Apports des architectures à composants pour le déploiement d'applications à la juste taille

Loris Bouzonnet

▶ **To cite this version:**

Loris Bouzonnet. Apports des architectures à composants pour le déploiement d'applications à la juste taille. Autre [cs.OH]. Université de Grenoble, 2011. Français. NNT : 2011GRENM034 . tel-00629760

HAL Id: tel-00629760

https://theses.hal.science/tel-00629760

Submitted on 6 Oct 2011

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

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

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Loris BOUZONNET

Thèse dirigée par **Jean-Bernard STEFANI**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de **L'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Apport des architectures à composants pour le déploiement d'applications à la juste taille

Thèse soutenue publiquement le **16 septembre 2011**,
devant le jury composé de :

Mme Françoise BAUDE
Professeur à l'Université de Nice Sophia-Antipolis, Rapporteur
M. Lionel SEINTURIER
Professeur à l'Université Lille 1, Rapporteur
M. Thomas LEDOUX
Maître de conférence à l'École des Mines de Nantes, Examinateur
M. Noël DE PALMA
Professeur à l'UJF, Examinateur
M. François EXERTIER
Responsable de l'équipe JOnAS/JASMINe à Bull S.A.S., Examinateur
M. Jean-Bernard STEFANI
Directeur de recherche à l'INRIA, Directeur de thèse

Résumé

L'informatique dans les nuages propose une alternative économique et performante au déploiement traditionnel sur site. Une utilisation efficace des nuages passe par une diminution des ressources consommées et une adaptation du déploiement à l'environnement cible. Nous proposons, dans cette thèse, une solution pour le déploiement de logiciels à la juste taille, c'est-à-dire en ne déployant que les dépendances nécessaires sur un environnement cible. Nous suivons une définition du déploiement basée sur les architectures à composants. Le contenu et les dépendances d'un logiciel sont capturés grâce au modèle à composants *Fractal SoftwareUnit*. Ce modèle offre une fine granularité de la représentation, le support de l'hétérogénéité des logiciels et des environnement cibles, ainsi qu'un contrôle distribué des logiciels. Nous proposons une mise en œuvre du modèle au travers du canevas *SU Framework*. Enfin, nous décrivons comment résoudre deux problèmes issus de besoins industriels, à l'aide de ce canevas : la définition de profils pour le serveur d'applications *JOnAS* et le déploiement d'applications hétérogènes sur cibles hétérogènes.

Mot-clés : architectures à composants, logiciels patrimoniaux, déploiement, modularité, hétérogénéité, service, modèle à composants *Fractal*, serveur d'applications *Java EE*

Abstract

Cloud computing offers an economical and efficient alternative to traditional deployment on site. Effective use of cloud computing goes through a reduction of resource consumption and deployment tailored to the target environment. We propose, in this thesis, a solution for deploying softwares at fair size, by considering only the necessary dependencies to a target environment. We follow a definition of deployment based on component architecture. The content and dependencies of softwares are captured through the *Fractal SoftwareUnit* component model. This model offers a fine granularity of representation, the support of software heterogeneity and environmental targets, just as a distributed control of software. *SU Framework* is an implementation of this model. Finally, we describe how to solve two problems arising from industrial needs, by using the proposed framework : définition of profiles for the *JOnAS* application server and deployment of heterogeneous applications on heterogeneous targets.

Keywords: component architectures, legacy softwares, deployment, modularity, heterogeneity, service, *Fractal* component model, *Java EE* application servers

Remerciements

Ces trois années de thèse m'ont été particulièrement formatrices et enrichissantes tant sur le plan scientifique que personnel. Je remercie donc vivement Jean-Bernard Stefani et François Exertier d'avoir assuré la direction de cette thèse, dans ces deux univers que sont la recherche académique et l'industrie, qui ont tant de choses à partager. Je remercie également Noël De Palma pour son soutien aux moments opportuns.

Je suis honoré que Françoise Baude et Lionel Seinturier ait accepté de rapporter cette thèse, ainsi que Thomas Ledoux ait accepté d'examiner celle-ci.

Je suis sincèrement reconnaissant envers Benoit Pelletier, pour son implication et ses idées toujours lumineuses, tout au long de cette aventure.

Je suis gré à l'équipe *JOnAS* pour son accueil, sa disponibilité, ainsi que le bonheur qu'elle m'a apportée. Adriana, Benoit, Florent, François, Guillaume, Hélène, Jacques, Julien, Philippe, vous êtes géniaux et passionnants !

Au cours de cette thèse, j'ai découvert au sein de l'équipe *Sardes* un vivier de connaissances, des gens formidables et de véritables amis.

J'embrasse affectueusement Yolande, Julien, mes parents, ainsi que toute ma famille.

Enfin, je pense à tous mes Ardéchois au cœur fidèle dont l'amitié m'est chère.

Table des matières

Table des figures

Liste des tableaux

Listings

« Nous parlons le même langage
Et le même chant nous lie
Une cage est une cage
En France comme au Chili. »

La complainte de Pablo Neruda
Jean Ferrat (1930-2010)

Chapitre 1

Introduction

Sommaire

1.1 Contexte

Le déploiement d'applications hétérogènes à la juste taille est un défi aux multiples enjeux pour une utilisation efficace de l'informatique dans les nuages (a.k.a. *cloud computing* [5, 64, 65]). Une application est dite à la juste taille si celle-ci n'est composée que des fonctionnalités attendues. Déployer une application à la juste taille permet en outre de minimiser le nombre d'étapes du processus de déploiement, en ne sélectionnant que celles réellement nécessaires. Le défi du déploiement d'applications hétérogènes à la juste taille peut être lié à des enjeux à la fois économiques, écologiques et de qualité de service. L'économie du *cloud computing* est ainsi souvent basée sur la tarification des ressources informatiques consommées, l'écologie cherchant à réduire les consommations de ces ressources pour des raisons plus altruistes. La qualité de service nécessite l'ajustement des capacités informatiques à la demande. Cet ajustement repose sur l'élasticité de l'infrastructure sous-jacente mais également sur le temps de démarrage des ressources allouées. L'intérêt d'une application à la juste taille est d'offrir un temps de démarrage moindre, permettant une meilleure réactivité dans l'allocation de ressources. Le déploiement à la juste taille aide non seulement à minimiser les consommations en ressources mais également à accélérer les reconfigurations.

Le défi se présente en terme de complexité, le déploiement étant susceptible d'impliquer de nombreuses briques logicielles hétérogènes. La problématique de gestion des dépendances se présente notamment à la fabrication d'une *appliance*, un assemblage fonctionnel d'un logiciel avec toutes ses dépendances (incluant le système d'exploitation

et les intergiciels) fréquemment requis par les fournisseurs d'infrastructure. Cette problématique se pose également aux fournisseurs de plate-forme. Pour qu'une plate-forme soit attractive, celle-ci doit laisser une certaine liberté aux développeurs l'utilisant. Cette liberté passe notamment par la possibilité d'utiliser des librairies du système d'exploitation. Laisser une telle liberté aux utilisateurs impose de la flexibilité quant à la gestion des dépendances des applications supportées.

Le défi se pose également sur le terrain de l'interopérabilité, l'absence de standard entre les différents fournisseurs de plate-forme contraignant le développement sur les plate-formes *PaaS*. Pour être pérenne et attractive, une solution de déploiement de logiciels à la juste taille doit supporter l'hétérogénéité des plate-formes.

Enfin, le défi concerne l'adaptabilité des applications, notamment en terme de configuration. Une solution de déploiement doit intégrer les évolutions que peut subir le logiciel après un déploiement initial. Ainsi, dans le contexte du *cloud computing*, la gestion autonome des ressources, par exemple à des fins de mise à l'échelle de l'infrastructure (*scale up*), nécessite fréquemment une mise à jour des configurations des applications déployées (eg. celle des répartiteurs de charge). Une application doit donc être décrite finement au risque de construire des silos dans le nuage. Par silos, nous entendons des machines virtuelles cloisonnées, n'ayant aucune interaction entre elles. Or, l'élasticité peut impliquer des lectures (eg. copie) et écritures (eg. reconfiguration) des applications actives, ce qui nécessite d'avoir une connaissance précise de celles-ci, mais également d'avoir accès à des interfaces d'administration des applications.

Parallèlement, les intergiciels adaptables rencontrent de plus en plus de succès comme en témoigne l'adoption massive d'*OSGi*[1] comme plate-forme d'exécution, par les principaux serveurs d'applications *Java EE*[2]. Les serveurs d'applications ne sont ainsi plus monolithiques et peuvent être adaptés aux applications déployées. La spécification *Java EE 7* tiendra compte de ce contexte en facilitant le déploiement d'applications et d'infrastructures sur le nuage.

Cette thèse explique comment l'utilisation d'intergiciels adaptables peut contribuer au déploiement de logiciels à la juste taille.

1.2 Problématique

Le déploiement d'un logiciel peut-être défini comme le processus allant de l'acquisition du logiciel à son exécution [50]. Le déploiement est alors vu comme une activité effectuée pour ou par le client, après la production du logiciel. D'autres définitions du processus de déploiement, comme celles proposées par *Characterization Framework* [16] (présenté dans la section 2.1) ou T. Coupaye et J. Estublier [19], ajoutent l'étape de sortie du logiciel et intègrent le site de production.

Le déploiement logiciel constitue une étape des cycles de vie logiciel, qu'ils soient incrémentaux (par ex. *Scrum* [54]) ou non (par ex. cascade ou spirale [7]).

1. OSGi Alliance : `http://www.osgi.org`
2. Oracle Java EE : `http://www.oracle.com/technetwork/java/javaee/overview/index.html`

Enfin, R. S. Hall [37] propose un support du déploiement dans des canevas extensibles, mis en œuvre dans le projet *Oscar*, implémentant la spécification *OSGi*.

Dès la sortie du logiciel, le déploiement d'applications doit adresser un certain nombre de difficultés [22].

1.2.1 Le paradoxe de la modularité

Un module logiciel est une unité de logiciel sans-état, déployable, administrable, nativement réutilisable et composable [43]. Un module est associé à une représentation physique du logiciel, mais peut également être représenté à l'exécution (eg. *OSGi*). La modularité est la capacité à décomposer un logiciel sous forme de modules. Si la modularité des logiciels augmente leur réutilisabilité en fournissant une plus grande maintenabilité et évolutivité, avec notamment une flexibilité accrue dans l'assemblage, celle-ci peut nuire à son utilisabilité (voir figure 1.1).

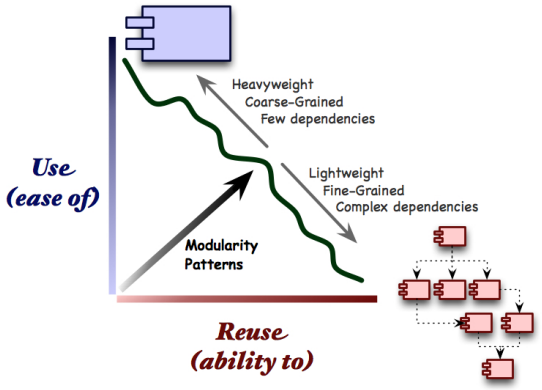


FIGURE 1.1 – Le paradoxe de la modularité

Dans un article discutant du rôle de l'architecte [33], *Martin Fowler* rapporte les propos suivants de *Ralph Johnson* :

> Making something easy to change makes the overall system a little more complex, and making everything easy to change makes the entire system very complex.

Nous verrons dans cette thèse comme rendre la gestion de la modularité plus simple, pour un projet fortement modulaire.

1.2.2 Hétérogénéité des briques logicielles

Les applications déployées selon une architectures multi-niveaux (correspondant au modèle *Layers* dans [15]) et multi-tiers peuvent comporter des dépendances entre les différents niveaux. Par exemple, la figure 1.2 décrit une application *Java EE* exécutée sur plusieurs niveaux et tiers.

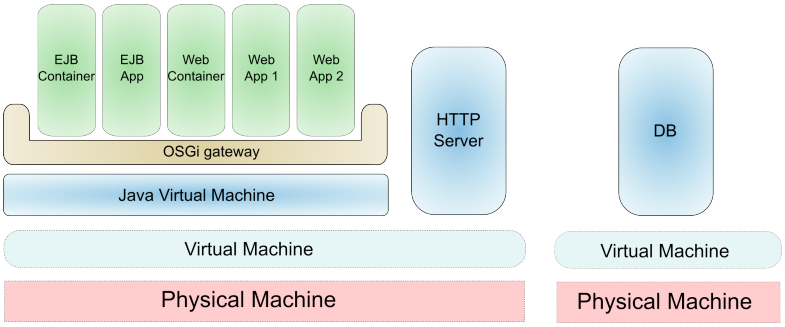


FIGURE 1.2 – Une architecture Java EE multi-niveaux, multi-tiers

Dans cet exemple, chaque niveau utilise une technologie de logiciel différente. Il illustre une application tournant sur une pile d'intergiciels, celle-ci étant composée de conteneurs *EJB* et *Web* déployée sur une plate-forme *OSGi*, elle-même exécutée sur une machine virtuelle *Java*[3].

Si, au sein d'un même niveau, les dépendances sont faciles à exprimer, celles mettant en jeu plusieurs niveaux sont le plus souvent non déclarées. Par exemple, les technologies *Java* et *OSGi* ne permettent pas de spécifier une dépendance sur une librairie du système d'exploitation. L'expression des dépendances ne peut être effectuée qu'entre modules de même technologie et ne peut être modifiée au cours du cycle de vie du logiciel. Or, l'instrumentation d'un logiciel peut lui créer des dépendances nouvelles.

1.2.3 Hétérogénéité des cibles du déploiement

La définition de plans de déploiement devrait être également indépendante de l'environnement de la cible de déploiement, s'adaptant à celui-ci. Par exemple, la figure 1.3 illustre l'hétérogénéité des environnements cibles pour le déploiement d'un serveur d'applications *JOnAS* connecté à une base de données *MySQL*.

Dans cet exemple, le système d'exploitation est inconnu au moment de la création de l'assemblage de l'application. Pourtant, selon le système de gestion de paquets utilisé, le processus de déploiement diffère. Si la cible exécute le système d'exploitation *GNU/Linux Fedora*, le système de gestion de paquets *rpm* devra être utilisé pour l'installation de la machine virtuelle *Java* ou de la base de données. Si la cible exécute le système d'exploitation *GNU/Linux Ubuntu*, le système de gestion de paquets *dpkg* sera préféré.

1.2.4 Hétérogénéité des outils de déploiement

Le déploiement du logiciel est marqué par de nombreuses étapes, parmi lesquelles l'assemblage, l'installation, l'exécution, la mise à jour ou la désinstallation. Ces étapes sont généralement confiées à des outils différents, chacun étant spécialiste dans son domaine, ou alors effectuées manuellement. Nous souhaitons simplifier l'intégration de tels

3. Oracle Java SE : `http://www.oracle.com/technetwork/java/javase/overview/index.html`

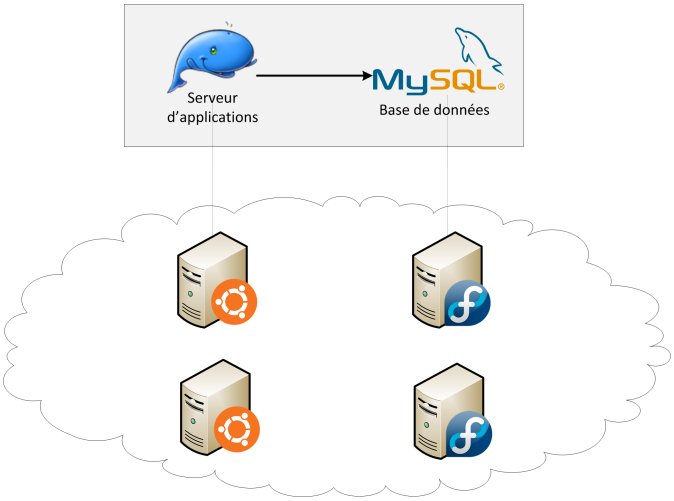


FIGURE 1.3 – Déploiement dans le nuage

outils en :

– Les encapsulant dans des abstractions ;
– Leur faisant partager une représentation commune du système.

Nous souhaiterions également avoir un outil commun pour la gestion des dépendances, non limité à un système de gestion de modules.

L'écriture de plan de déploiement se ferait alors à partir d'un langage unique.

1.2.5 Contrôle de la cohérence du déploiement

Une cible de déploiement peut être sujette à un gestionnaire autonome de reconfigurations. Par exemple, chaque plate-forme *OSGi* exécute un système autonome de résolution des dépendances et du cycle de vie. La définition d'un processus de déploiement, sur une telle plate-forme, doit donc tenir compte de celui-ci, afin de maintenir à la fois la cohérence de la plate-forme et le respect du processus défini.

1.2.6 Étude de cas : le serveur d'applications *OW2 JOnAS*

Afin d'illustrer cette problématique, considérons le cas d'un serveur d'applications *Java EE*.

1.2.6.1 Présentation de *JOnAS*

OW2 JOnAS[4] est un serveur d'applications open-source, certifié *Java EE 5* à partir de la version 5.1, et exécuté sur une plate-forme OSGi [24, 26]. *JOnAS 5* est composé d'environ 200 *bundles OSGi*, dont la majorité sont optionnels. Les services techniques

4. OW2 JOnAS : `http://jonas.ow2.org`

du serveur (eg. *EJB*, *Web Services*) sont implémentés sous forme de services *OSGi*. L'architecture globale du serveur est donnée par la figure 1.4.

FIGURE 1.4 – Architecture globale de *JOnAS 5*

La gestion du dynamisme de ceux-ci est déléguée à *iPOJO*[5] [30, 31].

1.2.6.2 Construction des distributions de *JOnAS*

JOnAS est un serveur d'applications modulaire pouvant, en théorie, être construit à la carte parmi des services techniques au choix. Cependant, en pratique, seulement deux distributions de *JOnAS* sont proposées aux utilisateurs : *Micro JOnAS* et *JOnAS complet*.

La distribution *Micro JOnAS* est l'assemblage minimal, ne proposant que la plate-forme d'exécution. La plate-forme d'exécution comporte entre autres la passerelle *OSGi Apache Felix*[6], des API *Java EE*, un système de déploiements, un registre *RMI* et un agent *JMX*. Cette version est proposée avec une taille d'environ 10 MO pour la version 5.2.

La distribution complète de *JOnAS* fournit une implémentation de la spécification *Java EE 5*, comprenant ainsi une vingtaine de services techniques, le tout pour une taille d'environ 200 MO.

La construction des distributions est actuellement réalisée avec l'outil de gestion de projet *Apache Maven*[7], via le plugin *maven-assembly-plugin*. La définition d'une distribution est constituée d'un descripteur d'assemblage associé à une séquence d'instructions, la plupart étant des opérations sur des fichiers. La création d'une nouvelle distribution est une tache complexe car chaque service technique possède des dépendances diverses

5. Apache iPOJO : `http://felix.apache.org/site/apache-felix-ipojo.html`
6. Apache Felix : `http://felix.apache.org`
7. Apache Maven : `http://maven.apache.org/`

non spécifiées, telles que des fichiers de configuration, des librairies ou d'autres services techniques.

1.2.6.3 Pré-requis pour la définition de profils

Ces distributions de *JOnAS* correspondent à des profils différents d'utilisation. Cependant, *JOnAS* ne propose actuellement pas d'outil permettant de définir des distributions à partir d'une description de profil. Les développeurs de *JOnAS* prévoient la création d'un tel outil (appelé *JOnAS Builder* et illustré par la figure 1.5), permettant la construction de distributions pour des profils d'utilisation divers.

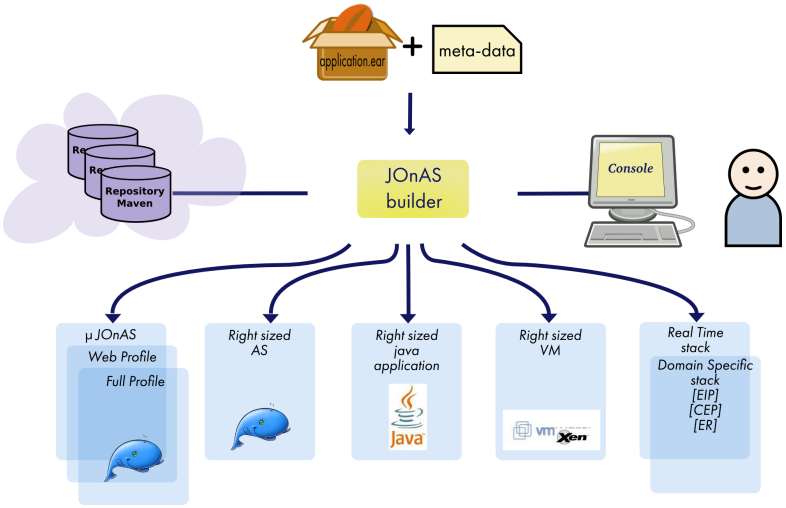


FIGURE 1.5 – Construction de profils *JOnAS*

Les distributions de *JOnAS* pourront être adaptées à la fois aux applications et à la cible de déploiement, car assemblées selon divers formats, allant du fichier traditionnel d'archive à l'*appliance*, destinée à être déployée sur un hyperviseur ou dans le nuage.

Une description de profil devrait contenir une déclaration de l'ensemble des services techniques requis. À partir d'une telle description et d'une base de connaissance décrivant les dépendances et le contenu de tous les éléments possibles composant une distribution de *JOnAS* (parmi lesquels les services techniques), l'outil *JOnAS Builder* générera un assemblage satisfaisant les besoins décrits par le profil. Cet outil requiert entre autre une telle base de connaissance. Dans cette thèse, nous proposons une implémentation de celle-ci, à l'aide d'un dépôt contenant une représentation abstraite (sous forme de composant) de chacun des éléments composant une distribution de *JOnAS*. Par exemple, la définition d'une abstraction d'un service technique permet de le caractériser et de lui associer l'ensemble de ses dépendances. D'une manière plus générale, une représentation abstraite du serveur, de son noyau indivisible et de ses extensions seront nécessaires à l'outil *JOnAS Builder*.

Une difficulté est qu'un serveur d'applications *Java EE* a également des dépendances avec des librairies du système d'exploitation (telle que la machine virtuelle *Java*). Ces dépendances peuvent être provoquées à la fois par un service technique et une application métier. Par exemple, une application Java EE peut être associée à une base de données contenant des tables de cette application. Un service *JOnAS* offrant la réplication des sessions *EJB3* utilise le cache distribué fourni par *Ehcache*[8], lequel requiert l'exécution d'un processus externe à la machine virtuelle *Java*.

1.2.6.4 Motivations pour un serveur d'applications léger

Excepté la taille de la distribution, le choix d'un profil spécifique est également important afin de diminuer les ressources utilisées à l'exécution. Selon la distribution, le nombre de services activés par défaut est ainsi différent. À l'exécution, ceci se matérialise par une occupation mémoire de 182 MO (sur *MS Windows*) pour le profil *Micro JOnAS* contre 417 MO pour le profile *JOnAS complet*. De plus, la différence du temps de démarrage est significative, puisque le temps de démarrage est de l'ordre de trois secondes pour *Micro JOnAS* contre dix pour *JOnAS complet*. Ce temps de démarrage doit être pris en compte lorsqu'une instance du serveur est allouée dynamiquement lors de pics de charges. Ces exemples (mémoire réduite et gain de temps/ rapidité d'exécution) illustrent ainsi l'intérêt d'avoir un serveur d'applications à la juste taille dans le contexte d'économie de ressources informatiques.

1.2.6.5 Déploiement d'applications hétérogènes

Notre souhait est d'être capable de déployer une application *Java EE* ainsi que son environnement d'exécution. Cet environnement comprend un assemblage de *JOnAS* adapté à l'application, ainsi que des librairies systèmes. Les trois sortes d'hétérogénéité sont gérées lors d'un tel déploiement :

Les briques logicielles sont de types bundle *OSGi*, application *Java EE*, paquet système ou simplement fichier ;

Les outils de déploiement sont constitués de *Maven* pour l'assemblage de *JOnAS*, *SSH* pour la connexion sur les cibles de déploiement et de diverses commandes shell ;

Les cibles de déploiement utilisent des systèmes d'exploitation différents.

1.3 Approche et contributions

Le travail présenté dans cette thèse s'inscrit dans le cadre des solutions de déploiement logiciel. Nous proposons d'automatiser le déploiement d'une application et de ses dépendances, à la juste taille. Nous considérons à la fois les applications patrimoniales et nouvelles.

8. *Ehcache* : `http://ehcache.org`

Notre contribution se compose de quatre volets. Dans un premier temps, nous poserons les fondations de celle-ci avec la spécification d'un modèle pour la représentation des logiciels et de leur déploiement. Puis, nous proposerons une mise en œuvre de ce modèle au sein d'un canevas logiciel dénommé *SU Framework*. Enfin, nous décrirons comment résoudre les problématiques citées en le démontrant sur l'étude de cas *JOnAS*, au travers de deux expérimentations issues de besoins industriels : la gestion de profils *JOnAS* et le déploiement d'applications *Java EE* avec leur environnement d'exécution (lequel est composé d'applications hétérogènes).

La figure 1.6 décrit les relations entre les quatre éléments de la contribution.

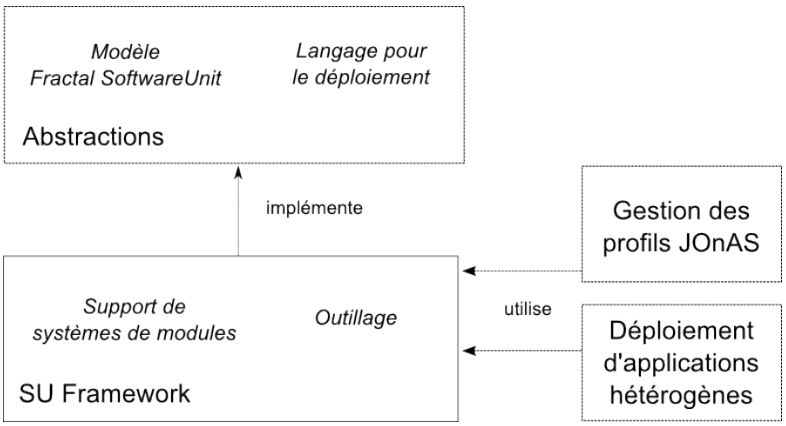


FIGURE 1.6 – Relations entre les contributions

1.3.1 Un modèle pour la représentation des logiciels et de leur déploiement

Nous proposons de résoudre la problématique du déploiement d'applications à la juste taille à l'aide du modèle à composants *Fractal SoftwareUnit*, lequel est une spécialisation du modèle à composants *Fractal*, pour la représentation des dépendances et du contenu des logiciels. Notre modèle supporte l'hétérogénéité des technologies de logiciel, une technologie logicielle étant représentée par un système *SoftwareUnit*. Enfin, ce modèle facilite le contrôle de la cohérence du déploiement.

Le cycle de vie du processus de déploiement considéré commence dès la sortie du logiciel, avec l'assemblage de celui-ci. Les composants *SoftwareUnit*, définis dans le modèle, sont des composants *Fractal* permettant de représenter les logiciels tout au long de ce processus. Nous adoptons une vision fonctionnelle du déploiement, dans laquelle les *opérations* sont des fonctions dont le domaine est défini par un ensemble de composants *SoftwareUnit* et le codomaine par un ensemble de composants *Fractal*.

D'une part, l'identification des dépendances permet de garantir l'exhaustivité du déploiement et d'autre part, aide à ne sélectionner que celles qui sont nécessaires. Sachant que le déploiement d'un composant implique l'exécution d'opérations, une telle sélection permet de minimiser le nombre d'opérations à invoquer. Si celles-ci consistent en l'ajout

d'éléments dans un assemblage, notre approche permet alors de minimiser leur nombre et en conséquent la taille de celui-ci.

La représentation des logiciels, que nous proposons, ambitionne les qualités suivantes :

Fine granularité Un composant peut représenter n'importe quel contenu et besoin d'un logiciel ;

Hétérogénéité Un composant peut représenter n'importe quelle technologie de logiciel ;

Contrôle distribué et extensible Un composant fournit des contrôleurs pour interagir avec le logiciel représenté. Le contrôle offert est extensible et autorise également différents niveaux de granularité. Les contrôleurs permettent enfin de maintenir la cohérence de la représentation du logiciel.

L'hétérogénéité est présente à la fois dans la technologie des logiciels et dans les cibles de déploiement. Notre modèle autorise la définition de descriptions architecturales génériques ne mentionnant pas l'implémentation lorsque celle-ci doit être découverte au cours du déploiement.

1.3.2 Une architecture dynamique pour l'ajout du support de logiciels et de primitives de déploiement

Nous avons conçu le canevas *SU Framework* (aka. *SoftwareUnit Framework*), ou *SUF* en abrégé, implémentant le modèle *Fractal SoftwareUnit*.

Son architecture logicielle est basée sur des composants *Fractal* à services, permettant de concilier le modèle à composants *Fractal* et les couches cycle de vie et service d'*OSGi*.

Cette implémentation autorise l'ajout à l'exécution de nouveaux systèmes *SoftwareUnit*, afin d'intégrer dynamiquement la prise en charge de technologies de logiciel. Dans cette thèse, nous avons notamment défini un support des systèmes de modules *OSGi*, *RPM* et *Debian*.

Le logiciel réalisé durant cette thèse est écrit principalement en langage *Java*. Celui-ci est disponible sous licence LGPL version 2.1 [9] au sein du projet *OW2 JOnAS* [10].

FIGURE 1.7 – Logo du *SU Framework*

9. Licence LGPL version 2.1 : `http://www.gnu.org/licenses/lgpl-2.1.html`
10. Dépôt SVN : `svn://svn.forge.objectweb.org/svnroot/jonas/sandbox/su-framework/trunk`

1.3.3 Gestion des profils *JOnAS*

Nous proposons une approche déclarative pour la définition de profils *JOnAS*, dans laquelle une opération construit un assemblage de *JOnAS* pour une spécification de besoins.

Un travail préalable a été de construire une représentation d'un assemblage fonctionnel de *JOnAS* et de chacun de ses éléments, le tout avec des composants *SoftwareUnit*. Un assemblage est construit incrémentalement à l'aide d'une opération permettant d'ajouter des éléments à un assemblage existant ou un patron de *JOnAS*.

1.3.4 Déploiement d'applications hétérogènes sur cibles hétérogènes

Une application hétérogène multi-tiers et multi-niveaux *Java EE* est décrite en terme de composants. Le déploiement est adapté selon, d'une part, la description de l'application et d'autre part, l'environnement des cibles du déploiement. Ainsi, le profil *JOnAS* à la juste taille est assemblé et copié sur une machine cible afin d'exécuter l'application *Java EE Soapsoo*, laquelle sera décrite dans le chapitre 6. *JOnAS* requérant une machine virtuelle *Java*, celle-ci est installée si nécessaire, quel que soit le système d'exploitation.

1.4 Organisation du document

Ce document est composé de neuf chapitres divisés en quatre parties, la présente introduction étant indépendante.

La première partie regroupe l'état de l'art du domaine, lequel est divisé en deux chapitres : les concepts et technologies dans le chapitre 2 et les solutions existantes dans le chapitre 3.

La seconde partie présente le modèle et les langages sur lesquels est basée notre solution. Cette partie est composée de deux chapitres et commence avec le chapitre 4, spécifiant le modèle et les langages proposés. Puis, la mise en œuvre de ceux-ci est décrite dans le chapitre 5.

La troisième partie propose une expérimentation et des résultats de notre approche. Dans le chapitre 6 l'approche proposée est validée pour le contexte particulier de l'assemblage avec comme cas d'utilisation le serveur d'applications *JOnAS*. Dans le chapitre 7, nous utilisons *SUF* pour automatiser le déploiement d'une application *Java EE* et de ses dépendances jusqu'au système d'exploitation, sur un environnement distribué. Enfin, cette partie se termine sur le chapitre 8 analysant le coût de notre solution.

La quatrième et dernière partie contient les annexes.

Première partie

État de l'art

Chapitre 2

Concepts et technologies liés au sujet

Sommaire

Nous présentons dans ce chapitre un certain nombre de concepts et de technologies liés à nos travaux. Dans un premier temps (section 2.1), nous caractériserons le déploiement logiciel. Puis, nous verrons quelle place les composants logiciels occupent dans le déploiement (section 2.2). Enfin, nous analyserons des acteurs incontournables du déploiement de logiciels patrimoniaux : les systèmes de gestion de modules (section 2.3).

2.1 Characterization Framework

Carzaniga et al. (1998) proposent un canevas pour caractériser les technologies destinées à supporter du déploiement logiciel distribué [16]. Quatre facteurs de caractérisation sont retenus :

- La couverture du processus de déploiement ;
- La variabilité du processus de déploiement ;
- La coordination entre processus de déploiement ;
- Le modèle d'abstraction.

Les logiciels déployés sont appelés *systèmes logiciels*. Un *système logiciel* est défini comme une collection cohérente d'artefacts logiciels fournissant des fonctionnalités à des utilisateurs finaux. Celui-ci requiert des *ressources* (logicielles ou matérielles) sur le *site client*, cible du déploiement. Lors du déploiement d'un système logiciel, ses constituants peuvent être copiés du *site de production* vers le *site client*.

2.1.1 Couverture du processus de déploiement

Le canevas identifie huit activités constituant généralement un processus de déploiement, celui-ci étant non linéaire et non séquentiel (voir la figure 2.1 pour les transitions possibles).

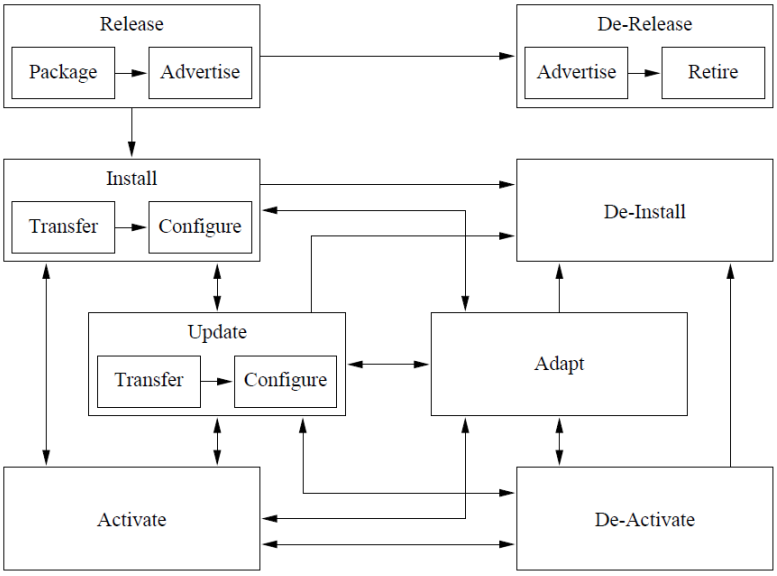


FIGURE 2.1 – Activités d'un processus de déploiement

Les activités intègrent des procédures de déploiement, lesquelles pouvant être exécutées sur les sites de production ou des clients.

Sortie (*release*) L'activité de sortie (ou de mise à disposition) interface le processus de déploiement avec celui de développement. Cette activité comporte l'emballage (*package*) du système, consistant à l'ajout des informations utiles pour les clients, telles que les dépendances et les ressources nécessaires. La seconde étape de cette activité est la publicité (*advertise*) auprès des clients de cette sortie.

Installation (*install*) L'activité d'installation couvre l'insertion du système logiciel dans un site client. Cette activité est composée du *tranfert* du produit du site de production vers le site client, puis de sa configuration (*configure*). La configuration rend le système prêt à être activé.

Activation (*activate*) L'activité d'activation démarre les composants exécutables d'un système.

Désactivation (*deactivate*) La désactivation arrête l'exécution de composants précédemment activés.

Mise à jour (*update*) La mise à jour est un cas particulier d'installation, pour laquelle un système logiciel est déjà installé, mais avec une version antérieure. Cette activité comporte donc les même sous-activités *tranfert* et *configure*.

Adaptation (*adapt*) L'activité d'adaptation est une modification isolée d'un système logiciel sur le site client, c'est-à-dire qu'elle n'est pas corrélée avec la sortie d'une nouvelle version, comme pour une mise à jour.

Désinstallation (*deinstall*) L'activité de désintallation efface un système logiciel d'un site client.

Fin de support (*derelease*) L'activité de fin de support intervient quand un logiciel est obsolète et n'est plus maintenu par les développeurs. Celle-ci commence par la publicité du retrait (*advertise*) et se termine avec le retrait (*retire*) du système.

2.1.2 Variabilité du processus de déploiement

Le critère de variabilité indique si un processus de déploiement peut être modifié après qu'il ait été défini.

2.1.3 Coordination entre processus de déploiement

Le critère de coordination permet de mesurer la capacité d'une technologie de déploiement à manipuler les systèmes composites et distribués, au travers notamment, de la synchronisation et de l'échange de données entre les différentes activités.

2.1.4 Modèle d'abstraction

Le critère de modèle d'abstraction permet d'évaluer l'effort requis pour la définition de procédures de déploiement. L'utilisation de modèles abstractions permet de définir des procédures de déploiement plus génériques.

Trois types de modèle d'abstraction sont cités :

Modèle de site Un modèle de site permet d'abstraire l'environnement du site client.

Modèle de produit Un modèle de produit permet de décrire les dépendances et les contraintes d'un système.

Modèle de politique Une politique de déploiement permet d'adapter l'exécution d'une procédure de déploiement. La résolution de dépendances constitue un exemple de politique : elle permet d'associer les dépendances d'un logiciel à l'exécution d'une procédure de déploiement. Un autre exemple de politique est le protocole de communication utilisé pour le déploiement (eg. FTP, SSH).

2.2 Composant logiciel

Les propriétés des composants logiciels en font des cibles incontournables pour le déploiement logiciel. En effet, « *un composant peut être déployé indépendamment et est sujet à la composition par des tiers* » selon C. Szyperski [59]. A. Brown décrit un composant comme « *un élément de fonctionnalité, indépendant et livrable, fournissant un accès à ses services à travers des interfaces* » [12]. Enfin, la spécification *UML* définit : « *un composant représente une partie modulaire d'un système qui encapsule son contenu*

et dont la manifestation est remplaçable dans son environnement » [51]. Toutes ces définitions décrivent un composant logiciel comme une unité indépendante de déploiement.

2.2.1 Le modèle à composants *Fractal*

Le modèle à composants *Fractal*[1] [6, 14] est destiné à la construction de systèmes logiciels adaptables. Il possède une spécification [13] et de multiples implémentations, *Julia* étant celle de référence. Ce modèle propose une définition de composant réflexif, récursif (hiérarchique sur plusieurs niveaux) et offrant des capacités de contrôle. Le modèle définit la notion de composants partagés, facilitant ainsi la représentation de ressources.

Le modèle *Fractal* impose peu de restrictions et peut être utilisé selon divers niveaux de conformité. Il n'est pas lié à un langage de programmation.

2.2.1.1 Éléments de base du modèle

Composant Un composant *Fractal* possède un contenu et une membrane. Le contenu implémente les fonctionnalités offertes par le composant et la membrane implémente ses capacités de contrôle, permettant, par exemple, d'administrer celui-ci.

Un composant non décomposable est dit primitif ; dans le cas contraire, il est appelé composite. Un composant peut être partagé, en appartenant à plusieurs composites à la fois.

Un composant possède un type, permettant notamment de vérifier la substituabilité de composants. Le système de type proposé par la spécification (celui-ci pouvant être modifié) définit le type d'un composant à partir de l'ensemble des types des interfaces métier de ce composant.

Interface Une interface est un point d'accès au composant. Un composant *Fractal* peut possèder des interfaces fonctionnelles (dites de métier) et des interfaces non-fonctionnelles (dites de contrôle).

Le système de type, proposé dans la spécification *Fractal*, définit le type d'une interface à partir d'une signature, correspondant au nom du type d'une interface langage.

Les interfaces sont dites de type *serveur* lorsqu'elles décrivent un service fourni et de type *client* lorsqu'elles décrivent un service requis. Fonctionnalités et capacités de contrôle sont ainsi décrites par des interfaces de type serveur.

Nommage et liaison Une interface est identifiée à l'aide d'un nom. Un nom peut avoir diverses formes, telles qu'une référence *Java* ou une référence *CORBA IOR*. Un nom est associé à un contexte de noms, ce dernier permettant de créer un nom à partir d'une interface de composant. Un nom peut être encodé sous une forme dite sérialisée, afin notamment de le communiquer à des tiers distants.

1. OW2 Fractal `http://fractal.ow2.org/`

Une liaison représente un canal de communication entre interfaces de composant. Elle est créée à partir du nom d'une interface. Une liaison primitive lie une interface de type client avec une interface de type serveur, dans le même espace d'adressage. Ce type de liaison correspond à une liaison locale. À la différence, une liaison composite peut mettre en jeu différents espaces d'adressage. Elle est dite composite car elle est un assemblage de composants de liaison (représentant des connecteurs) et de liaisons primitives.

La création d'une liaison peut être conditionnée à une compatibilité entre les types des interfaces client et serveur. D'après le système de type proposé dans la spécification *Fractal*, la compatibilité existe si le type associé à l'interface langage de l'interface serveur est un sous-type de celui associé à l'interface langage de l'interface client.

2.2.1.2 Cycle de vie des composants

Un composant est créé selon le patron de conception *factory*[2]. Une usine à composant générique délivre un composant à partir d'un type de composant, d'une description de la membrane et d'une description du contenu.

Le contrôle de l'exécution est offert par le contrôleur de cycle de vie, lequel est présenté dans la prochaine section.

2.2.1.3 Contrôle des composants

La spécification *Fractal* définit quatre contrôleurs, pouvant être intégrés, ou non, dans la membrane du composant.

Contrôle des attributs Le contrôleur d'attributs donne accès en lecture et en écriture aux propriétés configurables d'un composant.

Contrôle des liaisons Une interface de type client peut être liée à l'interface serveur d'un autre composant à l'aide du contrôleur de liaison. Ce dernier permet également de suivre et détruire une liaison.

Contrôle de contenu Le contrôleur de contenu permet la récupération, l'ajout et la retrait de sous-composants.

Contrôle de cycle de vie Le contrôleur de cycle de vie permet de changer l'état d'exécution d'un composant. La spécification définit un cycle de vie de base à deux états, correspondant soit à un composant arrêté, soit à un composant démarré. Ce cycle de vie peut être librement étendu.

2. `http://sardes.inrialpes.fr/~krakowia/MW-Book/Chapters/Basic/basic-body.html#tth_sEc2.3.2`

2.2.1.4 Intérêt

Le modèle étant générique et extensible, de nombreux travaux l'ont utilisé pour bâtir des intergiciels adaptables (eg. *DeployWare* [32], *OW2 CLIF*[3]). *JonasALaCarte* [1] propose d'utiliser les capacités du modèle à composants *Fractal* pour modulariser et faciliter la configuration du serveur d'applications *JOnAS*. Un composant *Fractal* est également applicable à l'administration de logiciels patrimoniaux à gros grain ou d'architectures logicielles (eg. applications multi-tiers) [10].

2.2.2 Configuration d'architectures à composants avec *FScript*

Les architectures à composants sont une solution à la complexité des logiciels. Elles offrent une abstraction facilitant la construction de logiciels, ainsi que leur réutilisation et évolution. Un processus de déploiement peut être défini à l'aide d'une description architecturale, offrant une vision de haut niveau du logiciel déployé.

Le modèle à composants *Fractal* peut être utilisé pour la construction de telles architectures. Il bénéficie notamment d'un riche écosystème logiciel en facilitant la conception. L'outil *FScript*[4] [20] permet notamment de programmer des reconfigurations dynamiques de telles architectures.

FScript définit un langage dédié, ainsi qu'un interpréteur, écrit en *Java* et supportant des reconfigurations transactionnelles.

2.2.2.1 Le langage *FScript*

Un programme *FScript* est constitué d'une séquence de définitions de procédure. Ces procédures peuvent être des fonctions ou des actions. Une fonction n'a pas d'effet de bord et est donc dédiée à l'introspection de composants et d'architectures. À l'inverse, une action est destinée à la modification de ces derniers. Le langage est impératif et dispose de structures de contrôle telles que l'itération et le branchement conditionnel. Il utilise un typage dynamique.

Afin de simplifier la sélection d'éléments dans une architecture à composants, *FScript* propose le langage d'expression *FPath*, lequel est inspiré de *XPath*[5]. Ce langage est basé sur une représentation alternative de telles architectures, à partir d'un modèle basé sur des graphes orientés étiquetés. Dans ce modèle, les éléments à réifier sont associés à un nœud du graphe. Un arc caractérise une relation entre deux éléments, la nature de cette relation étant spécifiée par une étiquette. Une architecture à composants est réifiée sous forme d'un graphe.

Une expression *FPath* permet de naviguer dans un tel graphe. Par exemple, une étape suivie du symbole *slash* (/) permet de sélectionner des nœuds à partir d'autres nœuds. La première étape correspond à une variable référençant un ensemble de noeuds du graphe. Chacune des étapes suivantes est décrite à l'aide de trois informations :

3. OW2 ClIF : `http://clif.ow2.org/`
4. FScript : `http://fractal.ow2.org/fscript/`
5. Langage XPath : `http://www.w3.org/TR/xpath/`

Un axe défini par l'étiquette d'arcs reliant les noeuds d'origine aux noeuds de cible (un axe pouvant référencer plusieurs arcs) ;

Le nom des noeuds de cible reliés par cet axe ;

Un prédicat optionnel permettant de sélectionner des noeuds de cible.

Par exemple, l'expression $comanche/child::fe[id(.)=="server1"] sélectionne les noeuds de nom *fe*, reliés aux noeuds référencés par la variable *comanche*, avec des arcs ayant pour étiquette *child*. Seuls les noeuds vérifiant le prédicat *id(.)=="server1"* sont retournés.

Le langage *FPath* étant indépendant d'un modèle à composants, le modèle proposé peut être vu comme un méta-modèle, permettant la représentation d'architectures d'un quelconque modèle à composants. Ainsi, *FScript* propose des instances de ce méta-modèle pour les modèles à composants *Fractal* et *MBean*[6].

2.2.2.2 Spécialisation du modèle

L'instanciation du méta-modèle revient à donner une sémantique à un nœud et à un arc. Concernant le modèle *Fractal*, les nœuds représentent des éléments importants des composants *Fractal*, tels que les composants, les interfaces et les attributs. Les axes correspondent, entre autres, aux relations de contenance entre composants (*child*), de liaison entre interfaces (*binding*) et d'appartenance d'une interface à un composant (*interface*).

Dans le modèle *Fractal*, l'expression *FPath* décrite précédemment, peut être interprétée comme la sélection des sous-composants de ceux référencés par la variable *comanche*, ayant pour nom *fe* et dont l'identifiant est *server1* (en supposant que *id* soit le nom d'une fonction retournant l'identifiant d'un composant).

FScript permet également de définir des procédures primitives. Celles-ci correspondent à des procédures nativement implémentées en *Java*. Ainsi, le modèle *FScript* pour *Fractal* définit la procédure primitive *name*, retournant le nom d'un composant.

2.2.2.3 Intérêt pour la définition de processus de déploiement

En facilitant la création et la modification d'architectures à composants, *FScript* est une solution pertinente pour la définition de processus de déploiement. Son extensibilité en fait une solution résolument générique.

2.3 Systèmes de gestion de modules

Dans la section 1.2.2, nous présentions un exemple d'applications composées de briques logicielles hétérogènes. Dans cette section, nous analysons plus finement les caractéristiques de cette hétérogénéité avec, en guise d'illustration, des systèmes de gestion de modules patrimoniaux que nous souhaitons être capable de déployer. Parmi ceux-ci,

6. Spécification JMX : `http://jcp.org/en/jsr/detail?id=3`

nous distinguons les systèmes de gestion de modules appartenant à des systèmes d'exploitation (section 2.3.1) avec ceux définis par l'intergiciel *Java* (section 2.3.2).

2.3.1 Systèmes d'exploitation

Chaque système d'exploitation adopte généralement un système de gestion de paquets afin de faciliter l'automatisation du déploiement des logiciels (il est préférable d'employer le terme de paquet plutôt que celui de module dans le contexte des systèmes d'exploitation). Un système de gestion de paquets privilégie un format de paquets. Pour l'assemblage des logiciels, *MacOS X* définit, par exemple, le format *PKG*, *MS Windows* recommande le format *MSI* et les distributions GNU/Linux optent principalement soit pour le format *RPM*, soit pour le format *Debian*.

Les distributions *GNU/Linux* ont été à la pointe dans ce domaine[7]. Les formats *RPM* et *Debian* permettent l'un comme l'autre de décrire des dépendances sur d'autres paquets, ainsi que le processus d'installation. Ceux-ci sont très similaires comme nous le constaterons dans les deux sections suivantes.

2.3.1.1 Les paquets *RPM*

Le format de paquet *RPM*[8] a été initialement développé pour *Red Hat Linux* et est aujourd'hui adoptée par de nombreuses distributions (eg. *Mandriva*, *Suse* ou *Fedora*). *Linux Standard Base*[9] recommande le choix du choix du format de paquet *RPM*.

Un paquet *RPM* est un fichier binaire divisé en quatre sections :

Lead permettant d'identifier le paquet (eg. version du format utilisé, paquet binaire ou source, architecture) ;

Signature permettant de vérifier l'intégrité et l'authenticité du paquet ;

Header décrivant le logiciel (la section nous intéressant) ;

Archive contenant le logiciel.

La section *Header* contient les informations décrivant le logiciel, chacune étant associée à un *tag*. Le nom du logiciel est indiqué par le tag *Name* et sa version par le tag *Version*. La version du paquet est donnée par le tag *Release*.

Concernant la description des dépendances du logiciel, le format *RPM* autorise les dépendances sur d'autres paquets *RPM* ou sur des fichiers. Le tag *Requires* contient la liste des paquets et fichiers requis pour que le logiciel fonctionne correctement. Au contraire, le tag *Conflicts* liste les paquets non compatibles. Ces deux tags peuvent être accompagnés d'une version ainsi que d'un opérateur de comparaison.

Le tag *Provides* est associé à un nom de paquet virtuel. L'utilisation de nom de paquets virtuels permet de définir des alternatives de paquets fournissant un même paquet virtuel (eg. les paquets *java-1.5.0-gcj* et *java-1.6.0-openjdk* fournissent le paquet virtuel *java*). Un nom de paquet virtuel peut également être un nom de fichier contenu

7. `http://ianmurdock.com/solaris/how-package-management-changed-everything/`
8. RPM Package Manager (RPM) v4 Homepage : `http://www.rpm.org/`
9. Linux Standard Base 4.0 : `http://ldn.linuxfoundation.org/lsb/lsb4-resource-page`

dans ce paquet. Par exemple, le paquet (non virtuel) de nom *bash* fournit le paquet virtuel de nom */bin/sh*, lequel désigne un fichier nécessaire à l'exécution de nombreux scripts shell.

Un paquet peut indiquer qu'il remplace un autre en utilisant le tag *Obsoletes*. Enfin, un fichier *RPM* peut avoir des dépendances sur l'environnement : les tags *ExcludeArch* et *ExclusiveArch* permettent d'exclure ou de restreindre l'architecture matérielle, tandis que les tags *ExcludeOS* et *ExclusiveOS* permettent d'exclure ou de restreindre le système d'exploitation.

Un fichier *RPM* peut contenir des scripts pour construire, installer ou effacer un logiciel. Le format *RPM* permet d'identifier les fichiers de configuration, évitant aux mises à jour d'écraser la configuration existante. La documentation est également identifiée séparément.

Le système de gestion de paquets *RPM* supporte l'installation de versions multiples. Néanmoins, cette gestion des versions multiples est minimale, le nom du paquet étant finalement redéfini à partir de la concaténation de celui-ci avec une version (en *Debian*, cela doit être fait explicitement).

L'outil de base pour la gestion de paquets *RPM* est le programme *rpm*. De nombreux gestionnaires de paquets (tels *yum*, *ZYpp* ou *urpmi* simplifient l'installation en automatisant le téléchargement des fichiers *RPM* (via des dépôts distants de paquets) et la résolution des dépendances de ceux-ci.

2.3.1.2 Les paquets *Debian*

Le format de paquet paquet *Debian* est défini dans le manuel des politiques de *Debian* [41]. Un paquet *Debian* (binaire) est une archive contenant les fichiers à installer, ainsi que des fichiers d'information de contrôle. Ces derniers contiennent notamment les méta-données décrivant le paquet ainsi que ses dépendances. Parmi les méta-données décrivant une capacité, le champ *package* indique le nom du paquet et le champ *version* donne la version du logiciel et du paquet.

Tout comme *RPM*, *Debian* permet de déclarer des dépendances (champ *depends*), des conflits (champ *conflicts*) sur des paquets versionnés ou d'indiquer qu'un paquet peut en remplacer un autre (avec le champs *replaces*). À la différence de *RPM*, *Debian* n'autorise pas les dépendances sur les fichiers mais définit des dépendances alternatives explicites (grâce à l'opérateur |). Néanmoins, cet opérateur n'augmente pas le pouvoir sémantique du langage de dépendance, puisque l'utilisation de paquets virtuels peut aisément le remplacer. *Debian* introduit également des niveaux d'importances pour les dépendances avec les champs *recommends* (pour des paquets recommandés) ou *suggests* (pour des paquets suggérés). *Debian* permet également à des paquets de compléter les dépendances d'autres paquets. Ainsi, le champ *enhances* permet d'ajouter le paquet le portant comme paquets suggérés aux paquets cités. De même, le champ *breaks* permet d'ajouter un paquet comme conflictuel aux paquets cités. Enfin, une dépendance sur l'architecture matérielle peut être exprimée avec le champ *architecture*.

Un paquet *Debian* peut avoir une priorité associée. Par exemple, le niveau *required* indique qu'un paquet est nécessaire au fonctionnement du système de base. Le niveau

important désigne l'ensemble minimal de paquets que l'on peut trouver sur n'importe quel système Unix. Un paquet ne devrait pas avoir de dépendance sur des paquets ayant des niveaux de priorité inférieurs.

Comme un paquet *RPM*, un paquet *Debian* peut contenir des scripts pour construire, installer ou effacer un logiciel, en permettant d'identifier la configuration de ce dernier.

L'outil de base pour la gestion de paquets *Debian* est le programme *dpkg*. Le gestionnaire de paquets *apt-get* simplifie l'installation, la désinstallation ou la mise à jour des paquets en automatisant la résolution des dépendances et le téléchargement des paquets depuis des dépôts distants.

2.3.2 Systèmes de gestion de modules pour *Java*

Notre étude de cas (ie. le serveur d'applications *JOnAS*) étant un intergiciel *Java*, nous commentons, dans cette section, le support de la modularité offert par la plate-forme *Java*. Puis, nous présentons la plate-forme à services *OSGi*, au dessus de laquelle *JOnAS* est exécuté.

2.3.2.1 La plate-forme *Java*

La plate-forme *Java* définit un langage de programmation orienté objet, une machine virtuelle (*JVM*), des librairies, ainsi qu'un environnement de développement. La machine virtuelle définit son propre langage machine (le *bytecode Java*) qu'elle interprète ou compile en langage machine à l'exécution.

Le *bytecode Java* est contenu dans un fichier *class*. Ce fichier définit une classe représentant un type en *Java*. Le chargeur de classes interprète le *bytecode Java* contenant dans un fichier *class* en une classe et permet également d'accéder à des ressources (ie. des fichiers) définis dans l'environnement d'exécution. Le chargeur de classe utilise un chemin dans le système de fichier (appelé le *Class-Path*) afin de trouver le *bytecode Java* et les ressources. Un *package Java* contient un ensemble de fichiers *class* ainsi que des ressources. Ils permettent notamment de contrôler la visibilité des types en *Java*.

La plate-forme *Java* définit le format *JAR* pour la distribution et le chargement de programmes *Java*. Un fichier *JAR* est une archive contenant des *packages Java* associés à un fichier *Manifest*. Ce fichier permet d'associer des méta-données au logiciel contenu. Cependant, la spécification du format *JAR* [52] ne définit pas de langage de description de dépendance, la plate-forme *Java* ne proposant pas de système de gestion de module. En outre, la plate-forme *Java* ne propose rien permettant de restreindre la visibilité du contenu du fichier *JAR*.

La plate-forme *Java* n'est elle même pas modulaire (les librairies de la plate-forme sont définies quasiment toutes dans le même fichier *rt.jar*) et est complètement monolithique : de nombreuses librairies sont inter-connectées et ne peuvent être séparées. Par exemple, M. Reinhold (architecte en chef de la plate-forme *Java*) déclara qu'un simple programme affichant *Ḧello World¨* sur une ligne de commande requiert au moins 300 classes différentes [10]. Néanmoins, le projet *Jigsaw* ambitionne la définition d'un nouveau

10. `http://blogs.sun.com/mr/entry/massive_monolithic_jdk`

système de modules pour la plate-forme *Java*[11].

2.3.2.2 La plate-forme à services *OSGi*

Suite à la JSR 8[12], la plate-forme à services *OSGi* a été initiée en 1999 par l'*OSGi Alliance*[13] comme passerelle domestique. Elle est aujourd'hui une solution pour la gestion de la modularité dans le monde *Java*.

La plate-forme *OSGi* est spécifiée en trois volumes (*core*, *compendium* et *enterprise*) par l'*OSGi Alliance*. Le volume *core*[63] décrit le coeur de la plate-forme : le *Framework OSGi*. Celui-ci est construit en terme de couches comme l'illustre la figure 2.2.

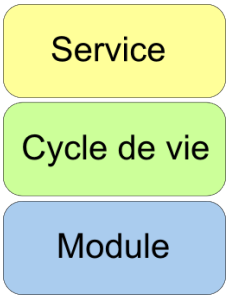


FIGURE 2.2 – Les 3 couches de bases du *Framework OSGi*

Module La couche module décrit un modèle à composants sans composite. Les composants représentent des modules. Ces modules ont des dépendances explicites vers d'autres modules. À partir de ce modèle à composants, la spécification *OSGi* définit un système de module pour *Java*, dans lequel l'unité de modularité est le *bundle*. Un *bundle* reprend le format *JAR* et possède un fichier *Manifest* décrivant les capacités et les besoins du module. Un *bundle* contient des *packages Java* et des ressources (correspondant à tout autre fichier). Les dépendances entre modules sont soit de type *bundle* soit de type *package*.

Une capacité de type *bundle* est exprimée dans le fichier *Manifest* par les entrées *Bundle-SymbolicName* et *Bundle-Version*. Ces entrées constituent l'identifiant du *bundle* dans le *Framework*. Une dépendance sur un *bundle* est exprimée avec l'entrée *Require-Bundle*. Une capacité de type *package* l'est avec l'entrée *Export-Package*. Un besoin de type *package* peut l'être de deux manières différentes selon le moment pour lequel le besoin doit être pris en compte. Ainsi, l'entrée *Import-Package* définit un besoin devant être pris en compte dès le processus de résolution, tandis que l'entrée *DynamicImport-Package* indique un besoin pouvant être ignoré lors de celui-ci (on parlera d'import dynamique). Le processus de résolution crée les liens entre modules et attribue à chaque

11. Projet Jigsaw : `http://openjdk.java.net/projects/jigsaw/`
12. Open Services Gateway Specification : `http://jcp.org/en/jsr/detail?id=8`
13. The OSGi Alliance : `http://www.osgi.org/`

bundle un chargeur de classes. Chaque import de *package* résolu (ie. associé à un lien au niveau du modèle) étend la visibilité des *packages* accessibles par le chargeur de classe du *bundle*. Un import dynamique est utile lorsqu'un *package* n'est pas accessible lors de la résolution. Le lien n'est alors construit que lorsque le chargeur de classe tente d'accéder au contenu du *package* importé.

Un import de *package* ou un export de *package* peuvent avoir des attributs associés, ainsi que des directives. Une directive est une instruction à destination du *Framework* concernant ce *package*. Un attribut est pris en compte lorsque deux *packages* sont comparés. L'utilisation d'attribut dans un import de *package* est ainsi interprété par le système de résolution de *packages* (aussi appelé *résolveur*) comme le choix de ne sélectionner que l'export ayant les attributs correspondants. Un attribut peut être la version du *package*, celle du *bundle* dans lequel est défini le *package* ou une quelconque chaîne de caractères. Un import de *package* facultatif est associé à la directive *resolution* dont la valeur est *optional*. Enfin, un export de *package* peut être associé à la directive *uses*, dont la valeur est une liste de noms de *package*, afin que les classes important ce *package* utilisent les mêmes versions de *package*.

Cycle de vie La couche cycle de vie ajoute un cycle de vie aux modules (voir la figure 2.3).

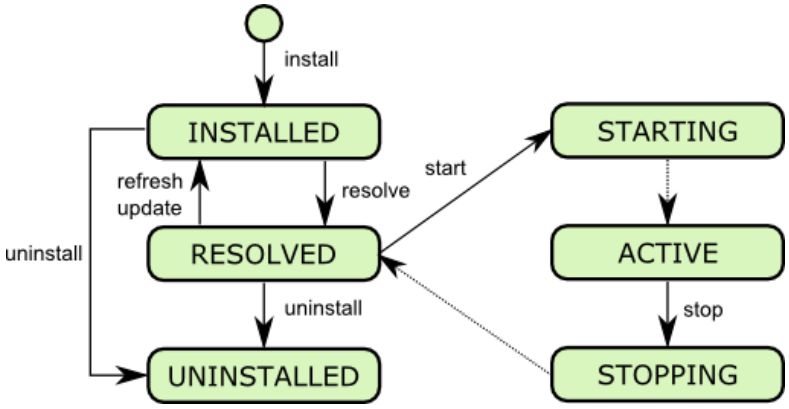


FIGURE 2.3 – Cycle de vie d'un *bundle OSGi*

Ce cycle de vie possède six états, dont deux sont transitoires (les états *STARTING* et *STOPPING*). Le processus de résolution décrit précédemment est effectué lors du passage de l'état *INSTALLED* à *RESOLVED*. Ce cycle de vie supporte l'installation, la mise à jour et le retrait de *bundle*. La couche cycle de vie propose une API permettant aux *bundles* d'avoir connaissance de leur état d'exécution. Similairement au contrôleur de cycle de vie d'un composant *Fractal*, l'activateur du *bundle* définit les opérations *start* et *stop* permettant d'appliquer le changement d'état au contenu. La gestion dynamique des *bundles* offerte par ce cycle de vie n'est exploitable que si les *bundles* sont faiblement couplés, ce que propose la couche supérieure.

Service La couche service définit un modèle de service dynamique proposant aux concepteurs de *bundles* de découpler leurs applications. Un service fournit des fonctionnalités et se caractérise par un contrat. Le modèle de service *OSGi* est basé sur l'approche à service décrite par M. Papazoglou [53]. L'approche à service repose sur une architecture orientée service, dont le patron de conception de base (illustré dans la figure 2.4) décrit les relations de trois sortes participants.

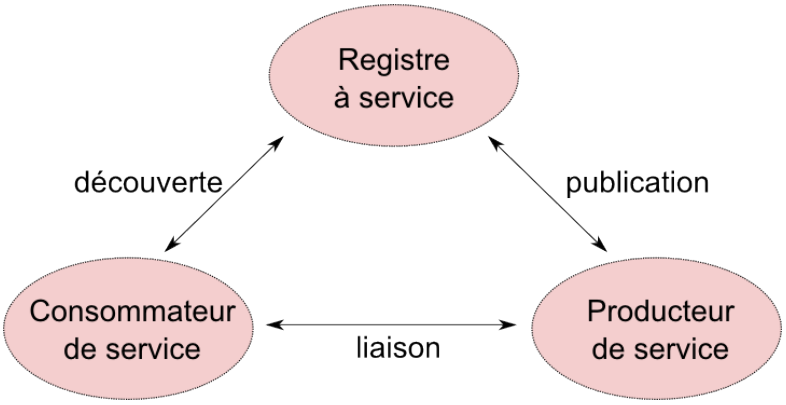


FIGURE 2.4 – Patron de base de l'architecture orienté service

L'approche à service dynamique autorise la mise à jour ou le retrait de service et permet au consommateur d'en être notifié.

La spécification permettait dans la version r3 de définir un service comme une capacité (avec l'entrée *Export-Service*) ou un besoin (avec l'entrée *Import-Service*) du module. Néanmoins, la description de dépendance de type service n'était pas prise en charge par le Framework mais destinée à des outils tiers (eg. *OBR*) et n'est plus supportée dans la spécification actuelle. La définition de dépendance de type service se fait via l'utilisation d'outils tels que *iPOJO* ou *Declarative Services*, lesquels définissent leurs propres méta-données.

2.3.3 Synthèse

Les technologies logicielles présentées illustrent l'hétérogénéité des logiciels. Leur format d'assemblage en est notamment un aspect. Les formats décrits dans cette section, présentent, à la fois, des similitudes et des différences.

Similitudes

Les trois systèmes de gestion de modules présentés sont associés à un format de fichier correspondant au module à déployer. Ce format définit un jeu de méta-données permettant d'exprimer ce que le module fournit et ce qu'il requiert, le tout dans un espace d'installation donné. *Debian* et *RPM* utilise le système de fichiers, *OSGi* repose sur une structure vivante à l'exécution de la plate-forme.

Différences

Debian et *OSGi* permettent l'expression de besoin facultatif.

OSGi permet l'expression de besoin indéterminé avec l'utilisation de joker dans un import dynamique.

Un paquet *Debian* ne peut avoir de dépendance sur le contenu d'un paquet. À l'inverse, un paquet *RPM* permet de déclarer une dépendance sur des fichiers, lesquels peuvent appartenir à un paquet. De même, un *bundle OSGi* peut exprimer des capacités et des besoins sur des *packages Java*, lesquels appartiennent au contenu du *bundle*. Les méta-données d'un *bundle OSGi* permettent d'exprimer des besoins de manière plus complexe avec notamment l'usage d'attributs.

En *OSGi*, un besoin ne peut exprimer de conflit, car cela est inutile. En effet, les *bundles* sont isolés et ne partagent pas d'états entre eux (comme peuvent le faire des paquets *RPM* ou *Debian* en partageant des fichiers, provoquant une collision).

Le support de versions multiples est offert par les systèmes *RPM* et *OSGi*, posant la question de la gestion de la cohérence. *OSGi* définit la directive *uses* afin de pouvoir la contrôler.

OSGi introduit un cycle de vie propre à ses *bundles*, lesquels offrent un contrôle d'exécution.

Manques

Dans les trois systèmes vus précédemment (*RPM*, *Debian* et *OSGi*), le type de dépendance supporté n'est pas extensible et se restreint à des capacités spécifiques aux système de module ou alors à une dépendance à très gros grain sur l'environnement. Comme le souligne A. Dearle [22], ce problème concerne particulièrement les intergiciels tels que *Java SE* ou *.Net*, à cause de l'abstraction de l'environnement qu'ils introduisent. L'utilisation de tels intergiciels complexifie le déploiement car leurs applications ont fréquemment des dépendances avec les couches inférieures. Ces dépendances ne sont pas réifiées au niveau du module. Les méta-données associées aux modules suffisent quand l'environnement d'exécution est déployé à part, mais se révèle insuffisant quand il s'agit de le construire à partir du module. Par exemple, les méta-données des *bundles* ne permettent pas de déclarer une dépendance sur le système d'exploitation. Ce manque est notamment un obstacle pour la construction d'une offre PaaS basée sur *OSGi*[14]. Néanmoins, la future spécification 4.3 (via la *RFC 154* [61]) définira des capacités et des besoins génériques permettant d'adresser les dépendances externes.

Les systèmes vus ne permettent pas l'expression de relation sur plusieurs modules ou de contrainte architecturale.

Enfin, chaque système de modules possède ses propres outils, ce qui oblige l'écriture de plan de déploiement tenant compte des spécificités de chacun ou alors à utiliser des outils génériques (nous en présenterons certains dans le chapitre suivant). Il serait utile d'avoir une définition commune de l'installabilité, comme celle proposée par le projet

14. RFP 133 Cloud Computing : `https://www.osgi.org/bugzilla/show_bug.cgi?id=114`

Edos[15] [9]. L'intérêt de définir un modèle formel, réunissant les propriétés principales des systèmes de paquetages les plus courantes, est de permettre la définition d'algorithmes, à la fois efficaces et universels, pour la manipulation des dépôts de paquets [8]. À cette fin, J. Boender propose le calcul des relations sémantiques entre paquets, afin de détecter des erreurs et de s'assurer de la qualité des distributions de logiciels libres.

2.3.4 Conclusion

Un processus de déploiement met en jeu diverses étapes du cycle de vie logiciel, telles que l'assemblage, l'installation et la mise à jour. Les logiciels peuvent être empaquetés dans un des nombreux formats disponibles. Nous avons notamment présenté, dans ce chapitre, les formats liés aux technologies *OSGi*, *RPM* et *Debian*. Leur étude nous a permis de de caractériser l'hétérogénéité des logiciels, et ce, à travers leurs similitudes et leurs différences.

Ainsi, ces formats partagent des similarités, relativement aux méta-données décrivant les fonctionnalités et les dépendances des modules. Par exemple, tous trois associent un module à un identifiant unique et permettent d'exprimer des dépendances sur d'autres modules. Certains formats autorisent l'expression de dépendances plus complexes. Par exemple, un *bundle OSGi* peut avoir des dépendances de natures diverses (ie. *bundle* ou *package*), celles-ci pouvant être associées à une version, des attributs ou encore un nom indéterminé.

L'hétérogénéité des technologies ne se matérialise pas seulement à travers le format d'assemblage. Ainsi, certaines technologies de logiciels, telles que *OSGi*, associent un cycle de vie à l'exécution, propre à chaque module.

La définition de processus de déploiement peut être construite à l'aide d'architecture à composants. Le modèle à composants *Fractal* est déjà largement utilisé dans le domaine du déploiement et bénéficie d'outils tels que *FScript*, facilitant la construction et la modification d'architectures. *Fractal* et *FScript* sont tous deux génériques et semblent offrir une solution à la représentation et au contrôle de logiciels hétérogènes, tels ceux que nous avons présentés.

15. Projet Edos : `http://www.edos-project.org`

Chapitre 3

Solutions existantes

Sommaire

Ce chapitre recense et analyse des travaux liés au problème du déploiement. Dans une première section, nous poserons des exigences (section 3.1), puis évaluerons ces travaux selon la satisfaction de celles-ci. Ces travaux sont divisés en trois parties : ceux qui définissent seulement un modèle (section 3.2), les outils pour le déploiement (section 3.3) et les intergiciels pour le déploiement (section 3.4).

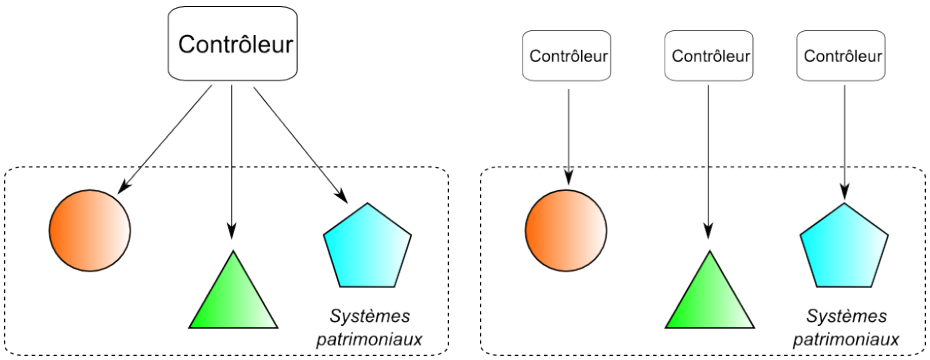
3.1 Exigences

Dans la problématique énoncée à la section 1.2, nous exposions notre attente d'une solution permettant d'automatiser le déploiement d'un logiciel et de son environnement à la juste taille. Dans ce contexte, nous reprenons les critères de caractérisation énoncés par Carzaniga et al. [16] (cf. la section 2.1) et posons des exigences pour chacun d'eux.

En premier lieu, nous souhaitons créer une description suffisamment fine des logiciels, englobant notamment leurs dépendances ainsi que leur contenu. Dans le chapitre précédent (section 2.3), nous décrivons sous quelles formes existaient leur hétérogénéité, notamment à travers celle de leur format d'assemblage. La capacité de décrire ces différents formats requiert une fine granularité dans la représentation du logiciel. L'exigence de fine granularité fait partie du critère de *modèle d'abstraction*, énoncé par Carzaniga et al. [16], et plus précisément de *modèle de produit*. La **fine granularité** de la représentation du logiciel constitue notre première exigence. Nous attendons que cette

représentation puisse évoluer au cours du cycle logiciel, afin de tenir compte des modifications pouvant être apportées aux logiciels. Celles-ci concernent, notamment, l'acquisition de nouvelles capacités ou de nouveaux besoins, ainsi qu'une modification de leur contenu.

Notre seconde exigence porte sur le **contrôle distribué et extensible** des procédures de déploiement. En effet, nous ne souhaitons pas que l'exécution des procédures de déploiement passe par un contrôleur unique (comme illustré sur la figure 3.1).



(a) Contrôle centralisé (b) Contrôle distribué

FIGURE 3.1 – Distribution du contrôle

L'utilisation d'un contrôleur unique présente en effet certains inconvénients :
- il est un intermédiaire inutile pour les procédures de déploiement ne concernant qu'un seul logiciel :
- il peut être un goulot d'étranglement si de nombreux logiciels sont à déployer.

Nous souhaitons que chaque logiciel ait un contrôleur de déploiement associé, permettant d'introspecter et de configurer le logiciel, notamment à l'exécution. Ce contrôleur peut être représenté au niveau du modèle d'abstraction du logiciel, si le modèle intègre une représentation du logiciel à l'exécution. Par exemple, une représentation d'un *bundle OSGi* doit proposer l'accès à son contrôle d'exécution ou encore, le contrôle de la résolution de ses imports.

Un contrôleur doit également garantir la cohérence du modèle. Ainsi, une modification de l'état du logiciel doit être représentée au niveau du modèle, que cette modification soit initiée par le contrôleur ou non.

Enfin, nous souhaitons que le modèle d'abstraction offre un contrôle extensible, c'est-à-dire, n'imposant pas d'opérations de contrôle et autorisant leur spécialisation selon la technologie du logiciel représenté.

L'hétérogénéité des logiciels est liée, en partie, à celle des plate-formes de déploiement. Selon la nature de celles-ci, un logiciel dispose d'une implémentation spécifique. Par exemple, la machine virtuelle *Java* est assemblée différemment selon le système d'exploitation visé.

La prise en compte de l'**hétérogénéité des plate-formes de déploiement** consti-

tue notre troisième exigence. Ce critère est lié au premier, dans le sens où lorsque le support de l'hétérogénéité est limité, la granularité de la représentation du logiciel peut l'être également. Cette exigence fait également partie du critère de *modèle d'abstraction* énoncé par Carzaniga et al. [16], mais s'applique plutôt au *modèle de site*.

Nous souhaitons, en particulier, que le modèle d'abstraction autorise l'utilisation d'une représentation commune (à gros grain) aux différentes implémentations d'un logiciel.

Une quatrième exigence concerne le modèle d'abstraction utilisé pour les *politiques de déploiement*. Nous souhaitons que la **résolution des dépendances** puisse être automatisée selon des politiques de résolution diverses, idéalement à n'importe quel moment du déploiement. Parmi les politiques de résolution, nous recherchons notamment une qui provisionne les dépendances manquantes.

Notre cinquième exigence concerne la *couverture du processus de déploiement*. Parmi les activités de déploiement citées par Carzaniga et al. [16], nous nous focalisons sur la **sortie du logiciel**. En effet, nous souhaitons que l'assemblage du logiciel effectué lors de la sortie du logiciel soit à la juste taille, influençant positivement la suite du déploiement.

Enfin, un assemblage doit bénéficier d'une modélisation, facilitant sa mise à jour, directement à partir des contrôleurs qui lui sont associés.

La liste suivante récapitule chacune de nos exigences (en gras) et les classe selon les critères proposés par Carzaniga et al. [16] :

- Modèle d'abstraction
  - ➢ Produit
    - **Fine granularité**
    - **Contrôle distribué et extensible**
  - ➢ Site
    - **Hétérogénéité**
  - ➢ Politique
    - **Résolution de dépendances**
- Couverture du processus de déploiement
  - ➢ **Sortie**

3.2 Un modèle pour le déploiement : OMG D&C

3.2.1 Présentation

La spécification *OMG D&C*[1] propose une définition des mécanismes requis pour le déploiement d'applications distribuées à base de composants [50]. Celle-ci introduit le modèle *PIM* (pour *Platform Independent Model*) couvrant tout le processus de déploiement. Ce modèle est indépendant de toute technologie de composant et regroupe une

1. Object Management Group : `http://www.omg.org`

représentation du composant logiciel, de l'environnement cible et de son exécution.

L'approche *MDA* [2] est utilisée pour l'implémentation de technologies de composant : le *PIM* est raffiné via des transformations successives vers un modèle spécifique, appelé *PSM* (pour *Platform Specific Model*), comme l'illustre la figure 3.2.

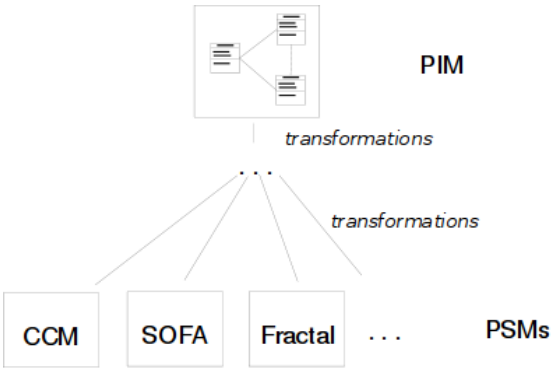


FIGURE 3.2 – Processus de transformation *MDA*

Une application est définie comme un composant logiciel, lequel peut être implémenté de deux manières :

Monolithique représentant une implémentation de composant ;

Assemblage représentant un ensemble de composants et d'interconnections, une interconnection étant un chemin de communication entre interfaces.

Un composant avec une implémentation monolithique correspond à un composant primitif, tandis qu'une implémentation d'assemblage caractérise un composite. Un *package* de composant contient des implémentations alternatives afin de supporter différentes technologies, permettant ainsi de fournir un seul *package* pour des systèmes hétérogènes (eg. des systèmes d'exploitation alternatifs).

Ces différentes entités sont représentées dans le modèle *D&C*. Celui-ci est divisé selon deux dimensions. La première dimension distingue le modèle de données et le modèle d'administration. La seconde dimension contient les éléments du modèle qui sont acteurs dans le processus de déploiement, à savoir les composants logiciels, l'environnement cible et l'exécution.

Package et composant sont notamment décrits dans le modèle de données de composants, correspondant à la partie du modèle de données relative aux composants logiciels. La figure 3.3 en présente un aperçu.

La classe *PackageConfiguration* décrit une configuration d'un package de composant, laquelle inclut une définition des besoins du *package*. Un besoin peut être exprimé soit à partir d'une URI, soit à partir d'une référence sur un composant installé dans un dépôt ou soit en sélectionnant des capacités, lesquelles sont définies dans la classe *ComponentImplementationDescription* représentant une implémentation spécifique d'une interface

2. OMG Model Driven Architecture : `http://www.omg.org/mda/`

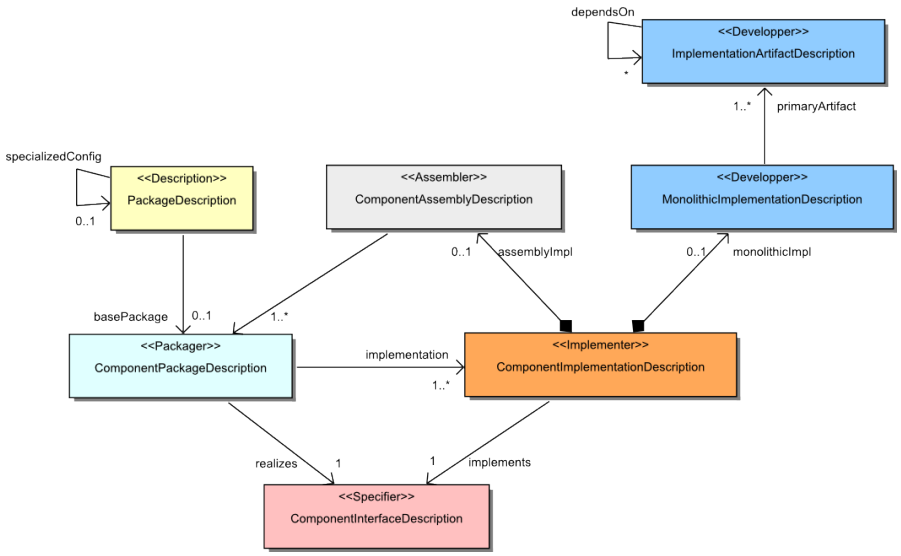


FIGURE 3.3 – Aperçu du modèle de données de composants

de composant. La classe *ComponentPackageDescription* décrit les implémentations alternatives d'une interface de composant. La classe *ComponentAssemblyDescription* contient les informations sur les sous-composants d'un composant d'assemblage. La classe *ComponentInterfaceDescription* représente une interface de composant. La classe *MonolithicImplementationDescription* décrit les artefacts associés à une implémentation monolithique. Un artefact est défini comme un morceau d'information physique, pouvant être utilisé ou produit par le processus de déploiement. La classe *ImplementationArtifactDescription* représente un artefact.

L'environnement cible du déploiement est appelé un domaine, lequel est composé de noeuds interconnectés à l'aide de ponts, représentant les liens réseaux.

Le modèle de données de l'exécution définit la notion de plan de déploiement. Celui-ci contient des informations sur les artefacts prenant part au déploiement, comment créer les instances des composants à partir de ces artefacts et où les instancier.

Le modèle d'administration de l'exécution définit un gestionnaire d'exécution, retournant un gestionnaire d'application pour un plan de déploiement donné. Le gestionnaire d'application représente une usine à déploiement permettant également d'exécutant une application. Le gestionnaire d'application retourne un objet permettant de contrôler le cycle de vie de l'application, c'est à dire de la démarrer ou de l'arrêter.

Dans le modèle *OMG D&C*, le déploiement est un processus commençant à partir de la mise à disposition au client. Le logiciel est alors déjà assemblé avec des méta-données le décrivant dans un *package*. Ce processus comprend les cinq étapes suivantes :

L'installation consiste au déploiement du logiciel assemblé dans un dépôt. Ce dépôt peut être distant à la fois du producteur et du consommateur du logiciel.

La configuration fournit des options de configuration par défaut pour une exécution

ultérieure.

La planification est une première étape dans la résolution des dépendances du logiciel. A partir des dépendances de celui-ci et des ressources disponibles sur l'environnement cible, la planification produit un plan de déploiement. Aucune modification à l'environnement cible n'est effectuée.

La préparation exécute le plan de déploiement produit durant la phase de planification, permettant ainsi au logiciel d'être prêt à l'exécution sur l'environnement cible.

L'exécution démarre l'application à base de composants.

La planification et la préparation ne peuvent être effectuées qu'à l'exécution selon le paradigme *just in time*.

3.2.2 Évaluation

Cette section évalue si *OMG D&C* valide chacune des exigences énoncées dans la section 3.1.

3.2.2.1 Modèle d'abstraction

Granularité Le modèle *D&C* n'impose aucune limitation de la granularité : la composition du logiciel, ainsi que ses fonctionnalités peuvent être représentées.

Contrôle Une application peut être démarrée ou arrêtée directement à partir d'un objet la représentant. Néanmoins, le modèle n'intègre pas la définition de nouveaux contrôleurs, ni la gestion de la cohérence du modèle.

Hétérogénéité Le modèle supporte l'hétérogénéité des technologies en permettant à un même composant d'être associé à différentes implémentations.

Politique de résolution Les étapes de planification et de préparation peuvent correspondre à la résolution des dépendances. Le modèle laisse libre cours à l'implémentation de politiques.

3.2.2.2 Processus de déploiement

Assemblage L'empaquetage du logiciel est une précondition du processus de déploiement et n'est donc pas couvert.

3.2.2.3 Bilan

La spécification *D&C* propose une large couverture du processus de déploiement, lequel reste flexible. Son modèle d'abstraction offre une fine granularité dans la représentation des logiciels et prend en compte le déploiement sur des technologies logicielles hétérogènes. La spécification ne définit pas de mécanisme de résolution mais le suggère dans la définition du processus de déploiement.

Cependant, la spécification n'indique pas comment étendre le contrôle des composants, afin d'introspecter et d'administrer les logiciels représentés. De plus, elle ne décrit

pas comment maintenir la cohérence du modèle lorsqu'un logiciel est modifié. Enfin, le processus de déploiement n'inclut pas la sortie du logiciel (et donc son assemblage).

La spécification *D&C* adresse de nombreuses problématiques du déploiement, mais ne répond pas à certaines exigences, importantes à nos yeux : le contrôle des applications et leur assemblage sur le site de production.

3.3 Outils spécialisés dans l'assemblage

3.3.1 Apache Maven

3.3.1.1 Présentation

Apache Maven[3] est un outil pour la gestion de projet logiciel regroupant un modèle objet de projet, un ensemble de standards, un cycle de vie de projet, un système de gestion des dépendances et une logique pour l'exécution de plugins selon le cycle de vie.

Représentation du logiciel Le modèle objet de projet définit implicitement un modèle à composant grâce aux caractéristiques suivantes :

- Un projet définit les projets requis ;
- Un projet offre une vue globale dans laquelle les relations entre sous-projets sont explicites.

Un projet est représenté par un artefact. Cet artefact appartient à un groupe et a un numéro de version associé. Un groupe permet d'organiser les projets et constitue une partie du nom de l'artefact. Ce dernier est basé sur le schéma [groupId]:[artifactId]:[version]. *Maven* définit ainsi un référentiel de nommage unique. Un logiciel représenté par un artefact peut être retrouvé à partir de son nom et d'un dépôt.

Représentation du déploiement À la différence d'autres outils tels que *Make* ou *Ant*, *Maven* adopte une approche pour la définition du déploiement plus déclarative que procédurale. Si *Make* et *Ant* ne fournissent ainsi pas une définition abstraite du processus de déploiement, ceux-ci sont de puissants outils permettant de définir des déploiements spécifiques à partir d'un langage de scripts. *Maven* propose plutôt d'associer des actions (des *plugins*) à chaque phase du processus de déploiement. Le cycle de vie du déploiement peut varier selon les *plugins*[4]. Par exemple le cycle de vie par défaut pour la construction de logiciel comprend vingt et une phases, tandis que celui pour la construction de site Internet en comprend seulement quatre, lesquelles sont différentes.

L'invocation du processus de déploiement s'effectue à partir d'un *projet*, d'un *plugin* et d'un *goal* donné. Un *goal* correspond à l'ultime phase du cycle de vie à exécuter et est une information facultative car chaque *plugin* en utilise un par défaut.

3. Apache Maven Project : `http://maven.apache.org/`
4. `http://www.sonatype.com/books/maven-book/reference/lifecycle.html`

Maven peut construire des assemblages logiciel via le plugin *maven-assembly-plugin*. Ce plugin requiert des informations supplémentaires telles que les projets ou les fichiers à assembler (ceux-ci n'appartenant pas à un projet).

Maven fournit un résolveur de dépendances supportant deux types de version : les versions stables et les versions de développement. Lorsqu'une dépendance utilise une version de développement, le résolveur vérifie l'existence d'une nouvelle version dans les dépôts connus.

3.3.1.2 Évaluation

Cette section évalue si *Apache Maven* valide chacune des exigences énoncées dans la section 3.1.

Modèle d'abstraction

Granularité La granularité des logiciels est limitée : soit un artefact, soit un fichier. Néanmoins, il reste possible de construire des plugins supportant d'autres types d'éléments. Par exemple, le projet *Tycho*[5] propose de tenir compte des métadonnées *OSGi* et *Eclipse*, pour la construction de logiciels par *Maven*. On notera toutefois qu'il s'agit d'une solution spécifique.

Contrôle La représentation du logiciel ne permet pas de reconfiguration. Seuls les plugins effectuent des opérations de reconfiguration. Le contrôle est donc centralisé.

Hétérogénéité Un artefact *Maven* peut appartenir à n'importe quelle technologie logicielle.

Politique de résolution Un plugin peut invoquer le résolveur. Cependant la politique de résolution est imposée.

Processus de déploiement

Assemblage La tâche d'assemblage est assurée par *maven-assembly-plugin* et est donc intégrée au processus de déploiement. Néanmoins, un assemblage est considéré comme un banal artefact *Maven* et ne peut être mis à jour, à partir de sa représentation abstraite.

Bilan *Maven* est un outil hautement extensible grâce à son mécanisme de plugins.

Cependant, par défaut *Maven* ne permet pas d'exprimer des dépendances autres que sur des artefacts ou des fichiers. Le contrôle des logiciels est de plus centralisé. Enfin, un assemblage n'a pas de représentation, ne facilitant pas sa mise à jour.

Maven ne répond que très partiellement à nos exigences relatives au modèle d'abstraction et ne prend pas en compte l'administration de logiciels, à l'exécution.

5. Sonatype Tycho : `http://tycho.sonatype.org/`

3.3.2 Eclipse Buckminster

3.3.2.1 Présentation

Eclipse Buckminster[6] est un ensemble d'outils pour automatiser les processus de construction, d'assemblage et de déploiement (BA&D) lors du développement de logiciels complexes et distribués.

Eclipse Buckminster définit un modèle à composants, dans lequel un composant est une abstraction d'une unité de contenu nommée, versionnée et typée dans un système logiciel. Une unité de contenu est chargée par un *ComponentReader* qui interprète et traduit ses méta-données en une spécification de composant (*CSPEC*). *Eclipse Buckminster* fournit des *ComponentReader* pour les systèmes suivants :

Eclipse plugin, feature, product, fragments

OSGi bundle

Maven Maven POM (version 1 et 2)

Buckminster CSPEC et Component Specification Extension (CSPEX).

Les composants peuvent comporter des attributs et des dépendances sur d'autres composants. Un attribut représente statiquement ou dynamiquement une collection de fichiers. Un attribut statique est appelé un *artefact*, tandis qu'un attribut dynamique est appelé une *action*. Les implémentations des actions sont fournies par des *acteurs*. Par exemple *Eclipse Buckminster* délivre les acteurs suivants :

Java Compilation, création de *JAR*, etc.

PDE Construction de *bundles*, *features*, *fragments* et *products*, assemblage, signature

p2 Construction de dépôt

Général Récupération de fichier et exécution de commandes systèmes

ANT Invocation de taches *ANT*.

Il est possible de spécifier la génération d'autres composants *generators*, à partir de la définition d'un composant.

Une *RMAP* est un dépôt à composant. La recherche d'un composant dans un dépôt s'effectue à l'aide d'une *CQUERY*. La résolution de ce composant génère un *BOM* (*Bill Of Materials*), contenant l'ensemble des dépendances de celui-ci. Une étape de matérialisation peut être associée à la requête à l'aide d'une spécification de matérialisation (un fichier *MSPEC*), la matérialisation consistant en la mise à disposition localement de composants par un *Materializer*.

Buckminster propose trois *Materializers* préexistants :

Système de fichier Copie vers un emplacement dans le système de fichiers ;

p2 Création d'une plate-forme *Eclipse* (*Target Platform*) sous forme d'un dépôt d'artefacts *p2* ;

Workspace Importe les sources et les binaires d'un projet dans le *workspace*.

6. Eclipse Buckminster : `http://www.eclipse.org/buckminster/`

Buckminster est principalement destiné et utilisé pour l'assemblage d' applications basées sur *Eclipse*. Néanmoins, il est possible d'utiliser les composants *Buckminster* avec n'importe quel type de projet. Ainsi, un processus de déploiement dans *Buckminster* peut être défini comme une séquence d'*actions* initiée par une *CQUERY* (comme illustré sur la figure 3.4).

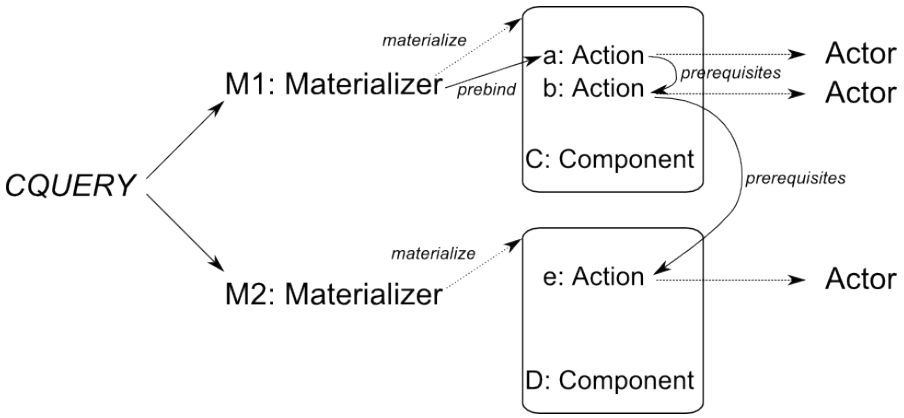


FIGURE 3.4 – Un processus de déploiement dans Buckminster

Une requête *CQUERY* sélectionne les composants à installer. Le résultat de cette requête est copié localement par les *Materializers*. Ces derniers peuvent invoquer des actions sur les composants. Une action est implémentée par un acteur et peut impliquer l'exécution d'autres actions.

3.3.2.2 Évaluation

Cette section évalue si *Eclipse Buckminster* valide chacune des exigences énoncées dans la section 3.1.

Modèle d'abstraction

Granularité *Buckminster* permet d'exprimer des dépendances entre composants mais n'a pas de notion de composite. De plus, une dépendance ne peut être définie que sur un composant. Par exemple, il n'est pas possible de définir une *CQUERY* pour un *package Java*.

Contrôle Le contrôle du déploiement est centralisé : *Buckminster* s'exécute via une interface *Eclipse* ou en ligne de commande.

Hétérogénéité *Buckminster* privilégie les applications *Eclipse*, mais définit un mécanisme extensible pour l'ajout de nouveaux systèmes.

Politique de résolution Celle-ci s'effectue à l'aide d'une *CQUERY*. La manière dont les dépendances sont récupérées peut être personnalisé en spécifiant un *MSPEC*.

Processus de déploiement

Assemblage C'est une des fonctions principales de *Buckminster*. Cependant, aucun mécanisme n'est spécifié pour mettre à jour un assemblage.

Bilan *Buckminster* offre un mécanisme intéressant de création de composant à partir de logiciel existant. Cependant, les composants créés offrent une représentation à gros grain et n'offre aucun contrôle sur le logiciel.

3.4 Intergiciels pour le déploiement

3.4.1 Software Dock

3.4.1.1 Présentation

Software Dock [38, 39, 36] est un framework de déploiement logiciel distribué, basé sur des agents.

Cycle de vie Software Dock supporte toutes les activités d'un processus de déploiement définies par Carzaniga et al. (1998) (cf. la sous-section 2.1). En outre, il en définit une nouvelle, la *reconfiguration*, qui correspond à la sélection d'une nouvelle configuration pour un système logiciel déjà installé. Ces activités sont également réparties sur le site de production (pour la sortie et la fin de support) ou sur le site client (pour l'installation, l'activation, la désactivation, la reconfiguration, la mise à jour, l'adaptation et la désinstallation).

Architecture L'architecture de *Software Dock* (voir figure 3.5) est composé de quatre acteurs majeurs :

- Le *field dock* est un serveur résidant chez le consommateur du logiciel, celui-ci maintenant un registre de la configuration locale ;
- Le *release dock* est un serveur résidant chez le producteur du logiciel ;
- L'*agent* est un programme exécutable effectuant des taches du processus de déploiement pour le compte des producteurs et consommateurs ;
- Le *WAM/E*, système d'échange de messages et d'évènements pour les agents.

Définition des processus de déploiement Afin de rendre les définitions des processus de déploiement plus génériques et conformément au critère du modèle d'abstraction vu précédemment (cf. la sous-section 2.1), *Software Dock* utilise un modèle appelé *Deployable Software Description* (*DSD*).

Ce modèle est implémenté sous forme de collections de propriétés, partagées selon les cinq classes d'information sémantiques suivantes [40] :

- La *configuration* décrit la configuration d'un système logiciel, ainsi que les ressources (eg. des interfaces ou des services) fournit par celui-ci ;

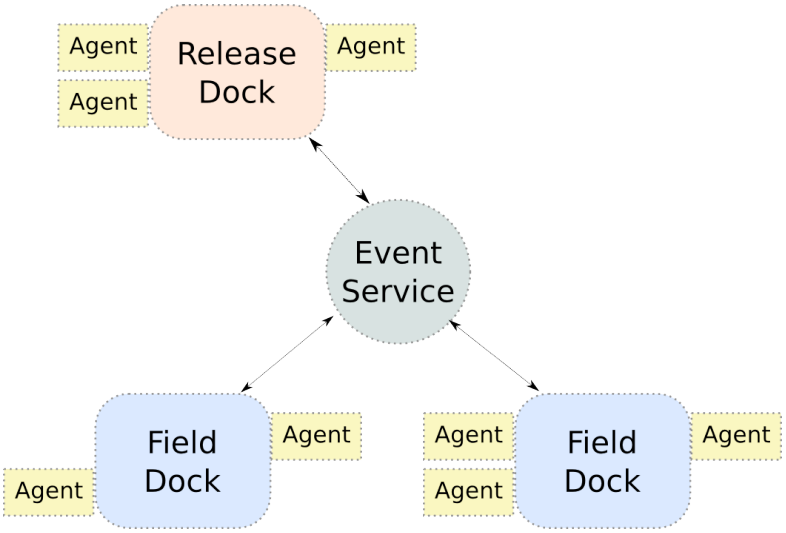


FIGURE 3.5 – Architecture de Software Dock

- Les *assertions* décrivent des contraintes critiques (pour le processus de déploiement) sur les ressources du client (eg. relatives au matériel ou au système d'exploitation) ;
- Les *dépendances* décrivent des contraintes sur les ressources du client, lesquelles sont non bloquantes pour le processus de déploiement (eg. l'absence de logiciels nécessaires à l'exécution) ;
- Les *artefacts* décrivent les éléments réels et physiques constituants le logiciel (eg. des fichiers) ;
- Les *activités* décrivent toutes les activités spécialisées qui n'apparaissent pas les processus standards de déploiement (eg. des mises à jour de base de données).

Un agent manipule une spécification *DSD* afin d'effectuer une tâche du processus de déploiement. Par exemple, l'agent d'installation déploie une nouvelle configuration d'un logiciel sur un site client. En fonction de l'état du site client, l'agent d'installation télécharge depuis le *release dock* les artefacts requis.

3.4.1.2 Évaluation

Cette section évalue si *Software Dock* valide chacune des exigences énoncées dans la section 3.1.

Modèle d'abstraction

Granularité *Software Dock* permet d'exprimer les dépendances des logiciels afin de décrire un système. Cependant, le contenu d'un logiciel n'est pas réifié.

Contrôle *Software Dock* bénéficie d'une architecture distribuée, néanmoins la représentation du logiciel ne semble pas offrir d'interface de contrôle et une reconfiguration du côté client se fait nécessairement via son composant *FieldDock*.

Hétérogénéité *Software Dock* ne distingue pas les technologies logicielles des artefacts.
Politique de résolution La résolution de dépendances est automatisée par les agents.

Processus de déploiement

Assemblage *Software Dock* ne semble pas prendre en charge l'assemblage du logiciel avant sa sortie.

Bilan *Software Dock* couvre un large spectre du déploiement en intégrant la sortie du logiciel. Le déploiement est adapté au besoin du client. Cependant, si l'utilisation d'artefact permet de décrire globalement un système à déployer, il n'offre pas la granularité nécessaire pour assembler un logiciel.

3.4.2 Application Buildbox

3.4.2.1 Présentation

Yu David Liu et Scott F. Smith proposent *Application Buildbox*, un canevas pour le déploiement de composants logiciels [49]. Celui-ci est bâti au dessus du modèle à composants *Assemblages*[48] et adresse le cycle de vie complet de l'évolution d'une application, quelque soit sa plate-forme et son vendeur. La validité du processus de déploiement est prouvée en se basant sur une spécification formelle.

Le modèle à composant Les composants logiciels sont dénommés assemblages [48]. Un assemblage est une unité de code nommée, versionnée, indépendamment livrable et indépendamment déployable (voir la représentation d'un assemblage sur la figure 3.4.2.1). Le modèle distingue deux types d'interfaces : *mixers* et *pluggers*. Les interfaces de type *mixer* caractérisent une liaison entre composants appelée *mixing*, celle-ci étant créée avant le chargement du composant. Au contraire, les interfaces de type *plugger* caractérisent une liaison entre composants appelée *plugging*, celle-ci étant créée après le chargement du composant. Une interface est bidirectionnelle, dans le sens où elle peut à la fois importer et exporter des fonctionnalités. Le format des fonctionnalités est libre (eg. une fonction, une classe...). Une interface est nommée mais n'a pas de version propre.

Une application *BuildBox* est constituée d'un ensemble de composants *Assemblage* reliés par leurs interfaces (voir la figure 3.4.2.1). Une exécution d'application est composée d'exécutions d'assemblage (voir la figure 3.4.2.1). Une exécution d'assemblage est un assemblage dont le code a été chargé et dont les interfaces de type *plugger* peuvent être liées.

Cycle de vie Le canevas modélise les actions fondamentales impliquées dans le processus de déploiement.

Création La première étape dans le cycle de vie d'un composant est sa création. Celui-ci est créé à partir d'un nom, d'une version, d'interfaces et d'une implémentation.

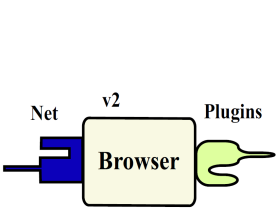


FIGURE 3.6 – Assemblage

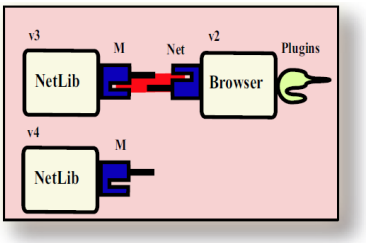


FIGURE 3.7 – Application

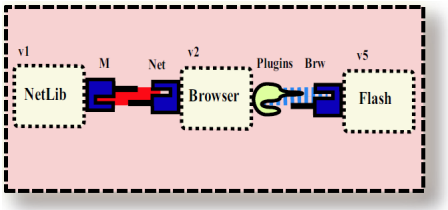


FIGURE 3.8 – Exécution

Assemblage L'assemblage consiste à la création des liaisons de type *mixing*.

Livraison La livraison d'un composant est effectuée en lui ajoutant les informations nécessaires à l'installation, telles que les versions des dépendances. La structure de données obtenue est appelée un *packaged assemblage*.

Installation L'installation d'un *packaged assemblage* retourne un composant connecté à ses dépendances.

Mise à jour Une mise à jour est caractérisée par l'échange d'une liaison de dépendance vers un composant fournissant une version plus récente.

Déploiement dynamique Lorsqu'un composant est démarré, des liaisons peuvent être créées mais seulement depuis ses interfaces de types *plugger*.

Un composant peut être obtenu soit en le créant « from scratch », soit à partir d'un *packaged assemblage*.

3.4.2.2 Évaluation

Cette section évalue si *Application Buildbox* valide chacune des exigences énoncées dans la section 3.1.

Modèle d'abstraction

Granularité Le modèle à composants spécifiés permet à une interface de représenter n'importe quelle capacité ou besoin. Le modèle supporte la composition hiérarchique, mais uniquement sur le site de développement et si le composite est une unité de déploiement indépendante. Sur le site de déploiement, un composant *Assemblage* ne peut être un composite.

Contrôle Les composants exposent une interface de contrôle de leur cycle de vie. Les auteurs ne spécifient pas si ce contrôle peut être étendu.

Hétérogénéité Le canevas supporte tout type de technologie logicielle.

Politique de résolution Aucune politique de résolution des dépendances n'est spécifiée.

Processus de déploiement

Assemblage L'assemblage de composant est intégré au cycle de vie du déploiement.

Bilan Les auteurs proposent un modèle à composants permettant de représenter n'importe quel type de logiciel. Cependant, les contraintes sur la composition hiérarchique sont fortes. De plus, la nécessité de distinguer les interfaces de type *mixer* et *plugger* est contestable : celle-ci introduit un couplage avec le cycle de vie spécifié (seul un *plugger* est utilisable après le chargement). Or, chaque technologie peut introduire son propre cycle de vie associée à ses propres politiques de chargements dynamiques.

3.4.3 DeployWare

3.4.3.1 Présentation

DeployWare[7] [4, 29] est un canevas logiciel basé sur le modèle à composant *Fractal* (voir l'introduction au modèle dans la section 2.2.1) permettant de déployer des systèmes logiciels hétérogènes et distribués.

DeployWare fournit principalement un langage permettant de décrire un processus de déploiement, une machine virtuelle exécutant des processus de déploiement pour une description dans ce langage et une bibliothèque de composants de déploiement. *DeployWare* définit un méta-modèle capturant les concepts abstraits du déploiement. La figure 3.9 présente celui-ci.

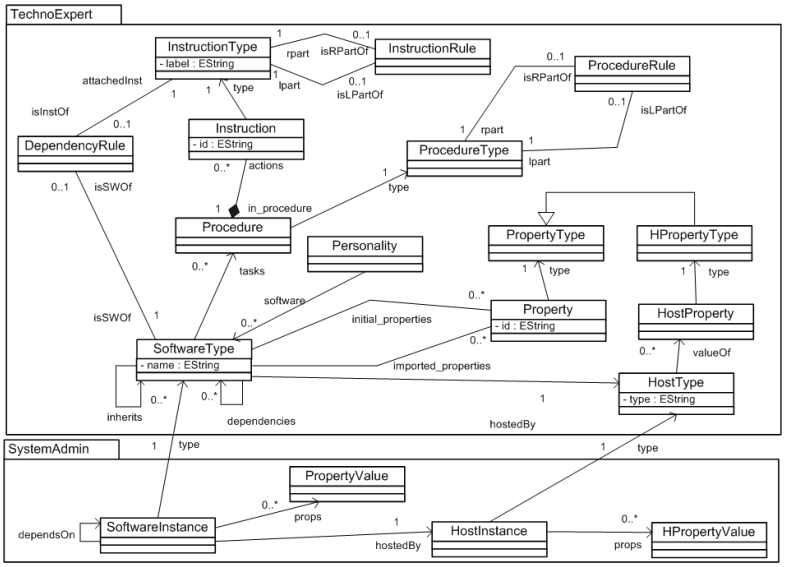


FIGURE 3.9 – Méta-modèle de *DeployWare*

7. DeployWare : `http://fdf.gforge.inria.fr/`

Le méta-modèle est partagé entre deux paquetages séparant les préoccupations : *TechnoExpert* permet de spécifier le déploiement d'un logiciel et *SystemAdmin* permet de construire des instances de logiciels ainsi que de les configurer.

Un logiciel d'une technologie donnée est représenté par l'interface SoftwareType. Un logiciel peut avoir des dépendances vers un autre logiciel et être associé à des procédures de déploiement (lesquelles sont représentées par l'interface Procedure). La technologie du logiciel est représentée par l'interface Personality. Une instance de logiciel est représentée par l'interface SoftwareInstance et est associé à un hôte, lequel est représenté par l'interface HostInstance.

La machine virtuelle de *DeployWare* construit des composants logiciels représentant les logiciels à déployer. Un exemple de composant logiciel est donné par la figure 3.10.

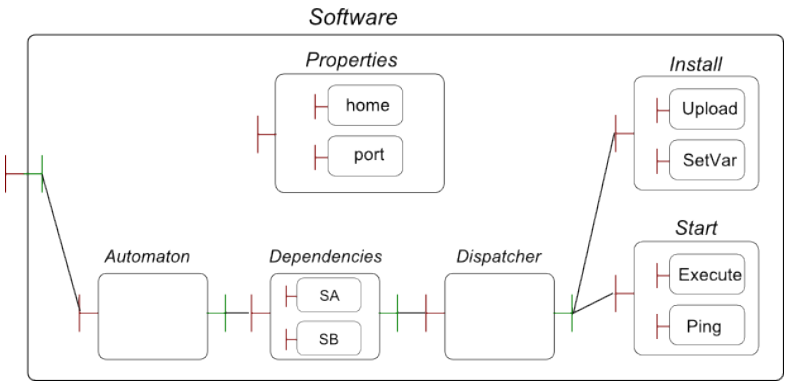


FIGURE 3.10 – Exemple de composant logiciel DeployWare

Le composant représentant le logiciel déployable est un composite contenant notamment :

le composite *Properties* contenant les propriétés de configuration ;

le composite *Automaton* représentant l'automate de déploiement ;

le composite *Dependencies* contenant des références vers les composants correspondant aux dépendances du logiciel décrit ;

le composite *Dispatcher* orchestrant l'appel de procédures sur les dépendances ;

d'autres composites représentant les procédures de déploiement.

Un composant logiciel expose une interface serveur permettant de contrôler le déploiement du logiciel.

3.4.3.2 Évaluation

Cette section évalue si *DeployWare* valide chacune des exigences énoncées dans la section 3.1.

Modèle d'abstraction

Granularité La représentation du logiciel se fait à gros grains. Par exemple, *DeployWare* n'est pas destiné à représenter le contenu d'une archive.

Contrôle Les opérations de déploiement sont directement accessibles sur le composant représentant le logiciel. Le contrôle est extensible. Cependant, la gestion de la cohérence du modèle n'est pas décrite.

Hétérogénéités Le support des systèmes est apporté par les personnalités.

Politique de résolution La machine virtuelle de *DeployWare* orchestre le déploiement en respectant les dépendances de chaque logiciel. L'automate orchestrant le déploiement peut être modifié, de part sa nature de composant *Fractal*.

Processus de déploiement

Assemblage *DeployWare* définit une bibliothèque de composants de déploiement opérant sur les fichiers. Néanmoins, l'absence de modélisation fine du logiciel limite le potentiel de construction d'un assemblage.

Bilan *DeployWare* est un outil hautement extensible pour le déploiement d'application patrimoniale. Il permet d'orchestrer le déploiement des dépendances entre logiciels quelques soient leur technologie. Cependant, *DeployWare* ne propose pas une granularité assez fine de la représentation d'un logiciel, ne le destinant pas à l'assemblage de logiciels.

3.4.4 Nix

3.4.4.1 Présentation

Nix[8] se définit comme un gestionnaire de paquets purement fonctionnel[27]. *Nix* a pour objectif de proposer un gestionnaire de paquets sûr et flexible. Un déploiement de logiciel flexible doit permettre d'installer différentes versions d'un même logiciel, de déployer à la fois des sources et des binaires sans politique imposée. Un déploiement sûr doit prendre en compte les dépendances du logiciel, respecter la cohérence des logiciels en évitant les collisions lors de l'installation de multiples versions.

L'installation selon *Nix* est représentée par une fonction dont les valeurs sont des paquets. La mise à jour et la désinstallation sont également représentées à l'aide de fonctions. Celles-ci n'ont pas d'effet de bord et les paquets installés ne peuvent donc pas être cassés par l'usage de fonctions sur d'autres paquets. *Nix* maintient une vue des paquets installés pour chaque utilisateur, ces derniers pouvant installer des versions différentes.

Un paquet est représenté sous forme de composant, lequel est décrit à partir d'expressions *Nix* écrites en langage fonctionnel. La définition d'un composant s'effectue à

8. Nix : `http://nixos.org/`

l'aide d'une fonction paramétrée par les dépendances du composant. La fonction *mkDerivation* permet notamment la construction d'un composant à partir d'un script système (voir l'exemple pour le programme *GNU Hello*[9] dans le listing 3.1).

```
{stdenv, fetchurl, perl}:
stdenv.mkDerivation {
  name = "hello-2.1.1";
  builder = ./builder.sh;
  src = fetchurl {
    url = ftp://ftp.nluug.nl/pub/gnu/hello/hello-2.1.1.tar.gz;
    md5 = "70c9ccf9fac07f762c24f2df2290784d";
  };
  inherit perl;
}
```

Listing 3.1 – Expression Nix définissant un composant pour GNU Hello

Le composant *hello* a des dépendances sur *stdenv*, *fetchurl* et *perl*. Le corps de l'expression est composé d'un ensemble d'attributs.

La composition s'exprime également à partir d'expression *Nix* : elle consiste en la définition des attributs correspondant aux arguments représentant les dépendances des composants. Un exemple est donné par le listing 3.2.

```
rec {
  hello = (import ../applications/misc/hello/ex-1 2) {
    inherit fetchurl stdenv perl;
  };
  perl = (import ../development/interpreters/perl) {
    inherit fetchurl stdenv;
  };
  fetchurl = (import ../build-support/fetchurl) {
    inherit stdenv; ...
  };
  stdenv = ...;
}
```

Listing 3.2 – Expression Nix exprimant la composition avec GNU Hello

Dans cet exemple, les attributs *stdenv*, *fetchurl* et *perl* sont définis, satisfaisant ainsi les dépendances du composant *hello*.

Un paquet installé a une entrée associée dans le dépôt *Nix*. Celui-ci associe chaque paquet avec les fichiers et dossiers le constituant, ainsi que ses dépendances sur d'autres paquets (comme l'illustre la figure 3.11).

Ce gestionnaire de paquets a été utilisé pour bâtir la distribution Linux *NixOS* [28].

3.4.4.2 Évaluation

Cette section évalue si *Nix* valide chacune des exigences énoncées dans la section 3.1.

9. GNU Hello : `http://www.gnu.org/software/hello/hello.html`

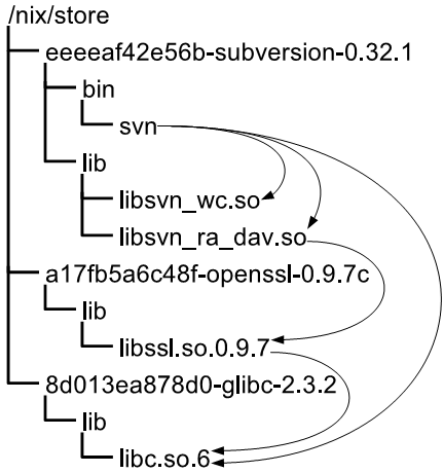


FIGURE 3.11 – Le dépôt Nix

Modèle d'abstraction

Granularité La granularité est limitée à la représentation de paquets et de fichiers.

Contrôle Le contrôle est centralisé : *Nix* ne s'utilise que dans un shell.

Hétérogénéité *Nix* est destiné aux systèmes basé sur *Unix*.

Politique de résolution *Nix* permet d'installer la fermeture transitive des dépendances d'un paquet.

Processus de déploiement

Assemblage L'assemblage peut être réalisé à l'aide de la fonction de dérivation.

Bilan *Nix* propose une approche originale, basée sur un langage fonctionnel, pour le déploiement de logiciel. *Nix* offre une grande flexibilité dans le déploiement, tout en adaptant celui-ci aux dépendances nécessaires.

Cependant *Nix* est destiné au monde Unix et ne couvre pas donc nos besoins.

3.4.5 Jade

3.4.5.1 Présentation

Jade est un canevas pour l'administration autonome d'infrastructures patrimoniales [21]. *Jade* permet de construire des systèmes autonomes basés sur l'architecture définie par Kephart et al.[42].

Le canevas fournit un environnement pour la représentation du système et la construction de gestionnaires autonomes. *Jade* offre une bibliothèque de gestionnaires autonomes spécialisés dans divers domaines : auto-réparation [57], auto-optimisation [60] ou auto-protection [18].

Le modèle à composants *Fractal* est utilisé pour la représentation du système administré (voir la figure 3.12) et offre une administration basée sur les composants.

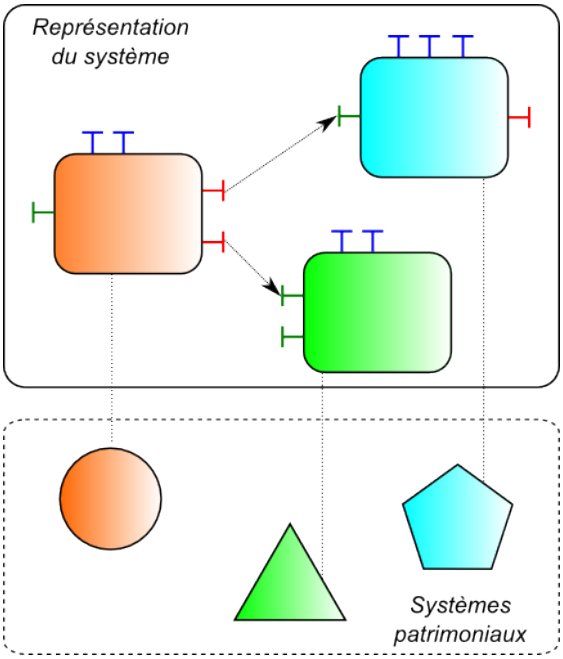


FIGURE 3.12 – Représentation de systèmes patrimoniaux dans *Jade*

Un élément administré est représenté sous forme de composant *Fractal* et dispose d'une interface d'administration uniforme, laquelle est définie par les contrôleurs du composant. Les éléments administrés peuvent être des logiciels patrimoniaux (eg. *Apache Tomcat*[10] ou *MySQL*)[11] ou des architectures logicielles (eg. une architecture multi-tiers *Java EE*). Les logiciels patrimoniaux sont encapsulés par des composants *Fractal* primitifs sous forme de boîtes noires. Les composites *Fractal* encapsulent les architectures logicielles. Les contrôleurs *Fractal* sont implémentés spécifiquement selon l'élément administré. Par exemple, le contrôleur de liaison d'un composant encapsulant un serveur *Apache HTTP*[12] permet d'associer celui-ci avec des composants encapsulant *Tomcat* et de le répercuter au niveau de la configuration du serveur. Le contrôleur d'attribut permet de modifier des éléments de la configuration tel que le port du serveur. Enfin, le contrôleur de cycle de vie permet de le démarrer et de l'arrêter.

Jade met à disposition des gestionnaires autonomes un certain nombre de services communs dont le gestionnaire de déploiement. Celui-ci a en charge l'exécution des plans de déploiement. Le gestionnaire de déploiement a notamment en charge d'installer l'implémentation des composants. Cette implémentation est représentée sous forme de com-

10. Apache Tomcat : `http://tomcat.apache.org/`
11. MySQL : `http://www.mysql.com/`
12. Apache HTTP Server : `http://httpd.apache.org/`

posant *Fractal*, appelé *module* [44]. Un composant *Fractal* est lié à son module par une dépendance non fonctionnelle. Un module peut avoir des dépendances sur d'autres modules, lesquelles sont représentées par des liaisons entre module. Les dépendances entre module font partie de la représentation des systèmes patrimoniaux. Deux versions de la mise en œuvre des modules sont proposées.

Dans une première version, la définition d'un module n'adresse pas la problématique du chargement de l'implémentation, celle-ci étant déléguée à un système de module patrimoniale. Cette version est destinée à un gestionnaire de déploiement pour systèmes de module patrimonial, *Jade* ciblant l'administration autonome de logiciel patrimonial. *Jade* définit une implémentation des modules à l'aide de *bundles OSGi*, les modules ayant des dépendances de *package Java*. Cette représentation était construite sur des bases instables. En effet, la plate-forme *OSGi* contient également une définition de module. Dans celle-ci les liaisons sont représentées entre *bundles* ou *packages*. La plate-forme *OSGi* dispose d'un résolveur de dépendance autonome créant les liaisons entre modules *OSGi*. Or, ce résolveur peut avoir avoir des choix contradictoires avec les liaisons spécifiées entre les modules *Fractal* et plus généralement la description architecturale. Ainsi, rien n'impose au résolveur de choisir des *packages* provenant d'un *bundle* différent de celui représenté par le module *Fractal*. La figure 3.13 illustre un tel cas de figure.

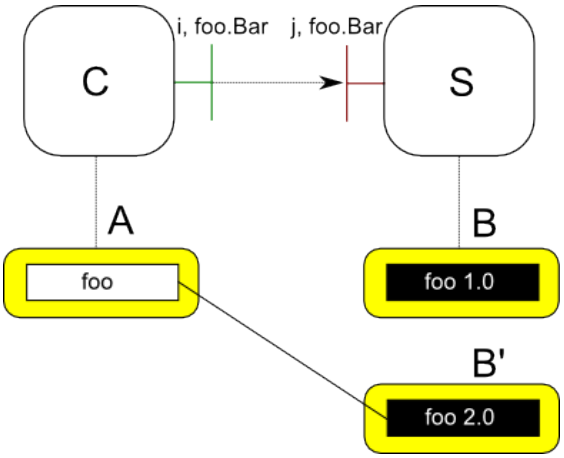


FIGURE 3.13 – Incohérence de la représentation du système dans Jade

Sur cette figure, un rectangle blanc désigne un import de *package Java* et un rectangle noir un export de *package*. Le composant C a une dépendance sur le composant S. Le composant C est implémenté par le *bundle* A et importe le *package* foo. Le composant S est implémenté par le *bundle* B, lequel exporte le *package* foo en version 1.0. Puisque l'interface i du composant C est liée à l'interface j de S, le type d'interface $foo.Bar$ devrait être le même et devrait être défini dans le *bundle* B. Or, le résolveur *OSGi* choisit la version du *package* la plus récente pour résoudre un import (en l'absence d'une spécification de version) et privilégie donc le type $foo.Bar$ fourni par le *package* foo en version 2.0 du *bundle* B, conduisant à une erreur à l'exécution. La description

architecturale n'est pas en cohérence avec les liens effectifs entre *packages*.

Le provisionnement de modules utilise *OBR*. *OBR* embarque son propre résolveur de dépendances ce qui est une seconde source de conflit avec le gestionnaire de déploiement de *Jade*.

Cette version des modules *Fractal* est destinée à supporter des systèmes de module patrimonial. Néanmoins, la définition donnée des modules reste minimaliste. Celle-ci ne propose pas une définition de module composite, impose un format pour la description des dépendances entre modules et ne décrit pas comment intégrer des systèmes patrimoniaux hétérogènes. En outre, cette version compose une représentation du système pouvant être cassée par le résolveur *OSGi*, ce qui a conduit à la définition d'une seconde version de module *Fractal*.

La seconde version propose d'abandonner le support de système de module patrimonial au profit d'un nouveau système de module. Celui-ci s'inspire d'*OSGi* dans le sens où les chargeurs de classes des modules n'ont pas une relation hiérarchique, mais basée sur la délégation. Les liaisons entre modules ont pour granularité le *package Java*. Un import d'un *package* se caractérise dans ce modèle par la délégation du chargement d'une classe appartenant à ce *package* au chargeur de classe du module exportant celui-ci. Les modules *Fractal* exposent pour chaque *package* exporté une interface fonctionnelle ayant pour type IExportPackage, laquelle offre des opérations permettant de charger des classes ou d'accéder aux ressources du *package*. Les liaisons sont créées par le seul gestionnaire de déploiement. Cette version résout le problème de coordination entre le gestionnaire de déploiement et la plate-forme *OSGi*, mais écarte le support des logiciels patrimoniaux.

3.4.5.2 Évaluation

Cette section évalue si *Jade* valide chacune des exigences énoncées dans la section 3.1.

Modèle d'abstraction

Granularité Les dépendances sont représentées entre éléments administrés. Cependant les logiciels patrimoniaux (tels que les modules) restent des boites noires encapsulés dans des composants primitifs.

Contrôle Chaque élément administré expose une interface de contrôle via les contrôleurs *Fractal* du composant l'encapsulant : le contrôle est donc distribué et peut être étendu. *Jade* maintient la cohérence du modèle en mettant à jour le système représenté.

Hétérogénéité L'hétérogénéité des logiciels patrimoniaux est supportées mais ce support reste limité pour le gestionnaire de déploiement. La première version de celui-ci propose une approche supportant des systèmes patrimoniaux hétérogènes, mais ne décrit pas comment les intégrer.

Politique de résolution Une politique de résolution est spécifiée pour les modules *Fractal*. La première version utilise *OBR* pour le provisionnement de module. La seconde version utilise un dépôt de modules spécifique.

Processus de déploiement

Assemblage *Jade* n'est pas destiné à l'assemblage de logiciels.

Bilan *Jade* montre l'apport des architectures à composants pour l'administration de systèmes patrimoniaux. Le modèle à composants *Fractal* est pleinement adapté à la représentation de systèmes patrimoniaux. La définition d'interface d'administration uniforme simplifie le déploiement de logiciel hétérogènes.

Le retour d'expérience sur l'administration de plate-forme *OSGi* a montré l'intérêt d'intégrer une représentation assez fine des modules dans la représentation du système et de gérer la cohérence entre les décisions du résolveur de dépendances *OSGi* avec celle du gestionnaire de déploiement de *Jade*.

3.4.6 DAnCE

3.4.6.1 Présentation

DAnCE (pour *Deployment And Configuration Engine*[13]) est un environnement de déploiement dédié aux composants adressant des contraintes de qualité de service [23, 58]. Le placement des composants, la consommation des mémoires et la charge des processeurs sont des exemples de contraintes supportées. *DAnCE* cible notamment le déploiement de systèmes logiciels distribués, temps réels et embarqués (systèmes *DRE*). Les systèmes *DRE* sont bâtis à l'aide de l'intergiciel *CIAO* [66], lequel implémente le standard *OMG Lightweight CCM*[14]. L'infrastructure de déploiement est elle-même implémentée sous forme d'objets *CORBA*. *DAnCE* fournit une plate-forme compatible avec la spécification *OMG D&C* décrite dans la section 3.2, en y étendant notamment le modèle de données avec des préoccupations additionnelles, telles que des besoins en qualité de service temps réel. Ce modèle de données est utilisé pour générer les schémas XML des descripteurs de déploiement.

DAnCE implémente le modèle d'administration de l'exécution spécifié par la spécification *OMG D&C* et illustré par la figure 3.14.

L'*ExecutionManager* est un démon permettant l'administration de domaines, ceux-ci représentant les environnements cibles. Il fabrique notamment des objets *DomainApplicationManager*, permettant d'administrer un domaine. Le *NodeManager* est implémenté comme un démon sur chaque noeud et permet d'administrer tous les composants qui résident sur ce noeud. Le *NodeApplicationManager* est colocalisé avec le *NodeManager* et est dédié à l'administration d'un groupe de composants relatifs à une application.

13. `http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/CIAO/docs/releasenotes/dance.html`
14. `http://www.omg.org/technology/documents/specialized_corba.htm`

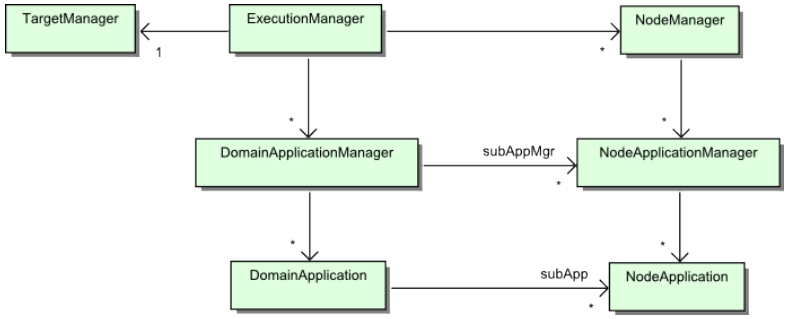


FIGURE 3.14 – Aperçu du modèle d'administration de l'exécution

Cette application est exécutée sur le noeud par le processus serveur de composants *NodeApplication*.

DAnCE fournit un dépôt pour stocker les implémentations de composants, lequel est utilisé par le *NodeApplicationManager* lors d'une requête de déploiement, afin de récupérer une version optimale de l'implémentation.

Le *DomainApplicationManager* coordonne le cycle de vie d'un assemblage de composants avec celui de ses composants. Celui-ci possède quatre états :

PREACTIVE indique qu'un composant a été créé et a eu ses paramètres environnementaux ;

ACTIVE indique qu'un composant a été activé avec l'intergiciel sous-jacent ;

PASSIVE indique qu'un composant est inoccupé et que ses ressources peuvent être utilisées par d'autres composants ;

DEACTIVATED indique que le composant a été désactivé et retiré du système.

Les composants d'un assemblage ne peuvent être activés ou connectés que lorsque la totalité des composants a atteint au moins l'état *PREACTIVE*.

3.4.6.2 Évaluation

Cette section évalue si *DAnCE* valide chacune des exigences énoncées dans la section 3.1.

Modèle d'abstraction

Granularité *DAnCE* utilise des composants *CORBA* pour la représentation de système *DRE*, la granularité est donc limitée.

Contrôle *DAnCE* offre un contrôle distribué des applications en implémentant le modèle d'administration d'exécution défini dans la spécification *OMG D&C*. Cependant, aucun mécanisme pour étendre le contrôle n'est proposé.

Hétérogénéité *DAnCE* supporte l'hétérogénéité des cibles de déploiement (eg. *Java*, *MS Windows*, *Linux*...).

Politique de résolution Aucune n'est spécifiée.

Processus de déploiement

Assemblage *DAnCE* ne traite pas l'assemblage lors de la sortie du logiciel.

Bilan *DAnCE* implémente la spécification *OMG D&C* en y ajoutant des aspects relatifs à la qualité de service. Le support de l'hétérogénéité des cibles de déploiement est conservé mais le modèle à composants choisi est trop spécifique.

3.4.7 SmartFrog

3.4.7.1 Présentation

SmartFrog (pour *Smart Framework for Object Groups*) [15] est un canevas écrit en langage *Java* pour le déploiement de systèmes logiciels [35]. Dans *SmartFrog*, un système est composé d'une collection de composants *SmartFrog*. Un composant est défini par trois éléments :

Données de configuration du composant

Gestionnaire de cycle de vie permettant de contrôler la configuration et le cycle de vie du composant

Entité administrée implémentant les fonctionnalités du composant.

Le gestionnaire de cycle de vie du composant a en charge l'encapsulation de l'entité administrée. Cette encapsulation adapte le cycle de vie de l'entité administrée pour être en cohérence avec celui du composant et traduit également les données de configuration du composant au format attendu par l'entité administrée.

La description d'un composant est composée d'un ensemble d'attributs, comme illustré par le listing 3.3. Chacun des attributs a pour valeur soit une valeur simple (eg. entier), soit une référence sur un autre attribut ou soit une autre description de composant.

```
clientServerSystem extends {
    serverPort webServer:port;
    webServer extends {
        port LAZY ENV webport;
    }
    apache extends {
        sfClass "org.smartfrog.apache,Apache";
    }
    myApacheServer extends webServer, apache;
    clientApp extends {
        port serverPort;
    }
}
```

Listing 3.3 – Descripteur de composant SmartFrog

Ce listing propose une configuration pour un client et un serveur Web *Apache*. Il décrit un composant *clientServerSystem* composé de cinq attributs : *serverPort*, *webServer*,

15. SmartFrog : `http://www.smartfrog.org/`

apache, *myApacheServer* et *clientApp*. L'attribut *serverPort* est associé à une référence sur la valeur de l'attribut *port*, lequel est défini dans la description associée à l'attribut *webServer*. La valeur de l'attribut *port* est déterminée à l'exécution à partir de la variable environnementale *webport*. L'attribut *apache* est associé à une description de composant. Celle-ci contient l'attribut particulier *sfClass* dont la valeur est le nom d'une classe d'implémentation d'un gestionnaire de cycle de vie. L'attribut *myApacheServer* a pour valeur une extension des descriptions définies dans les attributs *webServer* et *apache*. Enfin, l'attribut *clientApp* est associé à une description de composant dans laquelle l'attribut *port* a pour valeur une référence sur l'attribut *serverPort*.

Le système d'exécution de *SmartFrog* configure les logiciels à partir des descriptions de composant. Pour ce faire, des composants *SmartFrog* sont créés, lesquels offrent via leur gestionnaire de cycle de vie, une interface d'administration standard. Le cycle de vie des composants défini par le modèle est illustré par la figure 3.15.

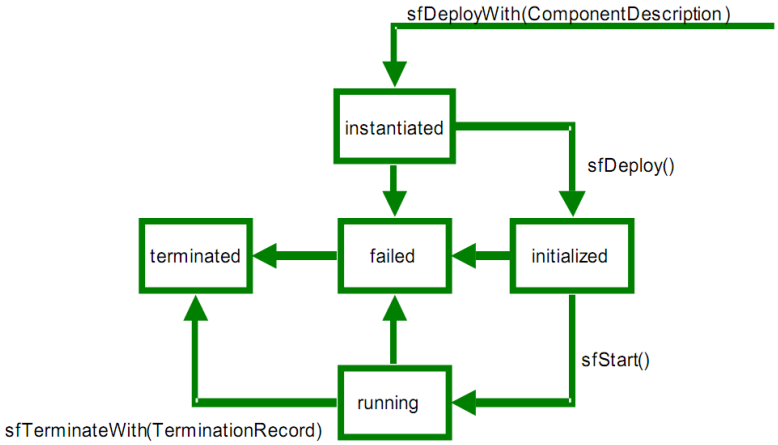


FIGURE 3.15 – Cycle de vie d'un composant SmartFrog

L'interface du gestionnaire de cycle de vie d'un composant expose quatre opérations permettant de modifier l'état du cycle de vie :

sfDeployWith permet de créer le composant à partir d'un descripteur de composant. L'état de l'entité administrée n'est pas modifié.

sfDeploy change l'état du composant à *initialized* et initialise l'entité administrée afin que celle-ci soit prête à recevoir les requêtes provenant d'autres composants.

sfStart change l'état du composant à *running* et démarre l'entité administrée.

sfTerminateWith change l'état du composant à *terminated* et stoppe l'entité administrée.

Les attributs (ie. la configuration) sont modifiables via ce gestionnaire. Une référence définit un besoin et peut être vu comme une interface cliente dans le modèle à composants *Fractal*, lorsque ce besoin porte sur un autre composant. Une référence peut également être résolue avec une valeur simple. Une liaison est créée lorsque la référence est associée

avec une valeur. Il est possible que cette liaison soit tardive (ie. à l'exécution). Dans l'exemple précédent, une telle référence était déclarée avec le mot clé *LAZY* dans la description de composant. Un composant possédant un attribut paresseux intercepte les accès aux références afin de résoudre la valeur des attributs référencés. Une liaison tardive est alors créée avec cette valeur.

L'expression des dépendances dans *SmartFrog* ne porte que sur des noms. Un nom peut être celui d'un attribut ou bien un identifiant qui sera résolu à l'exécution. Un composant ne permet pas de décrire les fonctionnalités offertes par une entité administrée. Le système d'exécution de *SmartFrog* adapte dynamiquement la configuration en liant les composants. Mais, à la différence d'un résolveur *OSGi* qui lie les modules *OSGi* en se focalisant sur leurs fonctionnalités, le système d'exécution de *SmartFrog* utilise des informations spécifiques au processus de déploiement, lesquelles sont non fonctionnelles.

Un composite est appelé dans la terminologie *SmartFrog*, *Compound*. Il peut supporter diverses politiques pour la gestion du cycle de vie de ses sous-composants. Trois politiques sont notamment définies :

Compound Les états du cycle de vie des sous-composants sont liés, dans le sens où si un composant s'arrête, tous sont arrêtés.

Sequence Les états du cycle de vie des sous-composants sont indépendants, chaque composant étant démarré après que le précédent se soit arrêté, selon l'ordre des attributs de la description.

Parallel Le cycle de vie des sous-composants est indépendant, chaque composant étant exécuté en parallèle.

D'autres politiques de déploiement peuvent être définies à l'aide du canevas de *Workflow* inclus dans *SmartFrog*. Ce canevas permet de contrôler l'échange d'évènements entre composants, offrant la possibilité d'écrire des politiques de déploiement sophistiquées.

3.4.7.2 Évaluation

Cette section évalue si *SmartFrog* valide chacune des exigences énoncées dans la section 3.1.

Modèle d'abstraction

Granularité Les logiciels sont représentés à gros grains. Ceux-ci restent des boites noires et ne permettent pas de décrire les fonctionnalités d'un logiciel.

Contrôle Le contrôle est distribué : une entité administrée peut être configurée directement à partir du composant la représentant. Il est possible d'ajouter et de modifier des attributs à l'exécution. Cependant, les interfaces de contrôle sont imposées par le modèle.

Hétérogénéité Une entité administrée peut être n'importe quel logiciel patrimonial. Cependant, le modèle ne propose pas d'abstraction des types de logiciel.

Politique de résolution Les liaisons entre composants peuvent être construites tardivement au déploiement, à l'exécution ou seulement si nécessaire.

Processus de déploiement

Assemblage *SmartFrog* ne supporte pas l'assemblage de logiciels.

Bilan *SmartFrog* est une solution pour le déploiement de logiciels patrimoniaux. Une de ses originalités est de proposer un canevas de *Workflow* destiné à la définition de politiques de déploiement. Cependant, *SmartFrog* n'adresse pas l'assemblage de logiciel, ni le déploiement des modules.

3.5 Comparaison

Dans cette section, nous comparons les solutions présentées pour chacune des exigences proposées. Cette comparaison s'effectue à l'aide de notes qui seront décrites dans la prochaine sous-section.

3.5.1 Règles de notation

La notation utilisée pour l'ensemble des tableaux est la suivante :
- Un symbole « ? » signifie que nous n'avons pas assez d'informations pour en faire une évaluations ;
- Un symbole « - » indique une absence de fonctionnalité ;
- Un symbole « + » indique un support minimal de fonctionnalité ;
- Deux symboles « + » sont attribués quand une fonctionnalité bénéficie d'un support complet.

Les règles de notations sont données dans les sous-sections suivantes. Notre contribution ambitionne la note « ++ » pour chaque critère.

3.5.2 Modèle d'abstraction

En premier lieu, nous comparons comment chaque solution valide les exigences sur le modèle d'abstraction utilisé. Les règles d'attribution des notes sont les suivantes :

Granularité La note « + » est attribuée lorsque le modèle d'abstraction offre une représentation du logiciel à fine granularité, c'est-à-dire lorsque le contenu et les dépendances fonctionnelles d'un logiciel peuvent être représentées. La note « ++ » indique que le modèle d'abstraction est générique et peut supporter n'importe quelle nouvelle fonctionnalité du logiciel.

Contrôle La note « + » est attribuée lorsque le modèle d'abstraction du produit intègre des opérations de contrôle et offre un contrôle extensible. La note « ++ » indique que la gestion de la cohérence est prise en compte par le modèle.

Hétérogénéité La note « + » est attribuée lorsque le modèle d'abstraction prend en charge n'importe quelle technologie de logiciel. La note « ++ » indique que des implémentations provenant de différentes technologies peuvent être représenté par la même abstraction.

Politique de résolution La note « + » est attribuée lorsqu'une politique de résolution peut être appliquée au cours du déploiement. La note « ++ » indique que les politiques de résolution sont modifiables.

Le tableau 3.1 contient les notes attribuées pour chacune des solutions présentées, relativement aux exigences du modèle d'abstraction.

	Granularité	Contrôle	Hétérogénéité	Politique
OMG D&C	++	-	++	++
Apache Maven	+	-	+	+
Eclipse Buckminster	-	-	+	++
Software Dock	-	?	-	+
Application Buildbox	+	-	+	?
OW2 DeployWare	-	+	+	++
Nix	-	-	-	+
Jade	+	++	+	+
DAnCE	-	-	++	?
SmartFrog	-	-	+	++

TABLE 3.1 – Comparaison des modèles d'abstraction

3.5.3 Processus de déploiement

Dans un second temps, nous comparons comment chaque solution valide l'exigence posée sur la définition du processus de déploiement utilisée. La règle d'attribution des notes est la suivante :

Assemblage La note « + » est attribuée lorsque le processus de déploiement comprend l'assemblage du logiciel. La note « ++ » indique que l'assemblage peut être mis à jour.

Le tableau 3.2 contient les notes attribuées pour chacune des solutions présentées, relativement aux exigences du processus de déploiement.

3.6 Synthèse

Les solutions présentées ne répondent pas à toutes les exigences posées. La granularité de la représentation des logiciels est l'une des parties du modèle d'abstraction la moins couverte par les outils existants. Deux raisons peuvent expliquer cette observation : soit ces outils ne supportent pas, à la fois la représentation du contenu et celle des fonctionnalités d'un logiciel (eg. *DeployWare*, *SmartFrog*), soit le type de contenu ou de fonctionnalité est imposé (eg. *Apache Maven*, *Nix*). Pourtant, la définition d'un plan de déploiement générique modifiant le contenu et les fonctionnalités de logiciels devrait

	Assemblage
OMG D&C	-
Apache Maven	+
Eclipse Buckminster	+
Software Dock	-
Application Buildbox	+
OW2 DeployWare	-
Nix	+
Jade	-
DAnCE	-
SmartFrog	-

TABLE 3.2 – Comparaison des processus de déploiement

utiliser un modèle d'abstraction permettant une fine représentation du logiciel. Notre contribution propose un tel modèle d'abstraction n'imposant aucun type de logiciel ou de fonctionnalité.

Les solutions présentées se distinguent entre elles également par les capacités offertes de contrôle des reconfigurations. Certaines solutions proposent uniquement un contrôle centralisé via une interface en ligne de commande (eg. *Nix*) ou graphique (*Eclipse Buckminster*), d'autres proposent des interfaces de contrôle distribuées, lesquelles sont accessibles directement à partir de la représentation abstraite du logiciel (eg. *DeployWare*, *Jade*). On remarquera que les outils basés sur le modèle à composant *Fractal* tirent profit de la réflexivité des composants *Fractal* pour offrir un contrôle distribué et extensible des logiciels représentés. La spécification *OMG D&C* ainsi que les outils *DAnCE* et *SmartFrog* définissent bien des contrôleurs spécifiques pour chaque logiciel, cependant ils ne proposent pas de mécanisme pour étendre ces contrôleurs. Enfin, la gestion de la cohérence du modèle n'est abordée que par peu de solutions étudiées. Notre contribution réutilisera le modèle à composants *Fractal*, afin d'offrir des interfaces de contrôle directement à partir de la représentation du logiciel, ces interfaces pouvant être très simplement ajoutées ou modifiées.

La plupart des outils présentés dans ce chapitre sont dédiés aux logiciels patrimoniaux et n'imposent pas de technologie logicielle (exceptés *Nix* et *Software Dock*). Cependant, l'hétérogénéité des cibles du déploiement n'est pas prise en compte dans les modèles d'abstraction proposés. Seule la spécification *OMG D&C* et son implémentation *DAnCE* proposent d'associer un composant à de multiples implémentations, le choix se faisant au déploiement. Une telle approche facilite l'écriture de plan de déploiement générique. Notre contribution propose de supporter à la fois l'hétérogénéité des technologies logicielles et des cibles du déploiement, en définissant une représentation abstraite du logiciel laquelle est indépendante de son implémentation.

Chaque solution propose un mécanisme pour la définition de politiques de déploiement permettant ainsi de modifier un processus de déploiement sans modifier le plan

décrivant celui-ci. *Apache Maven* autorise ainsi la définition de plugins pouvant être intégrés dans le cycle de vie du déploiement. Cependant, *Maven* n'offre pas de moyen standard pour modifier la politique de résolution des dépendances. *Eclipse Buckminster* permet de choisir la manière dont les dépendances sont téléchargées localement à l'aide de *Materializers*. *Software Dock* offre une résolution des dépendances via les agents responsables de chaque tâche du déploiement, mais ne spécifie pas comment modifier celle-ci. *DeployWare* encapsule les politiques de déploiement dans des composants *Fractal*, parmi lesquels l'automate de déploiement orchestre le déploiement des dépendances. *Maven* et *SmartFrog* permettent d'associer une dépendance à une temporalité décrivant à quel moment celle-ci doit être résolue. Notre approche offre un canevas simple et performant pour la définition de politique de déploiement. En outre, nous proposerons des politiques de résolution des dépendances.

Enfin, tous ces outils ne prennent pas en charge l'assemblage de la sortie d'un logiciel. Lorsque c'est le cas (*Maven*, *Buckminster*, *Application Buildbox*, *Nix*), l'assemblage généré n'est pas associé à une représentation abstraite intégrant les éléments assemblés et facilitant la mise à jour de ceux-ci. Notre contribution propose de construire une représentation d'un assemblage permettant d'accéder au contenu et de le modifier.

Deuxième partie

Modèle et langages

Chapitre 4

Spécification

Sommaire

Ce chapitre décrit l'approche suivie pour répondre aux exigences spécifiées dans l'état de l'art. Après avoir justifié de l'utilisation d'un modèle à composants dans la section 4.1, nous présenterons succinctement, dans la section 4.2, l'approche suivie pour résoudre la problématique. Puis, nous expliquerons dans la section 4.3, comment le modèle à composants *Fractal SoftwareUnit* offre une représentation souple et extensible d'un logiciel, ainsi que de son déploiement. Enfin, dans la section 4.4, nous proposerons l'utilisation de langages afin de décrire le déploiement.

4.1 Motivations

Cette thèse cherche à automatiser le déploiement d'une application et de son environnement. Nous pensons que la représentation abstraite du logiciel est la clé de cette automatisation, celle-ci permettant de décrire une partie du processus de déploiement. Une représentation du déploiement à faible granularité nous semble indissociable de celle du logiciel, le déploiement pouvant modifier le logiciel.

La difficulté réside dans la construction de cette représentation. Celle-ci doit supporter n'importe quel type de logiciel sans limitation de la granularité de représentation. Cette granularité est relative au contexte d'utilisation du logiciel : un fichier *.deb* peut être interprété comme un paquet *Debian*, un fichier ou encore comme une archive. Ainsi, un paquet *Debian*, tout comme un paquet *RPM* ou un *bundle OSGi*, décrit des capacités et besoins, mais aussi est une archive contenant d'autres logiciels. La diversité des contextes d'utilisation et donc des points de vue provient de l'hétérogénéité des logiciels.

Si un paquet *Debian* est une archive, une archive peut être également un *bundle OSGi*. Une représentation de logiciel doit donc supporter une fine granularité de représentation, ainsi que l'hétérogénéité des logiciels.

Quels que soient les logiciels et le détail de la représentation, les logiciels présentent néanmoins des caractéristiques communes, telles que la composition et la définition de dépendances explicites. Les systèmes de module, tels que *Debian*, *RPM* ou *OSGi*, définissent par exemple des modules dont la définition des dépendances est finalement très semblable (voir la section 2.3). Les paquets *Debian* et *RPM* décrivent ainsi des capacités et besoins en terme de paquet et contiennent des fichiers. Un *bundle OSGi* décrit des dépendances en terme de *bundle* et de *package*, et contient des fichiers et des *packages*.

Un logiciel peut également avoir un état associé, même si celui-ci n'a pas nécessairement connaissance de cet état (il peut être réifié au niveau du système de module comme pour les paquets *RPM* par exemple). Un logiciel peut ainsi avoir un cycle de vie qui lui soit propre. Un *bundle OSGi* possède par exemple un cycle de vie à 6 états.

Ces relations de composition et de dépendances explicites entre logiciels, ainsi que l'existence d'état spécifique au logiciel peuvent être capturées grâce à une représentation basée sur des composants réflexifs. Des travaux ont déjà montré l'intérêt de la modélisation par composant pour le déploiement logiciel [27, 35]. Néanmoins, certains sont liés à un système de modules particulier [2], d'autres se focalisent sur le processus de déploiement [4].

Dans la continuité de ces travaux, nous proposons un nouveau modèle à composants *Fractal SoftwareUnit*, spécialisé dans la modélisation de logiciels hétérogènes.

4.2 Aperçu de l'approche proposée

Cette section présente succinctement notre approche et notamment le modèle à composants *Fractal SoftwareUnit*. Dans un premier temps, nous décrivons le modèle d'abstraction utilisé. Puis, nous montrons comment celui-ci permet de résoudre la problématique du déploiement de logiciels à la juste taille (sous-section 4.2.3).

4.2.1 Représentation d'un processus de déploiement

Nous proposons d'associer les étapes d'un processus de déploiement à des fonctions opérant sur des abstractions des logiciels à déployer. Ces fonctions sont nommées *opérations* et les abstractions des logiciels sont des composants *SoftwareUnit*. Ces derniers sont des composants *Fractal* spécialisés dans la représentation abstraite du contenu et des dépendances de logiciels. Le domaine d'une opération est défini par un ensemble de composants *SoftwareUnit* et le codomaine par un ensemble de composants *Fractal*. Une opération peut créer de nouveaux composants *SoftwareUnit* (eg. création d'un assemblage de logiciels), mais également des composants *Fractal* (eg. lors de l'activation d'un *bundle OSGi* représenté par un composant *SoftwareUnit*). Une opération peut avoir des effets de bord en modifiant les liaisons entre composants (eg. résolveur *OSGi*).

Un processus de déploiement peut être décrit soit de manière implicite, à partir d'une description architecturale, soit de manière explicite, en terme de plan de déploiement.

Une description architecturale s'écrit avec l'*ADL Fractal*. Un plan de déploiement est construit à l'aide d'opérations sur des composants *SoftwareUnit*. Celui-ci peut être écrit dans différents langages, tels que *FScript* ou *XML SUD* (présenté dans la section 4.4.2). Une opération est définie, dans le modèle *FScript*, comme une procédure primitive.

4.2.2 Représentation de logiciels

Les composants *SoftwareUnit* sont définis pour abstraire des caractéristiques de logiciels, telles que leurs fonctionnalités, leurs dépendances et leur contenu. Notre modèle autorise ces caractéristiques à évoluer au cours du temps. Une capacité ou un besoin d'un logiciel est exprimé à l'aide d'une interface sur le composant le représentant. Dans notre modèle, ces interfaces sont nommées des *descripteurs*. Le contenu d'un logiciel correspond à la relation de composition.

Ces composants permettent également la définition d'interfaces de contrôle uniformes des logiciels. Une opération peut ainsi agir sur les interfaces de contrôle, par exemple en modifiant les dépendances ou le contenu de logiciels, sans être attachée à l'implémentation des logiciels représentés par les composants.

Nous distinguons deux natures de composants *SoftwareUnit* : les composants concrets et abstraits. Un composant *SoftwareUnit* concret expose le contenu d'un logiciel sous forme d'une boite blanche. À l'inverse, un composant abstrait présente l'aspect d'une boite noire dont le contenu n'est pas visible. Il permet néanmoins de représenter les caractéristiques externes d'un logiciel (fonctionnalités et dépendances). Un composant abstrait est associé à une usine à déploiement, laquelle retourne un composant concret. Une usine à déploiement invoque des opérations sur des composants *SoftwareUnit*. L'intérêt d'un composant abstrait est de fournir à la fois une description limitée d'un logiciel, notamment parce que son implémentation n'est pas connue, et un processus de déploiement permettant de déployer une implémentation.

Notre modèle propose une représentation générique des méta-données exprimant les capacités et besoins des logiciels. Celle-ci permet de construire des outils de résolution, indépendamment d'une technologie de logiciel. Le langage d'expression des méta-données étant relatif à une technologie, les méta-données doivent être traduites dans la représentation générique. Dans notre modèle, un système *SoftwareUnit* apporte le support d'une technologie logicielle et permet l'interprétation des méta-données écrites dans son langage.

Un composant *SoftwareUnit* est associé à un type, construit à partir du type de chacun de ses descripteurs. Le type d'un descripteur représente une méta-donnée décrivant une capacité ou un besoin. Le typage est utile dans différents cas :

Résolution de dépendance Une capacité satisfait un besoin si les deux types sont compatibles. Ceci est vérifié en comparant les méta-données des deux descripteurs selon la relation de *pattern matching*.

Mise à jour La mise à jour d'un logiciel n'est possible que si celui-ci est substituable à l'ancien, c'est à dire que le nouveau logiciel soit représenté par un *sous-type* de celui existant.

4.2.3 Déploiement de logiciels à la juste taille

Nous avons constaté, dans le chapitre 1 d'introduction, que l'hétérogénéité des dépendances des logiciels, ainsi que celle des cibles de déploiement, rendaient la définition de processus de déploiements complexe. L'intérêt de notre modèle d'abstraction est de masquer cette double hétérogénéité et de faciliter ainsi la définition de processus de déploiement, minimisant à la fois le nombre d'opérations invoquées et la taille du logiciel déployé.

Cette minimisation est permise grâce à l'identification des dépendances du logiciel et de l'utilisation d'un algorithme de déploiement cherchant à n'exécuter que les opérations de déploiement requises. La représentation des dépendances, combinée à la possibilité d'associer celles-ci à des opérations de déploiement, facilite l'identification des opérations requises.

4.3 Le modèle à composants *Fractal SoftwareUnit*

Le modèle à composants *Fractal* (présenté dans la section 2.2.1) a été choisi pour sa réflexivité et sa flexibilité. Il propose une définition de membrane adaptable à la nature du logiciel. La plate-forme *SCA FraSCAti*[1] [55] démontre l'ouverture du modèle à composants *Fractal* en implémentant des composants *SCA*[2] à l'aide de composants *Fractal*. *SCA* (pour *Service Component Architecture*) est un ensemble de spécifications qui décrivent un modèle pour bâtir des applications s'inscrivant dans une architecture orientée service (appelée aussi *SOA*). Les applications basées sur *SCA* sont un assemblage de composants *SCA*.

En outre, *Fractal* a déjà été utilisé pour la représentation de logiciel. La seconde implémentation de *Jade* (présentée dans la section 3.4.5) définit un nouveau système de modules en utilisant *Fractal* afin de modéliser ceux-ci (lesquels ressemblent fortement à des *bundles OSGi*). Les interfaces clientes représentent ainsi des imports de *package*, les interfaces serveurs des exports. Nous proposons de généraliser cette approche en spécifiant le modèle *Fractal SoftwareUnit* dans lequel n'importe quel logiciel existant pourra être modélisé. Cette extension du modèle *Fractal* vise à décrire précisément les points spécifiques à la représentation de logiciels, tels que le système de types, les contextes de nommage.

Le modèle à composants *Fractal SoftwareUnit* n'est pas un modèle pour le développement d'applications, mais pour l'après-développement, de l'assemblage jusqu'au retrait. Ce modèle n'impose pas d'environnement d'exécution. Par exemple, un logiciel modélisé

1. OW2 FraSCAti : `https://wiki.ow2.org/frascati/Wiki.jsp?page=FraSCAti`
2. Standard SCA : `http://www.osoa.org/`

peut être exécuté sur une plate-forme *OSGi* ou alors directement par le système d'exploitation. Au contraire, n'importe qu'elle machine d'exécution peut y être représentée.

Le modèle à composants *Fractal SoftwareUnit* constitue la fondation de notre contribution.

4.3.1 Relations avec le modèle à composants *Fractal*

La spécification du modèle à composants *Fractal* [13] propose une définition de contexte de nommage afin (entre autre) de supporter l'hétérogénéité des plate-formes d'exécution des composants. Dans le contexte de la représentation de logiciel, on ne parle plus de plate-forme d'exécution mais plutôt de *système SoftwareUnit*, lequel représente une technologie logicielle (eg. *OSGi*, *Debian*, *RPM*...). Dans la spécification *Fractal* un contexte de nommage associe un nom avec une interface. Dans notre spécification, un contexte de nommage associe un nom avec un logiciel de cette technologie. Un système *SoftwareUnit* représente donc un ensemble de logiciels et est associé à un contexte de nommage permettant d'adresser chacun d'entre eux.

La spécification *Fractal* propose une définition d'un système de type supportant l'hétérogénéité des langages d'implémentation des composants. Dans ce système de type, le type d'une interface *Fractal* est construit à partir d'une signature correspondant au nom du type de l'interface langage. Dans notre spécification, le type d'une interface *Fractal* est construit à partir d'une description des besoins et des capacités du logiciel. De la même manière que la spécification *Fractal* supporte l'hétérogénéité des langages d'implémentation, notre spécification supporte l'hétérogénéité des langages de descriptions des capacités et des besoins. Une technologie logicielle peut définir un langage de description des capacités et des besoins qui lui soit propre.

La spécification du modèle à composants *Fractal SoftwareUnit* distingue la représentation du logiciel (présentée dans la section 4.3.2), laquelle est une extension directe du modèle à composants *Fractal*, et la définition d'un système *SoftwareUnit* (présentée dans la section 4.3.3). Enfin, une troisième partie de la spécification est consacrée à la définition d'opérations de déploiement logiciel exploitant le modèle d'abstraction introduit (dans la section 4.3.4).

La figure 4.1 décrit les relations entre les trois parties du modèle à composants *Fractal SoftwareUnit* et le modèle à composants *Fractal*.

Les interfaces en vert sont relatives à la représentation du logiciel. Les classes en bleu appartiennent au modèle à composants *Fractal*. Les classes en jaune définissent un système *SoftwareUnit*. Enfin, les classes en rouge définissent une opération de déploiement. Chacune de ces classes sera décrite par la suite.

4.3.2 Modélisation du logiciel

Cette section fournit un modèle générique du logiciel à déployer.

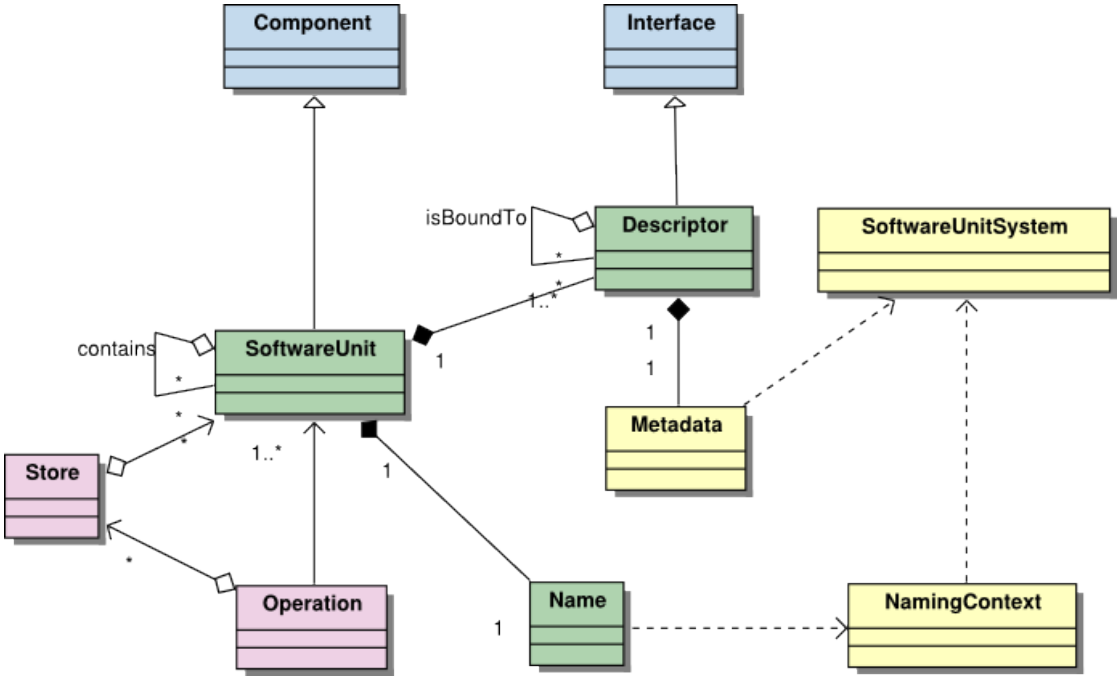


FIGURE 4.1 – Méta-modèle *Fractal SoftwareUnit* simplifié

4.3.2.1 Définition d'un composant *SoftwareUnit*

Un composant *SoftwareUnit* est un composant *Fractal* spécialisé, modélisant la structure, les fonctionnalités et les dépendances d'un logiciel.

Dans le modèle à composants *Fractal SoftwareUnit*, le contenu d'un logiciel est caractérisé par la relation de composition. Par exemple, si une archive est représentée par un composant *SoftwareUnit*, le contenu de cette archive peut être représenté avec des sous-composants.

Les fonctionnalités et dépendances d'un logiciel sont décrites par les interfaces fonctionnelles du composant. Dans notre modèle, nous les appelons des *descripteurs*. Par exemple, un composant représentant un paquet *RPM* possède un descripteur représentant le nom et la version du paquet, ainsi que des descripteurs pour chacune de ses dépendances sur d'autres paquets.

Une dépendance entre logiciels est dite résolue lorsqu'une liaison entre interfaces de composant existe.

La membrane du composant permet d'introspecter et de modifier un logiciel, tout en vérifiant la cohérence du modèle. Par exemple, le contrôleur de liaisons d'un composant représentant un *bundle OSGi*, doit à la fois notifier la plate-forme *OSGi* de la création d'une liaison et détecter la rupture d'une liaison initiée par la plate-forme.

Chaque composant *SoftwareUnit* identifie le logiciel modélisé par un nom unique, ce nom étant défini dans le contexte de nommage d'un système *SoftwareUnit* (voir la section 4.3.3.1).

4.3.2.2 Vue externe d'un composant *SoftwareUnit*

En tant que composant *Fractal*, un composant *SoftwareUnit* expose des interfaces fonctionnelles et non-fonctionnelles. Les interfaces fonctionnelles d'un composant *SoftwareUnit* sont appelées des *descripteurs*. Ceux-ci décrivent les capacités (interfaces serveur) et besoins (interfaces clients) du logiciel (comme illustré par la figure 4.2).

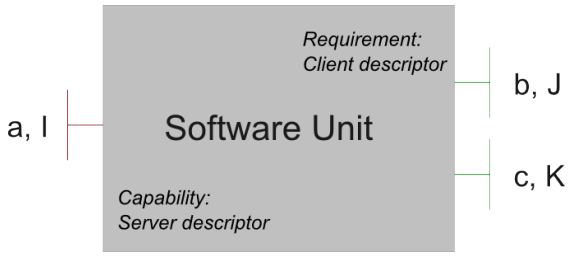


FIGURE 4.2 – Vue externe d'un composant *SoftwareUnit*

Sur cette figure, chaque descripteur est associé à un nom, ainsi qu'une méta-donnée. Par exemple, le descripteur de capacité dont le nom est *a* est associé à la méta-donnée *I*. Les méta-données remplacent les interfaces langage pour la construction du type des descripteurs (défini dans la section 4.3.2.6).

Afin d'accéder aux descripteurs d'un composant *SoftwareUnit*, celui-ci fournit l'interface non-fonctionnelle SoftwareUnit (définie dans le listing 4.1). Celle-ci étend l'interface Component.

```
package org.ow2.fractal.su.api;
interface SoftwareUnit extends Component {
    Descriptor[] getFcDescriptors();
    Descriptor getFcDescriptor(String descName)
        throws NoSuchDescriptorException;
    SoftwareUnitType getFcType();
}
```

Listing 4.1 – API d'introspection d'un composant SoftwareUnit

L'opération getFcDescriptor retourne le descripteur pour un nom de descripteur. Il est à noter que l'opération getFcInterface retourne le même résultat que l'opération getFcDescriptor pour les noms de descripteur. L'opération getFcType retourne le type du composant, représenté par interface SoftwareUnitType. Celle-ci étend l'interface ComponentType et sera décrite dans la section 4.3.2.6.

Introspection d'un descripteur Un descripteur est une interface fonctionnelle du composant *SoftwareUnit*. Un descripteur implémente l'interface Descriptor, laquelle étend l'interface Interface (définie dans le listing 4.2).

```
package org.ow2.fractal.su.api;
interface Descriptor extends Interface {
    SoftwareUnit getFcItfOwner();
    DescriptorType getFcItfType();
}
```

Listing 4.2 – API d'introspection d'un descripteur

L'opération getFcItfOwner retourne l'interface SoftwareUnit du composant propriétaire de ce descripteur. L'opération getFcItfType retourne le type du descripteur, lequel est représenté par l'interface DescriptorType.

4.3.2.3 Vue interne d'un composant *SoftwareUnit*

Un composant *SoftwareUnit* composite peut contenir d'autres composants *SoftwareUnit* (comme illustré par la figure 4.3), un composant primitif n'a pas de contenu.

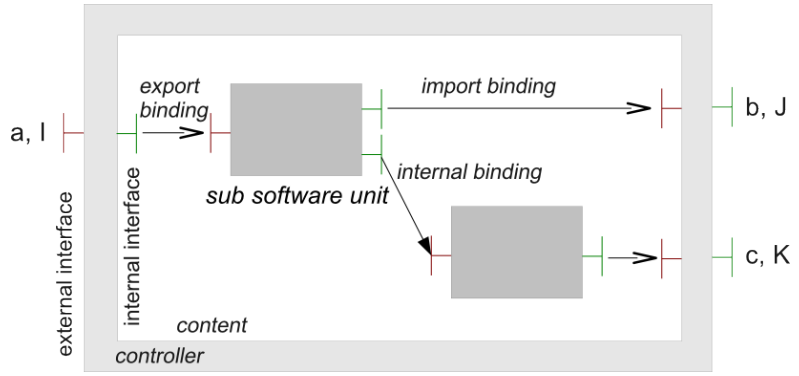


FIGURE 4.3 – Vue d'interne d'un composant *SoftwareUnit* composite

Cette figure illustre un composite, fournissant une fonctionnalité décrite par le descripteur de nom a et dont la méta-donnée est I. Cette fonctionnalité est implémentée par des sous-composants primitifs. Ces derniers ont des dépendances représentées au niveau du composite, par les descripteurs b et c.

Exemple de représentation d'un module patrimonial Un *bundle OSGi* peut être représenté comme un composite, dont les sous-composants (primitifs) définissent des *packages Java* (comme illustré sur la figure 4.4). La construction de cette représentation sera décrite dans la section 5.4.2.

Dans cet exemple, le *bundle* a pour nom symbolique foo et a la version 1.0.0. Il exporte le *package* foo en version 1.0.3 et requiert le *bundle* de nom *bar*. Ce *bundle* possède également une dépendance obligatoire sur le *package* de nom *bar*, dont la version doit être comprise entre 1.23 et 1.24. Enfin, il possède une dépendance optionnelle sur le

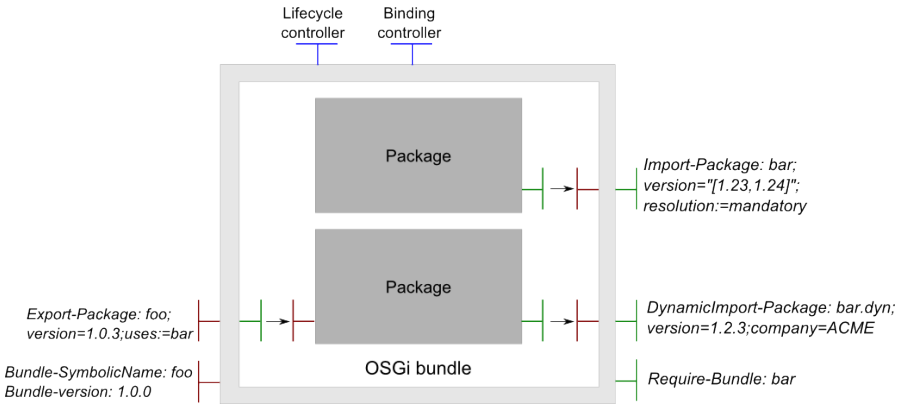


FIGURE 4.4 – Modélisation d'un *bundle OSGi*

package de nom *bar.dyn*, dont la version doit être 1.2.3 et la valeur de l'attribut *company* doit être *ACME*.

Ce *bundle OSGi* est modélisé à l'aide d'un composite *SoftwareUnit*. Ses *packages* sont également modélisés, en tant que sous-composants primitifs. Les *packages* exportés sont représentés par des descripteurs de capacités, présents à la fois sur le composite du *bundle* et le composant primitif du *package*, les deux étant connectés par une liaison interne. De même, les *packages* importés sont modélisés par des descripteurs de besoins, présents à la fois sur le composite du *bundle* et le composant primitif du *package*, les deux étant connectés par une liaison interne. Notre modèle permet d'identifier aisément quels sont les *packages* utilisant les imports déclarés sur un *bundle*.

4.3.2.4 Configuration

Ce modèle étend différents contrôleurs définis dans la spécification *Fractal*. Leur sémantique est spécialisée pour le contexte de la représentation de logiciels. Certains contrôleurs présentés permettent de modifier des logiciels et maintiennent la cohérence de la représentation. Par exemple, la figure 4.4 illustre un composant représentant un *bundle OSGi*, lequel offre deux contrôleurs : le contrôleur de liaisons et le contrôleur de cycle de vie. Le rôle de ces contrôleurs est décrit dans les paragraphes suivants.

Contrôle des liaisons Le contrôleur de liaisons permet de représenter la résolution d'une dépendance entre deux logiciels, en associant un descripteur de capacité du logiciel requis, avec un descripteur de besoin du logiciel requérant. La création d'une liaison au niveau du modèle implique que le logiciel représenté ait effectivement cette dépendance résolue. Par exemple, si le logiciel représenté est un *bundle OSGi*, la création d'une liaison entre un import et un export de *package* doit être prise en compte par la plate-forme *OSGi*. Inversement, si un *package* est résolu sur la plate-forme, le contrôleur doit mettre à jour les liaisons du composant concerné.

Une liaison entre un besoin exprimant une dépendance et une capacité ne peut être

créée que si cette capacité résout le besoin (formalisé dans la section 4.3.2.6).

Un composant *SoftwareUnit* peut fournir l'interface SoftwareUnitBindingController pour le contrôle de ses liaisons (définie dans le listing 4.3).

```
package org.ow2.fractal.su.api.control.binding;
interface SoftwareUnitBindingController
        extends BindingController {
    Descriptor lookupFc (String clientDescName)
        throws NoSuchDescriptorException;
}
```

Listing 4.3 – API de contrôle des liaisons

L'opération lookupFc est redéfinie en surchargeant le type de retour avec l'interface Descriptor. Cette opération retourne le descripteur de capacité associé au descripteur de besoin ayant le nom spécifié. En effet, un descripteur ne peut être lié qu'à un autre descripteur (et non à une interface *Fractal* quelconque).

Contrôle du contenu Le contrôleur de contenu permet d'ajouter ou d'enlever un composant *SoftwareUnit* à un composite. Concrètement, cela signifie qu'un élément du logiciel représenté est ajouté ou supprimé. Par exemple, si ce logiciel est un répertoire de fichiers, ce contrôleur autorise l'ajout et la suppression de fichiers dans celui-ci.

Un composite *SoftwareUnit* peut fournir l'interface SoftwareUnitContentController pour le contrôle de son contenu *SoftwareUnit* (définie dans le listing 4.4).

```
package org.ow2.fractal.su.api.control.content;
interface SoftwareUnitContentController
        extends ContentController {
    Descriptor[] getFcInternalInterfaces();
    Descriptor getFcInternalInterface (String descName)
        throws NoSuchDescriptorException;
    SoftwareUnit[] getFcSubComponents();
    boolean containFcSubSoftwareUnit(SoftwareUnit softwareUnit);
}
interface SoftwareUnitSuperController extends SuperController {
    SoftwareUnit[] getFcSuperSoftwareUnits();
}
```

Listing 4.4 – API de contrôle du contenu

Les opérations getFcInternalInterfaces et getFcInternalInterface permettent d'accéder aux descripteurs internes d'un composite. L'opération getFcSubComponents retourne les composants SoftwareUnit contenu dans le composite. Enfin, l'opération containFcSubSoftwareUnit teste si le composite contient le composant SoftwareUnit dont le nom est spécifié.

L'opération getFcSuperSoftwareUnits de l'interface SoftwareUnitSuperController retourne les composants SoftwareUnit contenant ce composant.

Contrôle du cycle de vie Le cycle de vie de base d'un composant *SoftwareUnit* est celui à deux états (STARTED et STOPPED) hérité du cycle de vie d'un composant *Fractal*. Ainsi, les différentes étapes du déploiement ne sont pas intégrées car celles-ci peuvent varier d'un logiciel à un autre. Néanmoins, le cycle de vie peut être étendu si besoin est, selon le type de logiciel représenté.

Chaque étape du cycle de vie a une sémantique relative au logiciel représenté. Par exemple, si ce logiciel est un fichier exécutable, l'état STARTED correspond à un fichier exécuté. Si le logiciel représenté est un *bundle*, cet état correspond alors à un *bundle* activé. Le cycle de vie doit être cohérent avec celui du logiciel. Ainsi, ce contrôleur doit détecter qu'un *bundle* est arrêté, même lorsqu'il est court-circuité par la plate-forme *OSGi*.

Contrôleur de système Le contrôleur de système permet d'accéder au système *SoftwareUnit* auquel le composant est lié. Il permet également de retrouver le nom du composant dans celui-ci.

Un contrôleur de système est représenté par l'interface SoftwareUnitSystemController (définie dans le listing 4.5).

```
package org.ow2.fractal.su.api.control.system;
interface SoftwareUnitSystemController {
    SoftwareUnitSystem getFcSoftwareUnitSystem();
    Name getFcSoftwareUnitName();
    Descriptor getFcSymbolicDescriptor();
    void setFcSymbolicDescriptor(Descriptor descriptor);
}
```

Listing 4.5 – API de contrôle du système

L'opération getFcSoftwareUnitSystem retourne le système *SoftwareUnit* associé au composant, lequel est représenté par l'interface SoftwareUnitSystem (définie dans la section 4.3.3). L'opération getFcSoftwareUnitName retourne le nom du logiciel tel qu'il est défini dans le référentiel du système *SoftwareUnit* (voir la section 4.3.3.1).

Les opérations getFcSymbolicDescriptor et setFcSymbolicDescriptor permettent d'accéder et de modifier le descripteur symbolique du composant. Le descripteur symbolique n'est pas seulement un descripteur de capacité, mais il identifie aussi le composant de manière unique, dans un système *SoftwareUnit* et pour une capacité donnée. Le choix du descripteur symbolique est donc relatif au système *SoftwareUnit* pour lequel le logiciel est représenté.

Par exemple, le descripteur symbolique d'un composant représentant un paquet *RPM*, dans le système *SoftwareUnit RPM*, est le descripteur défini à partir du nom et de la version du paquet. Un descripteur représentant un paquet virtuel ne peut être un descripteur symbolique. Le descripteur symbolique d'un composant représentant un paquet *RPM*, dans le système *SoftwareUnit File*, est le descripteur défini à partir du nom du fichier *.rpm*.

Un système *SoftwareUnit* ne définissant pas de descripteur symbolique représente

une technologie de logiciels dans laquelle ceux-ci ne peuvent être identifiés de manière unique à partir de leurs capacités. À la différence, le nom d'un composant dans son système *SoftwareUnit* est obligatoire, car il identifie le logiciel représenté.

Le fait qu'un logiciel ait un nom dans un système donné, n'impose pas que celui-ci soit composé exclusivement de descripteurs dont les méta-données soient définies dans ce système. Par exemple, un composant représentant un *bundle OSGi*, peut avoir son nom dans le système *SoftwareUnit File* et des descripteurs dont les méta-données sont définies dans le système *SoftwareUnit OSGi* (ie. des descripteurs de *bundles* ou de *packages*).

Contrôleur de contraintes Certaines contraintes sur les dépendances entre logiciels peuvent porter simultanément sur plusieurs d'entre eux et ne peuvent être exprimées seulement au niveau des méta-données d'un descripteur.

La coprésence de logiciels est une telle contrainte. Celle-ci permet de définir un ensemble de dépendances optionnelles devenant obligatoires lorsque l'une d'entre elle est résolue (« tout ou rien »). Cette contrainte est utile pour exprimer des dépendances n'apparaissant qu'à l'exécution. Ce cas peut se présenter, par exemple, lors de l'utilisation de *bundles OSGi fragment*. Un *bundle* hôte peut être étendu par plusieurs *bundles* fragment. La contrainte de coprésence permet de vérifier que si un bundle *hôte* est résolu, ses fragments doivent l'être également.

Au niveau de la représentation *Fractal*, la coprésence implique que si une liaison optionnelle est créée, d'autres doivent l'être obligatoirement. Cette contrainte peut être exprimée à l'aide d'une expression *FScript*.

La cohérence de logiciels, dont les dépendances ont des versions, peut être vérifiée avec une contrainte. Dans certains cas, la fermeture transitive des dépendances d'un logiciel ne devrait pas contenir plusieurs versions d'un même logiciel. La figure 4.5 illustre une telle situation, dans laquelle un *bundle OSGi* utilise un espace des classes inconsistant, dû à un import de mêmes classes à des versions différentes.

Sur cette figure, les composants représentent des *bundles OSGi*. Le composant A est lié à un composant D lui fournissant le *package* $foo.api$, en version 2. Le composant A est également lié à un composant B lui fournissant le *package* bar. Enfin le composant B est lié à un composant C lui fournissant le *package* $foo.api$, en version 3. Si ce dernier *package* est utilisé dans le *package* bar, le composant A utilisera deux versions simultanément du *package* $foo.api$, son espace des classes n'est alors plus cohérent. Il s'agit d'une incohérence architecturale que seul le concepteur de l'application peut résoudre, soit en choisissant une version unique du *package* $foo.api$ et s'il ne le peut pas, en faisant en sorte que le *package* bar n'utilise plus ce *package*.

Le résolveur des plate-formes *OSGi* peut provoquer ce type d'incohérences liées aux versions, car celui-ci a pour seules informations les méta-données des *bundles*, lesquelles n'offrent pas une description globale des dépendances. La spécification *OSGi* [63] définit une solution, en permettant de signaler au résolveur, lorsqu'un *bundle* exporte un *package* utilisant un *package* importé. Pour cela, la directive *uses* doit être associée aux *packages*

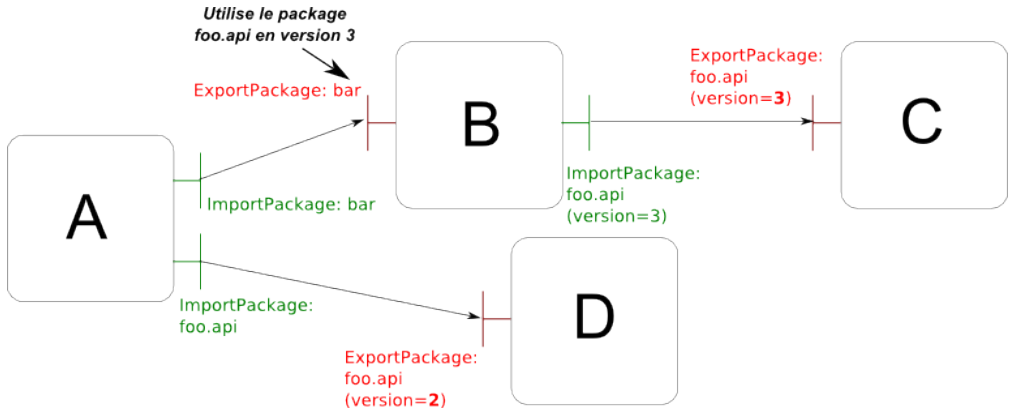


FIGURE 4.5 – Incohérence due à des imports de packages ayant des versions différentes

exportés. La figure 4.6 illustre l'utilisation de la directive *uses*.

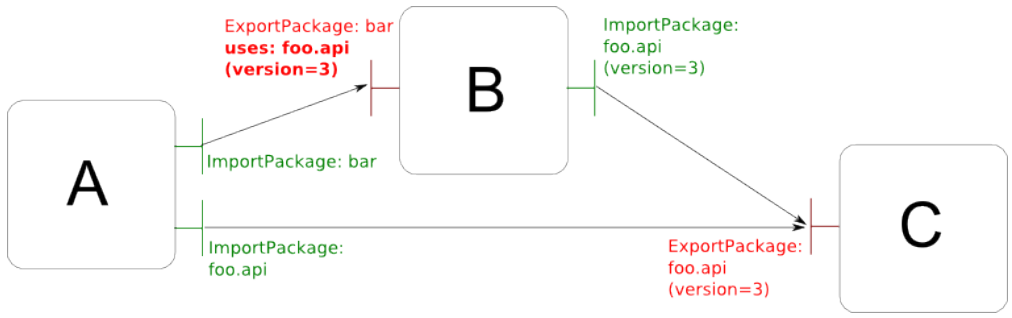


FIGURE 4.6 – Utilisation de la directive uses

Grâce à la directive *uses* et parce que le *bundle A* ne spécifie pas de version pour l'import du *package foo.api*, le résolveur détecte qu'une seule version doit être utilisée et choisit la seule préalablement spécifiée. Sans l'usage de la directive *uses*, le résolveur n'essaie pas de trouver une version unique, pouvant conduire à l'incohérence illustrée par la figure 4.5.

Nous proposons l'utilisation d'une description architecturale associée à des contraintes, permettant de valider la cohérence des versions avant la résolution. Si l'utilisation de la directive *uses* résout le problème cité, celle-ci peut-être complexe à mettre en œuvre et peut entraîner des violations de contraintes *uses*, un problème fréquent en *OSGi* et difficile à diagnostiquer.

Grâce à la vision architecturale adoptée, notre approche permet à la fois d'identifier les *bundles* utilisant la directive *uses* (celle-ci étant réifiée dans les méta-données des descripteurs de composant) et une solution alternative, dans laquelle cette directive est remplacée par la définition de liaisons au sein de la description architecturale.

L'expression *FScript* suivante vérifie que les composants A et B utilisent bien la même version du *package* $foo.api$:

```
intersection(
    $a/interface::client-package-bar/binding::*/
        component::*/interface::client-package-foo.api/binding::*,
    $a/interface::client-package-foo.api/binding::*) > 0
```

Afin d'assurer la cohérence des configurations et des reconfigurations au niveau du composant, nous intégrons le contrôleur de contraintes définis par M. Léger [46, 47]. Ce contrôleur est représenté par l'interface ConstraintController (définie dans le listing 4.6).

```
package org.ow2.fractal.su.api.control.constraint;
interface ConstraintController {
    String getConstraint(String name);
    void addConstraint(String name, String constraint);
    void removeConstraint(String name);
    Set<String> getContraints();
    void checkConstraints(boolean recursive)
            throws ConstraintViolationException, ConstraintParsingException;
    ConsistencyManager getChecker();
    void setChecker(ConsistencyManager checker, boolean recursive);
}
interface ConsistencyManager {
    void check(Component component, boolean recursive)
            throws ConstraintViolationException, ConstraintParsingException;
}
```

Listing 4.6 – API de contrôle des contraintes

L'opération getConstraint retourne la forme textuelle de la contrainte dont le nom a été spécifié. Dans notre implémentation, les contraintes sont des expressions *FScript*. L'opération addConstraint permet l'ajout d'une nouvelle contrainte. L'opération removeConstraint permet le retrait d'une contrainte. L'opération getConstraints retourne tous les noms de contraintes enregistrés. L'opération checkConstraints lance une vérification des contraintes. Les opérations getChecker et setChecker permettent de retrouver et de modifier le gestionnaire de cohérence associé au contrôleur.

Le gestionnaire de cohérence est représenté par l'interface ConsistencyManager et implémente le vérificateur de contraintes. L'opération check lance la vérification.

4.3.2.5 Création d'un composant

Nous distinguons deux natures de composants *SoftwareUnit* : les composants concrets et abstraits. Un composant *SoftwareUnit* concret expose le contenu d'un logiciel sous forme d'une boite blanche. À l'inverse un composant abstrait présente l'aspect d'une boite noire dont le contenu n'est pas visible, notamment parce qu'il est inconnu. Un composant abstrait est associé à une usine à déploiement permettant de créer une boite blanche offrant les fonctionnalités déclarées.

Comme tout composant *Fractal*, un composant *SoftwareUnit* concret peut être créé à partir d'une usine à composants (paragraphe 4.3.2.5). La création d'un composant abstrait est effectuée par une usine à composants spécifique (paragraphe 4.3.2.5).

L'usine à composant On distingue les usines génériques, qui peuvent créer des composants de types divers, et les usines spécialisées, qui ne peuvent générer que des composants de type similaire.

Similairement à l'usine générique à composants *Fractal*, celle dédiée aux composants *SoftwareUnit* requiert, pour la création de composants, une description de leur contenu. Dans notre modèle, cette description est apportée par un nom, identifiant le logiciel à représenter. Ce nom est défini dans le contexte de nommage d'un système *SoftwareUnit*. Il ne doit pas être confondu avec celui du composant *Fractal* dont le rôle est notamment de référencer les sous (et super) composants. Un nom est encodé et décodé à partir du contexte de nommage d'un système *SoftwareUnit* (pour une définition d'un nom, voir la section 4.3.3.1). Par exemple, la forme textuelle d'un nom désignant le paquet *Debian gnugo* sur la machine dont l'adresse IP est *192.168.56.7* est : rpm:/192.168.56.7|apt/gnugo.

L'usine générique est représentée par l'interface SoftwareUnitGenericFactory (définie dans le listing 4.7).

```
package org.ow2.fractal.su.api.factory;
interface SoftwareUnitGenericFactory extends GenericFactory {
    SoftwareUnit newFcInstance (
        Type type, Object controllerDesc, Object contentDesc)
            throws SoftwareUnitInstantiationException;
    SoftwareUnit newFcSoftwareUnitInstance(SoftwareUnitType type,
            String controllerDesc, Name suName)
                throws SoftwareUnitInstantiationException;
}
interface SoftwareUnitFactory extends Factory {
    SoftwareUnitType getFcInstanceType ();
    String getFcControllerDesc();
    Name getFcContentDesc();
    SoftwareUnit newFcInstance ()
            throws SoftwareUnitInstantiationException;
}
```

Listing 4.7 – API d'instanciation

L'usine spécialisée est représentée par l'interface SoftwareUnitFactory. Celle-ci en surcharge les différentes opérations, le type des instances étant SoftwareUnitType, la description du contenu de type Name et l'instance de type SofwareUnit.

Création d'un composant *SoftwareUnit* abstrait La modélisation d'un logiciel en composant peut requérir des opérations de déploiement préalable. De plus, les informations sur un logiciel peuvent être incomplètes lors de la création de sa représentation.

Ce cas ce présente notamment lorsque l'on désire modéliser un logiciel alors que son implémentation ne sera connue qu'ultérieurement au cours du déploiement.

Nous proposons la définition d'un composant *SoftwareUnit* abstrait représentant un logiciel fournissant des fonctionnalités, mais dont la description du contenu est incomplète (eg. le système *SoftwareUnit* est inconnu, le logiciel n'est pas encore référencé dans le contexte du système *SoftwareUnit*). Un tel composant permet alors de s'abstraire de l'environnement de déploiement lors de la conception d'un logiciel.

Le listing 4.8 présente l'API d'introspection d'un composant abstrait.

```
package org.ow2.jonas.su.api.abstractsu;
interface AbstractSoftwareUnit extends SoftwareUnit {
    Name getSignature();
    AbstractSoftwareUnitDeploymentFactory
        getAbstractSoftwareUnitDeploymentFactory();
    Map<?, ?> getConfiguration();
}
interface AbstractSoftwareUnitDeploymentFactory {
    SoftwareUnit newDeployment(Name name,
            IEnvContext envContext, Map<?, ?> config)
    throws DeploymentFactoryException;
}
```

Listing 4.8 – API d'introspection d'un composant abstrait

Un composant abstrait est constitué des éléments suivants :

Une signature identifiant le logiciel qui sera déployé (éventuellement de manière incomplète) ;

Une configuration destinée au déploiement du logiciel ;

Une usine à déploiement qui à partir de la signature et de la configuration du composant abstrait, ainsi que d'un contexte environnemental, crée un composant *SoftwareUnit* concret, correspondant au logiciel déployé.

La signature est représentée par l'interface Name. Son encodage en chaîne de caractères est préfixé par le nom d'un système *SoftwareUnit* (eg. *rpm :/*) ou du mot clé *any :/*, si le système *SoftwareUnit* n'est pas spécifié.

L'usine à déploiement est représentée par l'interface AbstractSoftwareUnitDeploymentFactory.

Un contexte environnemental est représenté par l'interface IEnvContext. Il est fourni par le site cible du déploiement et contient les informations spécifiques à la cible, lesquelles sont requises par l'usine à déploiement. Il permet notamment d'identifier le type du système d'exploitation et de sélectionner le système *SoftwareUnit* adéquat pour effectuer le déploiement.

La figure 4.7 décrit un composant *SoftwareUnit* abstrait, représentant l'installation d'une base de données *MySQL*.

Dans cet exemple, la signature du composant abstrait ne spécifie pas de système *SoftwareUnit* cible : l'usine à déploiement spécifiée, ici un installateur générique de *pa-*

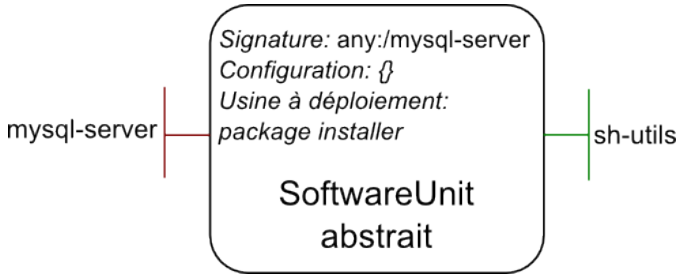


FIGURE 4.7 – Exemple de définition d'un composant abstrait

ckages, se chargera de le résoudre grâce au contexte environnemental fourni au moment du déploiement.

Tout comme un composant *SoftwareUnit*, un composant abstrait comporte également des descripteurs.

Le listing 4.9 définit l'API d'instanciation d'un composant abstrait.

```
package org.ow2.jonas.su.api.abstractsu;
interface AbstractSoftwareUnitFactory {
    AbstractSoftwareUnit newFcAbstractSoftwareUnitInstance(
        SoftwareUnitType type,
        String controllerDesc,
        Name signature,
        AbstractSoftwareUnitDeploymentFactory deploymentFactory,
        Map<?, ?> config)
            throws SoftwareUnitInstantiationException;
}
```

Listing 4.9 – API d'instanciation d'un composant abstrait

L'interface AbstractSoftwareUnitFactory ne dispose que d'une seule opération newFcAbstractSoftwareUnitInstance, permettant de construire un composant abstrait.

4.3.2.6 Typage

Les composants *SoftwareUnit* utilisent un système de type différent de celui défini dans la spécification du modèle à composants *Fractal*.

Contingence et cardinalité Un descripteur client peut exprimer trois types de besoins : dépendance obligatoire (*MANDATORY*), dépendance optionnelle (*OPTIONAL*), ou conflit (*CONFLICT*). Un descripteur client représentant un conflit ne doit pas être lié.

La sémantique de la cardinalité reste inchangée.

Système de type Le type d'un composant *SoftwareUnit* est défini par l'ensemble des types de ses descripteurs. Celui-ci est représenté par l'interface SoftwareUnitType (définie dans le listing 4.10), laquelle étend l'interface ComponentType. Les opérations getFcInterfaceTypes et getFcInterfaceType sont surchargées pour retourner des types de descripteur.

```
package org.ow2.fractal.su.api.type;
interface SoftwareUnitType extends ComponentType {
    DescriptorType[] getFcInterfaceTypes();
    DescriptorType getFcInterfaceType(String name)
        throws NoSuchDescriptorException;
}
interface DescriptorType extends InterfaceType {
    Metadata getFcMetadata();
    Modality getModality();
}
enum Modality {MANDATORY, OPTIONAL, CONFLICT}
interface SoftwareUnitTypeFactory extends TypeFactory {
    DescriptorType createFcDescriptorType (
        String name,
        Metadata metadata,
        boolean isClient,
        Modality modality,
        boolean isCollection)
            throws SoftwareUnitInstantiationException;
    DescriptorType createFcItfType (
        String name,
        String metadata,
        boolean isClient,
        boolean isOptional,
        boolean isCollection)
            throws SoftwareUnitInstantiationException;
    SoftwareUnitType createFcType (
        InterfaceType[] interfaceTypes)
            throws SoftwareUnitInstantiationException;
    SoftwareUnitType createFcSoftwareUnitType(
        DescriptorType... descriptorTypes)
            throws SoftwareUnitInstantiationException;
}
```

Listing 4.10 – API de typage

Le système de type, proposé par la spécification *Fractal*, définit la signature d'une interface comme le nom du type d'une interface langage. Dans cette spécification, nous définissons la signature d'un descripteur comme la forme textuelle de ses méta-données.

Le type d'un descripteur est représenté par l'interface DescriptorType, laquelle étend l'interface InterfaceType. Ainsi, l'interface DescriptorType définit l'opération getFcMetadata afin d'obtenir les méta-données associées au descripteur.

Méta-données Les méta-données expriment les capacités ou besoins des descripteurs. Leur forme textuelle constitue la signature des descripteurs et sont exprimées à l'aide du langage de description d'un système *SoftwareUnit*. Notre modèle propose une représentation générique des méta-données exprimant les capacités et besoins des logiciels.

Une méta-donnée, dans notre modèle, est associée à un ensemble d'attributs et de directives. Un attribut apporte une information permettant de comparer des méta-données (par exemple un numéro de version). Une directive est une instruction destinée à la création de liaisons entre descripteurs (eg. la directive *uses* dans le système *OSGi*, décrite dans la section 4.3.2.4).

Une méta-donnée est représentée par l'interface Metadata (définie dans le listing 4.11)

```
package org.ow2.fractal.su.api.metadata;
interface Metadata {
    String getSignature();
    boolean isCapability();
    boolean isPrimitive();
    String getSoftwareUnitSystemName();
    Collection<Attribute> getAttributes();
    Attribute getAttribute(String name);
    Collection<Directive> getDirectives();
    Directive getDirective(String name);
    boolean isCompatibleWith(Metadata metadata);
}
interface Capability extends Metadata {
}
interface Requirement extends Metadata {
    boolean match(Capability capability);
}
```

Listing 4.11 – API des méta-données

L'opération getSignature retourne la forme textuelle de la méta-donnée. L'opération isCapability indique si la méta-donnée représente une capacité. L'opération isPrimitive indique si la méta-donnée est primitive ou si elle est composée d'autres méta-données. L'opération getSoftwareUnitSystemName retourne le nom du système *SoftwareUnit* auquel appartient cette méta-donnée. Si celle-ci n'appartient à aucun système, le mot clé *any* doit être retourné. Les opérations getAttributes et getAttribute permettent d'accéder aux attributs de la méta-donnée. Les opérations getDirectives et getDirective permettent d'accéder aux directives de la méta-donnée.

Enfin, l'opération isCompatibleWith indique si la méta-donnée est compatible avec une autre. Si les méta-données $c1$ et $c2$ représentent des capacités, $c1$ est compatible avec $c2$ si et seulement si la capacité exprimée par $c1$ implique celle exprimée par $c2$. Si les méta-données $b1$ et $b2$ représentent des besoins, $b1$ est compatible avec $b2$ si et seulement si le besoin exprimé par $b2$ implique celui exprimé par $b1$. Ces définitions seront formalisées au prochain paragraphe.

Les interfaces Capability (représentant une capacité) et Requirement (représentant un besoin), étendant l'interface Metadata, définissent les deux types de méta-donnée. L'interface Requirement définit l'opération match permettant de tester si une capacité donnée satisfait le besoin représenté.

Représentation en terme de logique propositionnelle Les méta-données spécifiques de chaque système *SoftwareUnit* sont traduites dans notre représentation générique des méta-données. À cette fin, nous avons choisi d'utiliser une représentation basée sur la logique propositionnelle. Celle-ci présente l'avantage d'être à la fois simple et suffisamment complète pour supporter les langages de méta-données des systèmes de gestion de modules les plus répandus.

Chaque système *SoftwareUnit* définit un ensemble de variables propositionnelles.

Pour un système *SoftwareUnit* donné, les méta-données d'un descripteur de capacité définissent une interprétation, laquelle est la fonction de valuation dont le domaine est l'ensemble défini par le système *SoftwareUnit*.

Les méta-données d'un descripteur de dépendance définissent des *formules propositionnelles* dont les variables sont définies pour un système *SoftwareUnit* donné.

Si la formule issue d'un descripteur de dépendance est vraie pour l'interprétation issue d'un descripteur de capacité, on dit que la dépendance est résolue.

Par exemple, dans le système *SoftwareUnit Deb*, la dépendance sur le paquet *gnugo*, dont la version doit être supérieure à 2, est représentée par la formule propositionnelle F :

$$F = A \wedge B$$

où A est vraie si le paquet est *gnugo* et B est vraie si la version est supérieure à 2.

Une capacité sur le paquet *gnugo* dont la version est 3 définit une interprétation I satisfaisant A et B, donc F.

Relation de sous-typage En vertu de la spécification *Fractal*, la relation de sous-typage pour les composants *SoftwareUnit* et leurs descripteurs est basée sur la *substituabilité*.

Un type de descripteur d_1 est un sous-type d'un type de descripteur de capacité d_2 si les conditions suivantes sont satisfaites :

- d_1 a le même nom et le même rôle que d_2 ;
- les méta-données de d_1 sont compatibles avec les méta-données de d_2 ;
- si la modalité de d_2 est *obligatoire*, alors la modalité de d_1 est également *obligatoire* ;
- si la modalité de d_1 est *conflit* ou si la modalité de d_2 est *conflit*, alors les deux sont égales ;
- si la cardinalité de d_2 est *collection*, alors la cardinalité de d_1 est également *collection*.

Définition 4.3.1. Une capacité c_1 est compatible avec une capacité c_2 pour le système S si :

$$\forall f \in \mathbb{F}, \quad \mathcal{I}_1(f) \implies \mathcal{I}_2(f)$$

où $\mathcal{I}_1$ est l'interprétation définie par c_1, $\mathcal{I}_2$ est l'interprétation définie par c_2 et $\mathbb{F}$ est l'ensemble des formules propositionnelles pour S.

Un type de descripteur d_1 est un sous type d'un type de descripteur de besoin d_2 si les conditions suivantes sont satisfaites :
- d_1 a le même nom et le même rôle que d_2 ;
- les méta-données de d_1 sont compatibles avec les méta-données de d_2 ;
- si la modalité de d_2 est optionnelle, alors la modalité de d_1 est également *optionnelle* ;
- si la cardinalité de d_2 est *collection*, alors la cardinalité de d_1 est également *collection*.

Définition 4.3.2. Un besoin r_1 est compatible avec un besoin r_2 pour le système S si :

$$\forall \mathcal{I} \in \mathbb{I}, \quad \mathcal{I}(f_1) \implies \mathcal{I}(f_2)$$

où f_1 est la formule propositionnelle définie par r_1, f_2 est la formule propositionnelle définie par r_2 et $\mathbb{I}$ est l'ensemble des interprétations pour S.

Un type de composant *SoftwareUnit* T_1 est un sous type d'un type de composant *SoftwareUnit* T_2 si et seulement si les conditions suivantes sont satisfaites :
- chaque type de descripteur de besoin défini dans T_1 est un sous type du type d'un descripteur défini dans T_2 ;
- chaque type de descripteur de capacité défini dans T_2 est un super type du type d'un descripteur défini dans T_1.

4.3.3 Définition d'un système *SoftwareUnit*

Un système *SoftwareUnit* apporte le support d'une technologie logicielle. Une technologie logicielle peut être associée, entre autre, à un format d'assemblage, un modèle de module ou un modèle d'exécution. Un système de gestion de modules représente une technologie logicielle. Par exemple, la technologie *OSGi* définit le *bundle* comme format d'assemblage et détermine de tels modèles.

Dans la précédente section, nous avons introduit le concept de nom, identifiant le logiciel représenté par un composant *SoftwareUnit*. Nous l'avons défini par rapport au contexte de nommage d'un système *SoftwareUnit*. Dans la section 4.3.3.1, nous proposons une définition de contexte de nommage, permettant d'encoder et de décoder les noms, mais aussi d'accéder aux logiciels d'une même technologie.

Dans la précédente section, nous avons également introduit le concept de méta-données. Celles-ci décrivent les capacités et les besoins de logiciels appartenant à une technologie logicielle. Elles peuvent être représentées soit dans un format spécifique à cette technologie, soit dans le format générique de méta-données que nous avons proposé. Dans la section 4.3.3.2, nous définissons un analyseur de méta-données, permettant ainsi de traduire les méta-données écrites dans le format spécifique vers notre format générique.

Enfin, nous avons proposé d'utiliser les contrôleurs des composants pour gérer la cohérence du modèle avec le logiciel représenté. Un système *SoftwareUnit* peut ainsi définir une membrane spécifique, propageant les reconfigurations, à la fois du modèle vers le logiciel et du logiciel vers le modèle. Notamment, lorsqu'un système de gestion de modules construit un modèle des liaisons entre modules, le contrôleur de liaisons doit être capable de synchroniser les deux modèles. Par exemple, dans la technologie *OSGi*, la plate-forme maintient un modèle des liaisons entre *bundles* et *packages*. Dans la section 4.3.3.3, nous proposons la définition d'un gestionnaire de dépendances, permettant de contrôler la représentation des liaisons, propre à de telles technologies logicielles.

Un système *SoftwareUnit* est représenté par l'interface SoftwareUnitSystem (définie dans le listing 4.12).

```
package org.ow2.fractal.su.api.system;
interface SoftwareUnitSystem {
    String getFcSystemName();
    INamingContext getFcNamingContext();
    FormatError hasProperFormat(
        SoftwareUnitType type, Name name);
    SoftwareUnitGenericFactory getFcSoftwareUnitGenericFactory();
    MetadataParser getFcMetadataParser();
    ResolverHook getFcResolverHook();
    SoftwareUnit createSoftwareUnit(Name name)
            throws SoftwareUnitInstantiationException,
            NamingException, SoftwareUnitSystemException;
}
```

Listing 4.12 – API d'un système de composants SoftwareUnit

L'opération getFcSystemName retourne l'identifiant du système *SoftwareUnit*, dans le contexte de nommage des systèmes *SoftwareUnit*. L'opération getFcNamingContext retourne le contexte de nommage associé au système *SoftwareUnit* représenté par cette interface (voir la section 4.3.3.1). L'opération hasProperFormat permet de vérifier si le type associé au nom d'un logiciel est valide. L'opération getFcSoftwareUnitGenericFactory retourne l'usine à composants *SoftwareUnit* générique. L'opération getFcMetadataParser retourne l'analyseur de méta-données du système (voir la section 4.3.3.2). L'opération getFcResolverHook retourne le gestionnaire de dépendances (voir la section 4.3.3.3).

Enfin, l'opération createSoftwareUnit crée un composant *SoftwareUnit* pour le logiciel dont le nom est indiqué. Cette opération est une facilité évitant de spécifier un type ainsi qu'une description de membrane. Le système crée automatiquement le type correspondant au logiciel spécifié et déduit quelle est la membrane la plus adaptée. Sans cette facilité, la création d'un composant *SoftwareUnit*, à partir d'un nom, peut s'effectuer selon la succession d'étapes suivantes :

1. Extraction des méta-données à l'aide de l'analyseur de méta-données ;
2. Construction du type à l'aide des méta-données et de l'usine à type *SoftwareUnit* ;
3. Création du composant à l'aide de l'usine générique à composant *SoftwareUnit*.

4.3.3.1 Contexte de nommage

Un contexte de nommage définit un espace d'adressage pour un ensemble de logiciel. La définition donnée par la spécification *Fractal* [13] est reprise et raffinée dans notre modèle. Ainsi, un nom doit pouvoir être encodé sous forme de séquence de chaînes de caractères (et par conséquence de chaîne de caractères). Cette représentation des noms permet de modéliser les contextes de nom sous forme d'arbres, un nom étant un chemin dans un arbre, facilitant ainsi leur gestion par l'utilisateur. Si la forme des arbres n'est pas fixée, ceux-ci doivent suivre les règles suivantes :

- la racine est étiquetée avec le nom du système associé à l'arbre ;
- les feuilles sont associées à des composants *SoftwareUnit*.

Un contexte de nom est représenté par l'interface INamingContext (définie dans le listing 4.13).

```
package org.ow2.fractal.su.api;
interface INamingContext {
    Name createName(String name) throws InvalidNameException;
    Object getValue(Name name) throws NamingException;
    boolean exist(Name name);
}
```

Listing 4.13 – API d'un contexte de nommage

Un nom est représenté par l'interface Name (définie dans le listing 4.14).

```
javax.naming;
interface Name
    extends Cloneable, java.io.Serializable, Comparable<Object> {
    ...
    Enumeration<String> getAll();
    String get(int posn);
    ...
}
```

Listing 4.14 – Extrait de l'interface d'un nom

4.3.3.2 Analyseur de méta-données

L'analyseur de méta-données a en charge la construction de la représentation des méta-données à partir d'une signature écrite dans le langage de description des capacités et des besoins du système.

Un analyseur de méta-données est représenté par l'interface MetadataParser (définie dans le listing 4.15).

```
package org.ow2.fractal.su.api.metadata;
interface MetadataParser {
    Capability createCapability(String capability)
            throws MetadataParserException;
    Requirement createRequirement(String requirement)
            throws MetadataParserException;
}
```

Listing 4.15 – API d'analyse des méta-données

L'opération createCapability crée une méta-donnée représentant une capacité à partir d'une chaîne de caractères. L'opération createRequirement crée une méta-donnée représentant un besoin à partir d'une chaîne de caractères. La création des capacités et besoins s'effectue par des opérations différentes, car certains systèmes peuvent ne pas les différencier dans leur forme textuelle.

4.3.3.3 Gestion des dépendances

Certains systèmes de gestion de modules construisent une représentation des liens effectifs entre chaque logiciel, celle-ci étant utilisée à l'exécution. C'est le cas, par exemple, des plate-formes *OSGi*, pour lesquelles les liens entre *package* et *bundle* permettent de construire l'espace des classes visible par le chargeur de classes d'un *bundle*. Dans ce type de système de gestion de modules, un logiciel ne peut être utilisé que si un lien vers celui-ci a été préalablement créé.

Nous proposons la définition d'un gestionnaire de dépendances, lequel est une interface d'administration uniforme permettant de contrôler les liens entre modules. Le contrôleur de liaisons des composants peut utiliser ce gestionnaire afin d'appliquer les modifications de liaisons au niveau modèle. Un système *SoftwareUnit* doit en fournir une implémentation spécifique. Par exemple, le gestionnaire de dépendances du système *OSGi* est implémenté en opérant sur le résolveur de la plate-forme.

Le gestionnaire de dépendances est représentée par l'interface ResolverHook (définie dans le listing 4.16).

```
package org.ow2.fractal.su.api.operation.resolver;
interface ResolverHook {
    void selectForResolution(Descriptor clientDesc,
        Descriptor serverDesc)
            throws ResolverException;
    void unselectForResolution(Descriptor clientDesc,
        Descriptor serverDesc)
            throws ResolverException;
}
```

Listing 4.16 – API de sélection des résolutions

L'opération selectForResolution notifie le gestionnaire de dépendances qu'un des-

cripteur de besoin doit être lié avec un descripteur de capacité. L'opération inverse unselectForResolution notifie le gestionnaire que cette liaison n'est plus souhaitée.

4.3.3.4 Gestionnaire de systèmes *SoftwareUnit*

Un gestionnaire de systèmes offre un accès centralisé aux systèmes. Celui-ci est représenté par l'interface SoftwareUnitSystemManager (définie dans le listing 4.17).

```
package org.ow2.fractal.su.api.system;
interface SoftwareUnitSystemManager {
    SoftwareUnitSystem getFcSoftwareUnitSystem(String susName);
    Collection<SoftwareUnitSystem> getFcSoftwareUnitSystems();
    void addFcSoftwareUnitSystem(SoftwareUnitSystem sus)
        throws IllegalSoftwareUnitSystemException;
    void removeFcSoftwareUnitSystem(String susName)
        throws IllegalSoftwareUnitSystemException;
    String extractSoftwareUnitSystemName(String signature)
        throws MetadataParserException;
}
```

Listing 4.17 – API de gestion des systèmes

Les opérations getFcSoftwareUnitSystem et getFcSoftwareUnitSystems permettent de retrouver des systèmes. L'opération addFcSoftwareUnitSystem permet d'en ajouter un au gestionnaire et à l'inverse l'opération removeFcSoftwareUnitSystem permet d'en retirer. Enfin, l'opération extractSoftwareUnitSystemName permet d'identifier le système pour lequel une méta-donnée est définie.

Afin de faciliter la création de composant *SoftwareUnit*, le composant *Bootstrap* devrait offrir cette interface.

4.3.4 Définition d'opérations de déploiement

Le déploiement est défini comme une séquence d'opérations sur des composants *SoftwareUnit*, chaque opération retournant elle-même des composants *Fractal*.

4.3.4.1 Opérations sur les composants *SoftwareUnit*

Une opération constitue ainsi une étape du processus de déploiement. Une opération peut représenter par exemple l'installation d'un *bundle* dans une plate-forme *OSGi*, la création d'une archive...

Une opération est représentée par l'interface SUOperation (définie dans le listing 4.18).

```
package org.ow2.fractal.su.api.operation;
interface SUOperation<T extends Component> {
    Collection<T> match(Collection<SoftwareUnit> softwareUnits)
        throws OperationException;
    String getFcOperationName();
}
interface IStatefulOperation<T extends Component, Session>
        extends SUOperation<T> {
    Collection<T> match(
        Collection<SoftwareUnit> softwareUnits, Session session)
            throws OperationException;
}
interface ReversibleSUOperation<T extends Component> extends
        SUOperation<T> {
    void matchAgainst(Collection<SoftwareUnit> softwareUnits)
            throws OperationException;
}
```

Listing 4.18 – API d'une opération sur les composants SoftwareUnit

L'opération getFcOperationName retourne le nom de l'opération sur les composants. L'opération match exécute l'opération sur les composants passés en paramètres et retourne un autre ensemble de composants.

L'interface IStatefulOperation représente une opération sur les composants dont l'opération match accepte un état. Le type de cet état est spécifié par l'implémentation de l'opération.

L'interface ReversibleSUOperation représente une opération réversible. L'opération matchAgainst annule les effets de l'opération match.

4.3.4.2 Dépôt à composants *SoftwareUnit*

Un dépôt à composants permet de stocker des composants *SoftwareUnit*. Similairement à un dépôt de *bundles OSGi* (*OSGI Bundle Repository*[3]), un dépôt à composant permet de chercher des composants à partir de besoins. Un dépôt constitue une pièce essentielle du déploiement en permettant de conserver et de retrouver le résultat d'une opération.

Dans ce dernier, les composants sont indexés via un nom et selon une structure hiérarchique (comme illustré sur la figure 4.8). Cette structure hiérarchique provient des contraintes spécifiées sur l'encodage des noms en séquence de chaînes de caractères (voir la section 4.3.3.1).

Un noeud de l'arbre définit une ressource contenant d'autres ressources ou des composants *SoftwareUnit*, lesquels constituent les feuilles de l'arbre. Une ressource est représentée par un composite *Fractal*, comme indiqué sur la figure 4.9.

Les ressources sont liées au contexte de nom du dépôt, ce dernier permettant d'accéder à leur contenu. Une ressource représente un groupe de composants *SoftwareUnit*.

3. RFC 112 : `http://www.osgi.org/download/rfc-0112_BundleRepository.pdf`

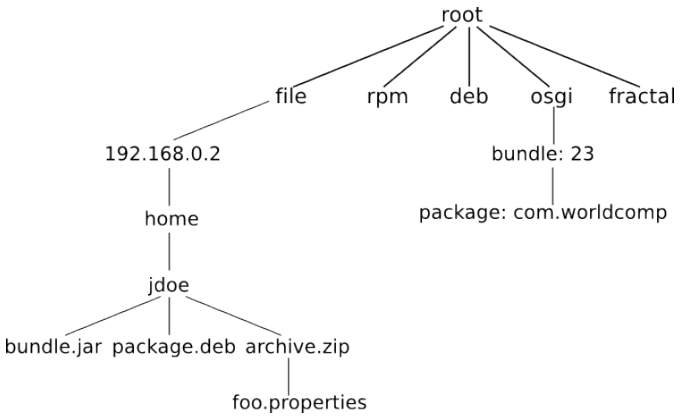


FIGURE 4.8 – Exemple d'un contenu du dépôt

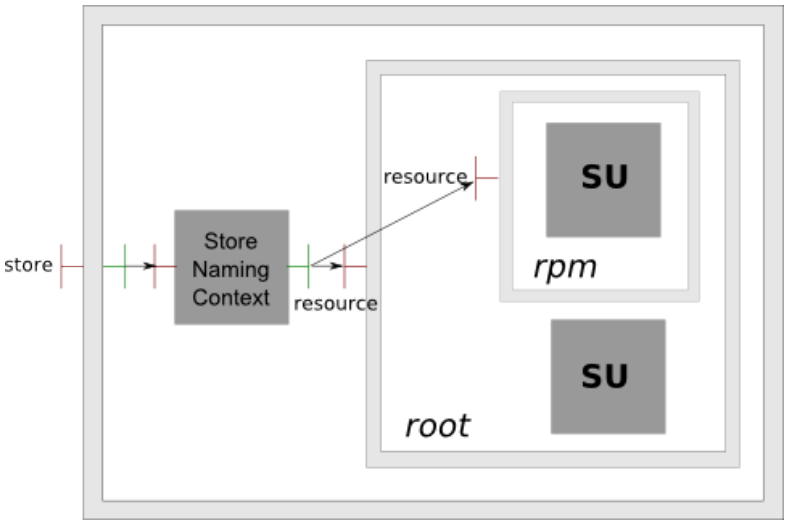


FIGURE 4.9 – Exemple d'implémentation d'un dépôt

Un dépôt est implémenté dans cet exemple comme un composite ayant une interface *store*. Celle-ci est associée à l'interface langage de type SoftwareUnitStore (définie dans le listing 4.19).

```
package org.ow2.fractal.su.api.store;
interface SoftwareUnitStore {
    Collection<String> list(String resourceName)
            throws NoSuchComponentException;
    Collection<String> list(Name resourceName)
            throws NoSuchComponentException;
    Collection<SoftwareUnit> listSoftwareUnits(String resourceName)
            throws NoSuchComponentException;
    Collection<SoftwareUnit> listSoftwareUnits(Name resourceName)
            throws NoSuchComponentException;
    boolean contains(String name);
    boolean contains(Name name);
    Component lookup(String softwareUnitName)
            throws NoSuchComponentException;
    Component lookup(Name softwareUnitName)
            throws NoSuchComponentException;
    void bind(String softwareUnitName, SoftwareUnit installed)
            throws IllegalComponentException;
    void bind(Name softwareUnitName, SoftwareUnit installed)
            throws IllegalComponentException;
    void rebind(String softwareUnitName, SoftwareUnit installed)
            throws IllegalComponentException;
    void rebind(Name softwareUnitName, SoftwareUnit installed)
            throws IllegalComponentException;
    void remove(String name) throws NoSuchComponentException;
    void remove(Name name) throws NoSuchComponentException;
    Collection<SoftwareUnit> removeSoftwareUnits(String name)
            throws NoSuchComponentException;
    Collection<SoftwareUnit> removeSoftwareUnits(Name name)
            throws NoSuchComponentException;
    Set<Descriptor> getCapabilities();
}
```

Listing 4.19 – API d'un dépôt de composant SoftwareUnit

Les opérations de l'interface SoftwareUnitStore ayant un nom en argument acceptent également celui-ci dans sa forme encodée en chaînes de caractères. L'opération list retourne les noms des ressources ou composants *SoftwareUnit* contenus dans la ressource au nom donné. L'opération listSoftwareUnits retourne les composants *SoftwareUnit* contenus dans la ressource au nom donné. L'opération contains indique si la ressource au nom donné existe. L'opération lookup retourne la ressource ou le composant *SoftwareUnit* au nom donné. L'opération bind ajoute un composant *SoftwareUnit* dans le dépôt, pour un nom donné. Si le nom existait déjà, l'opération échoue. L'opération rebind ajoute également un composant *SoftwareUnit* dans le dépôt, mais si le nom indiqué existe déjà, le composant *SoftwareUnit* est remplacé. L'opération remove retire le composant *SoftwareUnit* du dépôt. L'opération getCapabilities retourne toutes les capacités des

composants *SoftwareUnit* enregistrés. Cette dernière méthode sert en particulier pour la résolution de dépendances (voir la section suivante 4.3.4.4).

4.3.4.3 Installabilité

L'installation, définie dans le projet *Edos* [9], peut être représentée comme une opération sur les composants *SoftwareUnit*. Un composant est installable si :

- Toutes les dépendances de ce composant peuvent être résolues avec les composants déjà installés ;
- Aucun composant installé n'entre en conflit.

4.3.4.4 Résolution des dépendances

Dans la section 4.3.2 sur la modélisation de logiciels, nous proposons une manière de décrire des dépendances sur des logiciels hétérogènes. Dans le cadre du déploiement, nous cherchons maintenant à automatiser leur résolution, c'est-à-dire la création des liaisons entre composants. Nous proposons deux politiques complémentaires de résolution de dépendances, lesquelles pouvant être combinées et complétées. Celles-ci sont implémentées sous forme d'opérations sur les composants *SoftwareUnit*.

Politique basée sur un dépôt Cette politique cherche à lier les interfaces clientes à partir des capacités disponibles dans un dépôt. Celle-ci nécessite que le logiciel soit déjà installé.

Politique basée sur les composants abstraits La politique de déploiement de composants *SoftwareUnit* basée sur les composants abstraits permet d'automatiser l'installation de composants non disponibles dans le dépôt. Elle est définie comme une opération parcourant la fermeture transitive des dépendances d'un composant abstrait, afin de substituer les composants abstraits avec leur composant résultant du déploiement. Cette opération est décrite par l'algorithme AbstractSUResolver (en pseudo-code et ne tenant pas compte des composants partagés).

```
L ← ∅
pour tous les composants c liés à a faire
    si c est un composant abstrait alors
        e ← AbstractSUResolver(d)
        L ← L + e
    fin
fin
d ← DeployAbstractSU(a)
UpdateBindings(d,L)
retourner d
```

Fonction AbstractSUResolver(AbstractSoftwareUnit a) : SoftwareUnit

La fonction *DeployAbstractSU* invoque l'usine à déploiement du composant abstrait donné en argument.

Le parcours est effectué en profondeur d'abord, afin que les dépendances du composant soient déployées avant celui-ci.

La solution de substitution du composant abstrait est préférée à celle de sa modification, car cette dernière approche implique une contrainte sur l'implémentation des composants *SoftwareUnit*. En effet, la membrane de ceux-ci peut différer selon le logiciel représenté, c'est le cas par exemple d'un *bundle OSGi* (voir la section 5.4.2.3). La membrane du composant abstraite doit donc pouvoir être modifiée, ce qui implique une implémentation adéquate, telle que *AOKell*[4] [56]. Le composant concret utilisé pour la substitution doit avoir un sous-type de celui du composant abstrait.

4.3.4.5 Lancement

Un lanceur est une opération particulière ne retournant pas un composant *SoftwareUnit*, mais un composant d'exécution *Fractal*. Le résultat d'un lanceur ne peut être passé à une autre opération, celle-ci n'acceptant que des composants *SoftwareUnit*.

Un lanceur peut être implémenté de deux manières :

- Soit le composant *Fractal* est créé à partir d'un composant *SoftwareUnit* contenant le code de celui-ci (eg. un composant Fractal natif) ;
- Soit le système *SoftwareUnit* du composant offre une représentation du logiciel à l'exécution, sous forme de composants *Fractal* (eg. un composant Fractal représentant les services OSGi offerts et requis par un bundle).

Un composant résultant d'une opération de lancement peut être attaché au composant *SoftwareUnit* grâce au *SU contrôleur*, comme l'illustre la figure 4.10.

Le *SU contrôleur* est représenté par l'interface SoftwareUnitRuntimeController (définie dans le listing 4.20).

```
package org.ow2.fractal.su.api.control.runtime;
interface SoftwareUnitRuntimeController {
    SoftwareUnit getSoftwareUnit();
    void setSoftwareUnit(SoftwareUnit softwareUnit);
}
```

Listing 4.20 – Interface du contrôleur d'exécution d'un composant Fractal

L'opération getSoftwareUnit retourne le composant *SoftwareUnit* qui a permis la création du composant auquel appartient ce contrôleur.

4. AOKell : `http://fractal.ow2.org/aokell/index.html`

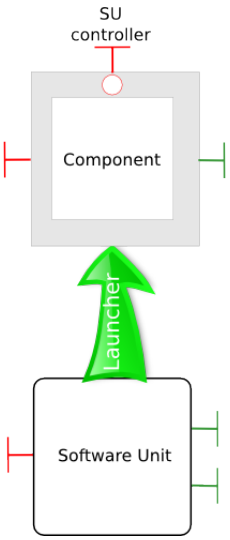


FIGURE 4.10 – Exécution d'un composant *Fractal* défini par un composant *SoftwareUnit*

4.4 Langages pour le déploiement

L'administration fondée sur les langages offre une vision globale des systèmes administrés, incluant la description de leurs dépendances, la gestion de leur déploiement et de leur cycle de vie. Dans notre contexte, les langages peuvent être utiles pour décrire un assemblage lors de la sortie d'un logiciel, mais également pour définir des liens d'exécution dans un système vivant. Par exemple, une description architecturale des liaisons entre *bundles OSGi* déployés sur une plate-forme *OSGi* peut être appliquée par le système de résolution de dépendances de celle-ci (et ce malgré son comportement autonome).

La section précédente 4.3 a défini les API de notre modèle. Au dessus de celles-ci, nous pouvons maintenant construire des langages afin de simplifier le déploiement. Deux points de vue peuvent être adoptés :

- La description architecturale décrit les composants, leurs liaisons et leur composition ;
- Le plan de déploiement décrit les opérations.

4.4.1 Description architecturale

La description architecturale donne une photographie instantanée des liaisons et du contenu des composants. Celle-ci peut être décrite avec le langage *Fractal ADL*[5] [45].

Nous proposons une extension de ce langage afin de définir des composants *SoftwareUnit*, mais aussi de construire n'importe quel composant *Fractal* à partir de ces derniers.

5. Fractal ADL : `http://fractal.ow2.org/fractaladl/`

4.4.1.1 Définition d'un composant *SoftwareUnit*

La définition d'un composant *SoftwareUnit* diffère légèrement de celle d'un composant *Fractal* traditionnel. Le descripteur de contenu d'un composant est ainsi obligatoire, que le composant soit primitif ou composite. De plus, le descripteur de contenu n'est pas le nom d'une classe d'implémentation, mais le nom encodé en chaîne de caractères du logiciel. Pour marquer cette différence de sémantique, l'élément *content* est remplacé par le nouvel élément *su-content*. Le listing 4.21 présente un exemple de définition d'un composant *SoftwareUnit* représentant un *bundle OSGi*.

```
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
    "classpath://org/ow2/jonas/su/adl/xml/standard-su.dtd">
<definition name="examples.adl.helloworld.ServerJAR">
  <interface name="vSysOutPrinter" role="server"
      signature="virtual:/sysOutPrinter" />
  <su-content
      description="file:/home/jdoe/osgi-helloworld-server.jar"
      system="osgi" />
</definition>
```

Listing 4.21 – Définition d'un composant SoftwareUnit représentant un bundle OSGi

L'élément *su-content* est composé dans cet exemple de deux arguments :

description contenant le nom encodé en chaîne de caractères du logiciel ;

system contenant le nom du système *SoftwareUnit* à utiliser pour décoder ce nom.

L'interface nommée *vSysOutPrinter* illustre la possibilité d'ajouter des capacités (ou des besoins) arbitrairement et indépendemment du système utilisé pour décoder le nom.

L'ADL *Fractal* peut également être utilisé pour la définition de composite *SoftwareUnit*, comme illustré sur le listing 4.22.

```
<definition name="org.ow2.jonas.su.service.WebJavaEE5">
    <component name="dp-web-base">
        <su-content description="url://web-base.xml@jojoservices"
                    system="file"
                    launcher="deploymentplan-loader" />
    </component>
    <component name="dp-web-tomcat6">
        <su-content description="url://web-tomcat6.xml@jojoservices"
                    system="file"
                    launcher="deploymentplan-loader" />
    </component>
    <component name="dp-web-jetty6">
        <su-content description="url://web-jetty6.xml@jojoservices"
                    system="file"
                    launcher="deploymentplan-loader" />
    </component>
    <group reference="sud:/JOnAS/jonas-web-container-tomcat-6.0-unpack">
        <interface name="lib" role="server" signature="jonas:/conf" />
    </group>
    <group reference="sud:/JOnAS/jonas-web-container-jetty-6.1-unpack">
        <interface name="lib" role="server" signature="jonas:/conf" />
    </group>
    <su-content description="service/web" system="jonas" />
</definition>
```

Listing 4.22 – Définition d'un composite SoftwareUnit représentant un service technique JOnAS

Dans cet exemple, un composite *SoftwareUnit* est déclaré, celui-ci représentant le service technique *web* pour *Java EE 5*. Ce service est composé d'au moins trois composants, lesquels sont des composants *SoftwareUnit* représentant des plans de déploiement *JOnAS*. L'élément **group** n'existe pas dans la grammaire *Fractal ADL* standard. Il définit un ensemble de composants, lequel est référencé dans le dépôt à composants (un groupe étant stocké dans le dépôt sous forme de ressource comme décrit dans la section 4.3.4.2). Les groupes sont utilisés dans l'exemple pour ajouter les fichiers de configurations du service *web*. L'élément **interface** imbriqué permet d'associer, à tous les composants du groupe, un descripteur de capacité indiquant que ces fichiers ont la valeur de configuration dans le système *SoftwareUnit JOnAS* (cette capacité est ensuite interprétée lors de l'assemblage d'un profil *JOnAS*, comme il sera expliqué dans le chapitre 6).

4.4.1.2 Lancement d'un composant *SoftwareUnit*

Un lanceur permet de construire un composant *Fractal* traditionnel à partir de composants *SoftwareUnit* (voir la section 4.3.4.5). Le listing 4.23 présente un exemple de définition de composants *Fractal* résultant du lancement de composants *SoftwareUnit*.

```
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
    "classpath://org/ow2/jonas/su/adl/xml/standard-su.dtd">
<definition
        name="examples.adl.helloworld.HelloWorldMixed">
  <interface name="r" role="server" signature="java.lang.Runnable" />
  <component name="client">
    <interface name="r" role="server" signature="java.lang.Runnable" />
    <interface name="s" role="client"
        signature="examples.osgi.helloworld.api.Printer" />
    <su-content
      description="fractal:/adlDesc/examples.fractal.helloworld.comp.Client"
      system="fractal"
      launcher="fractal-launcher(start=false,adl.backend=su)" />
  </component>
  <component name="server">
    <interface name="examples.osgi.helloworld.api.Printer"role="server"
        signature="examples.osgi.helloworld.api.Printer" />
    <su-content
      description="file:/home/jdoe/osgi-helloworld-server.jar"
      system="osgi" launcher="osgi-bundle-installer;osgi-bundle-launcher" />
  </component>
  <binding client="this.r" server="client.r" />
  <binding client="client.s"
        server="examples.osgi.helloworld.api.Printer" />
  <controller desc="liteComposite" />
</definition>
```

Listing 4.23 – Définiton de composants Fractal résultant du lancement de composants SoftwareUnit

Cet exemple illustre la création de deux composants nommés *client* et *server*.

Le composant *client* est créé à partir d'un composant *SoftwareUnit* représentant un fichier *ADL*. Celui-ci est défini dans le système *Fractal* et est lancé avec l'opération *fractal-launcher*, le composant *client* étant le résultat de l'opération.

Le composant *server* est créé à partir d'un composant *SoftwareUnit* représentant un *bundle OSGi*. Ce dernier composant est défini dans le système *OSGi* et est lancé avec une succession d'opérations :

osgi-bundle-installer installe le *bundle* dans la plate-forme *OSGi* courante ;

osgi-bundle-launcher démarre le *bundle*.

Le résultat de l'opération *osgi-bundle-installer* est un nouveau composant *SoftwareUnit*, mais celui n'apparaît pas car il est utilisé implicitement comme argument de l'opération *osgi-bundle-launcher*. La prochaine section 4.4.2 présentera un autre langage se focalisant sur les étapes du déploiement.

Les interfaces serveurs du composant *server* résultant de l'opération représentent les services enregistrés lors de l'activation du *bundle*.

Le comportement de ces opérations sera précisé dans la section 5.4.2, consacrée au système *OSGi*.

4.4.2 Plan de déploiement paramétré

La description architecturale ne nous permet pas de définir de nouvelles opérations de déploiement. Or, nous aimerions être capable de décrire des opérations composites représentant des plans de déploiement paramétrés et de les réutiliser dans des descriptions architecturales (notamment via l'attribut *launcher* de l'élément *su-content* décrit précédemment). Les langages que nous observerons, dans cette section, se focalisent sur les étapes du déploiement, l'élément central étant une opération.

4.4.2.1 Le langage *SUD*

Le langage *SUD* (pour *SU Descriptor*) est dédié à la définition d'opérations composites. Ce langage est basé sur un schéma XML [6] définissant deux éléments racines alternatifs :

<su-script> destiné à l'invocation d'opérations ;

<group> destiné à la définition de composants ou d'opérations composites.

Chacun de ces éléments peut contenir les quatre sous éléments suivants :

<operation> définissant une opération ;

<software-unit> définissant un composant *SoftwareUnit* ;

<group> définissant un ensemble de composants ;

<reference> sélectionnant des composants (par exemple, à l'aide d'une expression *FScript*).

Ces élément peuvent être eux-mêmes imbriqués. Le tableau 4.1 définit la sémantique de la relation de sous-élément.

6. Schéma SUD : `http://pub.bouzo.net/files/ns/sud-1.0.xsd`

Éléments	Sous éléments			
	operation	software-unit	group	reference
operation	le résultat définit des arguments	argument	arguments	la sélection définit des arguments
software-unit	le résultat définit des sous-composants	sous-composant	sous-composants	la sélection définit des sous-composants
group	le résultat appartient à l'ensemble	appartient à l'ensemble	sous-ensemble	la sélection appartient
reference	le résultat appartient au domaine de sélection	appartient au domaine de sélection	appartient au domaine de sélection	sélection appartient au domaine de sélection

TABLE 4.1 – Sémantique de la relation de sous élément

L'élément *software-unit* permet de définir explicitement le contenu d'un composite, via le sous élément *sub-software-units*, à l'aide des éléments définis dans le tableau 4.1. Ce contenu peut également être découvert automatiquement, en spécifiant le sous-élément *file* lequel identifie le logiciel à représenter.

Le listing 4.24 propose un exemple de définition de groupe.

```
<group id="jonas-autodeploy-bundles">
    <software-unit id="org.ow2.jonas.osgi:javaee-api:jar" system="osgi">
        <file xsi:type="m2:maven2-deploymentType">
            <m2:groupId>org.ow2.jonas.osgi</m2:groupId>
            <m2:artifactId>javaee-api</m2:artifactId>
            <m2:version>5.2.0-M4-SNAPSHOT</m2:version>
            <m2:type>jar</m2:type>
        </file>
    </software-unit>
</group>
```

Listing 4.24 – Définition d'un composant SoftwareUnit représentant un bundle OSGi

L'élément *group* définit l'ensemble de composants identifié dans cet exemple par le nom *jonas-autodeploy-bundles*. L'élément *software-unit* définit le seul composant du groupe, celui-ci étant construit à partir d'un artefact *Maven*. L'attribut *system* indique quel système *SoftwareUnit* utiliser pour la création du composant (ici *OSGi*).

Le listing 4.25 est un extrait du descripteur d'assemblage de la plate-forme *OSGi Apache Felix* et illustre la définition d'invocation d'opérations.

```
<su-script>
  <operation id="Felix_assembly" name="file-assembly-installer">
    <configuration>
      <formats>
        <format>zip</format>
        <format>dir</format>
      </formats>
      <target>/home/jdoe/felix-framework</target>
    </configuration>
    <software-unit id="My-Felix" name="felix-framework" system="file">
      <sub-software-units>
        <software-unit id="felix_conf" name="conf" system="file">
          <sub-software-units>
            <reference id="conf-files">
              <filter-ref>
                fscript:find-itf($this,"file-config.properties")
              </filter-ref>
              <operation id="Felix_unpack" name="file-unpack-installer">
                <reference id="felix_main">
                  <id-ref>
                    Felix/sources:jar
                  </id-ref>
                </reference>
              </operation>
            </reference>
          </sub-software-units>
        </software-unit>
      </sub-software-units>
    </software-unit>
  </operation>
</su-script>
```

Listing 4.25 – Extrait de la définition d'un composant SoftwareUnit représentant un assemblage de la plate-forme *Apache Felix*

Dans cet exemple, l'opération *file-assembly-installer* prend comme argument un *SoftwareUnit* composite nommé *felix-framework* défini dans le système *SoftwareUnit File*. Ce composant contient un autre composite nommé *conf*, également défini dans le système *File*. Ce dernier contient le composant appartenant également à l'archive *Felix/sources :jar* et possédant le descripteur *file-config.properties*.

Le chargeur de fichiers *SUD* construit une représentation du plan de déploiement à base de composants *SoftwareUnit* définis dans le système *SUD*. La figure 4.11 illustre la représentation d'un plan de déploiement.

Chacun des éléments décrits dans le tableau 4.1 est représenté par un composants *SoftwareUnit SUD*. Un composant *SUD* représentant une opération possède des descripteurs de besoin, pouvant être liés à des composants *SUD* représentant les arguments de l'opération. Un composant *SUD* représentant un groupe possède des descripteurs de besoin, pouvant être liés à des composants *SUD* représentant des éléments appartenant au groupe. Un composant *SUD* représentant un composant *SoftwareUnit* possède des

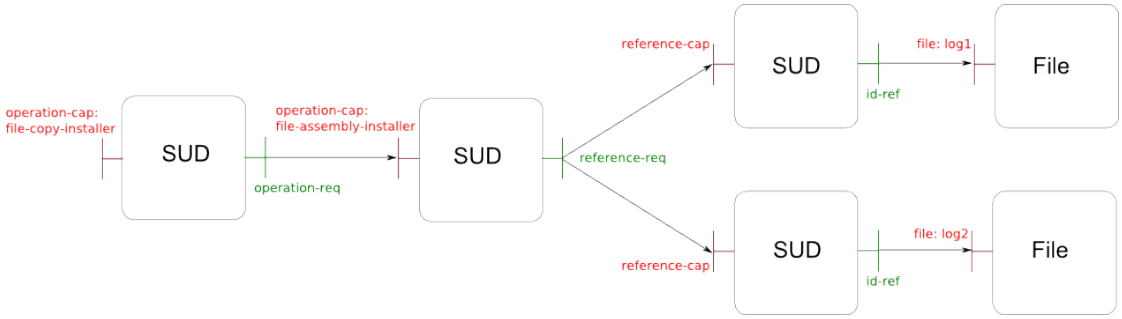


FIGURE 4.11 – Représentation d'un plan de déploiement *SUD* sous forme de composants *SoftwareUnit*

descripteurs de besoin pouvant être liés à des composants *SUD* représentant des sous-composants du composant *SoftwareUnit*. Enfin, un composant *SUD* représentant une référence peut être lié à des composants correspondant à la sélection donnée.

Le plan de déploiement modélisé est exécuté par une opération parcourant l'arbre de composants *SUD*, en profondeur d'abord. Chaque composant *SUD* est substitué selon les règles suivantes :

Un composant représentant une opération est remplacé par les composants *SoftwareUnit* retournés par l'exécution de l'opération ;

Un composant représentant un groupe est remplacé par les composants *SoftwareUnit* appartenant au groupe ;

Un composant représentant un composant SoftwareUnit est remplacé par le composant *SoftwareUnit* représenté ;

Un composant représentant une référence est remplacé par les composants *SoftwareUnit* issus de la sélection.

Dans l'exemple illustré par la figure 4.11, le résultat de l'exécution de ce plan de déploiement est la copie de l'assemblage des fichiers *log1* et *log2*.

Les composants *SUD* peuvent être stockés comme n'importe quel autre composant dans un dépôt, favorisant ainsi la réutilisation de plans de déploiement.

L'utilisation de notre modèle, pour représenter des plans de déploiement, illustre que celui-ci n'est pas uniquement destiné à la représentation de logiciels patrimoniaux.

4.4.2.2 Déploiement par des scripts

FScript fournit un langage plus puissant que *SUD* pour décrire des plans de déploiement à base composants *SoftwareUnit*. Ainsi, *SUF* étend le modèle *Fractal* de *FScript*, afin notamment de considérer les opération sur les composants *SoftwareUnit* comme des primitives natives.

L'annexe B contient un exemple de script, celui-ci étant expliqué dans le chapitre 7.

4.5 Conclusion

Dans ce chapitre, nous avons répondu aux exigences sur le modèle d'abstraction présentées dans le chapitre 3. Notre solution se base sur le modèle à composants *Fractal SoftwareUnit*. Celui-ci est une réponse à l'hétérogénéité des logiciels et de leurs cibles de déploiement. Ce modèle propose en outre de capturer les dépendances et le contenu de logiciels avec un faible niveau de granularité. Pour cela, notre approche s'est basée sur le modèle à composants réflexifs *Fractal*. Ce modèle simplifie la définition d'interfaces d'administration, permettant d'introspecter et de modifier les logiciels directement à partir du composant les représentant.

Nous avons proposé l'utilisation de langages afin de construire des instances de ce modèle. L'utilisation d'une description architecturale est ainsi une solution pour la définition d'assemblage de logiciels (appliquée dans le chapitre 6) et plus généralement pour le déploiement d'applications (appliquée dans le chapitre 7).

Le prochain chapitre expliquera les choix dans la mise en œuvre du modèle et des langages au sein de l'intergiciel *SU Framework*.

Chapitre 5

Mise en œuvre

Sommaire

Ce chapitre décrit l'implémentation du modèle à composants *Fractal SoftwareUnit* au sein du *SU Framework*. Dans la section 5.1, nous présentons l'architecture générale de celle-ci. Dans la section 5.2, nous expliquons comment nous avons utilisé conjointement le modèle à composants *Fractal* avec une plate-forme *OSGi*. Dans la section 5.3, nous présentons les besoins de ce canevas en terme de configuration. Puis dans la section 5.4, nous décrivons différents systèmes *SoftwareUnit* intégrés dans *SUF*. Enfin dans la section 5.5, nous exposons nos choix pour la distribution du déploiement.

5.1 Architecture générale

SUF est une application basée sur des composants *Fractal*. Les systèmes *SoftwareUnit* et les opérations sont implémentés à l'aide de composants. La figure 5.1 présente un exemple d'implémentation du système *SoftwareUnit OSGi*.

Un système *SoftwareUnit* est défini comme un composite contenant une implémentation du système *SoftwareUnit* sous forme de composant primitif, et des opérations fournissant des primitives de déploiement pour les logiciels supportés par ce système. Les autres éléments du canevas, tels que le gestionnaire de systèmes et le dépôt de composants, sont également définis à l'aide de composants *Fractal*.

Nous avons choisi *OSGi* comme plate-forme d'exécution afin de profiter du format de paquetage et du chargement dynamique. Les systèmes *SoftwareUnit* et les opérations peuvent être ainsi chargés dynamiquement. La section 5.2 décrira les difficultés

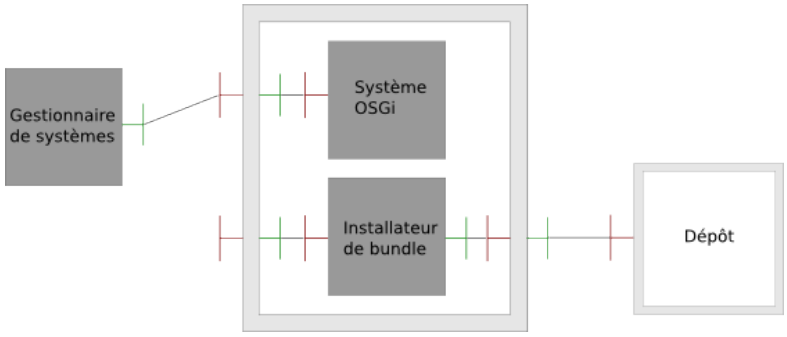


FIGURE 5.1 – Exemple d'implémentation d'un système *SoftwareUnit*

rencontrées pour l'intégration de l'implémentation *Java* de *Fractal*[1] avec la plate-forme *OSGi*.

5.2 Intégration de *Fractal* et *OSGi*

L'utilisation d'*OSGi* comme plate-forme d'exécution de *Fractal* a déjà été relatée dans un certain nombre de travaux. Après avoir analysés ceux-ci, nous expliquerons nos choix pour l'intégration de ces deux technologies.

5.2.1 Contexte

T. Abdellatif et al. [3] propose de conditionner le code d'implémentation des composants *Fractal* dans des *bundles* accompagnés de fichiers ADL décrivant les composants. Le code de la membrane et de la configuration, défini dans l'ADL, est ainsi généré dynamiquement au déploiement.

FROGI [25] définit un composant *Fractal* à services selon l'approche des modèles à composants orientés services [17, 31]. Selon cette approche, un service fournit des fonctionnalités et se caractérise par un contrat. Un composant implémente un contrat (ie. une spécification de service) et la composition peut être effectuée à partir d'un ensemble de contrats, lesquels permettent de sélectionner les composants constituant le composite. La création des liaisons n'est effectuée qu'à l'exécution, en interrogeant le registre à services (comme illustrée par la figure 5.2), rendant ainsi la création de liaisons explicites inutile.

Comme dans l'approche précédente, le code des composants *Fractal* est empaqueté dans des *bundles OSGi*. Dans *FROGI*, les interfaces des composants sont publiées dans le registre à services. Les liaisons entre instances de composant sont ainsi réalisées selon la technique de courtage proposée par l'approche orientée service. Le cycle de vie du composant peut contrôler celui du *bundle*, néanmoins parce que chaque *bundle* possède

1. Julia : `http://fractal.ow2.org/java.html`

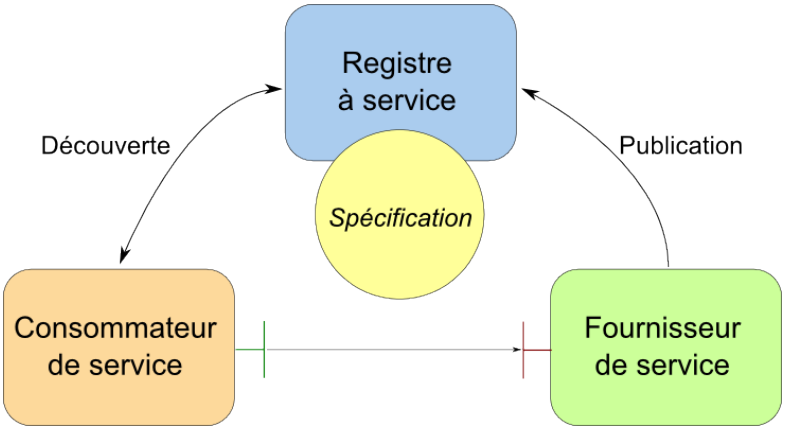


FIGURE 5.2 – Composant *Fractal* avec une liaison de type service

un cycle de vie autonome, le cycle de vie d'un composant est assujetti aux *bundles* dont il dépend. La figure 5.3 décrit de tels composants *Fractal* à services.

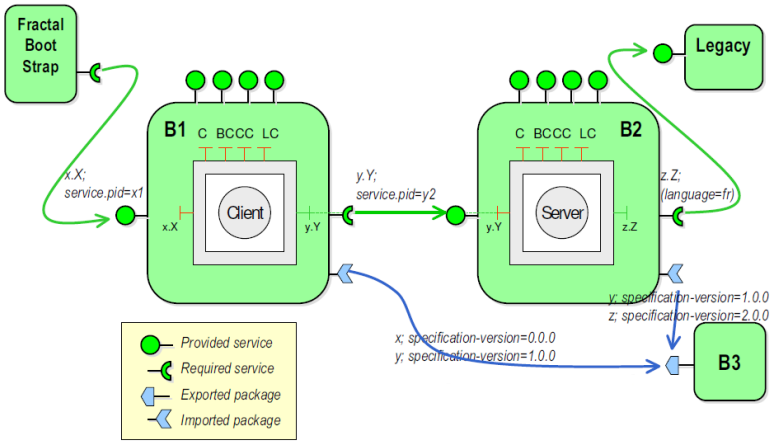


FIGURE 5.3 – Application *Fractal* conditionnée avec des *bundles*

Pour le développement des composants implémentant le canevas, nous reprenons la définition des composants sous forme de *bundle* proposée par les deux approches. Néanmoins, nous évitons tout chargement de code à l'exécution pour des raisons énoncées dans la section suivante. L'empaquetage des composants sous forme de *bundle* requiert de ne plus créer de liaison par référence entre des composants provenant de *bundles* différents. Une liaison entre deux composants implémentés dans des *bundles* différents doit être réalisée via un service. En effet, si un *bundle* est retiré de la plate-forme, l'interface cliente sera associée à une référence invalide. Pour cette raison, et afin de faciliter l'intégration des outils purement *OSGi*, nous utiliserons, pour l'implémentation

de canevas, des composants *Fractal* à services selon l'approche proposée par *FROGI*.

5.2.2 Utilisation conjointe de différentes implémentations *Fractal*

Le modèle à composants *Fractal* est associé à deux types d'utilisation dans notre canevas :

- Les composants exécutant les entités du canevas, telles que les systèmes *SoftwareUnit*, les opérations ou le dépôt ;
- Les composants *SoftwareUnit* créés à l'aide du canevas.

Les composants exécutant les entités du canevas sont implémentés avec *Julia*, l'implémentation de référence pour le langage *Java*. Néanmoins, celle-ci génère dynamiquement le code des membranes à l'exécution, causant de fréquents problèmes de chargement de classes. En effet, la génération de code peut induire de nouveaux imports de *packages*, lesquels ne sont pas réifiés par la plate-forme *OSGi*. Afin de palier à ce problème, le code d'implémentation des composants constituant notre canevas est généré avant l'exécution, avec l'outil *Juliac*[2]. Ainsi, la plate-forme *OSGi* a connaissance des imports exacts au chargement du *bundle*.

Si les composants d'implémentation du canevas sont définis par le concepteur de celui-ci, les composants *SoftwareUnit* sont eux définis par les utilisateurs de ce canevas. Leur définition est effectuée à l'exécution du canevas, via les API de notre modèle. La génération du code de la membrane pour les composants *SoftwareUnit* avant l'exécution du canevas n'est donc pas possible. Nous avons donc développé une implémentation à base de proxy *Java*[3]. Cette implémentation est une évolution de *FracLite*[4] adaptée à *OSGi*. Celle-ci est décrite dans la section 5.2.4.

Deux implémentations différentes de *Fractal* sont donc utilisées. Or, la spécification *Fractal* impose le patron de conception *singleton* [34] pour l'obtention d'un composant *Bootstrap*. Cette approche empêchant l'existence simultanée de plusieurs implémentations, nous avons repris l'outillage disponible dans le projet *Fractal Util*[5], dans lequel est défini un composant *Bootstrap* supportant des implémentations multiples de *Fractal*. Ce travail a été adapté pour que l'usine à composants définie dans chacune des implémentations soit enregistrée comme service dans le registre *OSGi*, avec comme propriété de service fractal.provider. Une implémentation de *Fractal* peut ainsi être ajoutée dynamiquement.

2. Juliac : `http://fractal.ow2.org/juliac/index.html`
3. `http://download.oracle.com/javase/6/docs/technotes/guides/reflection/proxy.html`
4. FracLite : `https://code.google.com/p/fraclite/`
5. Fractal Util : `svn://svn.forge.objectweb.org/svnroot/fractal/trunk/fractalutil`

5.2.3 Développement des composants du canevas

Dans la section 5.2.1, nous expliquons notre choix des composants à services pour l'implémentation du canevas. Dans cette section nous décrivons la manière dont nous les avons développés, ainsi que leur utilisation.

5.2.3.1 Définition de composants à services

La création de liaisons dynamiques entre composants *Fractal* est facilitée par l'utilisation d'une usine à liaisons définie dans le projet *Fractal BF* (pour *Fractal Binding Factory*) [6]. Cet outil permet de créer des liaisons en utilisant des protocoles divers (eg. *RMI*, *WebServices*), sans modifier les composants client et serveur. *Fractal BF* fonctionne selon le patron de conception *export-bind* [7]. L'opération *bind* crée un composant de type proxy et le lie au composant client. Si le protocole utilisé repose sur une approche à service, le proxy constitue le consommateur de services, décrit dans le schéma 5.2. Selon cette approche, l'opération *export* construit le fournisseur de services et publie un service représentant l'interface serveur dans le registre à services.

En *OSGi*, le proxy peut être implémenté avec un objet de type *ServiceTracker* [8]. L'opération *export* est directement associée à la publication de l'interface serveur dans le registre (pas besoin de créer de *skeleton* comme en *RMI*).

L'export et la liaison par *Fractal BF* peuvent être spécifiés dans l'ADL standard *Fractal*. Le listing 5.1 contient l'ADL du composant implémentant le système *OSGi*, celui-ci utilisant *Fractal BF* pour interagir avec le registre à services.

L'interface serveur sus−osgi du composant OSGiSUSystem, représentant le système *OSGi*, est enregistrée comme service *OSGi* par le connecteur *OSGi*. Les autres interfaces exportées sont des opérations. Celles-ci sont décrites dans la section 5.4.2 consacrée au système *OSGi*. Les interfaces clientes store des composants OSGiBundleContextInstaller et OSGiBundleResolver sont liées au service *OSGi* représentant le dépôt à composants.

6. Fractal BF : `svn://svn.forge.objectweb.org/svnroot/fractal/trunk/fractal-bf`
7. `http://proton.inrialpes.fr/~krakowia/MW-Book/Chapters/Naming/naming.html`
8. `http://www.osgi.org/javadoc/r4v42/org/osgi/util/tracker/ServiceTracker.html`

```
<definition name="org.ow2.jonas.su.osgi.comp.system.OSGiSUSystemComp">
  <component name="OSGiSUSystem"
    definition="org.ow2.jonas.su.osgi.comp.system.OSGiSUSystem" />
  <component name="OSGiBundleContextInstaller"
definition="org.ow2.jonas.su.osgi.comp.system.OSGiBundleContextInstaller"/>
  <component name="OSGiBundleResolver"
    definition="org.ow2.jonas.su.osgi.comp.system.OSGiBundleResolver" />
  <component name="OSGiBundleLauncher"
    definition="org.ow2.jonas.su.osgi.comp.system.OSGiBundleLauncher" />
  <binding client="OSGiBundleContextInstaller.sus-osgi"
    server="OSGiSUSystem.sus-osgi" />
  <binding client="OSGiBundleResolver.sus-osgi"
    server="OSGiSUSystem.sus-osgi" />
  <controller desc="compositeBundled"/>
  <exporter type="osgi" interface="OSGiSUSystem.sus-osgi">
    <parameter name="objectClass"
      value="org.ow2.jonas.su.osgi.system.OSGiSUSystem,
        org.ow2.jonas.su.api.system.SoftwareUnitSystem,
        org.objectweb.fractal.api.Interface" />
    <parameter name="properties" value="sus.name=osgi" />
  </exporter>
  <exporter type="osgi"
    interface="OSGiBundleContextInstaller.osgi-bundle-installer">
    <parameter name="objectClass"
      value="org.ow2.jonas.su.api.operation.SUOperation,
        org.objectweb.fractal.api.Interface" />
    <parameter name="properties"
      value="operation.name=osgi-bundle-installer" />
  </exporter>
  <exporter type="osgi"
    interface="OSGiBundleResolver.osgi-bundle-resolver">
    <parameter name="objectClass"
      value="org.ow2.jonas.su.api.operation.SUOperation,
        org.objectweb.fractal.api.Interface" />
    <parameter name="properties"
      value="operation.name=osgi-bundle-resolver" />
  </exporter>
  <exporter type="osgi"
    interface="OSGiBundleLauncher.osgi-bundle-launcher">
    <parameter name="objectClass"
      value="org.ow2.jonas.su.api.operation.SUOperation,
        org.objectweb.fractal.api.Interface" />
    <parameter name="properties"
      value="operation.name=osgi-bundle-launcher" />
  </exporter>
  <binder type="osgi" interface="OSGiBundleContextInstaller.store" />
  <binder type="osgi" interface="OSGiBundleResolver.store" />
  <binder type="osgi" interface="OSGiBundleResolver.genericFactory" />
</definition>
```

Listing 5.1 – ADL du composant implémentant le système OSGi

5.2.3.2 Chargement des composants

Pour le chargement des composants du canevas, nous appliquons le patron de conception *Extender*[9]. Ce patron de conception offre un cadre simple et robuste pour le chargement de *bundles* fournissant une fonctionnalité spécifique. Dans notre contexte, cette fonctionnalité est celle d'un composant à services. Ce patron de conception utilise l'inversion de contrôle (aka. IoC[10]) et n'est donc pas intrusif.

Nous étendons la sémantique du *bundle* définissant le composant en ajoutant une entrée dans l'en-tête de son fichier *manifest* : JuliacLauncher: classname=, la classe indiquée étant l'usine à composants générée par *Juliac*. Un composant du canevas surveille l'installation d'un tel *bundle*, charge l'usine à composants qui y est définie et invoque celle-ci pour la création d'un nouveau composant.

5.2.4 Implémentation de *Fractal* à base de proxy *Java*

Un système *SoftwareUnit* peut être ajouté dynamiquement. Or un tel système peut requérir des définitions de membranes qui lui sont propres. Dans notre implémentation de *Fractal*, nous autorisons ainsi l'ajout dynamique de définitions de membranes à une usine à composants. Une telle usine à composants est dite dynamique, car elle propose en outre, la création et l'ajout de nouvelles interfaces à des composants *Fractal* existants.

Une usine à composants dynamique est représentée par l'interface DynamicGenericFactory (définie dans le listing 5.2).

```
package org.ow2.jonas.su.impl.proxy.factory;
public interface DynamicGenericFactory extends GenericFactory {
    void addMembraneDescriptor(URL membraneDesc)
            throws MembraneDescriptorManagerException;
    Interface addInterface(MutableComponent component,
        InterfaceType interfaceType, String itfName,
        boolean isInternal, Object impl)
            throws IllegalInterfaceException,
                InstantiationException;
}
```

Listing 5.2 – API d'une usine à composants SoftwareUnit

L'opération addMembraneDescriptor ajoute un descripteur de membranes à la fabrique. Le listing 5.3 présente un exemple de définition de membranes pour le système *OSGi*. L'opération addInterface crée une nouvelle interface au composant représenté par le type MutableComponent (définie dans le listing 5.4).

9. OSGi Extender Model : `http://www.osgi.org/blog/2007/02/osgi-extender-model.html`
10. IoC : `http://martinfowler.com/bliki/InversionOfControl.html`

```
<su-framework>
  <interface>
    <id>bundle-binding-controller-itf</id>
    <name>binding-controller</name>
    <signature>[...].BundleBindingController</signature>
  </interface>
  <membrane>
    <name>compositeBundle</name>
    <controller>
      <interfaceId>su-itf</interfaceId>
      <implementation>[...].SoftwareUnitImpl</implementation>
    </controller>
    <controller>
      <interfaceId>bundle-binding-controller-itf</interfaceId>
      <implementation>[...].BasicBundleBindingController</implementation>
    </controller>
    <controller>
      <interfaceId>su-content-controller-itf</interfaceId>
      <implementation>
        [...].BasicSoftwareUnitContentController
      </implementation>
    </controller>
    <controller>
      <interfaceId>super-controller-itf</interfaceId>
      <implementation>[...].BasicSuperController</implementation>
    </controller>
    <controller>
      <interfaceId>su-lifecycle-controller-itf</interfaceId>
      <implementation>[...].BundleLifeCycleController</implementation>
    </controller>
    <controller>
      <interfaceId>name-controller-itf</interfaceId>
      <implementation>[...].BasicNameController</implementation>
    </controller>
    <controller>
      <interfaceId>su-system-controller-itf</interfaceId>
      <implementation>
        [...].BasicSoftwareUnitSystemController
      </implementation>
    </controller>
  </membrane>
</su-framework>
```

Listing 5.3 – Exemple de définition de membrane pour le système OSGi

Les définitions de membranes, appartenant à des *bundles* contenant une implémentation de système, sont automatiquement chargées au déploiement de ceux-ci.

L'interface MutableComponent étend l'interface Component en y ajoutant les deux opérations addFcInterface et removeFcInterface, permettant d'ajouter ou de supprimer une interface du composant (modifiant ainsi le type du composant).

```
package org.ow2.jonas.su.api;
interface MutableComponent extends Component {
    void addFcInterface(String itfName, Interface itf)
            throws IllegalInterfaceException;
    void removeFcInterface(String itfName)
        throws NoSuchInterfaceException;
}
```

Listing 5.4 – API d'un composant dont le type peut évoluer

Une motivation pour une telle interface est la possibilité de représenter un logiciel dont les capacités ou besoins sont modifiés au cours de son cycle de vie.

5.3 Configuration du canevas

Le gestionnaire de configurations de *SUF* doit pouvoir supporter l'ajout et la suppression dynamique de *bundles* (définissant par exemple des systèmes) permis par la plate-forme *OSGi*. Le standard *OSGi ConfigurationAdmin* permet d'associer une configuration à un service donné. Nous utilisons ainsi *ConfigurationAdmin* pour la gestion de la configuration des différents services composant *SUF*.

Le gestionnaire de granularité est un service configuré selon cette approche. Celui-ci permet de sélectionner les niveaux de granularité afin d'optimiser la construction des représentations de logiciels, chaque système *SoftwareUnit* définissant des niveaux de granularité. Ceux-ci sont décrits dans les sections relatives à chacun des systèmes.

SUF dispose également d'un gestionnaire d'alias permettant d'associer des alias à des fonctionnalités. Ceux-ci permettent de ne pas se lier à une implémentation particulière. Par exemple le paquet système correspondant à l'environnement d'exécution pour *OpenJDK* version 1.6 est fourni par le paquet *openjdk-6-jre* sur *Debian* ou le paquet *java-1.6.0-openjdk* sur *Fedora*. L'alias *java-1.6.0* peut être utilisé pour ces deux paquets.

5.4 Implémentation de systèmes *SoftwareUnit*

Cette section décrit des implémentations de système *SoftwareUnit*. Les systèmes considérés peuvent être associés à des formats de module logiciel : la section 5.4.1 décrit le système *File*, la section 5.4.2 le système *OSGi*, la section 5.4.3 le système *RPM* et la section 5.4.4 le système *Deb*. Les autres systèmes proposés, le système *Fractal* dans la section 5.4.5 et le système *SUD* dans la section 5.4.6, ne correspondent pas à des formats de module.

5.4.1 Système *File*

Le système *File* ajoute le support pour la représentation de fichiers locaux ou distants. Ce système supporte également les fichiers archivés ou compressés. La figure 5.4 donne un exemple de représentation d'un répertoire avec le système *File*.

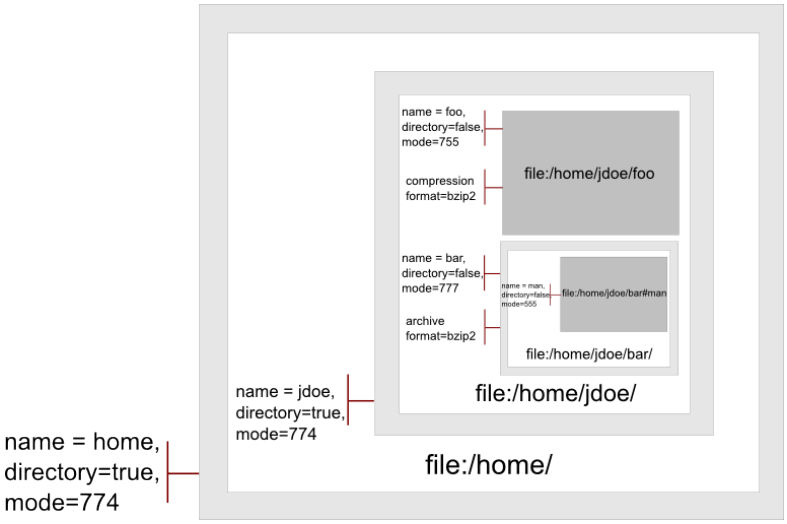


FIGURE 5.4 – Exemple de représentation d'un répertoire avec le système *File*

La représentation du contenu des répertoires et des archives peut être désactivée dans le gestionnaire de granularité.

Le système *File* ne définit pas de besoins et ne donne par conséquent aucune sémantique à la liaison.

5.4.1.1 Nommage

Le contexte de nommage du système *File* adresse les fichiers et dossiers, ceux-ci pouvant être locaux ou distants. Ce contexte de nommage permet en outre d'adresser le contenu d'archives.

L'encodage en chaînes de caractères d'un nom de fichier dans ce système correspond à son chemin absolu, préfixé par l'adresse de l'hôte le cas échéant. L'encodage suit le format suivant :

file :/[hôte]|[chemin](#[entrée])

Par exemple le nom file :/192.168.56.8|/ home/jdoe désigne le répertoire dont le chemin est /home/jdoe sur la machine ayant pour adresse IP 192.168.56.8.

Un nom permet également d'identifier le contenu d'une archive à partir de son entrée, dans ce cas le nom est suffixé par le caractère # suivi de l'entrée.

5.4.1.2 Méta-données

Chaque répertoire ou fichier possède un descripteur de capacité décrivant le nom de celui-ci, s'il est un répertoire ou non et ses permissions d'accès (ses modes). Ces informations sont décrites dans les méta-données associées au descripteur. Par exemple la méta-donnée dont la forme textuelle est home(directory=**true**,mode=774) caractérise

un répertoire dont le nom est *home* et dont les permissions d'accès correspondent au mode *774*.

Lorsqu'un composant modélise le contenu d'une archive ou d'un fichier compressé, celui-ci expose également un descripteur de capacité précisant le format utilisé (eg. *zip*, *tgz*...).

5.4.1.3 Membrane spécifique

Le système *File* redéfinit certains contrôleurs afin d'interagir avec le fichier représenté. Ainsi le contrôleur de contenu d'un répertoire associe l'ajout d'un sous-composant avec la copie du fichier représenté par celui-ci. Le contrôleur de contenu d'une archive associe l'ajout d'un sous-composant avec l'ajout dans l'archive du fichier représenté par celui-ci.

Le contrôleur de cycle de vie d'un fichier permet de l'exécuter si ses permissions l'autorisent.

5.4.1.4 Opérations

Le système *File* propose les opérations courantes sur fichiers : copie (avec changement des permissions), archivage (et extraction), compression (et décompression).

5.4.2 Système *OSGi*

Le système *OSGi* ajoute le support pour la représentation de *bundles*, de *packages* et de services *OSGi*. La figure 4.4 donne un exemple de représentation d'un *bundle* avec le système *OSGi*.

Le système peut construire une représentation d'un *bundle* pour un fichier ou un *bundle* déjà installé.

La représentation des *packages Java* et des services peut être désactivée dans le gestionnaire de granularité.

5.4.2.1 Nommage

Le contexte de nommage du système *OSGi* adresse les fichiers de type *bundle* ainsi que les *bundles* installés sur une plate-forme *OSGi*. Celui-ci permet également d'adresser les *packaqes* et les services des *bundles*.

5.4.2.2 Méta-données

Les méta-données *OSGi* sont définies dans le fichier *Manifest* d'un *bundle*, ou bien directement par la méthode Bundle.getHeaders() lorsque le *bundle* est installé. Si la spécification *OSGi* [63] définit un jeu standard de méta-données, des modèles à composant à services (eg. *Declarative Services* ou *iPOJO*), construits au dessus d'*OSGi*, peuvent en définir de nouvelles. L'analyseur de méta-données que nous proposons pour le système *OSGi* est extensible afin de supporter les méta-données définies par ces outils.

Capacité de type *bundle OSGi* Cette capacité est décrite dans l'entête d'un *bundle* par les entrées standards *Bundle-SymbolicName*, *Bundle-Version* et *Fragment-Host*. Cette dernière entrée identifie les *bundles* complétant un *bundle* hôte.

Capacité de type *package Java* Cette capacité est décrite dans l'entête d'un *bundle* par l'entrée standard *Export-Package*.

Capacité de type service Cette capacité est décrite dans l'entête d'un *bundle* soit par l'entrée standard *Export-Service*, soit par l'entrée *iPOJO-Components*.

Besoin de type *bundle OSGi* Ce besoin est décrit par l'entrée standard *Require-Bundle*.

Besoin de type *package Java* Ce besoin est décrit par les entrées standards *Import-Package* et *DynamicImport-Package*.

La contingence du descripteur est optionnelle lorsque l'entrée est *Import-Package* avec l'attribut *resoluton=optional* ou alors lorsque l'entrée est *DynamicImport-Package*. L'utilisation du joker (*) avec cette dernière entrée exprime une dépendance dont la cardinalité est collection.

Besoin de type service Ce besoin est décrit par l'entrée standard *Import-Service*.

5.4.2.3 Membrane spécifique

Le système *OSGi* redéfinit certains contrôleurs afin d'interagir avec la plate-forme *OSGi*. Ainsi le contrôleur de liaisons utilise le gestionnaire de dépendances du système afin de forcer le résolveur de la plate-forme à choisir un *package* ou un *bundle*.

Le contrôleur de cycle de vie d'un *bundle* permet de le démarrer ou de le stopper. Afin de préserver la cohérence du modèle, ces contrôleurs sont associés à des *listeners* exposés par la plate-forme. Ainsi, si un *package* est résolu en passant directement via l'API exposé par le plate-forme, le contrôleur de liaisons propage les nouvelles dépendances au sein du modèle.

5.4.2.4 Opérations

Le système fournit des opérations pour installer (*osgi-bundle-installer*), résoudre (*osgi-bundle-resolver*), activer un *bundle* (*osgi-bundle-launcher*), ainsi que les opérations inverses.

osgi-bundle-installer Cette opération crée un *bundle* à partir d'un fichier et installe celui-ci dans le dépôt.

osgi-bundle-resolver Cette opération lance la résolution au niveau de la plate-forme *OSGi*. Ce résolveur ne modifie par les liaisons au niveau du modèle, mais répercute ces dernières au niveau de la plate-forme. Une résolution simultanée des dépendances au niveau du modèle et de la plate-forme est permise par l'utilisation conjointe du résolveur générique basé sur le dépôt (défini dans la section 4.3.4.4) avec ce résolveur.

Au niveau du modèle, le chargement d'un *bundle fragment* se traduit par une copie des descripteurs du composant le représentant vers les descripteurs du composant de son hôte : l'hôte acquiert les capacités et besoins du *fragment*.

osgi-bundle-launcher Cette opération démarre un *bundle* en passant par le contrôleur de cycle de vie de son composant et retourne un composant *Fractal* dont les interfaces de type serveur sont implémentées par les services enregistrés par le *bundle*. Précédemment nous décrivions comment construire des services à partir de composant *Fractal*, ici nous faisons l'approche inverse en créant des composant *Fractal* à partir de services.

5.4.2.5 Interactions avec le résolveur

Le résolveur de la plate-forme *OSGi* utilise un mécanisme autonome afin de créer des liens entre les capacités et besoins des *bundles*. Les capacités sont de type export de *bundle* ou de *package*, les besoins sont de type import de *bundle* ou de *package*. Le résolveur essaie ainsi de satisfaire les besoins qui lui sont soumis, en fonction des capacités disponibles. L'algorithme de résolution étant spécifique à une implémentation de plate-forme *OSGi*, le résultat de la résolution est donc indéterministe du point de vue de la spécification (jusqu'à la version 4.2 de celle-ci). À partir de la version 4.3 [61], la spécification introduit, via la *RFC 138* (nommée *Framework Hooks*), un mécanisme permettant de restreindre les capacités visibles pour la résolution de certains besoins.

L'implémentation du gestionnaire de dépendances pour *OSGi* (cf. la section 4.3.3.3) a été réalisée avant la définition de la *RFC 138*. Cette version est définie sous forme de *bundle extension* afin de minimiser les modifications apportées à la plate-forme, mais reste spécifique à une version antérieure de l'implémentation *Felix*. Une nouvelle version, plus pérenne, pourrait être réalisée à partir de la nouvelle API introduite par la *RFC 138*.

5.4.2.6 Application à la cohérence des types

Dans la section 3.4.5, nous décrivons un problème de cohérence des types, dû à l'absence de coordinations entre la plate-forme *OSGi* et l'usine à ADL. Lorsqu'une liaison entre interfaces de composant *Fractal* est associée à un partage de types, celle-ci doit être prise en compte au niveau du résolveur et cela avant même le chargement des *bundles OSGi* (donc avant la création des composants *Fractal*).

Nous proposons une solution basée sur les composants *SoftwareUnit*, lorsque l'implémentation des composants *Fractal* est définie dans des *bundles OSGi*. Chaque *bundle* est représenté par un composant *SoftwareUnit* décrivant les imports et exports de celui-ci. Notre solution utilise une description architecturale (ie. les déclarations de liaison), afin

d'identifier les échanges de type entre composants *Fractal* et de créer les liaisons correspondantes entre composants *SoftwareUnit*.

La figure 5.5 présente un exemple de notre solution basée sur des composants *SoftwareUnit*.

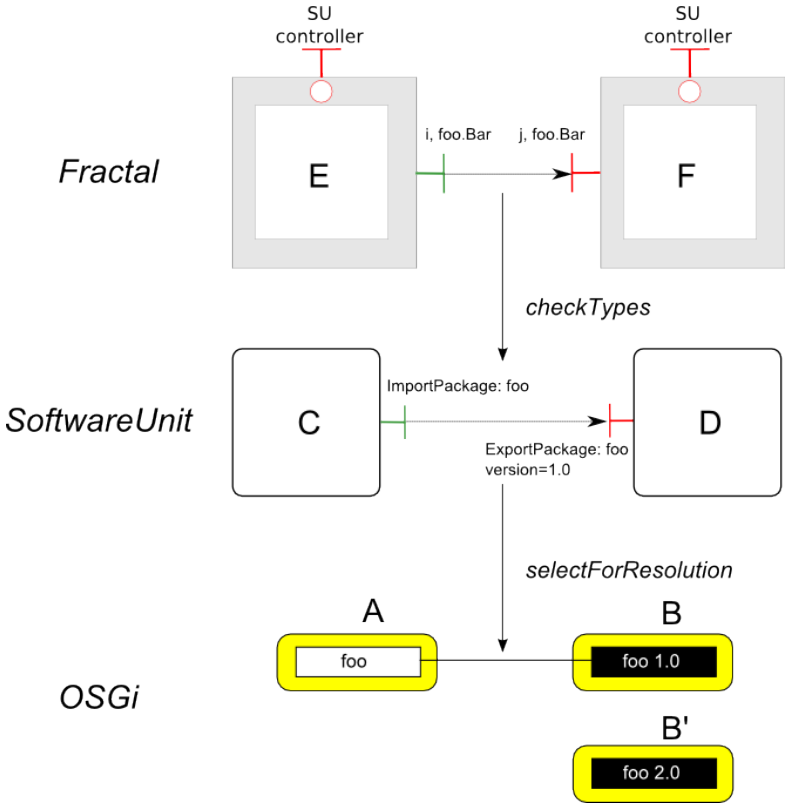


FIGURE 5.5 – Cohérence des liaisons entre composants définis avec des *bundles OSGi*

La description architecturale définit une liaison entre l'interface cliente i de type $foo.Bar$ du composant *Fractal* E avec l'interface serveur j du composant *Fractal* F. Le *bundle* A contient le code binaire permettant l'exécution du composant *Fractal* E. Ce dernier est associé au composant *SoftwareUnit* C, représentant le *bundle* A. Le *bundle* B contient le code binaire permettant l'exécution du composant *Fractal* F. Ce dernier est associé au composant *SoftwareUnit* D, représentant le *bundle* B.

On souhaite s'assurer que l'import du *package foo*, par le *bundle* A, soit bien résolu avec le *package* exporté par le *bundle* B, et non avec le bundle B'. Celui-ci propose en effet une version du *package* plus récente, le rendant ainsi prioritaire à la résolution.

La description architecturale est analysée avant la création des liaisons entre composants *Fractal*. Ainsi pour chaque liaison déclarée entre composants *Fractal*, l'existence d'un partage de types est vérifiée. Cette vérification peut être réalisée superficiellement ou

de manière approfondie. Une vérification superficielle s'effectue en vérifiant uniquement les imports et exports des *bundles*. Une vérification approfondie nécessite l'introspection du type des interfaces langage de chaque interface serveur de composant, mais sans chargement de *bytecode* (par exemple, à l'aide de l'outil *OW2 ASM*[11]).

Un échange de type détecté est représenté par une liaison, au niveau du modèle utilisant les composants *SoftwareUnit*.

Dans l'exemple précédent, puisque les interfaces i et j partagent le type $foo.Bar$, une liaison est créée entre les composants *SoftwareUnit* C et D, afin que la version du *package* foo importée soit celle du composant D.

Chaque liaison entre descripteurs est propagée au résolveur *OSGi*, à l'aide du gestionnaire de dépendances du système *SoftwareUnit OSGi*. Le démarrage des composants *SoftwareUnit* provoque une exécution du résolveur *OSGi*, lequel a eu connaissance des liaisons effectuées entre les composants *SoftwareUnit*. Finalement, les liaisons entre composants *Fractal* peuvent être effectuées sans risque d'incohérence de types.

Dans l'exemple précédent, le contrôleur de liaisons du composant *SoftwareUnit* C notifie le gestionnaire de dépendances. Lorsque les composants C et D sont démarrés, les *bundles* A et B sont activés et le résolveur *OSGi* crée le lien vers l'export de foo en version 1.0. Cette version est bien celle que contenait le *bundle* associé au composant F.

Le partage des types est une préoccupation de bas niveau qui est inhérente au système de modules utilisé. Certains modèles à composants intègrent ces préoccupations (tels que *iPOJO* [30] ou *Blueprint* [62]) mais en imposant un système de module spécifique (en l'occurrence *OSGi* pour les exemples cités). Nous proposons dans cette thèse une approche alternative, dans laquelle le modèle à composants décrivant le système à l'exécution (*Fractal* dans notre contexte) reste agnostique du système de module sous-jacent.

5.4.3 Système *RPM*

Le système *RPM* ajoute le support pour la représentation de paquets *RPM*. Le système *RPM* permet la création de composants *SoftwareUnit* à partir de fichiers *RPM*, mais aussi de paquets distants provenant de n'importe quel gestionnaire de paquets *RPM*. Le support d'un gestionnaire de paquets peut être apporté au système *RPM* comme illustré sur la figure 5.6.

5.4.3.1 Nommage

Le contexte de nommage du système *RPM* adresse les fichiers *.rpm*, ainsi que les paquets provenant de gestionnaires. La forme textuelle des noms est formée comme suit :

11. OW2 ASM : `http://asm.ow2.org/`

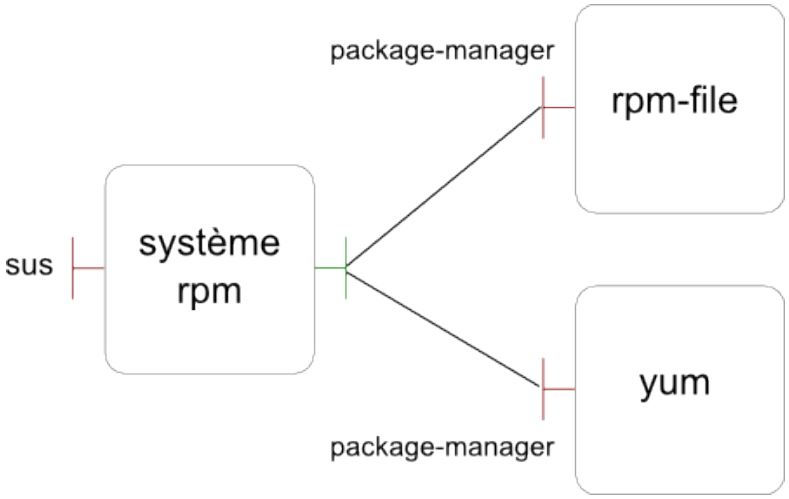


FIGURE 5.6 – Extension du système *RPM* avec des gestionnaires de paquets

rpm :/[hote]|[gestionnaire de paquets]/[nom du paquet]

Par exemple, le nom dont la forme textuelle est rpm :/192.168.56.7|yum/gnugo représente le paquet dont le nom est *gnugo* sur la machine dont l' adresse IP est 192.168.56.7, et qui est définit par le gestionnaire de paquets *yum*.

5.4.3.2 Méta-données

Les descripteurs sont construits à partir des tags *RPM*.

Capacités Les méta-données de capacité d'un paquet sont construites à partir des tags *name* et *version*. Ces méta-données sont associées au descripteur symbolique.

Lorsqu'un paquet *RPM* fournit un paquet virtuel (défini par le tag *provides*), un second descripteur de capacité est construit.

Besoins Deux type de descripteurs de besoins existent : ceux décrivant des paquets requis (tag *requires*) et ceux décrivant des paquets en conflit (tag *conflicts*).

5.4.3.3 Opérations

Le système *RPM* définit des opérations pour installer des fichiers *.rpm* avec le programme *rpm* ainsi que pour installer des paquets avec des gestionnaires de paquets (eg. *yum*).

5.4.4 Système *Deb*

Le système *Deb* ajoute le support pour la représentation de paquets *Debian*. Comme le système *RPM*, celui-ci est extensible afin d'être compatible avec n'importe quel ges-

tionnaire de paquets *Debian* (eg. *apt-get*).

5.4.4.1 Nommage

Le contexte de nommage du système *Deb* adresse les fichiers *.deb*, ainsi que les paquets provenant de gestionnaires. Similairement au système *RPM*, la forme textuelle des noms est formée comme suit :

deb :/[hote]|[gestionnaire de paquets]/[nom du paquet]

Par exemple, le nom dont la forme textuelle est deb :/192.168.56.8|apt/gnugo représente le paquet dont le nom est *gnugo* sur la machine dont l' adresse IP est 192.168.56.8, et qui est définit par le gestionnaire de paquets *apt-get*.

5.4.4.2 Méta-données

Les descripteurs sont construits à partir des champs définis dans le fichier de contrôle du paquet.

Capacités Les méta-données de capacité d'un paquet sont construites à partir des champs *package* et *version*. Ces méta-données sont associées au descripteur symbolique.

D'autres descripteurs de capacité peuvent également être créés pour chaque nom de paquet virtuel associé au champs *provides*.

Besoins Les paquets requis sont exprimés avec le champs *depends*. Les paquets optionnels peuvent être exprimés avec les champs *recommends* ou *suggests*. Enfin, les paquets en conflit sont exprimés avec le champ *conflicts*. Le système de paquets *Debian* autorise des alternatives dans les besoins, ce qui est traduit par un seul descripteur dont les méta-données contiennent l'opération booléen *ou*.

5.4.4.3 Opérations

Le système *Deb* définit des opérations pour installer des fichiers *.deb* avec le programme *Deb* ainsi que pour installer des paquets avec des gestionnaires de paquets (eg. *apt-get*).

5.4.5 Système *Fractal*

Le système *Fractal* est différent des précédents dans le sens où il n'est pas attaché à un format de module patrimonial. Ce système définit une capacité relative à la création de composant *Fractal*. Dans la section 5.2.3.2 nous définissions un format de méta-données permettant de déclarer qu'un *bundle* contenait une usine à composant. En l'occurrence, il s'agissait de l'entrée JuliacLauncher du fichier *Manifest*. Nous proposons dans ce système de construire un descripteur représentant cette capacité et plus généralement de n'importe quelle information permettant de construire un composant *Fractal*.

Ces informations peuvent être de trois natures :

– le nom d'une usine à composants (non générique naturellement) ;
– le type du composant, la description de son contenu et de sa membrane ;
– un fichier ADL.

L'opération fractal −launcher, définie dans ce système, retourne des composants *Fractal* à partir de composants *SoftwareUnit* (pouvant représenter des *bundles*) ayant une capacité décrivant une de ces trois informations.

5.4.6 Système *SUD*

Le système *SUD* apporte la représentation des plans de déploiement *SUD*, décrit dans la section 4.4.2. Il encapsule sous forme de composants les différents éléments du plan de déploiement. Le système *SUD* propose en outre des opérations permettant de modifier un plan de déploiement à partir de sa représentation.

5.5 Déploiement distribué

Le déploiement peut être contrôlé à distance selon deux possibilités : soit en créant la représentation directement sur la machine administrée, soit en la créant sur une tierce machine. La première possibilité utilise *Fractal RMI* pour la communication avec la machine administrée. Cependant, cette approche nécessite le déploiement de *SUF* sur la machine administrée, ce qui peut ne pas être souhaitable en production. Nous proposons alors de ne déployer *SUF* que sur une tierce machine laquelle coordonne le déploiement sur l'ensemble des machines administrées.

Pour implémenter cette solution, nous avons choisi d'encoder l'adresse des machines dans le nom associé aux logiciels représentés. Un composant *SoftwareUnit* est ainsi associé à un logiciel sur le réseau. Les opérations nécessitant une interaction avec des environnements d'exécution (pouvant être locaux ou distants) utilise une couche d'abstraction [12] afin de ne pas être lié à un protocole de communication (eg. *ssh*). Celle-ci permet ainsi à des opérations telles que la copie de fichier ou l'installation de paquets *RPM* de rester génériques.

5.6 Conclusion

Nous avons présenté dans ce chapitre une mise en œuvre du modèle à composants *Fractal SoftwareUnit* au sein du canevas *SUF*. Celui-ci est implémenté à l'aide de composants *Fractal* à services. À cette fin, nous avons proposé d'exploiter les opportunités offertes par *OSGi* pour ajouter du dynamisme à la plate-forme *Fractal* elle-même. Une implémentation de *Fractal* et de membrane peut ainsi être chargée dynamiquement. Divers systèmes ont été proposés ainsi que des solutions pour la distribution du déploiement. *SUF* offre des gestionnaires de configuration permettant de sélectionner le niveau de granularité des représentations, mais également de définir des alias évitant de se lier

12. `svn://svn.forge.objectweb.org/svnroot/easybeans/trunk/ow2-bundles/protocol`

à une implémentation spécifique lors de la définition des besoins d'un logiciel.

Les deux prochains chapitres proposeront des expérimentations construites à partir de *SUF*.

Troisième partie

Expérimentations et résultats

Chapitre 6

Gestion des profils *JOnAS*

Sommaire

Une première expérimentation du modèle à composants *Fractal SoftwareUnit* et de son implémentation est consacrée à la définition d'un outil pour la gestion de profils pour le serveur d'applications *JOnAS*, présenté dans la section 1.2.6. Dans celle-ci était souligné le besoin de création de nouveaux profils de la distribution de *JOnAS*, que ce soit à destination de l'éditeur ou de l'utilisateur. Ce chapitre décrit la démarche suivie pour la réalisation d'un tel outil, et en fournira une évaluation.

Après avoir exposé la problématique dans la section 6.1, la modélisation d'un assemblage du serveur d'applications permettra dans un premier temps (section 6.2) l'identification des différents éléments le constituant. Puis, nous définirons un profil *JOnAS* dans la section 6.3. Enfin, nous mettrons en œuvre la création d'un nouveau profil dans la section 6.4.

6.1 Problématique

Le découpage des fonctionnalités de *JOnAS* en *bundles OSGi* a permis à celui-ci d'être un serveur d'applications modulaire. Néanmoins, la création d'un ensemble fonctionnel de modules, représentant un profil *JOnAS*, nécessite de jongler avec les dépendances de chaque module, celles-ci pouvant être explicites ou non. Par exemple, les dépendances sur l'environnement fournies par le système d'exploitation sont invisibles. De plus *JOnAS* définit des fonctionnalités, telles que les services techniques, qui ne sont pas réifiées au niveau des *bundles*. En outre, tout module de *JOnAS* n'est pas un *bundle*. La plate-forme d'exécution constituant le noyau de *JOnAS* est notamment composée

de fichiers de configuration, d'exécutables ou d'exemples destinés à illustrer un ou plusieurs services techniques... Ces fichiers n'ont ainsi aucune définition d'appartenance ou de dépendance. Enfin, l'absence de modularité de la plate-forme *Java* est intrusive en requérant que certaines librairies, dites *endorsed* (telles que *CORBA* ou *SAX*), soient chargées via un mécanisme exclusif [1].

6.2 Modélisation d'un assemblage de *JOnAS*

Dans cette section, nous présentons une modélisation d'un assemblage de *JOnAS* sous forme de composant *SoftwareUnit* nous permettant, dans la section suivante, la construction de profils du serveur d'applications.

6.2.1 Rétro-ingénierie de *JOnAS*

Le serveur d'applications *JOnAS* est constitué d'un ensemble de fichiers classés selon leur rôle. Ainsi, chaque rôle est associé à un répertoire :

les exécutables permettant de démarrer le serveur ou de lancer des outils sont stockés dans le répertoire *bin* ;

la configuration du noyau et des services du serveur est stockée dans le répertoire *conf* ;

les librairies contenant les binaires *Java* qui ne sont pas des *bundles OSGi* sont stockées dans le répertoire *lib* ;

les applications à déployer, pouvant être de type *OSGi*, *Java EE* ou plan de déploiement, sont stockées dans le répertoire *deploy* ;

les exemples afin d'illustrer certaines fonctionnalités de *JOnAS* appartiennent au répertoire *examples* ;

les dépôts contenant les fichiers référencés dans des plans de déploiement sont stockés dans le répertoire *repositories*.

Cette structure est définie dans le noyau de *JOnAS* et reste identique quelque soit le nombre de services installés. À partir de l'identification des éléments constituant *JOnAS*, nous pouvons définir un méta-modèle (décris par la figure 6.1).

Chaque classe en bleu du diagramme représente un élément de l'assemblage, chacun de ces éléments étant un fichier ou un répertoire. Ces éléments devront apparaître dans la représentation de *JOnAS*.

6.2.2 Instanciation du méta-modèle avec des composants *SoftwareUnit*

Le méta-modèle décrit par la figure 6.1 peut être instancié avec des composants *SoftwareUnit*. Cette instanciation passe par la définition d'un nouveau système de composants, capturant un nouveau jeu de métadonnées et proposant un contexte de nommage spécifique.

1. `http://download.oracle.com/javase/6/docs/technotes/guides/standards/index.html`

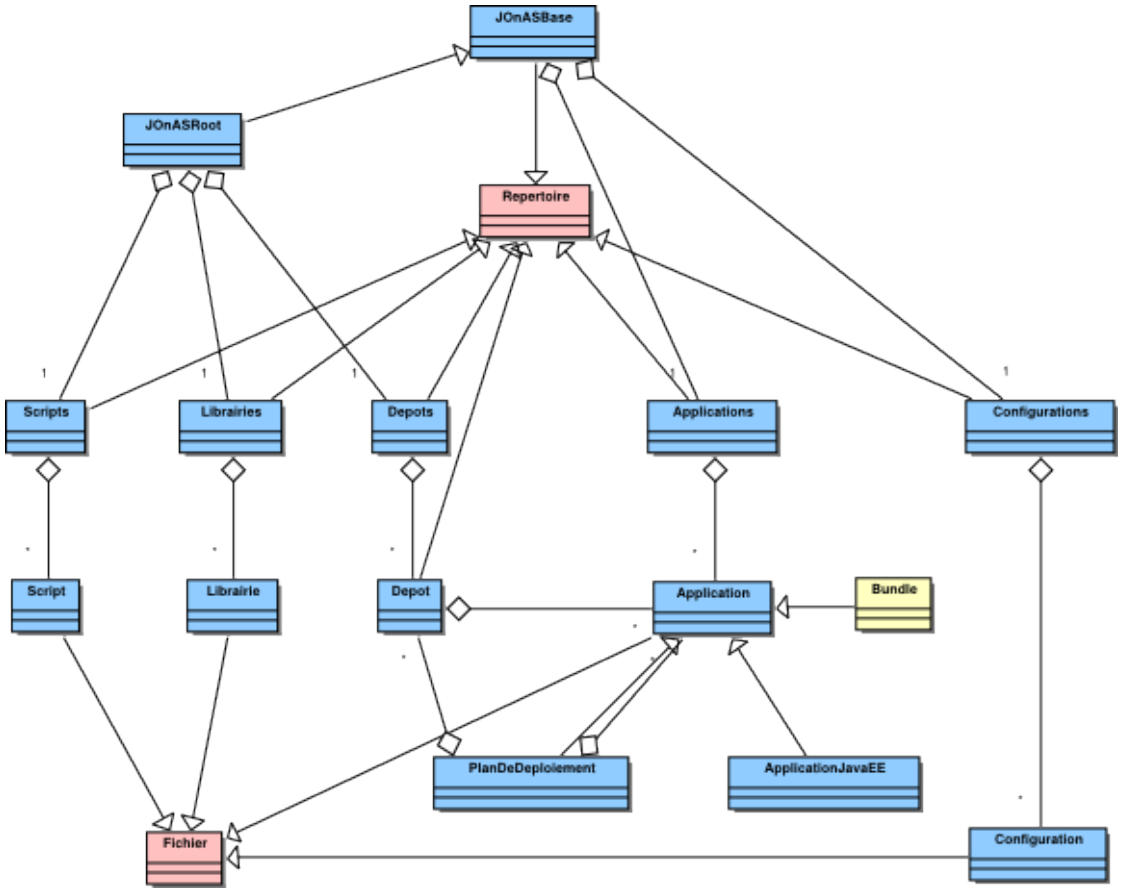


FIGURE 6.1 – Méta-modèle du serveur d'applications *JOnAS*

6.2.2.1 Définition du système *SoftwareUnit JOnAS*

Le système *SoftwareUnit JOnAS* permet la représentation du serveur d'applications ainsi que celle des applications pouvant y être déployées. C'est le cas par exemple des plans de déploiement *JOnAS* et de leur contenu.

Contexte de nommage Le contexte de nommage identifie les éléments constituant un assemblage de *JOnAS*.

Méta-données Les méta-données expriment les capacités et besoins des éléments constituant un assemblage de *JOnAS*. Ainsi, chaque classe en bleu du méta-modèle 6.1 définit une capacité réifiée dans notre modèle.

Opérations Le système *JOnAS* fournit l'opération maven−repository−assembler de création de dépôt *Maven* à partir de plans de déploiement. Celle-ci permet notamment de peupler le dépôt *Maven* des *bundles* référencés dans le plan de déploiement.

L'opération jonas−deployer permet l'ajout de composants dans un assemblage de *JOnAS*. Cette opération permet la construction de profils à partir d'une représentation de *JOnAS* et de librairies, de services ou d'applications à inclure.

6.2.2.2 Patron du serveur d'applications *JOnAS*

Le système *JOnAS* délivre un patron d'assemblage de *JOnAS* permettant de construire de nouveaux profils.

La figure 6.2 décrit un tel patron d'assemblage.

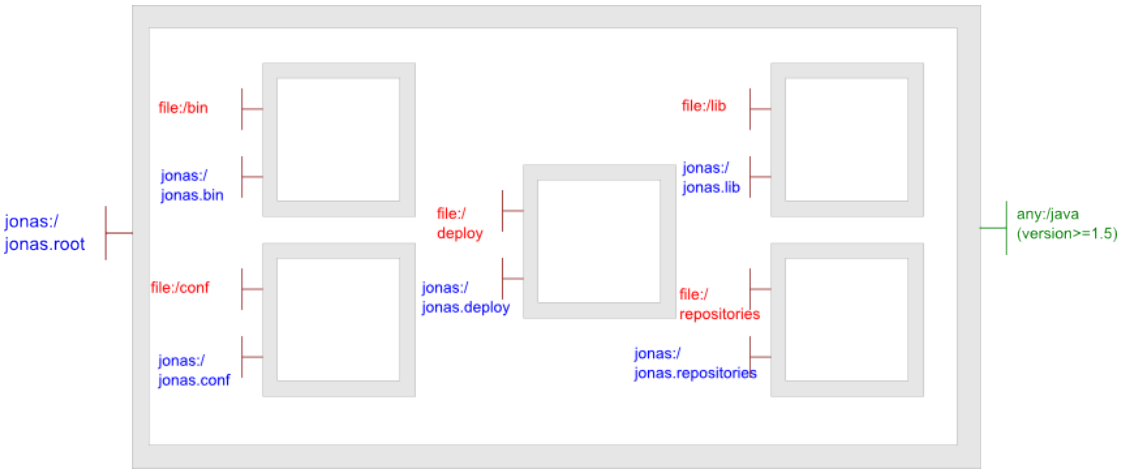


FIGURE 6.2 – Patron d'assemblage du serveur d'applications *JOnAS*

Le patron est un composite dont les sous-composants possèdent un descripteur de capacité pour le système *JOnAS* et un autre pour le système *File*. La dépendance sur

Java ne spécifie pas de système et utilise l'alias *java* : le système et le nom de paquet seront découverts à l'exécution.

Ce patron contient une représentation de tous les éléments obligatoires du serveur d'applications. Ainsi, la figure 6.2 omet les nombreux composants primitifs représentant ces éléments.

6.2.3 Cohérence du modèle et de l'assemblage physique

Les composants, contenus dans le patron de *JOnAS*, possèdent des descripteurs appartenant à différents systèmes *SoftwareUnit* (eg. *JOnAS* ou *OSGi*). Ces descripteurs sont utilisés lors du déploiement, afin, par exemple, d'identifier les différents éléments à ajouter dans l'assemblage (réalisé par l'opération *jonas-deployer*). De plus, tous les composants de ce patron possèdent un descripteur défini dans le système *File*. Celui-ci est interprété par l'opération *file-assembly-installer* construisant l'assemblage physique correspondant au modèle. Les informations lues concernent notamment le nom du fichier (ou du dossier) ainsi que ses droits d'accès. Cette opération conserve les descripteurs relatifs aux différents systèmes, de sorte que le composant résultant de l'opération *file-assembly-installer* peut être réutilisé comme un nouveau patron du serveur d'applications. Si le format de l'assemblage est un répertoire utilisé pour l'exécution de *JOnAS* (ie. les variables d'environnement *JONAS_ROOT* ou *JONAS_BASE* ont pour valeur ce répertoire), l'utilisation de l'opération *jonas-deployer* autorise alors le déploiement à chaud de *bundles*, de services ou d'applications *Java EE*.

6.3 Modélisation d'un profil *JOnAS*

On appelle profil *JOnAS*, une spécialisation du serveur d'applications. La section précédente expliquait notre méthode pour représenter un assemblage dans notre modèle. Dans cette section, nous décrirons une représentation de profil.

6.3.1 Définition d'un méta-modèle de profil *JOnAS*

La figure 6.3 propose un méta-modèle d'un profil *JOnAS*.

Un profil est composé d'un ensemble de scripts, configurations, librairies, plans de déploiement, services techniques et autres applications.

Une première étape avant la construction d'un profil est la définition d'une représentation de chacun de ces éléments sous forme de composants *SoftwareUnit*. La plupart de ceux-ci sont de simples fichiers tels des scripts ou des librairies. Ceux-ci n'ont pas de description de dépendance ou de contenu et leur représentation sous forme de composant est bâtie à l'aide du système *File*, à laquelle on ajoute un descripteur de capacité défini par le système *JOnAS*. Les services techniques possèdent, à la différence, une définition de dépendance et de contenu propre au système *JOnAS*. La prochaine section détaillera la représentation d'un service technique. Chaque élément pouvant intervenir dans un

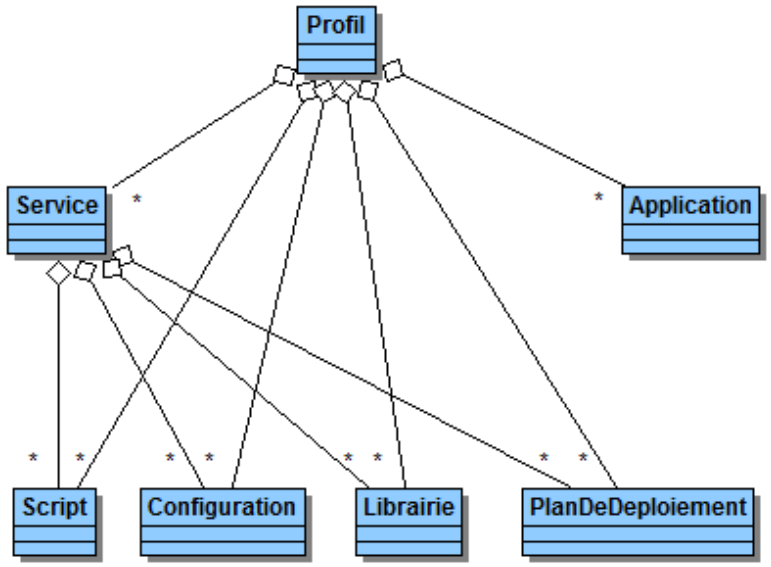


FIGURE 6.3 – Méta-modèle d'un profil

profil *JOnAS* est ensuite ajouté dans un dépôt, afin de constituer ainsi une bibliothèque des composants de *JOnAS*.

6.3.2 Spécification des services techniques

La représentation d'un service technique est importante à deux titres, lors de construction d'un assemblage :

Cohérence La connaissances des dépendances d'un service permet d'inclure **au moins** ses dépendances ;

Juste taille La connaissances des dépendances d'un service permet d'inclure **au plus** ses dépendances.

Un service technique est représenté par un composite, dont le contenu comporte des scripts, configurations, librairies et plans de déploiement. Par exemple, le listing 6.1 présente un extrait de la description du service technique *ejb3* pour *Java EE 5*.

```
<definition name="org.ow2.jonas.su.service.Ejb3JavaEE5">
    <interface name="jtm" role="client"
        signature="jonas:/service.jtm" contingency="mandatory" />
    <interface name="wm" role="client"
        signature="jonas:/service.wm" contingency="mandatory" />
    <component name="dp-ejb3">
        <su-content description="url://ejb3.xml@jojoservices"
                    system="file"
                    launcher="deploymentplan-loader" />
    </component>
    <component name="dp-easybeans-core">
        <su-content description="url://easybeans-core.xml@jojoservices"
                    system="file"
                    launcher="deploymentplan-loader" />
    </component>
    ...
    <component name="conf-easybeans">
        <interface name="lib" role="server" signature="jonas:/conf" />
        <su-content description="$(sud:/JOnAS/easybeans-config-unpack)" />
    </component>
    <su-content description="service/ejb3" system="jonas" />
</definition>
```

Listing 6.1 – Extrait de la description du service *ejb3* pour *Java EE 5*

Un service technique possède des dépendances sur d'autres services. Dans l'exemple précédent, le composant représentant le service *ejb3* possède deux interfaces clientes dont les méta-données sont *jonas :/service.jtm*, représentant une dépendance sur le service *jtm*, et *jonas :/service.wm*, en représentant une seconde sur le service *wm*. Le composite représentant le service *ejb3* contient à la fois des composants représentant des plans de déploiement *JOnAS*, et un composant représentant le fichier de configuration d'*EasyBeans* (comme représenté sur la figure 6.4).

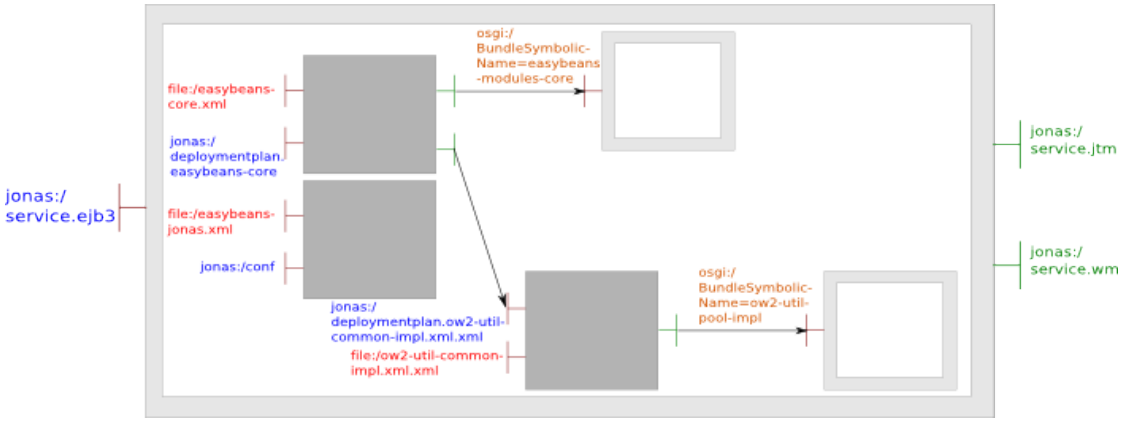


FIGURE 6.4 – Extrait du composite représentant le service *ejb3*

Les composants représentant les plans de déploiement sont chargés à l'aide de l'opération *deploymentplan-loader*. Dans notre exemple, ceux-ci ont des dépendances sur d'autres plans de déploiement et des *bundles*.

6.3.3 Construction des profils

L'opération jonas−deployer retourne un profil de *JOnAS* pour une spécification de contenu et un patron d'assemblage, tel celui illustré par la figure 6.2. Cette spécification peut être de deux ordres : des méta-données correspondant à des besoins (enregistrés dans un dépôt) ou des composants *SoftwareUnit*. Ainsi, un service peut être déployé soit à partir d'une méta-donnée exprimant une dépendance sur celui-ci, soit directement à partir d'un composite le représentant. Par exemple, le service *ejb3* peut être spécifié soit avec la méta-donnée *jonas :/service.ejb3*, soit avec le composant créé à partir de la description présentée sur le listing 6.1.

L'opération jonas−deployer analyse les descripteurs des composants à déployer et copie ceux-ci (ou leur contenu) dans un composite du patron d'assemblage. Par exemple, un composant ayant un descripteur de capacité dont la méta-donnée est *jonas :/conf* est ajouté dans le composite du patron correspondant à la configuration. La figure 6.5 illustre le composant d'assemblage résultant du déploiement du service *ejb3* (représenté sur la figure 6.4) dans le patron d'assemblage (illustré sur sur la figure 6.2).

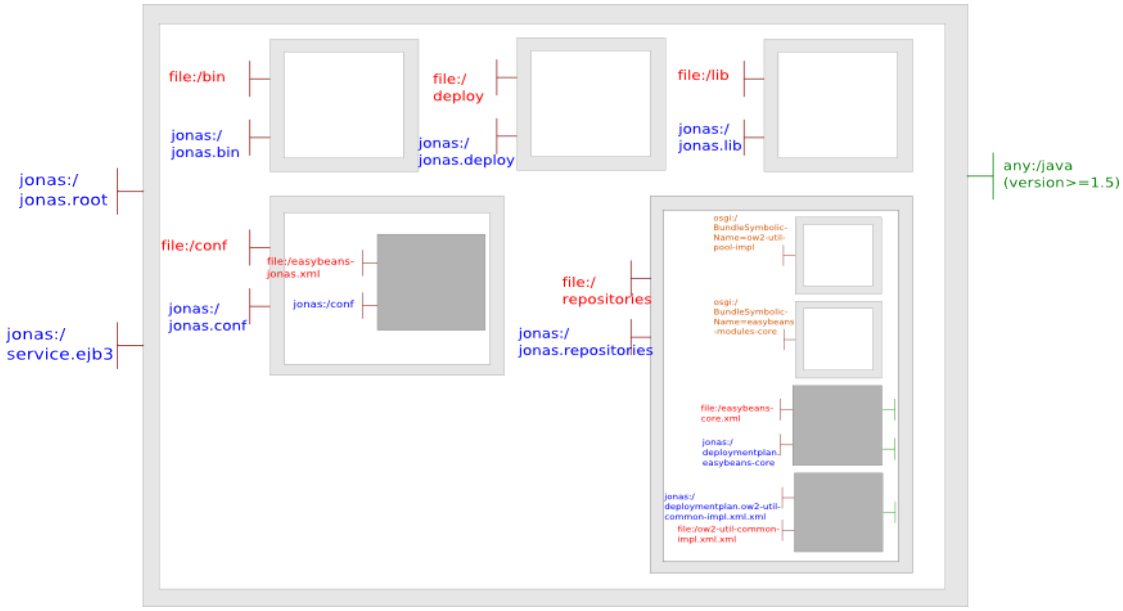


FIGURE 6.5 – Extrait de l'assemblage *JOnAS* après le déploiement du service *ejb3*

6.4 Étude de cas : création de profils *Web* pour *JOnAS*

Dans cette section, nous décrivons la définition de deux profils *Web* pour *JOnAS* : un de type *Java EE 6 Web Profile* et un autre de type *Web léger*.

6.4.1 *Java EE 6 Web Profile*

La spécification *Java EE 6*[2] identifie un sous ensemble de celle-ci ciblant uniquement les applications *Web*. Nous proposons l'utilisation de *SUF* afin de générer un assemblage de *JOnAS* comprenant les fonctionnalités décrites dans la spécification *Java EE 6 Web Profile*. Néanmoins l'assemblage proposé ici ne valide pas cette certification, *JOnAS 5.2* complet n'étant lui même pas certifié *Java EE 6*.

Le profil *Java EE 6 Web Profile* doit contenir les services techniques suivants :

web pour le conteneur *web* (*Servlet*, *JSP*) ;

jaxrpc pour le support des *Web Services* compatible *J2EE 4* ;

jaxws pour le support des *Web Services* compatibles *Java EE 5* ;

cdi pour le support de la *JSR-299* (*Contexts and Dependency Injection for the Java EE platform*) ;

ejb3 pour les conteneurs *EJB3* et *JPA* ;

validation pour le support de la *JSR 303* (*Bean Validation*) ;

jtm pour le support de *JTA*.

Les services *web* et *ejb3* sont disponibles en deux versions selon la version de *Java EE* (eg. *Java EE 6* requiert *Tomcat 7* et *OpenJPA 2*).

Le profil *Java EE 6 Web Profile* contient les services techniques suivants : *db*, *dbm*, *wc*, *depmonitor*, *jaxrpc*, *jtm*, *web*, *ear*, *resource*, *cdi*, *jaxws*, *validation*, *wm* et *ejb3*.

6.4.2 *Web léger*

Le profil *Web léger* est destiné aux applications *Java EE 5* ne nécessitant pas de conteneur *EJB*. Par rapport à l'utilisation d'un serveur *Tomcat* (ou *Jetty*) seul, ce profil offre des services techniques à forte valeur ajoutée telle que le service *versioning* offrant la gestion de versions multiples d'application *Web*. Ce profil sera utilisé dans le chapitre suivant car particulièrement adapté à l'application *Soapsoo* déployée.

Le profil *Web léger* contient les services techniques suivants : *db*, *dbm*, *wc*, *depmonitor*, *jaxrpc*, *jtm*, *web*, *ear*, *resource*, *wm* et *versioning*.

L'annexe A contient le script *SUD* définissant l'assemblage correspondant à ce profil.

6.4.3 Comparaison des assemblages

Le tableau 6.1 compare la taille de l'assemblage correspondant à ces profils et le temps de démarrage des serveurs *JOnAS*, avec les données des distributions existantes.

2. JSR 316 : `http://jcp.org/en/jsr/detail?id=316`

Profil	Micro	Web léger	Web Java EE 6	Complet
Taille du *ZIP* (en MO)	10	61	173	183
Temps de démarrage (en sec.)	3	6	8	10

TABLE 6.1 – Comparaison de la taille et du temps de démarrage de profils *JOnAS*

On notera que le fort écart de taille entre les profils *Web léger* et *Web Java EE 6* s'explique en partie par la présence du conteneur client (64 MO) ainsi que celles des librairies *endorsed* (*JacORB*, *Xalan*,...).

Concernant le temps de démarrage, celui-ci diminue logiquement lorsque l'on réduit les fonctionnalités du serveur d'applications.

6.5 Conclusion

Ce chapitre a expliqué comment la problématique de gestion des profils *JOnAS* peut être résolue. Le système *JOnAS* apporte la représentation d'un assemblage du serveur d'applications et des opérations permettant la construction de profils. Nous avons notamment proposé une abstraction des services techniques *JOnAS* comprenant, entre autre, la configuration et les librairies de ceux-ci.

Les résultats obtenus prouvent l'intérêt d'un serveur d'applications à la juste taille. La taille de l'assemblage et le temps de démarrage diminue pour les profils que nous avons proposés. Le temps de démarrage est important dans le contexte de l'allocation dynamique de serveurs dans le contexte du *cloud computing* (élasticité).

Dans ce chapitre, nous avons défini le profil d'assemblage *Web léger*. Le prochain chapitre explique notre démarche pour le déploiement de l'environnement d'exécution de celle-ci, lequel comprend ce profil. Enfin, le chapitre 8 valide l'utilisation de *SUF* par rapport à la solution actuelle qu'est *Apache Maven*.

Chapitre 7

Déploiement d'une application hétérogène

Sommaire

La seconde expérimentation de ce travail vise à illustrer le déploiement d'une pile de logiciels, indépendamment d'une technologie. Ainsi, nous proposons le déploiement d'une application *Java EE* sur un système d'exploitation nu (ie. sans machine virtuelle *Java*).

7.1 Problématique

L'assemblage d'applications défini dans le chapitre précédent constitue juste une étape du déploiement. Celle-ci restait locale et a permis notamment de compiler les dépendances sur l'environnement d'exécution. Les étapes suivantes comprennent l'installation de l'application et de ses dépendances sur des machines dont l'environnement d'exécution diffère.

7.2 Modélisation d'une application *Java EE*

Une application *Java EE* peut requérir des services techniques, des librairies, ainsi que des paquets systèmes. Ces dépendances sont représentées sur la figure 7.1.

En terme de composants *SoftwareUnit*, l'application et ses dépendances sont représentées à l'aide de composants liés.

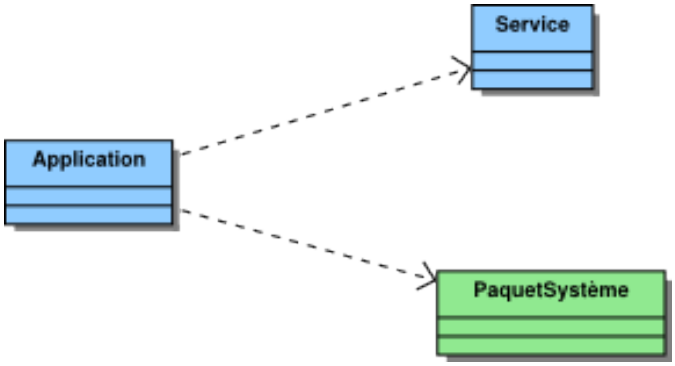


FIGURE 7.1 – Méta-modèle d'une application

7.3 Étude de cas : déploiement de l'application *Soapsoo*

7.3.1 Présentation de l'application *Soapsoo*

Soapsoo est une application *web* basée sur les technologies *Java EE* permettant la gestion de catalogues d'articles. Ces derniers sont stockés dans une base de données. Cette application a été développée par *France Telecom/Orange* pour la qualification de serveur d'applications *Java EE*.

Le serveur d'applications et la base de données sont déployées sur des machines virtuelles différentes, comme le décrit la figure 7.2.

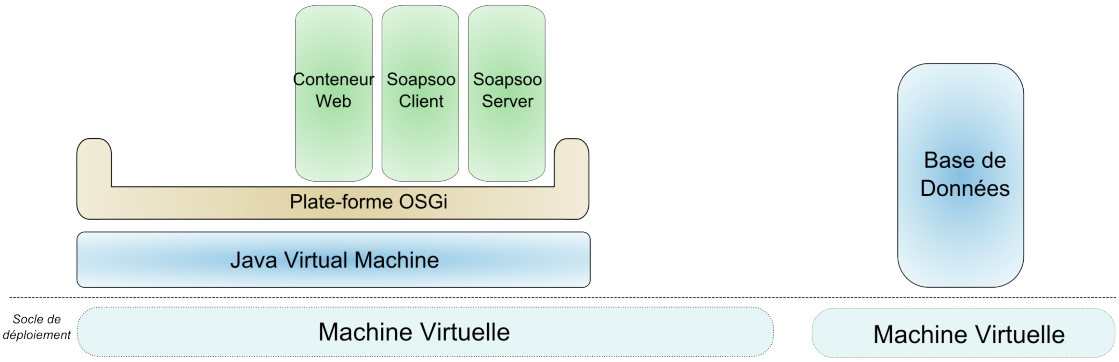


FIGURE 7.2 – Architecture de l'application *Soapsoo*

Soapsoo est empaqueté dans deux archives *ear* contenant les applications *web Soapsoo client* et *Soapsoo server*. *Soapsoo client* contient les parties contrôleur et vue. *Soapsoo server* définit le modèle de données. Les deux applications communiquent à l'aide de *web services*.

7.3.2 Description du processus de déploiement

Afin d'illustrer le support de l'hétérogénéité, nous avons choisi d'héberger la base de données et le serveur d'applications sur des systèmes d'exploitation ayant des systèmes différents de gestion de paquets. Les sous-sections suivantes décrivent les différentes étapes attendues du processus de déploiement. Celles-ci sont encapsulées par des opérations sur des composants *SoftwareUnit*.

7.3.2.1 Installation de la base de données

L'installation de la base de données s'effectue sur une machine exécutant la distribution *GNU/Linux Fedora*, à l'aide du paquet *RPM mysql-server.i686* et d'un script de configuration de la base.

7.3.2.2 Installation de la *JVM*

La machine virtuelle s'installe sur une seconde machine exécutant la distribution *GNU/Linux Ubuntu*, à l'aide du paquet *Debian openjdk-6-jre*. Cette machine hébergera une instance de *JOnAS*.

7.3.2.3 Installation de *JOnAS*

L'installation de *JOnAS* se décompose en trois étapes :
- Génération d'un profil d'assemblage correspondant au besoin de l'application *Soapsoo* ;
- Copie de cet assemblage sur la machine hébergeant *JOnAS* ;
- Configuration des variables d'environnement et démarrage du serveur.

L'application *Soapsoo* requiert les services techniques suivants :

web car *Soapsoo* est une application *web* ;

jaxrpc pour la communication entre *Soapsoo client* et *Soapsoo server* ;

jtm pour la gestion des transactions ;

ear car *Soapsoo* est empaqueté dans un *ear* ;

resource pour le déploiement du *Resource Adapter* pour *MySQL*.

Le profil *Web léger* défini dans le chapitre précédent satisfait parfaitement ces besoins.

7.3.2.4 Installation de *Soapsoo*

L'installation de *Soapsoo* se décompose en trois étapes :
- *Soapsoo* est copié sur la machine hébergeant *JOnAS* ;
- *JOnAS* est configuré pour *Soapsoo* (copie du driver *MySQL* et de la définition de *datasource*) ;
- Les tables de l'application sont créées dans la base de données.

7.3.3 Modélisation du déploiement de l'application

Le processus de déploiement est modélisé à l'aide d'un graphe de composants abstraits. Chacune des quatre étapes précédentes est représentée par un composant abstrait, comme décrit par la figure 7.3.

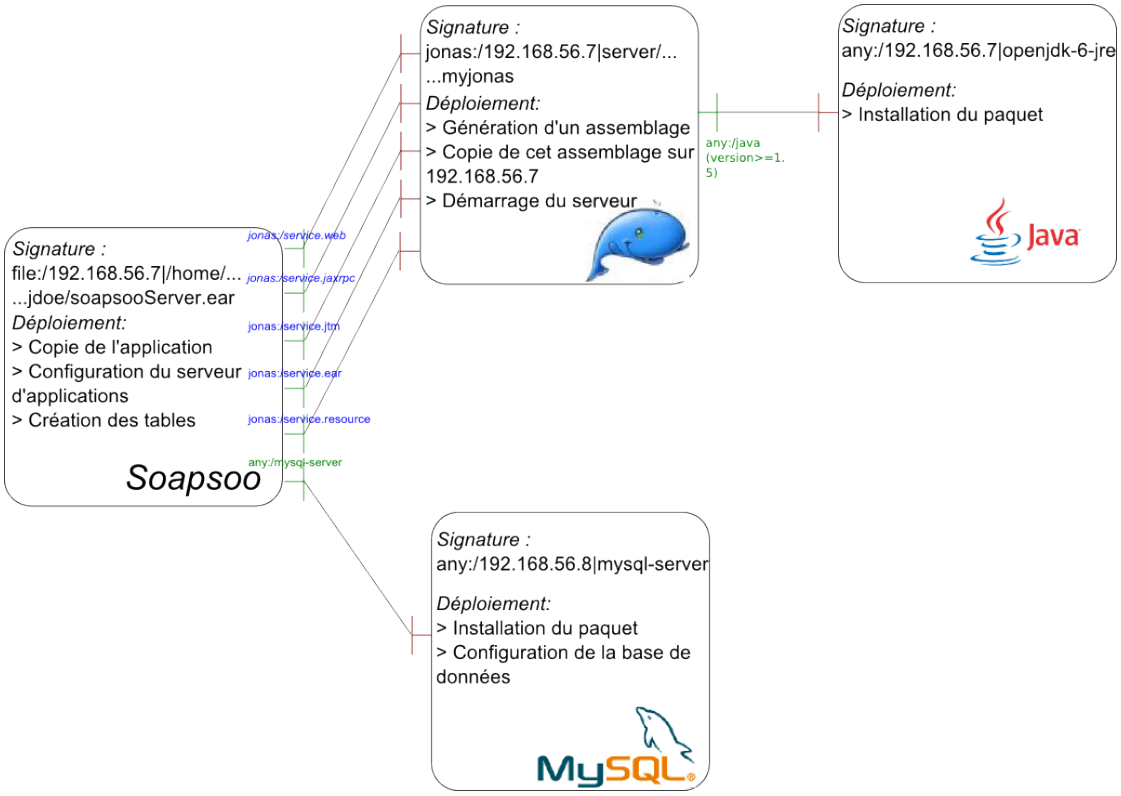


FIGURE 7.3 – Modélisation du déploiement à partir de composants abstraits

La signature des composants abstraits représentant *Java* et *MySQL* ne spécifie pas de système *SoftwareUnit*. Le descripteur de besoin du composant représentant *Soapsoo*, dont la méta-donnée est *any :mysql-server*, contient l'alias *mysql.server* afin de ne pas dépendre d'un système de gestion de paquets (le nom du paquet fournissant un serveur *MySQL* est différent selon les distributions *GNU/Linux*). Similairement, le descripteur de besoin du composant représentant *JOnAS*, dont la méta-donnée est *any :java*, contient l'alias *java*.

Les systèmes *SoftwareUnit* et les alias sont résolus lors du déploiement. Dans le contexte du *cloud computing*, les adresses des machines ne peuvent être connues qu'au déploiement. Notre modèle autorise donc à ce qu'elles soient fournies après la définition de la description architecturale.

Les composants abstraits décrits dans la figure 7.3 n'utilisent pas la même usine à déploiements :

- Le composant représentant *Java* utilise une usine effectuant seulement l'installation de paquets ;
- Une usine exécutant un script *FScript* est préférée pour les autres composants, afin d'effectuer des taches plus complexes. Ce script contient des invocations d'opérations sur des composants *SoftwareUnit*.

L'annexe B contient le script *FScript* construisant le graphe de composants abstraits décrit par la figure 7.3.

7.3.4 Exécution du déploiement

Le déploiement s'effectue selon la politique de résolution basée sur les composants abstraits (définie dans la section 4.3.4.4). Le graphe de composants est parcouru en profondeur d'abord afin de déployer en priorité les dépendances. Le déploiement d'un composant abstrait s'effectue en invoquant son usine à déploiements avec comme arguments, sa signature, sa configuration et le contexte environnemental de la cible du déploiement. Cette politique ordonnance l'exécution des opérations selon l'ordre des étapes énoncées dans la section 7.3.2. Le composant *SoftwareUnit* résultant représente le logiciel déployé sur la cible et peut lui être substitué, car possédant un sous-type du composant abstrait.

Dans notre exemple, le graphe de composants résultant est représenté par la figure 7.4.

7.4 Conclusion

Nous avons présenté dans ce chapitre une solution pour le déploiement d'applications hétérogènes sur des cibles hétérogènes.

Notre approche consiste en la définition du processus de déploiement sous forme d'un graphe de composants abstraits, chacun de ces composants représentant une étape du déploiement. Cette définition ne se préoccupe pas des implémentations des logiciels, celles-ci étant découvertes au dernier moment. Le processus de déploiement est exécuté à l'aide d'une politique de résolution des composants permettant d'ordonnancer l'invocation des opérations sur les composants *SoftwareUnit*. Au final, l'opération de résolution retourne un graphe de composants représentant l'application déployée.

Le prochain chapitre proposera une analyse des coûts de nos contributions.

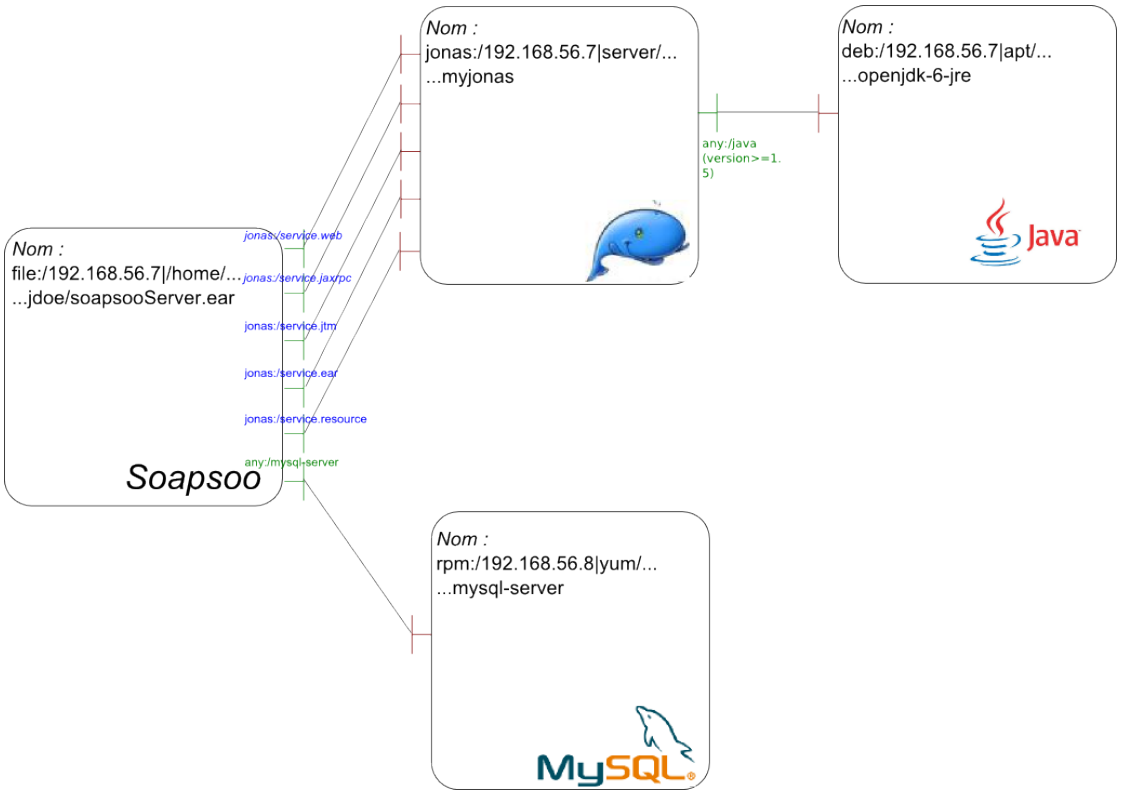


FIGURE 7.4 – Modélisation de l'application déployée

Chapitre 8

Analyse de coûts

Sommaire

Une analyse des contributions proposées dans cette thèse est donnée dans ce chapitre. Ainsi, nous mesurons quantitativement les coûts de notre approche, tout en comparant ceux émanants des solutions actuelles. Dans un premier temps (section 8.1), nous évaluons le coût pour l'utilisateur, puis le coût pour l'intégration d'un nouveau système *SoftwareUnit* (section 8.2). Enfin, l'efficacité de notre approche, à travers l'impact sur les ressources du système, est commentée dans la section 8.3.

8.1 Coût pour l'utilisateur

Dans cette section, nous expliquons pourquoi notre approche est plus simple pour l'utilisateur concernant l'assemblage de logiciel qu'une solution actuelle. Nous considérons l'application *JOnAS* construite à partir de l'outil *Maven*, lequel est très largement utilisé dans la communauté *Java* (à titre de justification, la majorité des projets matures hébergés par le consortium *OW2*[1] utilise *Maven*).

8.1.1 Usages de *Maven* dans *JOnAS*

Maven (présenté dans la section 3.3.1) constitue l'état de l'art des outils d'assemblage. Cet outil est notamment utilisé dans le projet *JOnAS* pour effectuer les tâches suivantes :

Génération de *bundles* Les *bundles* contiennent les API et implémentations de services, ainsi que des librairies.

1. Consortium OW2 : `http://ow2.org/`

Génération de plans de déploiement Les plans de déploiement répertorient tous les *bundles* nécessaires pour l'implémentation d'un service.

Génération du conteneur client Le conteneur client est une plate-forme d'exécution simplifiant l'écriture des clients lourds *Java EE*.

Publication et récupération de fichiers sur des dépôts *Maven* *Maven* définit un référentiel pour publier et récupérer des fichiers. Les dépendances de *JOnAS* sont ainsi obtenues via celui-ci.

Intégration continue *Maven* fournit des facilités pour les tests unitaires et d'intégration. Le cycle de vie de déploiement de *Maven* intégrant la phase de tests, chaque *bundle* ou librairie est testé lors de sa construction.

Génération de l'assemblage Les distributions *Micro JOnAS* et *JOnAS complet* sont générées sous forme de répertoires ou d'archives.

L'assemblage consiste principalement en l'exécution séquentielle d'opérations sur des fichiers. L'écriture d'un profil nécessite actuellement que l'utilisateur ait une bonne connaissance de ce processus d'assemblage. De plus, la définition d'un assemblage met en jeu deux types de fichiers pouvant se recouvrir :

Descripteur d'assemblage (assembly.xml) Fichier de type déclaratif décrivant le contenu de l'assemblage en termes de fichiers et d'artefact *Maven* ;

Modèle objet de projet (pom.xml) Fichier contenant notamment une séquence d'opérations sur les fichiers (sous forme de plugins).

Le besoin de ces deux fichiers provient des faiblesses du fichier de descripteur d'assemblage. Celui-ci n'offre que des opérations prédéfinies et n'est pas extensible. L'utilisation de plugins associés au projet devient alors indispensable.

Nous proposons une alternative dans laquelle l'utilisateur se focalise sur le besoin de fonctionnalités et non sur le moyen de les obtenir.

8.1.2 Solution *SUF*

Notre solution propose de séparer les préoccupations relatives à l'assemblage. Ainsi, en préambule chaque élément d'un assemblage de *JOnAS* doit être modélisé à l'aide de composants *SoftwareUnit*. Cette représentation est faite une seule fois et est indépendante de la définition des assemblages. Elle peut être effectuée par l'équipe *JOnAS*. La définition de l'assemblage peut être également effectuée par la même équipe, mais aussi par des utilisateurs de *JOnAS* désirant avoir un profil spécifique.

JOnAS étant associé à un projet *Maven*, la construction de la représentation peut être automatisée, en partie grâce à l'utilisation d'un plugin générant une description *SUD*, pour chacun des artefacts du projet. Certains éléments ne sont pas réifiés dans le processus d'assemblage *Maven*. C'est notamment le cas des services techniques et des fichiers de configuration. Nous avons ainsi défini un fichier *ADL*, pour chacun des services (comme expliqué dans la section 6.3.2), définissant les dépendances et le contenu de ceux-ci. Concernant la configuration, nous avons défini des opérations dont le résultat

est un ensemble de fichiers de configurations. Par exemple, le listing 8.1 décrit l'opération permettant de représenter la configuration de *Jetty 6.1*[2].

```
<group id="JOnAS">
    <operation id="jonas-web-container-jetty-6.1-unpack"
            name="file-unpack-installer">
        <configuration>
            <includeBaseDirectory>false</includeBaseDirectory>
            <includes>
                <include>jetty6.*.xml</include>
            </includes>
        </configuration>
        <software-unit id="org.ow2.jonas:jonas-web-container-jetty-6.1:jar"
                system="file">
            <file xsi:type="m2:maven2-deploymentType">
                <m2:groupId>org.ow2.jonas</m2:groupId>
                <m2:artifactId>jonas-web-container-jetty-6.1</m2:artifactId>
                <m2:version>5.2.0-M3</m2:version>
            </file>
        </software-unit>
    </operation>
</group>
```

Listing 8.1 – Définition d'une opération dont le résultat est l'ensemble des fichiers de configuration de *Jetty 6.1*

Dans cet exemple, l'opération extrait la configuration contenue dans le *bundle* contenant le conteneur *web Jetty*, puis filtre les fichiers de configuration grâce à une expression régulière sur leur nom. Cette opération est référencée dans la description ADL du service *web* de *JOnAS* (cf. le listing 4.22), afin d'inclure la configuration dans le composite représentant le service.

Chaque élément du serveur étant représenté, la définition d'un assemblage peut être effectuée selon deux alternatives : l'écriture d'un fichier XML *SUD* ou celle d'une expression *FScript*. Le listing 8.2 définit le fichier *SUD* servant à générer un assemblage du profil *Micro JOnAS*.

2. Jetty : `http://jetty.codehaus.org/jetty/`

```
<su-script>
    <operation id="Micro_JOnAS_assembly" name="file-assembly-installer">
        <configuration>
            <formats>
                <format>dir</format>
            </formats>
            <target>C:/Users/loris/Desktop/toto</target>
        </configuration>
        <!-- Micro JOnAS with minimal services -->
        <operation id="Micro_JOnAS_with_services" name="jonas-deployer">
            <configuration>
                <jonas-template>${jonas:/jonas.root}</jonas-template>
                <requirements>
                    <requirement>service.wc</requirement>
                    <requirement>service.depmonitor</requirement>
                </requirements>
            </configuration>
        </operation>
    </operation>
</su-script>
```

Listing 8.2 – Fichier SUD définissant un assemblage du profil Micro JOnAS

Une manière alternative est d'utiliser une expression *FScript*, comme décrit dans le listing 8.3.

```
file-assembly-installer(
    jonas-deployer(
        {"jonas-template"="${jonas:/jonas.root}",
        "requirements"=["service.wc","service.depmonitor"]}),
    {"formats"=["dir"]})
```

Listing 8.3 – Expression FScript définissant un assemblage du profil Micro JOnAS

Ces définitions d'assemblage sont centrées sur l'essentiel et ne contiennent pas d'instructions autres que celles concernant la génération de l'assemblage. Pour simplifier au maximum la tache de l'utilisateur, nous avons trié toutes les instructions spécifiques aux services techniques ou aux librairies, afin de les regrouper dans nos abstractions des éléments de *JOnAS*. Ceci allège la description, car l'utilisateur n'a plus besoin de se préoccuper de ces instructions, celles-ci étant automatiquement exécutées lors de la sélection de l'élément associé.

8.1.3 Comparaison de la taille des fichiers requis pour la définition d'un assemblage

Le tableau 8.1 compare la taille des fichiers à écrire pour la création du profil *Micro* (chaque fichier étant de type XML).

Le tableau 8.2 compare la taille des fichiers à écrire pour la création du profil complet.

La forte différence entre les deux solutions, quelque soit le profil cible, s'explique par

	Nombre de lignes
Maven	230 (POM) + 27 (assembly)
SUF	20 (SUD)

TABLE 8.1 – Comparaison du nombre de lignes de codes nécessaires pour la définition d'un assemblage du profil *Micro JOnAS*

	Nombre de lignes
Maven	207 (POM) + 143 (assembly)
SUF	54 (SUD)

TABLE 8.2 – Comparaison du nombre de lignes de codes nécessaires pour la définition d'un assemblage du profil *JOnAS complet*

un mélange des préoccupations : les fichiers *POM* contiennent de nombreuses instructions qui ne sont pas spécifiques aux profils et qui devraient être définies ailleurs. Ce manque de séparation des tâches augmente ainsi notoirement la complexité de création d'un nouveau profil selon le modèle actuel (pour preuve il n'y en a pas).

Ce point explique le fait que le fichier *POM* du profil *JOnAS complet* soit plus petit que celui du profil *Micro*. En effet, les fichiers *POM* supportant l'héritage, le fichier *POM* du profil *JOnAS complet* hérite ainsi d'instructions du profil *Micro*.

8.2 Coût pour l'intégration d'un système *SoftwareUnit*

Dans cette section nous décrirons l'effort nécessaire pour intégrer un système *SoftwareUnit*.

8.2.1 Implémentation du coeur du système

Au minimum trois interfaces doivent être implémentées :

org.ow2.jonas.su.api.system.SoftwareUnitSystem Interface représentant le système ;

org.ow2.jonas.su.api.INamingContext Interface représentant le contexte de nom du système ;

org.ow2.jonas.su.api.metadata.MetadataParser Interface représentant l'analyseur de méta-données.

8.2.2 Définition d'une nouvelle membrane

Le système peut définir une nouvelle membrane afin de représenter les interfaces de contrôles du système au niveau du modèle. Le système *OSGi* définit ainsi une membrane permettant d'interagir avec le résolveur de la plate-forme (voir la section 5.4.2.3).

8.2.3 Définition d'opérations

Les opérations de déploiement spécifiques au système peuvent être encapsulées par un objet exposant l'interface org.ow2.jonas.su.api.operation.SUOperation (voir la section 4.3.4.1).

8.2.4 Empaquetage du système

Les classes du système (comprenant opérations et membranes) doivent être empaquetées sous forme de un ou plusieurs *bundles OSGi*. Chaque système fourni est ainsi divisé en deux *bundles* afin de séparer les APIs et implémentations.

8.2.5 Enregistrement du système

Systèmes et opérations sont enregistrés dans le registre à services *OSGi*. Un système doit être enregistré avec la propriété de service sus.name et une opération avec la propriété operation.name. La section 5.2.3 décrit comment le faire de manière transparente (ainsi que la membrane) lorsque le système est implémenté sous forme de composants *Fractal*. Enfin, il reste possible d'enregistrer un système sans passer par le registre à services en utilisant le gestionnaire de services (voir la section 4.3.3.4).

À titre d'information, le tableau 8.3 donne pour chaque système le nombre de classes le constituant.

Système	*OSGi*	*RPM*	*Debian*	*JOnAS*
Nombre de classes	54	18	21	23

TABLE 8.3 – Nombre de classes constituant les systèmes *SoftwareUnit*

8.3 Efficacité

Nous évaluons quantitativement, dans cette section, l'utilisation des ressources induites par notre solution. Dans un premier temps, nous comparons les temps d'exécution pour la création d'assemblage entre *Maven 3* et *SUF*. Puis, nous observons les surcoûts dans l'utilisation de primitives de déploiements représentatives. Enfin, l'empreinte mémoire de notre outil est évaluée.

8.3.1 Environnement de tests

SUF est déployé sur un système dont les caractéristiques sont les suivantes :
Processeur : Intel Core i5 CPU M 560 @ 2.67GHz
Mémoire : 4 Go
Systèmes d'exploitation : Microsoft Windows 7, Ubuntu 10.04 et Fedora 14

JVM : JRE version 1.6.0_23

Nous précisons lorsqu'une expérience requiert un type de système d'exploitation en particulier.

Afin d'éviter la prise en compte de temps aléatoires de téléchargement dans résultats, les accès réseaux ont été désactivés autant que possible.

8.3.2 Temps exécution

SUF se positionne comme une alternative à *Maven 3* pour les tâches d'assemblage et de déploiement. La section précédente compare qualitativement les deux outils. Les performances concernant l'assemblage sont comparées dans cette section. La comparaison porte sur le temps d'assemblage pour les deux profils déjà existants : *Micro JOnAS* et *JOnAS complet*. L'assemblage généré est un répertoire contenant *JOnAS 5.2M3*. Chaque comparaison comporte trois mesures :

Assemblage avec *Maven* Temps donné par l'exécution d'une commande *Maven* dans le répertoire des sources de *JOnAS 5.2M3* ;[3]

Assemblage avec *SUF* avec modélisation existante Chaque service et librairie de *JOnAS* étant déjà modélisé, le temps retourné est celui de l'agrégation de composants et de la génération de l'assemblage sous forme de fichiers ;

Assemblage avec *SUF* sans modélisation existante Le temps retourné prend en compte l'étape initiale de modélisation de toutes les composantes de *JOnAS*.

Nous faisons apparaître un temps d'assemblage n'intégrant pas la modélisation car, à la différence de *Maven*, cette modélisation existe indépendamment de l'assemblage et peut être sauvegardée pour être réutilisée ultérieurement. Il convient de noter que les différents assemblages de *JOnAS* partagent dans *SUF* la même modélisation du serveur d'applications permettant ainsi de n'effectuer cette étape qu'une seule fois.

Afin d'accompagner les mesures qui vont suivre, le tableau 8.4 compare le contenu des deux distributions de *JOnAS*.

	Bundles	Librairies	Services techniques
Micro JOnAS	83	5	2
JOnAS complet	242	29	52

TABLE 8.4 – Comparaison du contenu des deux distributions de *JOnAS*

8.3.2.1 Assemblage du profil *Micro JOnAS*

L'assemblage sous *Maven 3* est réalisé avec la commande mvn install −pl :micro−jonas. Celui-ci intègre la génération des plans de déploiement, que nous n'avons pas pris en compte pour être équitable.

3. `svn://svn.forge.objectweb.org/svnroot/jonas/jonas/tags/JONAS_5_2_0_M3`

Le tableau 8.5 décrit les résultats pour chacune des trois mesures.

	Temps d'exécution (en secondes)
Maven 3	23
SUF avec modélisation déjà existante	3
SUF sans modélisation existante	14

TABLE 8.5 – Comparaison du temps d'assemblage du profil *Micro JOnAS*

SUF reste plus rapide que la modélisation soit comprise ou non. Le coût pour un assemblage avec modélisation existante est principalement lié aux opérations d'accès disques. Le coût pour un assemblage sans modélisation existante est majoritairement lié au coût de modélisation du serveur d'applications. La modélisation effectuée ici ne prend en compte que les briques incluses dans le profil *Micro*. Une modélisation de toutes les briques rétrograde *SUF* derrière *Maven* comme nous le verrons dans la section suivante.

8.3.2.2 Assemblage du profil *JOnAS complet*

L'assemblage sous *Maven 3* est réalisé avec la commande mvn install −pl :jonas−full. Nous ne prenons pas en compte la génération du fichier jclient.jar ayant un coût important (64 MO), laquelle étant intégrée dans le processus d'assemblage *Maven*.

Le tableau 8.6 décrit les résultats pour chacune des trois mesures.

	Temps d'exécution (en secondes)
Maven 3	38
SUF avec modélisation déjà existante	20
SUF sans modélisation existante	96

TABLE 8.6 – Comparaison du temps d'assemblage du profil *JOnAS complet*

Lorsque la modélisation est déjà existante *SUF* reste plus rapide. Néanmoins, celle-ci pénalise *SUF* dès lors que le nombre de services techniques devient conséquent. Ceci s'explique par le fait que *SUF* construit une représentation des services techniques de *JOnAS*, celle-ci n'étant pas faite par *Maven*.

8.3.2.3 Évolution du temps d'assemblage selon le profil

Afin de vérifier l'impact du nombre de services techniques sur le temps d'assemblage des profils, nous avons mesuré l'évolution de ce dernier par rapport au nombre de plans de déploiement inclus dans l'assemblage (un service étant composé de plans de déploiement). La figure 8.1 présente les résultats obtenus.

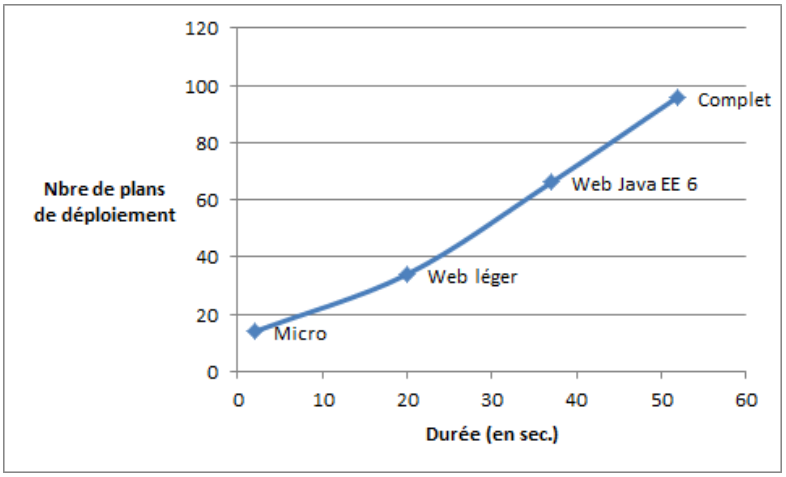


FIGURE 8.1 – Évolution du temps d'assemblage selon le nombre de plans de déploiement

Si d'autres éléments entrent en compte dans l'assemblage (le conteneur client est, par exemple, inclus à partir du profil *Web Java EE 6*), le nombre de services techniques est le principal élément impactant cette évolution.

8.3.3 Surcoût sur des primitives de déploiement natives

À la différence de l'utilisation précédente proposant une solution alternative à l'assemblage, *SUF* cherche également à faciliter l'utilisation d'outils patrimoniaux en encapsulant des primitives de déploiement existantes. Nous avons sélectionné trois primitives de déploiement spécifiques, relatives à des systèmes de modules différents. Pour le système *OSGi*, nous considérons la résolution de *bundle*, pour le système *RPM* l'installation d'un paquet local et pour le système *Debian* l'installation d'un paquet distant.

Les résultats pour chaque primitive sont obtenus en effectuant une moyenne sur 50 expériences.

8.3.3.1 Résolution d'un *bundle*

Dans ce test, nous considérons l'installation puis la résolution d'un *bundle* donné. L'interface BundleContext fournit une méthode standard pour l'installation de *bundle* et l'interface PackageAdmin en fournit une pour la résolution de *packages* ou de *bundles*. *SUF* définit les opérations osgi−bundle−installer et osgi−bundle−resolver pour respectivement l'installation et la résolution de *bundles*, lesquelles utilisent les interfaces précitées. La résolution de *bundle* met également en jeu, au sein du modèle, des mécanismes asynchrones comme décris dans la section 5.4.2.3.

Le listing 8.4 contient la définition d'une fonction *FScript* créant une modélisation d'un *bundle* sous forme de composant *SoftwareUnit*, puis l'installant et le résolvant.

```
function resolve-bundle(name) {
    file=create-su("osgi",$name);
    bundle=osgi-bundle-installer($file);
    osgi-bundle-resolver($bundle);
}
```

Listing 8.4 – Script de résolution d'un *bundle OSGi*

Le *bundle* déployé est l'implémentation du service EventAdmin de *Felix*. Le tableau 8.7 décrit les résultats pour chacune des trois mesures.

	Temps d'exécution (en secondes)
Méthode standard	0.1
SUF avec modélisation déjà existante	0.1
SUF sans modélisation existante	0.1

TABLE 8.7 – Comparaison des temps de résolution d'un *bundle OSGi*

Le surcoût pour un *bundle* est nul, que ce soit au niveau de la modélisation ou de l'encapsulation des primitives.

8.3.3.2 Installation d'un fichier *RPM*

Nous comparons le temps d'installation du fichier flash−plugin−10.1.102.64−release.i386.rpm, via la commande rpm −i sur *Fedora 14*. La mesure du temps d'exécution des commandes systèmes est effectuée à l'aide de la commande time.

Le tableau 8.8 décrit les résultats pour chacune des trois mesures.

	Temps d'exécution (en secondes)
rpm -i	1.2
SUF avec modélisation déjà existante	1.3
SUF sans modélisation existante	1.3

TABLE 8.8 – Comparaison des temps d'installation d'un fichier *RPM*

Pour un fichier *RPM*, le surcoût pour l'installation est quasi nul que ce soit au niveau de la modélisation ou de l'encapsulation des primitives. La figure 8.2 compare les temps d'installation de plusieurs fichiers *RPM*.

Les temps d'installation restent linéaires et le surcoût reste faible quand la modélisation est réalisée (les courbes bleu et rouge étant superposées).

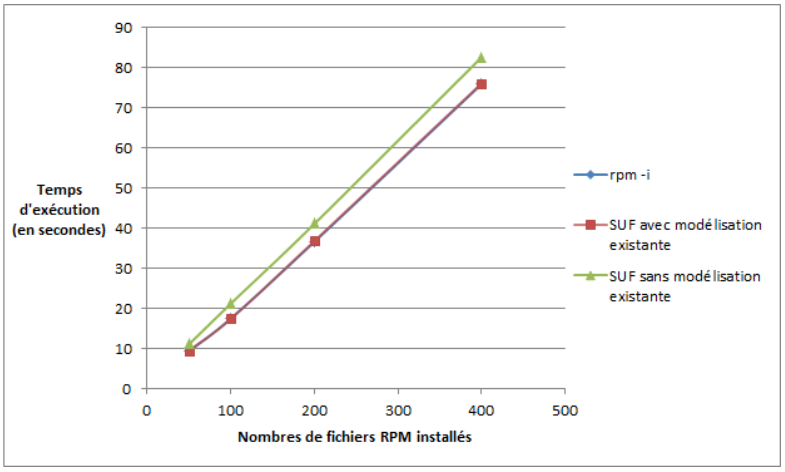


FIGURE 8.2 – Comparaison des temps d'installation de plusieurs fichiers *RPM*

8.3.3.3 Installation d'un paquet *Debian*

Nous comparons le temps d'installation du paquet gnugo, via la commande apt−get install sur *Ubuntu 10.04*. Le listing 8.5 contient la définition d'une fonction *FScript* créant une modélisation d'un paquet *Debian* sous forme de composant *SoftwareUnit* et installant celui-ci.

```
function deb-apt(name,args) {
    su=create-su("deb",$name);
    deb-apt-installer($su,$args);
}
```

Listing 8.5 – Script d'installation d'un paquet *Debian*

Le tableau 8.9 décrit les résultats pour chacune des trois mesures.

	Temps d'exécution (en secondes)
apt-get install	7.8
SUF avec modélisation déjà existante	7.8
SUF sans modélisation existante	7.9

TABLE 8.9 – Comparaison des temps d'installation de plusieurs paquets *Debian*

Le surcoût pour l'installation du paquet est quasi nul, que ce soit au niveau de la modélisation ou de l'encapsulation des primitives.

8.3.4 Empreinte système

L'empreinte système est mesurée sur le système d'exploitation *Windows 7*. Nous utilisons l'indicateur Private bytes [4], caractérisant la mémoire allouée non partageable avec d'autres processus.

Nous comparons les empreintes mémoires suivantes :

1. profil *Micro JOnAS* ;
2. *SUF* déployé sans système *SoftwareUnit* ;
3. *SUF* déployé avec les systèmes *SoftwareUnit* décrits dans la section 5.4 ;
4. profil *JOnAS complet*.

SUF sans système *SoftwareUnit* contient 48 *bundles*, lesquels sont constitués des librairies *Fractal*, du coeur de *SUF*, ainsi que quelques outils tels qu'un connecteur *RMI* et un *shell*.

SUF avec systèmes *SoftwareUnit* ajoute 13 *bundles* supplémentaires, parmi lesquels se trouvent quelques outils, tels le modèle *SUF* pour *FScript* et le support des fichiers *SUD*.

Le tableau 8.10 décrit les résultats pour chacune des quatre mesures.

	Empreinte mémoire (en MO)
Micro JOnAS (services par défaut)	158
SUF sans système	209
SUF avec systèmes	336
JOnAS complet (services par défaut)	417

TABLE 8.10 – Empreinte mémoire du *SU Framework*

On notera le coût plus important apporté par les systèmes *SoftwareUnit* plutôt que par le coeur du canevas. Ceci découle du fait que le système *OSGi* crée une représentation de tous les bundles installés sur la plate-forme *OSGi*. En outre, nous avons déployé ici tous les systèmes existants, chose pouvant être inutile selon les contextes d'utilisation.

8.3.5 Analyse des résultats

Le coût en terme d'efficacité est relatif aux fonctionnalités utilisées de *SUF*. Ce coût est quasi nul pour une utilisation de primitives de déploiement natives telles que *rpm* ou *apt-get*. Le surcoût introduit dépend plutôt du niveau de représentation des modules et de leur quantité. Par exemple, dans le cas de l'expérimentation sur *JOnAS*, nous avons constaté que le temps nécessaire pour la génération d'un assemblage était tributaire du nombre de services techniques sélectionnés. Ces éléments, inclus dans l'assemblage, bénéficient d'une représentation dans *SUF*, ce qui n'est pas le cas dans *Maven*. Pourtant celle-ci est nécessaire afin de construire un assemblage à la juste taille et cohérent.

4. `http://technet.microsoft.com/en-us/library/cc958292.aspx`

De manière générale, le surcoût dépend du niveau de granularité de la représentation. L'utilisateur a donc intérêt à sélectionner le niveau le plus juste à l'aide du gestionnaire de granularité présenté dans la section 5.3. Nous pensons que le coût introduit par cette représentation est acceptable par rapport aux bénéfices obtenus en terme de fonctionnalités.

8.4 Conclusion

Ce chapitre donne une évaluation de nos contributions. Nous avons montré que notre approche simplifie à l'utilisateur la description d'assemblages, grâce à une séparation des préoccupations. L'utilisateur ne sélectionne dans le dépôt que les modules dont il a besoin et n'a plus besoin d'écrire des instructions spécifiques aux éléments de *JOnAS*. Notre approche est plus déclarative.

L'intégration d'un système *SoftwareUnit* est conçue pour être relativement simple, tant sur le plan de la conception que de l'activation. *SUF* fournit une API à implémenter et requiert l'empaquetage du système à l'aide de bundles *OSGi*.

Enfin, nous pensons que l'efficacité de notre solution en terme de performances est tout à fait positive relativement aux fonctionnalités apportées. *SUF* fournit en outre des possibilités de configuration afin d'optimiser les performances.

Chapitre 9

Conclusion

Sommaire

Ce chapitre de conclusion clôt la présentation de nos contributions. Dans un premier temps, nous rappelons la problématique dans la section 9.1. Puis, nous dressons un bilan des contributions dans la section 9.2. Ensuite, nous indiquons quelles sont leurs limitations actuelles dans la section 9.3. Enfin, nous présentons des pistes pour exploiter ce travail dans la section 9.4.

9.1 Rappel de la problématique

La problématique de cette thèse est le déploiement d'applications hétérogènes à la juste taille. Nous considérons un large cycle de vie du processus de déploiement, puisque intégrant l'assemblage lors de la sortie du logiciel. Cette problématique est illustrée par deux exemples issus d'un besoin industriel : la gestion de profil *JOnAS* et le déploiement d'applications *Java EE* avec leur environnement d'exécution.

9.2 Bilan des contributions

Notre approche se base sur un modèle d'abstraction permettant de construire une représentation unifiée du déploiement et du logiciel. Un processus de déploiement peut être décrit soit de manière implicite, à partir d'une description architecturale, soit de manière explicite, en terme de plan de déploiement. Les étapes d'un processus de déploiement sont représentées par des fonctions (appelées *opérations*), opérant sur des abstractions des logiciels à déployer. Les abstractions des logiciels sont des composants *SoftwareUnit*, offrant des interfaces de contrôle du logiciel.

La mise en œuvre de ce modèle est réalisée dans le projet *SU Framework* et a été expérimentée sur les scénarios relatifs au serveur d'applications *JOnAS*, cités précédemment.

Afin d'évaluer le modèle proposé, nous avons choisi de fixer des exigences sur les critères de caractérisation proposés par Carzaniga et al. [16] (dans la section 3.1).

Les sous-sections suivantes expliquent comment chaque exigence est validée par notre solution.

Fine granularité

Nous avons proposé une solution dans laquelle tout élément d'un logiciel peut être représenté avec un composant *SoftwareUnit*. Cette représentation peut évoluer au cours du temps.

Contrôle distribué et extensible

Grâce au modèle à composants *Fractal*, il est possible de contrôler un logiciel directement à partir de sa représentation en composant. Un contrôleur garantit la cohérence de la représentation. Chaque système *SoftwareUnit* apporte un jeu de contrôleurs spécialisés.

Hétérogénéité

Le support de méta-données est extensible grâce au mécanisme de systèmes *SoftwareUnit*. Les composants *SoftwareUnit* abstraits permettent de s'abstraire de l'hétérogénéité de l'environnement des cibles du déploiement.

Politique de résolutions

La résolution de dépendances entre composants *SoftwareUnit* peut être automatisée selon divers algorithmes. Nous avons proposé une solution basée sur un dépôt, utile lorsque les composants sont disponibles, et une autre utilisant les composants abstraits pour créer les composants manquants.

Assemblage

L'assemblage est une étape du cycle de vie du déploiement. Nous représentons un assemblage sous forme de composants *SoftwareUnit*, ainsi que les éléments le constituant. Celui-ci peut avoir des formes diverses, puisque pouvant être un fichier d'archive ou même un répertoire servant de support d'exécution. Dans notre solution, les opérations ajoutent incrémentalement les éléments et la fermeture transitive de leurs dépendances dans l'assemblage, mettant à jour celui-ci.

9.3 Limitations

Cette section précise les limitations de notre implémentation et propose des solutions à celles-ci. Ces limitations concernent notamment le passage à l'échelle.

Déploiement centralisé

La solution proposée utilise une seule machine pour coordonner le déploiement. Afin de faciliter le passage à l'échelle et de simplifier le déploiement, une solution serait de définir un agent responsable du déploiement sur chaque machine administrée (eg. *Jade Node*).

Déploiement par extension

Dans notre implémentation du modèle, chaque composant *SUF* est associé à une instance d'un logiciel (identifiée notamment par une adresse réseau). Le déploiement d'une instance est décrit par extension et non par intention. Pour le déploiement à grande échelle, une solution serait la définition d'une sur-couche d'administration permettant la définition par intention de plans de déploiement. Cet outil pourrait suivre l'approche de *TUNe* [11], lequel propose l'utilisation de profils *UML* et d'un langage d'encapsulation pour spécifier le déploiement.

Installation de *SUF*

SUF est exécuté à l'aide du serveur d'applications *JOnAS* (profil *Micro*). Les utilisateurs doivent donc installer *JOnAS* sur leur machine. L'exécution à l'aide d'un fichier *JAR* unique faciliterait l'utilisation de *SUF*. Une solution peut être apportée par l'outil *JOnAS AutoStart*[1], lequel permettrait de construire de tels fichiers *JAR* embarquant le serveur et l'applicatif.

9.4 Perspectives

Nous proposons des idées de travaux réalisables grâce à *SUF*, à court et moyen terme. D'une manière générale, la définition du modèle *Fractal SoftwareUnit* ouvre la porte à de nombreuses contributions dans l'administration de logiciels. Nous en proposons deux, dans le domaine du *cloud computing*, qui nous semblent particulièrement intéressantes.

Déploiement d'appliance virtuelle

L'assemblage et le déploiement du serveur d'applications *JOnAS* à la juste taille sont permises par *SUF*. De la même manière, la création d'appliances virtuelles et leur déploiement sur une infrastructure sont possibles.

1. JOnAS AutoStart : `http://wiki.jonas.ow2.org/xwiki/bin/view/AutoStart/`

La création d'appliances utiliserait *SUF* pour modéliser l'ensemble des paquets pertinents d'une distribution *Linux*. L'installation de ces paquets sur un système d'exploitation type *JeOS* (aka. *just enough operating system*) permettrait d'obtenir une appliance légère et une solution concurrente de l'outil *UForge Appliance Factory*[2].

Le support d'une infrastructure ainsi que les opérations permettant de déployer les machines virtuelles seraient apportés à l'aide d'un nouveau système *SoftwareUnit*. Le contexte de nommage de l'infrastructure permettrait d'adresser les machines virtuelles déployées.

Solution de PaaS basée sur *SUF*

Une solution de PaaS pourrait être basée sur *SUF*. Notre approche propose une solution indépendante des langages de développement et des systèmes d'exploitation, pour la description d'architectures logicielles. Celle-ci a l'avantage d'intégrer à la fois la création d'assemblage et le déploiement dynamique de fonctionnalités. Le fournisseur de la plate-forme peut ainsi proposer une librairie de contenus, sous la forme d'un dépôt de composants *SoftwareUnit*. Le déploiement d'une application, sur notre plate-forme, déclencherait une résolution des dépendances à l'aide de ce dépôt.

Un autre apport de notre modèle est dans la souplesse du niveau de représentation, permettant ainsi de restreindre ou non la vision de la plate-forme, offerte aux clients. Le modèle permet d'exposer très simplement des détails de la plate-forme, tels que les permissions sur le systèmes de fichiers.

2. `https://www.usharesoft.com/products/uforge-appliance-factory.html`

Quatrième partie

Annexes

Annexe A

Script *SUD* définissant le profil *JOnAS Web léger*

```
<su-script>
  <operation id="weblight_assembly" name="file-assembly-installer">
    <configuration>
      <formats>
        <format>zip</format>
      </formats>
      <target>C:/Users/loris/weblight</target>
    </configuration>

    <operation id="weblight_with_services" name="jonas-deployer">
      <configuration>
        <jonas-template>${jonas:/jonas.root}</jonas-template>
        <requirements>
          <requirement>service.db</requirement>
          <requirement>service.dbm</requirement>
          <requirement>service.wc</requirement>
          <requirement>service.depmonitor</requirement>
          <requirement>service.jaxrpc</requirement>
          <requirement>service.jtm</requirement>
          <requirement>service.web</requirement>
          <requirement>service.ear</requirement>
          <requirement>service.resource</requirement>
          <requirement>service.versioning</requirement>
        </requirements>
      </configuration>

      <software-unit id="mysqlconnector" name="lib/ext" system="jonas">
        <sub-software-units>
          <software-unit id="mysql" system="file">
            <file xsi:type="url:url-deploymentType">
              <url:resource>mysql-connector-java-5.1.14-bin.jar</url:resource>
            </file>
          </software-unit>
        </sub-software-units>
      </software-unit>

      <operation id="add_deployable_capability" name="type-modifier">
        <configuration>
          <interfaceTypeName>deployable</interfaceTypeName>
          <metadata>jonas:/deployable</metadata>
        </configuration>
        <software-unit id="soapsooClient" system="file">
          <file xsi:type="url:url-deploymentType">
            <url:resource>soapsooClient.ear</url:resource>
          </file>
        </software-unit>
        <software-unit id="soapsooServer" system="file">
          <file xsi:type="url:url-deploymentType">
            <url:resource>soapsooServer.ear</url:resource>
          </file>
        </software-unit>
        <software-unit id="jonasMysqlDatasource" system="file">
          <file xsi:type="url:url-deploymentType">
```

```
      <url:resource>jonasMysqlDatasource.rar</url:resource>
    </file>
  </software-unit>
</operation>
<operation id="add_conf_capability" name="type-modifier">
  <configuration>
    <interfaceTypeName>conf</interfaceTypeName>
    <metadata>jonas:/conf</metadata>
  </configuration>
  <software-unit id="jonas.properties" system="file">
    <file xsi:type="url:url-deploymentType">
      <url:resource>jonas.properties</url:resource>
    </file>
  </software-unit>
  <software-unit id="MySQL.properties" system="file">
    <file xsi:type="url:url-deploymentType">
      <url:resource>MySQL.properties</url:resource>
    </file>
  </software-unit>
</operation>

<software-unit id="libClientBootstrap" name="lib/bootstrap"
               system="jonas">
  <sub-software-units>
    <operation id="client_bootstrap_copy" name="file-copy-installer">
      <configuration>
        <target>client-bootstrap.jar</target>
      </configuration>
      <reference id="client_bootstrap">
        <id-ref>ClientLibs/org.ow2.jonas:client-bootstrap</id-ref>
      </reference>
    </operation>
  </sub-software-units>
</software-unit>

<software-unit id="libCommon" name="lib/common" system="jonas">
  <sub-software-units>
    <operation id="util-ant-tasks_copy" name="file-copy-installer">
      <configuration>
        <target>util-ant-tasks.jar</target>
      </configuration>
      <reference id="util-ant-tasks">
        <id-ref>AntTasks/org.ow2.util:util-ant-tasks</id-ref>
      </reference>
    </operation>
    <operation id="cxf-anttasks_copy" name="file-copy-installer">
      <configuration>
        <target>cxf-anttasks.jar</target>
      </configuration>
      <reference id="cxf-anttasks">
        <id-ref>AntTasks/org.apache.cxf:cxf-anttasks</id-ref>
      </reference>
    </operation>
```

```
        </sub-software-units>
      </software-unit>

      <operation id="add_lib_capability" name="type-modifier">
        <configuration>
          <interfaceTypeName>lib</interfaceTypeName>
          <metadata>jonas:/lib</metadata>
        </configuration>
        <operation id="jonas-generators-raconfig_copy" name="file-copy-installer">
          <configuration>
            <target>jonas-generators-raconfig.jar</target>
          </configuration>
          <reference id="jonas-generators-raconfig">
            <id-ref>ClientLibs/org.ow2.jonas:jonas-generators-raconfig</id-ref>
          </reference>
        </operation>
        <operation id="jonas-cluster-daemon_copy" name="file-copy-installer">
          <configuration>
            <target>jonas-cluster-daemon.jar</target>
          </configuration>
          <reference id="jonas-cluster-daemon">
            <id-ref>ClientLibs/org.ow2.jonas:jonas-cluster-daemon</id-ref>
          </reference>
        </operation>
        <operation id="jaxb-xjc_copy" name="file-copy-installer">
          <configuration>
            <target>jaxb-xjc.jar</target>
          </configuration>
          <reference id="jaxb-xjc">
            <id-ref>ClientLibs/com.sun.xml.bind:jaxb-xjc</id-ref>
          </reference>
        </operation>
      </operation>

      <operation id="dps" name="deploymentplan-loader">
        <reference id="dp-ctxroot">
          <id-ref>Deployables/ctxroot</id-ref>
        </reference>
        <reference id="dp-doc">
          <id-ref>Deployables/doc</id-ref>
        </reference>
        <reference id="dp-jonasAdmin">
          <id-ref>Deployables/jonasAdmin</id-ref>
        </reference>
      </operation>
    </operation>
  </operation>
</su-script>
```

Annexe B

Script *FScript* de déploiement de l'application *Soapsoo*

```
function soapsoo-install (name,env,context) {
    soapsoo = map-get($context,"abstract-su");
    script=create-su("file","192.168.56.8|/home/loris/soapsoo-config.sh");
    file-launcher($script);
    return $soapsoo;
}

function jonas-install(name,env,context) {
    jonas=weblight();
    jonas_copy=file-copy-installer($jonas,
                        {"url"="192.168.56.7","target"="/home/loris"});
    script=create-su("file","192.168.56.7|/home/loris/jonas-config.sh");
    file-launcher($script);
    return $jonas;
}

function mysql-install (name,env,context) {
    mysql-su=package-manager-depfactory($name,$env,{"yes"="true"});
    script=create-su("file","192.168.56.8|/home/loris/mysql-config.sh");
    file-launcher($script);
    return $mysql-su;
}

action full-deploy() {
    adlparam = {"start"="false","adl.backend"="su"};

    soapsoo-adl=create-su("fractal",
                        "fractal:/adlDesc/org.ow2.jonas.su.soapsoo.Soapsoo");
    soapsoo-su=fractal-launcher($soapsoo-adl,$adlparam);
    set-abstract-su-config($soapsoo-su,{"function"="soapsoo-install"});

    java-adl=create-su("fractal", "fractal:/adlDesc/org.ow2.jonas.su.soapsoo.Java");
    java-su=fractal-launcher($java-adl,$adlparam);
    set-abstract-su-config($java-su,{"yes"="true"});

    jonas-adl=create-su("fractal", "fractal:/adlDesc/org.ow2.jonas.su.soapsoo.JOnAS");
    jonas-su=fractal-launcher($jonas-adl,$adlparam);
    set-abstract-su-config($jonas-su,{"function"="jonas-install"});

    mysql-adl=create-su("fractal", "fractal:/adlDesc/org.ow2.jonas.su.soapsoo.MySQL");
    mysql-su=fractal-launcher($mysql-adl,$adlparam);
    set-abstract-su-config($mysql-su,{"function"="mysql-install"});

    bind($soapsoo-su/interface::mysql,$mysql-su/interface::mysql);
    bind($soapsoo-su/interface::web,$jonas-su/interface::web);
    bind($jonas-su/interface::java,$java-su/interface::java);

    su=abstract-su-resolver($soapsoo-su);
}
```

Bibliographie

[1] Takoua Abdellatif. *Apport des architectures à composants pour l'administration des intergiciels*. PhD thesis, Institut National Polytechnique de Grenoble, 2006.

[2] Takoua Abdellatif, Jakub Kornas, and Jean-Bernard Stefani. J2EE Packaging, Deployment and Reconfiguration Using a General Component Model. In *Component Deployment, 3rd International Working Conference, CD 2005*, volume 3798 of *LNCS*. Springer, 2005.

[3] Takoua Abdellatif, Jakub Kornas, and Jean-Bernard Stefani. Reengineering J2EE Servers for Automated Management in Distributed Environments. *IEEE Distributed Systems Online*, 8(11) :1, 2007.

[4] Areski Flissi, Jérémy Dubus, Nicolas Dolet, and Philippe Merle. Deploying on the Grid with DeployWare. In *Proceedings of IEEE CCGrid'08 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 177–184, France, 2008.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4) :50–58, 2010.

[6] Gordon Blair, Thierry Coupaye, and Jean-Bernard Stefani. Component-based architecture : the Fractal initiative. *Annals of Telecommunications*, 64 :1–4, 2009. 10.1007/s12243-009-0086-1.

[7] B Boehm. A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, 11(4) :14–24, 1986.

[8] J. Boender. *Étude formelle des distributions de logiciel libre*. PhD thesis, Université Paris Diderot — Paris 7, 2011.

[9] J. Boender, R. Di Cosmo, B. Durak, X. Leroy, M. Lijour, F. Mancinelli, T. Milo, M. Morgado, D. Pinheiro, R. Suarez, R. Treinen, P. Trezentos, and T. Zur. WP2 - D2.2 - Formal management of software dependencies. Technical report, EDOS Project, 2006.

[10] S. Bouchenak, N. De Palma, D. Hagimont, and C. Taton. Autonomic Management of Clustered Applications. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–11, 2006.

[11] Laurent Broto, Patricia Stolf, Jean-Paul Bahsoun, Daniel Hagimont, and N. Depalma. Spécification de politiques d'administration autonome avec Tune. In *Conférence Française sur les Systèmes d'Exploitation (CFSE), Fribourg, Suisse*, page (support électronique), http ://www.sigops-france.fr/, février 2008. Chapitre Français de l'ACM SIGOPS.

[12] Alan W. Brown. *Large-Scale, Component Based Development.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[13] E. Bruneton, T. Coupaye, and J.B. Stefani. *The Fractal Component Model.* OW2 Consortium, 2.0.3 edition, 2004.

[14] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java : Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, 36(11-12) :1257–1284, 2006.

[15] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture : a system of patterns.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

[16] Antonio Carzaniga, Alfonso Fuggetta, Richard S. Hall, Dennis Heimbigner, André Van Der Hoek, and Alexander L. Wolf. A Characterization Framework for Software Deployment Technologies. Technical report, University of Colorado, 1998.

[17] Humberto Cervantes and Richard S. Hall. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 614–623, Washington, DC, USA, 2004. IEEE Computer Society.

[18] Benoit Claudel, Noël De Palma, Renaud Lachaize, and Daniel Hagimont. Self-protection for Distributed Component-Based Applications. In Ajoy Datta and Maria Gradinariu, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 4280 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin / Heidelberg, 2006.

[19] Thierry Coupaye and Jacky Estublier. Foundations of enterprise software deployment. In *CSMR*, pages 65–74, 2000.

[20] Pierre-Charles David, Thomas Ledoux, Marc Léger, and Thierry Coupaye. FPath and FScript : Language support for navigation and reliable reconfiguration of Fractal architectures. *Annals of Telecommunications*, 64 :45–63, 2009. 10.1007/s12243-008-0073-y.

[21] Noël de Palma, Sara Bouchenak, Fabienne Boyer, Daniel Hagimont, Sylvain Sicard, and Christophe Taton. Jade : Un Environnement d'Administration Autonome. *Techniques et Sciences Informatiques*, 2008.

[22] A. Dearle. Software Deployment, Past, Present and Future. In *Future of Software Engineering (FOSE '07)*. IEEE, 2007.

[23] Gan Deng, Jaiganesh Balasubramanian, William Otte, Douglas C. Schmidt, and Aniruddha Gokhale. DAnCE : A QoS-enabled Component Deployment and Configuration Engine. In *Proceedings of the 3rd Working Conference on Component Deployment*, pages 67–82, 2005.

[24] Mikael Desertot. *Une architecture flexible et adaptable pour les serveurs d'applications.* PhD thesis, Université Joseph Fourier de Grenoble, 2007.

[25] Mikael Desertot, Humberto Cervantes, and Didier Donsez. FROGi : Fractal Components Deployment over OSGi. In Welf Löwe and Mario Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 275–290. Springer Berlin / Heidelberg, 2006.

[26] Mikael Desertot, Didier Donsez, and Philippe Lalanda. A Dynamic Service-Oriented Implementation for Java EE Servers. In *SCC '06 : Proceedings of the IEEE International Conference on Services Computing*, pages 159–166, Washington, DC, USA, 2006. IEEE Computer Society.

[27] Eelco Dolstra, Merijn De Jonge, and Eelco Visser. Nix : A safe and policy-free system for software deployment. In *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92. USENIX, 2004.

[28] Eelco Dolstra and Andres Löh. NixOS : a purely functional Linux distribution. In *ICFP '08 : Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 367–378, New York, NY, USA, 2008. ACM.

[29] Jérémy Dubus. *Une démarche orientée modèle pour le déploiement de systèmes en environnements ouverts distribués.* PhD thesis, Université des Sciences et Technologies de Lille, 2008.

[30] Clement Escoffier and Richard S. Hall. Dynamically adaptable applications with iPOJO service components. In *SC'07 : Proceedings of the 6th international conference on Software composition*, pages 113–128, Berlin, Heidelberg, 2007. Springer-Verlag.

[31] Clément Escoffier. *iPOJO : Un modèle à composant à service flexible pour les systèmes dynamiques.* PhD thesis, Université Joseph Fourier de Grenoble, 2008.

[32] A. Flissi and P. Merle. A Generic Deployment Framework for Grid Computing and Distributed Applications. In *OTM Confederated International Conferences, Grid computing, high performAnce and Distributed Applications (GADA 2006)*, volume 4276 of *LNCS*. Springer, 2006.

[33] Martin Fowler. Who Needs an Architect ? *IEEE Software*, 20 :11–13, 2003.

[34] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[35] Patrick Goldsack, Julio Guijarro, Steve Loughran, Alistair Coles, Andrew Farrell, Antonio Lain, Paul Murray, and Peter Toft. The SmartFrog configuration management framework. *SIGOPS Oper. Syst. Rev.*, 43(1) :16–25, 2009.

[36] R. S. Hall, D. Heimbigner, and A. L. Wolf. A cooperative approach to support software deployment using the Software Dock. In *Proc. Int Software Engineering Conf*, pages 174–183, 1999.

[37] Richard Hall. A policy-driven class loader to support deployment in extensible frameworks. In Wolfgang Emmerich and Alexander Wolf, editors, *Component Deployment*, volume 3083 of *Lecture Notes in Computer Science*, pages 81–96. Springer Berlin / Heidelberg, 2004.

[38] Richard S. Hall, Dennis Heimbigner, Andre Van Der Hoek, and Er L. Wolf. The Software Dock : A Distributed, Agent-based Software Deployment System. Technical report, University of Colorado, 1997.

[39] Richard S. Hall, Dennis Heimbigner, Andre van der Hoek, and Alexander L. Wolf. An Architecture for Post-Development Configuration Management in a Wide-Area Network. In *ICDCS '97 : Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*, page 269, Washington, DC, USA, 1997. IEEE Computer Society.

[40] Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. Requirements for software deployment languages and schema. In *ECOOP '98 : Proceedings of the SCM-8 Symposium on System Configuration Management*, pages 198–203, London, UK, 1998. Springer-Verlag.

[41] Ian Jackson and Christian Schwarz. *Debian Policy Manual*.

[42] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, 2003.

[43] Kirk Knoernschild. Modular Architecture. `http://www.kirkk.com/modularity/`.

[44] Jakub KORNAŚ. *Contributions to software deployment in a component-based reflexive architecture*. PhD thesis, Universite Joseph Fourier de Grenoble, 2008.

[45] Matthieu Leclercq, Ali Erdem Ozcan, Vivien Quema, and Jean-Bernard Stefani. Supporting heterogeneous architecture descriptions in an extensible toolset. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 209–219, Washington, DC, USA, 2007. IEEE Computer Society.

[46] Marc Léger. *Fiabilité des reconfigurations dynamiques dans les architectures à composants*. PhD thesis, Mines ParisTech, 2009.

[47] Marc Léger, Thomas Ledoux, and Thierry Coupaye. Reliable dynamic reconfigurations in the Fractal component model. In *ARM '07 : Proceedings of the 6th international workshop on Adaptive and reflective middleware*, pages 1–6, New York, NY, USA, 2007. ACM.

[48] Yu David Liu and Scott F. Smith. Modules with interfaces for dynamic linking and communication. In Martin Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 19–76. Springer Berlin / Heidelberg, 2004.

[49] Yu David Liu and Scott F. Smith. A formal framework for component deployment. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 325–344, Portland, Oregon, USA, 2006. ACM.

[50] OMG. Deployment and configuration of component-based distributed applications specification. `http://www.omg.org/spec/DEPL/`, April 2006.

[51] OMG. Unified modeling language (uml) superstructure version 2.3 specification. `http://www.omg.org/spec/UML/2.3/`, May 2010.

[52] Oracle. *JAR File Specification.*

[53] Mike P. Papazoglou. Service-Oriented Computing : Concepts, Characteristics and Directions. *Web Information Systems Engineering, International Conference on*, 0 :3, 2003.

[54] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[55] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *Proceedings of the 2009 IEEE International Conference on Services Computing*, SCC '09, pages 268–275, Washington, DC, USA, 2009. IEEE Computer Society.

[56] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien, and Thierry Coupaye. A component model engineered with components and aspects. In Ian Gorton, George Heineman, Ivica Crnkovic, Heinz Schmidt, Judith Stafford, Clemens Szyperski, and Kurt Wallnau, editors, *Component-Based Software Engineering*, volume 4063 of *Lecture Notes in Computer Science*, pages 139–153. Springer Berlin / Heidelberg, 2006.

[57] Sylvain Sicard. *Vers de l'Auto-Réparation dans les Systèmes Répartis.* PhD thesis, Université Josepf Fourier, 2009.

[58] Venkita Subramonian, Gan Deng, Christopher Gill, Jaiganesh Balasubramanian, Liang-Jui Shen, William Otte, Douglas C. Schmidt, Aniruddha Gokhale, and Nanbor Wang. The design and performance of component middleware for QoS-enabled deployment and configuration of DRE systems. *J. Syst. Softw.*, 80(5) :668–677, 2007.

[59] Clemens Szyperski. *Component software : beyond object-oriented programming.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[60] Christophe Taton. *Vers l'Auto-Optimisation dans les Systèmes Autonomes.* PhD thesis, INPG, 2008.

[61] The OSGi$^{\text{TM}}$Alliance. *OSGi Service Platform Release 4 Core and Compendium Version 4.3 Early Draft 3 - Novembre 2010*, 2010.

[62] The OSGi$^{\text{TM}}$Alliance. *OSGi^{TM}Service Platform Release 4, Version 4.2 - Compendium Specification*, 2010.

[63] The OSGi^{TM}Alliance. *OSGi^{TM}Service Platform Release 4, Version 4.2 - Core Specification*, 2010.

[64] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds : towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1) :50–55, 2009.

[65] Toby Velte, Anthony Velte, Robert C. Elsenpeter, and Robert Elsenpeter. *Cloud Computing : A Practical Approach*. McGraw Hill Professional, October 2009.

[66] Nanbor Wang, Chris Gill, Douglas C. Schmidt, and Venkita Subramonian. Configuring real-time aspects in component middleware. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2004 : CoopIS, DOA, and ODBASE*, volume 3291 of *Lecture Notes in Computer Science*, pages 1520–1537. Springer Berlin / Heidelberg, 2004.