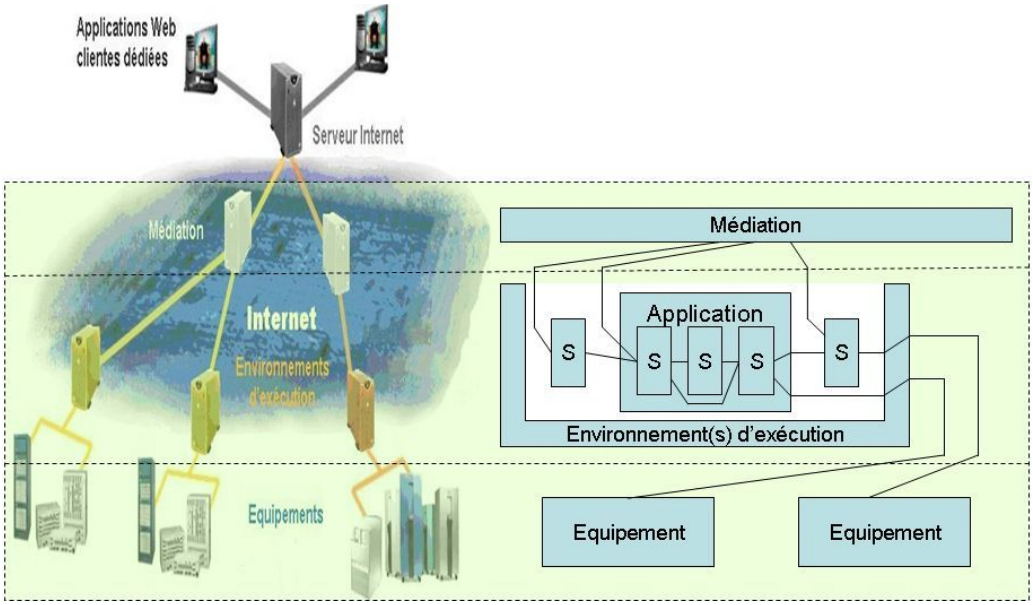


Figure 30. *Big Picture*.

Les interactions entre les tiers "passerelles" et "équipements" peuvent se faire via les environnements d'exécution, mais aussi directement via des services présents dans les environnements. Les interactions entre les couches "passerelles" et "médiation" se font, elles, directement via des services déployés dans les environnements.

Dans cette architecture n-tiers, les applications à services collectent les données provenant des (services représentant des) équipements, transforment ces données et envoient régulièrement les données transformées au tiers de médiation.

Enfin, il est important de noter que, dans cette thèse, nous nous intéressons au déploiement d'applications à services. Des travaux relatifs aux interactions entre les tiers peuvent être trouvés dans [BDE+06] et [EBL+08].

2.2 Passerelles pervasives et environnements OSGi

Dans cette sous-section, nous allons présenter et préciser quelles sont les contraintes relatives à l'utilisation de passerelles pervasives contenant des environnements embarqués/réactifs basés sur la technologie OSGi.

En ce qui concerne le déploiement, l'utilisation de passerelles pervasives définit plusieurs contraintes, à savoir :

- il n'est, en général, pas possible de stocker et de persister des informations dans le système de fichiers des passerelles. Le système de fichiers n'est destiné qu'à l'environnement d'exécution (et transitivement aux implémentations de services déployées dans l'environnement),
- les passerelles embarquent un unique environnement d'exécution qui est basé soit sur OSGi R3, soit sur OSGi R4.1,
- les passerelles sont connues (via leur adresse IP) et gérées par l'administrateur
- enfin, du fait que les passerelles sont enfouies dans les locaux des sites cibles (usines, maisons, etc.), la commande du déploiement ou le déploiement doit être effectué à distance. En effet, il n'est pas envisageable que l'administrateur se déplace auprès de chaque passerelle.

Les environnements orientés service que nous ciblons sont ainsi basés sur la technologie OSGi. Ils distinguent trois niveaux d'abstraction autour des services :

- le niveau service correspond aux contrats des services (nous l'appelons aussi niveau modèle de service),

- le niveau implémentation de services se réfère aux entités logicielles packagées implémentant/réalisant les contrats/modèles de services. Nous le nommons aussi niveau unité de déploiement, car les unités de déploiement que nous manipulons sont des implémentations de services packagées.
- enfin, le dernier niveau correspond aux instances d'implémentations de services.

L'utilisation de tels environnements d'exécution introduit plusieurs contraintes techniques (liées à la technologie OSGi) qui influencent le déploiement d'une manière générale et le déploiement d'applications à services en particulier. En effet :

- Chaque environnement possède son propre registre d'implémentations de services nommé *BundleContext*. Il liste et permet d'accéder aux implémentations de services déployées dans l'environnement d'exécution via, par exemple, l'utilisation de la méthode *getBundles()*.
- Chaque environnement possède aussi un *broker* SOA (ou registre SOA) chargé de maintenir des références de services. Un service doit être activé pour apparaitre dans le *broker* SOA. En effet, l'activation d'un service inclut le fait que le service s'enregistre dans le *broker*. Il est alors accessible en utilisant le *BundleContext* présenté ci-dessus via, par exemple, la méthode *getAllServiceReferences(...)*.
- OSGi est une technologie qui promeut la vision (de l'approche à service) nommée "composant orienté service" et qui se base sur la notion de contrat de service. Un contrat de service OSGi est constitué d'une interface JAVA qui peut ensuite être décorée de propriétés lors de l'enregistrement du (contrat de) service dans le *broker* SOA. Un contrat de service est lié à un service (requis ou fourni) d'un composant.
- Les interactions entre les services mettent en jeu, d'un côté, un fournisseur de service et d'un autre côté, un demandeur de service. Ces interactions reposent sur le mécanisme de liaison retardée offert par OSGi (la liaison retardée étant une des propriétés caractéristiques des services).
- Les interactions établies via OSGi peuvent correspondre à trois patrons d'interaction distincts, à savoir : Client - Serveur, Publieur - Souscripteur et Producteur - Consommateur.

En complément des contraintes liées à la technologie OSGi présentées ci-dessus, nous allons maintenant rappeler quels sont les deux mécanismes de composition de services offerts par la technologie OSGi via l'utilisation de composants orientés service :

- Le premier mécanisme de composition de services porte sur la composition de fournisseurs de services avec des demandeurs de services. L'idée est de permettre la liaison de fournisseurs et de demandeurs de services. Dans le cas d'OSGi, une telle liaison est établie en attachant fournisseurs et demandeurs à un même composant. Ainsi, un composant orienté service peut spécifier qu'ils possèdent zéro, un ou plusieurs fournisseurs de services et zéro, un ou plusieurs demandeurs de services.
- Le second mécanisme de composition de services porte sur la composition de demandeurs de services avec des fournisseurs de services (où demandeurs et fournisseurs n'appartiennent pas au même composant). Dans le cadre des services en général et dans le cas de la technologie OSGi en particulier, une telle composition est exprimée via un contrat de service et la composition est effectivement mise en place en appliquant le patron SOA. Le contrat de service est utilisé, d'un côté par le fournisseur afin de spécifier le service qu'il offre et d'un autre côté par le demandeur afin de spécifier le service qu'il requiert. Concernant, maintenant uniquement la technologie OSGi, l'approche suivie afin d'établir effectivement une interaction entre un demandeur et un fournisseur de service est une approche commerçante (appelée aussi approche *trading*). Cela signifie qu'un demandeur de service requérant un contrat de service va rechercher dans le broker SOA tous les fournisseurs de services qui fournissent ce contrat ; ensuite, à partir de ces fournisseurs, il va choisir, en fonction de sa propre algorithmique, de sa propre stratégie, quel est le fournisseur ou les fournisseurs qu'il retient. En guise de remarque, cette approche a été poussée à l'extrême dans les travaux autour de *ServiceBinder*, où l'idée était de maximiser, automatiquement et de manière transparente, le nombre de fournisseurs mis à disposition pour un demandeur donné [CH04].

Enfin, OSGi offre un mécanisme de mise à jour statique pour les unités de déploiement, c'est-à-dire pour les *bundles*. Ce mécanisme s'applique sur le *bundle* cible de la mise à jour, ce dernier devant préciser l'*URL* à laquelle le "nouveau" *bundle* peut être téléchargé. Concrètement, les *bundles* OSGi contiennent obligatoirement un fichier manifest.mf qui peut optionnellement contenir l'attribut standard nommé *Bundle-UpdateLocation*, la valeur associée à cet attribut correspond à l'*URL* où la mise à jour peut être trouvée.

Ce mécanisme implique qu'un bundle doit contenir, dès sa publication, l'*URL* à laquelle sa mise à jour sera publiée. Il implique aussi qu'un *bundle* ne pourra être mis à jour que par le "nouveau" *bundle* se trouvant à l'*URL* spécifiée. Cela peut être un avantage pour le vendeur du *bundle*, puisqu'ainsi, il s'assure que les mises à jour du *bundle* qu'il a produit proviendront de l'*URL* qu'il a spécifiée dans le *bundle* et dont il maitrise, *a priori*, le contenu. Cependant, un tel mécanisme est aussi une limite, puisqu'il va à l'encontre de la propriété de substitution des services. Ainsi, un administrateur qui souhaite effectuer la mise à jour statique d'un *bundle* (c'est-à-dire d'une implémentation de service packagée), par un autre *bundle* de son choix, ne peut le faire en utilisant le mécanisme fourni par OSGi.

3. PROBLÉMATIQUE

Notre problématique porte sur le déploiement d'applications à services sur des environnements d'exécution à services. Nous cherchons à rendre possible le déploiement d'applications à services et donc à concevoir et fournir une architecture et un système de déploiement utilisables par un administrateur souhaitant déployer des applications à services sur les environnements d'exécution qu'il administre.

L'état des lieux, le contexte et cette problématique font apparaitre plusieurs défis :

- Le premier défi vise la définition d'une notion d'application à services permettant le "déploiement au plus près", la production d'un méta-modèle correspondant à cette définition et l'expression effective de telles applications (c'est-à-dire la création de modèles d'application à services dissociés de toute implémentation).
- Le second défi cible l'adaptation de la définition du déploiement que nous estimons être la plus complète et la plus cohérente (c'est-à-dire celle de l'université du Colorado [HHC+98]) avec pour but avoué de préciser la définition du déploiement pour l'approche à service et pour les services et applications à services.
- Le troisième défi vise l'intégration notre système de déploiement dans le contexte global présenté.
- Le quatrième défi cible le déploiement de modèles d'application à services. Nous verrons que nous nous sommes focalisés sur les activités d'installation, d'activation, de désactivation, de désinstallation d'applications à services et la mise à jour statique d'une manière générale. Ce quatrième défi se décompose en quatre sous-défis :
    - Le premier est relatif au déploiement de modèles et se focalise sur la résolution de dépendances (c'est-à-dire sur la validation du modèle de l'application et la sélection des implémentations de services packagées qui le réaliseront effectivement) et sur l'existence et le maintien d'un "lien" entre le niveau "réalité" (c'est-à-dire les implémentations packagées / unités de déploiement déployées sur les environnements) et le niveau "modèle" (c'est-à-dire les modèles d'application à services).
    - Le second concerne le déploiement dans l'approche à service et, pour ce faire, cible l'ordonnancement des activations et désactivations des implémentations de services, ainsi que la gestion du niveau instance d'implémentation de service et l'introduction d'une approche *Naming* pour la mise en place des compositions entre demandeurs et fournisseurs dans une même application.
    - Le sous-défi suivant cible l'activité de mise à jour statique.
    - Enfin, le dernier sous-défi vise à proposer un mécanisme transactionnel au niveau du déploiement des applications. Pour être tout à fait précis, le système de déploiement d'applications doit rattraper les erreurs qui peuvent apparaitre lors de l'installation, de l'activation, de la désactivation, de la mise à jour statique ou de la désinstallation d'implémentations de services. Rattraper une telle erreur signifie éviter un crash du système et revenir (*rollback*) sur les actions déjà effectuées lors de l'activité de déploiement en cours.
- Le cinquième défi vise l'étude et la gestion du partage d'implémentations de services, de l'isolation entre applications et du partage d'instances d'implémentations de services.
- L'avant dernier défi cible la minimisation de la taille des applications à services déployées (nous parlons ici en espace disque) et du nombre d'implémentations de services packagées (c'est-à-dire d'unités de déploiement) déployées dans tout environnement.
- Enfin, le septième et dernier défi se découpe en deux sous-défis :
    - Le premier sous-défi porte sur le déploiement de modèles d'application à services sur des environnements d'exécution à services.
    - Le second se concentre sur l'exécution d'un ensemble d'activités de déploiement concernant des modèles d'application à services (c'est-à-dire sur l'exécution de plusieurs activités de déploiement les unes à la suite des autres). Notre objectif est la proposition d'un algorithme d'exécution plus performant que l'algorithme d'exécution séquentiel trivial.

Enfin, en complément, les défis de notre thèse proviennent :

- de l'état des lieux sur les applications à services dans les visions actuelles de l'approche à service,
- de l'état des lieux autour du "déploiement au plus près" et de la montée en abstraction du déploiement.
- de l'état des lieux sur la définition du déploiement (c'est-à-dire sur l'absence de définition consensuelle) et de la définition du déploiement que nous avons adaptée,
- des (modèles d') applications à services que nous manipulons,
- du déploiement de (plusieurs) modèles d'application sur un et des environnements d'exécution,
- et, enfin, du contexte pervasif des passerelles et embarqué des environnements d'exécution.

IV. PROPOSITION

Dans ce chapitre, nous allons présenter les différentes propositions que nous avons réalisées afin de répondre à la problématique (et aux défis) du déploiement d'applications à services dans des environnements d'exécution à services en suivant une approche dirigée par les modèles et cela dans notre contexte et suivant notre état des lieux. La structure de ce chapitre "proposition" se base sur la structure des défis que nous avons détaillés dans le chapitre précédent.

1. NOTRE APPROCHE

Dans cette thèse, nous proposons un système de déploiement dirigé par les modèles. Ce système, nommé DMSA pour *Deployment Manager for Services Applications*, se base sur des modèles d'application à services, sur le modèle global associé à chaque environnement d'exécution ciblé et sur des modèles implémentations de service (c'est-à-dire d'unité de déploiement) pour réaliser/exécuter des activités de déploiement.

Notre objectif est d'identifier et d'isoler les informations relatives au déploiement d'applications à services et aux propriétés de déploiement attendues afin d'offrir un système, dirigé par les modèles issus des méta-modèles définis, qui automatise le déploiement (c'est-à-dire qui joue le rôle de déployeur). Pour ce faire, nous avons défini un méta-modèle pour les applications à services (il permet de spécifier une vue "déploiement" des applications à services), un méta-modèle pour les environnements et un méta-modèle pour les implémentations de services (c'est-à-dire pour les unités de déploiement).

Nous utilisons l'approche dirigée par les modèles afin de bénéficier de son haut niveau d'abstraction et du découplage technologique qu'elle permet d'atteindre (c'est-à-dire un découplage entre le niveau modèle et le niveau "réalité"). En effet, dans le cadre du déploiement nous ne souhaitions plus manipuler des unités de déploiement constituées d'un descripteur et de l'implémentation décrite, mais uniquement les descripteurs (c'est-à-dire des modèles de services). Notre idée est de nous abstraire des implémentations afin de pouvoir tirer parti de la propriété de substituabilité des services. En guise de rappel, cette propriété permet de substituer une implémentation de service utilisée pour réaliser un contrat de service par une autre implémentation de service de contrat équivalent. En conséquence, les modèles d'application spécifiés et passés en paramètre au DMSA n'ont aucun lien avec une quelconque implémentation de service, le découplage entre les niveaux modèle et "réalité" est total. L'objectif final est d'appréhender au mieux le fait que les environnements d'exécutions ciblés relèvent du domaine de l'embarqué/réactif et le fait qu'ils sont hétérogènes et/ou qu'ils contiennent des implémentations de services hétérogènes.

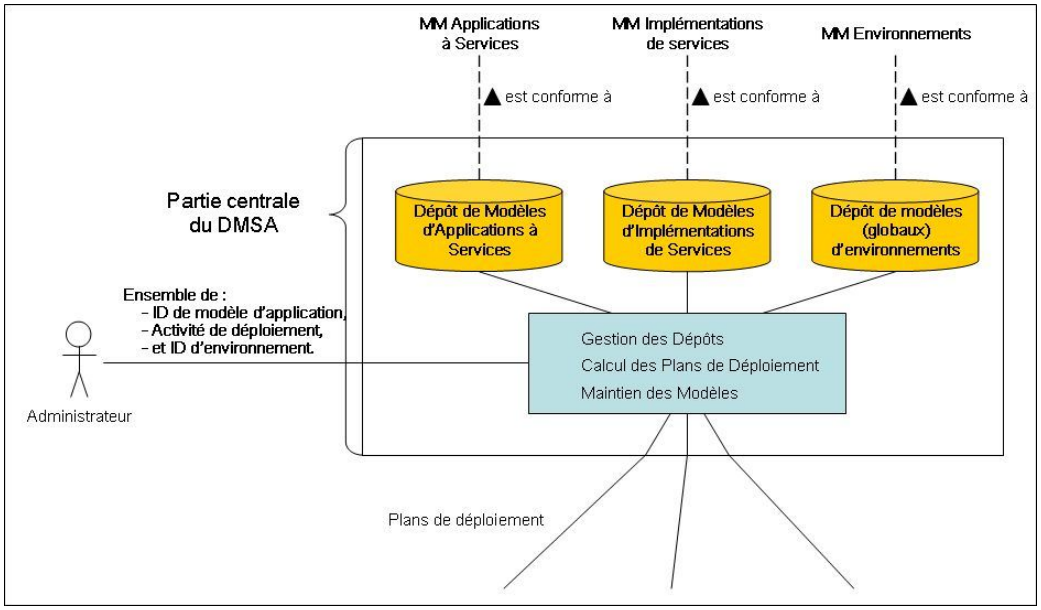


Figure 31. Schéma de notre approche (*DMSA's Main Part*).

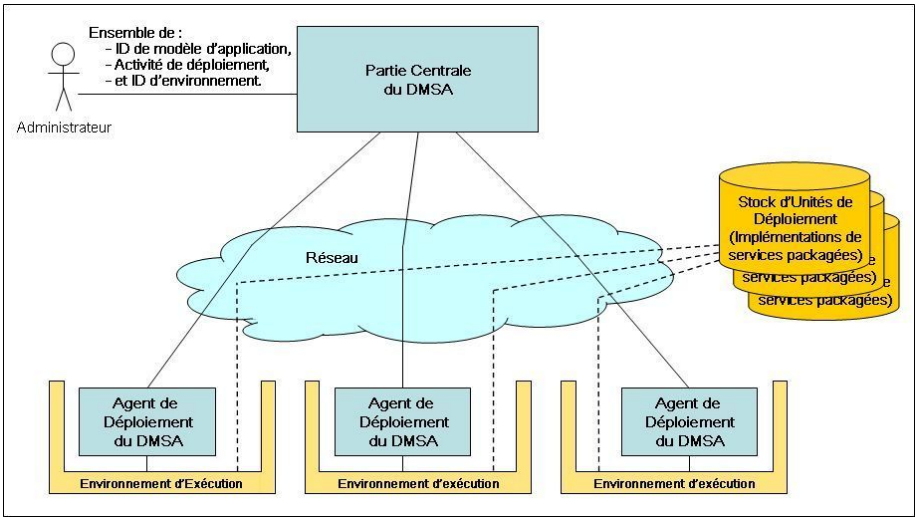


Figure 32. Schéma de notre approche (*DMSA's Deployment Agents*).

L'élément central du DMSA est le modèle global qu'il construit et maintient pour chaque environnement d'exécution géré. Chaque déploiement est tout d'abord appréhendé par rapport aux modèles d'application ciblés et aux modèles globaux des environnements ciblés, avant d'être traduit en instructions de déploiement aux niveaux implémentations de services et instances, puis exécuté dans chaque environnement ciblé, pour être enfin répercuté au niveau du modèle global de chaque environnement.

Plus simplement, chaque déploiement passé en paramètre au DMSA est appréhendé comme une instruction visant à modifier le modèle global de l'environnement cible. Ainsi, si l'instruction est sensée par rapport au modèle global de l'environnement ciblé, alors un plan de déploiement correspondant à cette instruction est généré et exécuté dans l'environnement et le modèle global de l'environnement est mis à jour en conséquence, sinon une erreur est retournée et le modèle global reste inchangé. Si l'exécution du plan de déploiement échoue, le modèle global de l'environnement reste inchangé et le DMSA tente un retour en arrière sur les éléments du plan de déploiement qui ont été exécutés, afin d'assurer la cohérence entre le modèle global associé à l'environnement et l'état "réel" de l'environnement.

Concrètement, le DMSA prend en entrée :

- des identifiants de modèles d'application à services précédemment enregistrés dans le dépôt de modèles d'application du DMSA,
- le dépôt de modèles d'implémentations de services du DMSA ainsi que les dépôts d'implémentations de services correspondants,
- des identifiants d'environnements d'exécution à services précédemment enregistrés auprès du gestionnaire des modèles globaux des environnements du DMSA
- et un ensemble d'activités de déploiement.

Lors de chaque déploiement, le DMSA reçoit des instructions de l'administrateur qui gère les environnements d'exécutions cibles. Il vérifie si le déploiement a un sens par rapport au modèle de chaque environnement ciblé (c'est-à-dire si le modèle global de chaque environnement peut être modifié par le déploiement donné) et, le cas échéant, il génère le plan de déploiement correspondant (afin de modifier la "réalité" de l'environnement).

Chaque plan de déploiement est ensuite envoyé aux dits environnements pour être exécuté. Le résultat de chaque exécution est ensuite retourné à la partie centrale du DMSA qui modifie le modèle global associé à chaque environnement ciblé et qui informe l'administrateur du résultat de l'exécution de chaque déploiement.

Ensuite, il est important de noter que notre choix d'une approche dirigée par les modèles est la suite logique de la montée en abstraction qu'il est possible d'observer en génie logiciel. Ainsi, comme

nous l'avons présenté dans notre état de l'art, les paradigmes de développement conçus et mis en avant au fil des ans ont toujours cherché à gagner en abstraction par rapport à leurs prédécesseurs, preuve en est, l'introduction des objets pour faire suite aux modules, puis des composants, des services et à l'heure actuelle des modèles. De son côté, le déploiement logiciel n'échappe pas à ce mouvement global du génie logiciel, ainsi, les déployeurs qui déployaient initialement des implémentations monolithiques (tels les archives JAVA .jar [JAR] ou les *packages* [ACR08]), ont cherché ensuite à déployer des logiciels décrits à l'aide de descripteurs de déploiement [SUN03b] [BCS03], puis à déployer des logiciels en ne se basant que sur leur seul identifiant [CLM05] [.Net06] [MAV08] et enfin, à déployer des modèles qui permettent un découplage envers les implémentations et une bonne abstraction puisque seules les informations nécessaires au processus de déploiement entrent en jeu et sont manipulées, c'est par exemple le cas pour [OMG06] (où un ensemble d'implémentations est mis à disposition pour un modèle de composant donné) et dans notre approche (où les modèles manipulés sont dissociés du niveau implémentation et où le choix, la sélection des implémentations revient au déploiement).

La spécification D&C de l'OMG a été la première spécification à introduire ce type d'approche dans le cadre du déploiement [OMG06]. Son objectif était la création d'un modèle générique pour le déploiement et la configuration de logiciels (distribués). Malheureusement, cette volonté de généricité a rendu l'approche de l'OMG particulièrement complexe et difficile à utiliser et à mettre en place. De plus, les bénéfices apportés par l'utilisation de cette spécification sont discutables puisque, par exemple, le processus de déploiement ne couvre pas les activités de désinstallation et de mises à jour et puisque le modèle générique de l'OMG est particulièrement difficile à appréhender et à manipuler.

Afin d'éviter les écueils rencontrés par l'OMG, nous nous sommes concentrés sur l'identification, la conception et la mise en place d'un ensemble minimal de modèles répondant à notre problématique et à ses contraintes, c'est-à-dire permettant le déploiement d'applications à services centralisées dans des environnements d'exécution à services.

Pour mettre en œuvre notre approche, il nous a été nécessaire de :

- Définir un méta modèle pour les applications à services dans notre contexte de "déploiement au plus près" (c'est-à-dire à définir une vue déploiement pour les applications à services satisfaisant notre contexte et les propriétés de déploiement que nous ciblons),
- Utiliser ce méta-modèle pour modéliser les applications, les implémentations de services (c'est-à-dire les unités de déploiement) et les environnements d'exécution.
- Définir le déploiement pour les applications à services dans notre contexte.

Il nous a aussi été nécessaire :

- D'appréhender ce déploiement, c'est-à-dire permettre l'installation, l'activation, la désactivation, la mise à jour statique et la désinstallation de modèles d'application sur un environnement d'exécution à services en agissant sur ces dits modèles et sur le modèle global de l'environnement.
- De gérer le partage d'implémentations de services et d'instances, ainsi que l'isolation entre les modèles d'application déployés.
- D'appréhender le contexte embarqué/réactif des environnements d'exécution à services que nous ciblons, c'est-à-dire maximiser le partage des implémentations de services et des instances d'implémentations de services dès que cela est possible.
- De permettre le déploiement sur plusieurs environnements, ainsi que l'exécution de listes d'activités de déploiement.

Enfin, l'utilisation d'une approche dirigée par les modèles dans le cadre de cette thèse nous fait manipuler quatre niveaux distincts, à savoir, le niveau modèle d'application, le niveau modèle de service, le niveau implémentation de service (c'est-à-dire le niveau unité de déploiement dans notre objectif de déploiement "au plus près") et le niveau instance d'implémentation de service. La figure ci-dessous schématise ces quatre niveaux.

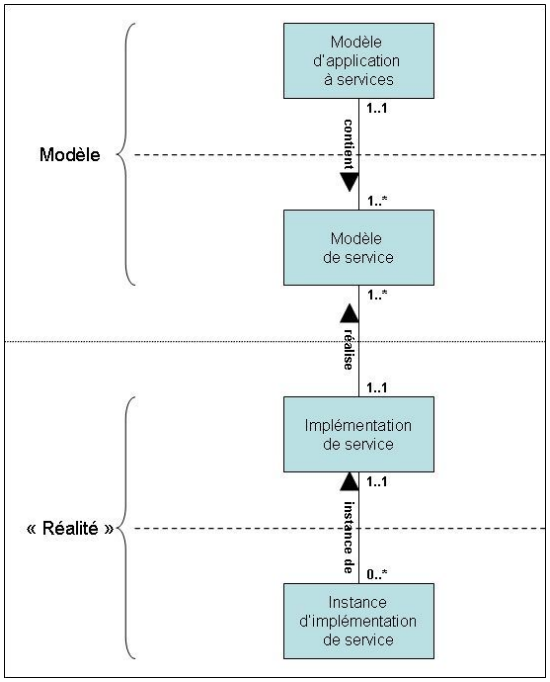


Figure 33. Schéma des quatre niveaux manipulés.

Pour rappel, chaque modèle d'application est identifié de manière unique par le DMSA et chaque modèle de service est identifié de manière unique au sein de son application. Cela explique la cardinalité "1..1" associée à la classe "Modèle d'application à services" pour l'association entre les classes "Modèle d'application à services" et "Modèle de service". Dans une application à service, chaque contrat de service fait partie d'un modèle de service. Si plusieurs modèles de services ont des contrats de services similaires, alors ces modèles de services peuvent être réalisés par la même implémentation de service ; cela explique la cardinalité "1..*" sur la classe "Modèle de service" pour l'association entre les classes "Modèle de service" et "Implémentation de service". Exprimé autrement, si l'attribut "ServiceIdInTheServicesApplication est enlevé d'un modèle de service apparaissant dans un modèle d'application, alors le modèle de service "tronqué" (qui se trouve être le contrat de service) peut tout à fait apparaitre dans plusieurs applications, voir même plusieurs fois dans la même application.

Dans la suite de ce chapitre, nous présentons notre méta-modèle pour les applications à services, la définition du déploiement, ainsi que l'architecture de notre système de déploiement ; nos propositions autour du déploiement d'un modèle d'application sur un environnement et autour du déploiement de modèles d'application, puis notre appréhension du contexte embarqué des environnements et nos propositions autour de l'exécution (de listes) d'activités de déploiement sur des environnements. Enfin, la dernière section synthétise le contenu de notre proposition.

2. MÉTA-MODÈLE DES APPLICATIONS À SERVICES

La définition d'applications basées sur l'approche à service fait apparaitre trois besoins. Le premier concerne la composition entre fournisseur et demandeur. Le second concerne la composition entre demandeur et fournisseur. Le troisième et dernier souligne le fait qu'une application doit expliciter d'une part les fournisseurs qu'elle offre et d'autre part les demandeurs qu'elle requière. Enfin, bien évidemment, la notion d'application ainsi définie doit permettre le "déploiement au plus près".

Pour rappel, nous utilisons l'expression "déploiement au plus près" pour désigner un déploiement dans lequel les unités de développement et de déploiement ont le même grain ; cela signifie que toute unité de déploiement ne contient qu'une et une seule unité de développement. Le contraire du "déploiement au plus près" est le déploiement d'un logiciel monolithique (c'est-à-dire d'un logiciel dont l'ensemble des unités de déploiement est packagé sous la forme d'une et une seule unité de déploiement).

2.1 Notation

L'approche à service, d'une manière générale, est basée sur trois éléments de base, à savoir, le service fourni (ou fournisseur de service), le service requis (ou demandeur de service) et le registre de services (fournis). Ces trois éléments de base apparaissent très clairement dans le patron SOA. Dans cette section, nous allons nous intéresser plus particulièrement au service fourni et au service requis. La figure ci-dessous présente la notation que nous utilisons pour les représenter (cette notation est issue de celle introduite par la technologie *OpenWings* [Gen01])

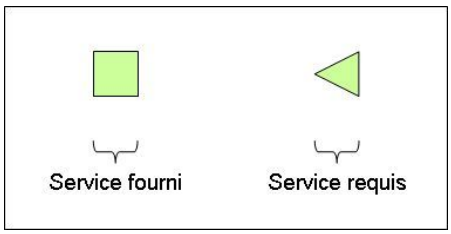


Figure 34. Notation pour les fournisseurs et demandeurs de service.

2.2 Composition entre fournisseur et demandeur

Nous avons répondu au premier besoin en nous basant sur la vision composant orienté service. La figure ci-dessous présente, à travers trois exemples, la notation que nous utilisons pour représenter des composants orientés service.

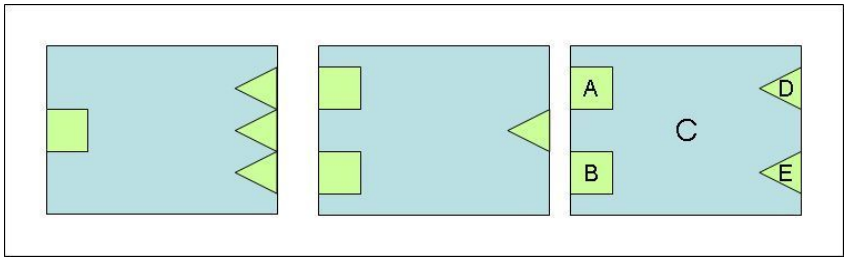


Figure 35. Notation pour les composants orientés service.

Dans la figure ci-dessus, l'exemple de composant orienté service le plus à droite (identifié par la lettre "C") possède deux fournisseurs de services (notés "A" et "B") et deux demandeurs de services (notés "D" et "E").

La vision composant orienté service nous satisfait pour exprimer la composition entre fournisseur et demandeur de service, cependant, elle ne permet pas d'atteindre notre objectif de "déploiement au plus près". En effet, un composant peut proposer un ensemble de fournisseurs de

services. De plus, dans le cadre de la vision composant orienté service, il est possible que seule une partie de ces fournisseurs (par exemple un seul ou plusieurs) soit effectivement utilisée à un moment donné. Or, les fournisseurs de services non utilisés occupent de l'espace de "disque" inutilement. Dans l'exemple précédent de composant orienté service "C", un tel cas peut apparaitre, si, par exemple, le fournisseur "A" est utilisé et si le fournisseur "B" ne l'est pas.
De façon annexe et toujours en reprenant l'exemple du composant ("C") précédent, nous pouvons nous interroger sur les perturbations qui peuvent exister au niveau des données fournies respectivement par les fournisseurs "A" et "B". Ainsi, qu'en est il, si, par exemple, "A" est utilisé sans "B" ou si "B" est utilisé sans "A" et si ensuite, "A" et "B" sont, tous les deux, utilisés en même temps ?

Afin d'atteindre notre objectif de "déploiement au plus près", nous avons donc introduit une contrainte sur la vision composant orienté service. Cette contrainte impose, qu'afin de permettre le/un "déploiement au plus près", tout composant orienté service ne doit posséder qu'un unique fournisseur de service. Ainsi, tout composant possède un unique fournisseur et peut posséder zéro, un ou plusieurs demandeurs. La figure ci-dessous présente deux exemples de composants orientés services respectant cette contrainte et permettant donc d'atteindre la propriété de "déploiement au plus près".

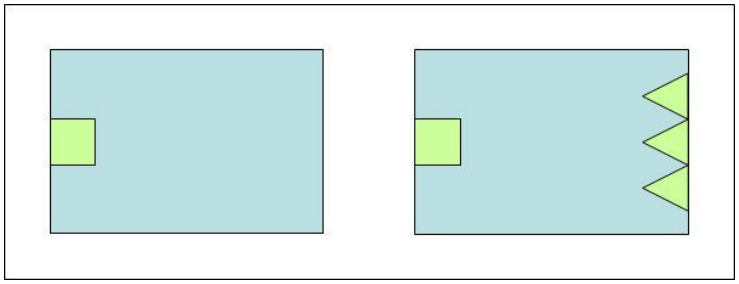


Figure 36. Composants orientés service contraints supportant le déploiement au plus près.

Ici, il est très important de souligner que le modèle d'entité logicielle ainsi défini, à savoir le "composant orienté service contraint" (COSC), n'est pas une révolution. En effet, il est très proche du modèle des *Enterprise Java Bean* (EJB), à ceci près que, contrairement aux EJB, il spécifie explicitement ses requis. Pour être tout à fait exact, un EJB fournit deux interfaces, une pour les accès locaux (c'est-à-dire pour les accès depuis l'intérieur du conteneur) et une autre pour les accès à distance, cependant, d'un point de vue plus abstrait, ces deux interfaces peuvent être vues comme une unique interface où chaque méthode est annotée par un *tag* (*@Remote* et/ou *@Local*). Toujours d'un point de vue abstrait, ce type d'entité logicielle est aussi proche des modèles objet et méthode. Pour faire le parallèle entre la vision composant orienté service et le modèle méthode, voyons une méthode comme un composant, voyons ses paramètres comme des services requis et voyons les données qu'elle retourne comme des services fournis ; imaginons nous invoquer une méthode (sans maitriser ses paramètres d'entrée) pour n'utiliser qu'une partie des données qu'elle va effectivement nous retourner ?

2.3 Service et contrat de service

Ensuite, il est important de noter que la définition du contrat d'un service est un élément crucial. Par exemple, dans le cadre d'une interaction entre un demandeur et un fournisseur de service (comme dans la figure ci-dessous) le contrat de service explicite les éléments fonctionnels et non fonctionnels sur lesquels le demandeur et le fournisseur se sont mis d'accord. Dans la technologie OSGi, le contrat est typiquement une interface Java.

Maintenant, dans le cadre d'une interaction entre un demandeur d'un composant et un fournisseur d'un autre composant au sein d'une application où l'architecture de l'application est importante, si le contrat de service utilisé se contente d'expliciter les éléments fonctionnels et non fonctionnels sur lesquels le demandeur et le fournisseur se sont mis d'accord, alors il est possible qu'une application d'architecture donnée (soit "C1", "C2" et "C3" à t0) soit corrompue lors d'une substitution du composant "C2" par le composant "C4" (ci-dessous à t1). La figure ci-après illustre un tel cas. Pour rappel, dans la technologie OSGi, chaque demandeur de service choisit, selon sa propre stratégie/algorithmique, le fournisseur de service qu'il va utiliser.

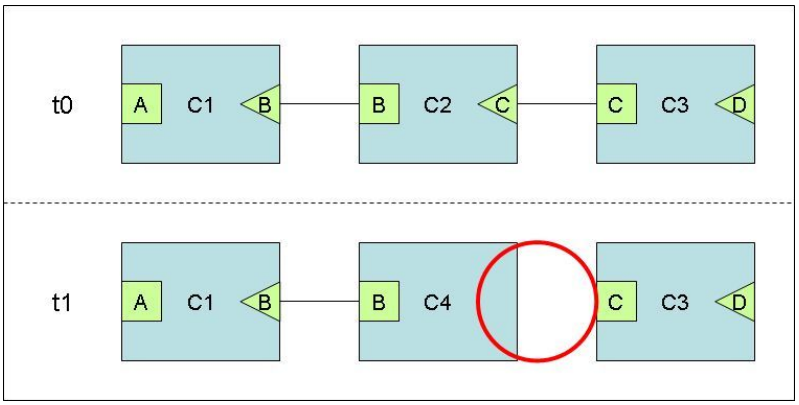


Figure 37. De l'importance du contrat de service dans les applications à services.

Ainsi, dans le cadre d'applications à services basées sur la vision composant orienté service, nous utiliserons le terme "service" pour désigner un "composant orienté service contraint" (COSC). Nous utiliserons aussi le terme "contrat de service" pour désigner l'ensemble formé par le contrat/spécification du fournisseur et par les contrats/spécifications des demandeurs appartenant au COSC, ainsi que par les informations propres au COSC lui-même. Par exemple, dans la figure ci-dessus, nous désignons l'entité logicielle "C2" comme étant le "service" d'identifiant "C2" dont le "contrat de service" est composé par le fournisseur (identifié par) "B" et le demandeur (identifié par) "C".

Cette translation de vocabulaire s'explique par le fait que le service réellement mis en jeu est, dans l'exemple ci-dessus, le fournisseur "B" appartenant à un composant (en l'occurrence "C2") possédant le demandeur "C" et non juste un fournisseur "B". Pour remarque, l'information concernant l'interaction entre les services "C1" et "C2" sera exprimée, non pas dans le contrat du service, mais au niveau de l'application.

Dans la suite de ce document, nous nous plaçons dans le cadre des applications à services. En conséquence, nous utiliserons les termes "service" et "contrat de service" comme expliqué ci-dessus.

La figure ci-dessous représente le méta-modèle succinct des services (c'est-à-dire des COSC) que nous manipulons.

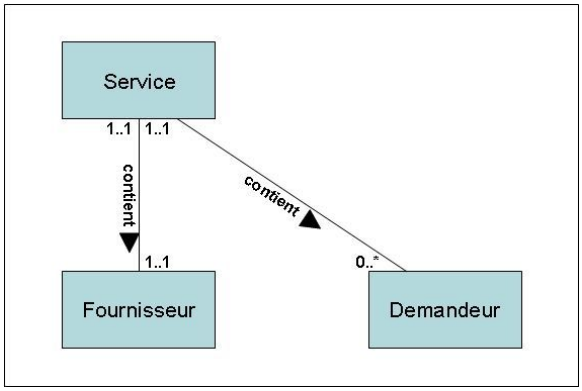


Figure 38. Méta-modèle succinct pour les services.

Chaque fournisseur et chaque demandeur est identifié de manière unique dans un service.

Ensuite, afin de garantir la propriété de substitution que nous avons identifiée comme étant une des propriétés fondamentales de l'approche à service, nous faisons l'hypothèse que l'implémentation d'un service respecte le contrat du service en fournissant obligatoirement le fournisseur défini dans le contrat du service et en utilisant systématiquement les possibles demandeurs définis dans ce même contrat. La figure ci-dessous donne deux exemples de services que nous manipulons et représente les "liens" qui existent entre l'implémentation du service, le fournisseur et les possibles demandeurs définis dans les contrats de service respectifs.

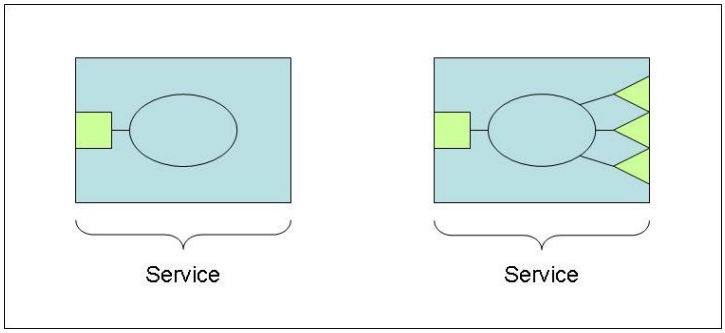


Figure 39. Exemples de services manipulés.

2.4 Composition entre demandeur et fournisseur

Nous allons maintenant présenter notre proposition autour du besoin de composition entre demandeur et fournisseur. Une première possibilité s'offrait à nous pour répondre à ce besoin. Nous la qualifions de "Composant - Composite". Elle est constituée d'un niveau "composant" et d'un niveau "composite". Ainsi, la classe "composite" représente un assemblage de "composant" et hérite, elle-même, de la classe "composant". La figure ci-dessous modélise succinctement cette première possibilité.

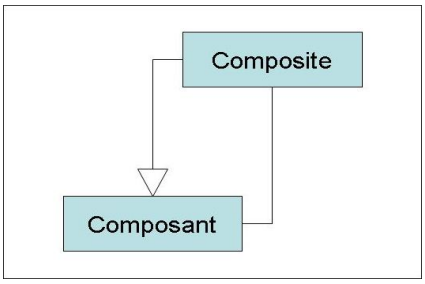


Figure 40. Schéma "Composant - Composite".

Cette première possibilité a pour avantage d'uniformiser le nom des activités de déploiement entre les niveaux composant et composite. D'un autre côté, elle laisse à penser que le déploiement d'un composite est aussi trivial que celui d'un composant (or l'ensemble de cette thèse montre l'inverse). De plus, elle ne permet pas de déployer au plus près. En effet, étant donné qu'un composite peut être manipulé comme un composant et étant donné qu'un composite peut posséder plusieurs fournisseurs de services (appartenant à un ou plusieurs composants), un composant peut posséder plusieurs fournisseurs et donc ne pas respecter notre proposition de "composant orienté service contraint" (et donc ne permet pas le déployer au plus près).

Notre proposition, que nous qualifions "Service - Application", est constituée d'un niveau service et d'un niveau application. La classe application représente une application qui peut être constituée de services et d'applications. Toute application doit contenir soit au moins deux services, soit au moins un service et une application, soit au moins deux applications. La figure ci-dessous représente notre proposition "Service - Application".

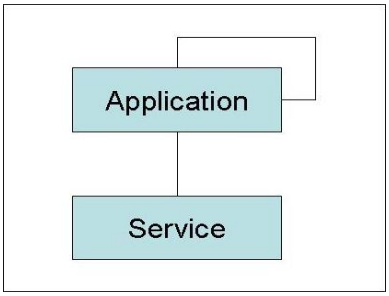


Figure 41. Schéma "Service - Application à services".

Les avantages de notre proposition sont qu'elle distingue clairement le déploiement d'un service du déploiement d'une application et qu'elle permet de déployer "au plus près". Elle possède cependant l'inconvénient d'imposer un modèle de déploiement qui doit être pris en compte dès le développement de l'application. Néanmoins, il est admis que tout informaticien choisit un modèle de développement en fonction des propriétés (de développement) qu'il souhaite obtenir et qu'il accepte par conséquent les contraintes de son modèle de développement ; de même, il faut prendre conscience que pour obtenir des propriétés (de déploiement), il faut choisir le modèle de déploiement qui offre de telles propriétés et accepter sa/ses contraintes !

Dans le cadre de cette thèse, nous nous sommes focalisés sur le déploiement d'applications à services constituées uniquement de services. Notre idée était de résoudre le premier pas de la récursion avant d'attaquer la preuve de la récursion elle-même.

Pour remarque, tout service (et application) est identifié de manière unique au sein d'une application. Et les modèles "Composant - Composite" et " Service - Application" ont une expressivité équivalente.

Nous venons de présenter le niveau "application" que nous avons utilisé dans cette thèse afin de déployer des applications à services. Nous allons maintenant présenter la classe "Interaction" grâce à laquelle les compositions entre demandeurs et fournisseurs sont spécifiées. Une interaction réalise la composition d'un demandeur avec un fournisseur. Nous avons souhaité faire apparaitre explicitement les interactions afin de permettre une spécification complète de l'architecture des applications à services. De plus, dans le cadre des activités de déploiement liées au changement de l'architecture d'une application s'exécutant ou non, il est indispensable de pouvoir manipuler les interactions en tant qu'objet premier. En outre, afin de tirer profit des conclusions autour des langages de descriptions d'architectures ("Architectural Description Language", ADL) [MT00] nous aurions souhaité faire apparaitre la classe "connecteur" comme objet premier, afin de pouvoir la manipuler (et donc de pouvoir exécuter des activités de déploiement portant sur les connecteurs eux-mêmes). Cependant, la technologie OSGi (qui est notre technologie cible) ne permet pas de manipuler les connecteurs en tant qu'objet premier (c'est-à-dire en tant qu'entité logicielle à part entière).

Au final, la classe "Interaction" permet de spécifier une liaison entre un demandeur appartenant à un composant et un fournisseur appartenant à un autre composant. L'information relative au type de communication utilisée pour l'interaction a été déportée, pour moitié, sur la partie du contrat de service du service possédant le fournisseur et, pour l'autre moitié, sur celle du service possédant le demandeur. Ainsi, un fournisseur explicitera qu'il joue le rôle de "Serveur", "Publieur" ou "Producteur" et un demandeur explicitera qu'il joue le rôle de "Client", "Souscripteur" ou "Consommateur". Toute interaction est identifiée de manière unique au sein d'une application.

Enfin, les informations concernant les fournisseurs offerts et les demandeurs requis par l'application sont notées sous la forme d'attributs propres à la classe "application".

2.5 Méta-modèle pour les applications à services

La figure ci-dessous présente la syntaxe abstraite du méta-modèle pour les applications à services que nous proposons dans le cadre de cette thèse.

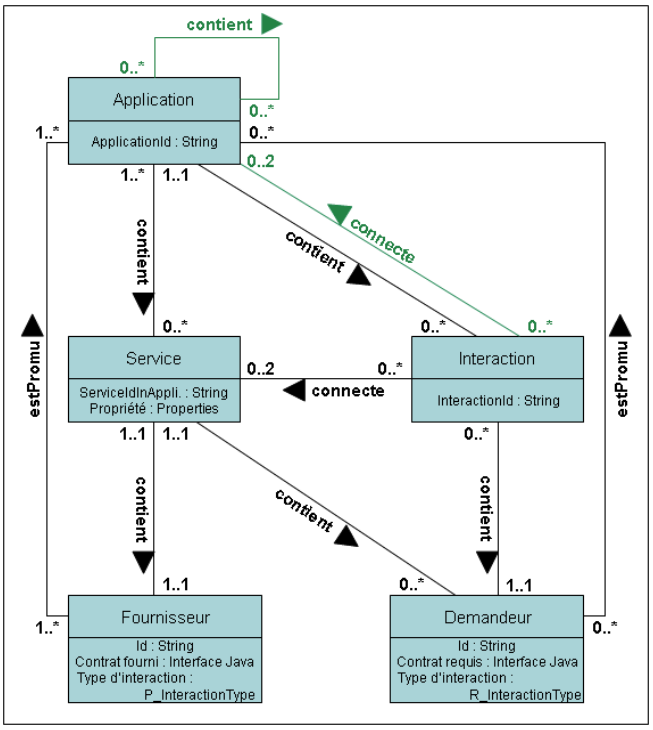


Figure 42. Syntaxe abstraite du méta-modèle pour les applications à services.

Il est à noter que deux contraintes accompagnent ce méta-modèle. La première contrainte, que nous avons déjà présentée, stipule que toute application doit contenir soit au moins un service, soit au moins un service et une application, soit au moins deux applications. La seconde contrainte cible la classe interaction et spécifie que toute interaction définie dans une application ("A") connecte soit deux services (c'est-à-dire un fournisseur et un demandeur), soit un service et une application (contenue par "A"), soit deux applications (contenues par "A").

Dans le cadre de cette thèse, nous nous focalisons sur la résolution du cas de base et sur les classes et associations notées en noir dans la figure ci-dessus.

La figure ci-dessous présente la notation globale que nous utilisons pour représenter les applications à services, les services et les interactions entre services. Cette notation doit être considérée comme la syntaxe (graphique) concrète du méta-modèle pour les applications à services.

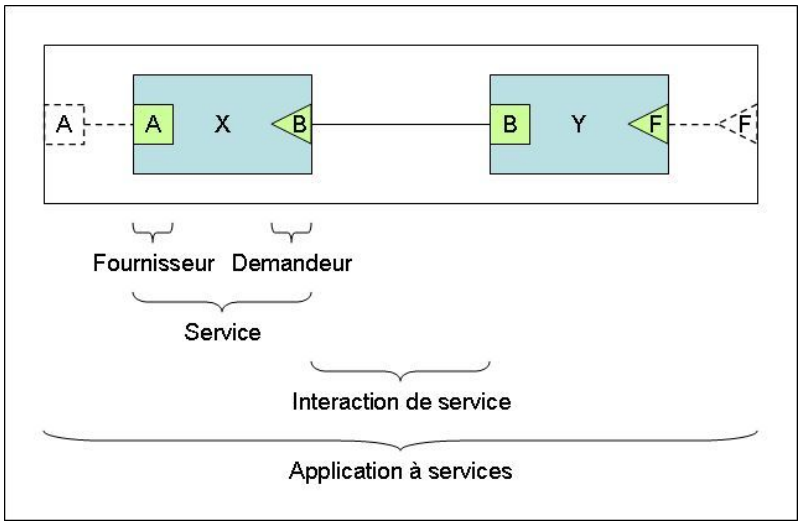


Figure 43. Notation de représentation des applications à services, services et interactions.

En complément, la figure ci-dessous représente la syntaxe abstraite d'une partie du modèle de l'application CEPC que nous avons utilisé pour valider nos travaux. Les abréviations "ACS", "AETS" et "DS" signifient respectivement "AffichageConsoleService", "AggregationEtTransformationService" et "DateService" ; afin d'augmenter la lisibilité de la figure, nous n'avons pas reporté les identifiants des fournisseurs et demandeurs utilisés dans CEPC. La syntaxe concrète du modèle de l'application CEPC utilisant notre notation graphique est présentée dans la figure 67 "Représentation du modèle de l'application CEPC dans notre notation" ; celle basée sur XML est présentée dans la figure 68 : "Modèle de l'application CEPC sous la forme d'un fichier XML".

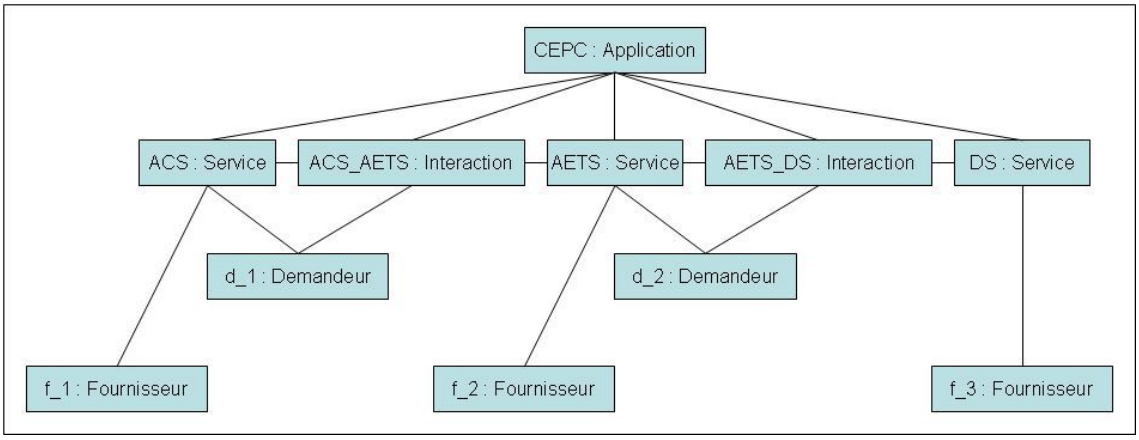


Figure 44. Syntaxe abstraite d'une partie du modèle de l'application CEPC.

3. DÉFINITION DU DÉPLOIEMENT

Dans cette section, nous allons détailler le résultat de l'adaptation de la définition du déploiement de l'université du Colorado [HHC+98], ainsi que sa signification au niveau des services et des applications à services introduites précédemment. Pour remarque, l'état de l'art "déploiement" de ce document de thèse présente en détail la définition du déploiement de l'université du Colorado, ainsi que nos conclusions et critiques.

Nous avons choisi d'adapter la définition du déploiement de l'université du Colorado puisque, parmi les trois définitions majeures présentées, elle est la seule à offrir un processus non linéaire et non séquentiel, elle est la plus complète (car elle couvre l'insertion, l'exécution, une partie de la maintenance et la suppression d'un système logiciel sur un site) et elle est aussi la plus cohérente (car elle définit à la fois l'insertion d'un système logiciel sur un site et sa suppression, ainsi qu'à la fois le lancement et l'arrêt de l'exécution du système logiciel).

3.1 Définition

Notre proposition concernant la phase de déploiement est un processus non linéaire et non séquentiel qui est constitué par cinq activités de déploiement, à savoir, l'installation, l'activation, la maintenance, la désactivation et la désinstallation. En détail :

- L'installation d'un logiciel sur un site signifie l'insertion de son ou de ses unités de déploiement sur le site.
- L'activation d'un logiciel signifie le lancement de l'exécution de la ou des implémentations contenues dans la ou les unités de déploiement relatives au logiciel.
- En ce qui concerne l'activité de maintenance, nous la considérons du point de vue du déploiement, c'est-à-dire qu'il n'est pas question ici d'incorporer la détection et la correction de "bug" (le développement de "patch") dans la phase de déploiement. Ainsi, dans le cadre du déploiement, la maintenance d'un logiciel englobe sa mise à jour et la modification de son architecture. L'activité de maintenance n'est pas une activité de déploiement en elle-même, c'est un terme qui regroupe quatre sous activités de déploiement, à savoir :
    - la mise à jour statique d'un logiciel qui désigne le remplacement d'une ou plusieurs implémentations (contenues dans une ou plusieurs unités de déploiement réalisant le logiciel) par une ou plusieurs autres implémentations (contenues dans une ou plusieurs unités de déploiement) lorsque le logiciel n'est pas activé (c'est-à-dire lorsqu'il n'est pas en train de s'exécuter). Attention, l'architecture du logiciel doit être conservée ; de même, les architectures des implémentations remplacées et des implémentations remplaçantes doivent être similaires.
    - la mise à jour dynamique qui est en tout point similaire à la mise à jour statique à ceci près qu'elle cible un logiciel activé.
    - l'évolution d'un logiciel qui désigne la modification de l'architecture du logiciel lorsque ce dernier n'est pas activé.
    - la dernière sous-activité est le dynamisme. Il désigne la modification de l'architecture d'un logiciel lorsque ce dernier est activé.
- La désactivation d'un logiciel est la fonction inverse de l'activité d'activation, c'est-à-dire qu'elle vise à arrêter l'exécution du logiciel.
- La désinstallation d'un logiciel est elle aussi une fonction inverse mais, cette fois, de l'activité d'installation, c'est-à-dire qu'elle vise à supprimer un logiciel inséré sur un site.

Les activités d'installation, d'activation, de désactivation et de désinstallation sont quatre activités qui aboutissent à un état de déploiement stable (respectivement installé, activé, désactivé, désinstallé). Par exemple, une fois l'activité d'activation réalisée, le logiciel est dans l'état actif. L'activité maintenance est quant à elle une activité transitoire. Par exemple, si un logiciel est actif et qu'une des activités de maintenance est exécutée alors si l'exécution se déroule bien, le logiciel se retrouvera dans

l'état actif (sinon, soit l'erreur d'exécution de la maintenance est rattrapée et le logiciel se retrouvera dans l'état actif, soit il pourra se retrouver au mieux dans l'état installé, au pire dans un état "corrompu").

Il est intéressant de noter que les états de déploiement installé et désactivé sont relativement similaires, au point qu'il est possible de les factoriser. Cependant, l'état désactivé n'est accessible que depuis l'état activé, ainsi il informe le déployeur sur le fait qu'un logiciel désactivé a déjà pu être activé (au moins une fois). C'est pourquoi nous avons conservé la distinction entre les états installé et désactivé.

Pour remarque, dans le cadre de la mise à jour dynamique et du dynamisme, deux propriétés sont à distinguer. La première porte sur le gel, la propagation et le dégel de l'état (interne) des implémentations ciblées. La seconde porte sur le gel et le dégel des communications vers et partant des implémentations ciblées. Typiquement, la technologie OSGi permet les mises à jour dynamiques, néanmoins, les deux propriétés ci-dessus ne sont pas offertes.

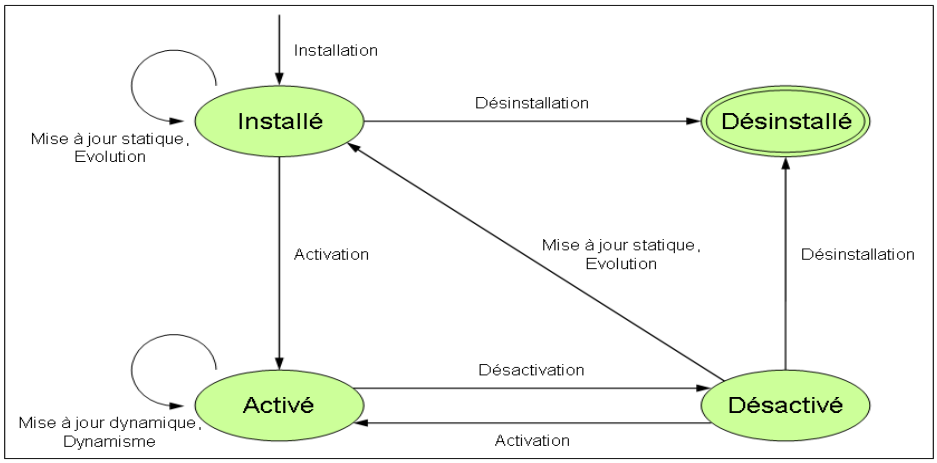


Figure 45. Proposition de phase de déploiement contenant les états et activités de déploiement.

Dans le cadre de l'approche dirigée par les modèles que nous suivons, le point d'entrée de la phase de déploiement est non pas un logiciel, mais un modèle d'application à services. Ce dernier contient des modèles de services et d'interactions et est totalement découplé d'un quelconque niveau "implémentation". Ensuite, dans le cadre du déploiement "au plus près" nous avons choisi que les implémentations de services et les unités de déploiement que nous manipulons soient de même grain, ce qui signifie que toute unité de déploiement contient une unique implémentation de service et inversement que toute implémentation soit packagée seule dans son unité de déploiement.

3.2 Définition du déploiement de services

Ci-dessous, nous précisons la signification de la phase de déploiement pour les services. Ainsi :

- l'installation d'un service sur un site signifie l'insertion du service (en réalité de son unité de déploiement) sur le site.
- l'activation d'un logiciel signifie le lancement de l'exécution du service (en réalité de l'exécution de l'implémentation de service contenue dans l'unité de déploiement correspondant au service). Dans le cas général, l'activation d'un service peut échouer si ses dépendances ne sont pas satisfaites.
- dans le cadre du déploiement, la maintenance d'un service englobe sa mise à jour et la modification de son architecture. L'activité de maintenance regroupe quatre sous activités de déploiement, à savoir :
    - la mise à jour statique d'un service qui désigne le remplacement de l'implémentation actuelle/courante du service (contenue dans l'unité de déploiement correspondante) par une autre implémentation de service (elle-même contenue dans une autre unité de déploiement) lorsque le service n'est pas activé. Attention, l'architecture du service (et donc son contrat de service) doit être conservée.

  - la mise à jour dynamique qui est en tout point similaire à la mise à jour statique à ceci près qu'elle cible un service activé.
  - l'évolution d'un service qui désigne la modification de l'architecture du service lorsque ce dernier n'est pas activé. Une modification peut, par exemple, être l'ajout d'un demandeur au contrat d'un service.
  - le dynamisme d'un service désigne la modification de l'architecture du service lorsque ce dernier est activé.
- la désactivation d'un service vise à arrêter son exécution.
- la désinstallation d'un service vise à supprimer/enlever un service inséré sur un site.

3.3 Définition du déploiement d'application à services

Nous allons maintenant présenter la signification de la phase de déploiement pour les applications à services. La seule différence majeure entre le déploiement d'un service et le déploiement d'une application à services réside dans le fait que déployer un service met en jeu une unité de déploiement alors que déployer une application en met en jeu une ou plusieurs. Ainsi :

- l'installation d'une application à services sur un site signifie l'insertion, sur le site, du ou des unités de déploiement réalisant le ou les services qui appartiennent à l'application.
- l'activation d'une application à services signifie le lancement de l'exécution de l'application et donc le lancement de l'exécution de la ou des implémentations de service contenues dans la ou les unités de déploiement correspondant aux services de l'application).
- dans le cadre du déploiement, la maintenance d'une application à services englobe sa mise à jour et la modification de son architecture. L'activité de maintenance regroupe quatre sous-activités de déploiement, à savoir :
  - la mise à jour statique d'une application à services qui désigne le remplacement d'une ou plusieurs implémentations de services actuelles/courantes (appartenant à l'application) par une ou d'autres implémentations de service lorsque l'application n'est pas activé. Attention, l'architecture de l'application doit être conservée, de même que les architectures des services ciblés par la mise à jour.
  - la mise à jour dynamique qui est en tout point similaire à la mise à jour statique à ceci près qu'elle cible une application à services activée.
  - l'évolution d'une application à services qui désigne la modification de l'architecture d'une application à services lorsque cette dernière n'est pas activée.
  - le dynamisme d'une application à service désigne la modification de l'architecture de l'application lorsque cette dernière est activée.
- la désactivation d'une application à services vise à arrêter l'exécution de l'application.
- la désinstallation d'une application à services vise à supprimer/enlever une application à services insérée sur un site.

Enfin, la figure ci-dessous présente quelles sont les activités de déploiement possibles en fonction de l'état de déploiement de l'application ciblée.

Activité : / Etat :	Installation	Activation	Désactivation	Maintenance	Désinstallation
Installé	Non	Oui	Non	Oui	Oui
Activé	Non	Non	Oui	Oui	Non
Désactivé	Non	Oui	Non	Oui	Oui
Désinstallé	Oui	Non	Non	Non	Non

Figure 46. Les activités de déploiement possibles en fonction de l'état de l'application ciblée.

4. ARCHITECTURE DE NOTRE SYSTÈME DE DÉPLOIEMENT

L'architecture de déploiement que nous proposons est représentée dans la figure ci-dessous.

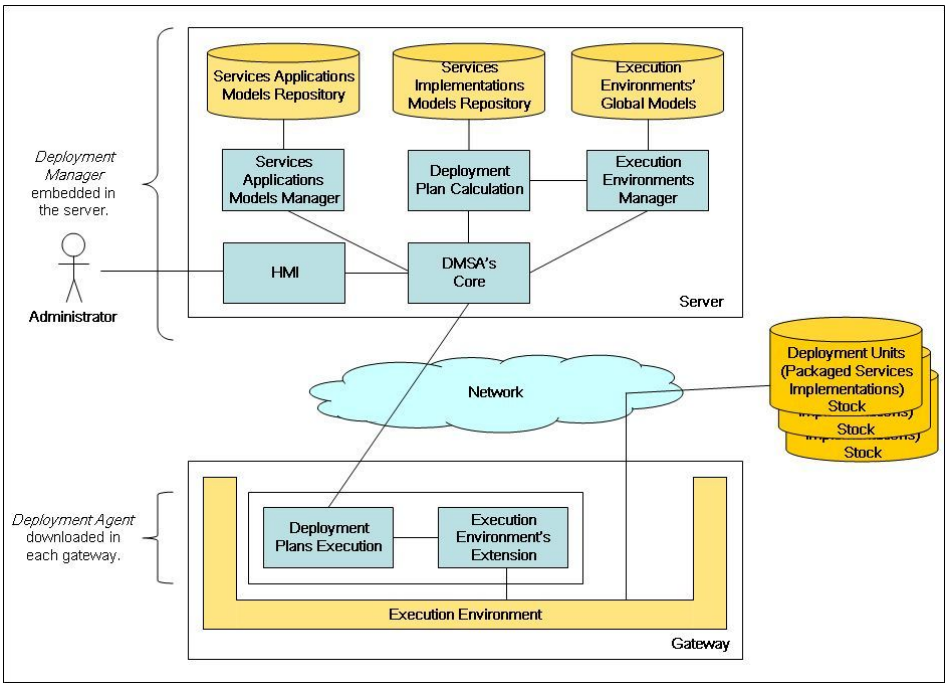


Figure 47. Architecture de notre système de déploiement (DMSA).

Notre proposition est une architecture répartie sur deux tiers, à savoir, le tiers serveur internet et le tiers passerelles pervasives. Nous avons choisi de faire une architecture en forme d'arbre. Le sommet de l'arbre est le serveur internet et les feuilles sont les passerelles (plus précisément ce sont les environnements d'exécutions à services embarqués par les passerelles). Nous avons positionné l'administrateur de façon à ce qu'il interagisse avec la partie de notre DMSA qui réside sur le serveur internet ; en effet, les passerelles sont enfouies, par exemple dans les usines, et peuvent être distribuées (géographiquement), or il n'est pas imaginable que l'administrateur doive se déplacer auprès de chaque passerelle pour commander des actions de déploiement au DMSA (sachant, de plus, qu'il peut ne pas avoir l'autorisation d'accéder aux sites dans lesquels les passerelles sont enfouies).

Ensuite, nous avons choisi de positionner les parties de notre DMSA "gourmandes" en calculs dans le serveur internet (ces parties sont, par exemple, la validation des modèles d'application, leur ordonnancement, le calcul des plans de déploiement et le système d'exécution parallèle). En effet, les passerelles que nous ciblons ont clairement moins de capacité de calcul et sont moins aptes réaliser des calculs que le serveur internet.

Nous avons aussi choisi de stocker les informations relatives au modèle global (ou image globale) de chaque environnement d'exécution dans le serveur internet. En effet, les passerelles que nous ciblons ne nous permettent pas de stocker et de persister des informations dans leur système de fichiers. De plus, nous avons souhaité éviter que les parties de calcul soient pénalisées par la récupération d'informations au-dessus du réseau. Pour remarque, le problème de la synchronisation entre l'état réel des environnements et leur modèle global ne se pose pas, puisque toutes les actions/activités de déploiement ciblant les environnements gérés sont réalisées par le biais du DMSA.

Au final, l'architecture que nous proposons doit être considérée comme une architecture centralisée plutôt que comme une architecture décentralisée puisque, les informations concernant les environnements, les calculs des plans de déploiement, etc. sont centralisés dans le serveur internet et non décentralisés dans chaque environnement, au plus près de chaque environnement. En définitive, seules les

parties responsables de l'exécution (en local) des plans de déploiement et de l'extension de l'environnement (pour offrir une approche *naming*) sont présentes dans chaque environnement et sont donc décentralisées.

5. DÉPLOIEMENT D'UN MODÈLE D'APPLICATION SUR UN ENV.

Dans cette section, nous détaillons notre proposition concernant le déploiement d'un modèle d'application à services sur un environnement d'exécution à services. Nous allons ainsi détailler les défis relatifs au déploiement de modèles au-dessus de l'approche à service, à la sous-activité de mise à jour statique et à l'établissement d'un déploiement transactionnel via un mécanisme capable de tenter un "retour en arrière" (*rollback*) lorsqu'une activité de déploiement échoue.

Dans le cadre de cette thèse, nous nous sommes intéressés à l'installation, l'activation, la désactivation, la mise à jour statique et la désinstallation d'applications à services. Pour ce faire, nous nous basons sur les activités d'installation, d'activation, de désactivation et de désinstallation d'unités de déploiement qui sont fournies par les environnements d'exécution à services OSGi. Nous n'utilisons pas le mécanisme de mise à jour statique fourni par la technologie OSGi, nous nous en sommes expliqués précédemment dans ce document. Pour être tout à fait explicite, le rôle DMSA est de prendre en entrée une ou plusieurs activités de déploiement au niveau modèle d'application et, d'une part, de les traduire en des ensembles d'activités de déploiement aux niveaux implémentation de service et instance d'implémentation de service et, d'autre part, d'exécuter les ensembles calculés (c'est-à-dire d'exécuter les plans de déploiement calculés).

La figure ci-dessous présente les activités de déploiement réalisables sur un modèle d'application à services en fonction de son état de déploiement.

Activité : / Etat :	Installation	Activation	Désactivation	Mise à jour statique	Désinstallation
Installé	Non	Oui	Non	Oui	Oui
Activé	Non	Non	Oui	Non	Non
Désactivé	Non	Oui	Non	Oui	Oui
Désinstallé	Oui	Non	Non	Non	Non

Figure 48. Activités de déploiement possibles et état de déploiement de l'application.

Pour effectivement déployer un modèle d'application à services, trois pré-requis sont nécessaires à notre DMSA. Ainsi, le modèle de l'application doit être enregistré, par l'administrateur, dans le dépôt de modèles d'application du DMSA ; l'environnement cible doit faire partie des environnements gérés par le DMSA (il peut donc être nécessaire de l'introduire dans le gestionnaire d'environnements du DMSA) ; enfin, le dépôt de modèles d'implémentations de services (c'est-à-dire d'unités de déploiement) doit être peuplé (il peut ne pas être peuplé, mais alors, la sélection des implémentations de services qui réaliseront le modèle de l'application n'aboutira pas).

Le point d'entrée du déploiement (et donc des activités de déploiement) est généralement un triplet composé par :

- l'identifiant du modèle de l'application (qui a servi à l'enregistrer dans le DMSA),
- l'identifiant de l'environnement d'exécution cible (dans notre contexte, nous utilisons son adresse IP)
- et, enfin, l'activité de déploiement à effectuer.

Néanmoins, deux des trois mises à jour proposées nécessitent non pas un triplet, mais un quadruplet comme nous le verrons dans la sous-section 5.4 ci-après.

5.1 Le déploiement dirigé par les modèles

Le déploiement dirigé par les modèles soulève deux défis. Le premier concerne la résolution des dépendances. Le second porte sur l'existence et le maintien d'un lien entre la "réalité" et le niveau "modèle", c'est-à-dire entre les implémentations (et les instances) effectivement présentes dans un environnement et le modèle global créé (et maintenu) par le DMSA.

Dans le cadre de l'approche dirigée par les modèles, le premier défi se répartit entre deux sous-défis, à savoir la validation du modèle de l'application et la sélection des implémentations de service qui le réaliseront effectivement.

De fait, l'étape de validation valide/vérifie des aspects triviaux tels le fait que :

- les identifiants des modèles de services et des interactions qui apparaissent dans l'application sont bien disjoints deux à deux,
- les fournisseurs proposés et les demandeurs requis par l'application apparaissent réellement dans un des modèles de services de l'application.
- les interactions entre demandeurs et fournisseurs portent bien sur des demandeurs et fournisseurs qui apparaissent dans l'application, ainsi que sur un même contrat (dans le cas de la technologie OSGi sur une même interface Java) et sur des rôles de communication compatibles (par exemple, un client avec un serveur ou un souscripteur avec un publieur)

La validation porte aussi sur des aspects moins triviaux tels :

- le fait que tout demandeur est soit satisfait par un fournisseur (n'appartenant pas au même service) présent dans le modèle de l'application (via une interaction), soit exposé comme un requis de l'application,
- le fait que tout fournisseur est soit utilisé par un demandeur (n'appartenant pas au même service) présent dans le modèle de l'application (via une interaction), soit exposé comme une fourniture de l'application.

Pour remarque, la présence de cycle est détectée lors de l'ordonnancement des activations et des désactivations des modèles de services, nous en discutons plus après. Il est aussi intéressant de noter que nous permettons qu'une application soit aussi bien un graphe, qu'un ensemble de graphes (disjoints).

L'étape de validation d'un modèle d'application à services est effectuée lors de l'enregistrement du modèle de l'application dans le dépôt de modèles d'application à services du DMSA. Effectuer cette étape lors de l'introduction d'un modèle d'application dans le DMSA est intéressant puisque cela permet de distinguer, au plus tôt, les modèles valides des modèles invalides et puisque cela permet de n'effectuer cette étape qu'une seule fois. Cette étape aurait, par exemple, pu être réalisée au cours de l'activité d'installation, cependant, cela se serait traduit par la manipulation, dans le DMSA, de modèles pouvant être invalides et par l'exécution de l'étape de validation pour chaque installation.

En ce qui concerne le second sous-défi (c'est-à-dire la sélection d'implémentations), nous avons mis en avant le fait que les modèles d'application que nous appréhendons sont distincts de tout lien avec le niveau implémentation (de service). La sélection des implémentations de service/unités de déploiement revient donc au DMSA. Elle consiste à parcourir les modèles d'implémentations de services disponibles en recherchant un modèle d'implémentation de service pour réaliser chaque modèle de service apparaissant dans le modèle de l'application à déployer. Nous verrons, dans la section numéro sept de ce chapitre (intitulé : "appréhension du contexte embarqué des environnements d'exécution"), que notre algorithme de sélection des implémentations de services pour un modèle d'application donné favorise les implémentations d'ores et déjà déployées dans l'environnement d'exécution ciblé par l'activité de déploiement. Enfin, cette étape de sélection est réalisée lors de l'installation de l'application dans un environnement. Il est intéressant de remarquer que l'ensemble des implémentations sélectionnées pour un modèle d'application peut varier d'un environnement à l'autre (en fonction des implémentations déployées dans l'environnement). Cet ensemble peut aussi varier en fonction du temps, ainsi une application

installée à t0 aura un ensemble d'implémentations sélectionnées e0, puis sa nouvelle installation à t2 (après sa désinstallation à t1) sera un ensemble e2 pouvant être différent de e0.

Le second défi du déploiement par les modèles est un défi qui a trait à l'utilisation de modèles d'une manière générale. En effet, l'utilisation de modèles implique la création et le maintien d'un lien entre le niveau modèle et la "réalité".

Dans le cadre du DMSA, ce lien est créé et maintenu (environnement par environnement) via un modèle global qui est créé et maintenu pour chaque environnement géré. Ce modèle global est composé :

- par les identifiants des modèles d'application déployés sur l'environnement,
- par l'état de déploiement des modèles d'application déployés,
- par la sélection (en termes d'implémentations de services) de chaque modèle d'application déployé,
- par l'ordre d'ordonnancement des activations et désactivations de chaque modèle d'application déployé,
- par l'ensemble des implémentations de services "nécessaires", c'est-à-dire l'information concernant le fait que toute implémentation de service déployée est nécessitée par tel modèle de service de tel modèle d'application (le terme "nécessaire" désigne les implémentations qui ne sont pas actives, c'est-à-dire qui ne sont pas "utilisées"/en cours d'exécution dans un modèle d'application),
- par l'ensemble des implémentations de services "utilisées" (c'est-à-dire utilisées par une ou plusieurs applications actives/en cours d'exécution). L'information stockée concerne alors le fait que la *factory* d'une implémentation de service est à l'origine de telle instance d'implémentation de service qui réalise tel modèle de service de tel modèle d'application.

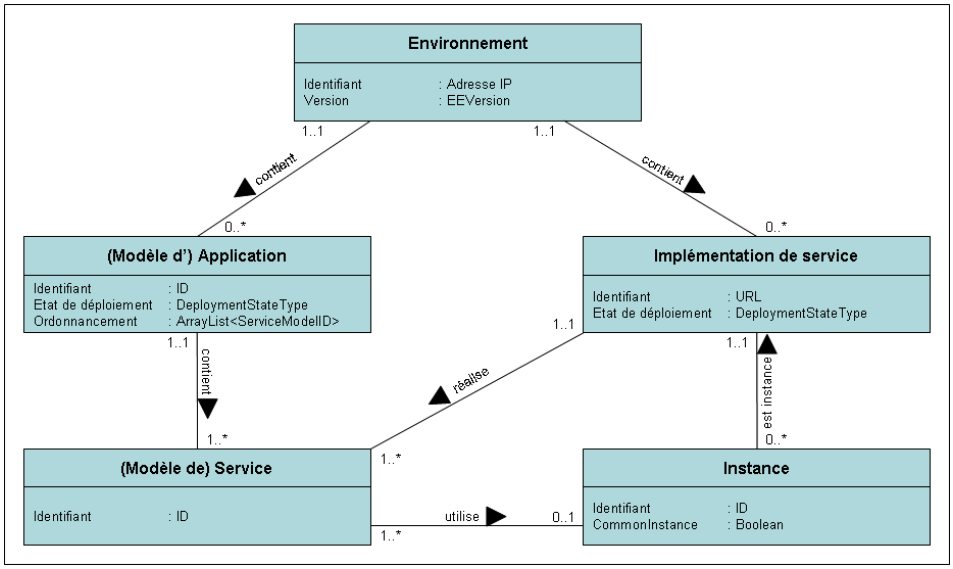


Figure 49. Méta-modèle pour les environnements.

La figure ci-dessus mérite quelques précisions. Elle ne met en avant que les classes et les attributs qui sont spécifiques à la vue déploiement des environnements ; le méta-modèle complet inclut les classes fournisseur, demandeur et interaction, ainsi que tous les attributs des classes (modèle d') application et (modèle de) service. Ce méta-modèle définit qu'une implémentation de service n'est contenue que par "1..1" environnement ; dans les faits, plusieurs copies de la même implémentation de service peuvent être déployées (et donc contenues) dans plusieurs environnements. Ensuite, l'association "réalise" peut aussi être appelée "nécessite" à condition d'inverser le sens de la flèche et si l'état de déploiement de l'implémentation est installée, désactivée ou désinstallée ; de même si l'implémentation est activée, alors elle peut aussi être nommée "utilise". Les deux verbes "nécessite" et " utilise" correspondent bien évidemment aux notions d'implémentations "nécessaires" et d'implémentations "utilisées" définies dans le paragraphe précédant la figure ci-dessus. Lorsqu'une instance de la classe

"(Modèle d') Application" est active (respectivement installée, désactivée ou désinstallée), alors les instances de la classe "(Modèle de) Service" qu'elle contient sont, chacune, liées à une instance de la classe "Instance" (respectivement les instances de la classe "(Modèle de) Service) qu'elle contient ne sont liées à aucune instance de la classe "Instance"). Le type EEVersion définit les valeurs "R3.0" et "R4.1" qui correspondent aux deux versions d'environnement d'exécution OSGi que nous ciblons dans notre contexte. Le type DeploymentStateType définit les valeurs *installed*, *activated*, *deactivated* et *deinstalled* qui correspondent aux quatre états stables de notre définition du déploiement. Le type associé à l'attribut "ordonnancement" est une *ArrayList* qui contient les identifiants des modèles de services appartenant au modèle d'application. Ces identifiants sont ordonnés afin de permettre l'activation de l'application ; concrètement, le service correspondant au premier élément doit être activé avant le second élément, qui doit lui-même l'être avant le troisième, etc.

Nous considérons l'approche que nous avons suivie comme une approche dirigée par les modèles. En effet, toute activité de déploiement soumise à notre DMSA est appréhendée comme une demande de modification du modèle global associé à l'environnement cible. Ainsi, l'activité est tout d'abord étudiée par rapport au modèle global associé à l'environnement cible. Si elle est réalisable (par exemple, l'activation d'une application installée est réalisable, alors que la désactivation d'une application qui n'est pas active ou qui est déjà désactivée n'a pas de sens), la partie du DMSA embarquée dans le serveur internet crée un plan de déploiement et commande son exécution par la partie du DMSA qui est embarquée dans l'environnement cible. Si l'exécution du plan de déploiement aboutit, alors, le modèle global de l'environnement est mis à jour. Maintenant, si, par exemple, l'activité n'est pas réalisable ou si la création de son plan de déploiement échoue ou si l'exécution de ce dernier n'aboutit pas, alors le modèle global de l'environnement n'est pas mis à jour (c'est-à-dire qu'il reste dans l'état où il était avant que l'activité ne soit commandée au DMSA). Nous verrons dans la section autour du déploiement transactionnel que nous essayons aussi de restaurer l'environnement cible dans l'état dans lequel il se trouvait avant que l'activité de déploiement n'ait été exécutée et cela afin de maintenir la cohérence entre le modèle global associé à l'environnement et la "réalité" de l'environnement.

5.2 Le déploiement au dessus de l'approche à service

L'approche à service est basée sur le patron SOA (Service-oriented Architecture). Le fonctionnement/la mise en œuvre de ce patron est généralement le suivant :

- un fournisseur commence par être activé,
- cela implique qu'il s'enregistre dans le registre de services,
- ensuite, un demandeur requérant ce fournisseur devient lui aussi actif,
- cela implique qu'il va alors aller chercher le fournisseur en question dans ce même registre de services afin de pouvoir effectivement l'utiliser.

La séquence ci-dessus décrit la mise en œuvre par défaut du patron SOA dans la technologie OSGi.

De fait, il devient primordial d'ordonnancer les activations des implémentations de services :

- afin de garantir qu'un modèle d'application est effectivement activable. Cela se traduit par la recherche d'un ordre d'activation pour les différentes implémentations de services apparaissant dans l'application. Cette recherche passe par la détection des cycles, ces derniers empêchant la définition d'un quelconque ordre d'activation.
- mais aussi afin d'éviter les erreurs, *crashs* et exceptions qui peuvent apparaitre, dans le cas général, lorsqu'un demandeur (entrain d'être activé) ne parvient pas à trouver de fournisseur le satisfaisant dans le registre de service. Pour être tout à fait explicite, dans le cas général, lorsqu'un service ayant un demandeur vient à être activé, si le demandeur en question ne trouve pas de fournisseur le satisfaisant, alors l'activation du service échoue.

Nous venons de voir que la définition d'un tel ordre est nécessaire pour l'activation d'une application. Cependant, il est aussi important de noter que la définition d'un ordre similaire est aussi nécessaire pour la désactivation d'une application. Lorsqu'une application est active, cela signifie que les demandeurs peuvent utiliser (voire utilisent) les fournisseurs. Nous rappelons aussi que la désactivation

d'une application (comme toutes les activités de déploiement au niveau application) n'est pas une activité unitaire ; en l'occurrence c'est une somme de désactivations au niveau unité de déploiement/(implémentation de service). Ainsi, si aucun ordre de désactivation n'est défini pour une application qui va être désactivée, alors, par exemple, à t0 un demandeur utilise un fournisseur, à t1 la désactivation de l'application commence, à t2 le service possédant le fournisseur est désactivé, à t3 le demandeur tente d'utiliser le fournisseur (qui n'est plus actif) ce qui se traduit, dans le cas général, par le *crash* du service qui possède le demandeur (et cela avant qu'il n'ait pu être désactivé, il l'aurait été à t4).

Dans le cadre du déploiement nous avons présenté la désactivation comme étant la fonction inverse de l'activation. En ce qui concerne l'ordonnancement de l'activation et de la désactivation dans notre contexte, cette idée de fonction inverse est aussi valide. Ainsi, l'ordre de désactivation qui est utilisé par le DMSA pour une application donnée est l'inverse de l'ordre d'activation de l'application. Pour remarque, pour la plupart des applications, utiliser un ordre de désactivation qui est l'inverse de l'ordre d'activation n'est qu'une possibilité parmi l'ensemble des possibles. En effet, il est généralement possible d'établir plusieurs ordres d'activation et de désactivation différents pour une même application.

La technologie OSGi n'est pas dupe de cette problématique d'ordonnancement des activations et des désactivations. En conséquence, elle fournit des mécanismes comme *Declarative Services* (DS) en réponse. Dans le cas de DS, un service (nommons le "S") peut être activé sans que le registre de service ne contienne de quoi satisfaire les requis de "S". Cependant, DS n'assure pas qu'un service activé est réellement entrain de s'exécuter (ce qui est pourtant le résultat attendu à l'issue de l'activité d'activation). Ainsi, une fois "S" activé, il ne sera en réelle exécution que lorsque ses requis seront effectivement satisfaits. Ici il est important de souligner que DS et les mécanismes assimilés ne proposent aucun niveau "application", ainsi l'état "en exécution" du service "S" activé dépend totalement des enregistrements et des désenregistrements de fournisseurs dans le registre de services (ces enregistrements et désenregistrements étant laissés au bon vouloir des implémentations de services allant et venant sur l'environnement).

Dans notre contexte, le rôle du déployeur (c'est-à-dire du DMSA) est d'amener une application dans l'état de déploiement ciblé par l'activité de déploiement commandée (par exemple, l'installation cible l'état installé, l'activation cible l'état activé, c'est-à-dire en exécution). En effet, il n'est pas envisageable que le déployeur puisse avertir son administrateur que telle application est active (c'est-à-dire en exécution) et que, par conséquence, le client soit informé que son application de mesure de données électriques est fonctionnelle si cette dernière ne l'est pas, c'est-à-dire si au moment où il va l'essayer cette dernière ne s'exécute pas parce qu'un de ses service est "suspendu".

En conséquence l'utilisation d'un mécanisme comme Declarative Services ou d'un mécanisme similaire n'est pas envisageable dans notre contexte.

Ensuite, l'établissement d'un ordonnancement pour l'activation et pour la désactivation dépend certes au final des demandeurs et des fournisseurs ; cependant, la définition d'un demandeur ou d'un fournisseur dépend du patron de communication qui est associé à toute interaction (de deux services). Dans notre contexte, trois patrons de communication sont possibles, le Client - Serveur, le Souscripteur - Publieur (*Publish - Subscribe*) et le Producteur - Consommateur. Ainsi, un demandeur est défini pour le rôle de communication, soit Client, soit Souscripteur ou soit Consommateur. Un fournisseur est quand à lui défini soit pour un Serveur, soit pour un Publieur ou soit pour un Producteur. En définitive, pour le DMSA, l'établissement d'un ordonnancement pour l'activation et la désactivation d'une application à service est réalisé à partir de la connaissance des services et du patron de communication associé à chaque interaction.

Enfin, du fait de notre approche dirigée par les modèles et des informations que nous avons retenues pour notre méta-modèle orienté déploiement des applications à services, le DMSA ordonnance les modèles d'application dès leur enregistrement dans le dépôt de modèles d'application. L'idée est encore une fois de ne peupler le dépôt de modèles d'application que de modèles effectivement activables (et donc d'éviter de manipuler et déployer des modèles erronés) [CL07].

5.3 Le déploiement dirigé par les modèles au-dessus de l'approche à service

Nous avons présenté notre but concernant le DMSA, à savoir concevoir et réaliser un système de déploiement qui prenne en entrée des activités de déploiement au niveau modèle d'application et qui sache, d'une part, les traduire en des ensembles d'activités de déploiement aux niveaux implémentation de service et instance d'implémentation de service et qui sache, d'autre part, exécuter les ensembles calculés (c'est-à-dire qui sache exécuter les plans de déploiement calculés, sur les environnements cibles). Ainsi, dans le cadre du déploiement d'un modèle d'application à services sur un environnement d'exécution à services, nous nous sommes intéressés à la gestion du niveau implémentation de service, à la gestion du niveau instance d'implémentation de service et à assurer la conformité entre l'application déployée et son modèle.

La gestion du niveau implémentation de service passe par la définition et l'utilisation d'instructions de déploiement relatives au niveau implémentation de service. Ces instructions sont "needInstallation", "needActivation", "needDeActivation" et "needDeInstallation". Pour être effectivement utilisée, chacune de ces instructions est associée à une implémentation de service (c'est-à-dire à une unité de déploiement, en l'occurrence un *bundle* OSGi). Concrètement, le DMSA identifie chaque implémentation de services grâce à l'URL à laquelle elle peut être récupérée ; cette URL est utilisée telle quelle pour l'installation et donne lieu à une recherche dans l'environnement pour l'activation, la désactivation et la désinstallation.

Lorsque la partie du DMSA embarqué dans chaque environnement d'exécution reçoit un plan de déploiement à exécuter :

- l'instruction "needInstallation" lui commande d'utiliser la primitive de l'environnement d'exécution concernant l'installation d'une implémentation. Le résultat est qu'une copie de l'implémentation de service située à l'URL correspondante est transférée et "installée" sur l'environnement.
- "needActivation" lui commande d'utiliser la primitive de l'environnement d'exécution concernant l'activation d'une implémentation. Ce qui a pour effet, de faire exécuter la méthode "*start(BundleContext)*" située dans l'implémentation de service et de lancer la *factory* d'instance d'implémentation de service.
- "needDeActivation" commande l'utilisation de la primitive de désactivation d'implémentation de service. Cela se traduit par l'exécution de la méthode "*stop(BundleContext)*" et a pour effet d'arrêter l'exécution de l'implémentation ainsi que d'arrêter la *factory* d'instance correspondante.
- enfin, l'instruction "needDeInstallation" commande l'utilisation de la primitive de désinstallation d'implémentation. Cela se traduit par la suppression de la copie locale de l'implémentation.

Le DMSA utilise ces instructions, entre autres instructions, pour traduire les activités de déploiement au niveau application qui lui sont soumises.

La gestion du niveau instance d'implémentation de service passe par la définition et l'utilisation d'instructions de déploiement relatives au niveau instance d'implémentation de service. Dans notre contexte et du fait des propriétés de déploiement que nous avons souhaité obtenir, nous avons été amené à distinguer deux types d'instances. Le premier type est relatif aux instances qui sont utilisées pour réaliser un modèle de service précis dans un modèle d'application précis. Concrètement, une instance de ce premier type n'est utilisée que pour réaliser un unique modèle de service. Le second type d'instance désigne une instance qui peut être utilisée pour réaliser un ou plusieurs modèles de services. Le DMSA ne manipule qu'une seule instance de ce second type par *factory*. Nous nommons les instances de ce second type les "commonInstances". Les *factories* ne distinguent pas de type d'instance. Les instructions relatives au niveau instance sont : "createAndUseCommonInstance", "useCommonInstance", "createAndUsePersonalizedInstanceDestruction", "needCommonInstanceDestruction" et "needPersonalizedInstanceDestruction". Pour être effectivement utilisée, chacune de ces instructions est associée soit à une implémentation de service (et donc à la *factory* correspondante), soit à un couple

composé d'une implémentation de service (et donc à la *factory* correspondante) et de l'identifiant de l'instance.

Lorsque la partie du DMSA embarqué dans chaque environnement d'exécution reçoit un plan de déploiement à exécuter :

- l'instruction "createAndUseCommonInstance" lui commande d'utiliser la *factory* de l'implémentation ciblée afin de faire créer et d'utiliser une instance qui sera identifiée, dans le DMSA, comme la "commonInstance" de cette *factory*.
- "useCommonInstance" lui commande d'utiliser la *factory* de l'implémentation ciblée afin de récupérer l'instance correspondant à "commonInstance" et de l'utiliser,
- "createAndUsePersonalizedInstanceDestruction" commande d'utiliser la *factory* de l'implémentation ciblée pour faire créer et utiliser une instance qui sera dédiée au modèle de service donné appartenant au modèle d'application donné,
- "needCommonInstanceDestruction" commande d'utiliser la *factory* de l'implémentation ciblée pour détruire l'instance correspondant à la "commonInstance",
- enfin, "needPersonalizedInstanceDestruction" commande d'utiliser la *factory* de l'implémentation ciblée pour détruire l'instance correspondant au modèle de service donné du modèle d'application donné cible.

Pour remarque, l'instruction "useCommonInstance" désigne à la fois l'utilisation et la réutilisation de l'instance "commonInstance".

La gestion des niveaux implémentation de service et instance d'implémentation de service passe aussi, bien évidemment, par la mise à jour du modèle global associé à chaque environnement.

Nous présentons maintenant la vue IDM (ingénierie dirigée par les modèles) du DMSA dans la figure ci-dessous.

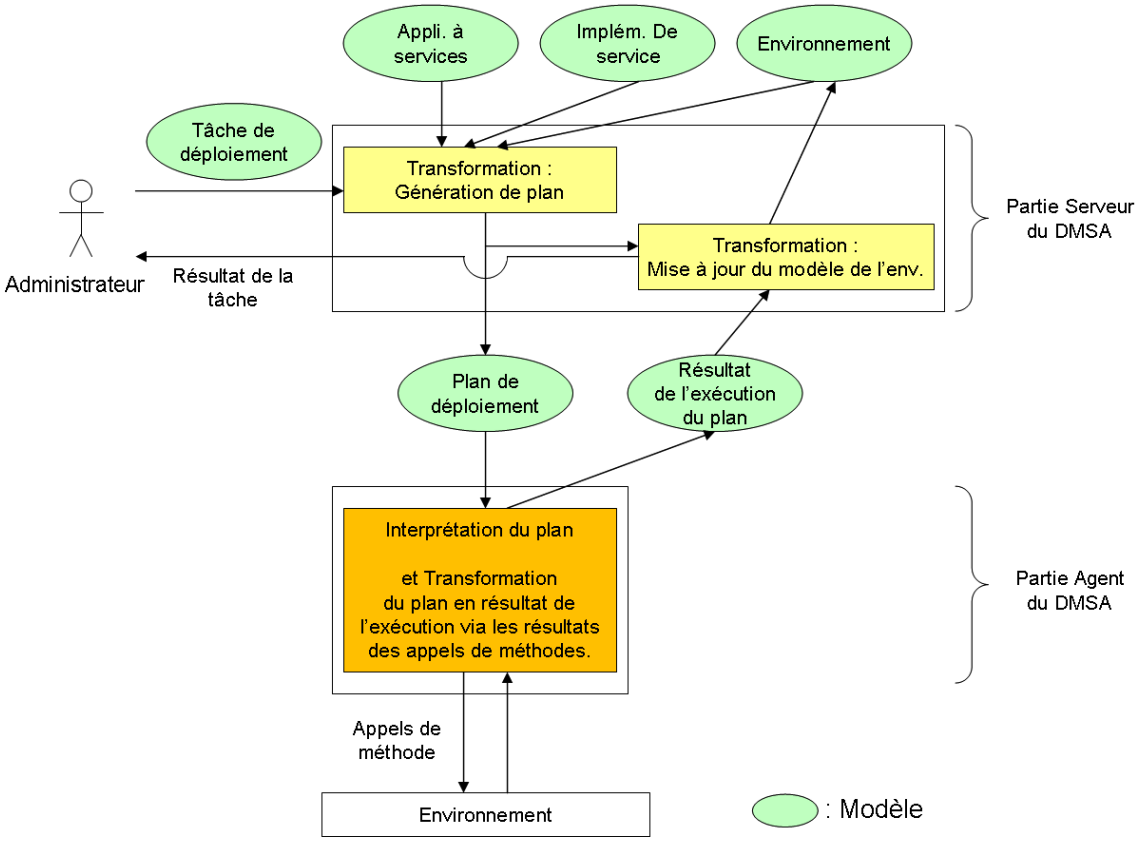


Figure 50. Vue IDM du DMSA

Précédemment, nous avons détaillé les mét-modèles pour les applications à services et les environnements d'exécution à services. Nous allons maintenant présenter les méta-modèles pour les plans de déploiement, les tâches de déploiement et les résultats de l'exécution d'un plan de déploiement. La présentation de ces trois méta-modèles est faite au travers des trois figures ci-dessous. Pour remarque le méta-modèle pour les implémentations de services est détaillé, quant à lui, dans la prochaine section.

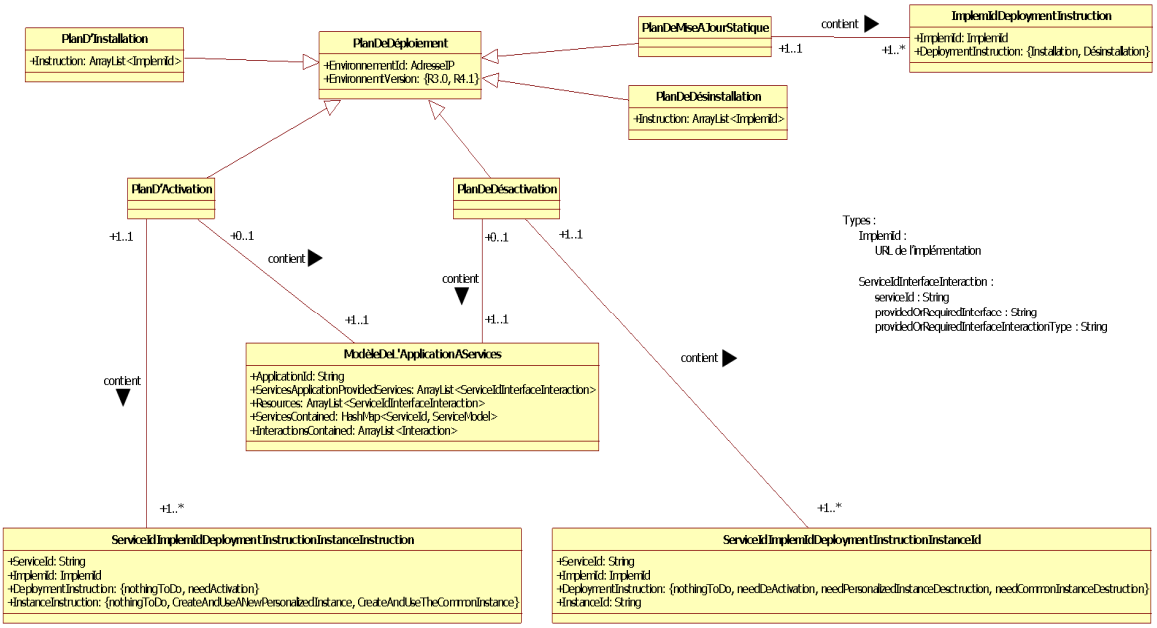


Figure 51. Méta-modèle pour les plans de déploiement

Le méta-modèle ci-dessus est complété par une contrainte qui stipule qu'une instance de la classe "ModèleD'ApplicationAServices" est associée soit à une instance de la classe "PlanD'Activation", soit à une instance de la classe "PlanDeDésactivation".

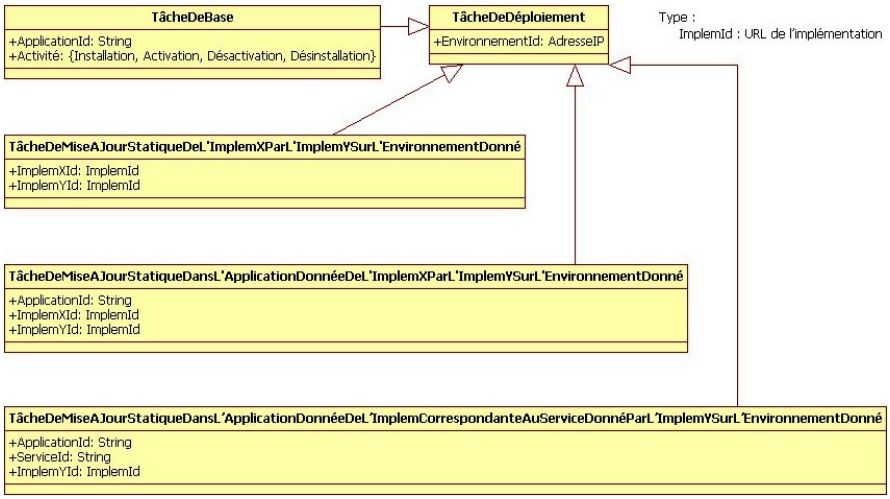


Figure 52. Méta-modèle pour les tâches de déploiement.

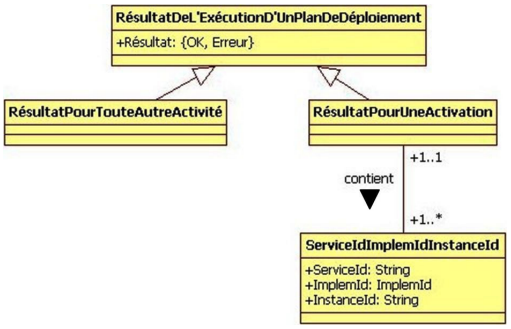


Figure 53. Méta-modèle pour les résultats de l'exécution d'un plan de déploiement.

Nous allons maintenant développer notre proposition autour de l'assurance de la conformité entre une application déployée et son modèle, c'est-à-dire entre la "réalité" et le modèle de l'application. Dans notre cadre, assurer la conformité signifie assurer la synchronisation descendante entre le modèle de l'application à déployer et l'application effectivement déployée sur l'environnement. Cette synchronisation est complexe puisque dans la technologie OSGi, la composition entre un demandeur et un fournisseur est basée sur une approche *trading*. Cela signifie que la sélection d'un fournisseur afin de satisfaire un demandeur est à la charge du demandeur lui-même, à l'image par exemple de tout un chacun choisissant un fournisseur d'accès internet. Or, dans le cadre des applications à services (où l'architecture de l'application doit être respectée), il n'est pas envisageable de laisser la sélection des fournisseurs aux demandeurs. En effet, les demandeurs n'ont pas la connaissance du niveau application ; par conséquent, ils sont incapables de sélectionner les fournisseurs de façon à ce que le modèle de l'application et donc son architecture soient respectés. La figure ci-dessous décline un exemple illustrant le problème posé par l'utilisation d'une approche *trading* dans le cadre des applications à services.

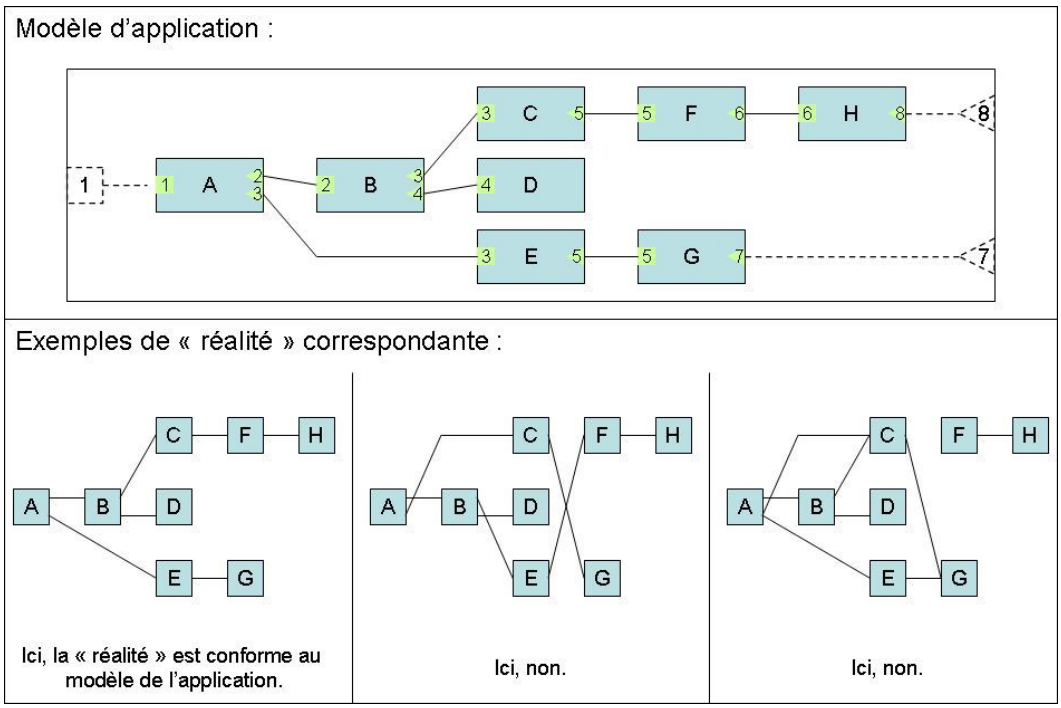


Figure 54. Problème posé par l'approche *trading* pour les applications à services.

Pour remarque, dans la technologie OSGi les demandeurs ne s'intéressent généralement qu'aux contrats des fournisseurs (c'est-à-dire à l'interface Java qu'ils offrent) et non aux services offrant les fournisseurs (c'est-à-dire aux ensembles constitués par un fournisseur et de possibles demandeurs). Bien que nous ne l'ayons pas représenté dans la figure précédente, il est important de noter que, dans la cadre de l'approche *trading*, tout demandeur appartenant à un service d'une application peut certes sélectionner

un fournisseur parmi ceux présents dans l'application, mais aussi parmi ceux présents dans l'environnement (c'est-à-dire parmi ceux présents hors de l'application elle-même !).

Pour pallier ce problème de conformité posé par l'utilisation d'une approche *trading* dans le cadre des applications à services, nous proposons d'introduire et d'utiliser une approche *naming*. Ainsi, au lieu qu'un demandeur sélectionne "à l'aveugle" un fournisseur, nous lui désignons le fournisseur qu'il doit utiliser pour respecter le modèle de l'application. En ce qui concerne les demandeurs qui sont exposés par l'application (par exemple le demandeur "8" du service "H" dans la figure ci-dessus), nous leur fournissons la liste des fournisseurs de services pouvant répondre à leur demande mais ne faisant pas partie des fournisseurs apparaissant dans l'application ; notre objectif est, ainsi, d'éviter qu'un demandeur exposé par l'application utilise un fournisseur appartenant, lui aussi, à l'application et perturbe donc l'application elle-même.

En complément, la propriété de conformité (c'est-à-dire de synchronisation descendante) que nous avons cherché à assurer est à comparer avec la synchronisation montante qui vise, elle, à propager des changements intervenant (et/ou intervenus) au niveau de l'application déployée (c'est-à-dire dans la "réalité") vers le modèle de l'application et le modèle de l'environnement (c'est-à-dire vers le niveau modèle). Dans le cadre de cette thèse, nous ne nous sommes pas intéressés à la synchronisation montante.

5.4 La sous-activité de mise à jour statique

Nous avons présenté la définition de la sous-activité de mise à jour statique d'une application à services dans une section précédente. Pour rappel, cette sous-activité désigne le remplacement d'une ou plusieurs implémentations de services actuelles/courantes (appartenant à l'application) par une ou d'autres implémentations de service lorsque l'application n'est pas activée (sachant que l'architecture de l'application doit être conservée, de même que les architectures des services ciblés par la mise à jour).

Dans notre contexte, nous avons investigué trois mises à jour statiques de portées différentes, à savoir :

- la mise à jour de l'implémentation de service correspondant à un modèle de service donné dans un modèle d'application à services donné par une "nouvelle" implémentation de service donnée dans un environnement d'exécution donné,
- la mise à jour statique d'une implémentation de service donnée (nommons la "IS") apparaissant dans un modèle d'application à services donné par une "nouvelle" implémentation de service donnée dans un environnement donné ("IS" réalisant au moins un modèle de service appartenant au modèle de l'application, mais pouvant en réaliser plusieurs)
- et, enfin, la mise à jour statique d'une implémentation de service donnée par une "nouvelle" implémentation de service donnée dans un environnement donné.

Pour être effectivement réalisables, les modèles applications ciblés ou touchés doivent être soit dans l'état de déploiement *installed*, soit dans l'état de déploiement *deactivated*. Dès qu'une de ces mises à jour est effectuée, les modèles d'application ciblés ou touchés passent tous dans l'état *installed*.

Enfin, le rôle du DMSA est de traduire chacune de ces mises à jour statiques en un ensemble d'activités d'installation et de désinstallation au niveau unité de déploiement/(implémentation de service) et d'exécuter l'ensemble généré dans l'environnement cible.

5.5 Le déploiement "transactionnel"

Le système de déploiement mis en place via le DMSA ne met à jour le modèle global associé à un environnement que si l'activité de déploiement tentée sur cet environnement s'est bien déroulée. L'idée est de protéger le modèle global de l'environnement. Cependant, en cas d'échec de l'activité de déploiement, la cohérence entre le modèle global de l'environnement et l'environnement (c'est-à-dire les implémentations et les instances, la "réalité") est, dans le cas général, compromise.

C'est pourquoi, nous avons mis en place un mécanisme qui, en cas d'échec de l'activité de déploiement, va tenter de ramener l'environnement dans l'état où il se trouvait avant l'exécution de l'activité de déploiement. Pour remarque, nous mettons volontairement des guillemets autour du mot "transactionnel" et nous utilisons volontairement les mots "tenter de ramener l'environnement dans l'état où il se trouvait". En effet, contrairement à un système transactionnel basé au-dessus d'une base de données où le système peut manipuler à volonté les données que la base contient tant qu'elle est accessible, un système transactionnel de déploiement peut échouer dans l'exécution d'un *rollback*. Par exemple, lors d'un *rollback* sur une activité de désinstallation ou de mise à jour statique, il est possible que des implémentations de service désinstallées ne puissent pas être (ré-) installées ; de même, lors d'un *rollback* sur une activité de désactivation, il est possible que des implémentations de services désactivées et des instances détruites ne puissent être respectivement (ré-) activées et (re-) créées (et ré-utilisées).

Dans le cadre du déploiement de modèles d'application à services et du fait que les activités de déploiement proposées par la technologie OSGi peuvent être considérées comme transactionnelles, un mécanisme de *rollback* pour chaque activité de déploiement au niveau modèle d'application à services consiste à exécuter les inverses des activités de déploiement aux niveaux implémentations et instances qui s'étaient bien déroulées jusqu'à l'apparition d'une erreur et cela dans l'ordre inverse de leur ordre d'exécution initial (l'inverse de l'installation est la désinstallation, l'inverse de l'activation est la désactivation, l'inverse de la désactivation est l'activation et enfin l'inverse de la désinstallation est l'installation). Pour chaque activité de déploiement au niveau modèle d'application à services, le mécanisme de *rollback* n'est déclenché que lorsqu'une activité de déploiement résultant de la traduction de l'activité de déploiement au niveau modèle d'application à services, échoue.

6. DÉPLOIEMENT DE MODÈLES D'APPLICATION SUR UN ENV.

Dans cette section, nous allons présenter notre proposition autour du déploiement de (plusieurs) modèles d'application à services sur un même environnement d'exécution. Notre objectif est d'étudier et de gérer le partage d'implémentations de services, ainsi que l'isolation entre les applications et le partage d'instances d'implémentations de services.

6.1 Le partage d'implémentations de services

L'inconvénient du déploiement de plusieurs applications à services dans un même environnement d'exécution à services (en même temps) est que ces dites applications peuvent nécessiter et utiliser conjointement une ou plusieurs implémentations de services.

Dans notre contexte, le partage d'implémentation de services entre (modèles d') applications (voire même entre plusieurs modèles de services du même modèle d'application) doit être géré à l'installation et à la désinstallation, mais aussi lors des mises à jour statiques (qui sont des combinaisons d'installations et de désinstallations) et lors des activités d'activation et de désactivation. La gestion du partage d'implémentations pour les activités d'installation et de désinstallation est une problématique qui a déjà été traitée par Richard S. Hall (prototype Software Dock), via l'introduction d'un compteur associé à chaque implémentation de services [HHW99].

Comme nous l'avons précisé ci-dessus, nous ciblons les activités de mises à jour statiques, d'activation et de désactivation en plus des activités d'installation et de désinstallation. Qui plus est, notre propos n'est pas de maintenir un compteur pour chaque implémentation. En effet, nous voulons maintenir, pour chaque implémentation de service déployée, quel modèle de service de quel modèle d'application la nécessite et l'utilise [CL07a]. Pour rappel, nous utilisons le verbe "nécessiter" pour désigner une implémentation de service associée à un modèle de service appartenant à un modèle d'application installé ou désactivé ; nous employons le verbe "utiliser" pour désigner une implémentation de service associée à un modèle de service appartenant à un modèle d'application activé.

Le défi de la gestion du partage d'implémentation de services consiste donc, par exemple, à ne pas installer une implémentation déjà installée, à ne pas activer une implémentation déjà activée, ainsi qu'à ne (surtout) pas désactiver une implémentation encore utilisée et à ne pas désinstaller une implémentation encore nécessité ou (pire) encore utilisée. La gestion du partage d'implémentations de services a une influence directe sur la génération des plans de déploiement proposée par le DMSA.

6.2 L'isolation entre les applications

Du fait du contexte embarqué des environnements d'exécution que nous ciblons, nous avons fait le choix de permettre le partage des implémentations de services entre modèles d'application à services, ainsi qu'entre modèles de services appartenant au même modèle d'application. Il nous faut donc gérer l'isolation entre les applications (et entre les services) qui partagent une même implémentation de service [CL07a].

Répondre à ce défi n'a été possible que via la prise en compte du niveau instance d'implémentation de service. Grâce à ce niveau instance, nous avons pu fournir un gestionnaire d'isolation. Il intervient lors des activités d'activation et de désactivation. Il se focalise sur la création, l'attribution et la destruction d'instances. Dans le cadre de la gestion de l'isolation, la décision de créer et d'attribuer une nouvelle instance est prise lors de l'activité d'activation. C'est une décision systématique. Nous verrons dans la sous section suivante que nous avons aussi appréhendé le partage d'instances d'implémentations de services et qu'ainsi nous avons tempéré quelque peu le systématisme de la décision de création et d'attribution d'instances. La décision de destruction d'instance est, elle, prise lors de l'activité de désactivation. Elle est systématique et ne peut être tempérée, sauf peut être via l'utilisation d'un *pool* d'instances (cela dit la technologie OSGi ne fait aucune proposition dans ce sens).

La gestion de l'isolation a une influence directe sur la génération des plans de déploiement proposée par le DMSA.

6.3 Le partage d'instances d'implémentations de services

Suite à notre étude du partage d'implémentations de services, au système de gestion correspondant que nous proposons et du fait du contexte embarqué des environnements que nous ciblons, nous avons étudié le partage d'instances d'implémentations de services [CL07a]. Tout comme le partage d'implémentations de services, le partage d'instances d'implémentations de services concerne le partage d'une même instance entre des modèles de services appartenant à des modèles d'application différents, mais aussi entre des modèles de services appartenant au même modèle d'application.

Le problème majeur du partage d'instances dans le cadre d'applications concerne le "routage" des communications/interactions. Par exemple, comme le montre la figure ci-dessous, les modèles de services B et X qui appartiennent respectivement aux modèles d'application M1 et M2 sont similaires, donc leurs contrats de services sont eux aussi similaires, ainsi ils peuvent partager la même implémentation de service (i_23) mais aussi, sous certaines conditions, la même instance (in_23). Or, pour que les applications M1 et M2 ne se perturbent pas entre elles (c'est-à-dire pour qu'elles soient bien isolées l'une de l'autre) tout en partageant une ou plusieurs instances, il est nécessaire que :

- L'utilisation d'un fournisseur de contrat 3 qui est déclenchée par une communication/interaction entre in_12 et in_23 (c'est-à-dire qui apparait à la suite d'une interaction dans le cadre de l'application M1) se fasse bien via l'utilisation du fournisseur de contrat 3 de l'instance in_3.
- De même, si une interaction entre in_35 et in_23 (c'est-à-dire une interaction ayant lieu dans le cadre de l'application M2) donne lieu à une interaction avec (c'est-à-dire à un envoi de données vers) un demandeur de contrat 2, alors le demandeur de contrat 2 qui doit être utilisé est celui de l'instance in_42 et cela afin que les interactions relatives au modèle d'application M2 reste bien dans le cadre de M2.

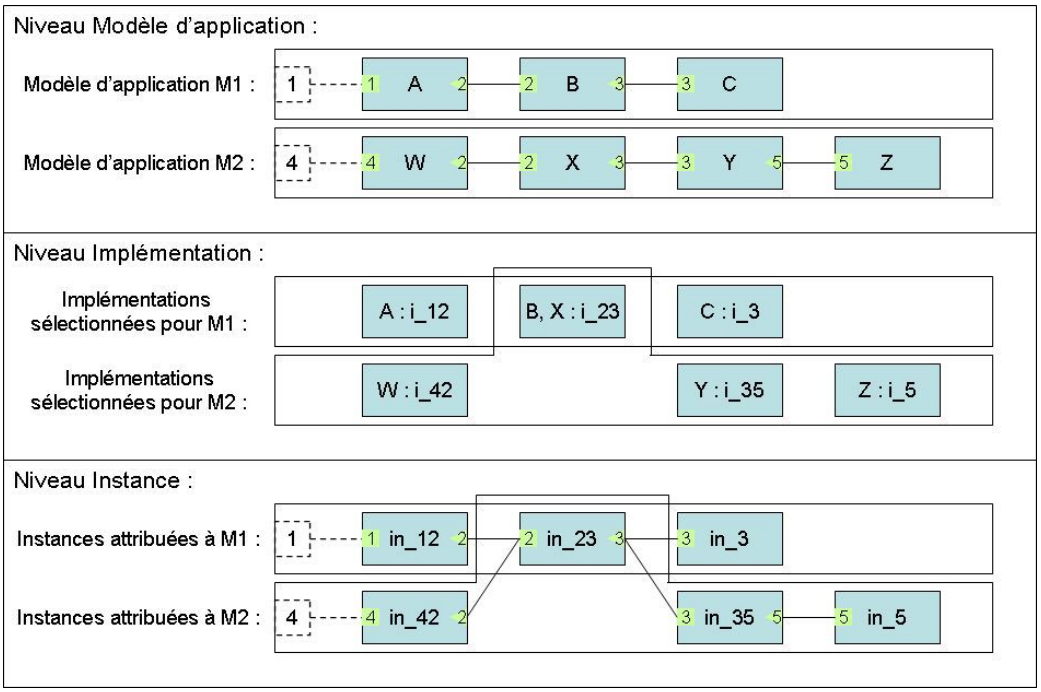


Figure 55. Problème du "routage" des communications lors du partage d'instances.

Or, la mise en place d'un mécanisme permettant un tel routage est très couteux. En effet, toute communication (c'est-à-dire tout appel de méthode, publication ou transfert de message) doit être identifiée, ensuite :

- soit toute communication passe par un tiers qui connait les quatre niveaux (modèle d'application, modèle de service, implémentation et instance) et qui est donc capable de router les communications entre les instances émettrices/utilisatrices et les instances réceptrices/utilisées et cela en respectant les modèles d'application dans lesquels ont lieu les communications,
- soit toute instance utilisée doit connaitre (en interne) dans le cadre de quelle(s) application(s) elle est utilisée, ainsi que dans le cadre de quelle application chaque instance qui émet vers elle (ou qui l'utilise) le fait et dans le cadre de quelle application elle est amenée à émettre vers (ou à utiliser) telle (ou telle) instance.

Il est aussi important de noter que l'utilisation d'un tel mécanisme est lui aussi très couteux, car :

- soit toute communication se traduit par une communication vers le tiers qui calcule et redirige ensuite la communication vers la bonne instance,
- soit c'est à l'instance elle-même de calculer puis de cibler la bonne instance.

En complément, l'utilisation d'un tiers parait être plus simple puisqu'à chaque activité de déploiement, la mise à jour des connaissances (c'est-à-dire des informations relatives aux quatre niveaux modèle d'application, modèle de service, etc.) doit se faire sur une seule entité centrale : le tiers. Au contraire, si les connaissances sont distribuées dans les instances, leur mise à jour sera elle aussi distribuée et donc plus complexe.

En conséquence, nous nous sommes concentrés sur le partage d'instances dont le modèle (et donc dont le contrat) de service ne possède pas de demandeur.

De manière abstraite, une instance d'implémentation de service peut être caractérisée par la fonction suivante O = f (I, S, C, RD), où :

- O ou *output* représente les données retournées par le fournisseur appartenant à l'instance,
- I ou *input* représente les données fournies au fournisseur appartenant à l'instance,
- S ou *state* ou *internal state* représente l'état interne de l'instance,
- C ou *configuration* représente la configuration de l'instance,
- RD ou *requesters' data* représente les données récupérées par le ou les demandeurs appartenant à l'instance, ces données sont nécessaires au "calcul" de O. Sachant que les RD peuvent être considérées comme faisant partie de I.
- f représente la fonction, l'algorithmique réalisée par l'instance.

La fonction correspondant aux instances que nous ciblons dans le cadre du partage d'instances, c'est-à-dire des instances sans demandeur (ni configuration), est O = f (I, S).

Donc, le partage d'une instance n'est tributaire que de la présence ou non de "S", c'est-à-dire d'un état interne dans l'instance.

En conséquence, les modèles d'implémentations de services qui sont enregistrés dans le dépôt de modèles d'implémentations de services contiennent un *tag* "stateType" qui précise si l'implémentation correspondante à un état interne ou non (pour être tout à fait exact, nous devrions nous intéresser à la présence d'un état interne qui influence, ou non, "O" ; cela dit quel est le sens d'un état interne qui n'influence par "O" ?). Le *tag* "stateType" peut prendre soit la valeur "withState" qui signifie que l'implémentation de service correspondante a un état interne (sa fonction est donc O = f (I, S)), soit la valeur "withoutState" qui signifie que l'implémentation de service correspondante n'a pas d'état interne (sa fonction est donc O = f (I)).

Au final, seules les implémentations de services dont l'état interne n'influence pas les données qu'elles émettent ou qu'elles retournent (c'est-à-dire dont la fonction est O = f(I)) peuvent voir une (seule) de leur instance partagée entre plusieurs modèles de services appartenant indifféremment au même modèle d'application et/ou à des modèles d'application différents. La gestion du partage d'instances d'implémentations de services a une influence directe sur la génération des plans de déploiement proposée par le DMSA.

La figure ci-dessous présente le méta-modèle que nous avons défini pour les implémentations de services.

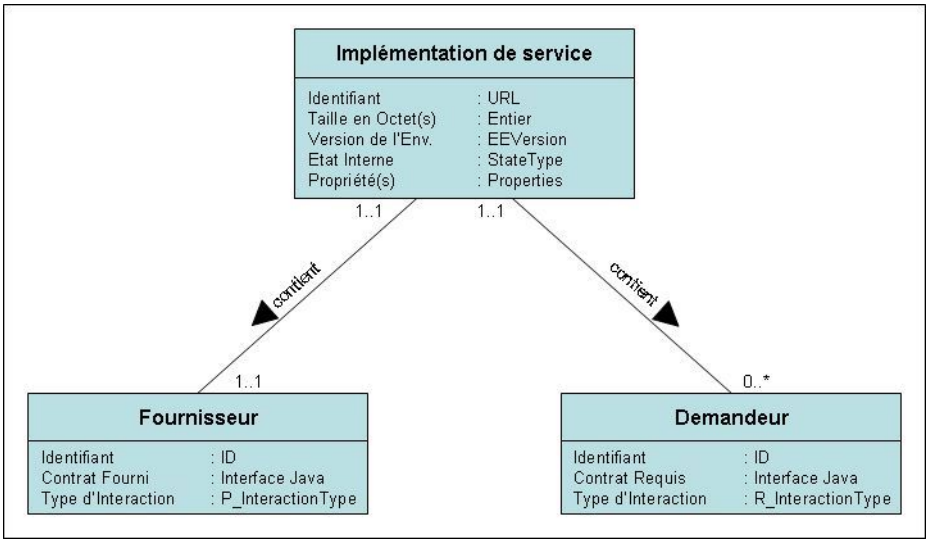


Figure 56. Méta-modèle pour les implémentations (c'est-à-dire pour les unités de déploiement)

En complément, le type EEVersion définit les valeurs "R3.0" et "R4.1". Le type StateType définit les valeurs "WithState" et "WithoutState". Le type "P_InteractionType" définit le type d'interaction des fournisseurs (c'est-à-dire des *providers*), les valeurs correspondantes sont "Server", "Publisher" et "Producer". Le type "R_InteractionType" définit le type d'interactions des demandeurs (c'est-à-dire des *requesters*), les valeurs correspondantes sont "Client", "Subscriber" et "Consumer".

7. APPRÉHENSION DU CONTEXTE EMBARQUÉ DES ENV.

Nous présentons, ici, notre proposition concernant l'appréhension du contexte pervasif des passerelles et embarqué/réactif des environnements d'exécution. L'objectif de notre proposition est de minimiser la taille des modèles d'application à services déployés (nous parlons ici en espace disque) et le nombre d'implémentations de services packagées (c'est-à-dire d'unités de déploiement) déployées dans chaque environnement géré via le DMSA.

Pour ce faire, nous avons décidé de maximiser la réutilisation des implémentations de services, cela signifie qu'il ne s'agit plus ici de se contenter de partager des implémentations de services, mais de maximiser ce partage. Ainsi, lors de l'étape de sélection des implémentations de services qui réaliseront un modèle d'application à services (c'est-à-dire lors de l'installation d'un modèle d'application dans un environnement), l'algorithme de sélection sélectionne en priorité les implémentations de services d'ors et déjà déployées dans l'environnement et si aucune implémentation d'ors et déjà déployée n'est satisfaisante, alors, il sélectionne l'implémentation de services de plus faible poids. Pour rappel, lorsqu'une implémentation de service et son modèle sont enregistrés dans le dépôt de modèles d'implémentations de services, le DMSA récupère de lui même le poids en octets de l'implémentation de service. Concrètement, un modèle d'implémentation de service passé comme paramètre lors d'un enregistrement ne contient pas le *tag* " sizeInOctet" ; une fois que le modèle est enregistré, il contient ce *tag* et la valeur associée correspond au poids (du ".jar") identifié/relevé par le DMSA.

L'appréhension de notre contexte embarqué passe aussi par la maximisation du partage d'instances, cependant, comme nous l'avons détaillé dans la section précédente, du fait de la technologie OSGi, nous ne pouvons considérer que le partage des instances qui n'ont pas d'état interne et qui n'ont pas de demandeurs. Malgré cette limite très contraignante, la partie de notre DMSA qui est en charge de la traduction des activités au niveau modèle d'application vers des activités aux niveaux implémentation et instance tient compte de ce partage et est programmée pour le maximiser.

L'appréhension de notre contexte embarqué a, lui aussi, une influence directe sur la génération des plans de déploiement proposée par le DMSA.

8. EXÉCUTION D'ACTIVITÉS DE DÉPLOIEMENT SUR DES ENV.

Dans cette section, nous allons présenter nos deux dernières propositions. La première cible le déploiement de modèles d'application à services sur des environnements d'exécution à services. La seconde vise l'exécution d'une liste ordonnée d'activités de déploiement concernant des modèles d'application à services (c'est-à-dire l'exécution de plusieurs activités de déploiement dans un ou plusieurs environnements d'exécution).

8.1 Le déploiement sur plusieurs environnements

L'objectif de cette proposition est de permettre la gestion du déploiement sur plusieurs environnements d'exécutions. Attention, il n'est pas question ici d'appréhender le déploiement de modèles d'application distribuées, mais d'étoffer le DMSA afin qu'il puisse gérer plusieurs environnements et par conséquent qu'il puisse déployer des modèles d'application sur plusieurs environnements.

Dans notre contexte, nous sommes amenés à cibler des environnements d'exécution OSGi de version R3.0 et R4.1. Les environnements eux-mêmes sont identifiés par leur adresse IP.

Concrètement, pour réaliser cette proposition, nous avons introduit un gestionnaire d'environnements d'exécution dans le DMSA. Ce gestionnaire permet d'ajouter un environnement à la liste des environnements gérés par le DMSA (cet ajout se fait via l'adresse IP de l'environnement, la version de l'environnement doit aussi être renseignée). Il permet aussi de tester si l'environnement est accessible et si il contient bien la partie "embarquée" du DMSA. Il est aussi possible de lister les environnements gérés, de rechercher un environnement et de supprimer un environnement. Enfin, ce gestionnaire intervient dans l'association d'un modèle global à chaque environnement et donc lors de l'exécution de chaque activité de déploiement.

8.2 L'exécution de plusieurs activités de déploiement

L'objectif de cette dernière proposition est de permettre l'exécution d'une liste ordonnée d'activités de déploiement. En outre, nous proposons un algorithme d'exécution plus performant que l'algorithme d'exécution séquentiel trivial.

Comme pour l'exécution d'une seule activité de déploiement, ici, c'est l'administrateur qui spécifie la liste d'activités de déploiement à exécuter. Il est important de noter que cette liste est ordonnée, plus précisément, cela signifie que la première activité de la liste est exécutée avant la seconde, que la seconde l'est avant la troisième, etc. Concrètement chaque activité de déploiement apparaissant dans une telle liste est composée soit par un quadruplet, pour les activités d'installation, d'activation, de désactivation et de désinstallation, soit par un quintuplet, dans le cas de l'activité de mise à jour statique d'une implémentation de service par une autre implémentation de service, soit par un sextuplet pour la mise à jour statique d'une implémentation de service par une autre implémentation de service dans une application et pour la mise à jour de l'implémentation de service correspondant à un service d'une application par une autre implémentation de service.

Concernant les activités définies par les quadruplets, quintuplets et sextuplets, nous utilisons les termes "tâche de déploiement" pour les désigner. Un quadruplet est constitué de quatre champs qui contiennent respectivement l'identifiant de la tâche de déploiement, l'activité de déploiement ciblée, l'identifiant de l'environnement ciblé et l'identifiant du modèle d'application ciblé. Un quintuplet contient l'identifiant de la tâche de déploiement, l'activité de déploiement, l'identifiant de l'environnement, l'identifiant de l'implémentation de service à mettre à jour et l'identifiant de la nouvelle implémentation de service. Enfin, un sextuplet contient l'identifiant de la tâche de déploiement, l'activité de déploiement, l'identifiant de l'environnement cible, puis soit l'implémentation de service à mettre à jour, ainsi que la nouvelle implémentation de service et l'identifiant de l'application ciblée, soit l'identifiant du service ciblé

dans l'application, l'identifiant de la nouvelle implémentation de service et l'identifiant de l'application ciblée. Pour rappel, chaque implémentation de service est une unité de déploiement.

Conjointement, l'exécution de plusieurs activités de déploiement doit respecter deux contraintes. La première provient de notre contexte. Elle stipule que l'ordre défini dans la liste doit se retrouver au niveau de l'exécution des activités de déploiement (c'est-à-dire au niveau de l'exécution des tâches de déploiement) et cela environnement par environnement. Par exemple, si une liste ordonnée contient les tâches identifiées par "1", "2", "3", "4" et "5" et si "1", "2" et "5" ciblent l'environnement d'identifiant "A" et "3" et "4" ciblent l'environnement "B", alors les tâches "1", "2" et "5" devront être exécutées dans cet ordre sur "A" et les tâches "3" et "4" devront être exécutées dans cet ordre sur "B" ; par contre l'ordre d'exécution global des tâches pourra être, par exemple, "1", "2", "3", "4", "5" ou "3", "1", "4", "2", "5" ou "3", "4", "1", "2", "5" ou les sous listes ["1", "2", "5"] et ["3", "4"] en parallèle ou etc.

La seconde contrainte provient du fait que le déploiement que nous proposons est un déploiement sensible au contexte. En conséquence, il n'est pas possible d'exécuter simultanément (c'est-à-dire en parallèle) deux tâches de déploiement sur un même environnement.

Pour remarque, le déploiement en général impose au moins une contrainte qui stipule qu'il n'est pas possible, dans le cas général, d'exécuter en parallèle deux activités de déploiement ciblant la même application. Par exemple, il n'est généralement pas possible d'installer et d'activer une application en parallèle, il n'est pas non plus possible de la mettre à jour statiquement et de l'activer, ou de la désactiver et de la mettre à jour statiquement ; de même il n'est pas non plus possible de l'activer et de la désactiver ou de l'installer et de la désinstaller dans le même temps.

Notre architecture de déploiement est une architecture centralisée (c'est-à-dire basée sur un serveur central). La partie qui est hébergée sur ce serveur central est chargée de calculer et d'optimiser les plans de déploiement afin de gérer le partage d'implémentations, le partage d'instances et l'isolation. Une des conséquences de cette architecture centralisée est qu'elle ne permet pas d'exécuter en parallèle deux tâches de déploiement (ciblant ou non la même application) dans le même environnement.

A contrario, si une architecture est décentralisée, ce qui signifie que les décisions relatives aux partages et à l'isolation sont prises par la partie de l'architecture qui est embarquée dans chaque environnement, alors il semble possible d'exécuter en parallèle toute tâche de déploiement ciblant un même environnement.

En conséquence, si l'objectif est la minimisation du temps pris par l'exécution d'une liste de tâches de déploiement ou la maximisation de la parallélisation de l'exécution des tâches de déploiement, alors il faut préférer une architecture décentralisée à une architecture centralisée.

Enfin, il est important de noter qu'une architecture décentralisée nécessite la persistance de données dans les environnements, or les environnements d'exécution n'offrent pas toujours une telle possibilité (c'est typiquement le cas dans notre contexte).

En ce qui concerne l'exécution effective des tâches de déploiement spécifiées dans les listes, nous fournissons deux algorithmes : l'algorithme séquentiel (trivial) et un algorithme parallèle.

L'algorithme séquentiel exécute les tâches de déploiement les unes à la suite des autres et dans l'ordre où elles ont été spécifiées dans la liste donnée en paramètre. Une fois toutes les tâches exécutées, le DMSA affiche l'identifiant de chaque tâche, ainsi que le résultat de son exécution. Si l'exécution d'une tâche échoue, le DMSA stoppe l'algorithme (cela signifie que les tâches non encore exécutées ne seront pas exécutées) et affiche les résultats des tâches exécutées, de la tâche dont l'exécution a échoué et les identifiants des tâches non exécutées.

Le temps théorique pris/consommé par l'exécution de toute liste est égal à la somme des temps d'exécution pris pour l'exécution de chaque tâche de la liste. Cet algorithme est le plus basic et le plus évident des algorithmes.

L'algorithme parallèle que nous proposons vise à minimiser le temps consommé par l'exécution d'une liste d'activité. Pour cela, nous avons choisi de maximiser la parallélisation des tâches tout en respectant les deux contraintes énoncées ci-avant (pour rappel, la première contrainte concerne respect de l'ordre spécifié dans la liste, la seconde contrainte provient de notre déploiement sensible au contexte). En conséquence, seules les tâches ciblant des environnements différents peuvent être parallélisées.

L'algorithme de nous proposons est une adaptation de l'algorithme *Work Stealing* [BFG01]. L'algorithme *Work Stealing* est un algorithme "glouton", il a été conçu pour exécuter un ensemble de tâches en parallèle, sachant que le nombre de threads est choisi arbitrairement, qu'il n'y a aucune dépendance entre les tâches et que les tâches sont distribuées de façon arbitraire entre les threads avant le début de l'exécution. La caractéristique principale de cet algorithme est que lorsqu'un thread n'a plus de tâche à exécuter/consommer, alors il va voler une tâche à un autre thread choisi aléatoirement. Plus précisément, il va voler la tâche que le thread "victime" aurait exécuté après avoir fini sa tâche courante.

L'algorithme, que nous proposons, adapte automatique le nombre de threads mis en jeu en fonction du nombre d'environnements ciblés par les tâches de la liste. Ainsi, chaque tâche ciblant le même environnement est inséré dans la liste de tâches ciblant l'environnement et cela tout en conservant l'ordre spécifié par la liste. Les différents threads sont ensuite exécutés (en parallèle). Une fois leur exécution finie, un compte rendu des exécutions environnement par environnement est affiché.

Notre adaptation est plutôt importante puisque le choix du nombre de threads ainsi que la distribution des tâches ne sont plus aléatoires et puisque le vol de tâches entre threads n'est plus permis (du fait des contraintes de notre contexte et du déploiement sensible au contexte que nous proposons).

Le temps théorique pris pour l'exécution de toute liste est égal au maximum de l'ensemble constitué par les sommes des temps d'exécution pris pour les tâches ciblant un même environnement d'exécution. Par conséquent, notre algorithme est plus efficace que l'algorithme séquentiel uniquement pour les listes qui ciblent au moins deux environnements ; dans le cas contraire les deux algorithmes sont équivalents.

Enfin, il est à noter qu'il doit être possible d'attribuer plusieurs threads pour les tâches ciblant un même environnement, cependant, cela nécessite un mécanisme d'ordonnancement dynamique (c'est-à-dire qui ordonnancerait les tâches attribuées aux threads tout au long de leur exécution). En outre, le choix du nombre de threads attribués par environnement s'avérerait particulièrement complexe.

9. SYNTHÈSE

Dans ce chapitre, nous avons développé les différentes propositions que nous faisons dans le cadre de cette thèse. Ainsi, nous avons développé notre approche, notre méta-modèle pour les applications à services, notre proposition autour de la définition du déploiement pour les services et les applications à services. Nous avons aussi présenté l'architecture que nous avons mise en place pour le DMSA (c'est-à-dire pour le système de déploiement que nous proposons), ainsi que nos propositions autour du déploiement de modèles d'application à services sur des environnements d'exécution, notre appréhension du contexte embarqué/réactif des environnements d'exécution et, enfin, notre proposition concernant l'exécution de listes d'activités/tâches de déploiement.

Pour remarque, il est admis que les informaticiens choisissent un modèle de développement (et en acceptent les contraintes) afin d'obtenir des propriétés de développement, de même, il faut prendre conscience que pour obtenir des propriétés de déploiement, il faut choisir un modèle de déploiement et en accepter les contraintes.

Enfin, le tableau synthétique ci-dessous présente le DMSA du point de vue des points de caractérisations que nous avons utilisés dans notre état de l'art sur le déploiement logiciel.

	DMSA
Définition de la phase de déploiement	Processus non linéaire et non séquentiel : ***Install***, ***Activate***, ***DeActivate***, ***(Static et Dynamic) Update***, ***Evolution***, ***Dynamism*** et ***DeInstall***.
Unité de développement	**Service** (grain fin)
Unité de déploiement	**Implémentation de service *packagée*** (grain fin).
Point d'entrée de la phase de déploiement	**Modèle d'application à services**, ainsi qu'un dépôt de modèles d'implémentations de services et un gestionnaire d'environnements d'exécution.
Résolution de dépendances et validation	**Résolution des dépendances de services :** dans notre contexte, cela revient à sélectionner (en étant sensible au contexte de l'environnement cible) et attribuer une implémentation de service par modèle de service présent dans un modèle de l'application validé. **Validation** des modèles d'applications à services.
Partage d'implémentation, Gestion de ce partage.	**Oui**, toute implémentation de services peut être partagée entre des modèles de services appartenant ou non au même modèle d'application à services. **Oui**, la gestion de ce partage est réalisée par la partie serveur du DMSA.
Partage d'instance d'implémentation, Gestion de ce partage.	**Oui**, les instances d'implémentations de services sans requis et sans état interne peuvent être partagées entre des modèles de services. **Oui**, la gestion de ce partage est réalisée par la partie serveur du DMSA.
Gestion de l'isolation	**Oui**, la gestion de l'isolation est réalisée lors de la génération des plans de déploiement.
Remarque	L'intégration actuelle de la partie agent du DMSA avec la technologie OSGi R4.1 est limitée concernant le niveau instance. En effet, les fabriques d'instances proposées au-dessus d'OSGi ne sont pas satisfaisantes. En conséquence, la partie agent ne manipule que des instances singletons. La validation du fonctionnement de la partie agent du DMSA pour le partage d'instances et l'isolation n'a pas pu être validée. La partie agent du DMSA possède un mécanisme pour rattraper les erreurs intervenant au niveau implémentation lors de l'exécution d'un plan de déploiement.

Tableau 9. Application, au DMSA, des points de caractérisations définis dans l'état de l'art.

TROISIÈME PARTIE : REALISATION

V. IMPLÉMENTATION

Dans ce chapitre, nous allons présenter les différents algorithmes et architectures que nous avons conçus et développés pour implémenter notre *Deployment Manager for Services Applications* (DMSA). Comme nous l'avons précisé dans le chapitre précédent, le DMSA est constitué de deux parties. La première est la partie "serveur" ; elle propose une IHM succincte à l'administrateur et elle est à l'origine de la génération des plans de déploiement qui seront exécutés dans les environnements. La seconde est la partie "agent". Elle est embarquée dans chaque environnement ciblé par le DMSA ; elle est en charge d'exécuter les plans de déploiement reçus et de retourner le résultat de leur exécution. Du fait du support des versions R3 et R4.1 de la technologie OSGi, nous proposons une partie "agent" adaptée à chaque version.

Ce chapitre est structuré en deux sections. Nous commençons par présenter l'implémentation de la partie serveur. Pour ce faire, nous détaillons les beans et servlets correspondants, l'algorithme d'ordonnancement des modèles d'application à services et les diagrammes de séquence des activités de déploiement et des algorithmes d'exécutions de tâche de déploiement supportés par le DMSA. Ensuite, dans la seconde section, nous détaillons la partie agent. Plus précisément, nous explicitons l'architecture JMX sur laquelle elle est construite, la limitation actuelle de l'implémentation de la partie agent sur OSGi R4.1, le diagramme de séquence de l'exécution d'un plan de déploiement, le mécanisme de récupération des erreurs au niveau implémentation de service qui est une première étape vers l'établissement d'un mécanisme "transactionnel" pour l'exécution des plans de déploiement et, pour finir, nous présentons la partie agent du DMSA au-dessus d'OSGi R3. Enfin, nous clôturons ce chapitre par une conclusion.

1. PARTIE "SERVEUR" DU DMSA

La partie serveur du DMSA a été implémentée au-dessus de la technologie JEE EJB V3.0. En l'occurrence, nous avons utilisé le serveur JONAS qui est un serveur *open-source* téléchargeable à l'adresse http://jonas.objectweb.org/. Pour être tout à fait précis, nous utilisons la version 4.7.4 de JONAS dans laquelle un conteneur EJB V3.0 a été inséré. Ce conteneur est nommé EasyBeans, il est téléchargeable à l'adresse : http://www.easybeans.org/.

Cette section est composée de trois sous-sections. La première se focalise sur la présentation des beans développés, la suivante présente les diagrammes de séquences correspondant aux activités de déploiement supportées par le DMSA et la dernière se concentre sur les diagrammes de séquences relatifs au déploiement de listes ordonnées de tâches de déploiement.

1.1 Diagramme des *beans* (EJB v3.0)

Nous allons maintenant présenter les *beans* que nous avons implémentés afin de réaliser la partie "serveur" du DMSA. Pour gagner en clarté, la première figure ci-dessous ne se focalise que sur les *beans* intervenant dans les activités d'installation, d'activation, les seconde et troisième figures ne se focalisent, elles, respectivement que sur les *beans* relatifs d'un part à la désactivation et à la désinstallation et d'autre part aux trois mises à jour statiques.

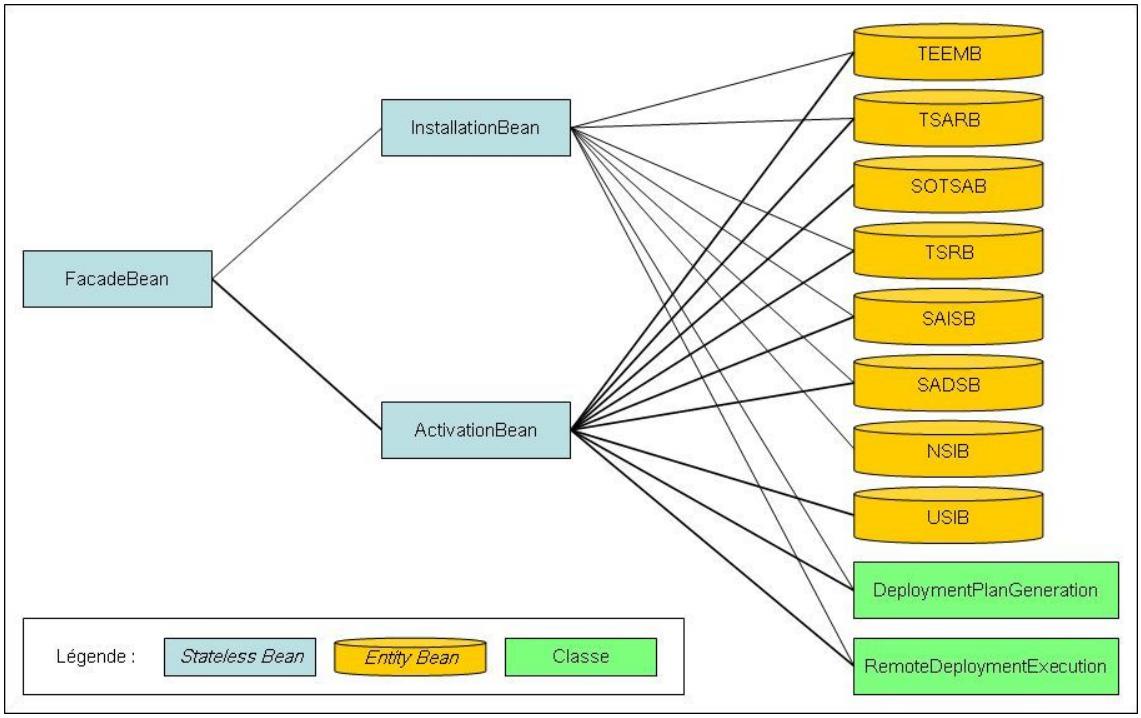


Figure 57. Diagramme des *beans* pour les activités d'installation et d'activation.

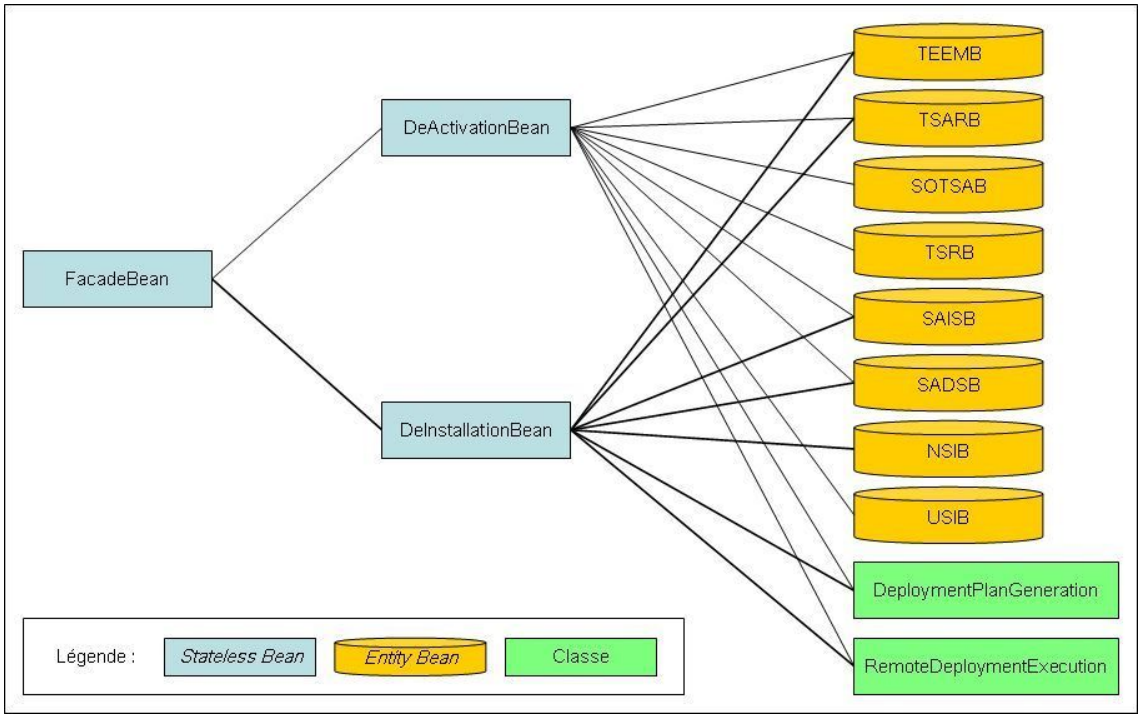


Figure 58. Diagramme des *beans* pour les activités de désactivation et de désinstallation.

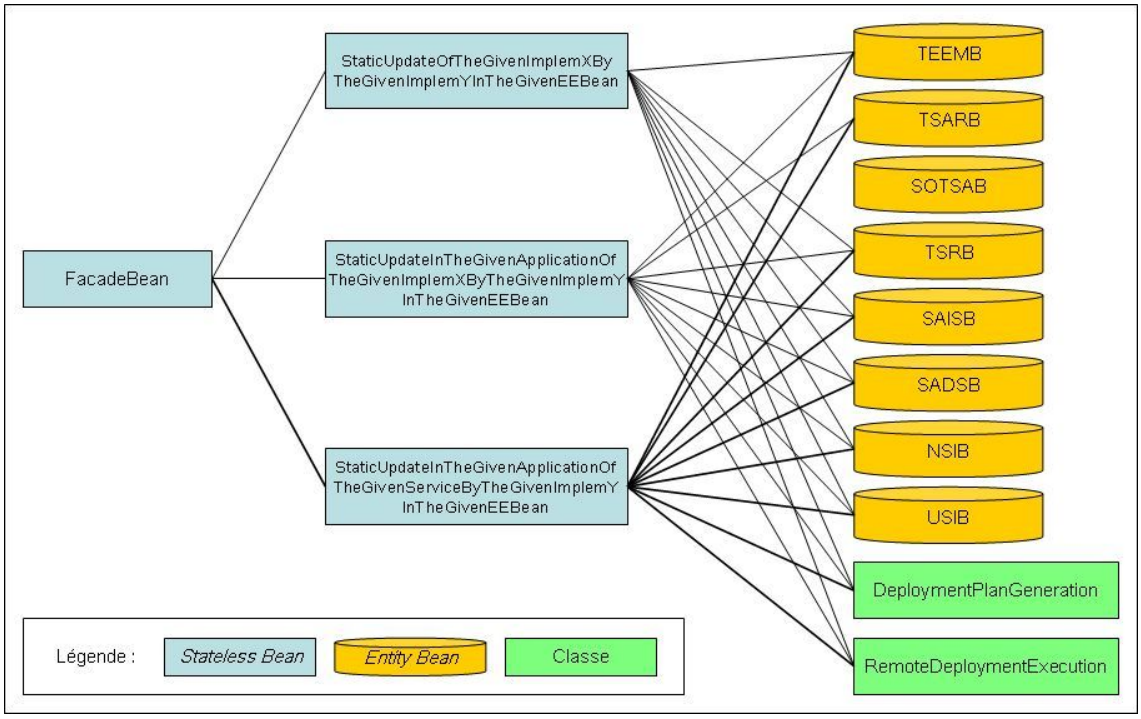


Figure 59. diagramme des *beans* pour les trois mises à jour statiques.

Dans les trois figures précédentes, les abréviations NSIB, SADSB, SAISB, SOTSAB, TEEMB, TSARB, TSRB et USIB signifient respectivement :

- "NeededServicesImplementationsBean" pour les informations concernant les implémentations de services nécessaires pour les applications déployées environnement par environnement ; ses attributs (qui définissent les colonnes de la table de données correspondant à l'*entity bean*) sont un String correspondant à un identifiant d'environnement d'exécution et une HashMap<String,

HashMap<String, ArrayList<String>>> où les trois "String" correspondent, dans l'ordre, à un identifiant de modèle d'implémentation de service, à un identifiant de modèle d'application et, enfin, à un identifiant de modèle de service appartenant à l'application,

- "ServicesApplicationsDeploymentStateBean" pour les informations relatives à l'état de déploiement des modèles d'application déployés environnement par environnement, ses attributs sont un String correspondant à un identifiant d'environnement et une HashMap<String, String> où le premier String correspond à une identifiant de modèle d'application et où le second est relatif à l'état de déploiement de l'application.

- "ServicesApplicationsImplementationsSelectionBean" pour les informations correspondant aux (modèles d') implémentations de services sélectionnées pour les modèles d'application déployés environnement par environnement, ses attributs sont un String correspondant à un identifiant d'environnement et une HashMap<String, ArrayList<CoupleApplicationIdServiceId>> où le String correspond à un identifiant de modèle d'implémentation de service et où le CoupleApplicationIdServiceId correspond à un couple associant un identifiant de modèle d'application et un identifiant de modèle de service appartenant à cette application,

- "SchedulingsOfTheServicesApplicationsBean" pour les ordonnancements (pour l'activation) calculés pour les modèles d'application lors de leur enregistrement dans le dépôt de modèles d'application du DMSA, ses attributs sont un String qui correspond à un identifiant de modèle d'application et une ArrayList<String> qui décrit l'ordonnancement de l'application (le premier élément de l'ArrayList doit être activé avant le second élément, etc.) où le String désigne un identifiant d'un modèle de service appartenant à l'application,

- "ExecutionEnvironmentsManagerBean" pour les environnements d'exécutions enregistrés dans le gestionnaire d'environnements d'exécution du DMSA, ses attributs sont deux Strings, le premier correspond à un identifiant d'environnement (dans notre cas, nous identifions un environnement par son adresse IP), le second renseigne la version de l'environnement, c'est-à-dire, soit R3, soit R4.1,

- "TheServicesApplicationsRepositoryBean" pour les modèles d'application enregistrés dans le dépôt de modèles d'application du DMSA. Ses attributs sont un String qui correspond à un identifiant de modèle d'application, une ArrayList<Triplet> qui décrit les services fournis par l'application, une seconde ArrayList<Triplet> qui décrit, elle, les services requis par l'application, une HashMap<String, ServiceOfAServicesApplicationOfTheRepository_DTO> dont la clé désigne un identifiant de modèle de service appartenant à l'application et dont la valeur est le modèle du service dans l'application. Enfin, le dernier attribut est une ArrayList<Interaction>, sachant que la classe Interaction est un triplet de Strings dont le premier correspond à un identifiant d'interaction dans l'application, dont le second désigne un identifiant de modèle de service de l'application jouant le rôle de *provider* et dont le troisième correspond à un identifiant de modèle de service de l'application jouant le rôle de *requester*.

- "TheServicesRepositoryBean" pour les modèles d'implémentations de services enregistrés dans le dépôt de modèles d'implémentations du DMSA, ses attributs sont un String qui correspond à un identifiant de modèle d'implémentation de service (en l'occurrence, nous identifions chaque modèle d'implémentation de service via l'URL à laquelle l'implémentation peut être récupérée), un entier qui précise le poids en octets de l'implémentation (ce champ est complété automatiquement par le DMSA lors de l'enregistrement du modèle d'implémentation), un autre String qui correspond à la version sur laquelle l'implémentation peut exécutée (c'est-à-dire, soit R3, soit R4.1), un String qui précise qu'elle est l'interface fournie par l'implémentation, un autre String qui correspond au type d'interaction relatif à l'interface fournie, ainsi qu'une HashMap<String, String> dont le premier String désigne une interface requise et dont le second String correspond au type d'interaction relatif à l'interface requise, ainsi qu'un dernier String qui précise si l'implémentation a un état interne et enfin un attribut Properties qui permet de spécifier des couples "clé-valeur" à volonté

- et, enfin, "UsedServicesImplementationsBean" pour les informations relatives aux implémentations de services utilisées pour les applications déployées environnement par environnement ; ses attributs sont un String pour un identifiant d'environnement et une HashMap<String, HashMap<String, ArrayList<CoupleServiceIdInstanceId>>> où le premier String correspond à un identifiant d'implémentation de service, où le second String est relatif à un identifiant de modèle d'application et où le CoupleServiceIdInstanceId correspond à un couple associant un identifiant de modèle de service appartenant à l'application et un identifiant d'instance d'implémentation de service le réalisant.

Il est à noter que le *bean* FacadeBean est relié (c'est-à-dire utilise) à chacun des huit *entity beans* ; en effet, il est le point d'entrée du DMSA pour le conteneur JEE (c'est-à-dire qu'il est le seul *bean* à posséder une interface *@Remote*), il est donc utilisé aussi bien par la servlet de spécification des activités/tâches de déploiement, que par les servlets de gestion des dépôts de modèles d'application à services, de modèles d'implémentations de services, que par celle de gestion des environnements, mais aussi pour l'affichage des données/informations contenues dans les différents *entity beans*. Il est aussi important de noter que les classes "DeploymentPlanGeneration" et "RemoteDeploymentExecution" appartiennent respectivement aux *packages* "dmsa.beans.deploymentplangeneration" et "dmsa.communications" et que chacun des neuf *stateless beans* est amené à utiliser la classe "RemoteDeploymentExecution" du *package* "dmsa.communications" afin d'interagir avec le ou les agents du DMSA ciblés (ces interactions concernent majoritairement l'envoi des plans de déploiement à exécuter).

Nous avons précisé précédemment que l'administrateur interagit avec la partie serveur du DMSA. Pour ce faire, nous avons implémenté une IHM succincte. Le but de cette IHM est de permettre à l'administrateur d'utiliser les fonctionnalités offertes par le DMSA (nous n'avons pas cherché à satisfaire un quelconque requis de plasticité, convivialité ou autre). En conséquence, elle est composée d'une page HTML d'accueil (accessible à http://adresse_ip_du_serveur:9000/DMSA), ainsi que de quatre *servlets* : ServicesRepository, ServicesApplicationsRepository, ExecutionEnvironmentsManager et ServicesApplicationsDeployment. Chaque *servlet* correspond respectivement à la gestion du dépôt de modèles d'implémentations de services, à la gestion du dépôt de modèles d'application à services, à la gestion des environnements et enfin, à la spécification des tâches de déploiement et à l'affichage des données internes au DMSA. Cette IHM succincte est spécifiée via le fichier web.xml (voir figure ci-dessous) et elle est déployée dans le serveur JONAS via un fichier DMSA.war.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
	<servlet>
		<servlet-name>ServicesRepository</servlet-name>
		<servlet-class>dmsa.servlets.ServicesRepository</servlet-class>
	</servlet>
	<servlet>
		<servlet-name>ServicesApplicationsRepository</servlet-name>
		<servlet-class>dmsa.servlets.ServicesApplicationsRepository</servlet-class>
	</servlet>
	<servlet>
		<servlet-name>ServicesApplicationsDeployment</servlet-name>
		<servlet-class>dmsa.servlets.ServicesApplicationsDeployment</servlet-class>
	</servlet>
	<servlet>
		<servlet-name>ExecutionEnvironmentsManager</servlet-name>
		<servlet-class>dmsa.servlets.ExecutionEnvironmentsManager</servlet-class>
	</servlet>

	<servlet-mapping>
		<servlet-name>ExecutionEnvironmentsManager</servlet-name>
		<url-pattern>/ExecutionEnvironmentsManager</url-pattern>
	</servlet-mapping>
	<servlet-mapping>
		<servlet-name>ServicesApplicationsDeployment</servlet-name>
		<url-pattern>/ServicesApplicationsDeployment</url-pattern>
	</servlet-mapping>
	<servlet-mapping>
		<servlet-name>ServicesApplicationsRepository</servlet-name>
		<url-pattern>/ServicesApplicationsRepository</url-pattern>
	</servlet-mapping>
	<servlet-mapping>
		<servlet-name>ServicesRepository</servlet-name>
		<url-pattern>/ServicesRepository</url-pattern>
	</servlet-mapping>
</web-app>
```

Figure 60. Contenu du fichier "web.xml".

En ce qui concerne le déploiement de listes ordonnées de tâches de déploiement, nous proposons deux mécanismes : une exécution séquentielle (triviale) et une exécution parallèle (qui minimise le nombre de threads lancés tout en maximisant le parallélisme autant que notre contexte le permet).

La méthode correspondant à l'exécution séquentielle est : "public ArrayList <CoupleDeploymentTaskIdAndDeploymentTaskResult> DeploymentTasksSequentialExecutionFromFile Location (final String locationOfTheXMLFile_) throws DeploymentTasksExecutionException". Elle est lancée via la servlet "ServicesApplicationsDeployment".

Son fonctionnement est simple. Elle va commencer par parser le fichier XML qui lui a été passé en paramètre, elle va ensuite créer un thread, via la classe "DeploymentTasksSequentialExecutionCore_viaThread", elle va l'initialiser via le constructeur de la classe en lui passant la liste ordonnée de tâches (c'est-à-dire une "ArrayList <DeploymentTaskInterface>"), après cela, elle va le lancer, il va exécuter séquentiellement la liste de tâche et retourner le résultat de cette exécution via une "ArrayList <CoupleDeploymentTaskIdAndDeploymentTaskResult>" qui sera affichée via la servlet qui a servie à spécifier la location du fichier XML.

La méthode correspondant à l'exécution parallèle est "public LinkedHashMap<String, Object> DeploymentTasksParallelExecutionFromFileLocation (final String locationOfTheXMLFile_) throws DeploymentTasksExecutionException". Elle est lancée via la servlet "ServicesApplicationsDeployment".

Son fonctionnement consiste à parser le fichier XML qui lui a été passé en paramètre, puis à trier les tâches de la liste ordonnée correspondante (via un fichier XML). Ce tri consiste à identifier les différents environnements cibles et à dispatcher les tâches environnement par environnement tout en conservant leur ordre, lui aussi, environnement par environnement. Ensuite, elle va créer un thread (via la classe "DeploymentTasksSequentialExecutionCore_viaThread") pour chaque environnement ciblé, puis elle va lancer les différents threads et attendre la fin de leur exécution (du fait de notre contexte

l'exécution des tâches attribuées à chaque thread est séquentielle). Une fois l'exécution de tous les threads terminée, une "LinkedHashMap<String, Object>" est retournée et enfin, la servlet l'affiche.

Pour être tout à fait précis, le champ "String" de la "LinkedHashMap" correspond à un identifiant d'environnement d'exécution et le champ "Object" peut être soit une "ArrayList <CoupleDeploymentTaskIdAndDeploymentTaskResult>" correspondant aux résultats des exécutions relatives à l'environnement identifié, soit une exception "DeploymentTasksExecutionException" si l'exécution d'une tâche de déploiement relative à l'environnement identifié a échoué (cette exception contient aussi le résultat des tâches s'étant correctement déroulées).

1.2 Algorithme d'ordonnancement des activations

Dans cette sous-section, nous présentons l'algorithme d'ordonnancement que nous avons mis au point afin de calculer l'ordre d'activation des modèles de services d'un modèle d'application. Le but de cet ordonnancement est de détecter la présence de cycles de dépendances entre services (ces derniers rendent impossible l'activation des applications) et de définir un ordre d'activation entre les services permettant à l'activation unitaire de chaque service de pouvoir aboutir.

Notre algorithme d'ordonnancement est un algorithme récursif.

Dans un premier temps, notre algorithme commence par extraire un graphe orienté du modèle de l'application à ordonnancer. Dans ce graphe, chaque sommet correspond à un modèle de service appartenant à l'application et chaque arc orienté défini une préséance entre deux sommets. La définition d'une préséance nécessite qu'un des deux services ait une dépendance vers l'autre service et est fonction du type d'interaction de l'interface de chaque service mise en jeu dans la dépendance. Pour rappel, pour une interaction entre un client et un serveur, c'est le serveur qui possède la préséance (c'est-à-dire qu'il doit être activé avant le client et, ensuite, désactivé après le client), pour une interaction entre un publieur et un souscripteur, c'est le publieur qui possède la préséance ; enfin, pour une interaction entre un producteur et un consommateur, c'est le producteur qui possède la préséance.

Dans un second temps, l'algorithme identifie les sommets qui ne sont pas préséants, c'est-à-dire qui soit n'ont pas de dépendances, soit ont toutes leurs dépendances satisfaites par des requis de services de l'application elle-même. Les différents sommets identifiés n'ont pas de contrainte d'ordre entre eux. Ils forment la base de l'ordonnancement. Lors de l'activation de l'application, ils seront les premiers services à être activés.

Dans un troisième temps, l'algorithme cherche un sommet qui ne dépend que des sommets présents dans la base de l'ordonnancement. Si il est possible de trouver un tel sommet, alors le modèle de service correspondant est noté comme devant être activé après les services présents dans la base ; l'algorithme fait alors sa récursion et cherche alors un sommet qui ne dépend que des sommets présents dans la base de l'ordonnancement et des sommets ajoutés précédemment (c'est-à-dire un sommet pour la première récursion et n sommets pour la $n^{ième}$ récursion). Sinon, soit tous les sommets ont déjà été ordonnancés, ce qui signifie que l'ordonnancement de l'application est terminé (il est alors retourné sous la forme d'une liste ordonnée contenant les identifiants des modèles de services appartenant au modèle de l'application), soit au moins un sommet n'a pas été ordonnancé, ce qui signifie que le modèle d'application donné ne peut pas être ordonnancé et donc que le modèle de l'application ne peut pas être enregistré dans le dépôt du DMSA.

Pour remarque, l'ordonnancement d'un modèle d'application est réalisé une fois pour toute lors de son enregistrement dans le dépôt de modèles d'application à services du DMSA (c'est une condition nécessaire à l'enregistrement d'un modèle d'application). Les classes relatives à l'ordonnancement de l'activation sont situées dans le *package* "dmsa.beans.scheduler".

1.3 Activités de déploiement

Les huit activités de déploiement que nous proposons au travers du DMSA suivent globalement toutes le même diagramme de séquence. Ainsi, elles commencent toutes par la spécification d'une tâche

de déploiement par l'administrateur. Le DMSA valide ensuite cette tâche en fonction des modèles et des informations correspondants avant de générer le plan de déploiement de la tâche. Puis, le plan de déploiement généré est exécuté sur l'environnement cible via la partie "agent" du DMSA. Enfin, les modèles et les informations sont mis à jour et persistés ; et le résultat de l'exécution de la tâche est affiché. La figure ci-dessous présente ce diagramme.

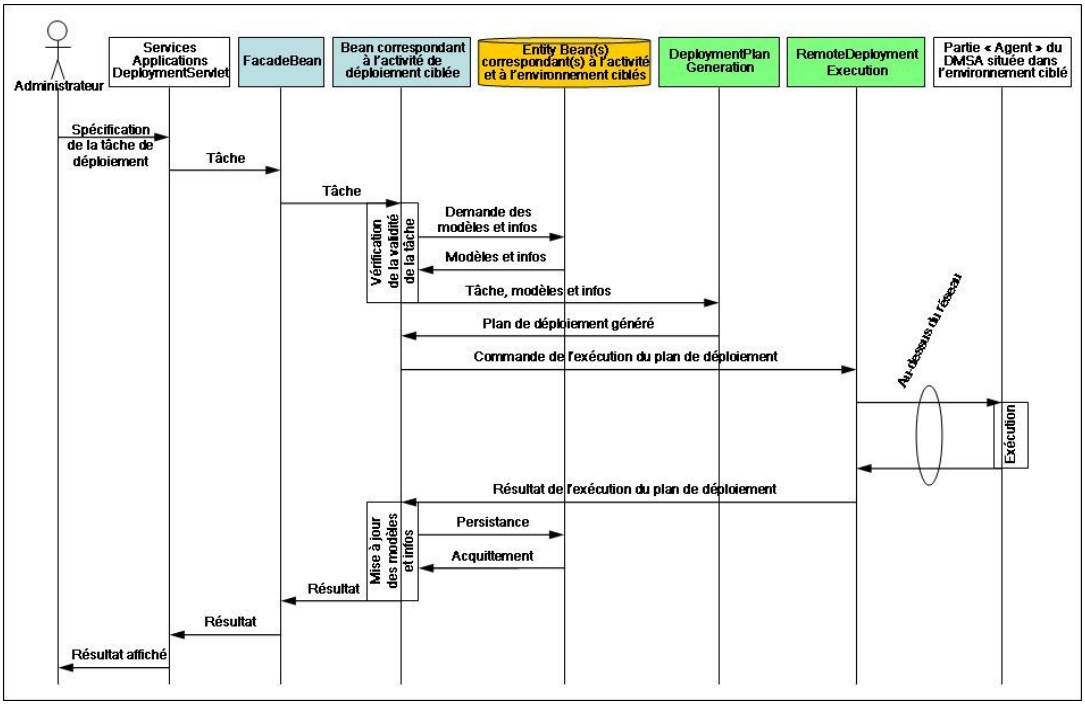


Figure 61. Diagramme de séquence pour les activités de déploiement proposées via le DMSA.

Pour rappel, le DMSA génère des plans de déploiement qui contiennent des instructions pour les niveaux implémentation de service et instance d'implémentation de service et cela via la classe "DeploymentPlanGeneration".

1.4 Déploiement de listes ordonnées de tâches de déploiement

Nous avons précisé précédemment que nous proposons deux mécanismes pour l'exécution de listes ordonnées de tâches de déploiement. Le premier mécanisme propose l'exécution séquentielle d'une liste de tâches alors que le second propose une exécution parallèle.

Le diagramme de séquence du mécanisme d'exécution séquentielle d'une liste ordonnée de tâches de déploiement se base sur le diagramme de séquence des activités de déploiement présenté dans la sous-section précédente.

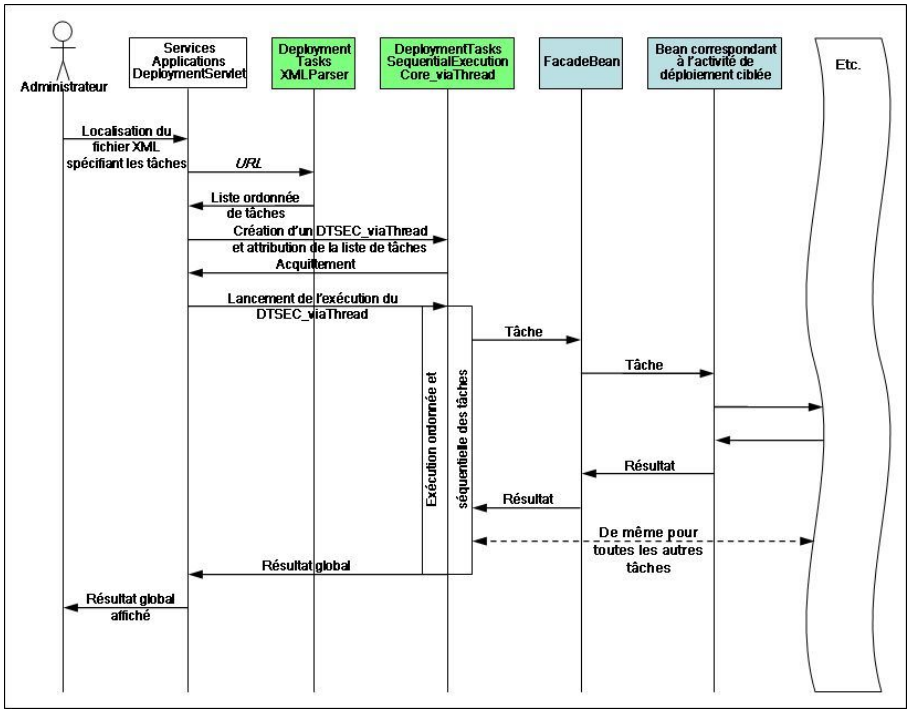


Figure 62. Diagramme de séquence pour le mécanisme d'exécution séquentielle.

Pour remarque, dans cette figure, nous avons choisi de tronquer le diagramme de séquence (via la courbe en épaisseur notée "Etc.") puisque la séquence tronquée peut être retrouvée dans le "diagramme de séquence pour les activités de déploiement" présenté dans la figure de la section précédente.

Le diagramme de séquence du mécanisme d'exécution parallèle d'une liste ordonnée de tâches de déploiement se base en partie sur le diagramme de séquence du mécanisme d'exécution séquentielle présenté dans la figure précédente.

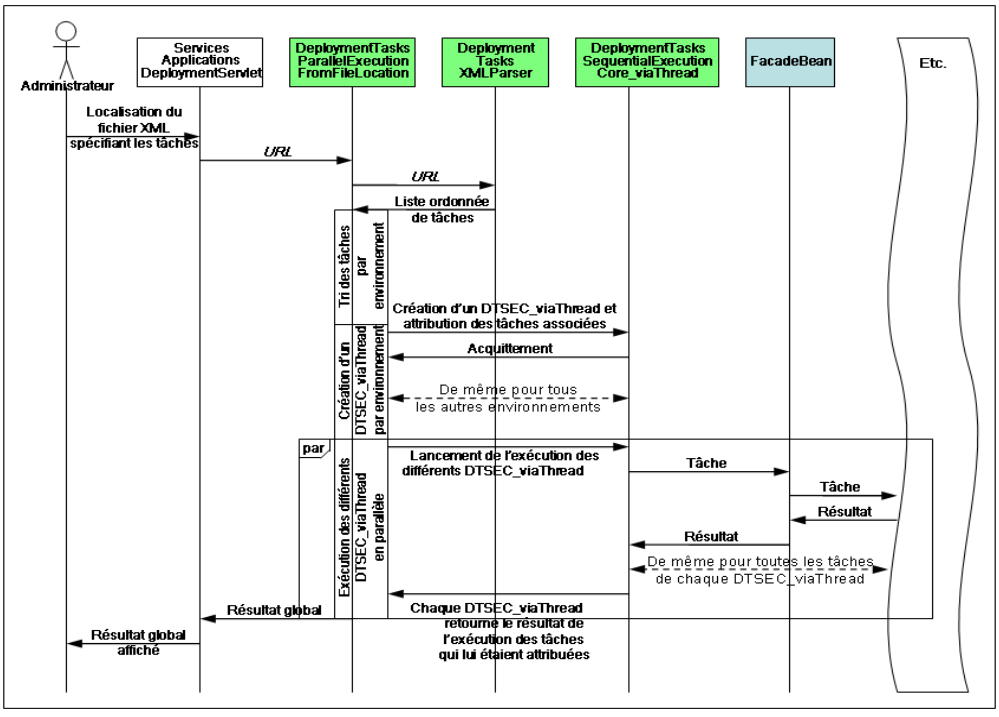


Figure 63. Diagramme de séquence pour le mécanisme d'exécution parallèle.

Pour remarque, dans la figure ci-dessus, nous avons choisi de tronquer le diagramme de séquence (via la courbe en épaisseur notée "Etc.") puisque la séquence tronquée peut être retrouvée dans le "diagramme de séquence pour le mécanisme d'exécution séquentielle " présenté dans la première figure de cette sous-section. En complément, il est important de noter que, dans le cadre du mécanisme d'exécution parallèle, les tâches ciblant un même environnement sont exécutées séquentiellement (c'est-à-dire tâche après tâche) et selon l'ordre défini par l'administrateur dans le fichier XML ; les tâches ciblant des environnements différents sont, quand à elles, exécutées en parallèle via des "DeploymentTasksSequentialExecutionCore_viaThread différents et donc concurrents.

2. PARTIE "AGENT" DU DMSA

Du fait de notre contexte, nous appréhendons les environnements d'exécution à services qui sont basés soit sur la version R4.1 soit sur la version R3 de la technologie OSGi. En conséquence, nous avons mis en place deux parties "agent", une pour la version R4.1 et une autre pour la version R3.

Dans les sous-sections suivantes nous nous focalisons sur la partie "agent" correspondant à la version R4.1 de la technologie OSGi.

2.1 Architecture JMX

La technologie JMX est proposée par l'entreprise Sun Microsystems [SUN08]. L'acronyme JMX signifie *Java Management eXtensions*. Elle définit une architecture de gestion de ressources qui est composée de trois niveaux. Le niveau "Agent" est le niveau principal, c'est le cœur de l'architecture JMX, il contient le serveur de "MBeans". Le second niveau est le niveau "Instrumentation", il contient les MBeans qui représentent des ressources. Concrètement, les MBeans sont destinés à être enregistrés auprès du serveur de MBeans afin d'être mis à disposition. Le troisième et dernier niveau est le niveau "Distributed Services". Typiquement, il contient les *adaptors*, c'est-à-dire des connecteurs et adaptateurs de protocoles. Ceux-ci doivent eux-aussi s'enregistrer auprès du serveur de "MBeans" et, une fois enregistrés, ils permettent d'accéder au serveur et donc aux "MBeans" associés suivant le protocole du connecteur. Les implémentations de l'architecture JMX contiennent souvent, par défaut, le connecteur RMI (*Remote Method Invocation*).

Du fait de notre position au -dessus de la technologie OSGi R4.1, nous avons utilisé la technologie MOSGi (*Managed OSGi*) qui propose le portage de l'implémentation JMX de Sun MicroSystems au-dessus de la technologie OSGi R4.1. Cette "bundle-isation" est constituée par trois *bundles* principaux. Le premier est le "MOSGi JMX-MX4J Agent Service", c'est le serveur de "MBeans". Le second et le troisième sont le "MOSGi JMX RMI-Registry" et le "MOSGi JMX-MX4J RMI Connector", qui fournissent l'infrastructure nécessaire à la communication au-dessus de RMI.

Pour tirer profit de cette architecture JMX bundlisée, nous avons développé notre propre "MBean" bundlisé (ou "Bundle MBean"). Le but de notre "Bundle MBean" est la mise à disposition des mécanismes de tests de l'environnement et d'exécution des plans de déploiement (ce dernier regroupant l'exécution de plans de déploiement, le mécanisme "transactionnel" et le mécanisme fournissant l'approche *naming*).

Au niveau de la communication entre les parties serveur et agent du DMSA, nous avons utilisé le connecteur RMI.

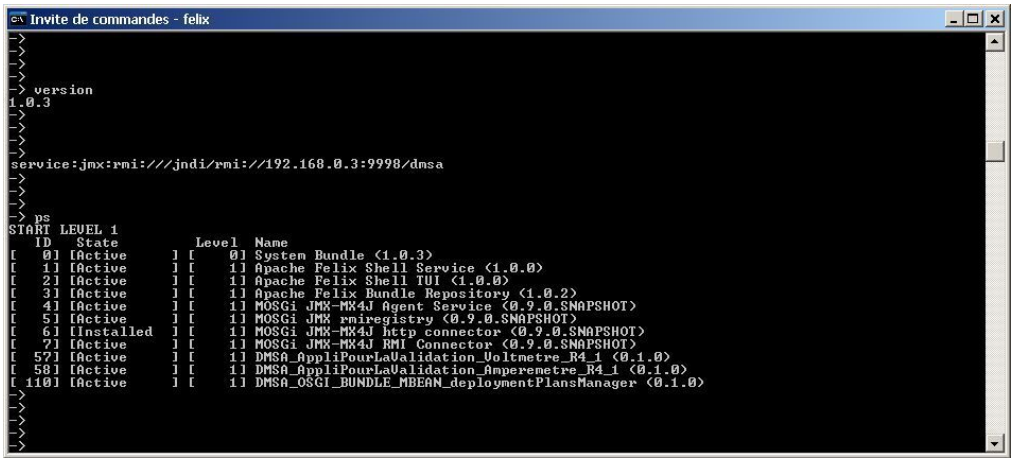


Figure 64. Felix 1.0.3 (R4.1), les *bundles* MOSGi et le *bundle* "MBean" du DMSA.

Pour remarque, l'architecture JMX de la technologie MOSGi nécessite un ordonnancement de son activation, à savoir, l'agent JMX doit être activé avant le registre RMI qui lui même doit être activé avant le connecteur RMI.

La figure ci-dessous schématise l'architecture JMX de la partie agent pour OSGi R4.1.

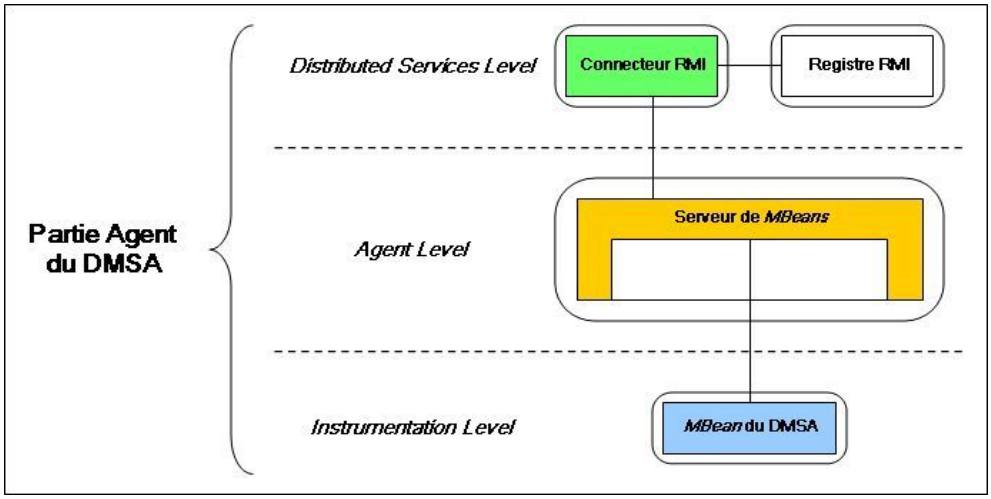


Figure 65. Schéma de la partie agent du DMSA au-dessus de la technologie OSGi R4.1.

Dans le schéma ci-dessus, chaque rectangle aux coins arrondis représente un *bundle* OSGi, ce qui permet d'apprécier le découpage des entités JMX au-dessus de la technologie OSGi dans MOSGi.

Pour remarque, nous avons configuré notre agent (du DMSA) pour la technologie OSGi R3 afin qu'il soit accessible sur le port 9999 et à l'emplacement "dmsa" (son URL est "service:jmx:rmi:///jndi/rmi:// adresse IP de l'environnement :9999/dmsa"). Pour la technologie OSGi R4.1, l'URL est " service:jmx:rmi:///jndi/rmi:// adresse IP de l'environnement :9998/dmsa".

Enfin, de plus amples informations et exemples autour de la technologie JMX peuvent être trouvés dans [Obe04] [Don06a].

2.2 Limitation actuelle de la partie agent sur OSGi R4.1

La partie serveur du DMSA génère des plans pour les niveaux implémentation et instances, or les fabriques d'instances actuellement proposées ne sont pas suffisantes pour permettre à la fois de gérer plusieurs instances (d'implémentations de services) et d'utiliser une même implémentation de service pour

réaliser plusieurs modèles de services appartenant ou non à la même application (formulé autrement, les fabriques proposées sont incompatibles avec l'approche *naming*).

En effet, la meilleure fabrique d'instances actuellement proposée au-dessus de la technologie OSGi fixe les requis de services de son implémentation de service lors de son activation [EHL07], cela signifie que toutes les instances créées utiliseront les fournisseurs de services choisis/attribués pour satisfaire les requis de services de l'implémentation de service (de la fabrique). Or dans le cadre des applications à services et donc dans le cadre de l'utilisation d'une même implémentation de service pour réaliser plusieurs modèles de services (appartenant ou non à la même application), il est nécessaire de pouvoir choisir/attribuer les fournisseurs de services satisfaisants les requis de services d'une implémentation en fonction du modèle de service qu'elle réalise. Ainsi, dans le cadre d'une application "A1" contenant les services "S1" et "S2", une implémentation de service "I1" réalisant "S1" devra être satisfaite par une implémentation de service "I2" service "S2", dans le même temps, dans le cadre d'une application "A2" contenant les services "S3" et "S4", l'implémentation de service "I1" (ce n'est pas une coquille) réalisant "S3" devra être satisfaite par l'implémentation "I4" réalisant "S4" ; or il n'est pas possible actuellement de lier à "I1" à "I2" dans le cadre de "A1" et "I1" à "I4" dans le cadre de "A2".

Le fait que la mise en place d'une approche *naming* ne soit pas possible implique que la technologie ne fournit pas les mécanismes suffisants pour assurer la cohérence entre le niveau modèle et le niveau "réalité" et cela dans le cas de la création de plusieurs instances pour une même implémentation de service.

Au final, bien que la partie serveur du DMSA génère des plans de déploiement contenant des instructions pour les niveaux implémentation et instance, la partie agent du DMSA qui est embarquée dans chaque environnement est complètement intégrée avec le niveau implémentation de service de l'environnement OSGi, mais ne (peut) manipule(r) qu'une seule instance par implémentation de service pour le niveau instance d'implémentation de service (en l'occurrence un *singleton* par implémentation).

Il est important de noter qu'il n'existe pas de fabrique d'instances d'implémentation de service pour la technologie OSGi R3. Cette dernière ne permet donc que de manipuler des *singletons*.

En conséquence, le degré d'intégration du niveau instance d'implémentation de service est le même pour les technologies cibles OSGi R3 et R4.1.

Nous verrons dans le chapitre validation que le fait que notre intégration avec la technologie OSGi ne soit pas entière ne nous permet pas de valider nos mécanismes de partage d'instances, de gestion de l'isolation, cependant il nous permet de valider l'ensemble des autres mécanismes, à savoir, notre approche dirigée par les modèles pour le déploiement, l'ordonnancement, la sensibilité au contexte, la gestion du partage d'implémentation, l'approche *naming*, le déploiement "transactionnel", les mises à jour statiques, les exécutions séquentielles et parallélisées de listes ordonnées de tâches de déploiement.

2.3 Activités de déploiement

Dans la figure ci-dessous, nous présentons un diagramme de séquence synthétisant l'exécution d'un plan de déploiement par la partie "agent" du DMSA. Cette figure agrège les séquences relatives aux instructions de déploiement pour les niveaux implémentation de service et instance intégrés.

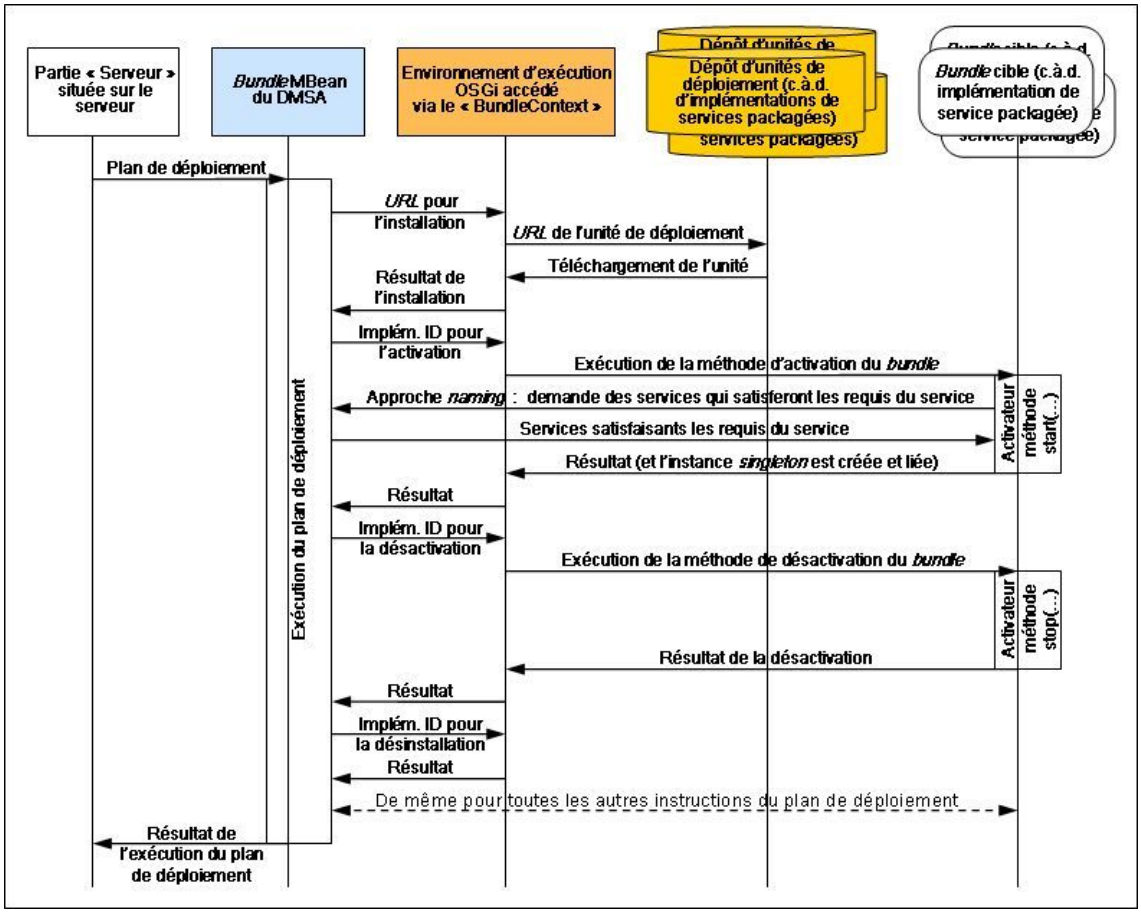


Figure 66. Diagramme de séquence de l'exécution d'un plan de déploiement.

Dans la figure ci-dessus, l'élément "Environnement d'exécution OSGi accédé via le *BundleContext*" désigne notre point d'entrée/d'interaction avec l'environnement OSGi ; ce point d'entrée, c'est le *BundleContext* fourni par l'environnement OSGi. Le *BundleContext* permet d'accéder aux implémentations de services déployées dans l'environnement, ainsi qu'au registre de services de l'environnement (c'est-à-dire au registre d'interfaces Java décorées servant de contrat pour les fournisseurs OSGi). A ce propos, seuls les fournisseurs dont les bundles sont actifs peuvent enregistrer leur contrat dans le registre de l'environnement.

Pour remarque, les activités de mises à jour statiques au niveau modèle d'application supportées par le DMSA sont ensuite exprimées dans les plans de déploiement via les activités d'installation et de désinstallation au niveau implémentation de service.

Pour rappel, lors de l'activation d'une application, chaque service fait appel à l'approche *naming* que nous avons implémentée. Trois cas se présentent alors, ainsi lorsqu'un service en cours d'activation fait appel à notre mécanisme *naming*, soit les requis du service sont satisfaits par d'autres services appartenant à cette application et dans ce cas, le mécanisme *naming* retourne les (fournisseurs de) services à contacter et utiliser, soit les requis de service sont satisfaits par des requis exposés par l'application, auquel cas notre mécanisme retourne la liste des services qui n'apparaissent pas dans l'application et qui offrent un fournisseur satisfaisant, enfin soit notre mécanisme *naming* doit fait face à un mélange des deux cas précédents et retourne, en conséquence, le mélange des deux solutions précédentes.

2.4 Déploiement "transactionnel"

Dans cette sous-section, nous présentons le mécanisme que nous avons développé pour supporter les transactions pour le niveau implémentation dans le cadre du déploiement de modèles d'application.

Le diagramme de séquence ci-dessous présente le *rollback* sur l'activité de désactivation. La transaction sur l'activité de désactivation est celle qui parmi les quatre activités (installation, activation, désactivation et désinstallation) au niveau implémentation de service est la plus complexe ; en effet si la désactivation d'une implémentation de service échoue lors de la désactivation d'une application, il faut alors remettre l'application dans l'état activé ; en conséquence, le *rollback* correspondant doit certes réactiver les implémentations de services qui ont été désactivées, mais il doit qui plus est le faire dans le cadre de l'approche *naming*.

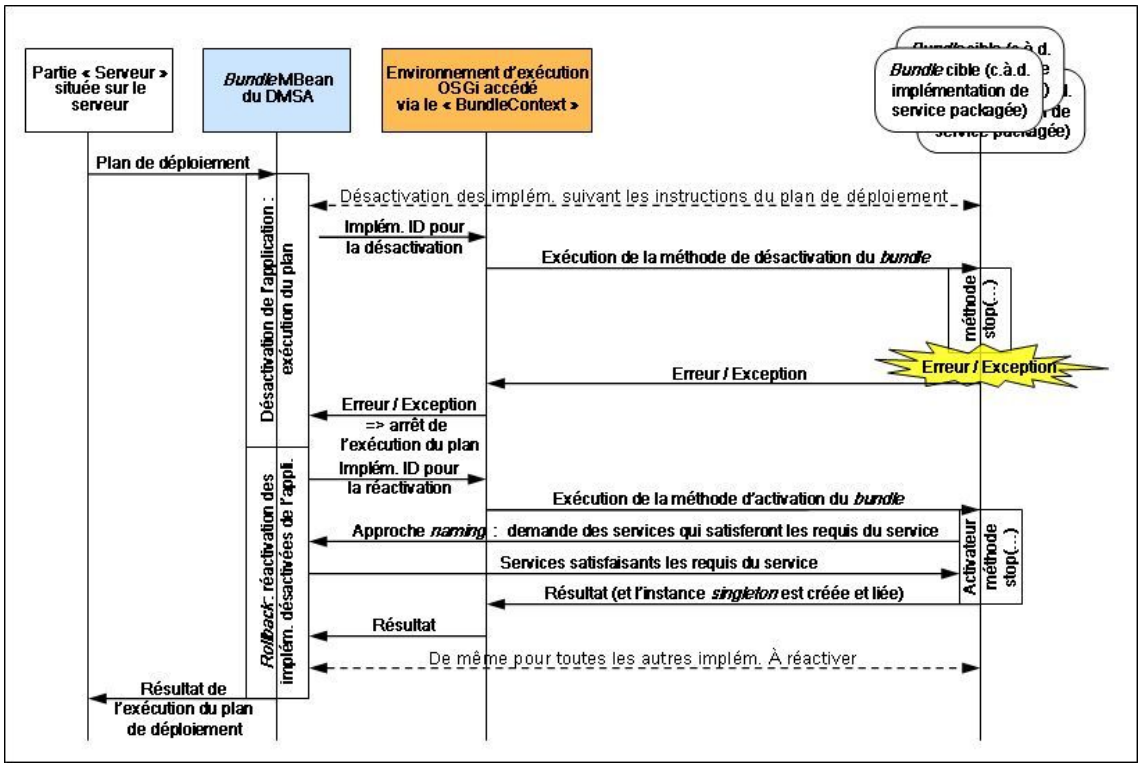


Figure 67. Diagramme de séquence de désactivation d'une application avec l'apparition d'une erreur/exception et le *rollback* correspondant.

Dans la figure ci-dessus, l'élément "Environnement d'exécution OSGi accédé via le *BundleContext*" désigne notre point d'entrée/d'interaction avec l'environnement OSGi ; ce point d'entrée, c'est le *BundleContext* fourni par l'environnement OSGi. Le *BundleContext* permet d'accéder aux implémentations de services déployées dans l'environnement, ainsi qu'au registre de services de l'environnement (c'est-à-dire au registre d'interfaces Java décorées servant de contrat pour les fournisseurs OSGi). A ce propos, seuls les fournisseurs dont les bundles sont actifs peuvent enregistrer leur contrat dans le registre de l'environnement.

Comme présenté dans le diagramme de séquence ci-dessus, nous avons choisi de placer notre mécanisme de *rollback* dans la partie agent du DMSA ; nous aurions pu choisir de remonter l'erreur/exception vers la partie serveur du DMSA pour lui faire calculer un plan de déploiement pour le *rollback*.

2.5 Partie "agent" du DMSA pour OSGi R3

La figure ci-dessous est une capture d'écran de l'environnement d'exécution R3 dans lequel le serveur de "MBean" et le connecteur RMI sont déployés (via le *bundle* JMX Agent 0.3.2). La partie agent du DMSA est déployée via le *bundle* "MBean Bundle DeploymentPlansManager".

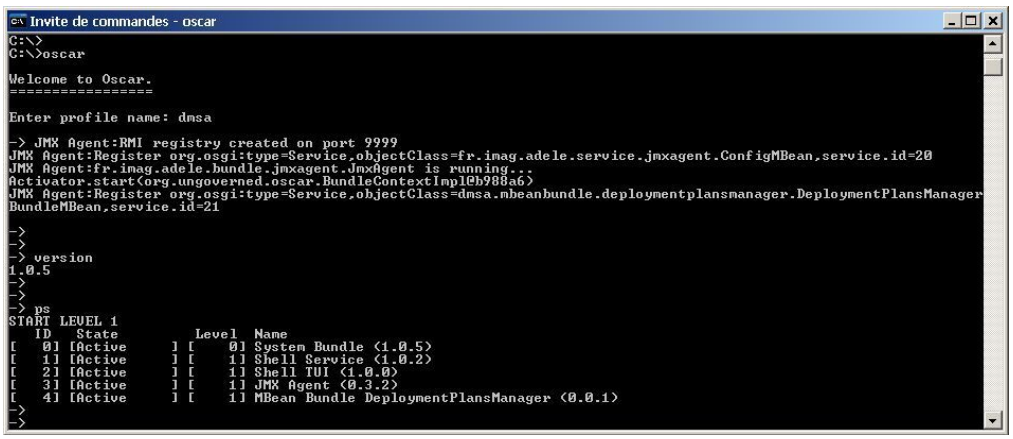


Figure 68. Oscar 1.0.5 (R3), les *bundles* JMX Agent et "MBean" de la partie "agent".

La figure ci-dessous représente la partie agent du DMSA au-dessus de la technologie OSGi R3. Comme il est possible de le voir dans cette figure, le serveur de MBeans et la connexion RMI que nous utilisons au-dessus d'OSGi R3 sont empaquetés dans le même bundle [Don05].

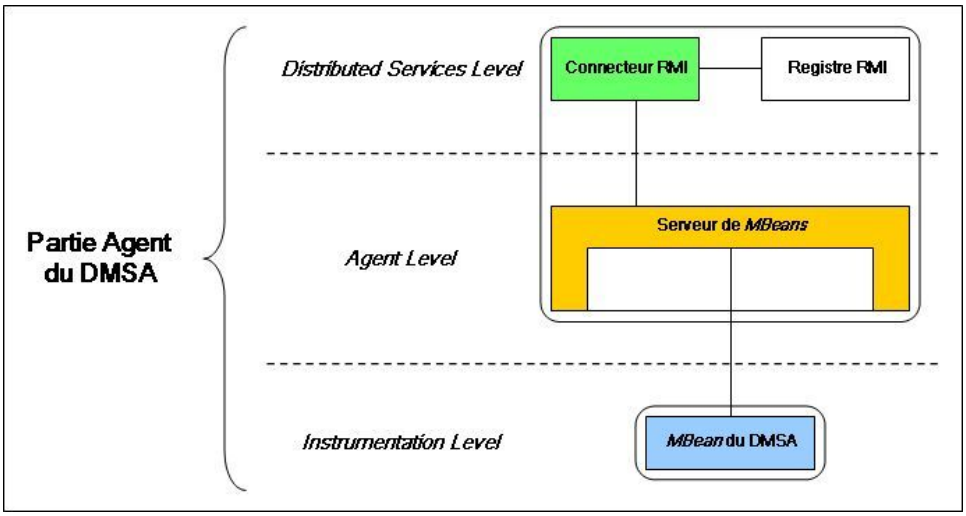


Figure 69. Schéma de la partie agent du DMSA au-dessus de la technologie OSGi R3.

3. CONCLUSION

Dans ce chapitre implémentation, nous avons présenté les différents algorithmes et architectures que nous avons conçus et développés pour réaliser notre *Deployment Manager for Services Applications* (DMSA). Pour ce faire nous avons détaillé successivement les parties "serveur" et "agent" du DMSA à l'aide de diagrammes de beans et de séquence.

Synthétiquement la partie serveur du DMSA propose une IHM succincte à l'administrateur et est à l'origine de la génération des plans de déploiement qui seront exécutés dans les environnements. La partie "agent" embarquée dans chaque environnement ciblé est, elle, en charge de l'exécution des plans de déploiement reçus et du retour du résultat de leur exécution. Du fait du support des versions R3 et R4.1 de la technologie OSGi, nous proposons une partie "agent" adaptée à chaque version.

Le chapitre suivant se focalise sur la validation de notre proposition.

VI. VALIDATION

Dans ce chapitre, nous allons présenter plus précisément notre contexte et ses besoins/problématiques. Nous détaillerons le déroulement du déploiement via notre DMSA. Nous préciserons les objectifs de notre contexte initial que nous avons remplis et ceux que nous n'avons pas abordés, ainsi que les objectifs supplémentaires qui sont apparus et que nous avons satisfaits. Enfin, nous conclurons ce chapitre.

En premier lieu, du fait de l'absence de fabriques d'instances d'implémentations de services satisfaisant nos besoins et de notre intégration incomplète avec la technologie OSGi, plus précisément avec le niveau instances d'implémentations de services, nous ne présenterons donc ici aucun résultat relatif au partage d'instances et à la gestion de l'isolation et cela bien que la partie serveur du DMSA génère des plans de déploiement prenant en compte le partage d'instances et la gestion de l'isolation.

En second lieu, bien que notre intégration entre la partie agent du DMSA et la technologie soit incomplète, l'intégration des instances singleton nous permet de valider notre approche et le DMSA, ainsi que de présenter des résultats pour toutes les autres fonctionnalités sur lesquelles porte cette thèse, c'est-à-dire sur notre approche dirigée par les modèles pour le déploiement, sur l'ordonnancement, sur la sensibilité au contexte, sur la gestion du partage d'implémentation, sur l'approche *naming*, sur le déploiement "transactionnel", sur les mises à jour statiques et sur les exécutions séquentielles et parallélisées de listes ordonnées de tâches de déploiement.

1. CONTEXTE ISSU DU PROJET PISE

L'exploitation des données collectées par les capteurs enfouis dans les entreprises offre de nouvelles opportunités économiques aux équipementiers. Cependant, la mise en place des services de haut niveau associés est actuellement faite de manière empirique. Le projet PISE a donc pour but la proposition de méthodes et d'outils pour la mise en place de tels services de haut niveau sur un parc de passerelles à grande échelle et cela de manière flexible et évolutive [BBB+06]. L'acronyme PISE signifie "Passerelles Internet Sécurisées et flExibles".

Le projet PISE (http://www.pise.imag.fr) est un projet RNRT (Réseau National de Recherche en Télécommunications) d'une durée de deux ans (sept. 2004 - déc. 2006). Le projet est fortement orienté par les besoins métier du domaine de la distribution électrique. Dans ce domaine, les passerelles Internet jouent un rôle prépondérant pour permettre à des équipementiers ou à des opérateurs d'offrir à leurs clients des services de haut niveau à forte valeur ajoutée accessibles au-dessus d'Internet. Ces passerelles sont reliées aux équipements via des bus de terrain et permettent la mise en œuvre des services au plus près des équipements ; cela est fondamental afin d'améliorer la sécurité, la fiabilité et la performance des services et de réduire le volume de communication.

Le projet est un partenariat entre plusieurs organismes de recherche publics et industriels, tels que l'équipe Adèle du Laboratoire Informatique de Grenoble et l'équipe OASIS de l'INRIA Sophia Antipolis, et des industriels tels Schneider Electric, Trialog et France Telecom R&D Grenoble.

Le projet répond à trois objectifs prioritaires du RNRT :

- Maîtriser l'hétérogénéité - Les équipements industriels de distribution électrique qui sont utilisés par le projet sont très hétérogènes. En conséquence, la notion de passerelle Internet développée doit permettre de faire face à cette diversité en concentrant les données de terrain issues des équipements et en fournissant une API d'accès standardisée. Le traitement transparent des aspects liés à la sécurité et à la distribution permet en outre d'établir une connectivité "sans couture" entre les passerelles et le serveur Internet ;
- Garantir la sécurité - L'un des buts essentiels du projet est de sécuriser les services fournis par les passerelles Internet. Il s'agit en particulier de permettre l'authentification et le contrôle des accès, de garantir la confidentialité et l'intégrité des communications, et de prévenir les dénis de fonctionnement. Ceci doit être fait de façon la plus transparente possible pour l'utilisateur final et pour les développeurs impliqués. Il est important de noter que la sécurité a été identifiée par les départements *marketing* des partenaires industriels du projet comme le facteur clé d'acceptation des services sur Internet en distribution électrique ;
- Développer de nouveaux services - La finalité du projet est de permettre le développement de nouveaux services accessibles depuis Internet dans le secteur de la distribution électrique. Des applications pilotes dans ce secteur ont démontré que des services à valeur ajoutée sont attendus par les clients mais que des infrastructures adaptées sont nécessaires pour leur mise en place à grande échelle. De telles infrastructures comprennent des passerelles sécurisées et flexibles et des outils d'administration distante de ces passerelles permettant de connaitre l'état de la passerelle et d'adapter facilement les services de haut niveau aux besoins de chaque client. Le projet a mis en œuvre des services définis conjointement avec les départements de *marketing* des partenaires industriels.

Le projet PISE vise à lever plusieurs verrous technologiques majeurs. Ceux relatifs au déploiement et à l'administration sont définis dans le sous projet numéro 4 dont le but est l'implantation d'un outil permettant de déployer et d'administrer des services sur un parc de passerelles. Les travaux de cette thèse ont été initiés afin de répondre au but (et donc aux besoins/problématiques) de ce sous-projet.

Plus précisément, nous avons cherché à adresser les besoins/problématiques identifiés par la tâche 4.1 définie dans le sous projet 4. Ces problématiques sont :

- La gestion des changements pour les logiciels déployées (et donc leur configuration) - L'évolution des systèmes informatiques est inévitable ; elle provoque des changements dans les

systèmes qui s'exécutent sur une machine, phénomène qui a un impact important sur le domaine du déploiement. Ces changements peuvent être de nature matériels ou logiciels. Ils peuvent impliquer des changements dans les logiciels déjà installés. Pour bien gérer ces changements, le déploiement doit utiliser comme élément de décision la configuration/l'état du site où un logiciel doit être installé ou d'un logiciel qui doit être déployé ; il doit aussi être capable de mettre à jour cet élément afin de disposer d'une information correcte.

- La coordination des activités du déploiement et la flexibilité associée - Le cycle de vie du déploiement est composé de plusieurs activités interconnectées. La complexité du déploiement est due aussi aux relations qui existent entre ces activités. Le traitement du déploiement consiste à couvrir toutes les activités, à gérer les relations qui peuvent exister entre elles et à coordonner leur exécution. L'utilisation de la technologie des procédés (*workflow*) est appropriée pour la gestion des procédés de déploiement qui peuvent s'exécuter en parallèle dans une entreprise. De plus, un système de déploiement doit offrir une manière flexible et efficace de définir les activités à exécuter.
- Les réseaux et l'intégration d'Internet - L'importance croissante des réseaux a offert de nouvelles possibilités au déploiement. Il est maintenant possible de déployer un logiciel à distance, c'est-à-dire à partir d'une machine du réseau vers une autre machine cible reliée à ce même réseau. Ce fait pose aussi des problèmes comme la gestion de la bande passante ou la gestion du transfert des données. Ainsi, le déploiement doit prendre en compte des variables comme la taille des unités de déploiement à transférer, le nombre de sites vers lesquels le transfert est exécuté et la qualité de service de la transmission.
- L'hétérogénéité des plateformes - Le développement des réseaux, des protocoles de communication et des intergiciels ont rendu possible l'apparition d'applications distribuées à grande échelle et la coopération de plateformes hétérogènes. L'interopérabilité de ces plateformes doit être prise en compte par le déploiement. La plateforme est devenue une variable du déploiement, quand il s'agit par exemple de résoudre des dépendances.
- Résolution des dépendances - Dans un système informatique dans lequel coexistent plusieurs applications, il peut exister des relations entre les entités composants chaque application et entre les applications elles-mêmes. Ces relations sont exprimées en termes de dépendances. Ces dépendances sont des dépendances de déploiement. En effet, elles sont soit nécessitées pendant le déroulement d'activités de déploiement, soit utilisées par l'application à l'exécution. Ces dépendances introduisent de nouveaux aspects dans le cas particulier du partage des composants. Ainsi, à l'installation d'une nouvelle version ou à la désinstallation d'une application, le déploiement doit prendre en compte les relations inter composants et leur utilisation par les autres applications.
- La sécurité - Avec l'utilisation de réseaux pour déployer à distance un logiciel, des demandes impératives sont apparues concernant la sécurité et plus particulièrement l'authentification. Un point particulièrement sensible est la sécurité de la transmission entre deux sites distants (notamment entre le site producteur et le site client). En effet, des données sensibles à la visibilité restreinte peuvent être transférées. Le déploiement aussi doit offrir des mécanismes pour sécuriser l'activité d'installation.
- Le traitement des exceptions - Le déploiement est un procédé sur la durée qui est sensible à l'apparition de situations exceptionnelles. L'apparition de ces exceptions peut mettre le système dans un état incohérent. Le déploiement doit donc intégrer des mécanismes de reprise sur panne pour les activités de déploiement, tout en minimisant les interventions que l'utilisateur aura à faire lors de l'apparition d'une exception. Pour ce faire, il faut ajouter un moyen de supervision du déroulement du déploiement qui détecte les exceptions apparues et puis essaie de réparer leurs conséquences. Le déploiement doit pouvoir décrire les exceptions susceptibles d'être émises ainsi que leurs traitements.

Dans le cadre de la tâche 4.1 de ce projet, l'équipe ADELE est chargée de concevoir et d'implanter l'outil qui permettra de déployer (c'est-à-dire installer, activer, etc.) des services sur plusieurs passerelles (et donc de connaitre leur état de déploiement).

C'est dans ce contexte que nous avons défini notre gestionnaire de déploiement (DMSA) afin de permettre et d'automatiser le déploiement d'applications à services dans des environnements d'exécutions à services.

2. DÉROULEMENT DU DÉPLOIEMENT

Dans cette section, nous allons présenter le déroulement du déploiement (dirigé par les modèles) du modèle de l'application à services que nous avons conçue et développée afin de valider nos travaux. Cette application est chargée de donner une image instantanée et datée de la puissance consommée par un équipement via la collecte de données électriques récupérées par le biais d'un voltmètre et d'un ampèremètre connectés à l'équipement. Cette application est nommée *Current Electrical Power Consumed* (CEPC). Pour ce faire, nous allons commencer par présenter les trois pré-conditions qui doivent être satisfaites avant tout déploiement, puis nous nous focaliserons sur le déploiement (dirigé par les modèles du modèle) de l'application CEPC, enfin nous présenterons les deux mécanismes relatifs à l'exécution d'une liste ordonnée de tâches de déploiement que nous proposons.

2.1 Satisfaction des pré-conditions au déploiement

Le DMSA nécessite la satisfaction de trois pré-conditions afin de permettre effectivement le déploiement d'un modèle d'application à service dans un environnement d'exécution à services. Ces trois pré-conditions sont :

- L'enregistrement du/des environnements d'exécution cibles (embarquant la partie agent du DMSA) auprès du gestionnaire d'environnements du DMSA. Les enregistrements d'environnements se font via la servlet ExecutionEnvironmentsManager et les informations à fournir sont l'identifiant de l'environnement d'exécution (c'est-à-dire son adresse IP) et sa version (qui est soit R4.1, soit R3). Cette servlet est accessible, soit via l'IHM succincte du DMSA, soit directement à l'adresse : http://adresse_ip_du_serveur:9000/DMSA/ServicesRepository. Il est aussi possible de tester l'environnement cible via cette même servlet en spécifiant l'identifiant de l'environnement à tester. Ce test porte sur la présence et l'accessibilité de la partie agent du DMSA (c'est-à-dire sur l'accessibilité de l'environnement et du serveur de MBeans et sur la présence et l'accessibilité du MBean du DMSA). Pour remarque, le DMSA effectue ce test de façon systématique avant l'exécution de toute activité de déploiement ;
- L'enregistrement du/des modèles d'implémentations de services dans le dépôt de modèles d'implémentations de services du DMSA. Ces enregistrements se font via la servlet ServicesRepository. Les informations à fournir sont un modèle d'implémentation de service conforme au méta-modèle que nous avons défini pour les implémentations de services. Ce modèle est passé en paramètre au DMSA via un fichier XML. La figure ci-dessous présente le fichier XML correspondant au modèle d'une implémentation de service qui entrera en jeu dans le cadre du déploiement de l'application de validation CEPC ; sachant qu'une fois enregistré, le modèle contiendra aussi l'attribut sizeInOctet qui aura été complété automatiquement par le DMSA. Pour remarque, un tag "providedInterface" de figure ci-dessous a été tronqué pour améliorer la mise en page.

```
<serviceOfTheRepository>
   <implemURL>http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.aggregationettransf
ormation.r4_1-0.1.0.jar</implemURL>
   <executionEnvironmentVersion>R4.1</executionEnvironmentVersion>
   <providedInterface>dmsa.applipourlavalidation.aggregationettransformation.r4_1.contract.AggregationEtTransformationService</providedI>
   <providedInterfaceInteractionType>Server</providedInterfaceInteractionType>
   <requiredInterfacesHashMap>
      <requiredInterfaceAndRequiredInterfaceInteractionType>
         <requiredInterface>dmsa.applipourlavalidation.voltacquisition.r4_1.contract.VoltageAcquisitionService</requiredInterface>
         <requiredInterfaceInteractionType>Client</requiredInterfaceInteractionType>
      </requiredInterfaceAndRequiredInterfaceInteractionType>
      <requiredInterfaceAndRequiredInterfaceInteractionType>
         <requiredInterface>dmsa.applipourlavalidation.ampereacquisition.r4_1.contract.AmpAcquisitionService</requiredInterface>
         <requiredInterfaceInteractionType>Client</requiredInterfaceInteractionType>
      </requiredInterfaceAndRequiredInterfaceInteractionType>
      <requiredInterfaceAndRequiredInterfaceInteractionType>
         <requiredInterface>dmsa.applipourlavalidation.date.r4_1.contract.DateService</requiredInterface>
         <requiredInterfaceInteractionType>Client</requiredInterfaceInteractionType>
      </requiredInterfaceAndRequiredInterfaceInteractionType>
   </requiredInterfacesHashMap>
   <stateType>withoutState</stateType>
   <properties></properties>
</serviceOfTheRepository>
```

Figure 70. Exemple d'un modèle d'implémentation de service.

- L'enregistrement du/des modèles d'application à services cibles dans le dépôt du DMSA. Dans le cadre de cette validation, l'application ciblée est : CEPC. En complément de son enregistrement dans le dépôt de modèles d'application à services du DMSA, le modèle de l'application est ordonnancé. La figure ci-dessous la représente dans notre notation. La seconde figure ci-dessous présente, elle, le fichier XML correspondant à son modèle.

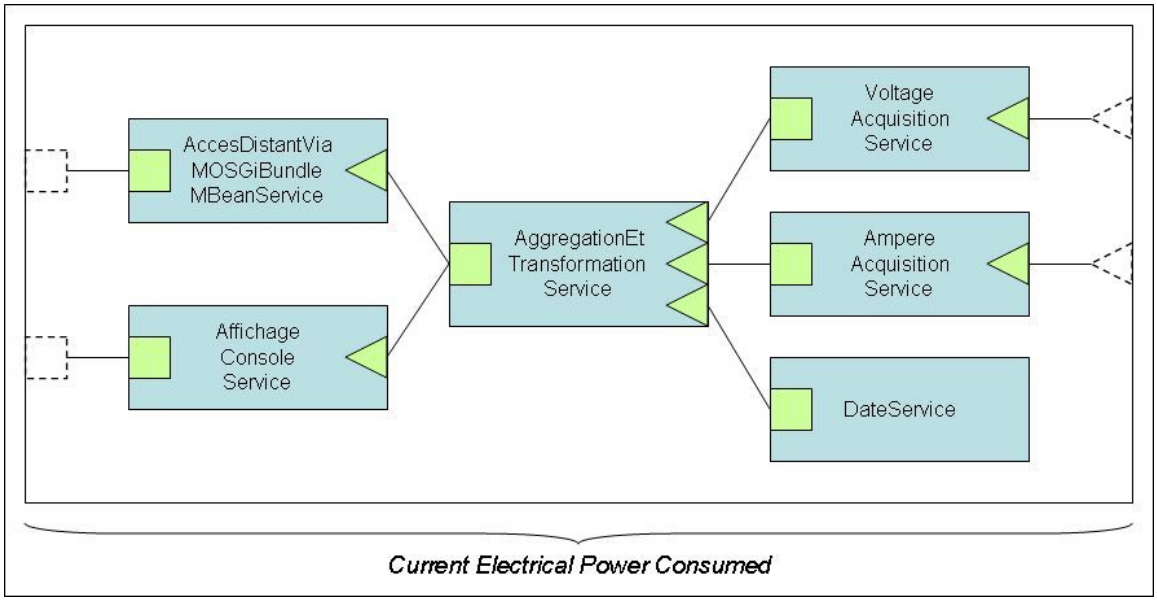


Figure 71. Représentation du modèle de l'application CEPC dans notre notation.

Pour remarque, nous n'avons pas fait apparaitre les identifiants des fournisseurs et demandeurs dans la figure ci-dessus afin de la rendre plus visible (cette information peut, par contre, être trouvée dans la figure ci-dessous).

```
<servicesApplication>

  <applicationId>CEPC</applicationId>

  <servicesApplicationProvidedServicesArrayList>
    <serviceIdProvidedInterfaceAndProvidedInterfaceInteractionType>
      <serviceId>AffichageConsoleService</serviceId>
      <providedInterface>dmsa.applipourlavalidation.affichageconsole.r4_1.contract.AffichageConsoleService</providedInterface>
      <providedInterfaceInteractionType>Server</providedInterfaceInteractionType>
    </serviceIdProvidedInterfaceAndProvidedInterfaceInteractionType>
    <serviceIdProvidedInterfaceAndProvidedInterfaceInteractionType>
      <serviceId>AccesDistantViaMOSGiBundleMBeanService</serviceId>
      <providedInterface>dmsa.applipourlavalidation.accesdistantviamosgibundlembean.r4_1.AccesDistantViaMOSGiBundleMBean</providedInterface>
      <providedInterfaceInteractionType>Server</providedInterfaceInteractionType>
    </serviceIdProvidedInterfaceAndProvidedInterfaceInteractionType>
  </servicesApplicationProvidedServicesArrayList>
  <resourcesArrayList>
    <serviceIdRequiredInterfaceAndRequiredInterfaceInteractionType>
      <serviceId>VoltageAcquisitionService</serviceId>
      <requiredInterface>dmsa.applipourlavalidation.voltmetre.r4_1.contract.VoltmetreService</requiredInterface>
      <requiredInterfaceInteractionType>Client</requiredInterfaceInteractionType>
    </serviceIdRequiredInterfaceAndRequiredInterfaceInteractionType>
    <serviceIdRequiredInterfaceAndRequiredInterfaceInteractionType>
      <serviceId>AmpereAcquisitionService</serviceId>
      <requiredInterface>dmsa.applipourlavalidation.amperemetre.r4_1.contract.AmperemetreService</requiredInterface>
      <requiredInterfaceInteractionType>Client</requiredInterfaceInteractionType>
    </serviceIdRequiredInterfaceAndRequiredInterfaceInteractionType>
  </resourcesArrayList>

  <contains>

    <service>
      <serviceIdInTheServicesApplication>VoltageAcquisitionService</serviceIdInTheServicesApplication>
      <providedInterface>dmsa.applipourlavalidation.voltacquisition.r4_1.contract.VoltageAcquisitionService</providedInterface>
      <providedInterfaceInteractionType>Server</providedInterfaceInteractionType>
      <requiredInterfacesHashMap>
        <requiredInterfaceAndRequiredInterfaceInteractionType>
          <requiredInterface>dmsa.applipourlavalidation.voltmetre.r4_1.contract.VoltmetreService</requiredInterface>
          <requiredInterfaceInteractionType>Client</requiredInterfaceInteractionType>
        </requiredInterfaceAndRequiredInterfaceInteractionType>
      </requiredInterfacesHashMap>
      <properties></properties>
    </service>

    <service>
      <serviceIdInTheServicesApplication>AmpereAcquisitionService</serviceIdInTheServicesApplication>
      <providedInterface>dmsa.applipourlavalidation.ampereacquisition.r4_1.contract.AmpAcquisitionService</providedInterface>
      <providedInterfaceInteractionType>Server</providedInterfaceInteractionType>
      <requiredInterfacesHashMap>
        <requiredInterfaceAndRequiredInterfaceInteractionType>
          <requiredInterface>dmsa.applipourlavalidation.amperemetre.r4_1.contract.AmperemetreService</requiredInterface>
          <requiredInterfaceInteractionType>Client</requiredInterfaceInteractionType>
        </requiredInterfaceAndRequiredInterfaceInteractionType>
      </requiredInterfacesHashMap>
      <properties></properties>
    </service>

    <service>
      <serviceIdInTheServicesApplication>DateService</serviceIdInTheServicesApplication>
      <providedInterface>dmsa.applipourlavalidation.date.r4_1.contract.DateService</providedInterface>
      <providedInterfaceInteractionType>Server</providedInterfaceInteractionType>
      <requiredInterfacesHashMap></requiredInterfacesHashMap>
      <properties></properties>
    </service>

    <service>
      <serviceIdInTheServicesApplication>AggregationEtTransformationService</serviceIdInTheServicesApplication>
      <providedInterface>dmsa.applipourlavalidation.aggregationettransformation.r4_1.contract.AggregationEtTransformationService</providedInterface>
      <providedInterfaceInteractionType>Server</providedInterfaceInteractionType>
      <requiredInterfacesHashMap>
        <requiredInterfaceAndRequiredInterfaceInteractionType>
          <requiredInterface>dmsa.applipourlavalidation.voltacquisition.r4_1.contract.VoltageAcquisitionService</requiredInterface>
          <requiredInterfaceInteractionType>Client</requiredInterfaceInteractionType>
        </requiredInterfaceAndRequiredInterfaceInteractionType>
        <requiredInterfaceAndRequiredInterfaceInteractionType>
          <requiredInterface>dmsa.applipourlavalidation.ampereacquisition.r4_1.contract.AmpAcquisitionService</requiredInterface>
          <requiredInterfaceInteractionType>Client</requiredInterfaceInteractionType>
        </requiredInterfaceAndRequiredInterfaceInteractionType>
        <requiredInterfaceAndRequiredInterfaceInteractionType>
          <requiredInterface>dmsa.applipourlavalidation.date.r4_1.contract.DateService</requiredInterface>
          <requiredInterfaceInteractionType>Client</requiredInterfaceInteractionType>
        </requiredInterfaceAndRequiredInterfaceInteractionType>
      </requiredInterfacesHashMap>
      <properties></properties>
    </service>
```

```
    <service>
      <serviceIdInTheServicesApplication>AffichageConsoleService</serviceIdInTheServicesApplication>
      <providedInterface>dmsa.applipourlavalidation.affichageconsole.r4_1.contract.AffichageConsoleService</providedInterface>
      <providedInterfaceInteractionType>Server</providedInterfaceInteractionType>
      <requiredInterfacesHashMap>
        <requiredInterfaceAndRequiredInterfaceInteractionType>
          <requiredInterface>dmsa.applipourlavalidation.aggregationettransformation.r4_1.contract.AggregationEtTransformationService</requiredInterface>
          <requiredInterfaceInteractionType>Client</requiredInterfaceInteractionType>
        </requiredInterfaceAndRequiredInterfaceInteractionType>
      </requiredInterfacesHashMap>
      <properties></properties>
    </service>

    <service>
      <serviceIdInTheServicesApplication>AccesDistantViaMOSGiBundleMBeanService</serviceIdInTheServicesApplication>
      <providedInterface>dmsa.applipourlavalidation.accesdistantviamosgibundlembean.r4_1.AccesDistantViaMOSGiBundleMBean</providedInterface>
      <providedInterfaceInteractionType>Server</providedInterfaceInteractionType>
      <requiredInterfacesHashMap>
        <requiredInterfaceAndRequiredInterfaceInteractionType>
          <requiredInterface>dmsa.applipourlavalidation.aggregationettransformation.r4_1.contract.AggregationEtTransformationService</requiredInterface>
          <requiredInterfaceInteractionType>Client</requiredInterfaceInteractionType>
        </requiredInterfaceAndRequiredInterfaceInteractionType>
      </requiredInterfacesHashMap>
      <properties></properties>
    </service>

    <interaction>
      <interactionId>AggregationEtTransformationService_VoltageAcquisitionService</interactionId>
      <provider>VoltageAcquisitionService</provider>
      <requester>AggregationEtTransformationService</requester>
    </interaction>

    <interaction>
      <interactionId>AggregationEtTransformationService_AmpereAcquisitionService</interactionId>
      <provider>AmpereAcquisitionService</provider>
      <requester>AggregationEtTransformationService</requester>
  </interaction>

    <interaction>
      <interactionId>AggregationEtTransformationService_DateService</interactionId>
      <provider>DateService</provider>
      <requester>AggregationEtTransformationService</requester>
    </interaction>

    <interaction>
      <interactionId>AffichageConsoleService_AggregationEtTransformationService</interactionId>
      <provider>AggregationEtTransformationService</provider>
      <requester>AffichageConsoleService</requester>
    </interaction>

    <interaction>
      <interactionId>AccesDistantViaMOSGiBundleMBeanService_AggregationEtTransformationService</interactionId>
      <provider>AggregationEtTransformationService</provider>
      <requester>AccesDistantViaMOSGiBundleMBeanService</requester>
    </interaction>

  </contains>

</servicesApplication>
```

Figure 72. Modèle de l'application CEPC sous la forme d'un fichier XML.

Pour remarque, dans la figure ci-dessus, les tags :

- "servicesApplicationProvidedServicesArrayList" définissent les fournisseurs de services (appartenant à des services de l'application) promus au niveau de l'application,
- "resourcesArrayList" définissent les demandeurs de services (appartenant à des services de l'application) promus au niveau de l'application,
- "service" servent à définir les services appartenant à l'application
- et les tags "interaction" servent, quant à eux, à spécifier les interactions (entre services) appartenant à l'application.

En complément, l'ordonnancement de l'ordre d'activation du modèle de l'application CEPC calculé par le DMSA lors de son enregistrement est :

1. AmpereAcquisitionService,
2. VoltageAcquisitionService,
3. DateService,
4. AggregationEtTransformationService,
5. AffichageConsoleService
6. et AccesDistantViaMOSGiBundleMBeanService.

Enfin, il n'y a aucune contrainte d'ordre quant à l'établissement de ces trois pré-conditions.

2.2 Déploiement dirigé par les modèles de l'application

Dans cette sous-section, nous allons dérouler le déploiement du modèle de l'application CEPC. Ainsi, nous allons expliciter le déroulement des activités d'installation, d'activation, de désactivation, de mise à jour statique de l'implémentation (qui effectue des calculs flottants) associée au service identifié par AggregationEtTransformationService dans CEPC par une autre implémentation de service (qui effectue les mêmes calculs mais qui retourne un résultat entier tronqué), d'activation (à nouveau), de désactivation et enfin de désinstallation.

Nous finirons cette sous-section avec deux tableaux qui présentent les performances du DMSA (en millisecondes). Le premier concerne les performances du DMSA pour le déploiement du modèle d'application CEPC en local, ce qui signifie que la partie serveur du DMSA, l'environnement cible et (donc) la partie agent du DMSA sont exécutés sur le même ordinateur et que les implémentations de services packagées (c'est-à-dire les unités de déploiement) sont elles aussi présentes sur ce même ordinateur. Le second tableau concerne les performances du DMSA pour le déploiement du modèle d'application CEPC en distribué, ce qui signifie que la partie serveur du DMSA est située sur un premier ordinateur (d'adresse IP : 192.168.0.3), que l'environnement d'exécution cible (et donc la partie agent du DMSA) est située sur un autre ordinateur (d'adresse IP : 192.168.0.4) et que les implémentations de services packagées sont situées sur un serveur web (d'adresse IP : 212.27.63.139).

L'environnement que nous ciblons contient d'ores et déjà un autre modèle d'application, à savoir l'application Date. L'application Date est constituée d'un service DateService similaire à celui utilisé dans CEPC. L'implémentation de service qui est utilisée pour réaliser le service DateService est sans état interne. Son identifiant est http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.date.r4_1-0.1.0.jar. L'application Date est dans l'état activé.

La première étape de ce déroulement est l'installation du modèle de CEPC. Pour ce faire, l'administrateur utilise la servlet ServicesApplicationsDeployment afin, d'une part, de spécifier l'identifiant du modèle de l'application (c'est-à-dire CEPC), l'identifiant de l'environnement d'exécution cible (c'est-à-dire 192.168.0.4) et l'activité de déploiement à effectuer (c'est-à-dire l'installation) et afin, d'autre part, de lancer l'exécution de l'activité. Une fois l'activité lancée, le DMSA commence par vérifier la validité de l'activité spécifiée, puis, il effectue la sélection des implémentations de services qui réaliseront les modèles de services appartenant à l'application (en favorisant les implémentations de services d'ores et déjà déployées dans l'environnement cible), ensuite, il génère le plan de déploiement correspondant qu'il envoie et fait appliquer par sa partie agent présente dans l'environnement cible, après cela, la partie agent du DMSA retourne le résultat de l'exécution du plan de déploiement à la partie serveur du DMSA qui va alors mettre à jour le modèle global associé à l'environnement d'exécution cible et afficher le résultat à l'administrateur.

Le plan de déploiement généré pour l'installation de CEPC comporte cinq instructions "needInstallation" pour les implémentations de services suivantes :

- **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.ampereacquisition.r4_1-0.1.0.jar
- **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.voltacquisition.r4_1-0.1.0.jar
- **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.affichageconsole.r4_1-0.1.0.jar
- **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.accesdistantviamosgibundlembean.r4_1-0.1.0.jar
- **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.aggregationettransformation.r4_1-0.1.0.jar

Pour remarque, le plan de déploiement ci-dessus ne contient pas d'instruction de déploiement pour le modèle de service DateService de l'application CEPC puisque l'environnement contient d'ores et déjà une implémentation de service (http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImple

mentations/dmsa.applipourlavalidation.date.r4_1-0.1.0.jar) qui correspond au modèle de DateService et puisque cette implémentation a été choisie par l'algorithme de sélection du DMSA.

L'exécution de ce plan par la partie agent du DMSA consiste à utiliser la primitive d'installation de *bundle* de l'environnement afin d'installer les implémentations de services. Elle consiste aussi à, si besoin est, rattraper les erreurs/exceptions d'installation, à tenter une restauration de l'état de déploiement de l'application et donc à tenter une restauration de l'état (du point de vue du déploiement) de l'environnement.

La seconde étape de ce déroulement est l'activation du modèle de CEPC. Pour ce faire, l'administrateur utilise la servlet ServicesApplicationsDeployment afin, d'une part, de spécifier l'identifiant du modèle de l'application (c'est-à-dire CEPC), l'identifiant de l'environnement d'exécution cible (c'est-à-dire 192.168.0.4), l'activité de déploiement à effectuer (c'est-à-dire l'activation) et afin, d'autre part, de lancer l'exécution de l'activité. Une fois l'activité lancée, le DMSA commence par vérifier la validité de l'activité spécifiée, puis, il génère le plan de déploiement correspondant qu'il envoie et fait appliquer par sa partie agent présente dans l'environnement cible, après cela, la partie agent du DMSA retourne le résultat de l'exécution du plan de déploiement à la partie serveur du DMSA qui va alors mettre à jour le modèle global associé à l'environnement d'exécution cible et afficher le résultat à l'administrateur.

Le plan de déploiement généré pour l'activation de l'application CEPC comporte une instruction pour chacun des six modèles de services qui la composent :

- **serviceId**=AmpereAcquisitionService,
  implemURL=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.ampereacquisition.r4_1-0.1.0.jar,
  deploymentInstruction=needActivation,
  instanceInstruction=createAndUseANewPersonalizedInstance,
- **serviceId**=VoltageAcquisitionService,
  implemURL=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.voltacquisition.r4_1-0.1.0.jar,
  deploymentInstruction=needActivation,
  instanceInstruction=createAndUseANewPersonalizedInstance,
- **serviceId**=DateService,
  implemURL=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.date.r4_1-0.1.0.jar,
  deploymentInstruction=nothingToDo,
  instanceInstruction=useCommonInstance,
- **serviceId**=AggregationEtTransformationService,
  implemURL=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.aggregationettransformation.r4_1-0.1.0.jar,
  deploymentInstruction=needActivation,
  instanceInstruction=createAndUseANewPersonalizedInstance,
- **serviceId**=AffichageConsoleService,
  implemURL=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.affichageconsole.r4_1-0.1.0.jar,
  deploymentInstruction=needActivation,
  instanceInstruction=createAndUseANewPersonalizedInstance,
- **serviceId**=AccesDistantViaMOSGiBundleMBeanService,
  implemURL=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.accesdistantviamosgibundlembean.r4_1-0.1.0.jar,
  deploymentInstruction=needActivation,
  instanceInstruction=createAndUseANewPersonalizedInstance

Pour remarque, les instructions de ce plan de déploiement sont ordonnancées pour l'activation.

Pour rappel, chaque implémentation de service (et donc *bundle* OSGi) contient une classe "Activator" dont la méthode "start(BundleContext)", respectivement "stop(BundleContext)", est exécutée par l'environnement d'exécution à l'activation, respectivement à la désactivation, du *bundle*. C'est grâce à cette classe que nous avons pu mettre en place l'approche *naming*. Ainsi, lors de l'activation d'un *bundle* et donc lors de la création de l'instance *singleton*, la méthode "start" correspondante appelle la partie agent du DMSA afin que cette dernière lui retourne l'instance (ou les instances) que le *singleton* en cours de création doit utiliser. Pour être tout à fait précis, notre approche *naming* retourne soit les bons services pour les dépendances (requis) de services internes à l'application, soit la liste de tous les services satisfaisants et externes à l'application pour les dépendances de services externes à l'application (c'est-à-dire pour les requis de services exprimés au niveau de l'application par des services internes à l'application).

Dans le cadre de l'application CEPC, ses requis de services VoltmetreService et AmpereService sont satisfaits par les implémentations de services :

- DMSA_AppliPourLaValidation_Voltmetre_R4_1
- et DMSA_AppliPourLaValidation_Amperemetre_R4_1.

Les figures ci-dessous présentent respectivement les implémentations de services activées telles qu'elles apparaissent dans l'environnement d'exécution OSGi R4.1 (c'est-à-dire Felix) et le message console qui est affiché (dans l'environnement d'exécution) par le service AffichageConsoleService lorsque CEPC répond à une demande d'informations ; ainsi que ces mêmes informations affichées du côté du serveur (le résultat du calcul de puissance est un résultat flottant).

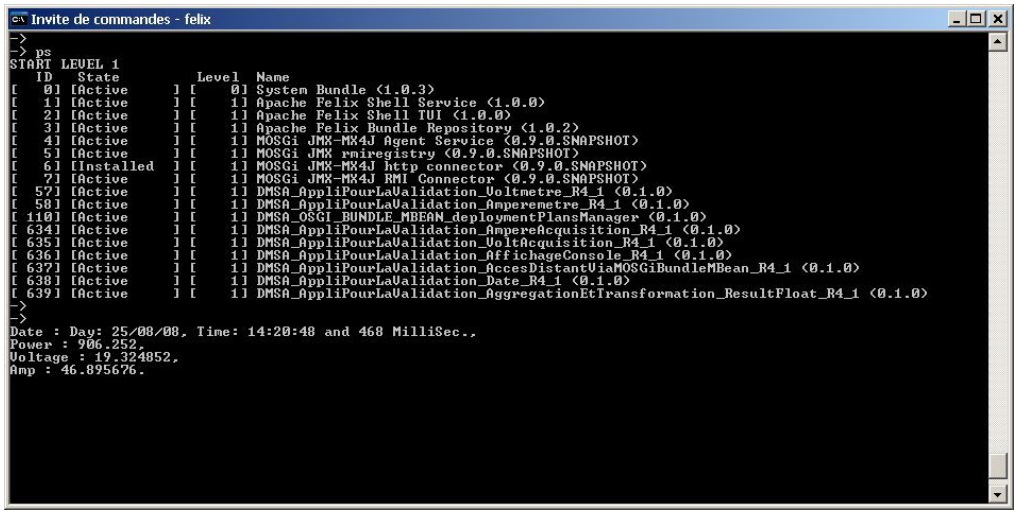


Figure 73. Le modèle d'application CEPC activé et utilisé dans l'environnement Felix.

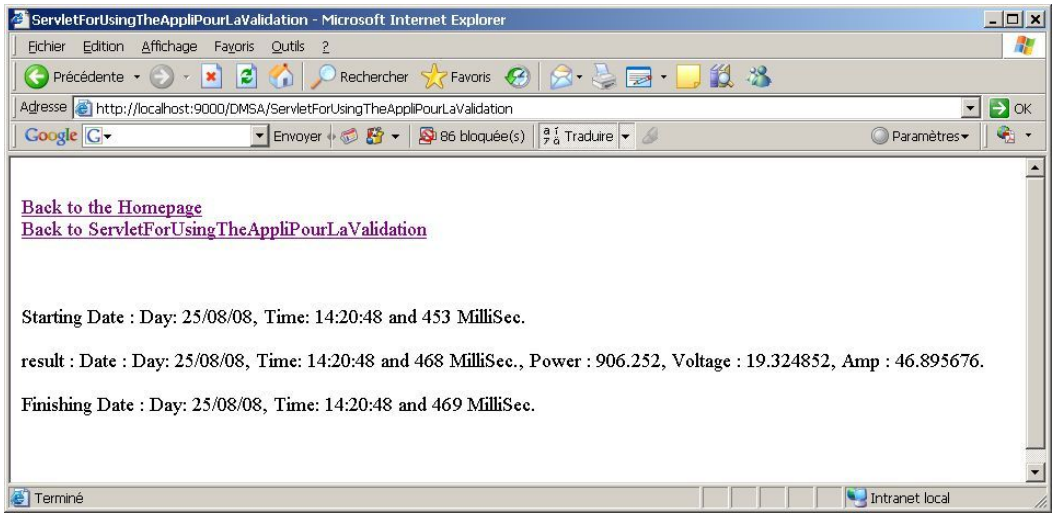


Figure 74. Affichage succinct, du côté du serveur, des données liées à l'utilisation de CEPC.

La troisième étape de ce déroulement est la désactivation du modèle de CEPC. Pour ce faire, l'administrateur utilise toujours la servlet ServicesApplicationsDeployment afin, d'une part, de spécifier l'identifiant du modèle de l'application (c'est-à-dire CEPC), l'identifiant de l'environnement d'exécution cible (c'est-à-dire 192.168.0.4) et l'activité de déploiement à effectuer (c'est-à-dire la désactivation) et afin, d'autre part, de lancer l'exécution de l'activité. Une fois l'activité lancée, le DMSA commence par vérifier la validité de l'activité spécifiée, puis, il génère le plan de déploiement correspondant qu'il envoie et fait appliquer par sa partie agent présente dans l'environnement cible, après cela, la partie agent du DMSA retourne le résultat de l'exécution du plan de déploiement à la partie serveur du DMSA qui va alors mettre à jour le modèle global associé à l'environnement d'exécution cible et afficher le résultat à l'administrateur.

Le plan de déploiement généré pour la désactivation de l'application CEPC comporte une instruction pour chacun des six modèles de services qui la composent :

- **serviceId**=AccesDistantViaMOSGiBundleMBeanService, **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.accesdistantviamosgibundlembean.r4_1-0.1.0.jar, **deploymentInstruction**=needDeActivation,
- **serviceId**=AffichageConsoleService, **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.affichageconsole.r4_1-0.1.0.jar, **deploymentInstruction**=needDeActivation,
- **serviceId**=AggregationEtTransformationService, **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.aggregationettransformation.r4_1-0.1.0.jar, **deploymentInstruction**=needDeActivation,
- **serviceId**=DateService, **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.date.r4_1-0.1.0.jar, **deploymentInstruction**=nothingToDo,
- **serviceId**=VoltageAcquisitionService, **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.voltacquisition.r4_1-0.1.0.jar, **deploymentInstruction**=needDeActivation,
- **serviceId**=AmpereAcquisitionService, **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.ampereacquisition.r4_1-0.1.0.jar, **deploymentInstruction**=needDeActivation,

Pour remarque, les instructions de ce plan de déploiement sont ordonnancées pour la désactivation. De plus, dans ce cas précis, l'implémentation de service correspondant au service DateService qui était utilisée via l'instance commune doit rester inchangée (cela signifie que l'instance "commonInstance" ne doit pas être détruite et que l'implémentation elle-même ne doit pas être désactivée), les cinq autres implémentations qui étaient utilisées chacune via une "PersonalizedInstance" doivent, quant à elles, être désactivées. Maintenant, si par exemple l'implémentation de service correspondante au service AmpereAcquisitionService était encore utilisée (dans le cadre d'au moins une autre application), alors l'instruction de déploiement, générée par la partie serveur du DMSA, la concernant aurait été :

- **serviceId**=AmpereAcquisitionService,
  implemURL=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.ampereacquisition.r4_1-0.1.0.jar,
  deploymentInstruction=needPersonalizedInstanceDestruction,
  instanceId=identifiant de l'instance.

La quatrième étape de ce déroulement est la mise à jour statique de l'implémentation de service correspondant au service ID (AggregationEtTransformationService) dans l'application d'ID (CEPC) par l'implémentation de service (aggregationettransformation.r4_1-0.1.1.jar). La première implémentation donne un résultat flottant de la puissance instantanée consommée, la seconde/nouvelle implémentation donne, quant à elle, un résultat entier tronqué. Pour ce faire, l'administrateur utilise la servlet ServicesApplicationsDeployment afin, d'une part, de spécifier l'identifiant du service cible (AggregationEtTransformationService), l'identifiant du modèle de l'application (CEPC), l'identifiant de l'environnement d'exécution cible (192.168.0.4), l'activité de déploiement à effectuer (c'est-à-dire la mise à jour statique de l'implémentation de service correspondant à l'identifiant de service donné) et afin, d'autre part, de lancer l'exécution de l'activité. Une fois l'activité lancée, le DMSA commence par vérifier la validité de l'activité spécifiée, puis, il génère le plan de déploiement correspondant qu'il envoie et fait appliquer par sa partie agent présente dans l'environnement cible, après cela, la partie agent du DMSA retourne le résultat de l'exécution du plan de déploiement à la partie serveur du DMSA qui va alors mettre à jour le modèle global associé à l'environnement d'exécution cible et afficher le résultat à l'administrateur.

Le plan de déploiement généré pour cette mise à jour statique de CEPC comporte deux instructions :

- **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.aggregationettransformation.r4_1-0.1.1.jar,
  deploymentInstruction=install,
- **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.aggregationettransformation.r4_1-0.1.0.jar,
  deploymentInstruction=deinstall.

Pour remarque, l'implémentation "aggregationettransformation.r4_1-0.1.0.jar" pèse 13313 octets et l'implémentation "aggregationettransformation.r4_1-0.1.1.jar" pèse, quant à elle, 13369 octets. C'est pourquoi la première a été choisie lors de l'installation initiale de CEPC.

La figure ci-dessous est une capture d'écran précisant l'état de l'environnement cible après cette mise à jour statique.

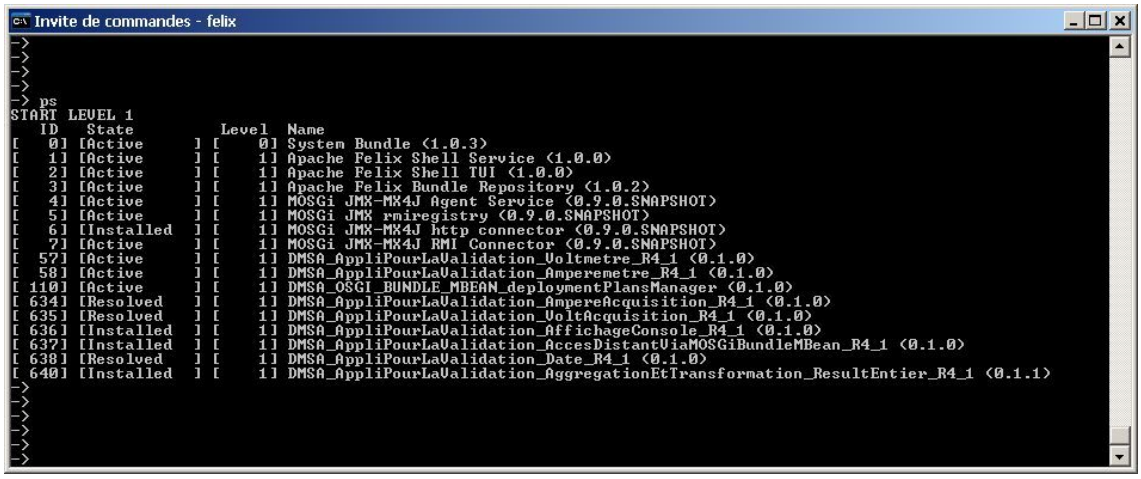


Figure 75. Capture d'écran de l'état de l'environnement Felix après cette mise à jour statique.

La cinquième étape de ce déroulement est la (ré-)activation de CEPC (la procédure suivie par le DMSA est toujours la même).

La figure ci-dessous présente les implémentations de services activées telles qu'elles apparaissent dans l'environnement d'exécution OSGi R4.1 (c'est-à-dire Felix) et le message console qui est affiché (dans l'environnement d'exécution) par le service AffichageConsoleService lorsque CEPC répond à une demande d'informations.

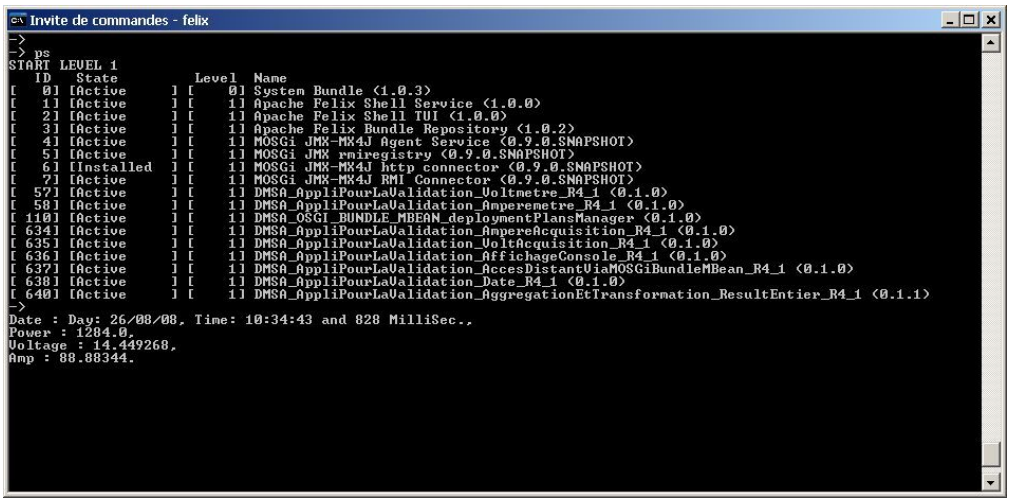


Figure 76. Le modèle d'application CEPC activé et utilisé dans l'environnement Felix.

La sixième étape de ce déroulement est la désactivation de CEPC (la procédure suivie par le DMSA est toujours la même). Le plan de déploiement généré pour la désactivation de CEPC cible cette fois l'implémentation dmsa.applipourlavalidation.aggregationettransformation.r4_1-0.1.1.jar au lieu de dmsa.applipourlavalidation.aggregationettransformation.r4_1-0.1.0.jar

La septième et dernière étape de ce déroulement est la désinstallation de CEPC. Le plan de déploiement généré pour la désinstallation de CEPC comporte cinq instructions "needDeInstallation" pour les implémentations de services suivantes :

- **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.aggregationettransformation.r4_1-0.1.1.jar
- **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.ampereacquisition.r4_1-0.1.0.jar

- **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.voltacquisition.r4_1-0.1.0.jar
- **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.affichageconsole.r4_1-0.1.0.jar
- **implemURL**=http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/dmsa.applipourlavalidation.accesdistantviamosgibundlembean.r4_1-0.1.0.jar

Le tableau ci-dessous présente les temps d'exécution des sept activités et sous-activités de déploiement offertes par le DMSA sur le modèle d'application CEPC, ainsi que ceux relatifs à l'utilisation de CEPC. Les résultats sont donnés en millisecondes. Ces exécutions ont été faites en distant, ce qui signifie que la partie serveur du DMSA est située sur la machine d'adresse IP : 192.168.0.3, que l'environnement d'exécution cible (et donc la partie agent du DMSA) est située sur une autre machine d'adresse IP : 192.168.0.4 et que les implémentations de services packagées sont situées sur un serveur web d'adresse IP : 212.27.63.139.

Pour remarque, les abréviations "M.à.J 1", " M.à.J 2" et " M.à.J 3"signifie respectivement la mise à jour statique d'une implémentation par une autre, la mise à jour statique d'une implémentation par une autre dans une application donnée et enfin, la mise à jour statique de l'implémentation correspondant à un service donné d'une application donnée par une autre implémentation.

Activité	Inst.	Act.	*Usage*	DeAct.	DeInst.	M.à.J.1	M.à.J.2	M.à.J.3
Temps en milli secondes	1297 1110 1156 797 735 735 766 1109 1063 735	375 235 219 234 188 188 203 172 187 203	16 31 31 16 31 16 15 16 19 16	187 203 109 94 94 94 94 109 94 94	234 157 172 156 156 141 140 141 156 140	454 422 359 344 359 360 344 344 344 344	406 375 375 360 359 344 328 344 344 344	469 296 444 375 359 344 360 390 375 359
Moyenne	**950,3**	**220,4**	**20,7**	**117,2**	**159,3**	**367,4**	**357,9**	**377,1**

Tableau 10. Résultat de l'exécution, en distant, des sept activités et sous-activités de déploiement sur le modèle d'application CEPC et de l'utilisation de CEPC.

Enfin, le tableau ci-dessous présente les temps d'exécution des sept activités et sous-activités de déploiement offertes par le DMSA sur le modèle d'application CEPC, ainsi que ceux relatifs à l'utilisation de CEPC. Les résultats sont donnés en millisecondes. Ces exécutions ont été faites en local. Le terme "en local" signifie que la partie serveur du DMSA, l'environnement d'exécution cible (et donc la partie agent du DMSA) ainsi que les implémentations de services sont situées sur la même machine.

Activité	Inst.	Act.	*Usage*	DeAct.	DeInst.	M.à.J.1	M.à.J.2	M.à.J.3
Temps en milli secondes	375	234	16	94	141	125	203	110
	312	188	31	78	172	125	79	125
	266	187	16	94	141	109	109	125
	266	203	16	109	141	78	93	78
	266	157	32	94	172	93	94	94
	265	187	15	109	140	109	94	94
	266	188	16	107	140	94	109	140
	323	187	16	110	141	94	94	104
	297	219	16	109	172	78	94	75
	296	188	16	94	141	94	93	109
Moyenne	**293,2**	**193,8**	**19,0**	**99,8**	**150,1**	**99,9**	**106,2**	**105,4**

Tableau 11. Résultat de l'exécution, en local, des huit activités et sous-activités de déploiement sur le modèle d'application CEPC et de l'utilisation de CEPC.

Comme attendu, les exécutions en local des sept activités et sous-activités de déploiement sur CEPC ainsi que l'utilisation de CEPC consomment toutes moins de temps que leur homologues exécutées en distant. En outre, le point discriminant concernant l'utilisation/la mise en jeu de la primitive d'installation d'implémentation de service (c'est-à-dire de *bundle*) et donc du transfert de *bundles* au-dessus du réseau permet de distinguer les activités qui sont très sensibles au réseau. En l'occurrence, les activités d'installation et les trois sous-activités de mise à jour statique sont toutes très sensibles au réseau, puisqu'elles utilisent toutes la primitive d'installation d'implémentation ; l'activité d'installation est, bien évidemment, celle qui l'utilise le plus. Pour les trois activités restantes et l'utilisation de CEPC, l'observation des résultats permet de distinguer une légère différence entre les temps réalisés en local et ceux réalisés en distant. Cette légère différence (pour rappel, les temps donnés ici le sont en millisecondes) s'explique par l'utilisation du réseau, c'est-à-dire par le contact, l'envoi du plan de déploiement et le retour du résultat correspondant.

2.3 Le mécanisme de *Rollback*

Dans cette sous-section, nous présentons les temps d'exécution relatifs au mécanisme de *rollback* lors de l'apparition d'une erreur/exception au niveau du déploiement de l'implémentation de service correspondant au service AggregationEtTransformationService dans le modèle d'application CEPC et cela pour les activités d'installation, d'activation, de désactivation et pour les trois sous-activités de mises à jour statique. Nous ne présentons aucun résultat concernant le mécanisme de *rollback* lié à l'activité de désinstallation ; en effet, nous n'avons pas trouvé le moyen de faire apparaitre une erreur/exception lors de l'activité de désinstallation (plus précisément lors de la désinstallation d'une implémentation de service suite à l'appel de la méthode de désinstallation d'un *bundle* fournie par l'environnement).

Pour rappel, le mécanisme de *rollback* lié à l'activité de désactivation est le plus complexe, puisqu'il implique le retour de l'application dans l'état activé et donc d'utilisation de l'approche *naming*.

Activité	Inst.	Act.	DeAct.	DeInst.	M.à.J. 1	M.à.J. 2	M.à.J. 3
Temps en milli secondes	1953 1921 1890 1906 1991 1906 1859 1860 1875 1859	157 157 110 109 125 125 125 110 109 140	125 110 93 94 109 94 93 94 109 94	// // // // // // // // // //	282 281 297 328 282 281 282 296 312 297	313 297 297 299 281 313 297 281 297 281	375 297 297 313 297 297 313 297 297 312
Moyenne	**1902**	**126,7**	**101,5**	**//**	**293,8**	**295,6**	**309,5**

Tableau 12. Résultat de l'exécution, en distant, des activités et sous-activités de déploiement avec l'intervention du mécanisme de *rollback* pour rattraper les exceptions liées au service AggregationEtTransformationService du modèle d'application CEPC.

2.4 Déploiement dirigé par les modèles d'une liste ordonnée de tâches

Dans cette sous-section, nous présentons les temps d'exécutions relatifs au déploiement de listes ordonnées de tâches de déploiement de façon séquentielle, puis de façon parallèle.

Dans un premier temps, nous nous focalisons sur une liste composée par les sept tâches de déploiement que nous avons déroulées tout au long de la sous-section 2.2. En conséquence, la liste manipulée spécifie l'installation, l'activation, la désactivation, une des trois mises à jour statiques, l'activation, la désactivation et la désinstallation de CEPC.

Dans un second temps, nous nous focaliserons sur l'exécution d'une même activité de déploiement sur un puis plusieurs environnements.

Il est à noter que chaque exécution sera répétée dix fois afin d'obtenir un résultat moyen. Le but de l'utilisation d'une moyenne est d'adoucir les variations liées à l'utilisation du réseau.

Pour le premier temps, la partie serveur du DMSA est située sur la machine d'adresse IP 192.168.0.3, les environnements cibles sont situés respectivement sur les machines d'adresses IP 192.168.0.4, puis 192.168.0.4 et 192.168.0.5, puis 192.168.0.4, 192.168.0.5 et 192.168.0.6 et, enfin, 192.168.0.3, 192.168.0.4, 192.168.0.5 et 192.168.0.6 ; les implémentations de services sont situées sur la machine d'adresse IP 212.27.63.139. Les résultats des exécutions de la liste composée par les tâches ciblant CEPC sur un unique environnement, puis sur deux environnements, en utilisant l'algorithme séquentiel puis l'algorithme parallèle sont donnés dans le tableau suivant.

	Exécution séquentielle sur 1 env.	Exécution parallèle sur 1 env.	Exécution séquentielle sur 2 env.	Exécution parallèle sur 2 env.
Temps en milli secondes	2186 2186 2201 2186 2165 2186 2201 2171 2155 2187	2342 2299 2303 2303 2316 2280 2288 2316 2316 2282	5734 5625 5562 5797 5859 5969 5687 5875 5547 5563	3375 3438 3376 3250 3312 3219 3390 3125 3281 3188
Moyenne	**2182,4**	**2304,5**	**5721,8**	**3295,4**

Tableau 13. Résultat de l'exécution, en distant et sur un puis deux environnements cibles, d'une liste ordonnée composée de sept tâches de déploiement par environnement via les algorithmes séquentiel et parallèle du DMSA.

Comme attendu, les exécutions de la liste ordonnée de tâches qui ciblent le même environnement prennent un peu plus de temps lorsqu'elles sont exécutées via l'algorithme parallèle que lorsqu'elles sont exécutées en utilisant l'algorithme séquentiel. Ce surcoût est vraisemblablement dû au calcul de la distribution des tâches qui est effectué par l'algorithme parallèle et qui est inutile ici. La perte moyenne est, ici, de 5,59%.

Les exécutions de la liste ordonnée de tâches qui ciblent deux environnements prennent plus de temps lorsqu'elles sont exécutées via l'algorithme séquentiel que lorsqu'elles le sont en utilisant l'algorithme parallèle. Ici, le gain est de 42,41%.

Les résultats des exécutions de la liste composée par les tâches ciblant CEPC sur trois environnements, puis sur quatre environnements, en utilisant l'algorithme séquentiel puis l'algorithme parallèle sont donnés dans le tableau suivant.

	Exécution séquentielle sur 3 environnements	Exécution parallèle sur 3 environnements	Exécution séquentielle sur 4 environnements	Exécution parallèle sur 4 environnements
Temps en milli secondes	8828 8500 8515 8531 8750 8756 8516 8563 8610 8421	3453 3532 3703 3594 3453 3515 3469 3515 3437 3343	12093 11844 12187 11797 11766 11953 11469 11531 11797 12157	4063 4047 4141 4046 3968 3829 4547 4359 4828 3953
Moyenne	**8599**	**3501,4**	**11859,4**	**4178,1**

Tableau 14. Résultat de l'exécution, en distant et sur trois puis quatre environnements cibles, d'une liste ordonnée composée de sept tâches de déploiement par environnement (donc vingt et une puis vingt huit tâches en tout) via les algorithmes séquentiel et parallèle du DMSA.

Comme attendu, les exécutions de listes ordonnées de tâches qui ciblent trois puis quatre environnements prennent plus de temps lorsqu'elles sont exécutées via l'algorithme séquentiel que lorsqu'elles le sont en utilisant l'algorithme parallèle. Dans nos jeux de tests, le gain est de 59,28% lorsque trois environnements sont ciblés et il est de 64,77% lorsque quatre environnements sont ciblés.

La figure ci-dessous présente deux captures d'écran. La première capture est celle de l'affichage du résultat de l'exécution séquentielle. La seconde est celle correspondant à l'exécution parallèle.

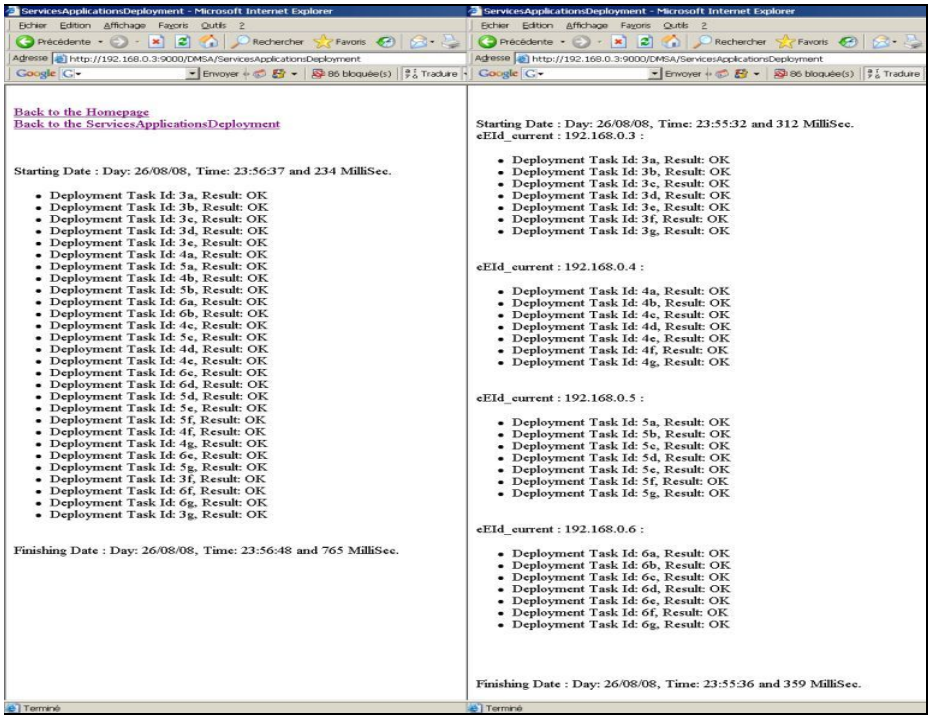


Figure 77. Résultats d'exécutions séquentielle et parallèle de la même liste de tâches.

La figure (ci-dessous) récapitule les gains de temps en pourcentage observés lorsque l'algorithme parallèle est utilisé au lieu de l'algorithme séquentiel pour l'exécution des sept tâches par environnement. Elle nous permet de supposer que plus le nombre d'environnements ciblés est grand, plus le gain de temps est grand.

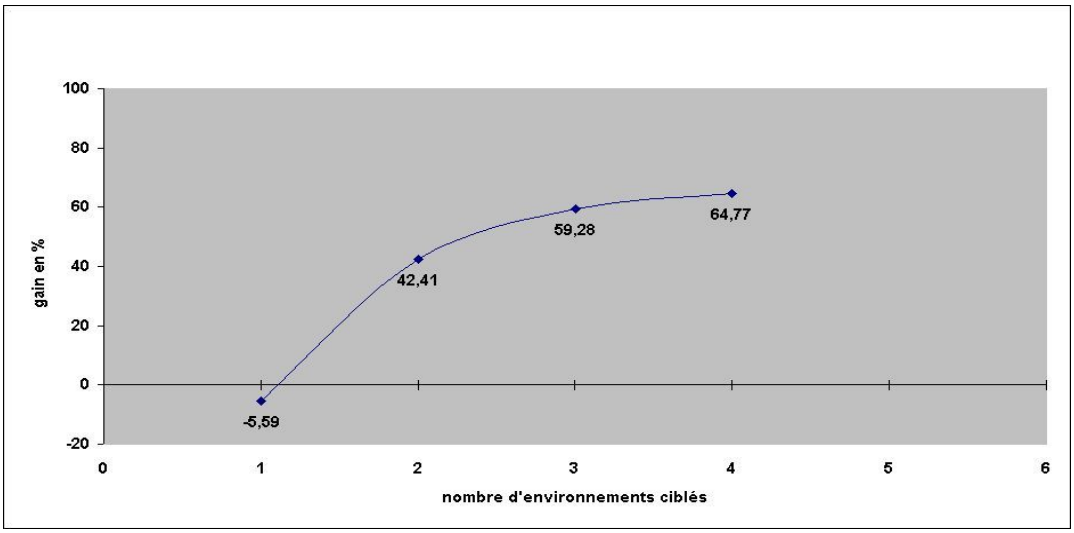


Figure 78. Courbe représentant les gains de temps réalisés en utilisant l'algorithme parallèle au lieu de l'algorithme séquentiel en fonction du nombre d'environnements ciblés.

En complément, il intéressant de noter :

- que le gain en pourcentage sera toujours strictement inférieur à 100% (c'est-à-dire à 1),
- que le temps théorique consommé par une exécution séquentielle est égal à la ∑(des temps d'exécution de chaque tâche),
- que le temps théorique consommé par une exécution parallèle est égal au MAX de la ∑(des temps d'exécution des tâches ciblant un même environnement),

Ensuite, de manière théorique et en faisant l'hypothèse que le nombre de tâches attribuées par environnement est le même et que le temps consommé par chaque tâche est le même, alors le gain maximal entre l'algorithme séquentiel et l'algorithme est de 100*(1-1/n) où "n" est le nombre d'environnements ciblés (compris entre 1 et *).

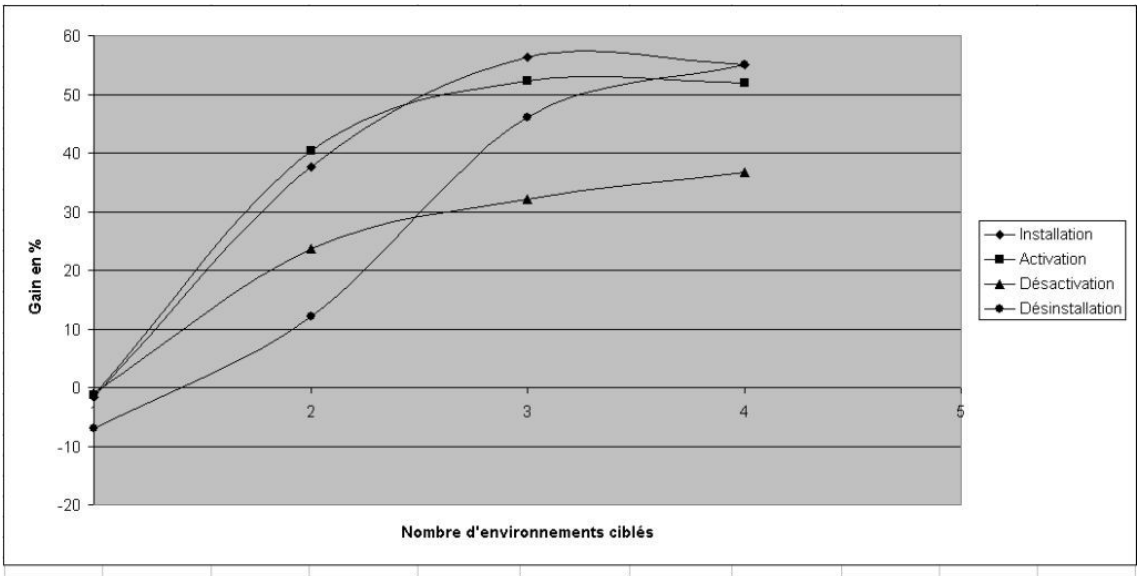
Une autre manière d'observer le gain apporté par l'utilisation de l'algorithme parallèle est, non pas de faire exécuter un scénario donné comprenant ici sept tâches de déploiement (ciblant CEPC) sur un puis plusieurs environnements, mais de faire exécuter des scénarios ne se concentrant que sur une unique activité de déploiement et cela toujours sur un puis plusieurs environnements.

Pour ce faire, nous nous sommes concentrés sur les quatre activités de déploiement les plus fondamentales, à savoir, l'installation, l'activation, la désactivation et la désinstallation. Ainsi le premier scénario porte sur l'installation de CEPC. Il est appliqué sur un, puis deux, puis trois et enfin quatre environnements cibles. Le second scénario porte sur l'activation de CEPC, le troisième porte sur sa désactivation, enfin, le quatrième porte sur sa désinstallation. Ils sont, eux-aussi, appliqués sur un, puis deux, puis trois et enfin quatre environnements. Chacun de ces scénarios est appliqué dix fois.

Le tableau ci-dessous présente les résultats moyens obtenus.

	Activité / Algorithme	Installation	Activation	Désactivation	Désinstallation
1 env.	séq.	950	220	117	159
	para.	966	223	118	170
2 env.	séq.	2162	474	232	665
	para.	1347	283	177	585
3 env.	séq.	3133	672	311	1287
	para.	1368	321	211	695
4 env.	séq.	3332	897	424	1493
	para.	1497	377	273	671

Le graphe ci-dessous donne quant à lui, les gains en pourcentage en fonction de l'activité de déploiement et du nombre d'environnements cibles.



Nombre d'env.	Installation	Activation	Désactivation	Désinstallation
1	-1,68	-1,36	-0,85	-6,92
2	37,7	40,3	23,71	12,03
3	56,34	52,23	32,15	46
4	55,07	51,97	36,81	55,06

Ces résultats montrent que le gain obtenu par l'utilisation de l'algorithme parallèle est intéressant, voire très intéressant dès que le nombre d'environnements cibles est supérieur ou égal à deux et quelque soit l'activité de déploiement. Pour remarque, comme constaté précédemment, l'utilisation de l'algorithme parallèle est pénalisante si un seul environnement est ciblé.

3. CONCLUSION

Dans cette thèse, nous proposons le DMSA comme un moyen de permettre et d'automatiser le déploiement d'applications à services dans des environnements d'exécutions à services. Plus précisément, nous avons cherché à adresser les besoins et problématiques suivants :

- La gestion des changements pour les logiciels déployés (et donc leur configuration).
- La coordination des activités du déploiement et la flexibilité associée.
- Les réseaux et l'intégration d'Internet.
- L'hétérogénéité des plateformes.
- Résolution des dépendances (isolation, partage des composants, le déploiement doit prendre en compte les relations inter composants et leur utilisation par les autres applications).
- La sécurité.
- Et le traitement des exceptions

Cette thèse répond à cinq des sept besoins, à savoir, la coordination des activités de déploiement, les réseaux et l'intégration d'Internet, l'hétérogénéité des plateformes, la résolution des dépendances et le traitement des exceptions pouvant apparaitre lors du déploiement.

En outre, nous avons choisi :

- de déployer non pas des services et leurs dépendances, mais des applications à services,
- de passer non pas par une configuration, mais par une vue déploiement des applications à services, c'est-à-dire par un méta-modèle déploiement,
- de séparer les niveaux modèles et réalité/implémentation en concevant le DMSA de façon à ce qu'il soit dirigé par les modèles, c'est-à-dire de façon à ce que son raisonnement se fasse au niveau modèle,

Et, nous avons du proposer :

- une solution à la problématique de l'ordonnancement des activations et désactivations des (modèles d') applications à services
- et un algorithme d'exécution de tâches de déploiement en parallèle (qui pourra plus tard être intégré avec un gestionnaire de parcs d'environnements d'exécutions).

CONCLUSION

CONCLUSION ET PERSPECTIVES

Dans le cadre de cette thèse, nous avons proposé un environnement automatisant le déploiement d'applications à services sur des environnements d'exécutions à services. Cet environnement de déploiement est nommé *Deployment Manager for Services Applications* (DMSA). Nous l'avons réalisé en suivant une approche dirigée par les modèles.

Dans ce dernier chapitre, nous synthétisons nos propositions et, à cette occasion, nous reprenons certaines réflexions dans le but de mettre en avant nos principales contributions, d'identifier les questions qui restent en suspens et de présenter les perspectives de notre travail.

1. BESOIN D'OUTILS DE DÉPLOIEMENT

L'approche à composant permet de manipuler des entités logicielles clairement identifiées. L'approche à service, dont les propriétés sont le couplage faible, la liaison retardée, la substituabilité, les accords de niveaux de service, le courtage et la négociation et, enfin, l'interaction au-dessus de domaines d'administration disjoints, permet de découpler les entités logicielles les unes des autres (ces entités pouvant appartenir au même logiciel ou au même système intégré/d'intégration). La propriété de découplage ainsi introduite n'est pas sans conséquence au niveau du déploiement logiciel. En effet, chaque service se trouve être une entité logicielle à part entière (c'est-à-dire *packagée* de façon individuelle) et donc, par la même, une unité de déploiement à part entière. L'approche à service offre donc un paradigme, une base intéressante pour la spécification et la création de compositions de services - des applications à services dans notre cas.

Ces applications à services peuvent, ensuite, être soumises à un "déploiement au plus près", c'est-à-dire à un déploiement capable d'appréhender des instructions de déploiement au niveau des environnements d'exécution, des applications à services et/ou des services et de les traduire en instructions qui seront appliquées aux niveaux implémentations de services et instances, c'est-à-dire au plus près des services (des unités de développement) qui composent les applications elles-mêmes.

De façon synthétique, grâce à l'approche à composants et aux langages de descriptions d'architectures (ADL, [MT00]) les logiciels sont passés du statut d'entité monolithique au statut d'entité modulaire (ou multi-lithiques) du point de vue de la phase de développement. Maintenant, avec l'approche à service, ces logiciels modulaires (du point de vue du développement), mais toujours monolithiques ou uniquement découpés suivant leurs contraintes de répartition/distribution du point de vue du déploiement, sont aussi devenus des logiciels modulaires pour la phase de déploiement ; en l'occurrence, le grain des unités de développement et des unités de déploiement se sont fortement rapprochés, voire sont devenus identiques.

Or déployer un logiciel modulaire sur plusieurs environnements cibles est complexe. Ainsi, le déploiement d'applications à services sur des environnements d'exécutions à services fait apparaitre plusieurs défis. En effet, il est nécessaire de définir la notion d'application à services (où l'application à services est clairement un logiciel modulaire), de définir le déploiement pour les applications à services, d'intégrer l'architecture globale définie par le contexte dans lequel le déploiement doit avoir lieu, de résoudre les dépendances de services, d'ordonnancer l'activation et la désactivation des applications à services, de proposer un mécanisme qui puisse rattraper des erreurs pouvant apparaitre lors du déploiement, de gérer les partages au niveau implémentation de service et instance d'implémentation de service, ainsi que l'isolation entre les applications et d'appréhender le contexte embarqué des environnements ciblés par le contexte. En outre, à la complexité du déploiement d'un logiciel modulaire dans un environnement vient s'ajouter la complexité du passage à l'échelle d'un tel déploiement. Il n'est nul besoin ici de penser grand ou très grand, il suffit de cibler le déploiement (sensible au contexte) de cinq applications composées chacune de dix services dans cinq environnements cibles. Ainsi, le dernier défi de notre problématique a été l'appréhension d'un déploiement sensible au contexte multi-applications et multi-environnements à travers des mécanismes d'exécutions de listes de tâches de déploiement.

2. APPROCHE DIRIGÉE PAR LES MODÈLES

Nous avons recherché les propriétés suivantes à travers l'utilisation d'une approche dirigée par les modèles qui, en outre, propose un découplage total entre les niveaux modèles et réalité :

- Passer à l'échelle. En effet, déployer des logiciels à la main ou en suivant une approche basée sur des scripts ne passe pas à l'échelle [HS05].
- Appréhender l'hétérogénéité des environnements et des implémentations de services.
- Introduire un niveau application à services au-dessus des niveaux implémentation de service et instance d'implémentation de service.
- Offrir un système de déploiement qui supporte tous les modèles d'application à services conformes au méta-modèle défini.

Exécuter un déploiement en étant dirigé par les modèles revient donc à valider la tâche/les instructions de déploiement (passée en paramètre d'entrée par l'administrateur) au regard des informations de déploiement maintenues par le système de déploiement, puis à traduire cette tâche exprimée au niveau modèle d'application à services (et donc au niveau modèle de services) en instructions de déploiement pour les niveaux implémentation et instance et à générer le plan de déploiement correspondant qui sera exécuté dans l'environnement ; le résultat de son exécution servira à la mise à jour des informations de déploiement maintenues et à informer l'administrateur.

L'utilisation d'une approche dirigée par les modèles pour le déploiement d'applications à services a relevé plusieurs points positifs, mais aussi plusieurs points moins positifs. Les points positifs sont :

- Les quatre propriétés que nous ciblions ont pu être obtenues.
- La conception et le développement des algorithmes sont simplifiés puisque seules les informations identifiées dans les méta-modèles interviennent.
- La validation des tâches de déploiement peut être faite directement via les modèles mis en jeu, c'est-à-dire avant de lancer des calculs (par exemple, de génération de plan de déploiement) et/ou avant de commencer à interagir avec la "réalité" (comme, par exemple, chercher à désactiver une application qui n'est pas active).
- Enfin, d'une manière générale, cette approche est efficace puisque dès lors qu'un outil est conçu et réalisé pour un méta-modèle, il permet d'appréhender les modèles qui sont conformes à ce méta-modèle.

Les points moins positifs sont :

- L'identification des informations qui doivent apparaitre dans les méta-modèles est difficile (c'est-à-dire l'identification des éléments relevant, dans notre cas, du déploiement).
- La projection des résultats des calculs (réalisés au niveau modèle) vers la réalité est complexe.
- De même, il est aussi complexe d'assurer la synchronisation entre les niveaux modèles et réalité. La mise en place d'un mécanisme transactionnel parait impérative, le DMSA fournit une première étape concernant le niveau implémentation de service.
- La mise à jour des modèles est difficile à faire, la représentation d'un modèle sous forme d'objets peut donner lieu à des objets complexes dont la manipulation (et donc la mise à jour) n'est pas triviale.
- L'évolution des méta-modèles est simple, mais l'évolution des outils liés à des méta-modèles ne l'est pas forcément.

Enfin, comme souhaité le découplage total entre le méta-modèle pour les applications à services et le niveau implémentation nous a permis de minimiser le nombre d'implémentations de services déployées par environnements. Cependant, l'introduction des mises à jour statiques ciblant respectivement une application donnée ou un service d'une application donnée nous montre les limites de l'utilisation d'un tel découplage. En effet, lorsque l'administrateur spécifie la mise à jour statique d'une implémentation de service par une autre implémentation de service dans une application (ou pour un service d'une application), cela signifie qu'il souhaite associer les services correspondants avec cette nouvelle implémentation et donc qu'il souhaite coupler ces services avec cette nouvelle implémentation. Ainsi,

l'utilisation de ces deux mises à jour casse, dans le cas général, la propriété de minimisation du nombre d'implémentations de services déployées puisque l'implémentation ciblée par la mise à jour peut être nécessitée par d'autres services appartenant à la même ou à d'autres applications, elle ne va donc pas être désinstallée de l'environnement. Au final, deux cas se présentent :

- Soit la propriété de minimisation du nombre d'implémentations de services déployées dans un environnement est vitale, alors le découplage doit être maintenu, ce qui va en conséquence restreindre les activités de déploiement offertes à l'administrateur.
- Soit cette propriété n'est pas vitale, alors il est peut être souhaitable de réintroduire du couplage pour pouvoir offrir un éventail d'activités de déploiement plus large, quitte à introduire un mécanisme de factorisation des implémentations de services (à exécuter de temps en temps).

3. PERSPECTIVES

En ce qui concerne l'avenir de ces travaux, six perspectives sont envisagées. La première cible les déploiements multiples, c'est-à-dire les déploiements multi-applications et multi-environnements. Pour cela, il serait intéressant d'intégrer les mécanismes de maintien des modèles des environnements et d'exécution parallélisée de tâches de déploiement du DMSA avec le prototype ORYA et plus particulièrement avec ses mécanismes de gestion de parcs d'environnements et de sélection de version d'unités de déploiement. Pour remarque, ORYA est un prototype qui a été développé au sein de l'équipe ADELE lors des thèses de V. Lestideau et N. Merle [Les03] [Mer05].

La seconde perspective cible la conception et le développement d'un algorithme de factorisation des implémentations de services. En effet, avec l'introduction du mécanisme de mise à jour statique de l'implémentation de service correspondant à un service donné d'une application donnée dans un environnement donné, il est possible que plusieurs implémentations de services de modèles similaires viennent à être déployées dans le même environnement. En conséquence, afin de toujours minimiser le nombre d'implémentations de services déployées dans chaque environnement, il serait intéressant de pouvoir bénéficier d'un tel mécanisme de factorisation d'implémentations de services.

La troisième perspective se focalise sur la conception et le développement d'un algorithme de déploiement d'un service. Ce mécanisme serait clairement très proche d'un mécanisme comme OBR [Hal07]. Cependant, son intérêt résiderait dans le fait qu'il serait basé au-dessus du DMSA, il pourrait donc, ainsi, tirer profit des propriétés de déploiement offertes par le DMSA. En conséquence, ce mécanisme commencerait par résoudre les dépendances de services du service ciblé, puis il enregistrerait le service ciblé et les services résolvants ces dépendances sous la forme d'une application à services auprès du DMSA, le service ciblé et les services résolvant ces dépendances pourraient alors être déployés sous la forme d'une application, ce qui permettrait, par la même, de tirer parti des propriétés de déploiement du DMSA pour le déploiement d'un service.

La quatrième perspective concerne le support des applications à services récursives, c'est-à-dire d'applications à services composées de services et d'applications à services (qui peuvent elles-mêmes être composées de services et d'applications à services, etc.).

La cinquième perspective cible l'extension du DMSA en lui ajoutant l'activité d'évolution pour les applications à services. L'ajout des activités de mises à jour dynamiques et de dynamisme pour les applications à services est lui aussi à envisager, cependant ces deux derniers ajouts risquent de couter très cher au niveau de l'intégration technologique.

Enfin, la sixième perspective, qui à notre avis est la plus intéressante, est d'introduire des connecteurs comme objets premiers dans les applications à services (c'est-à-dire, comme entité et donc unité de déploiement à part entière). L'objectif serait ensuite d'étudier le déploiement de ces dits connecteurs. Pour remarque, cette perspective de recherche n'est pas nouvelle, elle a déjà été mise en avant au niveau des langages de description d'architecture (ADL) [MT00], mais avec peu de succès. Cela dit, d'une manière générale, le monde des connecteurs reste un domaine de recherche ouvert, en particulier en ce qui concerne les connecteurs dans l'approche à service et le déploiement de connecteurs.

ANNEXES

Le premier élément de cette annexe est le fichier XML qui décrit la liste ordonnée qui a été utilisée pour la validation du gain entre l'utilisation des algorithmes séquentiel et parallèle pour l'exécution de cette liste sur deux environnements cibles.

```
<deploymentTasks>

<deploymentTask>
        <dTId>4a</dTId>
        <deploymentActivity>installation</deploymentActivity>
        <eEId>192.168.0.4</eEId>
        <applicationId>CEPC</applicationId>
</deploymentTask>

<deploymentTask>
        <dTId>5a</dTId><deploymentActivity>installation</deploymentActivity>
        <eEId>192.168.0.5</eEId><applicationId>CEPC</applicationId>
</deploymentTask>
<deploymentTask>
        <dTId>4b</dTId><deploymentActivity>activation</deploymentActivity>
        <eEId>192.168.0.4</eEId><applicationId>CEPC</applicationId>
</deploymentTask>
<deploymentTask>
        <dTId>5b</dTId><deploymentActivity>activation</deploymentActivity>
        <eEId>192.168.0.5</eEId><applicationId>CEPC</applicationId>
</deploymentTask>
<deploymentTask>
        <dTId>4c</dTId><deploymentActivity>deActivation</deploymentActivity>
        <eEId>192.168.0.4</eEId><applicationId>CEPC</applicationId>
</deploymentTask>
<deploymentTask>
        <dTId>5c</dTId><deploymentActivity>deActivation</deploymentActivity>
        <eEId>192.168.0.5</eEId><applicationId>CEPC</applicationId>
</deploymentTask>
<deploymentTask>
        <dTId>4d</dTId>
        <deploymentActivity>staticUpdate_ImplemX_by_ImplemY_in_AppliId_in_EE</deploymen...>
        <eEId>192.168.0.4</eEId>
        <implemX>http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementation/
        dmsa.applipourlavalidation.aggregationettransformation.r4_1-0.1.0.jar</implemX>
```

```
        <implemY>http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/
        dmsa.applipourlavalidation.aggregationettransformation.r4_1-0.1.1.jar</implemY>
        <applicationId>CEPC</applicationId>
</deploymentTask>
<deploymentTask>
        <dTId>4e</dTId><deploymentActivity>activation</deploymentActivity>
        <eEId>192.168.0.4</eEId><applicationId>CEPC</applicationId>
</deploymentTask>
<deploymentTask>
        <dTId>5d</dTId>
        <deploymentActivity>staticUpdate_ServiceId_by_ImplemY_in_AppliId_in_EE</deployment..>
        <eEId>192.168.0.5</eEId>
        <serviceId>AggregationEtTransformationService</serviceId>
        <implemY>http://212.27.63.139/DMSA_AppliPourLaValidation_ServicesImplementations/
        dmsa.applipourlavalidation.aggregationettransformation.r4_1-0.1.1.jar</implemY>
        <applicationId>CEPC</applicationId>
</deploymentTask>
<deploymentTask>
        <dTId>5e</dTId><deploymentActivity>activation</deploymentActivity>
        <eEId>192.168.0.5</eEId><applicationId>CEPC</applicationId>
</deploymentTask>
<deploymentTask>
        <dTId>5f</dTId><deploymentActivity>deActivation</deploymentActivity>
        <eEId>192.168.0.5</eEId><applicationId>CEPC</applicationId>
</deploymentTask>
<deploymentTask>
        <dTId>4f</dTId><deploymentActivity>deActivation</deploymentActivity>
        <eEId>192.168.0.4</eEId><applicationId>CEPC</applicationId>
</deploymentTask>
<deploymentTask>
        <dTId>4g</dTId><deploymentActivity>deInstallation</deploymentActivity>
        <eEId>192.168.0.4</eEId><applicationId>CEPC</applicationId>
</deploymentTask>
<deploymentTask>
        <dTId>5g</dTId><deploymentActivity>deInstallation</deploymentActivity>
        <eEId>192.168.0.5</eEId><applicationId>CEPC</applicationId>
</deploymentTask>

</deploymentTasks>
```

BIBLIOGRAPHIE

[ABD+07] Ayachitula, N., Buco, M., Diao, Y., Maheswaran, S., Pavuluri, R., Shwartz, L., Ward, C., "IT service management automation - A hybrid methodology to integrate and orchestrate collaborative human centric and automation centric workflows", IEEE International Conference on Services Computing, Juil. 2007, Page(s) : 574 - 581 , Digital Object Identifier : 10.1109/SCC.2007.75.

[ACR08] ACRESSO, "InstallShield - Installation Tool", http://www.acresso.com/products/installation/installshield.htm, 2008.

[AJ74] Aho, A. V., Johnson, S. C., "LR Parsing", ACM Computing Survey, Vol. 6, Num. 2, 1974, Page(s) : 99 - 124.

[AK03] Atkinson, C., Kühne, T., "Model-Driven Development : A Metamodeling Foundation", IEEE Software, Sept. 2003.

[And00] Andersson, J., "A Deployment System for Pervasive Computing", IEEE International Conference on Software Maintenance, Oct. 2000, Page(s) : 262 - 270, Digital Object Identifier : 10.1109/ICSM.2000.883058.

[AOS+99] Arnold, K., O'Sullivan, B., Scheifler, R. W., Waldo, J. and Wollrath, A., "The Jini Specification", Addison-Wesley, Reading, Mass., 1999.

[Ars04] Arsanjani, A., "Service-oriented Modeling and Architecture", http://www.ibm.com/developerworks/library/ws-soa-design1/, Nov. 2004.

[ASF08] Apache Software Foundation, "Apache Geronimo", http://geronimo.apache.org/, Avr. 2008.

[BBB+06] Baude, F., Bottaro, A., Brun, J.-M., Chazalet, A., Constancin, A., Donsez, D., Gurgen, L., Lalanda, P., Legrand, V., Lestideau, V., Marié, S., Marin, C., Moreau, A. and Olive, V., "Extension de passerelles OSGi pour les domaines de la distribution électrique : Modèles et outils", Atelier de travail OSGi, http://hal.archives-ouvertes.fr/hal-00097266/fr, Sept. 2006.

[BBM04] Bézivin, J., Belaunde, M., Marvie, R., "Transformations et modèles platesformes", Arago 30 "Ingénierie Dirigée par les Modèles", OFTA, 2004.

[BC07] Brian Blake, M., Cummings, D., J., "Workflow Composition of Service Level Agreements", IEEE International Conference on Services Computing, Juil. 2007, Page(s) : 138 - 145, Digital Object Identifier : 10.1109/SCC.2007.136.

[BCG+04] Blanc, X., Caron, O., Georgin, A., Muller, A., "Transformations de modèles : d'un modèle abstrait aux modèles EJB et CCM", Langages et modèles à objets (LMO'04), Lille, 2004.

[BCS03] Bruneton, E., Coupaye, T., Stefani, J.,"The Fractal Component Model", Rapport Technique Spécification Version 2, ObjectWeb Consortium, http://www.object.org/fractal, 2003.

[BDC00] Balsters, H., De Brock, B., Conrad, S., "Database Schema Evolution and Meta-Modeling", 9th International Workshop on Foundations of Models and Languages for Data and Objects, FoMLaDO/DEMM 2000, dagstuhl Castle, Allemagne, Sept. 2000.

[BDD+04] Benatallah, B., Dijkman, R., Dumas, M. and Maamar Z., "Service Composition : Concepts, Techniques, Tools and Trends", Service-oriented Software System Engineering, Editor: Stojanovic, Z. and Dahanayake, A., Idea Group Inc., 2004.

[BDE+06] Bourcier, J., Desertot, M., Escoffier, C., Marin, C., Chazalet, A. and Lalanda, P., "A Dynamic-SOA Home Control Gateway", IEEE International Conference on Services Computing, Sept. 2006, Page(s) : 463 - 470, Digital Object Identifier: 10.1109/SCC.2006.5.

[BEA08] BEA, "BEA Aqualogic Service Bus 3.0", http://www.bea.com/content/news_events/white_papers/BEA_AquaLogic_Service_Bus_ds.pdf, Mars 2008.

[BFG01] Berenbrink, P. Friedetzky, T., Goldberg, L. A., "The Natural Work-Stealing Algorithm is Stable", 42nd IEEE Symposium on Foundations of Computer Science, Oct. 2001, Page(s) : 178-189, Digital Object Identifier: 10.1109/SFCS.2001.959892.

[BGM+03] Bézivin, J., Gérard, S., Muller, P.-A., Rioux, L., "MDA Components : Challenges and Opportunities, Metamodelling for MDA", 1st International Workshop, York, Royaume Uni, http://www.cs.york.ac.uk/metamodel4mda/onlineProceedingsFinal.pdf, Nov. 2003.

[BHL+06] Bocchi, L., Hong, Y., Lopes, A., Fiadeiro, J. L., "From BPEL to SRML: A Formal Transformational Approach", International Workshop on Web Services and Formal Methods, WS-FM 2006, Vienna, Austria, Sept. 2006. Page(s) : 92 - 107, ISBN 3-540-38862-1.

[BJP+99] Beugnard, A., Jezequel, J.-M., Plouzeau, N., Watkins, D., "Making Components Contract Aware", IEEE Computer, Juil. 1999, Page(s) : 38 - 45, Digital Object Identifier : 10.1109/2.774917.

[BL06] Bichier, M. and Lin, K.-J., "Service-oriented Computing", IEEE Computer, Vol. 39, Issue 3, Mars 2006, Page(s) : 99 - 101, Digital Object Identifier : 10.11.09/MC.2006.102.

[Bos98] Bosch, J., "Product-Line Architectures in Industry : A Case Study", IEEE International Conference on Software Engineering, Nov. 1998.

[BP01] Bézivin, J., Ploquin, N., "Tooling the MDA framework : a new software maintenance and evolution scheme proposal", Journal of Object-oriented Programming, 2001.

[CC90] Chikofsky, E., Cross, J., "Reverse Engineering and Design Recovery : A Taxonomy", IEEE Software, Jan. 1990.

[CE00] Coupaye, T., Estublier, J., "Foundations of Enterprise Software Deployment", IEEE 4th European Conference on Software Maintenance and Reengineering (CSMR), Mars 2000, Page(s) : 65 - 73, Digital Object Identifier : 10.1109/CSMR.2000.827313.

[CH04] Cervantes, H., Hall, R.S., "Autonomous adaptation to dynamic availability using a service-oriented component model", 26th International Conference on Software Engineering (ICSE), Mai 2004, Page(s): 614 - 623, ISBN: 0-7695-2163-0.

[Che76] Chen, P. P. S., "The Entity-Relationship Model : Toward a Unified View of Data", ACM-TODS, Vol. 1, Num. 1, 1976, Page(s) : 9 - 36.

[Cle02] Clements, P. and Northrop, L., "Software Product Lines: Practices and Patterns", Addison-Wesley Professional, 2002, Page(s) : 564, ISBN 0-201 - 70332 - 7.

[CL07] Chazalet, A. Lalanda, P., "Deployment of Services-oriented Applications Integrating Physical and IT Systems", 21st International Conference on Advanced Information Networking and Applications (AINA), Mai 2007, Page(s) : 38 - 45, Digital Object Identifier: 10.1109/AINA.2007.51

[CL07a] Chazalet, A. Lalanda, P., "A Meta-Model Approach for the Deployment of Services-oriented Applications", IEEE International Conference on Services Computing (SCC), Juil. 2007, Page(s): 348 - 355, Digital Object Identifier : 10.1109/SCC.2007.11.

[CLM05] Cunin, P.-Y., Lestideau, L., Merle, N., "ORYA: A Strategy-oriented Deployment Framework", 3rd International Working Conference on Component Deployment (CD), Nov. 2005, Page(s) : 177 - 180, Digital Object Identifier : 10.1007/11590712_14.

[CNRS88] CNRS, "Le Temps-Réel", TSI - Technique et Science Informatiques, Vol. 7, 1988, Page(s) 493 - 500.

[CTA07] Chituc, C.-M., Toscano, C., Azevedo, A., "Collaborative Business Processes Integration and Management - Lessons learned from industry", IEEE International Conference on Services Computing, Juil. 2007, Page(s) : 451 - 457 , Digital Object Identifier : 10.1109/SCC.2007.41.

[Dan05] Dahan, O., "La Plate-forme .NET", e-Naxos, http://merlin.ftp-developpez.com/cours/delphi/dotnet/DOTNET.pdf , 2005.

[Dea07] Dearle, A., "Software Deployment, Past, Present and Future", IEEE Future of Software Engineering, hosted in the IEEE 29th International Conference on Software Engineering, Mai 2007, Page(s) : 269 - 284, Digital Object Identifier : 10.1109/FOSE.2007.20.

[Des07] Desertot, M., "Une Architecture Flexible et Adaptable pour les Serveurs d'Applications", Thèse de l'Université Joseph Fourier, http://www-adele.imag.fr/Les.Publications/phd_thesis.html, Avr. 2007.

[Don05] Donsez, D., "JMX Agent", http://www-adele.imag.fr/users/Didier.Donsez/dev/osgi/jmxagent/readme.html, Mars 2005.

[Don06] Donsez, D., "Objets, composants et services : intégration de propriétés non fonctionnelles", Habilitation à Diriger des Recherches de l'Université Joseph Fourier, http://www-adele.imag.fr/users/Didier.Donsez/pub/publi/hdr/, Déc. 2006.

[Don06a] Donsez. D., "Projet de Programmation Java Embarqué : Tutorial JMX", Laboratoire LIG, équipe Adèle, http://www-adele.imag.fr/users/Didier.Donsez/ujf/pjem/sujet/jmx.html, 2006.

[DRD99] Ducasse, S., Rieger, M., Demeyer, S., "A Language Independent Approach for Detecting Duplicated Code", IEEE International Conference on Software Maintenance (ICSM), Sept. 1999, Page(s) : 109 - 118.

[EBL+08] Escoffier, C., Bourcier, J., Lalanda, P., Yu, J., "Towards a Home Application Server", IEEE Consumer Communications and Networking Conference, Jan. 2008, Page(s) : 321-325, Digital Object Identifier: 10.1109/ccnc08.2007.78.

[EHL07] Escoffier, C., Hall, R.S., Lalanda, P., "iPOJO: an Extensible Service-oriented Component Framework", IEEE International Conference on Services Computing, Juil. 2007, Page(s): 474 - 481, Digital Object Identifier: 10.1109/SCC.2007.74.

[Ecl08] The Eclipse Foundation, "Eclipse - An Open Development Platform", http://www.eclipse.org/, 2008.

[EVI05] Estublier, J., Vega, G., Ionita, A., "Composing Domain-Specific Languages for Wide-Scope Software Engineering Applications", International Conference on Model Driven Engineering Languages and Systems, MODEL, 2005.

[Fav96] Favre, J.-M., "Maintenance et ré-ingénierie global des logiciels", Thèse de doctorat, Université de Grenoble, 1996.

[Fav04] Favre, J.-M., "Toward a Basic Theory to Model Model Driven Engineering", 3rd Workshop in Software Model Engineering, WiSME 2004, http://www-adele.imag.fr/~jmfavre.

[Fav04a] Favre, J.-M., "From Ancient Egypt to Model Driven Engineering", 2004, http://planetmde.org/fae2mde.

[FBP+03] Blay-Fornarino, M., Boudaoud, K., Pinna-Dery, A.-M., Mc Cathie Nevile, C., "A Flexible Approach to Semi-Automatic Accessibility Evaluation", IADIS International Conference WWW/Internet, IASIS, Algarve, Portugal, Nov. 2003, Page(s) : 1083 - 1088.

[FEF06] Favre, J.-M., Estublier, J., Blay-Fornarino, M., "L'Ingénierie Dirigée par les Modèles : Au-delà du MDA", Edition Hermes-Sciences Lavoisier, Informatique et Systèmes d'Information, IC2 : Information - Commande - Communication, Jan. 2006, ISBN : 2-7462-1213-7.

[Gen01] General Dynamics C4 Systems, "OpenWings Architecture White Paper", http://www.openwings.org/download/specs/openwingswp.pdf, Juin 2001.

[GMT+04] Gérard, S., Mraidha, C., Terrier, F., Baudry, B., ""A UML-based concept for high concurrency : The real-time object", 7th IEEE Symposium on Object-oriented Real-time distributed Computing (ISORC'2004), Vienne, Autriche, Mai 2004, Page(s) : 64 - 67.

[Hal07] Hall, R. S., "The Bundle Dilemma", http://www.osgi.org/wiki/uploads/Conference/OBR-OSGi-Community-RHall.pdf, Juin 2007.

[HHC+98] van der Hoek, A., Hall, R. S., Carzaniga, A., Heimbigner, D. and Wolf, A. L., "Software Deployment: Extending Configuration Management Support into the Field", Crosstalk, The Journal of Defense Software Engineering, vol. 11, number 2, Fév. 1998.

[HHW99] Hall, R. S., Heimbigner, D., Wolf, A. L., "A cooperative approach to support software deployment using the Software Dock", IEEE 21th International Conference of Software Engineering, Mai 1999, Page(s) : 174 - 183, Digital Object Identifier : 10.1109/ICSE.1999.841007.

[HS05] Huhns, M., N., Singh, M., P., "Service-oriented Computing: Key Concept and Principles", IEEE Internet Computing, Vol. 9, Issue 1, Jan.-Féb. 2005, Page(s) : 75 - 81, Digital Object Identifier : 10.1109/MIC.2005.21.

[Hud98] Hudak, P., "Modular Domain Specific Languages and Tools", 5th IEEE International Conference on Software Reuse, Juin. 1998, Page(s) : 134 - 142.

[IBM00] IBM, "Web Services Architecture Overview", http://www.ibm.com/developerworks/library/w-ovr/, Sept. 2000.

[IBM01] IBM, "CORBA Component Model, Introducing to next-generation CORBA", "http://www.ibm.com/developerworks/webservices/library/co-cjct6/index.html#h1, Avr. 2001.

[IBM05] IBM, "WebSphere Process Server", http://www.ibm.com/developerworks/websphere/library/techarticles/0509_kulhanek/0509_kulhanek.html, Sept 2005.

[IBM07] IBM, "Business Process Execution Language for Web Services version 1.1", http://www.ibm.com/developerworks/library/specification/ws-bpel/, Fév. 2007.

[IBM08] IBM, "Success Stories for Service-oriented Architecture", http://www-01.ibm.com/software/success/cssdb.nsf/CategoryL1ViewFM?ReadForm&Site=soa_industryL1VW, 2008.

[IBM08a] IBM, "WebSphere Software", http://www-306.ibm.com/software/websphere/, 2008.

[IEEE90] Radatz, J.and al., "IEEE Standard Glossary of Software Engineering Terminology", IEEE, Déc. 1990, ISBN : 1-55937-067-X, http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=159342&isnumber=4148&punumber=2238&k2dockey=159342@ieeestds&query=%28%28standard+glossary%29%3Cin%3Emetadata%29&pos=0&access=no.

[INRIA05] INRIA, "Model Transformation at INRIA", http://modelware.inria.fr, 2005.

[JINI05] Sun Microsystems, "Jini Specifications and API Archive version 2.1", http://www.sun.com/software/jini/specs/, Oct. 2005.

[JINI07] Sun Microsystems, "Jini Technology", http://www.jini.org/wiki/Main_Page, Mars 2007.

[Jon08] Java Open Application Server (JOnAS), http://wiki.jonas.objectweb.org/xwiki/bin/view/Main/JOnAS5, 2008.

[JS05] Jammes, F. and Smit, H., "Service-oriented Paradigms in Industrial Automation", IEEE Transactions on Industrial Informatics, Vol. 1, Issue 1, Fév. 2005, Page(s) : 62 - 70, Digital Object Identifier: 10.1109/TII.2005.844419.

[JW03] Jeronimo, M. and Weast, J., "UPnP Design by Example : A Software Developer's Guide to Universal Plug and Play", Intel Press, http://www.intel.com/intelpress/excerpts/upnp1.htm, ISBN-10 : 0971786119, Mai 2003.

[KC03] Kephart, J. O., Chess, D. M., "The Vision of Autonomic Computing", IEEE Computer, Vol. 36, Issue 1, Jan. 2003, Page(s) : 41 - 50, Digital Object Identifier : 10.1109/MC.2003.1160055.

[KWB03] Kleppe, A., Warmer, S., Bast, W., "MDA Explained. The Model Driven Architecture : Practice and Promise", Addison-Wesley, Avr. 2003.

[Lan05] Lalanda, P., "An E-Services Infrastructure in Power Distribution", IEEE Internet Computing, Vol. 9, Issue 3, Mai-Juin 2005, Page(s) : 52 - 59, Digital Object Identifier: 10.1109/MIC.2005.50.

[Les03] Lestideau, V., "Modèles et environnement pour configurer et déployer des systèmes logiciels.", Thèse de l'Université Joseph Fourier - Grenoble 1, Déc. 2003.

[LSJ00] Le Guennec, A., Sunyé, G., Jézéquel, J.-M., "Precise Modeling of Design Patterns", UML Conference, Springer-Verlag, Oct. 2000, Page(s) : 482 - 496.

[LYL07] Luo, N., Yan, J., Liu, M., "Towards Efficient Verification for Process Composition of Semantic Web Services", IEEE International Conference on Services Computing, Juil. 2007, Page(s) : 220 - 227, Digital Object Identifier : 10.1109/SCC.2007.120.

[MAV08] Apache Software Foundation, the MAVEN project, http://maven.apache.org/, 2008.

[MB02] Mellor, S. J., Balcer, M. j., "Executable UML : A Foundation for Model-Driven Architecture", Addison-Wesley, Boston (MA), 2002.

[Mer05] Merle, N., "Déploiement à grande échelle, entreprise, stratégies de déploiement, méta-modèle de déploiement, ORYA.", Thèse de l'Université Joseph Fourier - Grenoble 1, Déc. 2005.

[Mey99] Meyer, B., "A really good idea (object-oriented software development)", IEEE Computer Volume 32, Issue 12, Déc. 1999, Page(s) : 144 - 147, Digital Object Identifier : 10.1109/2.809257.

[Mic08] Microsoft, "About the WebService Behavior", http://msdn2.microsoft.com/en-us/library/ms531032.aspx, 2008.

[Mic08a] Microsoft, "Méthode de Localisation des Assemblies par le Runtime (.Net Framework 3.0)", http://msdn.microsoft.com/fr-fr/library/yx7xezcf(VS.85).aspx, 2008.

[ML07] Marin, C., Lalanda, P., "DoCoSOC- Domain Configurable Service-oriented Computing", IEEE International Conference on Services Computing, Juil. 2007, Page(s) : 52 - 59 , Digital Object Identifier : 10.1109/SCC.2007.51.

[MSU+04] Mellor, S. J., Scott, K., Uhl, A., Weise, D., "MDA Distilled : Principales of Model-Driven Architecture", Addison-Wesley, Mar. 2004.

[MT00] Medvidovic, N. and Taylor, R. N., "A Classification and Comparison Framework for Software Architecture Description Languages", IEEE Transactions on Software Engineering, Volume 26, Issue 1, Jan 2000, Page(s) : 70 - 93, Digital Object Identifier : 10.1109/32.825767.

[NB07] Nowlan, M., F., Blake, M., B., "Agent-Mediated Knowledge Sharing for Services Management", IEEE International Conference on Services Computing, Juil. 2007, Page(s) : 324 - 331 , Digital Object Identifier : 10.1109/SCC.2007.23.

[NE00] Nuseibeh, B., Easterbrook, S., "Requirements Engineering : A Roadmap", Future of Software Engineering (joint event of ICSE'00), Limerick, Irelande, Juin 2000, Page(s) : 35 - 46.

[.Net06] Microsoft, ".Net Framework 3.0", http://msdn.microsoft.com/en-us/library/aa338209.aspx , Nov. 2006.

[.Net07] Microsoft, "Deploying .NET Applications Lifecycle Guide", Microsoft Patterns & Pratices, Déc. 2007, http://www.microsoft.com/Downloads/details.aspx?FamilyID=5b7c6e2d-d03f-4b19-9025-6b87e6ae0da6&displaylang=en et http://support.microsoft.com/kb/913507.

[Obe04] Öberg, R., "", Présentation JMX, Entreprise JBoss, http://www.ece.uic.edu/%7Ecpress/jmx/Oberg_jmx_jboss.ppt, Avril 2004.

[OMG01] Object Management Group, "Model-Driven Architecture : A Technical Perspective", Fév. 2001.

[OMG03] The Object Management Group, "the MDA Guide Version 1.0.1", 2003, http://www.omg.org/docs/omg/03-06-01.pdf.

[OMG06] The Object Management Group, "Deployment and Configuration of Component-based Distributed Applications Specification, Version 4.0", http://www.omg.org/technology/documents/formal/deployment.htm, Avr. 2006.

[OMG06a] The Object Management Group, "CORBA Component Model Specification Version 4.0", http://www.omg.org/technology/documents/formal/components.htm, Avr. 2006.

[OMG08] The Object Management Group, http://www.omg.org/, 1997 - 2008.

[Ora07] The Oracle Corporation, "Oracle Application Server 10g", http://www.oracle.com/technology/products/ias/index.html, 2007.

[OSGi07] Alliance OSGi, "OSGi Service Platform Release 4.1 : Core Specification", http://www.osgi.org/Specifications/HomePage, Mai 2007.

[OSGi07a] Alliance OSGi, "OSGi Service Platform Release 4.1 : Compendium", http://www.osgi.org/Specifications/HomePage, Mai 2007.

[OSGi07b] Alliance OSGi, "OSGi Service Platform Release 4.1 : Compendium", Chapitre 111 : "UPnP Device Service Specification", http://www.osgi.org/Specifications/HomePage, Mai 2007.

[OSGi07c] Alliance OSGi, "OSGi Service Platform Release 4.1 : Compendium", Chapitre 103 : "Device Access Specification", http://www.osgi.org/Specifications/HomePage, Mai 2007.

[OSGi07d] Alliance OSGi, "OSGi Service Platform Release 4.1 : Core Specification", Section 5.4 : "Stale References", Page : 116, http://www.osgi.org/Specifications/HomePage, Mai 2007.

[OSGi07e] Alliance OSGi, "OSGi Service Platform Release 4.1 : Core Specification", Chapitre 8 : "Start Level Service Specification", Page(s) : 203 - 214, http://www.osgi.org/Specifications/HomePage, Mai 2007.

[OSGi07f] Alliance OSGi, "OSGi Service Platform Release 4.1 : Compendium", Chapitre 112 : "Declarative Services Specification", Page(s) : 281 - 316 ,http://www.osgi.org/Specifications/HomePage, Mai 2007.

[OSGi07g] Alliance OSGi, "OSGi Service Platform Release 4.1 : Compendium", Chapitre 701 : "Service Tracker Specification", Page(s) : 601 - 612, http://www.osgi.org/Specifications/HomePage, Mai 2007.

[OSOA08] Open Service-oriented Architecture, http://www.osoa.org/display/Main/Home, 2008.

[OW208] Object Web 2 Consortium, "EasyBeans", http://www.easybeans.org, Mar. 2008.

[PAB07] Paliwal, A. V., Adam, N. R., Bornhövd, C. "Web Service Discovery: Adding Semantics through Service Request Expansion and Latent Semantic Indexing", IEEE International Conference on Services Computing, Juil. 2007, Page(s): 106-113, Digital Object Identifier 10.1109/SCC.2007.131.

[Pap03] Papazoglou, M., P., "Service-oriented Computing : Concepts, Characteristics and directions", IEEE 4th Web Information Systems Engineering, Déc. 2003, Page(s) : 3 - 12, Digital Object Identifier : 10.1109/WISE.2003.1254461.

[PTD+06] Papazoglou, M. P., Traverso, P., Dustdar, S. and Leymann, F., "Service-oriented Computing Research Roadmap", Technical Report/vision paper on Service-oriented computing, European Union Information Society Technologies (IST), Directorate D, 2006, ftp://ftp.cordis.europa.eu/pub/ist/docs/directorate_d/st-ds/services-research-roadmap_en.pdf.

[PTD+07] Papazoglou, M. P., Traverso, P., Dustdar, S. and Leymann, F., "Service-oriented Computing: State of the Art and Research Challenges", IEEE Computer, Volume 40, Issue 11, Nov. 2007, PAGE(s) 38 - 45, Digital Object Identifier 10.1109/MC.2007.400.

[Red07] Red Hat, "JBoss, Enterprise Middleware", http://www.redhat.com/jboss/, Oct. 2007.

[RIV08] Apache Software Foundation, "The RIVER Project", http://incubator.apache.org/projects/river.html, Jan. 2008.

[SCA07] OSOA, "The SCA Project: Service Component Architecture Home", http://www.osoa.org/display/Main/Service+Component+Architecture+Home, Nov. 2007

[SCA07a] SCA, "SCA : Java Common Annotations and APIs", (section : "1.4.1.1. Using the Component Context API", page 4, ligne 342), http://www.osoa.org/download/attachments/35/SCA_JavaAnnotationsAndAPIs_V100.pdf, Mars 2007.

[SDE04] Skene, J., Davide Lamanna, D., Emmerich, W., "Precise Service Level Agreements", IEEE 26th International Conference on Software Engineering, Mai 2004, Page(s) : 179 - 188, ISBN : 0-7695-2163-0.

[Sei03] Seidewitz, E., "What Models Mean", IEEE Software, Vol. 20, Num. 5, Sept. 2003, Page(s) : 26 - 32.

[Sen05] SENSORIA-IST, "SENSORIA : Software Engineering for Service-oriented Overlay Computers", IST (Information Society Technologies), http://www.sensoria-ist.eu/, Sept 2005 - Aout 2009.

[Sta88] Stankovic, J. A., "Misconceptions about real-time : a serious problem for the next generation systems", IEEE Computer, Vol. 21, 1988, Page(s) 10 - 19.

[SUN99] Sun Microsystems, "Enterprise JavaBeans, 1.1 - Final Release", http://java.sun.com/products/ejb/docs.html, Déc. 1999.

[SUN03] Sun Microsystems, "Enterprise JavaBeans, 2.1 - Final Release", http://java.sun.com/products/ejb/docs.html, Nov. 2003.

[SUN03a] Sun Microsystems, "The Java Tutorials, Lesson: Packaging Programs in JAR Files", http://java.sun.com/docs/books/tutorial/deployment/jar/, 2003.

[SUN03b] Sun Microsystems, "JSR 153: Enterprise JavaBeans 2.1", http://jcp.org/en/jsr/detail?id=153, Nov. 2003.

[SUN06] Sun Microsystems, "JSR-000220, Enterprise JavaBeans 3.0 (Final Release)", http://jcp.org/aboutJava/communityprocess/final/jsr220/, Mai 2006.

[SUN06a] Sun Microsystems, "JSR-000244, Java Platform Enterprise Edition 5 Specification (Final Release", http://jcp.org/aboutJava/communityprocess/final/jsr244/, Mai 2006.

[SUN07] Sun Microsystems, "EntityManager (Java EE 5)", http://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#find(java.lang.Class, java.lang.Object), 2007.

[SUN08] Sun MicroSystems, "Java Management eXtensions (JMX) Technology", http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/, 2008.

[Szy03] Szyperski, C., "Component Technology - What, Where, and How ?", IEEE 25th International Conference on Software Engineering, Mai 2003, Page(s) : 684 - 693, Digital Object Identifier : 0-7695-1877-X/03.

[TBB03] Turner, M., Budgen, D., Brereton, P., "Turning Software into a Service", IEEE Computer, Vol. 36, Issue 10, Oct. 2003, Page(s) : 38 - 44, Digital Object Identifier : 10.11.09/MC.2003.1236470.

[Tem04] Templin, D., "How ClickOnce Manifest Generation works with MSBuild", http://windowsclient.net/articles//clickonce_msbuild.aspx, (à la fin de la page), Juin 2004.

[TGM03] Tessier, P., Gérard, S., Mraidha, C., Terrier, F., Geib, J.-M., "", 14th IEEE International Workshop on Rapid System Prototyping (RSP'03), San Diego, USA, Juin 2003, Page(s) : 9 - 15.

[TUS08] Apache Software Foundation, "The Apache Tuscany Project", http://incubator.apache.org/tuscany/, 2008.

[Ull89] Ullman, J. D., "Principles of Database Systems", Vol. 1 et 2, Computer Science Press, Rockville (MD), 1989.

[UPNP99] Universal Plug and Play Forum, http://www.upnp.org/, 1999.

[UPNP00] UPnP Forum, "Understanding UPnP™ : A White Paper", http://www.upnp.org/resources/whitepapers.asp, Juin 2000.

[UPNP07] UPnP Forum, "UPnP Resources", http://www.upnp.org/resources/default.asp, Sept. 2007.

[W3C02] W3C, "Web Services Activity Statement", http://www.w3.org/2002/ws/Activity.html, 2002.

[W3C05] W3C, "Web Services Choreography Description Language Version 1.0", http://www.w3.org/TR/ws-cdl-10/, Nov. 2005.

[W3C07] W3C, "Web Services Description Language (WSDL) Version 2.0", http://www.w3.org/TR/wsdl20/, Juin 2007.

[WVD+02] Wohed, P., Van der Aalst, W. M. P., Dumas, M. Ter Hofstede, A. H. M., "Pattern Based Analysis of BPEL4WS", Technical Report FIT-TR-2002-04, Queensland University of Technology, http://xml.coverpages.org/AalstBPEL4WS.pdf, Avr. 2002.

[ZMS07] Zisman, A., Mahbub, K., Spanoudakis, G., "A Service Discovery Framework based on Linear Composition", IEEE International Conference on Services Computing, Juil. 2007, Page(s) : 536 - 543 , Digital Object Identifier : 10.1109/SCC.2007.15.

[ZYC+07] Zhou, G., Yu, J., Chen, R., Zhang, H., "Scalable Web Service Discovery on P2P Overlay Network", IEEE International Conference on Services Computing, Juil. 2007, Page(s) : 122 - 129, Digital Object Identifier 10.1109/SCC.2007.95.