Une démarche orientée modèle pour le déploiement de systèmes en environnements ouverts distribués

Jérémy Dubus

. .

N° d'ordre: 4216

THÈSE

présentée

devant l'Université des Sciences et Technologies de Lille

pour obtenir le grade de :

DOCTEUR DE L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
Mention INFORMATIQUE

par

Jérémy DUBUS

Équipe d'accueil : GOAL / ADAM
École Doctorale : Sciences pour l'Ingénieur
Composante universitaire : LIFL

Titre de la thèse :

Une démarche orientée modèle pour le déploiement de systèmes en environnements ouverts distribués

À soutenir le 10 octobre 2008 devant la commission d'examen

Président :	Jean-Luc Dekeyser	PROFESSEUR - UNIVERSITÉ DE LILLE I
Rapporteurs :	Françoise Baude	MAÎTRE DE CONFÉRENCES (HDR) – UNIVERSITÉ DE NICE
	Daniel Hagimont	PROFESSEUR – INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE
Examinateurs :	Olivier Barais	MAÎTRE DE CONFÉRENCES – UNIVERSITÉ DE RENNES I
	Denis Conan	MAÎTRE DE CONFÉRENCES – TELECOM & MANAGEMENT SUDPARIS
	Sylvain Lecomte	PROFESSEUR – UNIVERSITÉ DE VALENCIENNES
Directeur :	Jean-Marc Geib	PROFESSEUR – UNIVERSITÉ LILLE I
Co-Encadrant :	Philippe Merle	CHARGÉ DE RECHERCHE – INRIA LILLE NORD-EUROPE

Avant d'entrer dans le vif du sujet, il convient naturellement de remercier les différents acteurs qui ont fait de ce travail de thèse ce qu'il est. Ces remerciements sont longs, mais je tenais à remercier tout le monde.

En premier lieu, je tiens à remercier chaleureusement Jean-Luc Dekeyser d'avoir accepté de présider le jury de cette thèse. Il s'agit d'un professeur que je respecte beaucoup, et je suis très honoré de sa présence dans ce jury. Merci également à Françoise Baude et Daniel Hagimont d'avoir accepté de rapporter cette thèse, deux chercheurs internationalement reconnus dans la communauté de l'intergiciel. C'est un immense honneur pour moi de les compter dans mon jury de thèse. Merci à Denis Conan et à Olivier Barais d'avoir accepté d'être examinateurs de cette thèse. J'ai beaucoup apprécié, tant sur le plan professionnel qu'humain de travailler avec Denis lors de sa visite dans l'équipe. C'est Olivier qui m'a donné envie de me lancer dans la recherche lors de mon projet de maîtrise, et c'est une belle manière de clore l'histoire que de l'inviter au jury de cette thèse. Merci enfin à Sylvain Lecomte d'avoir accepté un peu dans l'urgence l'invitation à cette soutenance.

Merci à Philippe Merle pour son encadrement de qualité pendant les deux premières années de thèse. La troisième fut plus compliquée, mais tout aussi riche en enseignements d'une autre nature. Un grand merci à Jean-Marc Geib qui, même s'il n'a pas pu beaucoup s'impliquer scientifiquement dans mon travail, a su répondre présent, malgré un agenda surchargé, dans les moments où j'avais le plus besoin de son soutien et de ses conseils bienveillants.

Merci à mes camarades du bureau 223, Nicolas Dolet, Damien Fournier, Alban Tiberghien et Guillaume Waignier. Plus que de simples collègues, ils sont de véritables amis sur lesquels on peut compter. Merci aussi à Areski Flissi [1] pour son aide scientifique et son soutien lors des moments de doutes de cette thèse.

Merci à tous les membres de l'équipe GOAL que j'ai croisés durant ces trois ans (la liste est aléatoirement ordonnée, et très probablement non-exhaustive et je m'excuse auprès de ceux que j'ai oubliés) : Anne-Françoise Le Meur, Lionel Seinturier, Ales Plsek, Carlos Noguera, Carlos Parra, Daniel Romero, Guillaume Dufrene, Romain Rouvoy, Jérôme Moroy, Christophe Contreras, Bassem Kosayba, Laurence Duchien, Cédric Dumoulin, Patricia Serrano-Alvarado, Naouel Moha, Nicolas Pessemier, Dolores Diaz, Frédéric Loiret, Corinne Davoust, Christophe Demarey, Olivier Caron, Bernard Carré, Gilles Vanwormhoudt, Raphaël Marvie, Alexis Muller, Renaud Pawlak, Maja D'Hondt.

La fin de cette thèse est aussi la fin d'un parcours scolaire et universitaire riche en rencontres enrichissantes. Je tiens donc à remercier des personnes comme Raphaël Trentesaux, Nicolas Waras et Marion Fleurette, qui sont autant de personnes qui m'ont beaucoup apporté, même si j'ai malheureusement un peu perdu contact avec eux. Je tiens également à remercier Alexandre Barbet, Adrien Vanreust, Grégory Sibione et Julie Boez, mes amis en dehors du cadre universitaire, avec qui j'ai passé des moments que je n'oublierai jamais.

J'en profite pour également remercier ma famille, mes frères Grégory et Samuel, leurs compagnes respectives Audrey et Christelle, ainsi que les petites Elise et Emma. Un petit clin d'œil également à la famille Patouilliart, devenue en quelque sorte ma deuxième famille, et qui m'a également aidé de différentes manières durant ces trois ans : Bernard, Monique, Ingrid, Henry-Louis et le petit Louis.

[1] mon camarade de lutte ;-)

Enfin je souhaite remercier plus particulièrement Michèle, ma mère. Sans son soutien indéfectible, son amour et ses encouragements au quotidien, rien de tout cela n'aurait été possible. Malgré un contexte financier difficile, j'ai toujours eu l'immense chance, grâce à elle, de pouvoir étudier en toute sérénité et de mener un train de vie identique voire meilleur que celui des autres étudiants. Il m'est évidemment impossible de ne pas avoir également une pensée pour mon père, Jean-Marc. J'espère juste qu'il est fier de moi, s'il me voit d'où il est.

Enfin mes derniers remerciements, les plus chaleureux, vont à Stéphanie, qui partage ma vie depuis maintenant 5 ans. Pendant ces trois ans, elle fut toujours à mon écoute, et fut aussi et surtout un rayon de soleil qui m'a aidé à me changer les idées lorsque je rentrais parfois avec beaucoup de soucis en tête. Elle m'a également montré qu'aimer quelqu'un ne signifiait pas approuver tous ses choix. Sans elle, il est certain que je n'aurais pas mené ce travail à son terme.

Résumé

Le déploiement reste l'une des étapes du cycle de vie des logiciels la moins standardisée et outillée à ce jour. Dans ce travail, nous identifions quatre grands défis à relever pour déployer des systèmes logiciels distribués et hétérogènes. Le premier défi est de réussir à initier le consensus manquant autour d'un langage générique de déploiement de logiciels. Le deuxième défi consiste en la vérification statique de déploiements logiciels décrits dans ce langage pour assurer un déroulement correct avant d'exécuter les opérations de déploiement. Le troisième défi est de réaliser une plate-forme intergicielle capable d'interpréter ce langage et d'effectuer le déploiement de n'importe quel système logiciel réparti. Enfin le quatrième défi est d'appliquer ces déploiements de systèmes dans les environnements ouverts distribués, c'est-à-dire les réseaux fluctuants et à grande échelle comme les réseaux ubiquitaires ou les grilles de calcul. Notre contribution consiste à définir une démarche de déploiement de systèmes distribués centrée sur quatre rôles pour relever ces défis : l'expert réseau, l'expert logiciel, l'administrateur système et l'architecte métier. D'un côté, l'approche DeployWare, conforme à l'ingénierie des modèles, est définie par un méta-modèle multi-rôles pour décrire le déploiement de la couche intergicielle du système ainsi que par une machine virtuelle capable d'exécuter automatiquement le déploiement de cette couche. L'utilisation d'un langage de méta-modélisation permet d'écrire des programmes de vérification statique des modèles de déploiement. De l'autre côté, l'approche DACAR propose un méta-modèle d'architecture générique pour exprimer et exécuter le déploiement d'une application métier à base de composants. Cette double approche DeployWare/DACAR permet de prendre en compte, lors de la description du déploiement, les propriétés des environnements ouverts distribués selon une approche conforme à l'informatique auto-gérée. Notre contribution est validée par plusieurs expériences pour valider la capacité de prise en charge des environnements ouverts ubiquitaires, et pour éprouver l'hétérogénéité des technologies déployables dans le monde des services d'entreprise.

Abstract

Deployment is one of the most difficult software lifecycle step, and the less standardized. First, in our work we identify four challenges to solve to handle software systems deployment. The first challenge is about to initiate consensus for standard generic software deployment language. The second challenge consists in static verification of software deployment processes described using this language. These verifications are supposed to ensure the coherency of the described deployment process. Third challenge is about implementing middleware platform able to interpret this language and perform deployment of any software system. Finally fourth challenge is to transpose these deployment processes into open distributed environements which are fluctuating, such as ubiquitous and grid environments. Our contribution then consists to define a distributed systems deployment process divided in four roles to handle these challenges : the network expert, the software expert, system administrator and business architect. On the one hand, the DeployWare approach is defined by a multi-roles metamodel to describe deployment of the middleware layer of a system, and by the virtual machine able to automatically execute the described deployment, in conformance with the model driven engineering. Using a metamodeling language allows to implement static verification programs of the deployment models. On the other side, the DACAR approach proposes a generic architecture model to express and execute the deployment of a component-based application. The DeployWare and DACAR approaches allows to take into account during the deployment description, the open distributed environments properties, in conformance with the autonomic computing approach. Our contribution is validated through many experiences in ubiquitous environments and in enterprise services world.

Table des matières

Table des figures

Liste des tableaux

Listings

Première partie

Introduction

Chapitre 1

Introduction générale

Sommaire

1.1 Contexte du travail

Les travaux présentés dans ce mémoire ont été effectués dans le cadre de l'équipe GOAL (Groupe sur les Objets et composAnts Logiciels) du Laboratoire d'Informatique Fondamentale de Lille. Ces travaux s'inscrivent également dans le cadre de l'équipe-projet INRIA Jacquard/ADAM[1].

Avec le besoin croissant de systèmes auto-gérés et l'émergence des environnements multi-échelles, les développeurs de logiciels ont besoin de prendre en charge la variabilité des machines utilisées pour exécuter leurs logiciels. Les logiciels doivent être développés de manière à pouvoir être adaptés et reconfigurés automatiquement sur des plates-formes hétérogènes, en accord avec les évolutions des technologies de communication. De ce fait, l'adaptation est désormais considérée comme un problème de première classe qui doit être pris en compte tout au long du cycle de vie du logiciel. L'objectif de l'équipe GOAL/ADAM est de fournir un ensemble de paradigmes, d'approches et de canevas basés sur des techniques avancées du génie logiciel telles que les technologies à composants [69], à aspects [40] ou encore à base de modèles [64], afin de construire des systèmes logiciels répartis adaptables, mis en œuvre dans des environnements multi-échelles pour prendre en compte l'adaptation tout au long des étapes du cycle de vie du logiciel. Deux grandes directions émanent de cet objectif : la définition d'intergiciels à base de composants pour l'adaptation et la conception d'applications distribuées s'exécutant sur des plates-formes d'adaptation.

[1] le projet Jacquard s'est terminé en 2006, et le projet ADAM en est la suite, qui a débuté en janvier 2007

1.2 Problématique

Dans le cycle de vie d'une application répartie, l'une des étapes les plus techniques à ce jour est l'étape de déploiement. Cette étape, qui survient après les étapes de conception et de tests d'un logiciel, consiste à mettre en place chaque élément du logiciel sur chacune des machines interconnectées en réseau. L'une des facettes de ce déploiement concerne le déploiement des composants métiers d'une application. En effet, un certain nombre de travaux de recherche actuels traitent de ce problème comme en témoignent les différentes conférences sur ce thème comme *Component Deployment*[2], ou encore la *Conférence sur le DEploiement et (Re)COnfiguRation (DECOR)*[3]. Cette procédure complexe est fortement dépendante de la technologie à déployer. En effet, les API de déploiement des différents modèles de composants sont différentes, et donc la manière de déployer des applications dépend de la technologie choisie. Néanmoins, les travaux de recherche actuels sur le déploiement se concentrent quasi-exclusivement sur le déploiement des architectures à composants, donc des applicatifs métiers. Cependant, la procédure de déploiement d'un système logiciel dans son ensemble démarre avant le déploiement des composants. En effet, pour pouvoir déployer et exécuter des composants, il est nécessaire de disposer de serveurs de composants sur les différentes machines visées par le déploiement. Ces serveurs de composants peuvent eux-même nécessiter le déploiement en amont d'un certain nombre de bibliothèques pour fonctionner correctement. Ainsi la procédure de déploiement d'un système logiciel complet —*i.e.* d'une architecture à base de composants et de l'ensemble de la couche intergicielle nécessaire— s'avère une tâche d'une complexité encore plus grande.

À ce jour, quelques grandes questions se posent quant à cette procédure de déploiement. La première de ces questions concerne le mode de description du déploiement d'un système logiciel complet. À l'heure actuelle, la quasi-totalité des actions à mettre en œuvre pour déployer les serveurs d'applications et autres bibliothèques restent manuelles. Il faut alors se connecter sur les machines distantes, envoyer des commandes pour transférer des fichiers, lancer des exécutables ou encore fixer des variables d'environnement. Certains travaux proposent des langages pour décrire le déploiement d'architectures à base de composant. Toutefois les questions suivantes n'ont pas encore de réponses satisfaisantes. Comment unifier et rationnaliser la description de l'intégralité d'un déploiement logiciel ? Quel langage utiliser alors pour décrire le déploiement de l'ensemble d'un système logiciel quelles que soient les technologies impliquées et les actions élémentaires de déploiement à mettre en place ?

La seconde question qui se pose concerne la vérification des programmes écrits dans un tel langage de déploiement. Le déploiement d'un système logiciel est un processus qui s'exécute sur plusieurs machines d'un réseau. Il est donc essentiel de s'assurer que le déploiement décrit est bien cohérent afin qu'il ne corrompe pas les machines hôtes qu'il concerne. Cette nécessité de vérification devient de plus en plus importante à mesure que la taille du système à déployer augmente, comme dans les grilles de calcul ou les environnements ubiquitaires. Ainsi, comment vérifier de manière statique qu'un programme de déploiement va se faire de manière correcte —*i.e.* sans erreurs ?

[2] `http://cd05.dcs.st-and.ac.uk`
[3] `http://decor.imag.fr/`

Les réseaux de machines sur lesquels déployer des systèmes logiciels tendent à évoluer. De nouveaux environnements émergent, comme les réseaux ubiquitaires, dans lesquels des terminaux apparaissent et disparaissent dynamiquement au fil du temps. Il y a également les environnements de type grille de calcul dans lesquels des ressources supplémentaires peuvent être réservées dynamiquement, et des réservations peuvent aussi s'achever, entrainant ainsi une disparition de ressources. Le déploiement de systèmes distribués dans ce genre d'environnements, que nous appelons *environnements ouverts distribués*, ne peut pas se faire de la même manière que sur un réseau statique : le déploiement doit s'adapter aux fluctuations de l'environnement. Une tendance actuelle consiste à utiliser les terminaux mobiles pour fournir des informations contextuelles à l'utilisateur. Ainsi, lorsque ce dernier entre dans une zone de couverture, on souhaite qu'il dispose d'un logiciel particulier lui proposant des services contextuels ; ces logiciels doivent alors être déployés de manière automatique sur les machines. Dans un hypothétique langage de déploiement pour les systèmes distribués, quelles constructions peuvent permettre de prendre en compte la dynamicité des environnements ouverts distribués afin de déployer dynamiquement des éléments du système logiciel sur des machines entrant sur le réseau ?

Une fois les descripteurs de déploiement établis dans le langage sus-cité, il est nécessaire de disposer d'une plate-forme capable d'exécuter automatiquement le déploiement du système décrit. Cette plate-forme doit alors être capable de supporter le déploiement des logiciels, indépendamment de toute technologie et de tout paradigme. Ce support d'exécution doit être capable de déployer automatiquement la couche intergicielle, mais également de déployer l'architecture de composants, quel que soit le modèle choisi. Cette plate-forme doit pouvoir automatiser l'envoi des commandes conformément aux caractéristiques de la machine qui héberge chaque logiciel, l'orchestration du déploiement du système dans sa globalité, et enfin le déploiement de l'architecture à composants. Comment réaliser une plate-forme capable d'automatiser le déploiement d'un système logiciel de manière extensible et générique ?

1.3 Contribution

Dans cette thèse nous présentons notre approche DeployWare/DACAR qui répond à ces questions. Dans un premier temps nous identifions quatre rôles intervenant dans une procédure de déploiement. Ces quatre rôles sont pourvus d'un ensemble d'outils pour mener à bien leurs tâches.

Expert réseau Sa mission consiste à proposer une interface uniforme d'accès aux machines hôtes de l'environnement, quelles que soient leurs moyens d'accès en terme de protocoles, de format de commandes, de transfert de fichiers, etc.

Expert logiciel Sa mission consiste à définir en terme d'actions élémentaires génériques le processus de déploiement d'un logiciel de la couche intergicielle dont il maîtrise l'utilisation. Il définit également des propriétés personnalisables pour ces logiciels. Sa description permet à elle seule de comprendre comment on déploie un logiciel, et quelles

sont ses dépendances.

Administrateur système Sa mission consiste à concevoir l'architecture de la couche intergicielle du système logiciel. Il réutilise et instancie les descriptions de l'expert réseau et de l'expert logiciel pour définir les machines hôtes de l'environnement, et les différents logiciels à déployer sur ces machines. Il est également en charge de prévoir le déploiement en cas d'entrées ou de sorties de machines sur le réseau.

Architecte métier Sa mission consiste à concevoir l'architecture de composants métiers et à prévoir le déploiement en cas d'entrées ou de sorties de machines sur le réseau.

Notre approche se compose de deux grandes parties. D'abord DeployWare est une approche mettant en œuvre l'ingénierie dirigée par les modèles pour le déploiement de la couche intergicielle des systèmes distribués. Un premier méta-modèle comprend donc deux paquetages contenant les concepts respectivement utiles pour l'expert logiciel et l'administrateur système. Le premier décrit, à l'aide des concepts idoines, le protocole de déploiement des logiciels dont il a l'expertise. Le second décrit l'assemblage des différents logiciels composant la couche intergicielle du système. Il fixe également les propriétés des différents logiciels. Le caractère dédié des concepts du méta-modèle DeployWare fait qu'il est possible d'effectuer des vérifications statiques sur le comportement global du processus de déploiement. Ces vérifications permettent entre autres de vérifier des propriétés de réversibilité, de complétude et de cohérence sur le déploiement décrit.

Une machine virtuelle a été conçue pour exécuter les déploiements décrits à l'aide du méta-modèle DeployWare. Cette machine virtuelle repose sur le modèle de composants Fractal. Elle fournit un cadre générique et extensible pour concevoir les protocoles de déploiement des différents logiciels, ainsi que l'assemblage des différentes instances de logiciel pour former le processus de déploiement global de l'ensemble du système logiciel. Elle fournit aussi des concepts provenant de l'informatique auto-gérée [39] afin de définir des politiques d'auto-gestion qui prennent en charge le déploiement dynamique de nouveaux logiciels sur les machines entrant sur le réseau Cette machine virtuelle propose également un cadre de conception à l'expert réseau afin de développer des composants *adaptateurs* pour créer des moyens de communication unifiés avec les machines hôtes de l'environnement.

La seconde partie de notre contribution, DACAR, s'adresse exclusivement à l'architecte métier. Il s'agit là aussi d'un environnement de modélisation pour concevoir des architectures de composants génériques. Le méta-modèle de composants génériques choisi repose sur la spécification de l'OMG *Deployment and Configuration* (OMG D&C) [52]. Il est également possible de définir des politiques d'auto-gestion sur ces architectures à composants, afin qu'elles s'adaptent aux fluctuations éventuelles de l'environnement conformément aux principes de l'informatique auto-gérée.

Notre approche combinée DeployWare/DACAR a été validée en l'appliquant à deux problèmes réels de déploiement de systèmes distribués issus des projets RNTL MOSAIQUES [72] et CAPPUCINO (en cours).

1.4 Plan du document

Le plan du document se compose de la manière suivante : le chapitre 1 dresse, après avoir introduit les notions nécessaires, un tour d'horizon des différentes technologies actuelles de déploiement et de reconfiguration dynamique. À partir de cette étude de l'existant, nous identifions les lacunes à combler dans ces domaines. Le chapitre 2 présente le méta-modèle DeployWare ainsi que les différentes vérifications possibles sur cette structure de données, pour la modélisation du déploiement des couches intergicielles d'un système. Le chapitre 3 détaille la machine virtuelle DeployWare capable d'exécuter les projections des modèles DeployWare. Le chapitre 4 introduit l'approche DACAR pour le déploiement auto-géré d'architectures à composants. Enfin le chapitre 5 présente les exemples utilisés pour valider la combinaison des approches DeployWare/DACAR. Viennent enfin la conclusion et les perspectives de ce travail.

Deuxième partie

Étude de l'existant

Chapitre 2

État de l'art

Sommaire

Ce chapitre dresse un tour d'horizon de différents travaux et approches traitant du déploiement logiciel. Ces approches sont ensuite comparées en fonction des critères pertinents pour notre étude. Le choix de ces critères tend à mettre en évidence certaines lacunes dans le déploiement logiciel réparti de systèmes hétérogènes, ainsi que dans la reconfiguration dynamique de ces logiciels. Nous identifions précisément ces lacunes dans une synthèse à la fin de cet état de l'art. Ce chapitre est construit comme suit. Tout d'abord, nous donnons quelques définitions liées aux domaines étudiés dans cet état de l'art, à savoir le déploiement logiciel et la reconfiguration dynamique des logiciels. Puis, nous donnons la liste exhaustive des critères servant à l'évaluation des travaux. Enfin, nous détaillons chacune des approches liées aux domaines que nous étudions dans cette thèse. Ces approches sont regroupées en quatre ensembles : les langages d'architecture, les modèles dédiés au déploiement logiciel, les plates-formes intergicielles de déploiement logiciel et enfin les approches se focalisant sur la reconfiguration dynamique. Nous achevons ce chapitre sur une conclusion quant aux points à traiter ou lacunes dans les domaines étudiés.

2.1 Définitions et contexte

2.1.1 Cycle de vie du logiciel

Le cycle de vie du logiciel regroupe toutes les étapes de l'existence d'un logiciel, de son apparition à sa disparition. Ce cycle de vie, tel que défini dans [11], se compose de plusieurs étapes afin de séparer les rôles impliqués dans la construction de logiciels. La définition d'un découpage en étapes permet d'isoler les étapes de construction du logiciel, afin de *vérifier* et *valider* chacune des étapes indépendamment les unes des autres, et d'assurer une certaine qualité du logiciel tout au long du cycle de vie. En effet, plus une erreur est détectée tôt dans le cycle de vie d'un logiciel, plus la correction est aisée à mettre en œuvre. À l'inverse si une erreur dans les premières étapes du cycle de vie est détectée en fin de cycle, le coût de mise en œuvre de correction du logiciel devient plus conséquent, puisque la modification du résultat de l'une des étapes du cycle de vie impacte toutes les étapes en aval de celle-ci.

Il existe différents modèles de cycle de vie du logiciel. Parmi ceux-ci, on peut citer le modèle en cascade, communément accepté et représenté sur la figure 2.1, mais il existe également le modèle dit en V, ou encore le modèle en spirale de Boehm [11].

Tout d'abord, l'étape d'**analyse des besoins** correspond à la phase d'étude des logiciels existants et des nouveaux besoins du *client*. Le terme de client désigne une personne ou un organisme qui fait appel aux services de concepteurs en vue d'acquérir un logiciel répondant

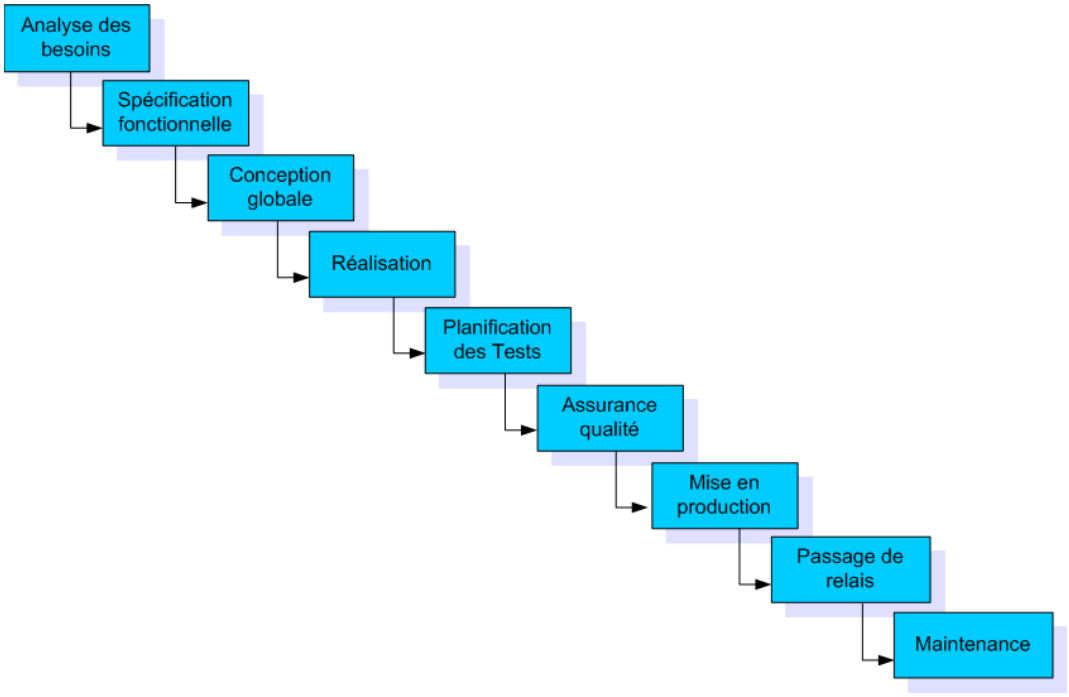

FIG. 2.1 – Modèle de cycle de vie du logiciel en cascade

à l'un de ses besoins. Puis arrivent les phases de **spécification fonctionnelle** et de **conception globale** du logiciel. La première de ces deux phases correspond à l'établissement d'un cahier des charges décrivant les principales fonctionnalités du logiciel, ainsi que l'interface homme-machine (IHM) du logiciel final. La conception globale consiste en la définition de l'architecture globale du logiciel. Chaque fonctionnalité du logiciel est contenue dans un module et l'interaction entre les différents modules est établie. Cette phase correspond à l'établissement de *l'architecture logicielle*. Vient ensuite la phase de **réalisation**, ou *implémentation* de chaque module de l'application. Enfin, après les **tests** (unitaires, d'intégration ou de qualification), viennent les phases d'**Assurance qualité** que l'on nomme aussi Vérification et Validation (V&V). La validation est la phase qui consiste à prouver que le logiciel réalisé accomplit bien l'action que l'on attend de lui, tandis que la vérification consiste à prouver que le logiciel a été fait correctement —*i.e.* qu'il ne produit pas d'erreur non prévue. Les dernières phases de ce cycle de vie sont l'**Assurance qualité**, qui consiste à faire en sorte que le logiciel passe tous les tests avec succès, le **Passage de relais** qui consiste à documenter le logiciel et transférer les compétences d'exploitation au client, et enfin la phase sur laquelle porte notre étude, la phase de **Mise en production**. Cette phase regroupe la totalité des opérations qui visent à mettre le logiciel en état de fonctionnement, à le mettre à jour vers de nouvelles versions, ou à adapter son fonctionnement suite à des événements externes au logiciel (dûs à son contexte). Cette phase porte également le nom de **Déploiement logiciel**, et c'est cette phase du cycle de vie du logiciel que nous étudions dans cette thèse.

Le déploiement regroupe l'ensemble des opérations nécessaires à la mise en fonctionnement d'un logiciel ou de ces sous-parties sur le *domaine de déploiement* —*i.e.* l'ensemble de machines sur lesquelles le logiciel doit être installé. Il existe plusieurs raisons de s'intéresser au processus de déploiement logiciel. La première de ces raisons est qu'il s'agit sans doute d'une des phases du cycle de vie du logiciel pour laquelle peu de standards existent, nous le verrons au travers de notre étude, et les standards existants restent souvent à très haut niveau. La seconde raison est que cette procédure de déploiement est l'une des plus techniques mais paradoxalement aussi l'une des moins outillées. L'étape de développement logiciel, qui est l'autre étape très technique de ce cycle de vie, est aujourd'hui très largement outillée. Des environnements integrés de développement existent, comme Eclipse[1] ou NetBeans[2] pour toutes les technologies relatives au langage Java (même si Eclipse propose aussi des outils de développement pour d'autres langages de programmation), comme Microsoft Visual Studio pour la technologie .NET[3]. En revanche, il n'existe aucun outil communément accepté pour déployer des systèmes logiciels généralistes. Les *administrateurs systèmes* —*i.e.* les acteurs du déploiement— effectuent d'ailleurs aujourd'hui encore bien souvent leurs déploiements manuellement ou à l'aide de scripts shell, c'est-à-dire d'un ensemble de commandes de très bas niveau exprimées dans le langage d'interpréteur de commandes de la machine. Cette solution est très éloignée de la puissance des outils existants actuellement pour prendre en charge les autres étapes du cycle de vie du logiciel, comme la conception avec UML [53], le développement ou encore les tests.

2.1.2 Cycle de vie du déploiement

En 1999, R. Hall [37] présente un cycle de vie du déploiement qui comprend différentes étapes, représentées sur la figure 2.2. Ces différentes étapes sont séparées en deux rôles : le rôle de *producteur* et celui de *consommateur* de logiciel. Le premier de ces rôles consiste à conditionner un logiciel afin d'en fournir une version livrable, et le second consiste à traiter cette version livrable pour la rendre exécutable pour l'utilisateur.

L'**installation** est la première étape de ce cycle. Ce processus est en charge de transférer les *paquets* du logiciel, c'est-à-dire les unités élémentaires de déploiement pour un logiciel, sur les machines concernées. Ces unités sont fournies par le producteur de logiciel et contiennent toutes les ressources (bibliothèques, exécutables, fichiers de configuration, etc.) du logiciel, ainsi que d'éventuels descripteurs spécifiant les ressources requises pour le déploiement du logiciel.

La **configuration** est l'étape qui suit directement l'installation. Durant cette étape, les paramètres du logiciel installé sont fixés en fonction des propriétés de la **machine hôte**, —*i.e.* la machine qui héberge ce logiciel. Par exemple, il s'agit de fixer le chemin vers un fichier dans le système de fichiers. Une fois cette étape achevée, le logiciel est prêt à fonctionner.

[1] http ://www.eclipse.org
[2] http ://www.netbeans.org/
[3] http ://msdn2.microsoft.com/fr-fr/netframework/default.aspx

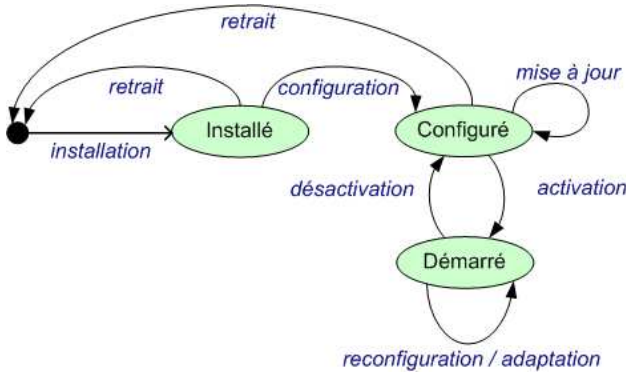

FIG. 2.2 – Cycle de vie du déploiement

L'**activation** (ou **démarrage**) est l'étape qui consiste à exécuter le logiciel installé et configuré proprement. Une fois cette étape achevée, le logiciel est en état de fonctionnement. Le logiciel ne changera d'état qu'après l'étape de **désactivation** (ou **arrêt**). Cette étape est symétrique au démarrage, —*i.e.* elle replace le logiciel dans son état après la configuration.

Ensuite la procédure de **mise à jour** peut être mise en œuvre. Cette étape consiste à faire migrer le logiciel configuré vers une version supérieure. Cette opération nécessite souvent l'arrêt préalable du logiciel, et est effectuée lors de changements réalisés par le producteur du logiciel.

Le logiciel peut également être soumis à des **reconfigurations**. Il s'agit là encore d'une modification du logiciel installé. Mais, contrairement à la mise à jour, la reconfiguration consiste non pas à remplacer le logiciel par une version supérieure, mais à modifier la configuration de la version courante. Qui plus est, ce changement doit pouvoir être opéré sans arrêt du logiciel.

Enfin un logiciel à l'exécution peut être sujet à **adaptation**. L'adaptation est une forme particulière de reconfiguration, guidée par des changements extérieurs. Ces changements peuvent concerner la machine hôte sur laquelle le logiciel est installé (système de fichiers, ressources système, etc.) ou le logiciel lui-même. L'adaptation est un processus visant à maintenir la cohérence et l'intégrité du logiciel dans le cas de changements éventuels.

Enfin, après l'arrêt du logiciel, la **désinstallation** du logiciel (ou **retrait**) peut être appelée. Il s'agit de l'étape symétrique de l'installation, visant à supprimer entièrement le logiciel de la machine hôte.

2.1.3 Paramètres du déploiement

Ces différentes étapes du cycle de vie du déploiement ne peuvent raisonnablement pas être appliquées de manière homogène. En effet, différents paramètres entrent en ligne de compte lors de l'exécution de ces tâches. Ces paramètres concernent le logiciel lui-même, mais également la machine hôte sur laquelle celui-ci est déployé.

Tout d'abord, il est évident que le processus de déploiement de n'importe quel logiciel dépend des caractéristiques de la machine hôte. La première caractéristique concerne les capacités matérielles de la machine. Ainsi la puissance de calcul, la capacité de stockage ou encore la connectivité de la machine hôte —*i.e.* les moyens d'accès au réseau de cette machine— sont des paramètres qui peuvent influer sur la réalisation du déploiement logiciel. En effet, un type de logiciel plus ou moins coûteux en termes de ressources peut ne pas être déployable sur une machine trop peu puissante. De la même manière, le format des commandes à envoyer sur les machines pour déployer un logiciel dépendent du protocole d'accès fourni par la machine concernée. D'autres paramètres à prendre en compte concernent la première couche logicielle de toute machine, c'est-à-dire le système d'exploitation. En effet, cette couche, ayant pour but de fournir un accès logiciel aux ressources de calcul de la machine, peut posséder différentes caractéristiques. Tout d'abord le protocole d'accès à distance, indispensable pour agir sur les machines, varie d'une machine à l'autre. Il existe différents protocoles pour l'accès à distance aux machines, parmi ceux-ci on trouve le *Secure SHell (SSH)*, le *TErminal NETwork (Telnet)*. Le protocole de transfert de fichiers peut lui aussi varier, il existe le *Secure CoPy (SCP)* ou encore le *File Transfer Protocol (FTP)*. Enfin, le dernier point de variation dans l'accès au système d'exploitation des machines est le langage d'interpréteur de commandes (ou shell). Là encore, il existe de nombreux langages différents comme le Bourne Shell, le C-Shell, ou encore le shell de Windows. La réalisation des étapes du cycle de vie du déploiement dépend de ces différents paramètres.

En outre, le contenu des étapes du cycle de vie, en termes d'actions élémentaires, varie également en fonction de la nature du logiciel. C'est là l'une des difficultés les plus grandes du déploiement logiciel à l'heure actuelle. Tout d'abord, il existe différentes technologies pour concevoir des logiciels. Ainsi, les opérations élémentaires de déploiement d'un logiciel peuvent dépendre du langage de programmation dans lequel le logiciel est réalisé. Ces opérations peuvent également dépendre du paradigme de programmation utilisé (programmation par objets, par aspects [40], par composants [69]). En outre, pour chacun de ces paradigmes, il existe une multitude d'implémentations. À titre d'exemple, on peut citer les modèles de composants Entreprise Java Beans (EJB) [68], CORBA Component Model (CCM) [51] ou encore Fractal [18]. Si ces trois modèles implémentent tous les concepts généraux des composants, ils sont orientés pour une utilisation particulière. Le modèle EJB est fortement orienté vers la manipulation de bases de données, alors que le modèle Fractal est fréquemment utilisé pour la construction d'intergiciels comme en attestent les travaux [58] et [60]. Le modèle CCM est, quant à lui, utilisé pour construire des applications à base de composants distribués hétérogènes en termes de langages de programmation. Chacune de ces technologies possède ses propres concepts et son propre processus de déploiement.

De plus, les logiciels possèdent également une granularité différente. En effet, différentes entités aux rôles différents interviennent dans le déploiement de logiciels. Certains logiciels sont des bibliothèques de fonctions utilisées par les applications. D'autres sont des serveurs d'applications qui hébergent des applications. Enfin, les applications métiers elles-mêmes sont des entités. L'ensemble de ces entités de granularité différente constituent ce que nous appe-

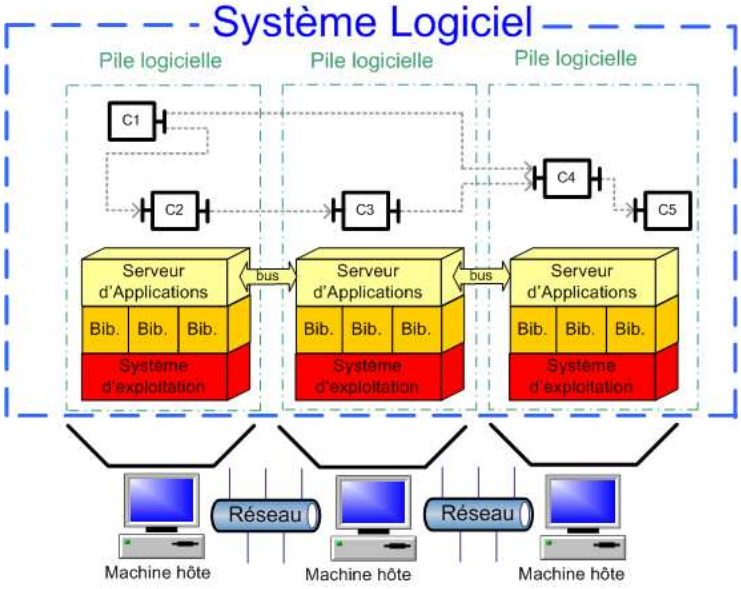

FIG. 2.3 – Représentation schématique d'un système logiciel

lons un *système logiciel*, et l'ensemble des logiciels d'un système déployés sur une machine est appelé la *pile logicielle* . Cette notion de système logiciel est représentée sur la figure 2.3. Ainsi selon la granularité du logiciel que l'on déploie, les actions élémentaires de déploiement sont différentes.

2.1.4 Description du déploiement

Les étapes du cycle de vie du déploiement varient selon les conditions matérielles —*i.e.* les paramètres de la machine hôte— ainsi que selon les paramètres du logiciel lui-même. Ainsi il faut posséder un moyen d'exprimer les différentes configurations et autres actions élémentaires à opérer. Par *langage de déploiement* nous entendons donc un langage dédié à la description du déploiement d'un système, qui permet d'exprimer une configuration de système logiciel. Ces langages doivent permettre de décrire à haut niveau les propriétés d'un logiciel, les liaisons qu'il possède avec d'autres logiciels, ainsi que les actions à effectuer pour déployer ce logiciel sur une machine. De ces descriptions à haut niveau, il est possible de déduire les opérations à plus bas niveau à opérer concrètement sur les machines hôtes afin de rendre le système logiciel opérationnel. Cette abstraction doit permettre de décrire toutes les entités du système, quelle que soit leur granularité, mais aussi les machines hôtes ainsi que leurs caractéristiques. Il existe une multitude de langages discutés dans la suite du chapitre.

2.1.5 Vérification du déploiement

Dès lors qu'un langage de déploiement est employé pour décrire le déploiement de systèmes logiciels, la question de la vérification des descripteurs se pose. En effet, il faut pouvoir vérifier que la description abstraite du déploiement est correcte. En premier lieu, une **vérification syntaxique** est nécessaire afin de vérifier que les concepts utilisés dans le descripteur correspondent bien à des concepts définis dans le langage de déploiement et que leur emploi est valide. Ensuite, une **vérification de typage** peut être effectuée afin d'assurer que les types des concepts du langage ont été employés conformément à leur définition. Enfin, nous appelons le troisième type de vérification des **vérifications sémantiques** (ou **comportementales**) spécifiques au processus de déploiement logiciel. Cette vérification doit survenir après les phases de vérification syntaxique et de typage. Elle consiste à assurer que le processus de déploiement décrit possède bien le comportement que l'on attend, sans effet de bord, ni erreurs survenant lors de l'exécution du processus de déploiement. Cette procédure vise à étudier, non plus la justesse de ce qui est écrit dans le descripteur, mais la cohérence des intentions exprimées dans ce descripteur.
À titre d'exemple, le travail d'un compilateur Java est de vérifier la syntaxe et l'usage des types dans un programme Java. Néanmoins, il existe un exemple de vérification sémantique dans les compilateurs Java : lorsqu'une instruction suit directement une instruction de type *return*, le compilateur relève une erreur. Le programme est syntaxiquement correct, il n'y a pas de problème de typage, mais il existe une incohérence dans l'intention du développeur, en l'occurrence une instruction qu'il a écrite ne pourra jamais être exécutée (à cause du *return* qui la précède directement).

Nous différencions trois types de vérifications différentes, dans le moment et la manière avec lesquelles elles sont notifiées. La vérification **statique** avertit l'usager d'éventuelles erreurs avant de commencer la procédure de déploiement. La vérification **dynamique directe** prévient l'usager à l'exécution d'une erreur précise rencontrée lors du déploiement. Enfin, la vérification **dynamique indirecte** prévient l'usager d'une erreur rencontrée lors du déploiement, mais le message d'erreur notifié à l'usager ne concerne pas directement le problème dans son déploiement, mais plutôt la conséquence de l'erreur faite par l'usager (par exemple l'apparition d'une exception de type *pointeur nul* à cause d'un serveur non démarré).

2.1.6 Reconfigurabilité

Le descripteur écrit dans un langage de déploiement est ensuite exécuté. Il en résulte un système entièrement déployé sur plusieurs machines hôtes. Néanmoins, nous l'avons vu dans le cycle de vie du déploiement, le système logiciel est sujet à d'éventuelles reconfigurations ou adaptations, avant son arrêt. Dès lors, le système doit connaître les informations qui lui permettront de savoir quand et comment il doit être adapté ou reconfiguré. En outre, le système logiciel doit avoir accès à des mécanismes de reconfiguration dynamique, c'est-à-dire capable d'analyser son environnement et d'exécuter des reconfigurations en fonction de cet environnement, le tout sans altérer son propre état.

2.1.7 Environnements ouverts distribués

Ce besoin de mécanisme d'observation et de reconfiguration est amplifié par l'émergence des *environnements ouverts distribués* évoqués dans l'introduction. En effet dans de tels environnements, de nouvelles machines hôtes sont susceptibles d'apparaître ou de disparaître du domaine de déploiement. Ainsi le système logiciel doit être conscient de cet environnement et va devoir adapter son architecture en réponse à ces changements. Il existe à ce jour principalement deux types d'environnements ouverts distribués. D'abord, les environnement ubiquitaires, dans lesquels évoluent des terminaux mobiles, sont très instables par définition. Des terminaux mobiles joignent et quittent le domaine très fréquemment. Puis, les grilles de calcul sont également des environnements ouverts distribués. En effet, dans de tels environnements, le mécanisme de réservation fait qu'il est possible de faire entrer dans l'environnement de nouvelles machines. Inversement, l'expiration de ces réservations aboutit à la disparition de ressources dynamiquement.

2.1.8 Critères d'étude de l'existant

La définition du contexte de ce travail, ainsi que l'éclaircissement de certains enjeux du déploiement, nous conduisent à établir un certain nombre de critères d'évaluation. Ces critères guideront notre étude de l'existant dans le domaine du déploiement logiciel. Ils nous permettent de mettre en exergue un certain nombre de lacunes dans le processus de déploiement logiciel tel qu'il est réalisé à ce jour.

Couverture du cycle de vie Nous avons identifié un certain nombre d'étapes dans le cycle de vie du déploiement. Le premier critère qui nous paraît pertinent d'observer est la couverture de ce cycle de vie pour les approches actuelles du déploiement. Cette observation peut nous permettre d'identifier le centre d'intérêt des approches, qu'elles soient davantage orientées installation et configuration, ou reconfiguration et dynamicité.

Abstraction des concepts de déploiement Nous souhaitons également observer, pour chacune des approches étudiées, l'abstraction utilisée pour décrire le processus de déploiement (ou, le cas échéant, un sous-ensemble de ce processus). Il est ainsi possible de confronter les différents concepts mis en avant dans ces approches pour décrire le déploiement logiciel. La couverture d'expression de ces concepts sera également inspectée, c'est-à-dire quelles sont les entités impliquées dans le déploiement (logiciel, paquets, machines hôtes, producteurs, consommateurs, etc.) qui sont présentes dans les langages ou abstractions étudiés.

Vérification Un troisième critère de comparaison de l'existant repose sur les capacités de vérification. Il est en effet intéressant d'étudier les besoins de chaque approche en terme de vérification, ainsi que de connaître les limites actuelles dans le domaine du déploiement logiciel. Cela signifie étudier ce qui est vérifiable dans les approches de déploiement actuelles, et ce qui ne l'est pas. En outre, il conviendra de faire la différence, le cas échéant, entre les différents types de vérifications fournies par les approches étudiées.

Automatisation du déploiement Les tâches de déploiement sont très techniques, mais sont souvent répétitives. En fait, les intentions d'un consommateur vis-à-vis du déploiement

d'un logiciel sont souvent les mêmes. Ainsi le déploiement d'un même logiciel sur différentes machines peut devenir redondant. Dès lors, il est intéressant de se pencher sur les automatisations mises en œuvre dans les approches existantes, afin de mesurer la réutilisabilité des descriptions de déploiement pour un logiciel donné.

Reconfiguration et Adaptation Pour les travaux qui implémentent cette étape du cycle de vie, il est intéressant de se pencher sur la manière dont ces mécanismes de reconfiguration dynamique et/ou d'adaptabilité sont implémentés et selon quels paradigmes. Il s'agit là d'un vaste critère, car la reconfiguration dynamique des applications est un thème de recherche à part entière.

Environnements ouverts distribués Ces environnements représentent un contexte particulier pour la reconfiguration dynamique. Ainsi il est intéressant d'étudier les mécanismes particuliers de reconfiguration mis en œuvre dans les approches actuelles visant la gestion de la dynamicité et de l'ouverture du domaine de déploiement destiné à héberger un système logiciel.

Maîtrise de l'hétérogénéité Ce dernier critère est l'un des plus déterminants de l'étude. En effet, l'hétérogénéité connait plusieurs formes au sein du processus de déploiement. L'hétérogénéité est un problème pour le déploiement car elle concerne aussi bien les machines hôtes que les logiciels. Les machines hôtes sont concernées par l'hétérogénéité des ressources système, des protocoles d'accès ou de transfert de fichiers, ou encore du langage de shell. Les logiciels eux sont concernés par l'hétérogénéité des langages et paradigmes de programmation, des implémentations choisies, ou encore de la granularité et donc de leur place dans la pile logicielle. Tous ces facteurs influent sur le processus de déploiement. Ainsi il est intéressant de se pencher sur les efforts apportés par les travaux actuels sur la gestion de cette hétérogénéité omniprésente qui représente à ce jour l'obstacle majeur au déploiement. Cette gestion impacte l'abstraction choisie pour exprimer le déploiement, mais aussi la vérification ou encore les mécanismes de reconfiguration. Il s'agit donc d'un critère transverse à tous ceux exposés précédemment.

Nous confrontons différentes approches aux critères mis en place dans cette section. Nous allons étudier quatre grandes catégories d'approches de déploiement. Dans un premier temps, nous allons nous pencher sur les *langages de definition d'architecture* (ou ADL pour *Architecture Definition Language*). Dans un second temps, nous étudions les travaux qui définissent des modèles pour le déploiement logiciel. Notre attention porte ensuite sur les plates-formes qui existent aujourd'hui pour déployer des logiciels, avant de nous focaliser sur les paradigmes et mécanismes de reconfiguration dynamique et d'adaptation existants.

2.2 Langages de description d'architectures

Depuis une dizaine d'années, la tendance en génie logiciel consiste à détourner l'attention des développeurs du code source des programmes, pour l'orienter vers des entités logicielles de plus grosse granularité. L'une des visions promue dans cette optique est l'idée de développement orienté *architecture* qui nous intéresse dans cette section. L'autre vision principale repose sur l'ingénierie dirigée par les modèles. Cette première vision propose de séparer le développement de logiciels en plusieurs modules interconnectés les uns aux autres, ce qui fait apparaître

clairement le concept d'architecture logicielle. Dès lors, cette pratique nécessite un paradigme de programmation plus évolué que celui de programmation par objets. En effet les interactions entre objets sont souvent cachées dans le code des objets, et la notion d'architecture d'objets est beaucoup plus difficile à dégager d'un programme.

C'est dans ce contexte que la notion de *composant logiciel* a emergé, la première occurrence du composant fut présentée lors d'un congrès de l'OTAN en 1968 [48]. Avec l'ouvrage de référence de Clemens Szyperski [69], le composant devient un concept de première classe dans les architectures. Dès lors que le développement de logiciels se fait en construisant des architectures de composants, il nécessite un langage de construction d'architectures. Ce langage doit permettre au développeur d'avoir une vision plus abstraite de son système et de spécifier les propriétés de ses composants et des interconnexions entre ces composants. Néanmoins, peu de consensus existe autour de ces paradigmes de construction d'architecture, appelés *Langages de Description d'Architectures* (ou ADL pour *Architecture Definition Language*) [49].

Ainsi il existe une multitude d'ADL différents dont certains sont génériques, d'autres spécifiques à un domaine donné. Parmi cette multitude d'ADL, on peut citer Darwin [47], Wright [3], AADL [5], UniCon [74], ACME [35], etc. Certains sont orientés conception générale —*i.e.* l'étape du cycle de vie logiciel— de haut niveau à des fins de documentation et de compréhension des architectures, alors que d'autres sont des langages formels dont le but est d'éprouver la cohérence et la fiabilité de l'architecture décrite. Ces langages sont intéressants à étudier dans le contexte du déploiement, car la description d'une architecture logicielle impacte directement le déploiement de l'application, voire le définit complètement. En effet, dans une application répartie à base de composants, les différents composants d'une application sont destinés à être déployés sur différentes machines. Dans ce cas, les ADL jouent le rôle de langage de déploiement. Dans la suite, nous décidons de découper les langages de description d'architectures en trois parties. La première partie regroupe les ADL orientés déploiement, c'est-à-dire les langages servant à décrire des instances de composants et à configurer et instancier ces composants éventuellement de manière répartie. La seconde partie regroupe les ADL orientés dynamicité, c'est-à-dire des ADLs introduisant la dynamicité des architectures durant l'exécution. Enfin, la troisième partie regroupe les ADL orientés vérifications, c'est-à-dire des ADL rendant possible le calcul de vérification sur la cohérence des architectures. Ainsi nous étudions trois ADL représentant chacune des trois grandes catégories, pour se forger une opinion objective sur cette approche d'un point de vue déploiement d'applications à base de composants.

2.2.1 Darwin

Le travail sur l'ADL Darwin provient de l'Imperial College de Londres [47]. Darwin est un ADL dédié à la construction de systèmes distribués à l'aide des notions de composants et de services fournis et/ou requis par ces composants. La communication entre ces différents composants se fait de manière évenementielle. Dans Darwin, les systèmes distribués sont représentés à l'aide de composants hiérarchiques pouvant contenir soit une implémentation, soit un ensemble de composants. L'accent dans Darwin est mis aussi bien sur l'administration de la structure statique des systèmes distribués que sur l'évolution de cette structure au fil de l'exé-

cution du système. Sur la figure 2.4 est représenté un exemple d'architecture Darwin, et sa description en langage d'architecture sur le listing 2.1.

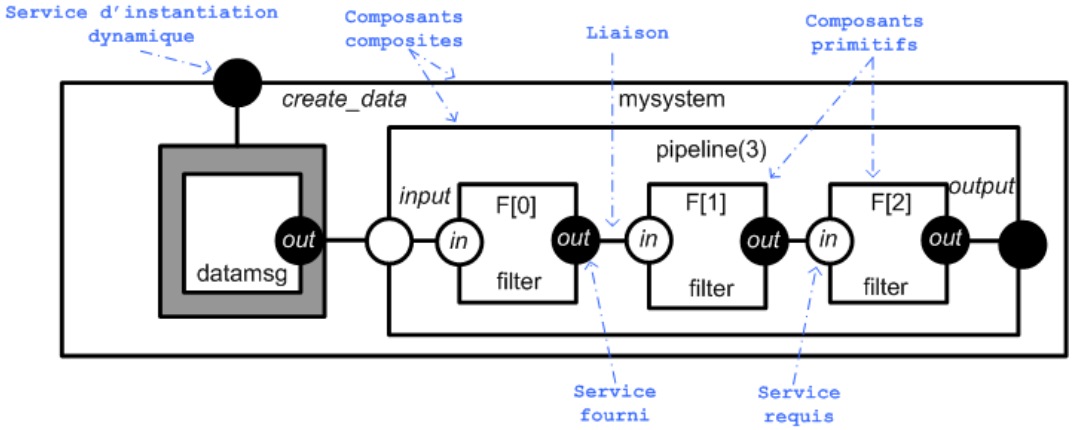

FIG. 2.4 – Architecture dynamique en Darwin

Darwin propose, en plus de la description d'architectures statiques, des mécanismes pour exprimer la dynamicité d'architecture. Le premier de ces mécanismes est l'instanciation paresseuse permettant d'assurer qu'un composant n'est physiquement créé qu'en cas de besoin. Cela signifie qu'un composant n'est instancié que lorsqu'un de ses services est appelé par un autre composant durant l'exécution.

Il est à noter que Darwin possède dans son langage des constructions itératives et conditionnelles. Ces constructions permettent entre autres de créer des types de composites paramétrés comme sur le listing 2.1 aux lignes 11–22. Le type de composant `pipeline` peut ainsi contenir un nombre paramétrable de composants `filter`.

Le deuxième mécanisme de dynamicité proposé par Darwin est appelé *direct instantiation*. Il s'agit d'un mécanisme permettant d'exprimer dans le descripteur de déploiement un nombre indéterminé d'instances d'un type donné. Il est également possible d'exprimer les liaisons de ces composants instanciés dynamiquement via un service donné. Cette construction est utilisée dans le listing 2.1 entre les lignes 25 et 27. L'interface `create_data` est un service d'instanciation dynamique de composant de type `datamsg` (ligne 26). La ligne 27 spécifie qu'une liaison doit être établie entre le service requis `p.input` et toutes les instances de `datamsg` créées dynamiquement.

```
1 component mysystem {
    component filter {
3     provide out<stream char>;
      require in<stream char>;
5   }

7   component datamsg {
      provide out<stream char>;
```

```
9   }

11  component pipeline(int n) {
      require input;
13    array F[n]: filter;
      forall k:0..n-1 {
15      inst F[k] @ k+1;
        when k < n-1;
17        bind F[k+1].in -- F[k].output;
      }
19    bind
        F[0].in -- input;
21      output -- F[n-1].output;
    }
23
    inst p:pipeline[3];
25  bind
      create_data -- dyn datamsg;
27    p.input -- datamsg.out;
}
```

Listing 2.1 – Extrait de description d'architecture en Darwin

Synthèse Le langage de description d'architecture Darwin possède une couverture de cycle de vie assez limitée. En effet, Darwin ne permet de gérer que l'installation, la configuration et le démarrage de composants logiciels. De plus, les concepts de l'abstraction dans Darwin ne permettent de ne manipuler que des instances de composants et leurs liaisons. Aussi, les vérifications sur le langage Darwin sont uniquement d'ordre syntaxique et de typage des composants. Le principal atout de Darwin repose sur ses apports en terme de dynamicité des architectures. Dans le cadre de la plate-forme Regis [41], les descriptions d'architectures Darwin sont transformables automatiquement en composants exécutables. Le mécanisme de *direct instantiation* permet d'étendre la structure des architectures en fonction d'événements extérieurs (modélisés dans Darwin sous forme de *external management agents*). Ce type de mécanisme est utile dans le cadre d'un déploiement en environnement ouvert distribué. Néanmoins, la modification de l'architecture ne peut se faire que par l'ajout. Ainsi il est impossible de modifier l'architecture en retirant des composants ou en modifiant les attributs des composants.
Enfin la gestion de l'hétérogénéité est inexistante dans Darwin, puisque ce langage ne permet que de manipuler des composants Darwin. Un seul paradigme et une seule technologie (les composants Darwin) sont déployables à l'aide de ce langage.

2.2.2 Fractal ADL

Fractal ADL[4] est un langage de description et de déploiement d'applications à base de composants Fractal [18]. Le modèle de composants Fractal, ainsi que son outillage, a été conjointement initié par la cellule de Recherche & Développement de France Télécom et l'INRIA. Ce modèle est défini par une spécification de la structure et du fonctionnement des composants Fractal [19]. L'implémentation de référence de la spécification Fractal se nomme Julia. Les travaux sur Fractal s'inscrivent dans le cadre du consortium OW2 (anciennement ObjectWeb).

[4] http ://fractal.ow2.org

Fractal a été utilisé entre autres pour la réalisation du canevas de communication Dream[5] [58], du canevas transactionnel GoTM[6] [60], ou encore du bus à services d'entreprise (ESB pour *Enterprise Service Bus*) PEtALS[7]. En outre, Fractal possède une extension appelée Fractal RMI qui permet de faire communiquer des composants en réparti.

Dans cette section, nous ne donnons que la description des composants Fractal selon le point de vue de Fractal ADL, afin de garder notre attention sur l'ADL de Fractal et moins sur le fonctionnement des composants.

Dans le modèle Fractal, un *composant* est une unité logicielle indépendante, configurable via l'utilisation d'*attributs*. Un composant posséde des *interfaces* dites *fournies* qui offrent des opérations, ainsi que des interfaces dites *requises*. Des *liaisons* sont établies entre interfaces requises et interfaces fournies de même type. Un composant peut contenir une implémentation, il est alors dit *primitif*, ou alors être récursivement composé d'autres composants, il est alors dit *composite*. Un composite délègue le traitement des interfaces qu'il expose en utilisant des *interfaces internes*. Un composant peut être contenu par plusieurs composites, lorsque c'est le cas il est dit *partagé*. La figure 2.5 représente un exemple simple d'architecture Fractal.

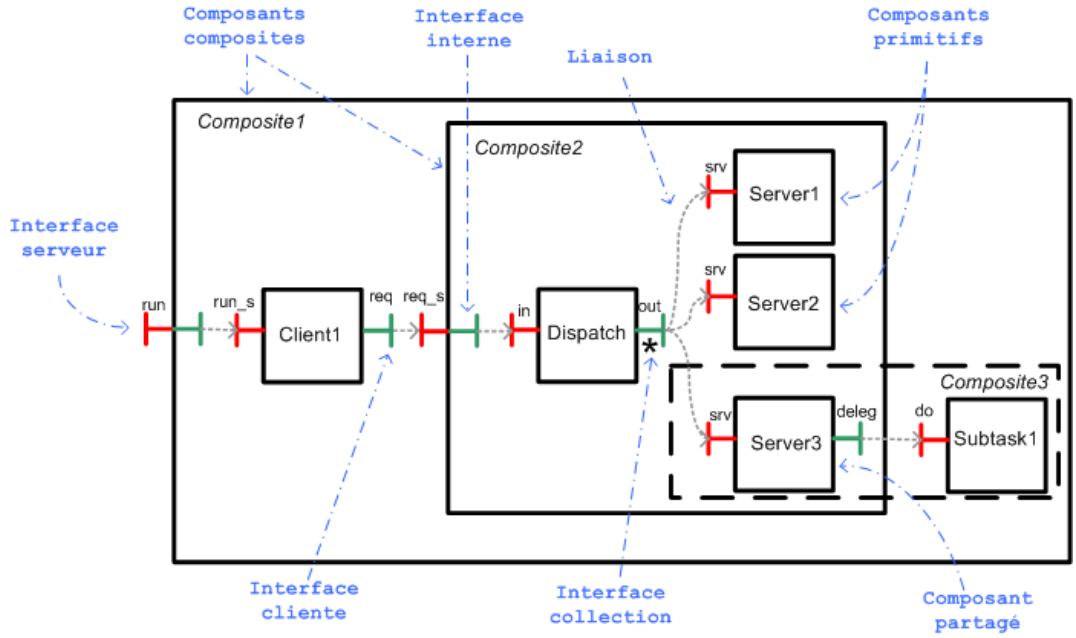

FIG. 2.5 – Modèle de composants Fractal

Fractal ADL est un langage déclaratif avec une syntaxe XML. Il permet de déclarer des composants primitifs, leurs attributs et leur implémentation. Il permet également de définir des

[5] http ://dream.ow2.org
[6] http ://gotm.ow2.org
[7] http ://petals.ow2.org

composants composites en décrivant récursivement leur contenu en terme de composants. Enfin il permet de décrire les liaisons entre interfaces de composants. Le listing 2.2 présente le descripteur de déploiement de l'architecture présentée en figure 2.5.

```
<definition name="Composite1">
  <interface name="run" role="server" signature="java.lang.Runnable" cardinality="
      collection"/>
  <!-- Définition de MyClient dans fichier Fractal ADL MyClient.fractal -->
  <component name="Client1" definition="MyClient"/>
  <component name="Composite2">
    <interface name="req_s" role="server" signature="mypackage.RequiredService"/>
    <!-- Définition de MyDispatch  dans fichier Fractal ADL MyDispatch.fractal -->
    <component name="Dispatch" definition="MyDispatch">
      <virtual-node name="myhost1.com"/>
    </component>
    <!-- Définition de MyServer  dans fichier Fractal ADL MyServer.fractal -->
    <component name="Server1" definition="MyServer">
      <virtual-node name="myhost2.com"/>
    </component>
    <component name="Server2" definition="MyServer">
      <virtual-node name="myhost3.com"/>
    </component>
    <component name="Server3" definition="MyServer">
      <virtual-node name="myhost4.com"/>
    </component>
    <binding client="this.req_s" server="Dispatch.in"/>
    <binding client="Dispatch.out1" server="Server1.srv"/>
    <binding client="Dispatch.out2" server="Server2.srv"/>
    <binding client="Dispatch.out3" server="Server3.srv"/>
  </component>
  <component name="Composite3">
    <!-- Composant partagé -->
    <component name="Server3" definition="./Composite2/Server3"/>
    <!-- Définition de MySubtask dans fichier Fractal ADL MySubtask.fractal -->
    <component name="Subtask1" definition="MySubtask">
      <virtual-node name="myhost2.com"/>
    </component>
    <binding client="Server3.deleg" server="Subtask1.do"/>
  </component>

  <binding client="this.run" server="Client1.run_s"/>
  <binding client="Client1.req" server="Composite2.req_s"/>
</definition>
```

Listing 2.2 – Extrait de description d'architecture en Fractal ADL

La balise *virtual-node* sert à spécifier le nom du *noeud* Fractal sur lequel un composant doit être déployé. Un noeud Fractal est en fait un processus de bootstrap distant —*i.e.* un processus racine capable d'instancier des composants Fractal de tout type— enregistré dans un registre sous un certain nom (qui sera récupéré via l'attribut de la balise `name` de la balise dans le descripteur Fractal ADL). Cette extension *virtual-node* provient initialement de l'ADL de ProActive (présenté dans la section 2.4.3) et a été intégré ensuite dans Fractal ADL.

Synthèse Comme Darwin, Fractal ADL ne couvre qu'un nombre limité d'étapes du cycle de vie du déploiement. L'installation est gérée par téléchargement de code (adapté du mécanisme de *codebase* de Java RMI) entre processus bootstrap Fractal. La configuration des composants

ne se fait qu'à l'aide d'*attributs* de composants. L'activation est gérée via un gestionnaire de cycle de vie des composants Fractal (appelé le *LifeCycleController*).

L'abstraction choisie dans Fractal ADL ne permet de ne manipuler que des instances de composants, leur configuration et leurs liaisons, et se limite donc aux applications métiers implantées dans ce paradigme.

Des vérifications existent au niveau de la syntaxe et du type des constructions utilisées. Les fichiers doivent être conformes à un *Document Type Definition* XML, qui définit la grammaire des descripteurs.

Le passage de la description d'architecture au déploiement de l'architecture est direct. Aucun mécanisme dans Fractal ADL ne permet nativement de spécifier une reconfiguration dynamique d'architecture. Enfin, Fractal ADL ne permet de déployer que des composants Fractal, donc la granularité du logiciel déployé est limitée aux composants métiers, et la technologie se limite au langage de l'implémentation Fractal choisie. Ainsi l'abstraction utilisée est restreinte à la description de composants métiers, de ses attributs et d'une notion très limitée de machine hôte (le *virtual-node*).

2.2.3 SafArchie

SafArchie est un langage de description d'architecture générique réalisé par O. Barais dans l'équipe GOAL/Jacquard du Laboratoire d'Informatique Fondamentale de Lille [7]. Ce travail propose de rapprocher les mécanismes parfois mis en œuvre dans les plates-formes adaptables de ce qui est exprimable et analysable dans les modèles de conception, c.-à-d. les ADL.

Ainsi l'auteur préconise, pour concevoir des systèmes répartis d'utiliser un modèle de composants abstrait, plutôt que d'utiliser un modèle de composants industriel existant. Cela doit permettre d'accroître la compréhension et la réutilisation des modèles d'architecture définis avec ce langage. L'auteur propose donc un modèle d'architecture reposant sur les notions de composant, port, opération et liaison. Un **composant** est une unité de calcul possédant une interface pour communiquer avec d'autres composants à l'extérieur. Cette interface décrit les services fournis et requis par les composants en utilisant le concept de **port**. Un port est un ensemble d'**opérations** fournies ou requises pour un composant donné. Les ports de composants sont connectés entre eux pour modéliser l'interaction entre composants : ce sont les **liaisons**.

Un exemple d'architecture SafArchie sest représenté sur la figure 2.6 et son expression dans le langage de SafArchie est détaillée sur le listing 2.3.

```
component gun {
  Port ProvideGas {
    provide Gas providegas(gasDescription g);
  }

  Port GetGas {
    require Gas getgas(gasDescription g);
  }
}
```

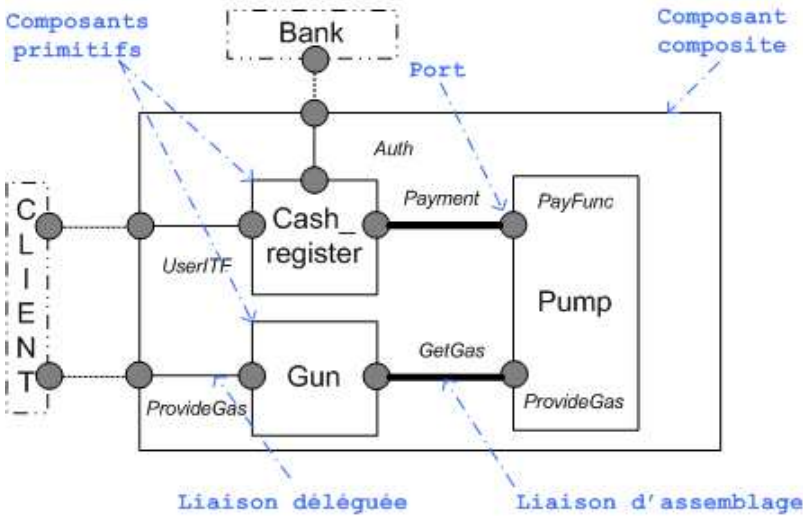

FIG. 2.6 – Architecture de station essence en SafArchie

```
10
   component Cash_Register {
12  Port Payment {
       require void start() ;
14     provide void stop(int prix);
    }
16  Port Auth {
       require boolean askauth(string clientID);
18     require boolean sale(string clientID, int price); /*@ [PRE : self.parameter.price
            > 5, POST : true] ; @*/
    }
20  Port UserITF {
       provide void putcard(Card c);
22     provide void entercode(int code);
       require void givecard(Card c);
24     require void givereceipt(Ticket t);
    } /*@ (putcard + givecard + entercode ) @*/
26 }

28 component Pump {
    Port ProvideGas {
30     provide Gas getgas(gasDescription g);
    }
32  Port PayFunc {
       provide void start();
34     require stop(int price);
    }
36 }

38 component Station {
    include Cash_Register, Gun, Pump;
40
    bind Cash_Register.Payment Pump.PayFunc;
42  bind Pump.ProvideGas Gun.GetGas;

44  Port ProvideGas redirect Gun.ProvideGas;
    Port UserITF redirect Cash_Register.UserITF;
```

```
46    Port Auth redirect Cash_Register.Auth;
}
```

Listing 2.3 – Station essence en langage SafArchie

SafArchie propose un certain nombre de vérifications syntaxiques et sémantiques sur les modèles d'architecture. Ces vérifications sont exprimées sous forme de **contrats** de quatre types différents.

Tout d'abord le **contrat syntaxique** consiste à assurer que les deux ports impliqués dans la définition d'une liaison sont bien compatibles en terme d'opérations fournies et requises.

Le **contrat d'assertion** consiste, comme son nom l'indique, à ajouter des assertions, —*i.e.* des expressions booléennes qui doivent être vérifiées à l'exécution. Elles sont de trois types, les pré-conditions, les post-conditions et les invariants. Un exemple d'un contrat de pré et post condition est représenté à la ligne 18 sur le listing 2.3.

Le **contrat d'assemblage** permet de préciser les dépendances entre opérations au sein d'un même port ——*i.e.* les opérations nécessaires au bon déroulement de l'interaction, un exemple de ce genre de contrat est présent sur la ligne 25 du listing 2.3.

Enfin le dernier type de contrat est appelé **contrat de comportement**. Ce dernier permet de décrire l'enchaînement des messages reçus et émis par le composant. Cette description du comportement permet de détecter des problèmes de dépendances dans les échanges de messages. Un exemple de comportement, celui du composant `Cash_Register` est le suivant : *(?putcard → ?putcode → !askauth → !givecard) | (?putcard → ?putcode → !askauth → !start → ?stop → !sale → !givereceipt → !givecard)*. Ce contrat spécifie les deux comportements, en terme d'enchaînement de messages reçus (?) et émis (!), que peut adopter le composant.

Synthèse Comme Darwin et Fractal ADL, la couverture du cycle de vie du déploiement dans SafArchie est très limitée. L'abstraction choisie reste elle aussi cantonnée à la description des composants et de leurs liaisons. SafArchie ajoute néanmoins dans cette abstraction la notion de contrat. Le principal apport de SafArchie par rapport aux autres ADL réside dans sa capacité de vérification. En effet, les 3 types de contrats qu'il propose en plus du simple contrat syntaxique commun à tous les ADL fait de SafArchie un ADL résolument orienté vers la sûreté des architectures logicielles. Le contrat comportemental est principalement intéressant car il permet une véritable vérification statique du comportement des architectures. Cette vérification va au delà des simples validations structurelles. L'automatisation du déploiement est réalisée par le biais de projections [8]. Une transformation des descripteurs SafArchie vers des descripteurs architecturaux équivalents vers Fractal ADL ou ArchJava est implémentée. En revanche aucun mécanisme pour la reconfiguration dynamique des architectures ou la gestion des environnements ouverts n'est possible. Enfin l'hétérogénéité reste aussi limitée que pour Fractal ADL et Darwin, puisque ce langage ne s'applique qu'aux composants métiers SafArchie.

2.2.4 Synthèse

Notre étude sur les langages de description d'architectures nous amène à penser que ces approches sont insuffisantes du point de vue du déploiement de systèmes logiciels dans leur

totalité. En effet, les ADL permettent bien souvent d'exprimer l'architecture métier du système logiciel. Dans un premier temps, cela signifie que les autres couches qui existent au sein d'un système logiciel —*i.e.* les bibliothèques ou autres mécanismes intergiciels— sont ignorées ou considérées comme déjà déployées au moment de l'utilisation de l'ADL. C'est d'autant plus dommageable que les technologies à composants requièrent toujours au minimum des *serveurs de composants* —*i.e.* les supports d'exécution des composants logiciels.

En conséquence, l'architecte doit intervenir manuellement sur les machines hôtes pour déployer ces serveurs et leurs dépendances, en amont de son utilisation de l'ADL pour déployer la partie métier de son système. Les ADL ne permettent donc de déployer qu'un sous-ensemble de système logiciel.

Dans un second temps, l'utilisation d'un ADL permet de décrire *ce que l'on souhaite déployer*, mais jamais *comment cela doit être déployé*. En effet les descripteurs sont transformés automatiquement en appels sur l'API de déploiement d'un modèle de composants donné. Ainsi, aucun contrôle n'est laissé sur la manière, en terme d'actions élémentaires, de déployer les parties du logiciel. En résulte une homogénéité technologique, puisque la même API est utilisée pour tous les composants de l'architecture.

En revanche, de nombreux ADL prennent en charge la dynamicité des architectures et rendent les configurations décrites reconfigurables ou adaptables à l'exécution (*e.g.* Darwin). Néanmoins cette dynamicité ne peut concerner que la couche métier du système logiciel, ce qui reste insuffisant dans le cadre des environnements ouverts distribués, dans lesquels aucune supposition ne peut être faite en terme de logiciels pré-installés sur les machines joignant dynamiquement le réseau. Ainsi l'intégralité de la pile logicielle doit parfois être déployée sur les machines.

Enfin, il convient de noter que l'aspect vérification et validation (parfois statique, comme dans SafArchie) est souvent présent dans les ADL récents ce qui rend la construction d'architectures plus fiable. En outre, certains ADL reposent sur un modèle formel solide tel que le π-calculus [63] (Darwin) et garantissent mathématiquement la vérification des architectures.

2.3 Modèles de déploiement

Dans cette section, nous allons étendre notre champ d'étude à une autre catégorie de modèles. Les ADL sont des modèles uniformes de déploiement, qui visent à spécifier le déploiement d'architectures à base de composants. Les modèles que nous allons étudier dans cette section ne se limitent pas aux seules architectures à base de composants. Dans ces approches un effort est fourni pour tenter de s'abstraire du paradigme utilisé. Ainsi les approches qui vont être explicitées dans la suite tentent de répondre à la problématique de déploiement de logiciel en définissant un méta-modèle, —*i.e.* un modèle de modèles, ou encore un modèle qui définit la structure des modèles de déploiement. Ces méta-modèles contiennent l'ensemble des concepts

du domaine de déploiement conformément à la vision des différents auteurs.

2.3.1 Modèle de déploiement de l'Unified Modeling Language

C'est en 1997 que l'Object Management Group (OMG), organisme de standardisation international, fait paraître la première spécification de l'Unified Modeling Language (UML). Le but d'UML est de couvrir l'expression de toutes les étapes du cycle de vie des logiciels. Pour cela, différentes vues d'un même système logiciel sont exprimables grâce à UML sous forme de modèles. UML est un langage semi-formel, qui repose sur un méta-modèle défini à l'aide du Meta-Object Facility (MOF) ——*i.e.* le langage de méta-modélisation proposé également par l'OMG. Le méta-modèle d'UML est découpé en plusieurs paquetages, chacun servant à décrire une vue du système. Le paquetage de concepts qui nous intéresse ici est le paquetage de déploiement, dont nous étudions la version 2.1.2 de la spécification [53].

Ce paquetage regroupe des concepts pour la définition de l'architecture d'exécution de systèmes décrits en UML. Un exemple d'un tel modèle, dans sa forme graphique, est représenté sur la figure 2.7.

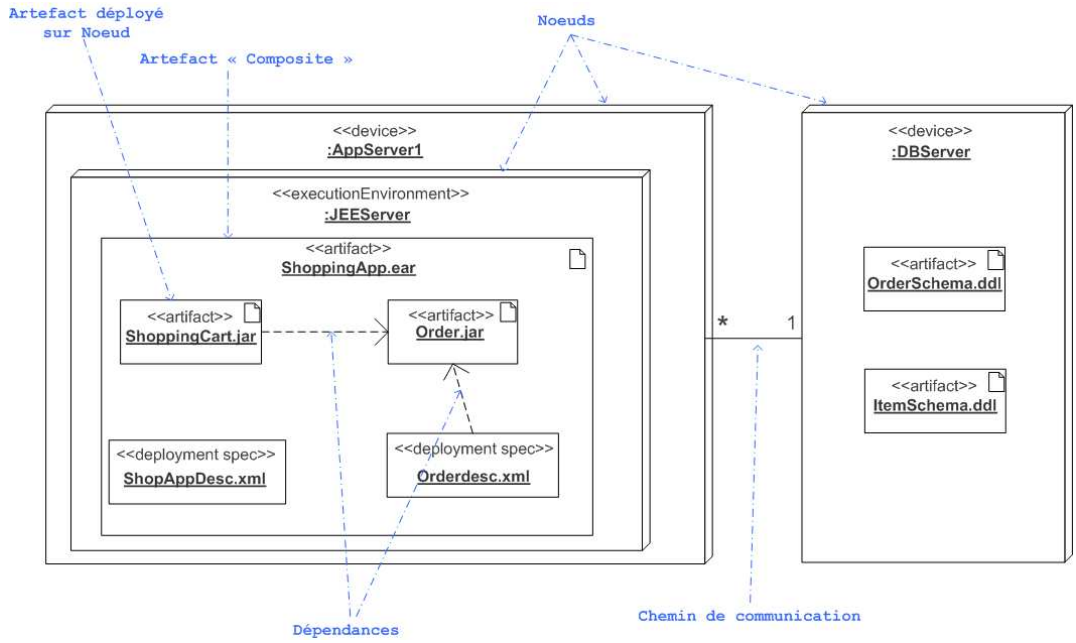

FIG. 2.7 – Représentation graphique informelle d'un modèle de déploiement UML

Les principaux concepts du méta-modèle de déploiement d'UML, représenté sur la figure 2.8, sont les suivants. Les **artefacts** (*artifact*) sont les entités de base du modèle. Ils représentent un élément physique utilisé ou produit par le processus de déploiement. Il s'agit par exemple de bases de données, de fichiers exécutables, ou encore d'archives de classes comme sur la figure 2.7. Ces artefacts peuvent fournir des opérations et être caractérisés par des propriétés particulières. Ils sont récursifs et peuvent être composés d'autres artefacts. Les **nœuds**

(*Node*) sont des ressources de calcul sur lesquels des artefacts peuvent être déployés.

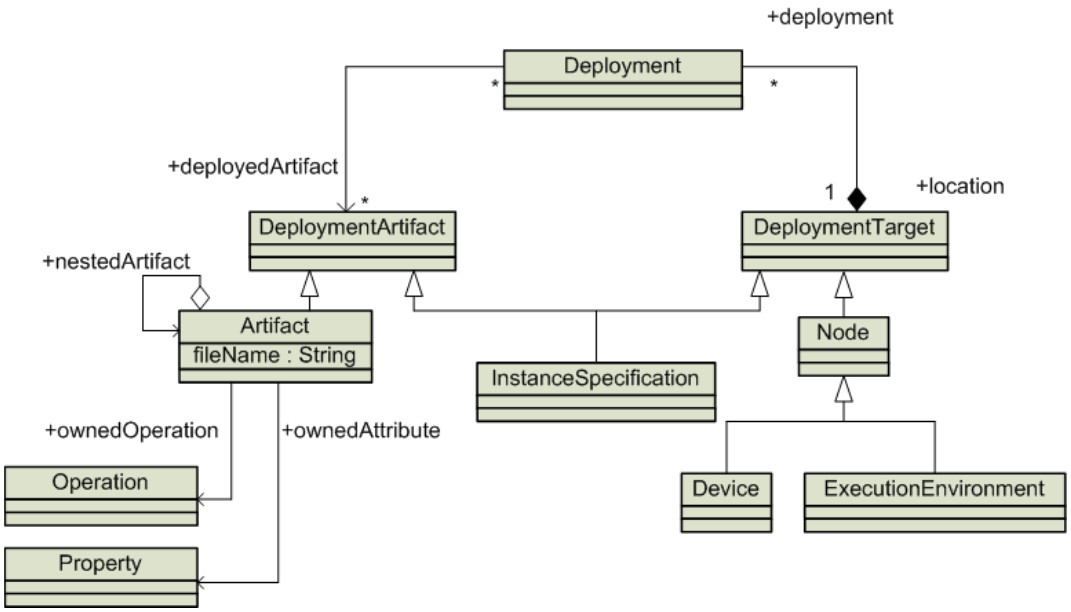

FIG. 2.8 – Méta-modèle de déploiement de la spécification UML

Les nœuds peuvent êtres liés les uns aux autres par des **Chemins de communication** (*CommunicationPath*), qui réifient dans le modèle le concept de lien réseau. Les *Device* et *ExecutionEnvironments* sont des sous-types de *Node*. Les Device représentent une entité physique capable d'héberger des artefacts, ou des ExecutionEnvironments. Les ExecutionEnvironments représentent des entités logicielles servant de support d'exécution à des artefacts.

Synthèse UML propose un modèle de déploiement intéressant, principalement grâce au recul pris par l'OMG pour évoquer le déploiement. Toute notion de cycle de vie du déploiement est omise dans cette approche, puisqu'elle est uniquement descriptive.

En effet, l'abstraction choisie permet uniquement de décrire l'état final du système une fois déployé, mais il est impossible de décrire de quelle manière on arrive à ce résultat. Le modèle de déploiement d'UML, comme les ADL, ne permettent que de décrire la structure du système logiciel à déployer. Ils ne permettent pas de spécifier les actions à mettre en œuvre pour parvenir à une telle architecture physique. En revanche, différentes granularités sont prises en compte dans un modèle de déploiement. Par exemple, la figure 2.7 contient un serveur d'applications, un sous-ensemble de ce serveur dédié à JEE, et enfin une archive correspondant à un composant logiciel à déployer sur le serveur JEE.

Cependant absolument aucune vérification sémantique n'est faite sur les modèles écrits, qui ont davantage une valeur de descripteur graphique, plutôt que de spécification du déploiement.

Ainsi, aucune vérification comportementale ne peut être faite sur un modèle de déploiement UML, et seules des vérifications structurelles basiques peuvent être effectuées.

De plus, l'effort de généralisation et de recul choisi par l'OMG pour ce méta-modèle a été fait au détriment de la sémantique du modèle. Ainsi pour représenter un déploiement complet, et envisager de le projeter sur une plate-forme d'exécution du déploiement, les mécanismes d'extension tels que le Profil UML doivent être utilisés pour personnaliser les modèles de déploiement, c.-à-d. restreindre les concepts du modèle afin d'en spécifier la sémantique pour une technologie donnée.

Enfin, il s'agit d'une des rares approches à évoquer des artefacts physiques pour parler du déploiement. En utilisant ce modèle, il est possible de spécifier la topologie des machines hôtes, et la distribution physique des artefacts d'un système logiciel. Cependant, aucune gestion d'environnements ouverts n'est traitée.

2.3.2 ORYA

ORYA, qui signifie *Open enviRonment to deploY Applications*, est une approche générique de déploiement proposé par V. Lestideau du LISTIC en Savoie [44]. Cette approche a pour but de simplifier la description des étapes du cycle de vie du déploiement dans un contexte industriel. Le but est de déployer la version d'un logiciel la plus adaptée aux besoins des utilisateurs, en respectant les contraintes de la machine. Le déploiement de ces logiciels doit se faire en conformité avec les stratégies d'entreprise (politiques de sécurité, configuration réseau, pares-feux, etc).

ORYA propose un support automatisé pour l'exécution des activités, en privilégiant la réutilisation des logiciels de déploiement existants. Cette approche propose une montée en abstraction pour exprimer de manière générique les déploiements d'applications d'entreprise à base de composants. ORYA repose donc sur des méta-modèles. Le travail sur ORYA a été approfondi pour deux étapes particulières du cycle de vie, la sélection et l'installation des logiciels d'entreprise. Le processus de sélection consiste à mettre en place un algorithme de sélection d'implémentation de composants logiciels conformes aux spécificités de la machine sur laquelle ces composants seront déployés. Le processus d'installation vise à décrire précisément et de manière générique les étapes à mettre en oeuvre pour installer un composant sur une machine de l'environnement d'entreprise. Pour cela, ORYA utilise un langage de *Procédés*, dont le méta-modèle est représenté sur la figure 2.9, qui regroupe les concepts nécessaires à la description des procédures d'installation des logiciels de manière générique.

La généricité est acquise grâce au concept de *Rôle* qui permet de remplacer un outil par son interface d'utilisation. Ainsi tout outil de transfert de fichiers sera représenté par le même rôle, qu'il s'agisse du protocole FTP ou SCP. Les autres concepts principaux du méta-modèle de procédés sont : les *activités* qui représentent des actions élémentaires, les *ports* qui représentent les points d'entrée et de sortie des activités, et enfin les *produits* qui représentent les entités physiques manipulées par les activités. ORYA dispose également d'un méta-modèle pour décrire les environnements d'entreprise représenté sur la figure 2.10. Ce méta-modèle comporte le concept de *serveur d'applications* —*i.e.* support d'exécution pour les applications à déployer.

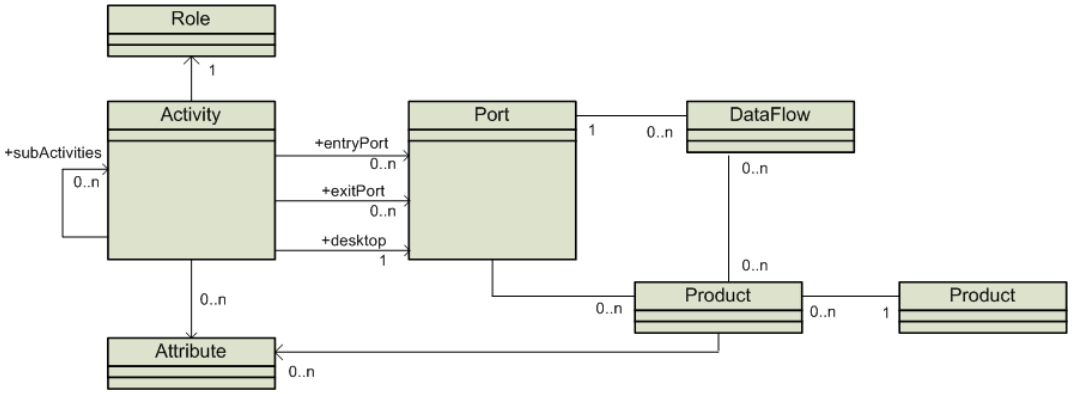

FIG. 2.9 – Méta-modèle de procédés d'ORYA

Il comporte également le *serveur de déploiement*, responsable de la gestion des procédés. Les *sites* sont également représentés et dénotent les ressources d'une machine à disposition pour le déploiement d'applications.

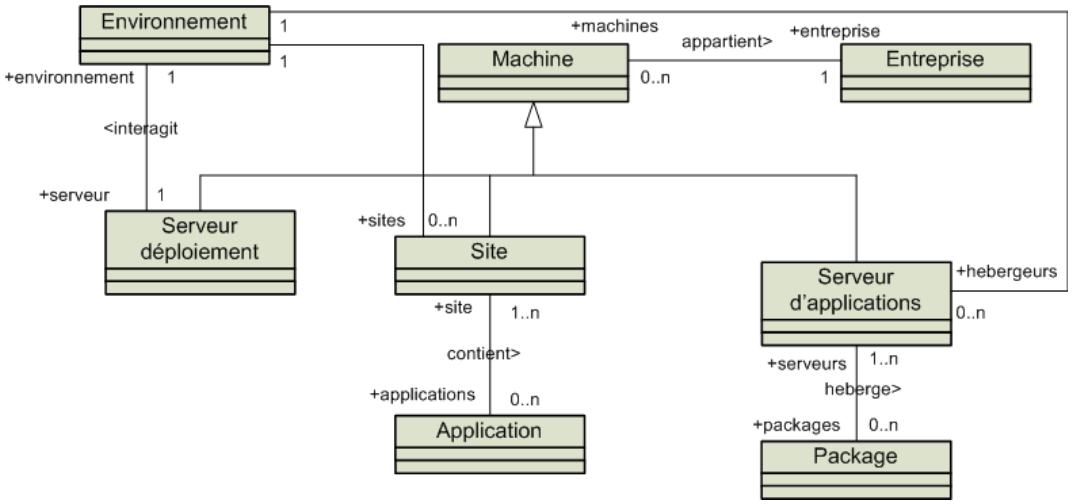

FIG. 2.10 – Méta-modèle d'environnement d'entreprise d'ORYA

Synthèse ORYA propose, à l'instar du méta-modèle de déploiement d'UML, une prise de recul par rapport à l'activité de déploiement logiciel dans un contexte réel. Cette approche ne se focalise pas seulement sur le déploiement de composants logiciels métiers, mais également sur des considérations matérielles et administratives.

En revanche, certaines étapes du cycle de vie du déploiement ne sont pas représentées dans ORYA, telles que la reconfiguration, l'adaptation dynamique ou même le retrait de logiciel. ORYA propose une abstraction suffisamment générique pour représenter n'importe quel type de

logiciel. Enfin, certains concepts du méta-modèle restent flous sémantiquement, *e.g.* le concept de rôle qui abstrait l'hétérogénéité des protocoles d'accès aux machines (transfert, commande distante, etc.), mais qui semble cacher la difficulté d'adaptation aux dits protocoles. Aucune vérification n'est faite sur les modèles afin de prévoir d'éventuels incompatibilités entre logiciels ou avec les machines. Les concepts du méta-modèle d'ORYA, dans l'état, sont difficilement projetables sur des plates-formes d'exécution données. En outre, les environnements d'entreprises sont considérés comme statiques dans ORYA, et la gestion d'environnements ouverts distribués est exclue. Si les hétérogénéités des machines hôtes semblent prises en compte, cette approche est fortement orientée vers le déploiement de composants logiciels, et homogénéise la granularité des logiciels déployables avec ORYA, ainsi que la technologie.

2.3.3 Modèle OMG D&C

En 2005, l'OMG a publié une nouvelle spécification, nommée *Deployment and Configuration of Component-based Distributed Applications* [52]. Cette spécification a pour ambition de définir un méta-modèle entièrement générique dédié au déploiement d'applications à base de composants distribués. Cette initiative a été lancée suite au manque de standardisation dans le déploiement de composants CORBA [51], modèle de composants également défini par l'OMG.

Le méta-modèle comporte deux grandes parties, une première définissant les méta-données du déploiement (le *data model*), et la seconde décrivant le processus de déploiement censé manipuler ces méta-données (le *target model*).

Les méta-données sont regroupées en trois parties. La première partie est dédiée à la définition des types de composants impliqués dans l'application. Le paquetage du méta-modèle correspondant à cette partie est représenté en figure 2.11. Parmi les concepts proposés, on peut trouver le **PackageConfiguration** qui est le concept de plus haut niveau servant à décrire un logiciel conditionné. Le concept de **ComponentPackageDescription** sert à décrire le contenu d'un paquetage logiciel. Enfin, les concepts de **ComponentImplementationDescription** et **ComponentInterfaceDescription** permettent respectivement d'exprimer les différentes implémentations existantes pour le logiciel considéré, et les différents ports de communication que ce composant propose ou requiert.

La seconde partie concerne la description du domaine de déploiement, c.-à-d. l'ensemble des machines sur lesquelles l'application est déployée. Le paquetage du méta-modèle correspondant à cette partie est représenté en figure 2.12. Ce paquetage propose entre autres des concepts tels que le **Node** pour représenter le support d'exécution de composants, le concept de **Bridge** et d'**Interconnect** pour représenter les liens réseaux entre les différents nœuds.

Enfin la troisième partie permet de spécifier une configuration donnée décrivant l'affectation d'instances de composants définis dans la première partie sur des noeuds, également appelée *plan de déploiement*. Le paquetage du méta-modèle correspondant à cette partie est représenté sur la figure 2.13. Ce paquetage propose des concepts tels que le **DeploymentPlan** pour représenter le plan de déploiement, le **InstanceDeploymentDescription** pour la configuration d'une instance de composant d'un type donnée sur un noeud donné, ou encore de **PlanConnectionDescription** pour la définition d'un lien entre les ports de différentes instances de composants décrites.

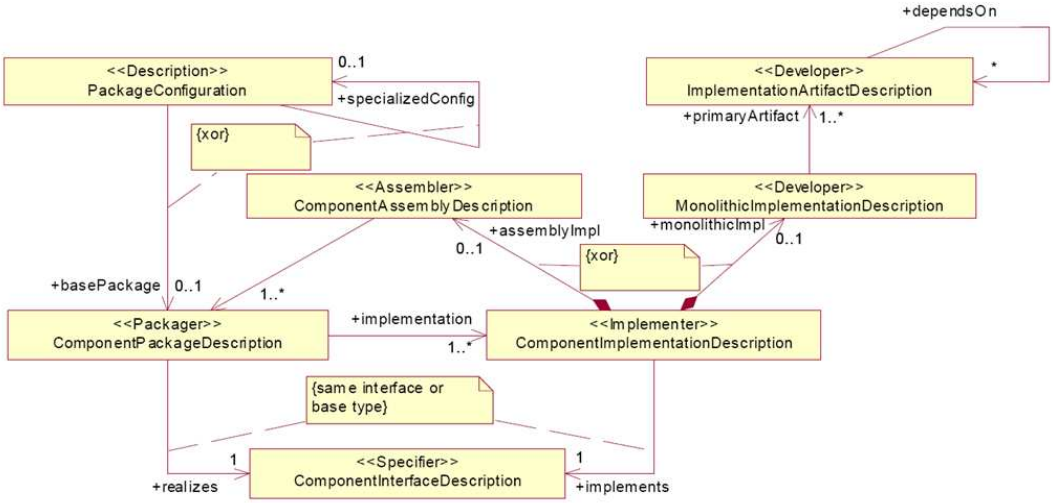

FIG. 2.11 – Paquetage de types du méta-modèle OMG D&C

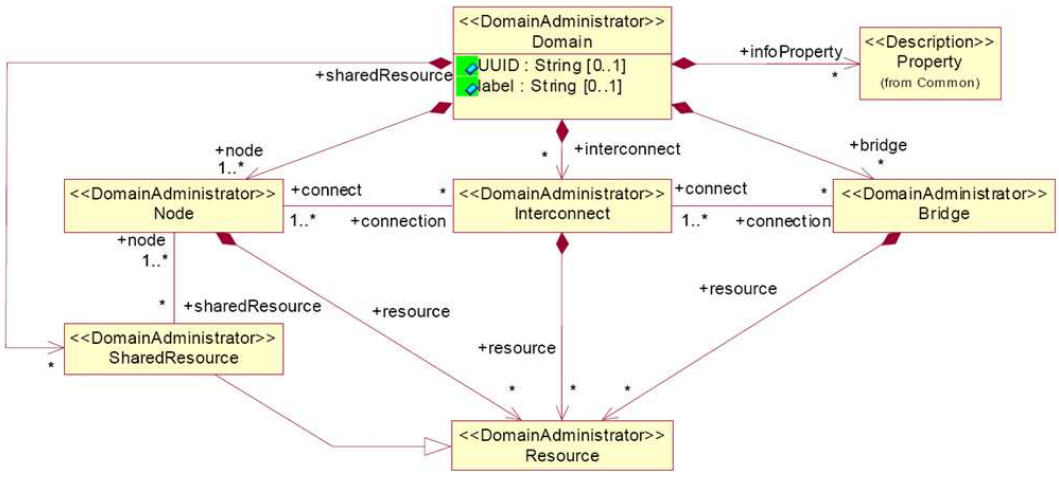

FIG. 2.12 – Paquetage du Domaine du méta-modèle OMG D&C

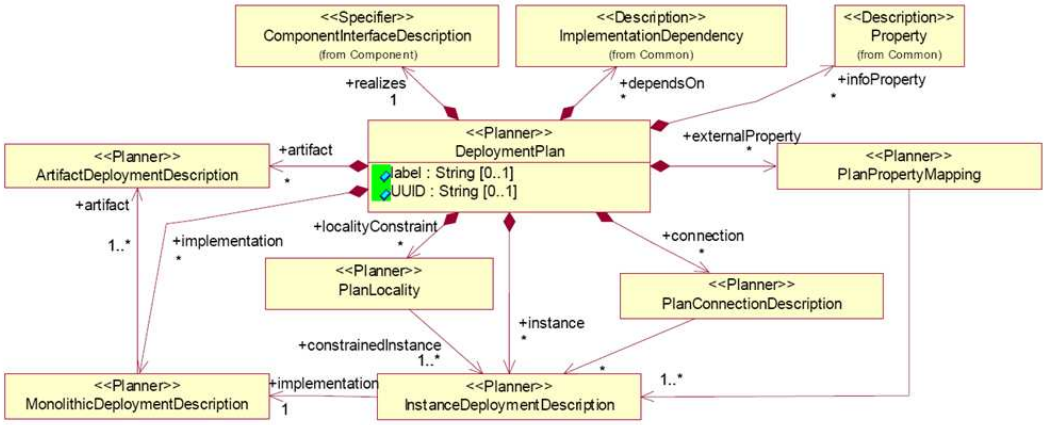

FIG. 2.13 – Paquetage du Plan du méta-modèle OMG D&C

La seconde partie du méta-modèle, dédiée à la description du processus d'interprétation des méta-données pour effectuer le déploiement, spécifie le comportement de différentes parties du cycle de vie du déploiement. Parmi ces étapes, on trouve l'installation, la configuration (considérée dans ce modèle comme une étape faite sur le logiciel avant d'être distribué chez le consommateur), la planification (établissement du plan de déploiement), la préparation, et enfin le lancement. Cette partie du méta-modèle a pour ambition de décrire les différentes interfaces et opérations standardisées des logiciels capables d'effectuer le déploiement d'applications à base de composants.

Synthèse Via la seconde partie du méta-modèle qui standardise les interfaces des logiciels qui déploient les applications, la spécification OMG D&C couvre une partie assez large du cycle de vie du déploiement, à l'exception des phases de reconfigurations dynamiques. Le modèle OMG D&C constitue finalement un ADL, qui possède en plus l'avantage de prendre en compte la description des machines hôtes dans la définition de l'architecture logicielle. Néanmoins, *a posteriori* ce modèle s'avère difficile à adopter[8] car il possède un nombre de concepts très important, et l'abstraction choisie est trop riche en concepts. Cette verbosité du langage est illustrée par l'extrait de définition D&C donné en annexe A. Aucune vérification, exceptées celles de la syntaxe et des types, n'est proposée dans cette spécification. Il est en outre parfois difficile de transposer ces concepts vers ceux d'une technologie précise. En revanche, les méta-données définies dans ce modèle sont très proches des méta-données présentes pour les composants CORBA, ce qui couple fortement les deux spécifications, et facilite la transformation des modèles OMG D&C vers des modèles d'assemblages de composants CORBA. D'ailleurs, les principaux travaux autour d'OMG D&C tels que [26] et [27] concernent le déploiement de composants CCM exclusivement, faisant oublier la généricité d'OMG D&C. La seule forme d'hétérogénéité traitée dans cette spécification reste l'hétérogénéité des modèles

[8] Comme nous le verrons dans la suite du document, et plus précisément en section 6.6.

de composants, bien qu'elle soit discutable tant les concepts sont proches de ceux de CORBA. En conclusion, OMG D&C souffre finalement des mêmes inconvénients que les ADL, mais bénéficie d'une généricité accrue, et étend sa vision à la description des machines hôtes et leurs interconnexions.

2.3.4 GADE

GADE est un modèle générique d'applications de calcul intensif sur grilles de calcul, réalisé par Sébastien Lacour et al. à l'IRISA de Rennes [43] [42].

La première étape du déploiement de ces applications passe par un *planificateur de déploiement*. Ce planificateur prend en entrée la description de la topologie des machines hôtes, la description de l'application à déployer, et enfin des paramètres de contrôle venant influer sur le processus de décision. En sortie, le planificateur produit un *plan de déploiement*, c.-à-d. une description de l'affectation de sous-parties de l'architecture de l'application sur les machines hôtes du réseau. Cependant, les applications de calcul intensif peuvent être de différents types.

Le problème soulevé par ce travail est que pour chaque paradigme de programmation des applications de la grille (*e.g.* à base de composants comme dans GridCCM[9], ou de programmes parallèles MPI [66]) et pour chaque technologie il faut réécrire un planificateur prenant en compte les spécificités de la technologie choisie pour générer un plan de déploiement cohérent. Or, la logique du planificateur n'est pas réellement modifiée par la technologie ciblée pour l'application. De manière abstraite, le planificateur se contente d'affecter des *entités de calcul* sur les machines hôtes, qu'il s'agisse de processus MPI ou de composants GridCCM.

La proposition consiste donc à abstraire les concepts manipulés par le planificateur, et d'en extraire un modèle générique, appelé GADE, d'applications destinées aux grilles de calcul. Les descriptions d'applications spécifiques à une technologie sont donc projetées vers ce modèle générique pour enfin générer un plan de déploiement produit par l'unique planificateur prenant en entrée des modèles GADE. L'exécution du plan de déploiement est bien sûr ensuite effectuée en accord avec les protocoles de déploiement des technologies choisies. Un exemple de modèle GADE est représenté sur la figure 2.14.

Le méta-modèle de GADE, représenté sur la figure 2.15, comporte quatre concepts de base. Le premier concept est **l'entité système** (*system entity*) qui correspond à une machine hôte physique. Ces entités systèmes sont reliées entre elles par le biais du concept de **connexion** (*connection*), deuxième concept du méta-modèle de GADE. Ces entités systèmes hébergent des **processus** (*process*). Ces processus correspondent à des entités logicielles exécutables, ou le cas échéant de support d'exécution pour d'autres logiciels. Dans le monde des composants, un processus correspond à la notion de serveur de composants —néanmoins les composants sous forme exécutable peuvent être directement représentés par un processus— et correspond également à la notion de processus MPI. Enfin, les logiciels s'exécutant sur les entités systèmes sont les **codes à charger** (*codes to load*). ils correspondent directement à la notion de composant. Notons que cette notion est inutile dans le monde MPI, puisque le processus est l'entité de base du modèle MPI.

[9] www.irisa.fr/paris/Gridccm/

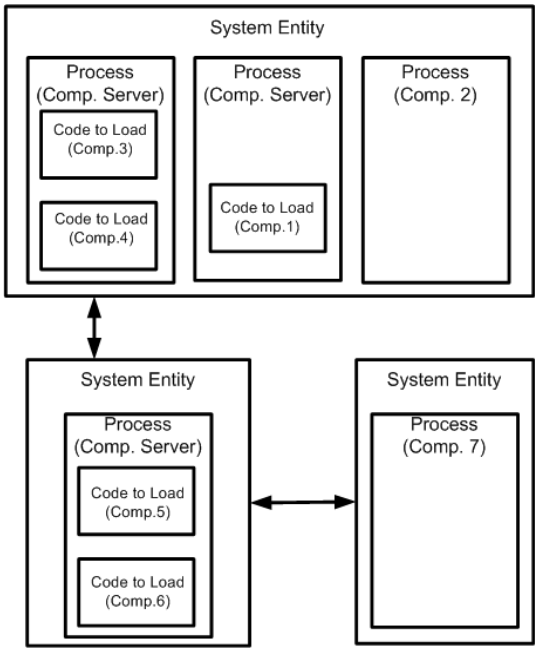

FIG. 2.14 – Exemple de modèle GADE dans le monde des composants

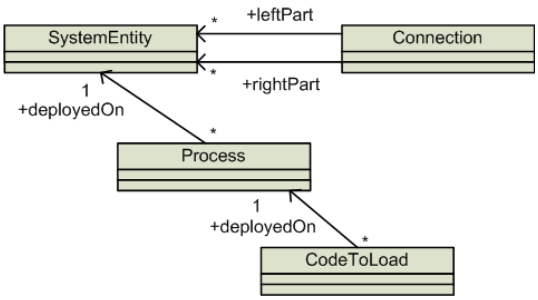

FIG. 2.15 – Méta-modèle de GADE

Synthèse La gestion du cycle de vie du déploiement dans GADE se limite à l'installation et au démarrage de processus. GADE propose une abstraction très simple, composée d'un nombre minimaliste de concepts. La vérification des modèles GADE se limite là encore à la syntaxe et au typage des concepts. Si ces modèles seuls sont nécessaires pour la planification, beaucoup d'autres informations manquent pour automatiser l'exécution du plan de déploiement, comme les propriétés des machines, les commandes élémentaires à lancer pour démarrer les processus, etc. Cependant les auteurs soulignent la possibilité de projection vers des plates-formes de déploiement telles qu'ADAGE [42]. En outre, ce méta-modèle minimaliste prend en considération le déploiement de logiciels à granularité différente comme le processus ou les composants. Cette généralisation du modèle d'applications pour grilles de calcul s'inscrit dans une démarche de génération automatique de plan de déploiement séduisante. Néanmoins, ce méta-modèle, par sa simplicité, ne permet de décrire qu'un nombre d'applications très limité. Le modèle GADE facilite donc grandement le calcul du plan de déploiement, atout primordial dans le contexte des grilles de calcul, mais ne donne que trop peu d'informations pour exécuter automatiquement ce plan.

2.3.5 Synthèse

Les modèles explicitement dédiés au déploiement ont une ambition commune qui est d'être générique. Néanmoins, pour le modèle de déploiement d'UML, cette volonté de généricité a conduit le modèle vers une sémantique floue. Ce défaut est partiellement corrigé par OMG D&C qui propose une pléthore de concepts, à la sémantique certes plus précise. Cependant, la vision des applications métiers a été restreinte une fois de plus aux seuls composants logiciels, et la généricité du modèle reste relative. D'un autre côté, GADE propose un méta-modèle générique de déploiement, mais qui reste, même si cela n'est pas dit explicitement, uniquement utilisable pour un sous-ensemble des logiciels existants. Enfin, ORYA propose un méta-modèle de déploiement d'applications d'entreprise intéressant, qui néanmoins ne prend pas en compte l'hétérogénéité sous toutes ces formes, et qui notamment restreint le nombre de technologies impliquées dans le système à déployer.

En conclusion, les modèles de déploiement existants peinent à fournir un langage de déploiement permettant de décrire de manière homogène le déploiement de toutes les couches d'un système logiciel, quelles que soient les technologies mises en œuvre. En outre, des étapes telles que la reconfiguration dynamique, qui sont centrales dans le cadre d'environnements ouverts et à large échelle, sont souvent ignorées par ces modèles génériques.

2.4 Plates-formes intergicielles pour le déploiement

Dans cette section, nous allons décrire le fonctionnement de quelques plates-formes existantes de déploiement de systèmes logiciels. Ces plates-formes reposent parfois sur des modèles similaires à ceux décrits dans la section précédente, mais les travaux que nous allons analyser ici fournissent également le support d'exécution du déploiement décrit à l'aide du modèle. Il s'agit donc d'approches plus opérationnelles.

2.4.1 Software Dock

En 1999, R. Hall du laboratoire SERL de l'Université du Colorado présente pour la première fois un environnement de déploiement logiciel distribué appelé Software Dock [37]. Cet environnement fait intervenir trois acteurs. Le premier est le **Release Dock**, un serveur rendant accessible des logiciels de *producteurs* pour installation sur machines distantes. Le deuxième acteur est le **Field Dock**, un serveur fournissant une interface d'action sur les machines hôtes, ainsi que les informations pertinentes quant à l'état de ces machines : le système d'exploitation, les ressources système, les logiciels déjà installés, etc. Enfin, les **agents** sont les acteurs responsables du processus de déploiement, donc de l'exécution des étapes du cycle de vie du déploiement des logiciels. Cette architecture est représentée sur la figure 2.16.

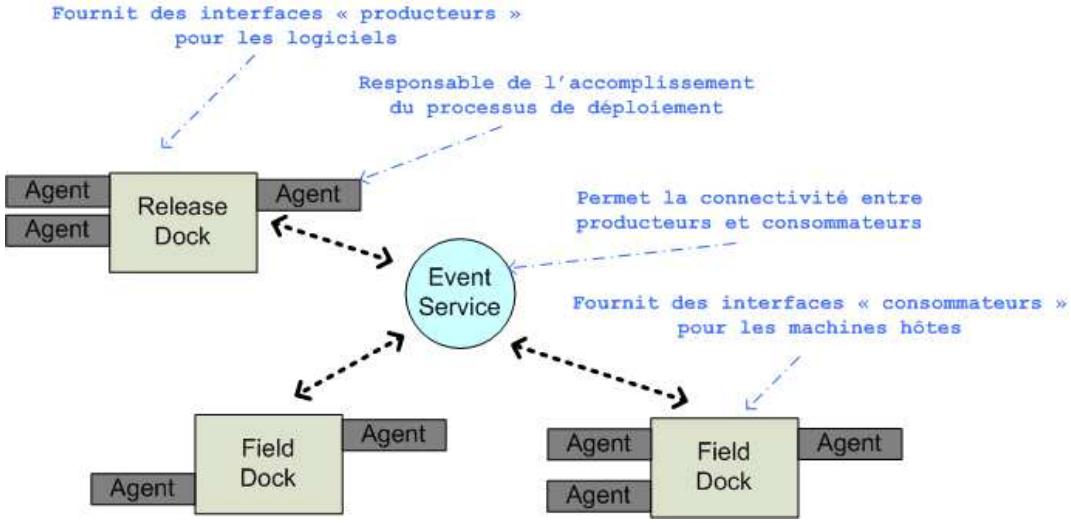

FIG. 2.16 – Architecture du Software Dock

Le concept central de l'approche est le modèle utilisé pour exprimer des *familles* de logiciel fournies par le producteur. Ce modèle, appelé *Deployable Software Description* (DSD), est subdivisé en 5 parties. La **Configuration** correspond à la définition de propriétés pertinentes pour le logiciel (*e.g.* implémentation, numéro de version, etc.) ainsi que les différentes valeurs qu'elles peuvent prendre. Les **Assertions** correspondent à des contraintes que doit remplir le site du consommateur, comme des contraintes matérielles (capacité de calcul, de mémoire) ou des contraintes sur le système d'exploitation. Les **Dépendances** correspondent à des pointeurs vers des logiciels devant être présents sur le site consommateur, ou devant être déployés en amont pour assurer le bon fonctionnement du logiciel en cours de déploiement. Les **Artefacts** sont la description des entités physiques (fichiers, etc.) devant être transférés vers le site consommateur pour faire fonctionner ce logiciel. Enfin, les **Activités** sont les différents comportements du logiciel une fois celui-ci déployé.

Les agents peuvent être en charge de n'importe laquelle des étapes du cycle de vie du

déploiement. Pour l'installation par exemple, l'agent d'installation va tout d'abord s'*amarrer* (traduction de *dock*) au site consommateur concerné. Ensuite, il va retirer le descripteur DSD correspondant au logiciel requis par le site consommateur. Cet agent va ensuite inférer automatiquement l'exécution de l'étape —*i.e.* les actions élémentaires à appeler sur l'interface du client— en fonction des informations contenues dans le descripteur DSD, et des informations fournies par les interfaces du site consommateur. Une fois la configuration identifiée en fonction des assertions, l'agent sollicite éventuellement d'autres agents pour le déploiement des dépendances, et enfin télécharge les artefacts. Un autre exemple, pour l'étape de mise à jour, prend en compte, en plus des propriétés du logiciel et du site consommateur, la configuration existante du logiciel. Ensuite, l'agent de mise à jour va comparer le descripteur du logiciel actuellement déployé avec le nouveau descripteur mis à jour chez le consommateur, et appliquer uniquement les changements entre les deux, ce qui signifie enlever ce qui n'est plus dans le nouveau descripteur, et ajouter ce qui n'était pas dans l'ancien.

Synthèse Software Dock est l'une des approches de déploiement logiciel les plus génériques et intéressantes qui soit, malgré sa relative ancienneté. Le cycle de vie du logiciel est presque intégralement pris en charge, notamment la mise à jour des logiciels déployés. L'abstraction choisie est générique car elle permet d'exprimer le déploiement de n'importe quel logiciel applicatif, à condition que celui-ci possède une liste d'artefacts à télécharger, et des propriétés guidant le choix de configuration en fonction du site consommateur. Cependant, la granularité des logiciels déployés reste limitée à des applicatifs. Contrairement aux ADL par exemple, Software Dock permet uniquement de déployer des exécutables et les fichiers de configuration qui les accompagnent. Le concept d'artefact n'est donc pas assez fin pour permettre le déploiement d'un programme métier à base de composants distribués par exemple.

Si aucune vérification n'est faite, à part sur la syntaxe, sur les descripteurs de déploiement, on peut noter toutefois que le déploiement s'adapte aux spécificités de la machine hôte concernée, ce qui limite le risque d'incompatibilité entre la machine et le logiciel par exemple.

En outre, cette approche permet une automatisation accrue des actions élémentaires de déploiement effectuées par les agents. Le processus de déploiement d'un logiciel n'est pas explicitement détaillé, mais est adapté en fonction de la configuration choisie et des propriétés du site consommateur. L'accès à distance à ces sites est lui aussi géré de manière générique par le biais d'interfaces.

Enfin, Software Dock souffre de quelques autres limitations comme le fait qu'elle requiert qu'un serveur *field dock* soit démarré au préalable sur toutes les machines hôtes du domaine de déploiement. De plus, la gestion des environnements ouverts distribués n'est pas envisageable avec Software Dock, puisque la reconfiguration dans cette approche ne concerne qu'un logiciel déjà installé, et ne concerne pas l'architecture du système distribué. Il est donc impossible de reconfigurer un système logiciel dans son intégralité en ajoutant un nouveau logiciel.

2.4.2 Outils de déploiement des serveurs d'applications

Dans cette section, nous ne détaillons pas une plate-forme de déploiement donnée, mais plutôt une famille d'outils de déploiement qui rassemble les mêmes avantages et les mêmes

limitations.

En effet, le succès des serveurs d'applications de type *n-tier* comme Java Enterprise Environment (JEE) [68] ou le framework .NET de Microsoft, ou de type architecture orientée services comme les bus à services d'entreprise JBI comme PEtALS[10], ServiceMix[11] ou OpenESB[12], a engendré un certain nombre de travaux de déploiement pour ces technologies.

Ainsi, les différents serveurs JEE tels que JOnAS, JBoss, Geronimo, ou même le serveur Tomcat proposent des outils pour déployer les applications par dessus ces serveurs.

Par exemple, dans le JSR-88, Sun définit un modèle de déploiement pour les applications d'entreprise conformes au standard JEE [68]. Ce modèle de déploiement définit une API, décrivant des interfaces standardisées, comme représentées sur la figure 2.17 pour réaliser le déploiement d'archives contenant des composants Web (archives WAR), des composants métiers (archives JAR), ou des applications d'entreprise complètes (archives EAR).

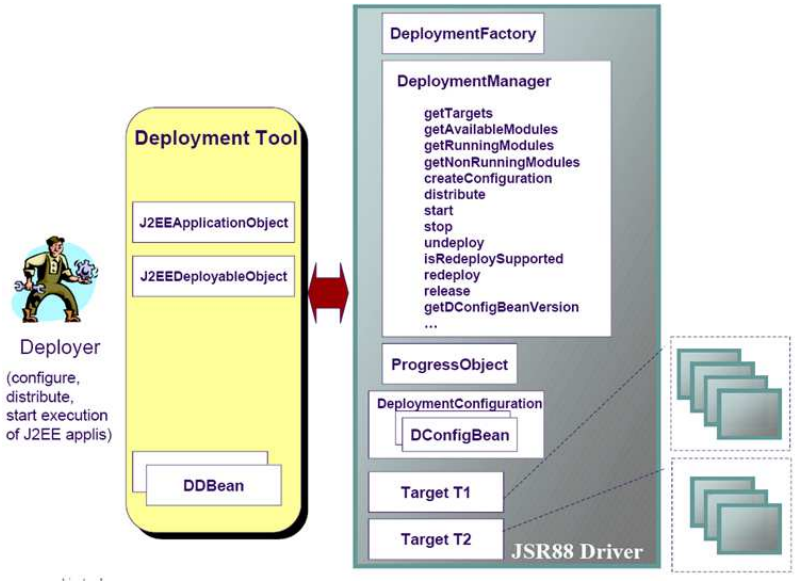

FIG. 2.17 – Architecture de déploiement d'un serveur JEE

De la même manière dans les serveurs d'applications basés sur OSGi [4] comme Felix du consortium Apache[13], il existe des standards de déploiement construits autour de la spécification de l'OSGi Bundle Repository (OBR). Ainsi, les différents composants OSGi sont conditionnés sous forme de *bundles*, ensuite chargés sur l'OBR. Le déploiement d'un bundle à partir de l'OBR extrait le descripteur de déploiement du bundle déployé, afin de connaitre ses éventuelles dépendances vers d'autres bundles à déployer en premier lieu.

Il existe également des outils de déploiement pour le moteur de servlet Tomcat, ou encore le canevas .NET de Microsoft. Ces outils de déploiement ont pour but de faciliter la tâche des

[10] http://petals.ow2.org
[11] http://servicemix.apache.org
[12] https://open-esb.dev.java.net/
[13] http://felix.apache.org

utilisateurs. L'objectif final de ce type d'outils est qu'une fois les composants conditionnés, ils soient déployables de manière la plus naturelle possible.

Ainsi dans les serveurs JEE, le déploiement d'archives se fait généralement par copie de fichier dans un répertoire. Les outils de déploiement du serveur JEE utilisé parcourent ensuite le répertoire de déploiement et automatisent le déploiement réel du composant, c'est-à-dire la mise à disposition de ses services sur le serveur. De la même manière, en OSGi, il suffit de rendre son bundle conditionné sous forme de JAR accessible à distance via le protocole HTTP, pour qu'il soit déployable sur une passerelle OSGi. Du fait de sa simplicité de déploiement, OSGi est souvent choisi comme plate-forme de déploiement pour d'autres technologies. À titre d'exemple, on peut citer l'environnement de développement Eclipse, qui a choisi une implémentation OSGi pour réaliser le déploiement de *plug-ins* [45].

Synthèse Les outils de déploiement embarqués dans les serveurs d'applications sont souvent similaires, et ne prennent en charge qu'un sous-ensemble du cycle de vie du déploiement logiciel. Ce sous-ensemble est souvent composé des opérations d'installation, de configuration, de démarrage et d'arrêt. L'abstraction des concepts est toujours spécifique à la technologie choisie, via un descripteur de déploiement (souvent dans une syntaxe XML) embarqué dans les paquetages de composants. Qui plus est, ces abstractions ne permettent de ne déployer que les applications métiers de cette technologie, et pas les supports d'exécution (serveurs de composants, etc) de ceux-ci ; on ne peut alors pas déployer un système complet. Les vérifications sont souvent dynamiques indirectes, c'est-à-dire que le déployeur doit souvent comprendre par lui-même pourquoi le déploiement de son archive a échoué. L'automatisation de la procédure du déploiement est par contre poussée à son paroxysme, grâce au caractère extrêmement dédié des archives de composants. Peu de reconfiguration ou de mise à jour est possible à l'aide de tels outils. Lorsqu'ils sont possibles, ces mécanismes sont souvent spécifiques à la technologie. La mise à jour consiste souvent à écraser l'ancienne version des composants déployés. Les serveurs d'application sont rarement distribués, plusieurs serveurs peuvent être connectés les uns aux autres, comme c'est le cas pour les ESB, le déploiement des composants se fait généralement de manière locale, et tout déploiement en environnement ouvert est exclu.

2.4.3 ProActive

ProActive est un environnement de programmation pour applications de calcul pouvant potentiellement s'exécuter sur des grilles de calcul [6]. Cet environnement est réalisé par l'équipe INRIA OASIS de Sophia-Antipolis. Le but est de proposer des outils couvrant une grande partie du cycle de vie du logiciel, du développement à l'administration. Pour cela, ProActive propose un modèle de programmation parallèle en Java, à base d'**objets actifs**. Ces objets fournissent des méthodes asynchrones appelables à distance. Les objets sont également **mobiles**, et peuvent être déplacés dynamiquement d'une JVM à une autre. ProActive propose également un mécanisme de communication de groupes entre objets actifs. En plus de ces éléments, ProActive propose également une surcouche de composants, afin de faciliter la construction de systèmes ProActive en utilisant le paradigme d'architecture. Le modèle de composants retenu pour réaliser des composants ProActive s'appelle le *Grid Component Model* (GCM), et

est une extension du modèle de composants Fractal définie par le réseau d'excellence CoreGrid.

Mais dans cette approche, ce qui nous intéresse sont les outils de déploiement d'applications à base de composants ProActive. En effet, deux langages et deux outils de déploiement associés sont proposés pour déployer des systèmes logiciels ProActive complets. Tout d'abord un langage de description d'architecture a été défini pour construire des applications à base de composants ProActive. Ce langage est une extension de Fractal ADL, qui permet de gérer les types de composants ProActive tels que les composants parallèles, non présents dans le modèle Fractal de base. Les nœuds virtuels (VN) présentés avec Fractal ADL sont présents dans l'ADL de ProActive, il s'agit de noms logiques (par opposition aux noms physiques des machines). Les VN définis dans un fichier ADL ProActive représentent une architecture logique, qu'il reste à projeter vers une architecture physique composée de JVM.
Une volonté des auteurs est précisément de découpler intégralement le réseau de JVM du code source des programmes. De cette manière, il est possible de redéployer une application ProActive sur un autre réseau sans avoir à modifier le code source : la préoccupation de placement est tissée au programme au moment du déploiement. Ainsi, un autre langage permet d'écrire des *descripteurs de déploiement*, comme représenté dans le listing 2.4. Ces descripteurs possèdent une partie *virtual nodes* qui permet de déclarer les VN de l'application, une autre partie *deployment* qui permet entre autres d'affecter (*mapping*) une ou plusieurs JVM pour chacun des VN logiques définis dans le code source (une même JVM peut également héberger plusieurs VN, on parle alors de collocalisation de JVM). Enfin, ce descripteur de déploiement, dans sa partie *infrastructure*, permet d'exprimer la localisation physique des JVM ainsi que le protocole de communication de ces machines (*e.g.* SSH, Globus), nécessaires pour l'automatisation de création de JVM.
Finalement, l'outil graphique IC2D (*Interactive Control and Debugging of Distribution*) permet dynamiquement d'administrer le système déployé : migration de composants pendant l'exécution, création de nouvelles JVM, etc.

```
<?xml version="1.0" encoding="UTF-8"?>
<ProActiveDescriptor xmlns="urn:proactive:deployment:3.3" xmlns:xsi="http://..."
    xsi:schemaLocation="...">
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="matrixNode" property="multiple" />
    </virtualNodesDefinition>
  </componentDefinition>

  <deployment>
    <mapping>
      <map virtualNode="matrixNode">
        <jvmSet>
          <vmName value="Jvm1" />
          <vmName value="Jvm2" />
        </jvmSet>
      </map>
    </mapping>
    <jvms>
      <jvm name="Jvm1">
        <creation>
          <processReference refid="ssh_crusoe" />
        </creation>
```

```
      </jvm>
      <jvm name="Jvm2">
        <creation>
          <processReference refid="ssh_waha" />
        </creation>
      </jvm>
    </jvms>
  </deployment>
  <infrastructure>
    <processes>
      <processDefinition id="localJVM">
        <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess" />
      </processDefinition>
      <processDefinition id="ssh_crusoe">
        <sshProcess
                class="org.objectweb.proactive.core.process.ssh.SSHProcess"
                hostname="crusoe.inria.fr">
          <processReference refid="localJVM"></processReference>
        </sshProcess>
      </processDefinition>
      <processDefinition id="ssh_waha">
        <sshProcess
                class="org.objectweb.proactive.core.process.ssh.SSHProcess"
                hostname="waha.inria.fr">
          <processReference refid="localJVM"></processReference>
        </sshProcess>
      </processDefinition>
    </processes>
  </infrastructure>
</ProActiveDescriptor>
```

Listing 2.4 – Définition de mapping Virtual-Node/JVM en ProActive

Synthèse L'approche proposée dans ProActive est très intéressante dans la mesure où elle couvre la quasi-totalité du cycle de vie du déploiement. L'outillage accompagnant cette approche permet de : développer, assembler, déployer automatiquement aussi bien les composants que leur support d'exécution (—*i.e.* les JVM) sur des machines distantes. L'abstraction choisie ne permet de décrire que le déploiement de composants GCM ou d'objets actifs Java, ainsi que la topologie de leur support d'exécution, c.-à-d. des machines virtuelles Java. La définition des architectures de systèmes logiciels est faite via une syntaxe XML sur laquelle aucune vérification n'est faite. Les architectures ProActive et la topologie de leur support d'exécution est déployable automatiquement grâce à l'outil IC2D. Bien qu'il ne soit pas possible d'exprimer la reconfiguration dans l'abstraction choisie, la gestion d'EOD est rendue possible, grâce à IC2D. Cela reste malgré tout entièrement manuel et donc difficilement gérable, voire impossible dans de larges environnements. Plusieurs niveaux de granularité sont pris en compte, puisqu'il est possible de déployer des serveurs d'applications (en l'occurrence ici de simples JVM jouent ce rôle), et des composants métiers (les composants ProActive). Qui plus est, aucune installation manuelle préalable de logiciel n'est nécessaire pour déployer une architecture ProActive sur une machine quelconque. Néanmoins cette approche est dédiée au déploiement d'applications conformes au modèle de programmation ProActive.

2.4.4 GoDIET

GoDIET est un outil de déploiement pour infrastructure DIET (*Distributed Interactive Engineering Toolbox*), tous deux réalisés par l'équipe GRAAL à l'ENS de Lyon [20]. DIET est un *Application Service Provider* (ASP), c.-à-d. un serveur d'applications distribuées pour la résolution de problèmes scientifiques à grande échelle. L'architecture de ce serveur est composé de trois types d'acteurs. Les *Clients* sont les utilisateurs finaux qui contactent directement le serveur pour lancer des requêtes de résolution de problèmes. Les *Serveurs* (SeDs) sont les programmes en charge d'effectuer une sous-partie du calcul global en vue de la résolution du problème soumis par un client. Enfin, les *Agents* sont responsables de la logique de répartition hiérarchique du calcul à effectuer pour résoudre le problème. Ces agents sont divisés en deux catégories. Les *Master Agents* (MA) représentent les points d'entrée pour soumettre un problème ; ils sont responsables d'une sous-partie de l'architecture. Les *Local Agents* (LA) sont optionnels mais peuvent être déployés pour permettre un meilleur passage à l'échelle. Les serveurs sont rattachés à un seul LA (mais en l'absence de LA ils peuvent être rattachés directement à un MA), ils sont responsables de la résolution du problème, ou lancent des exécutables qui le sont. Le scénario de déploiement d'une architecture DIET est représenté sur la figure 2.18.

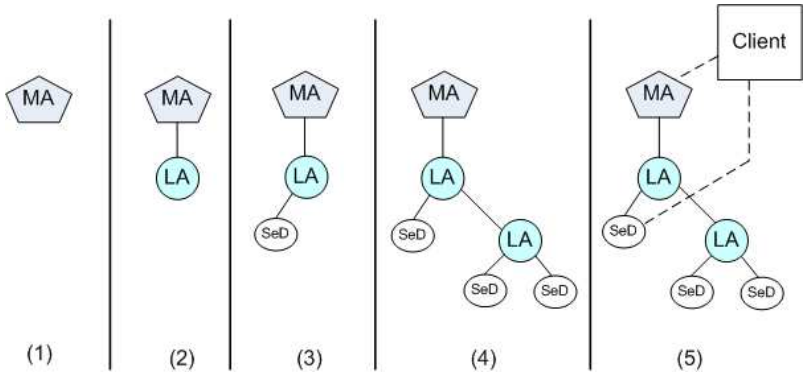

FIG. 2.18 – Scénario de déploiement d'une architecture DIET

Avant tout, non présent sur la figure, un service de nommage doit être déployé, pour retrouver facilement les éléments de l'architecture à l'aide d'une adresse logique. Puis les MA doivent être déployés en tant que racine de l'architecture. Ensuite chaque élément peut-être déployé dans l'architecture à condition de connaître son parent, et donc sa place dans la hiérarchie. Finalement, le client contacte directement un MA pour soumettre une requête de calcul, le MA lui retourne un ensemble de SeD capables de traiter cette requête. Enfin, le client dialogue directement avec les SeD.

GoDIET propose un langage XML permettant de déployer ces architectures, dont un exemple de fichier est donné sur le listing 2.5.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE diet_configuration SYSTEM "../GoDIET.dtd">
<diet_configuration>
```

```
    <goDiet debug="1" saveStdOut="no" saveStdErr="no" useUniqueDirs="no"/>
    <resources>
      <scratch dir="/home/hdail/tmp/scratch_godiet"/>
      <storage label="localDisk">
        <scratch dir="/home/hdail/tmp/scratch_runtime"/>
        <scp server="localhost"/>
      </storage>
      <compute label="localHost" disk="localDisk">
        <ssh server="localhost"/>
        <env path="/home/hdail/diet/omniORB-4.0.6/bin"
                  LD_LIBRARY_PATH="/home/hdail/diet/diet/build_dir/install/lib/>
      </compute>
    </resources>

    <diet_services>
      <omni_names contact="localhost" port="2810">
        <config server="localHost" remote_binary="omniNames"/>
      </omni_names>
    </diet_services>

    <diet_hierarchy>
      <master_agent>
        <config server="localHost" remote_binary="dietAgent"/>
        <local_agent>
          <config server="localHost" remote_binary="dietAgent"/>
          <SeD>
            <config server="localHost" remote_binary="server"/>
            <parameters string="T"/>
          </SeD>
        </local_agent>
      </master_agent>
    </diet_hierarchy>
</diet_configuration>
```

Listing 2.5 – Exemple de plan de déploiement en GoDIET

Les descripteurs écrits à l'aide de ce langage comportent trois parties. La première, *ressources*, contient la description des machines hôtes utilisées pour le déploiement, et de leur protocole d'accès. La seconde, *services*, contient les informations pour le déploiement des services de base comme le service de nommage, mais aussi des agents de journalisation (*loggers*), chargés de ramener les informations affichées sur les machines distantes. Enfin, la troisième partie des descripteurs, *hierarchy*, permet d'affecter, pour chaque MA, LA ou SeD, la machine hôte sur laquelle il est déployé, et le cas échéant l'exécutable à lancer (pour les SeD).

GoDIET propose un mécanisme de surveillance des déploiements et, par le biais des *loggers*, réachemine les sorties standard et d'erreur sur la machine d'où sont lancés les déploiements. Deux types de détection d'erreurs sont possibles. Le premier, sans le service de logger, détecte une sortie anormale lors du déploiement de l'un des éléments de la hiérarchie. Dans ce cas, il suspend le déploiement de tout élément inférieur dans la hiérarchie à l'élément défectueux, pour éviter d'autres erreurs probables, et il marque l'état de l'élément défectueux comme étant *confus* et attend une intervention du déployeur pour savoir si le déploiement de toute la hiérarchie est en danger ou non. La deuxième détection possible est, si le service de logger est démarré, le déploiement défectueux d'un élément est rapporté à ce service, lequel décide de manière autonome, en fonction de la connaissance des messages d'erreurs, si le déploiement

doit être annulé, poursuivi ou confus. Ces vérifications permettent de gagner un temps significatif lors du déploiement d'une architecture DIET à très large échelle qui produit une erreur. Elles permettent également d'éviter d'éventuels dégâts sur le domaine de déploiement.

Synthèse GoDIET propose une approche de déploiement très similaire à celle de ProActive. Celle-ci possède les mêmes limitations, à savoir la possibilité de déployer un ensemble restreint d'élements différents (trois dans le cas de GoDIET), et également la possibilité de déployer uniquement des applications DIET.

L'approche reste séduisante pour certains aspects, pour deux raisons principales. La première est que GoDIET permet, comme ProActive, de déployer des logiciels sur des machines pour lesquelles aucune supposition n'est faite (sinon qu'elles soient accessibles à distance). La seconde, pour laquelle GoDIET se démarque par rapport à ProActive, est l'effort de vérification qui a été fourni par les auteurs de GoDIET. En effet, les mécanismes de détection de fautes de GoDIET permettent de faire de la vérification dynamique indirecte avec intervention de l'utilisateur, pour limiter les dégâts dans l'hypothèse d'un échec dans le déploiement d'un élément de la hiérarchie.

2.4.5 DAnCE / CoSMIC

L'approche DAnCE a été élaborée à l'Université de Vanderbilt, au sein du groupe DOC [26]. Il s'agit d'un environnement de déploiement dédié aux applications à base de composants CORBA contrôlées par des contraintes de Qualité de Service (QoS pour *Quality of Service*). Il s'agit d'une implémentation de la spécification OMG D&C que nous avons détaillée en section 2.3.3.

Les défis que les auteurs souhaitent relever pour le déploiement efficace d'applications à base de composants avec besoin de QoS sont de quatre types. Le premier défi est de fournir un moyen d'accès aux différentes implémentations possibles de composants, pour rendre possible l'adaptation de l'implémentation (langage de programmation, etc.) des composants en fonction des caractéristiques de la machine (bibliothèques présentes, système d'exploitation, etc.) sur laquelle ils s'exécutent. Le second défi consiste à synchroniser les opérations post-installation du cycle de vie du déploiement au sein d'un *assemblage* de composants —*i.e.* plusieurs composants communiquant les uns avec les autres. Par exemple, si l'on souhaite supprimer un composant, il faut assurer que tous les autres composants communiquant avec celui-ci sont passivés, ce afin d'éviter tout envoi de message vers un composant en train d'être désactivé. Le troisième défi est de pouvoir découpler du code du composant les propriétés de QoS que le composant doit remplir. Les contraintes de QoS représentent un préoccupation transverse de l'implémentation d'un composant, et par souci de modularité on souhaite qu'elles soient exprimées à part afin d'être interprétées par les serveurs de composants pour qu'ils soient configurés en fonction des composants qu'il héberge. Enfin, le dernier défi consiste à respecter là encore le principe de modularité, mais cette fois-ci pour la configuration de services intergiciels de base tels que le service de nommage, le service de transactions, etc.

La proposition des auteurs pour résoudre ces défis repose sur deux grandes idées : l'utilisation de la modélisation pour l'expression du déploiement, en utilisant le *data model* d'OMG D&C (et en l'étendant pour les contraintes de QoS et la configuration des services intergiciel), et une architecture de serveurs, conforme au *target model* de l'OMG D&C, en charge d'interpréter ces données. Cette architecture est représentée sur la figure 2.19. Les descripteurs du Data model d'OMG D&C sont réalisés à l'aide de l'outil de modélisation CoSMIC (Component Specification for Model Integrated Computing) [36]. Ce dernier permet une construction graphique des descripteurs de déploiement qui sont ensuite soumis à l'infrastructure de déploiement dont les acteurs sont détaillés ci-après.

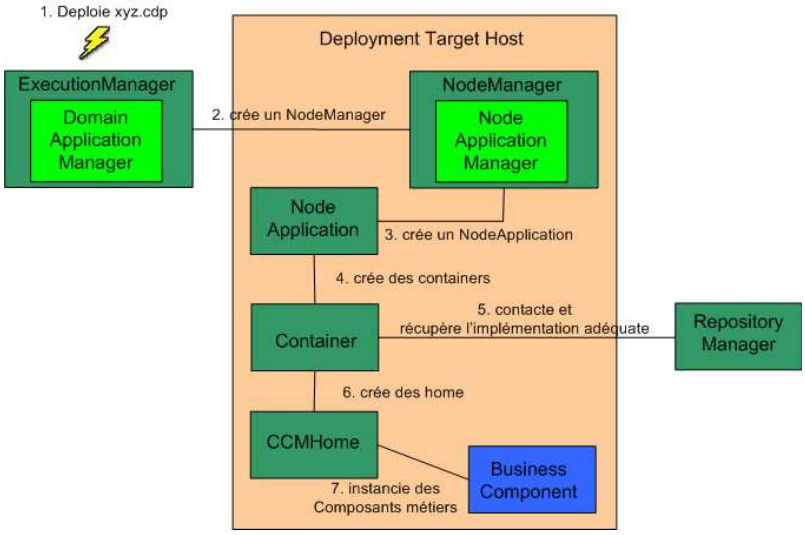

FIG. 2.19 – Architecture de déploiement de DAnCE

L'**ExecutionManager** est l'élément de plus haut niveau dans la hiérarchie des gestionnaires de DAnCE. Il est en charge de la gestion du déploiement sur plusieurs domaines gérés par des **DomainApplicationManagers**. C'est lui qui reçoit le descripteur global de déploiement OMG D&C. Le **DomainApplicationManager** est en charge d'un domaine donné, c'est lui qui sépare le processus de déploiement en plusieurs parties, une partie par **Node**. Il n'interprète que la partie du descripteur de déploiement qui concerne le node dont il a la charge. Le **NodeManager** est présent sur chacun des noeuds concernés par le déploiement d'une application. Il est en charge du déploiement des composants sur la machine qu'il gère, indépendamment de l'application dans laquelle ces composants sont impliqués, ce qui permet un partage d'implémentations de composants entre applications présentes sur une même machine. Le **NodeApplicationManager** est colocalisé avec le **NodeManager**, il reçoit les requêtes de déploiement de composants envoyées par le **NodeManager**, et est en charge d'instancier les serveurs de composants, et de les configurer au regard de leurs contraintes de QoS. Le **NodeApplication** est un serveur de composants, capable d'instancier des composants métiers en fonction d'une implémentation choisie. Il est également capable de relier les composants déployés aux services intergiciels qu'ils requièrent. Enfin, le **RepositoryManager** est un serveur dédié à un domaine capable de fournir aux **NodeApplicationManagers** la liste des implémentations existantes pour chaque

composant disponible. Il reçoit les informations relatives aux types de composants du descripteur OMG D&C.

Synthèse DAnCE est une approche de déploiement à base de composants très intéressante, notamment grâce à l'utilisation de modèles standardisés OMG D&C pour la description de l'architecture de l'application. La couverture du cycle de vie du déploiement s'étend jusqu'au lancement des composants. En outre, l'outil de modélisation CoSMIC permet d'exprimer de manière abstraite le déploiement d'un système logiciel, en y intégrant de manière transverse des préoccupations telles que les contraintes de QoS ou encore la configuration des serveurs de composants, notamment en terme de configuration de services middleware requis. Néanmoins, cette abstraction reste cantonnée à la description d'architectures à composants CORBA ainsi que leur support d'exécution. Aucune vérification des descripteurs n'est faite, conformément à la spécification OMG D&C qui n'en impose aucune au niveau du méta-modèle. Les opérations de déploiement de l'architecture, en terme de déploiement de composants ou de configuration des serveurs, sont directement inférées à partir des descripteurs générés par CoSMIC.

De plus, et c'est un point important, l'architecture de déploiement de DAnCE doit au préalable être déployée manuellement sur l'ensemble des machines hôtes du domaine de déploiement. Cette nécessité affaiblit grandement, voire annihile le potentiel de déploiement d'applications à large échelle et en environnements ouverts distribués. Enfin, cette approche se limite une fois de plus au seul déploiement de composants logiciels, qui plus est limité aux composants CORBA, et aucune gestion de l'hétérogénéité n'est soulevée dans ces travaux.

2.4.6 Déploiement sûr et flexible de composants logiciels

M. Belguidoum propose dans sa thèse [9] un méta-modèle générique de composants, ainsi qu'une architecture de déploiement de ces composants assurant un certain nombre de propriétés de sûreté.

Le méta-modèle global du déploiement comprend quatre grandes catégories. Les **ressources** représentent les entités concernées par le déploiement, principalement les composants et les machines les hébergeant. Les **mécanismes** représentent la description l'activité du déploiement de composant, et des effets qu'il engendre sur les machines. Les **politiques** représentent la description de la manière dont sont réalisés les choix de déploiement, elles dirigent le déploiement suivant des objectifs fixés par les ressources. Enfin, les **propriétés** représentent les différents paramètres des ressources ainsi que les buts à atteindre du système cible.

Dans le modèle de ressources, les auteurs spécifient le modèle des composants sur lequel ils raisonnent. Celui-ci est représenté sur la figure 2.20.

Ces composants exposent leurs fonctionnalités en fournissant des *services*. Ces composants possèdent également des dépendances vers des services requis de deux catégories. Les **inter-dépendances** modélisent la dépendance classique vers le service fourni d'un autre composant. Les **intra-dépendances** modélisent quant à elles les dépendances entre les services que le composant fournit, et ses exigences techniques (*e.g.* système d'exploitation, la mémoire,

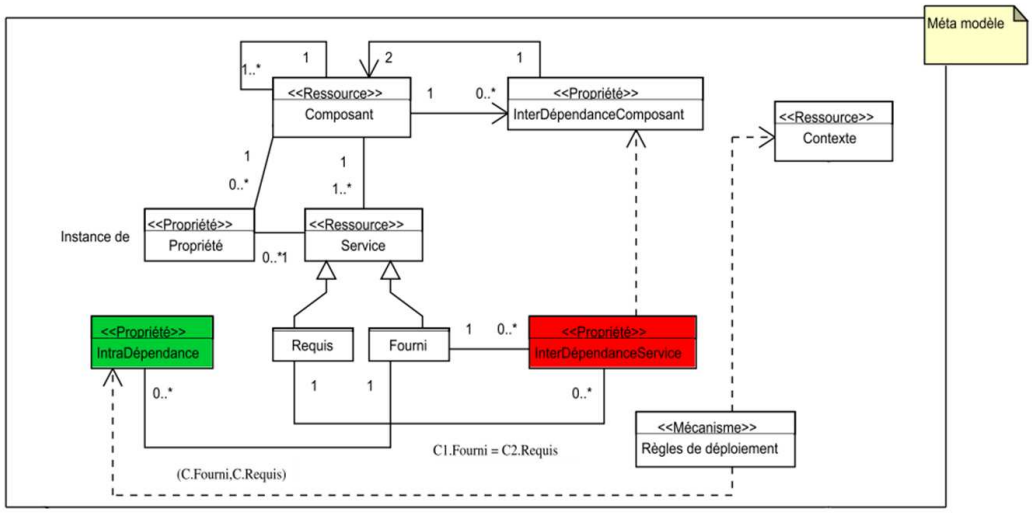

FIG. 2.20 – Méta-modèle proposé dans [9]

bibliothèques présentes). Ces dépendances peuvent être obligatoires, optionnelles ou négatives (pour exprimer des conflits potentiels avec d'autres logiciels).

Un langage basé sur un sous-ensemble de la logique des prédicats a été défini dans ce travail, afin d'évaluer la faisabilité des opérations d'installation, de retrait et de mise à jour des composants au sein d'un système. Le choix de l'utilisation de la logique de premier ordre est de pouvoir utiliser des règles d'inférence pour prouver la faisabilité de ces actions, à partir d'un modèle formel.

Une architecture de moteur de déploiement est également proposée, représentée sur la figure 2.21. Cette architecture comprend une partie formelle reposant sur un moteur de raisonnement chargé d'effectuer les calculs de faisabilité d'ajout, de retrait, de mise à jour d'un composant, à partir de politiques, de la description du contexte (propriétés du noeud sur lequel est déployé le composant), ainsi que des dépendances du composant. Ce moteur de raisonnement est en charge d'appeler les actions sur le moteur de déploiement, mais également d'évaluer l'impact de ce déploiement pour mettre à jour le contexte. Cette architecture est décentralisée et distribuée sur chacune des machines hôtes du déploiement. Un prototype de cette architecture a été implémenté pour le déploiement de bundles OSGi[14].

Synthèse Dans cette approche, toutes les étapes du cycle de vie ne sont pas couvertes par le modèle de déploiement, principalement la reconfiguration dynamique. L'abstraction choisie est également similaire à celle des ADL, puisqu'il s'agit d'un modèle à base de composants. Néanmoins, cette approche est très intéressante en ce qui concerne la vérification. En effet, elle propose un cadre formel pour prouver la faisabilité du déploiement d'un composant en fonction de ses dépendances et du contexte local. En outre, elle bénéficie d'une architecture d'exécution du déploiement embarquant des moteurs de raisonnement fonctionnels capables de faire de ces

[14] http ://www.osgi.org

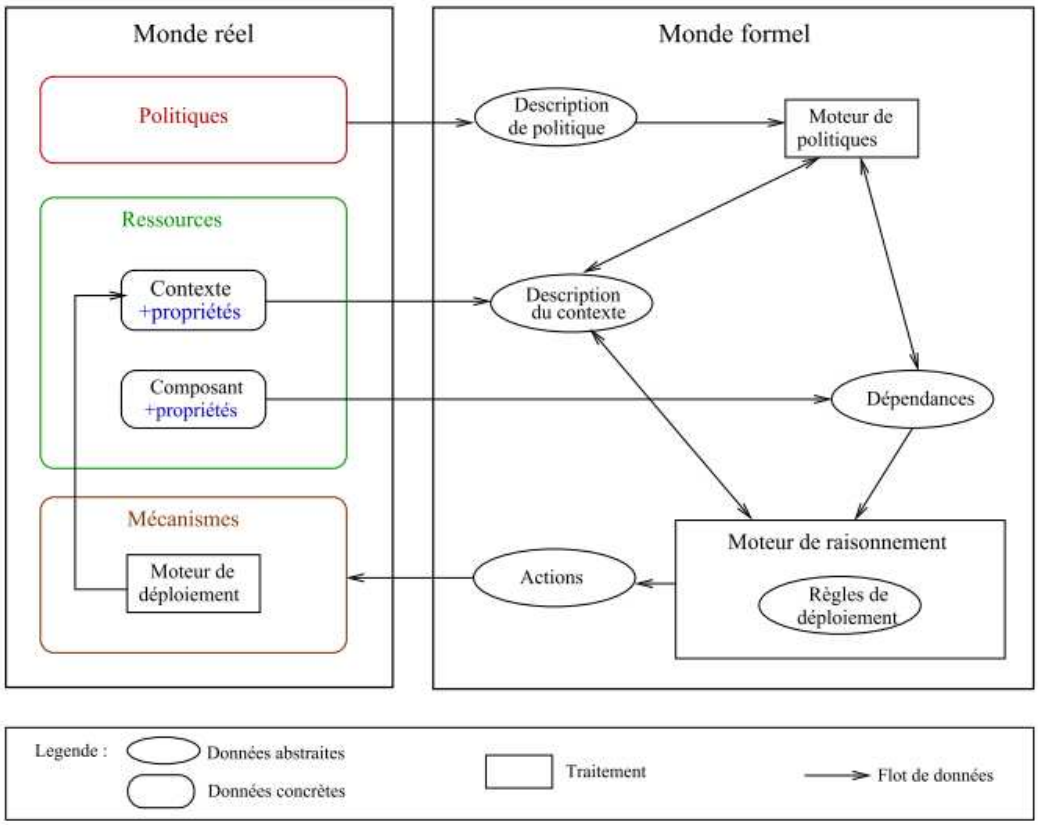

FIG. 2.21 – Architecture de déploiement proposée dans [9]

vérifications formelles une réalité du déploiement.
En revanche, l'architecture de déploiement définie dans ce travail exige de démarrer un serveur de déploiement sur chacune des machines hôtes concernées par le déploiement, ce qui limite une fois de plus le passage à l'échelle d'une telle approche, ainsi que sa capacité à être appliquée dans des EOD. De plus, l'approche prévoit de déployer les applications sous forme de composants, certes génériques, mais ce qui limite la granularité des logiciels déployables.

2.4.7 Synthèse

Cette étude met en lumière un certain nombre de limitations dans les outils de déploiement existants à ce jour. La première limitation est que beaucoup des outils actuels sont spécifiques à une technologie donnée. Les exemples des serveurs d'applications, de GoDIET, de ProActive et de DAnCE illustrent parfaitement ce propos. En effet dans ces quatre approches, un grand nombre d'étapes du cycle de vie du déploiement sont couvertes. Le déploiement d'une infrastructure DIET ou ProActive ne nécessite aucune action manuelle sur les machines hôtes : l'intégralité du système logiciel est déployé. Les communications avec les machines hôtes sont automatiquement établies, les transfert de binaires également ainsi que le démarrage des serveurs. Néanmoins, ces deux approches ne permettent de déployer que des infrastructures pour laquelle elles ont été prévues. L'approche proposée dans SoftwareDock représente quant à elle une démarche générique intéressante pour le déploiement à distance, bien que souffrant elle aussi de certaines limitations, comme l'incomplétude du modèle de logiciel, ou encore la supposition abusive de serveurs *dock* démarrés sur les machines hôtes. Enfin, certains travaux, toutefois assez rares, proposent d'étudier la sûreté du déploiement et d'effectuer des vérifications sur le déploiement comme dans les travaux de M. Belguidoum.

En conclusion, si les modèles génériques de déploiement manquent de sémantique pour en déduire un processus complet de déploiement, les outils de déploiement manquent bien souvent eux de généricité. En outre, la plupart de ces outils ignorent les étapes de reconfiguration dynamique du cycle de vie du déploiement.

2.5 Reconfiguration dynamique et l'adaptation

Dans cette section, nous allons nous intéresser à un sous-ensemble des étapes du cycle de vie du déploiement, à savoir les étapes d'adaptation et de reconfiguration dynamiques. Ces étapes consistent à modifier l'architecture d'un système pendant l'exécution. Ces changements peuvent être dictés par le contexte d'exécution du logiciel, il s'agit alors d'*adaptation* dynamique. En général, toute modification quant à la structure du système durant son exécution est qualifiée de *reconfiguration dynamique*. Ainsi, l'adaptation dynamique représente l'action de piloter la reconfiguration dynamique par des critères du contexte d'exécution du logiciel. Nous allons nous pencher sur différentes approches de reconfiguration et adaptation dynamique, et confronter ces approches à nos critères d'étude. Ces critères n'ont pas de sens pour les outils dédiés à l'une des étapes du cycle de vie qui est précisément la reconfiguration dynamique.

Nous allons étudier les langages et mécanismes existants pour exprimer la reconfiguration dynamique d'un système. Nous allons nous pencher dans un premier temps sur les concepts de l'*informatique auto-gérée* (*autonomic computing*) comme solution automatique de reconfiguration dynamique, pilotée par des politiques de haut niveau, et nous détaillerons quelques implémentation de ces principes.

2.5.1 Systèmes auto-gérés

IBM a introduit le concept d'*autonomic computing* en 2003 [39]. Le terme *autonomic* provient de la biologie. Il désigne les mécanismes humains gérés de manière automatique par le système nerveux central. On parle d'*autonomic reflexes* pour désigner les réflexes gérés à l'insu de la volonté d'un être humain. Par exemple, l'homme est capable de réguler sa température interne sans contrôler directement les mécanismes qui font que son corps se réchauffe. C'est ce type de mécanisme qu'IBM souhaite incorporer dans les systèmes distribués. Selon cette théorie, un système distribué doit posséder un ensemble d'objectifs globaux et doit être capable d'exécuter automatiquement toute action susceptible de réaliser ces objectifs. Il s'agit donc d'une transcription du système nerveux central dans un contexte technologique.

IBM propose une architecture générique pour implémenter des politiques d'auto-gestion dans les systèmes distribués. Cette architecture, appelée *boucle de contrôle*, est représentée sur la figure 2.22.

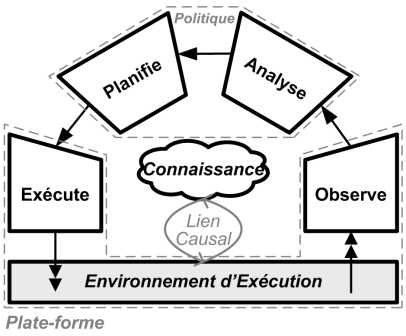

FIG. 2.22 – Boucle de contrôle : le cœur de l'auto-gestion

Cette boucle de contrôle comporte deux environnements et quatre étapes de fonctionnement. Le premier environnement est l'environnement d'exécution —*i.e.* Execution Environment— qui représente le domaine de déploiement du système ainsi que tout artefact logiciel déployé sur celui-ci. L'environnement de connaissance —*i.e.* Knowledge— est un modèle computationnel représentant les éléments de description de l'environnement d'exécution. L'information représentée dans le modèle de connaissance doit se limiter aux informations pertinentes pour la reconfiguration d'auto-gestion. Pour que cette connaissance soit exploitable par des programmes de reconfiguration auto-gérée, il faut qu'elle soit constamment conforme à la réalité de l'environnement d'exécution. C'est pourquoi un *lien causal* est instauré entre ces deux environnements, c'est-à-dire que tout changement dans l'un implique la mise à jour de l'autre. Les

quatre phases de la boucle de contrôle exploitent ces environnements. Tout d'abord, la phase d'*observation* —*i.e.* monitoring— consiste à surveiller l'environnement d'exécution afin de détecter les changements nécessitant une reconfiguration dynamique. Ensuite, la phase d'*analyse* consiste à intégrer les événements détectés par la phase d'observation, et à analyser le contexte de ces événements, en fonction de la connaissance actuelle du système. La phase de *planification* consiste à choisir l'action à effectuer pour reconfigurer le système face au changement observé et analysé. Enfin, la phase d'*exécution* consiste à traduire le résultat de la phase de planification en opérations à appeler sur l'environnement d'exécution. Il s'agit donc des actions concrètes opérées sur le système.

Cette architecture représente ainsi une spécification pour les systèmes autonomes, mais les détails d'implémentations des boucles de contrôle restent personnalisables. Le terme *autonomic* étant un terme anglais qui ne trouve pas d'équivalent dans la langue française (en effet *autonome* en français correspond à *autonomous* en anglais), l'adjectif français accepté dans la communauté est *auto-géré*.

2.5.2 SAFRAN

SAFRAN (Self-Adaptive FRactal compoNents) est une extension du modèle de composants Fractal qui a été réalisé au Laboratoire d'Informatique de Nantes Atlantique dans l'équipe OBASCO par Pierre-Charles David [24]. L'approche choisie dans SAFRAN consiste à rendre les composants d'une architecture Fractal sensibles à leur environnement d'exécution. Pour ce faire, les auteurs préconisent une approche par aspects [40]. Le comportement à adopter par les composants Fractal est encapsulé dans des *composants d'adaptation*, dont le code est dynamiquement *tissé* au code métier du composant via une interface non-fonctionnelle dédiée des composants Fractal, le *contrôleur d'adaptation*.

Trois concepts doivent être identifiés pour mettre en place une approche par aspects : le *plan de base*, c'est-à-dire le code à adapter, *les points de coupe*, c'est-à-dire la définition de points d'entrée dans le plan de base où le code des aspects est exécuté et les *advice*, c'est-à-dire le code à exécuter lorsque l'aspect est appelé. Dans leur vision, les auteurs identifient le plan de base comme étant une architecture Fractal. Les points de coupe sont eux représentés par des événements qualifiés d'endogènes —c'est à dire des appels sur les interfaces d'un composant, la modification de la structure de l'architecture—, ou d'exogènes —c'est-à-dire des événements d'observation du système, tels que la charge du processeur, l'état de la mémoire, etc. Les actions des advices sont eux volontairement limités à la reconfiguration de structure de l'architecture de composants Fractal.

Ces actions de reconfiguration sont exprimées à l'aide du langage FScript [25]. Ce langage repose sur le langage de requêtes FPath, lui-même fortement inspiré de XPath, le langage de requêtes sur des fichiers XML. Pour rendre possible la navigation, les composants Fractal d'une architecture Fractal sont modélisés sous la forme d'un graphe orienté dont les nœuds représentent les composants, et les arcs représentent des relations entre les composants (l'annotation sur les arcs permet de déterminer la nature de la relation modélisée par l'arc —*i.e.* liaison, composition, etc.). FPath est donc un langage pour la navigation et la sélection d'éléments

dans une architecture Fractal, de manière similaire à ce qu'il est possible de faire avec XPath et des fichiers XML. FScript quant à lui permet, en s'appuyant sur FPath, de selectionner des ensembles d'éléments d'architectures Fractal, et d'appeler des opérations (qui correspondent à des actions présentes dans l'API Fractal) sur ces éléments.
De plus, la consistance des reconfigurations est assurée dans FScript via différents critères. Ainsi, l'intégrité transactionnelle —*i.e.* l'atomicité, l'isolation et la consistance de l'état final— et la terminaison en temps borné sont autant de propriétés vérifiées statiquement par le moteur FScript, afin d'assurer le bon déroulement des reconfigurations, et la non-corruption du système en cours d'exécution.

Les auteurs étendent la notion de points de coupe, classiquement limitée au *code de base*, à l'occurrence d'événements dits exogènes. La détection d'événements endogènes nécessite des mécanismes d'observation du système. Pour cela, les auteurs proposent WildCat, un canevas en Java conçu pour la réalisation d'applications sensibles au contexte. Les événements détectés par WildCat sont ensuite injectés dans Fractal via le contrôleur d'adaptation.

Le paradigme choisi pour exprimer les politiques de reconfiguration à tisser sur les composants Fractal est de la forme Événement-Condition-Action (ECA) [22]. Il permet d'exprimer sous une forme réactive les opérations à effectuer en réponse aux événements définis. Ces politiques d'adaptation sont dynamiquement tissées sur le composant, en utilisant une construction propre au modèle Fractal, le *contrôleur*, une interface non-fonctionnelle qui va intégrer la politique d'adaptation et appeler cette politique lors des evènements qui surviennent durant la vie de composant, comme représenté sur la figure 2.23.

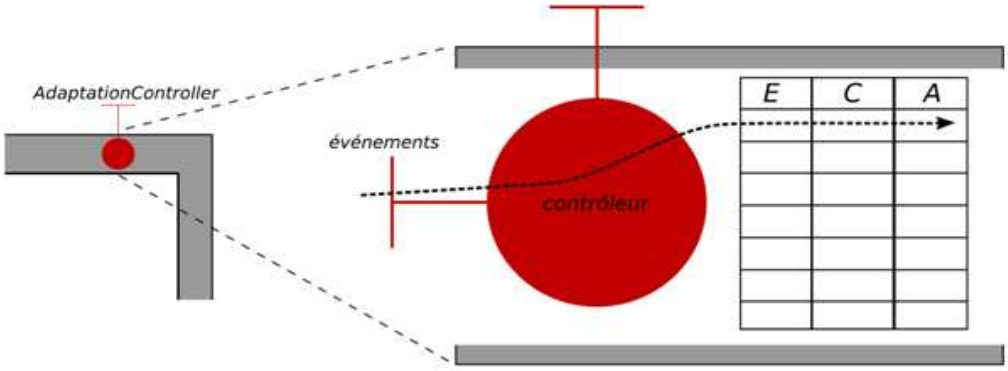

FIG. 2.23 – Contrôleur d'adaptation de Safran

Synthèse L'abstraction choisie pour représenter des politiques d'adaptation est pertinente, et le paradigme ECA paraît être la solution la plus concise pour exprimer des politiques de reconfiguration réactives. Le langage FScript pour implémenter les actions de reconfigurations est lui aussi assez concis. En outre, les propriétés ACID sont assurées par le moteur d'exécution des scripts de reconfiguration, ce qui renforce la sécurité de l'application, et permet la vérification

de certaines propriétés dans ces scripts. On peut également constater que les logiciels devant être adaptés dynamiquement par Safran doivent posséder un point d'entrée pour la reconfiguration, en l'occurrence le contrôleur d'adaptation. Il n'est donc pas possible de reconfigurer n'importe quel logiciel patrimonial. De plus, cela a pour conséquence de limiter la catégorie de logiciels reconfigurables avec Safran, à la seule catégorie d'applications implémentées à l'aide de composants Fractal. L'hétérogénéité des technologies est donc volontairement omise dans ce travail.

Enfin, même si le canevas Wildcat est extensible et donc censé pouvoir détecter n'importe quel type d'événements, une projection de Safran en environnements ouverts distribués est inconcevable pour la simple raison que les prototypes existants de Safran ne sont applicables qu'à des applications Fractal non distribuées.

2.5.3 MADAM

MADAM (*Mobility and ADaptation-enAbling Middleware*) est une approche provenant des laboratoires de recherche SINTEF ICT et Simula d'Oslo [32]. Cette approche préconise l'utilisation de modèles d'architecture pendant l'exécution pour rendre possible l'auto-adaptation. Les objectifs de ces auto-adaptations sont un meilleur confort d'utilisation, de meilleures performances et une plus grande fiabilité des applications s'exécutant sur terminaux mobiles. Ces adaptations sont opérées en fonction du contexte de l'utilisateur.

Ce contexte comprend des informations de deux types : le contexte de l'utilisateur (qui comprend des paramètres aussi variés que la position, la luminosité, le bruit ambiant, le porteur a-t-il les mains libres pour interagir avec le terminal, etc.) et le contexte système (niveau de batterie, capacités de calcul, qualité de service du lien réseau, etc.). Les architectures dans MADAM sont représentées à la manière des *lignes de produits* [56]. Ces architectures présentent donc des *points de variation*. Par exemple, dans le cas d'une architecture à composants, comme c'est le cas dans MADAM, un point de variation est représenté par de multiples implémentations pour un *type de composant* donné. Il est ainsi possible de modifier l'implémentation de ce type de composant en réponse à un changement de contexte.

Le choix des adaptations est guidé par des *fonctions d'utilité* (*utility functions*). En effet, à chaque implémentation d'un type de composant sont affectées des propriétés qui prennent des valeurs scalaires en fonction de valeurs du contexte global —*i.e.* contexte utilisateur et contexte système. L'architecte pondère les valeurs de différentes propriétés afin d'obtenir une valeur scalaire pour l'implémentation, en fonction d'un contexte donné. Un exemple de tel calcul est représenté sur la figure 2.24.

Lorsqu'un changement de contexte est détecté par la couche intergicielle de MADAM, le calcul d'utilité est effectué pour chaque variation et les choix d'implémentations sont effectués de manière à maximiser la fonction d'utilité globale du système. Les adaptations adéquates sont ensuite effectuées.

Synthèse Cette approche permet d'adapter les implémentations de composants de manière à optimiser l'utilité, la fiabilité et l'ergonomie des logiciels sur terminaux mobiles. Concer-

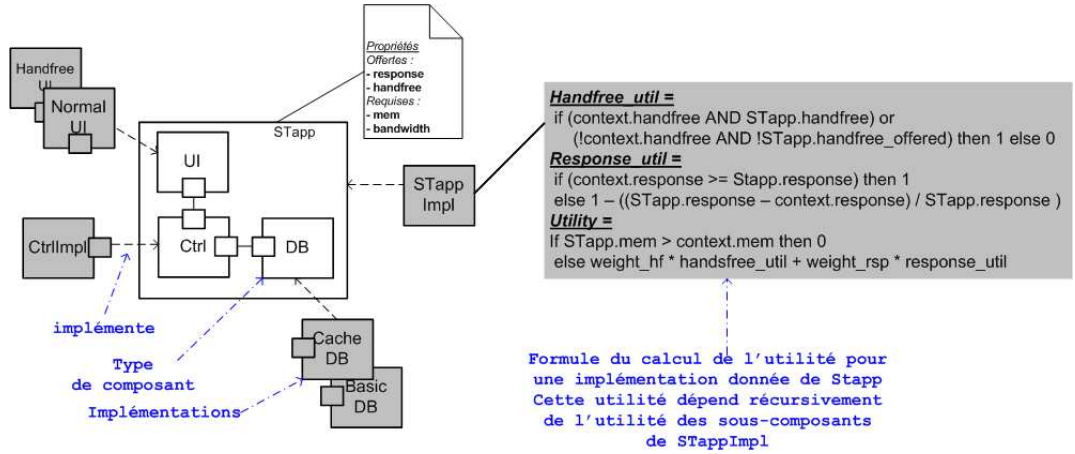

FIG. 2.24 – Exemple de types de composants et de calcul d'utilité dans MADAM

nant l'abstraction choisie, les fonctions utilitaires présentent deux défauts majeurs pour notre problématique. Premièrement, ils ne permettent pas d'exprimer la nécessité d'adapter l'architecture en fonction d'un événement discret comme l'apparition de nouvelles machines hôtes. Deuxièmement, ces fonctions sont complexes à exprimer et donc à mettre en œuvre. La paradigme mathématique utilisé est très éloigné des compétences que l'on requiert d'un architecte ou d'un administrateur système. Aucune préoccupation de vérification n'est abordée dans les travaux sur MADAM. Les types des composants sont automatiquement adaptés afin d'optimiser l'utilité de l'architecture. Cette approche s'avère utile en EOD pour adapter les composants à déployer sur des terminaux en fonction du contexte du terminal. Néanmoins, cette approche ne prend pas en compte le déploiement de nouveaux composants en fonction de l'arrivée de nouveaux terminaux dans le réseau. Enfin, MADAM ne permet que l'adaptabilité des composants métiers, et pas de leur support d'exécution. Un unique niveau de granularité est donc géré par MADAM.

2.5.4 Jade

Jade est une plate-forme d'administration autonome de grappes de serveurs JEE [13]. Un cluster JEE est utilisé dans un contexte d'applications JEE devant répondre à un grand nombre de requêtes simultanées. Dès lors, on réplique les traitements entre plusieurs serveurs applicatifs, et les données entre plusieurs serveurs de bases de données. Un exemple d'une telle architecture est représenté sur la figure 2.25

Pour déployer et administrer de tels systèmes, les auteurs proposent d'*encapsuler* les logiciels patrimoniaux sous forme de composants Fractal. Ainsi la représentation du système logiciel complet est représenté par un composite Fractal. Les dépendances entre les logiciels représentées par des flèches sur la figure 2.25 deviennent des liaisons entre composants Fractal.

Les *interfaces de contrôle* des composants Fractal fournissent des opérations non-fonctionnelles pour le composant considéré. Ainsi le *contrôleur de contenu* (*content-controller*) offre

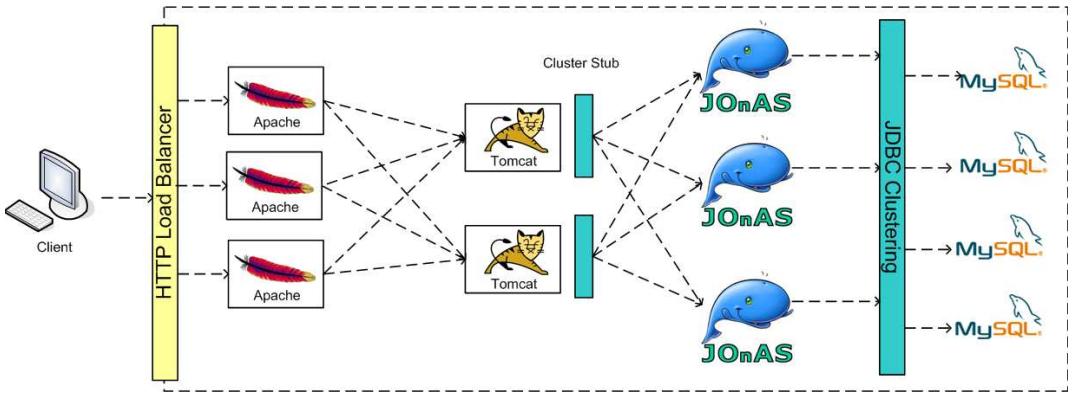

FIG. 2.25 – Architecture d'un cluster JEE

des opérations pour agir sur le contenu du composant : ajout, suppression de sous-composants, le *contrôleur de liaison* (*binding-controller*) offre des opérations pour lier les interfaces requises d'un composant aux interfaces fournies d'un autre composant, le *contrôleur d'attributs* (*attribute-controller*) offre des opérations de modification de propriétés des composants, et enfin le *contrôleur de cycle de vie* (*lifecycle-controller*) offre des opérations permettant de démarrer ou d'arrêter le logiciel encapsulé. Les opérations de ces différents contrôleurs possèdent une implémentation générique, que le développeur de composants peut enrichir ou redévelopper selon ses propres besoins.

L'administration d'un tel système se fait ensuite en configurant sa *représentation duale* en Fractal, appelée aussi *System Representation* (SR), à l'aide des interfaces de contrôle Fractal. Le lifecycle-controller permet de contrôler l'état du logiciel encapsulé, démarré ou arrêté.

L'architecture de la représentation duale d'un système est exprimée à l'aide d'une version étendue de Fractal ADL. Un exemple de descripteur pour un cas simple mettant en œuvre un serveur Apache et un serveur Tomcat reliés entre eux est représenté sur le listing 2.6

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN" "classpath://
    org/objectweb/jasmine/jade/service/deployer/adl/xml/jade.dtd">

<definition name="Apache-Tomcat">
  <interface name="service" role="server" signature="org.objectweb.jasmine.jade.
      service.Service" />
  <component name="start" definition="org.objectweb.jasmine.jade.resource.start.
      StartType">
    <controller desc="primitive"/>
    <virtual-node name="node1"/>
  </component>
  <!--                APACHE              -->
  <component name="apache" definition="fr.jade.resource.j2ee.apache.
      ApacheResourceType">
    <generic-attributes>
      <attribute name="resourceName" value="apache" />
      <attribute name="dirLocal" value="/tmp/j2ee" />
      <attribute name="user" value="jlegrand" />
      <attribute name="group" value="wheel" />
      <attribute name="port" value="9090" />
```

```
      <attribute name="serverAdmin" value="julien.legrand@inrialpes.fr" />
    </generic-attributes>

    <controller desc="primitive"/>

    <virtual-node name="node1" />
    <packages>
      <package name="Apache v1.3.29 linux x86 Wrapper">
        <property name="local.dir" value="/tmp/j2ee" />
      </package>
    </packages>
  </component>

  <!--                TOMCAT              -->
  <component name="tomcat" definition="fr.jade.resource.j2ee.tomcat.
      TomcatResourceType">
    <generic-attributes>
      <attribute name="resourceName" value="tomcat" />
      <attribute name="dirLocal" value="/tmp/j2ee" />
      <attribute name="javaHome" value="/usr/local/java/jdk1.5.0_05" />
      <attribute name="workerPort" value="8009" />
    </generic-attributes>
    <controller desc="primitive"/>
    <virtual-node name="node2" />
    <packages>
      <package name="Tomcat Wrapper">
        <property name="local.dir" value="/tmp/j2ee" />
      </package>
    </packages>
  </component>

  <!--              BINDING               -->
  <binding client="this.service" server="start.service" />
  <binding client="start.rsrc_apache" server="apache.resource" />
  <binding client="start.rsrc_tomcat" server="tomcat.resource" />
  <binding client="apache.worker1" server="tomcat.resource" />
  <controller desc="composite"/>
  <virtual-node name="node1" />

</definition>
```

Listing 2.6 – Extrait de description d'architecture de représentation duale Jade

L'attribute-controller permet de modifier des paramètres du logiciel encapsulé, le nom d'un serveur ou le port ouvert. Enfin, le contrôleur de liaison permet de modifier les connexions entre logiciels. Par exemple, la connexion entre un serveur Apache et un serveur Tomcat se fait en fixant des propriétés dans le fichier `worker.properties` du serveur Apache. Établir une connexion entre une instance de composant réifiant Apache et une instance réifiant Tomcat —*i.e.* appeler l'opération correspondant à la liaison sur l'interface de contrôle de liaison— signifie reconfigurer le `worker.properties` pour modifier la propriété correspondante.

En exposant l'administration du système via sa représentation duale et les interfaces de contrôle de Fractal, Jade rend possible une administration autonome, en laissant ces interfaces aux mains de programmes, au lieu d'humains, et rendre ainsi l'administration du système autonome. Pour cela, les auteurs proposent d'implémenter la boucle de contrôle de l'informatique auto-gérée, à l'aide de composants Fractal. Ainsi certains composants, appelés *senseurs*, sont chargés d'observer des changements, et d'appeler des composants *réacteurs* encapsulant les

phases d'analyse et de planification de la boucle de contrôle. Enfin les composants *actuateurs* sont chargés d'aller exécuter les reconfigurations en appelant des opérations des contrôleurs des composants de la représentation duale. Ce mécanisme d'autonomie, schématisé sur la figure 2.26, a été utilisé à différentes fins, comme l'auto-optimisation [70], l'auto-protection [21], ou encore l'auto-dimensionnement [65] de logiciels.

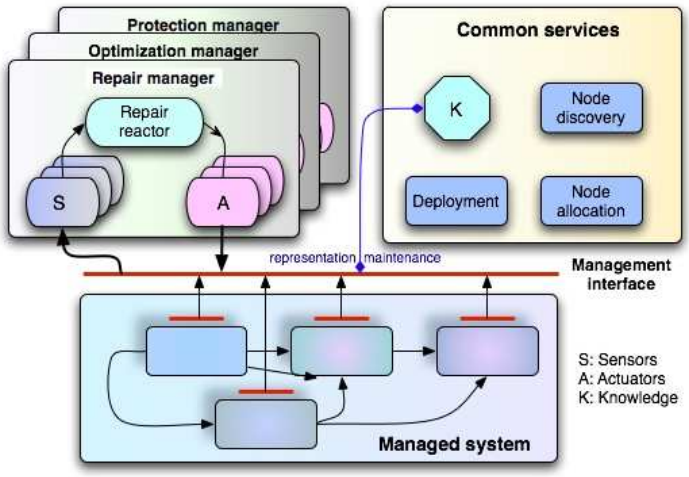

FIG. 2.26 – Architecture des boucles de contrôle dans Jade

Synthèse Le principe de représentation de l'architecture du système sous forme de composants Fractal constitue une approche efficace et novatrice. En effet, elle permet d'abstraire la notion de système, pour la représenter sous forme de composants en interaction. Le choix du modèle Fractal pour les composants est doublement judicieux. Les composants Fractal sont légers en terme d'exécution, et offrent des mécanismes de reconfiguration dynamique. L'architecture de composants offre un mécanisme de réflexivité pour l'administration des systèmes distribués.
Néanmoins, cette représentation à base de composants est loin d'être facile à mettre en œuvre. L'abstraction choisie, le paradigme de composant, est éloigné de la préoccupation de déploiement. Ainsi il faut assimiler les concepts d'architectures à composants, en plus des concepts spécifiques de Fractal, pour déployer et administrer son système.

Aucune vérification n'est faite, ni sur la représentation duale et son déploiement, ni sur l'exécution des boucles de contrôle.

En outre, le développement des composants encapsulant les logiciels du système sont loin d'être triviaux. Il faut connaître le modèle de programmation Fractal, mais également coder explicitement —en Java pour l'implémentation existante de Jade— les communications avec les machines hôtes. Cela présente le double inconvénient de : mélanger le code de communication avec le code d'administration, et de contraindre le protocole de communication utilisé, puisque codé en dur dans le composant. Si l'architecture de Jade est très efficace pour abstraire l'architecture du système et la reconfigurer dynamiquement, elle est très difficile à mettre en

œuvre pour un administrateur n'ayant pas de solides connaissances en composants logiciels.

Enfin, les environnements ouverts distribués ne sont pas gérés dans Jade.

2.5.5 TUNe

TUNe [17][16] est une extension du framework Jade proposée par le laboratoire IRIT de Toulouse, et a été mené en parallèle avec les travaux de thèse présentés dans ce manuscrit.

Ce travail est fondé sur le constat que le canevas Jade propose des mécanismes de trop bas niveau pour encapsuler des logiciels patrimoniaux et les administrer durant l'exécution. En effet, l'utilisation de Jade exige de connaître et de savoir mettre en oeuvre une architecture à base de composants Fractal. TUNe propose une approche dirigée par les modèles pour décrire les *wrappers* (composant d'encapsulation) de logiciels patrimoniaux, et décrire un système logiciel à déployer. Les auteurs proposent : 1) un langage XML de description des wrappers de logiciels 2) un profil UML pour décrire le déploiement d'un système ainsi que les politiques d'administration autonome.

Comme nous l'avons vu dans Jade, l'administration d'un logiciel patrimonial se fait en appelant des opérations sur les interfaces du composant Fractal jouant le rôle de wrapper. Cela implique de savoir implémenter un composant Fractal. Dans TUNe, un langage de description d'interface d'administration, appelé WDL (pour *Wrapper Definition Language*) permet de définir des ensembles d'opérations pouvant être appelées pour configurer ou reconfigurer le logiciel encapsulé. Un exemple de descripteur écrit à l'aide de la syntaxe WDL est représenté sur le listing 2.7.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<wrapper name="sed">
  <method name="start" key="appli.wrapper.util.GenericStart" method="
      start_with_pid_linux">
    <param_value="$dirLocal/$progName $dirLocal/$compName-cfg $arguments"/>
    <param_value="LD_LIBRARY_PATH=$dirLocal"/>
  </method>

  <method name="configure" key="appli.wrapper.util.ConfigurePlainText" method="
      configure">
    <param_value="$dirLocal/$compName-cfg"/>
    <param_value="traceLevel:$traceLevel"/>
    <param_value="parentName:$LA.compName"/>
    <param_value="name:$compName"/>
    <param_value="lsOutbuffersize:$lsOutbuffersize"/>
    <param_value="lsFlushinterval:$lsFlushinterval"/>
  </method>

  <method name="stop" key="appli.wrapper.util.GenericStop" method="
      stop_with_pid_linux">
    <param_value="$PID"/>
  </method>
</wrapper>
```

Listing 2.7 – Extrait de définition de wrapper Tune en WDL

Le profil UML proposé par TUNe comporte deux grandes sections. La première permet d'écrire des schémas de déploiement, c.-à-d. quel logiciel déployer, ainsi que les communications entre logiciels. Un tel schéma est représenté sous la forme de diagramme de classes UML, et est représenté sur la partie gauche de la figure 2.27. Il définit un schéma de déploiement pour une infrastructure DIET telle qu'étudiée en section 2.4.4. Ce schéma définit un MA, deux LA par MA, ainsi que 5 SeD par LA, le tout pour un total de 1 MA, 2 LA et 10 SeD. Dans ce schéma est décrit également le déploiement de trois types sondes (`probeMA`, `probeLA` et `probeSeD`), sous forme de logiciel encapsulé également, à raison d'une sonde par logiciel déployé. Les instances représentées sur le schéma de déploiement comportent des propriétés qui peuvent être propres au logiciel déployé, mais certaines autres propriétés sont exploitées par TUNe pour automatiser la génération de wrappers Fractal ainsi que leurs interconnexions pour former une architecture similaire à celle de Jade. Les propriétés propres au logiciel servent de paramètres aux commandes définies dans le fichier WDL correspondant au type de logiciel.

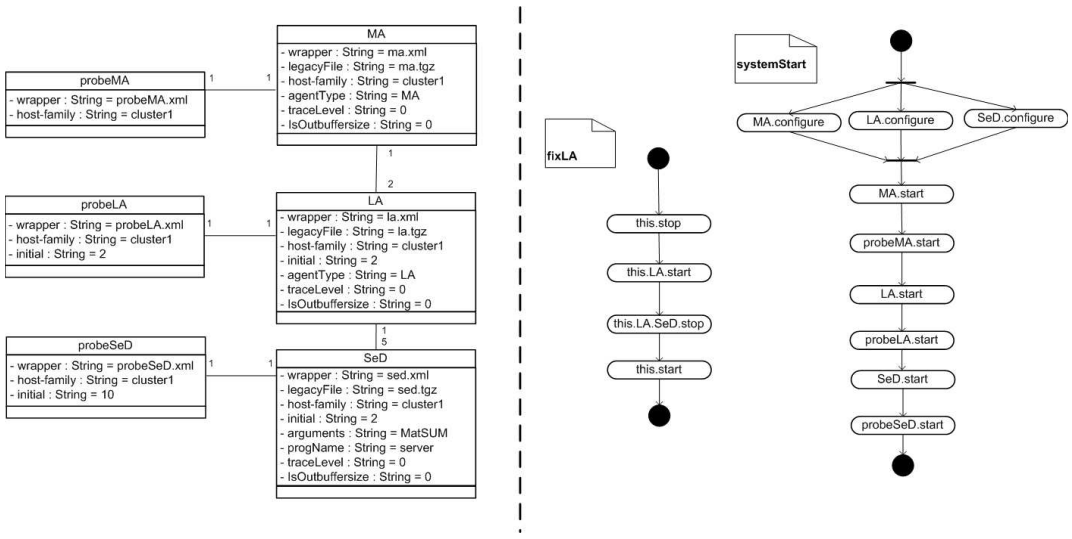

FIG. 2.27 – Schéma de déploiement (gauche) et diagrammes de reconfiguration (droite) dans TUNe

Enfin pour gérer l'administration de manière autonome, il est possible de définir des diagrammes d'états-transitions d'UML pour spécifier l'enchaînement des appels de procédures sur un logiciel donné. Les reconfigurations sont effectuées en fonction d'événements typés. L'enchaînement des actions correspondant à la reconfiguration est représenté sur la partie droite de la figure 2.27. Deux diagrammes sont représentés. Sur le premier d'entre eux est détaillée la séquence d'opérations à appeler dans le cas d'une panne d'un LA. Les actions à effectuer, exprimées du point de vue de la sonde ayant observé la panne, sont les suivantes : arrêter la sonde (pour éviter la notification multiple de la panne), redémarrer le LA à l'origine de la panne, et arrêter tous les SeD attachés au LA venant d'être redémarrés (leur propre sonde les fera redémarrer ensuite, et ils s'enregistreront à nouveau auprès du LA). Le deuxième diagramme d'états décrit le démarrage du système complet. Il faut d'abord configurer, de manière paral-

lèle, tous les MA, les LA et les SeD. La fin de la séquence consiste à démarrer, de manière séquentielle, les MA et leurs sondes, les LA et leurs sondes et enfin les SeD et leurs sondes.

Synthèse TUNe est incontestablement l'approche la plus aboutie en terme de déploiement et d'autonomie des systèmes distribués. Il bénéficie des avantages de Jade en résolvant son principal défaut, à savoir le manque d'abstraction fournie par Jade. Avec le profil UML, TUNe offre un moyen d'expression à plus haut niveau pour exprimer le schéma de déploiement de systèmes distribués, ainsi que les boucles de contrôle régissant les reconfigurations éventuelles des logiciels.
Néanmoins certains choix dans la démarche de modélisation proposée par les auteurs sont obscurs. Modéliser la représentation duale à l'aide d'un diagramme de classes n'est pas justifié, puisqu'elle dénature l'utilité initiale du diagramme de classe UML, utilisé ici à la place d'un diagramme d'instance. De plus, le méta-modèle utilisé dans TUNe nous paraît trop limité pour représenter n'importe quel système logiciel : pas d'expression de dépendances entre logiciels, pas de paradigme d'expression d'architecture de composants métiers s'exécutant au dessus des serveurs d'applications encapsulés.

Qui plus est, TUNe possède quelques autres limitations. Il y a dans TUNe une absence totale de vérification comportementale sur les opérations d'administration définies. En effet dans la mesure où ces opérations sont susceptibles d'être appelées automatiquement suite à une panne par exemple, un minimum de propriétés devraient être assurées sur les logiciels. L'une des vérifications pertinentes est par exemple de vérifier que l'opération `stop` est bien l'exacte opposée de l'opération `start`, pour assurer que la séquence `start/stop/start/`*etc.* ne produit pas d'effet de bord.

TUNe ne permet d'exprimer que la structure des logiciels déployés. La manière de les déployer, c'est-à-dire les instructions élémentaires à effectuer pour déployer les logiciels en question, est enfouie dans une classe Java. Le processus de déploiement d'un logiciel dans TUNe est donc une *boîte noire*, dans le sens où elle doit être écrite en Java, même si les opérations à appeler sont composées à plus haut niveau dans le fichier WDL.

En conclusion, TUNe qui est une approche se développant en parallèle des travaux présentés dans cette thèse, représente l'approche la plus aboutie de cet état de l'art en terme de déploiement et d'autonomie. Néanmoins l'approche nous paraît encore trop incomplète et immature pour fournir un outil générique de déploiement de systèmes logiciels complet en environnements ouverts distribués. Dans des travaux récemment publiés [15], un langage dédié reposant sur un méta-modèle TUNe remplace les diagrammes UML présentés dans cette section.

2.5.6 Synthèse sur la reconfiguration dynamique

La reconfiguration dynamique est initialement une étape du cycle de vie du déploiement. Or, dans les approches étudiées, la volonté est de découpler la préoccupation de reconfiguration dynamique du déploiement. Dans ces approches, le déploiement des logiciels est considéré comme déjà effectué, et la reconfiguration est exprimée séparément du déploiement (hormis pour Jade et Tune). En outre, les entités manipulées dynamiquement par les mécanismes de reconfiguration se situent exclusivement au niveau métier. À l'inverse, une approche comme

TUNe ne propose une reconfiguration qu'au niveau des serveurs d'applications ou librairies, mais pas aux composants métiers qui s'exécutent par dessus ces serveurs. Il en résulte que les procédures de reconfiguration/adaptation actuelles sont inadaptées pour la reconfiguration de systèmes logiciels complets. Enfin on constate que la problématique des environnements ouverts distribués et des vérifications comportementales des procédures de reconfiguration dynamiques sont totalement omises.

2.6 Synthèse de l'état de l'art

L'étude des quatre catégories de travaux que nous avons faite dans ce premier chapitre nous donne un aperçu assez complet de la mise en œuvre actuelle de la procédure de déploiement. La synthèse de cette étude est représentée dans le tableau de la figure 2.1. Les lignes de ce tableau sont les approches étudiées et les colonnes les critères d'études. Une case vide dans le tableau signifie que le critère n'est pas traité dans les approches. Une case à fond coloré signifie que la solution proposée parait satisfaisante pour le critère considéré, et il est intéressant de s'inspirer de ces mécanismes ou concepts. Une case blanche contenant du texte signifie que des éléments de solution intéressants sont fournis par le travail, mais que le tout reste insuffisant. Le constat qui en résulte, qui met en valeur les lacunes dans la gestion du déploiement logiciel actuel, se découpe en quatre grandes idées, qui vont correspondre aux 4 grands défis que nous souhaitons relever dans cette thèse.

Le premier point concerne les langages de déploiement. Il apparaît un clair manque de consensus autour du terme de déploiement. En effet, certains travaux ne concernent en fait que le déploiement de la couche métier d'un système logiciel (ADL, CoSMIC/DAnCE, etc.). D'autres ne concernent que les couches inférieures (serveur d'applications, bibliothèques, etc.), sans concerner la couche métier (Jade, TUNe). De plus concernant les couches métiers, chaque technologie qui apparaît conduit à la définition d'un outil de déploiement, ou d'un paradigme d'expression des architectures de cette technologie. À chaque fois qu'un administrateur doit déployer un nouveau système, il doit apprendre le nouveau modèle d'architecture et la nouvelle démarche de déploiement. Ainsi un premier défi de déploiement à relever, consiste à établir une **abstraction de déploiement entièrement générique**. Ce modèle ou langage de déploiement doit permettre d'**exprimer le déploiement de tout type de logiciel quelles que soient les propriétés de la machine hôte qui les héberge**. Les logiciels seront décrits de manière homogène quelle que soit leur position dans la pile logicielle et quelle que soit leur technologie. Ce langage doit comprendre tous les concepts nécessaires à la description de la pile logicielle, et ce de manière totalement générique. Enfin ce langage doit permettre de décrire non seulement le système à déployer, mais également de spécifier les actions élémentaires à mener pour réaliser le déploiement, c'est-à-dire **n'importe quelle étape du cycle de vie du déploiement**.

	Couverture du cycle de vie	Abstraction choisie pour expression	Vérifications	Automatisation	Reconfiguration / Adaptation	Environnements ouverts distribués	Hétérogénéités technologies / matérielle
Darwin	*Configuration / démarrage*	*Composants seulement*		*Projection vers une seule plate-forme*	*Constructions dynamiques*		*Indépendant des technologies de composants*
Fractal ADL	*Installation / configuration / démarrage*	*Composants seulement*		*Déploiement Fractal*			
SafArchie	*Conception seulement*	*Composants + Contrats*	*contrats structure, comportement et syntaxe*	*Projections Fractal ou ArchJava*			*Indépendant des technologies de composants*
UML Déploiement	*Conception seulement*	*Modèle à sémantique très floue*					*Indépendant des technologies des paradigmes et du matériel*
ORYA	*Tout sauf reconfiguration*	*Logiciel à gros grain et quelques simplifications*		*Projection difficile*			*Indépendant technologies et matériel mais composant*
OMG D&C	*Tout sauf reconfiguration*	*Composant seulement et trop riche + topologie*		*Définition d'un PIM là projeter en PSM*			*Indépendant des technologies à composants*
GADE	*Planification*	*Minimaliste, applicable principalement pour grilles*		*Plan généré mais projection ADAGE*			*Indépendant des technologies*
Software Dock	*Couverture intégrale*	*Tout logiciel applicatif - Pas de composant métier*	*Déploiement adaptatif*	*Plate-forme distribuée à base d'agents*	*Mécanisme de mise à jour*		*Modèle matériel = interfaces Logiciels applicatifs seulement*
ProActive	*Couverture intégrale*	*Spécifique GCM / Java description topologie matériel*		*Plate-forme IC2D*	*Intégrale mais manuelle*		*Hétérogénéité matérielle seulement gérée*
GoDIET	*Installation / configuration / démarrage*	*spécifique DIET*	*Détection dynamique de fautes et tolérance*	*Outil dédié*			*Hétérogénéité matérielle seulement gérée*
Serveurs d'applications	*Installation / configuration / démarrage / arrêt*	*descripteur XML dédié*		*Complète mais dédiée*			
DAnCE/CoSMIC	*Installation / configuration démarrage*	*idem OMG D&C avec configuration d'intergiciel*		*Outil dédié*			
Belguidoum et al.	*Installation / configuration / démarrage*	*Composant avec contrats*	*Vérifications formelles de faisabilité du déploiement*	*Outil dédié qui effectue les vérifications*			*Indépendant de technologie composants*
SAFRAN	*Configuration / démarrage*	*Composant + Aspect Reconfigurations Fscript*	*Assurance de consistance des reconfigurations*	*Moteur d'exécution Fscript*	*Aspects d'adaptation*		
MADAM	*Installation / configuration / démarrage*	*Composants, lignes de produits et fonctions d'utilité*	*Adaptation du déploiement au contexte*			*Adaptation terminaux inconnus*	
Jade	*Installation / configuration / démarrage*	*Composants Fractal pour représentation duale*		*Besoin d'écrire wrappers*	*Boucles de contrôle*		*Indépendant du logiciel déployé mais pas du matériel*
Tune	*Couverture intégrale*	*Profil UML et langage de description des wrappers*		*Besoin d'écrire wrappers*	*Boucles de contrôle*		*Indépendant du logiciel déployé mais pas du matériel*

TAB. 2.1 – Tableau de synthèse de l'état de l'art

En outre, concernant les modèles de déploiement, un choix semble devoir être fait entre sémantique des modèles et généricité. Les modèles génériques que nous avons étudié manquaient de sémantique pour certains (UML par exemple) et pour d'autres la généricité était restreinte (GADE orienté technologies Grid Computing, ou OMG D&C orienté composants métiers). Le second défi concerne donc également le langage, et plus précisément la sémantique du langage de déploiement. Nous l'avons vu, peu d'approches de déploiement logiciel pur mettent en œuvre des mécanismes de **vérifications comportementales statiques**. Il apparait d'ailleurs clairement dans le tableau que ce thème est très peu évoqué dans les approches étudiées. Pourtant à de grandes échelles ou en environnement ambiant, l'échec d'un déploiement peut avoir des conséquences désastreuses pour un administrateur système. En effet, si l'on se place dans un contexte Grid Computing, un déploiement qui échoue fait échouer la réservation d'un ensemble de ressources, ce qui est très regrettable au vu du temps parfois attendu pour obtenir ces ressources. Le deuxième défi consiste donc à ajouter un maximum de sens aux concepts du langage générique de déploiement, de manière à pouvoir valider statiquement la procédure globale de déploiement, et éviter un maximum d'erreurs ou d'incohérences durant l'exécution du déploiement.

Dans notre étude, nous n'avons rencontré quasiment aucune approche destinée aux environnements ouverts distribués. La plupart des outils de déploiement existants considèrent le réseau de machines comme étant figé, sans laisser place à d'éventuelles entrées et/ou sorties de machines dans le réseau. Les outils de reconfiguration dynamique ne prennent pas non plus en compte l'ouverture des environnements, ce qui freine l'utilisation de telles techniques dans le cadre d'un déploiement en environnements ouverts. Un défi supplémentaire consiste dès lors à réconcilier l'étape de reconfiguration dynamique avec le reste de la procédure de déploiement. En effet, cette préoccupation de **reconfiguration dynamique** doit, si elle se trouve au même plan que la phase de déploiement, permettre de **décrire un déploiement adaptatif spécifique aux environnements ouverts distribués**. Il s'agit dans ce cas de reconfigurer (et notamment d'ajouter) des logiciels sur les machines venant d'entrer sur le réseau, et pas seulement d'adapter les composants métiers.

Enfin, le dernier défi que nous tenons à relever dans cette thèse porte sur **l'automatisation de l'exécution du déploiement** en fonction de sa description à haut niveau. En effet, les divers outils de déploiement que nous avons étudié proposent un modèle spécifique à leur technologie. L'avantage immédiat de la spécificité réside dans la possibilité de transformer les concepts du modèle de déploiement en appels sur l'API de déploiement associée. Ainsi nous souhaitons que la définition d'un langage générique de déploiement n'empêche pas cette génération automatique. Il est alors nécessaire de réaliser une plate-forme d'exécution de déploiement générique pour effectuer les différentes opérations de déploiement en environnements ouverts distribués.

Ces quatre défis seront davantage explicités dans les chapitres qui suivent, ainsi que notre contribution pour relever ceux-ci inhérents à la procédure de déploiement logiciel. Cette contribution, appelée DeployWare/DACAR, vise à mettre en œuvre une démarche dirigée par les modèles pour le déploiement de systèmes logiciels distribués en environnements ouverts.

Troisième partie

Contribution

Chapitre 3

Vue d'ensemble de la contribution

Sommaire

Dans ce chapitre, nous exposons notre contribution pour le déploiement de systèmes distribués en environnements ouverts. L'objectif global est de fournir une approche permettant de décrire et d'exécuter le déploiement d'un système logiciel dans sa totalité. Cela signifie que nous souhaitons concevoir une approche dans laquelle il est possible de décrire le déploiement de la couche métier d'une application, c.-à-d. un assemblage d'instances de composants distribué sur plusieurs machines. Mais le déploiement d'un système logiciel ne se résume à ce seul déploiement de composants métiers. Le support d'exécution des différents composants, sous la forme de serveurs d'applications ou de serveurs de composants, doit également être déployé en amont. Le déploiement de ces différents serveurs peut lui-même nécessiter le déploiement d'un certain nombre de bibliothèques.

Nous choisissons de traiter séparément ces deux couches logicielles dans une approche en deux parties : DeployWare pour le déploiement de la couche intergicielle et DACAR pour le déploiement de la couche métier. En effet, les concepts manipulés pour le déploiement d'une architecture à base de composants et ceux pour le déploiement de la couche intergicielle sont différents. Du côté architecture métier, il est question d'instances de composants, de leurs propriétés et de liaisons entre les différents ports des composants, alors que du côté couche intergicielle il est question de logiciels, de leurs propriétés et des dépendances entre ces logiciels. De plus, pour le déploiement des logiciels de la couche intergicielle d'un système logiciel, il est nécessaire de prendre en compte de nombreux paramètres tels que le protocole de transfert de fichiers des machines (*e.g.* SCP, FTP), le protocole d'accès (*e.g.* SSH, Telnet), ou encore le langage d'interpréteur de commandes (*e.g.* SH, Shell Windows). La séparation en deux approches s'explique notamment par la grande différence entre les actions élémentaires du déploiement

de la couche intergicielle et celles du déploiement de la couche métier. En effet, les actions élémentaires du déploiement de la couche intergicielle s'expriment en actions directes sur les machines (en utilisant un langage d'interpréteur de commandes). De l'autre côté, le déploiement de composants logiciels s'effectue à l'aide d'opérations élémentaires sur l'API du modèle choisi, donc d'opérations sur les serveurs de composants, qui représentent une couche d'abstraction au dessus des machines. Une possibilité de rapprochement entre les deux approches n'est néanmoins pas exclue, elle est discutée dans les perspectives de ce travail.

Notre point de vue initial est que dans notre approche de déploiement globale, une seule et unique personne ne peut être responsable de tous ces paramètres de déploiement. Nous décidons donc de scinder notre approche en différents rôles que nous définissons dans cette vue d'ensemble avant de rentrer dans le détail de ces rôles dans les chapitres qui suivent. Notre étude du déploiement nous amène à considérer au moins quatre rôles distincts que sont l'*expert réseau*, l'*expert logiciel*, l'*administrateur système* et enfin l'*architecte métier*. Ce découpage en rôles ainsi que les interactions entre ces rôles est représenté sur la figure 3.1. Dans un premier temps, nous décrivons l'action des différents rôles, indépendamment de tout choix technologique pour réaliser cette action. Dans un second temps, nous décrivons les approches que nous proposons pour ces rôles.

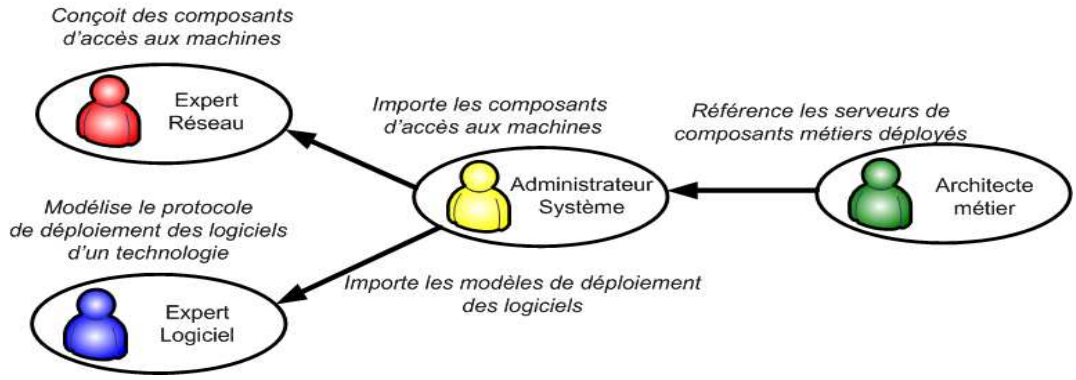

FIG. 3.1 – Rôles du déploiement et les interactions entre eux

3.1 Expert réseau

Tout d'abord, nous définissons le rôle d'*expert réseau* comme le responsable des communications avec les machines du domaine de déploiement. Il s'agit d'un rôle de très bas niveau, proche des considérations réseau. La figure 3.2 schématise les actions de ce rôle. L'expert réseau possède une connaissance accrue des protocoles réseaux et des bibliothèques de programmation qui réalisent ces protocoles. Sa mission est donc d'écrire des adaptateurs réutilisables pour chaque technologie de communication nécessaire pour le déploiement. Pour guider son action des interfaces génériques sont établies pour chacune des grandes fonctions de la communication à distance : le protocole d'accès, le protocole de transfert de fichiers, le langage d'interpréteur de commandes. Il implémente ses différentes interfaces à l'aide des bibliothèques

existantes pour ces différentes fonctions : SSH ou Telnet pour le protocole d'accès, SCP ou FTP pour le protocole de transfert de fichiers, SH ou CSH pour le langage d'interpréteur. Ces différents adaptateurs sont ensuite stockés dans un référentiel afin d'être réutilisés pour décrire les machines hôtes d'un domaine. Les moyens pour l'expert réseau sont détaillés dans le chapitre 5.

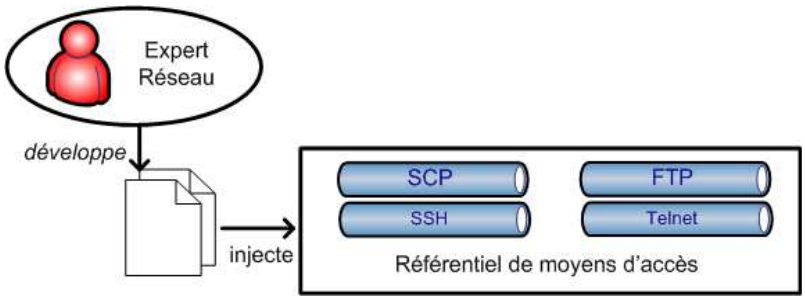

FIG. 3.2 – Rôle de l'expert réseau

3.2 Expert logiciel

Le second rôle que nous définissons est celui d'*expert logiciel*. La figure 3.3 schématise les actions de ce rôle. Un expert logiciel est une personne à forte compétence pour une technologie donnée. Il est capable, pour cette technologie, de définir précisément le protocole de déploiement, c'est-à-dire les différentes actions élémentaires à mener sur une machine hôte, pour y exécuter les différentes étapes du cycle de vie du logiciel de la technologie qu'il maîtrise. Ces actions sont exprimées en termes de chargement de fichiers à distance, de positionnement de variables d'environnement, de lancement d'exécutables, etc. L'objectif de l'expert logiciel est de fournir une forme exécutable et réutilisable de la description de son logiciel, qu'il publie dans un référentiel, pour permettre à l'administrateur système d'en paramétrer et déployer une instance, sans avoir à reproduire manuellement une à une les actions de déploiement.
Les actions élémentaires sont exprimées de manière générique par l'expert logiciel. La réalisation concrète de ces actions est déterminée en fonction des propriétés de la machine sur laquelle elles doivent être exécutées. Par ailleurs, nous souhaitons ajouter un certain nombre de vérifications statiques sur les protocoles de déploiement de logiciel décrits par l'expert logiciel. Ces vérifications concernent d'abord la complétude des dépendances décrites pour un logiciel donné, c.-à-d. vérifier qu'une dépendance n'est pas déclarée abusivement ce qui signifierait que le logiciel pourrait en fait fonctionner sans cette dépendance. Le second type concerne la réversibilité du déploiement, c.-à-d. vérifier que le logiciel peut être complètement supprimé après avoir été déployé. En d'autres termes, il s'agit de vérifier qu'une séquence déploiement/-repliement n'introduit pas d'effet de bord sur la machine concernée. Le langage pour l'expert logiciel ainsi que les vérifications qu'il est possible de faire sur ce langage sont discutées en détails dans le chapitre 4.

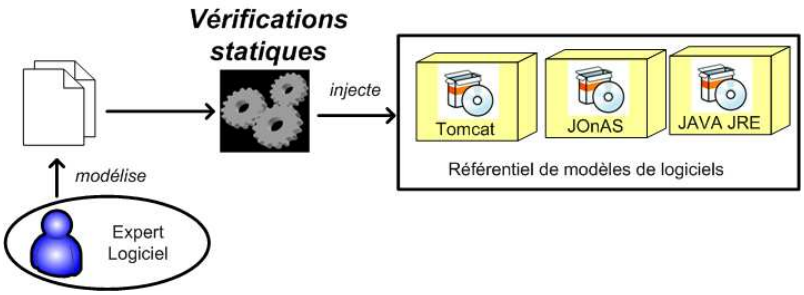

FIG. 3.3 – Rôle de l'expert logiciel

3.3 Administrateur système

Le troisième rôle de notre découpage est celui d'*administrateur système*. La figure 3.4 schématise les actions de ce rôle. Ce rôle consiste à décrire l'architecture d'un système logiciel, mais aussi de fixer les propriétés des machines du domaine de déploiement sur lequel le déploiement va s'effectuer. Cette architecture peut comporter n'importe quel élément de la *couche intergicielle* de la pile logicielle, c'est-à-dire n'importe quel logiciel, à l'exception des composants métiers (car la description de ces derniers est réservée à un autre rôle). L'administrateur système récupère à la fois les productions de l'expert réseau et de l'expert logiciel. En effet, il doit, dans un premier temps, décrire les moyens d'accès aux machines qui composent son domaine de déploiement. Cela revient à déterminer quels adaptateurs (implémentés par l'expert réseau) utiliser pour communiquer avec la machine. Dans un second temps, il définit des instances des logiciels (décrits par les experts logiciels), il configure ces instances, puis les assigne à une machine hôte qu'il a précédemment décrite. Des vérifications statiques sont également effectuées sur les modèles de l'administrateur système. Ces vérifications servent par exemple à vérifier qu'une instance de logiciel est bien conforme à la définition de son type défini par l'expert logiciel (conformité des propriétés, des dépendances, etc.). Une machine virtuelle DeployWare permet ensuite de faire le déploiement, conformément aux actions élémentaires définis par les experts logiciels, et aux protocoles d'accès aux machines définies par l'expert réseau.

En plus de la configuration statique de son système logiciel, l'administrateur système peut décrire des politiques d'auto-gestion, pour notamment prendre en charge dynamiquement le déploiement de logiciels sur des machines faisant leur apparition dans le domaine de déploiement, comme c'est le cas dans les environnements ouverts distribués. Le langage de l'administrateur système, les vérifications qu'il est possible de faire sur ce langage sont présentés dans le chapitre 4

3.4 Architecte métier

Enfin, le dernier rôle de notre découpage est celui d'*architecte métier*. La figure 3.5 schématise les actions de ce rôle. Il est en charge de la couche la plus haute d'un système logiciel, celle des composants métiers. Nous choisissons le paradigme de programmation par compo-

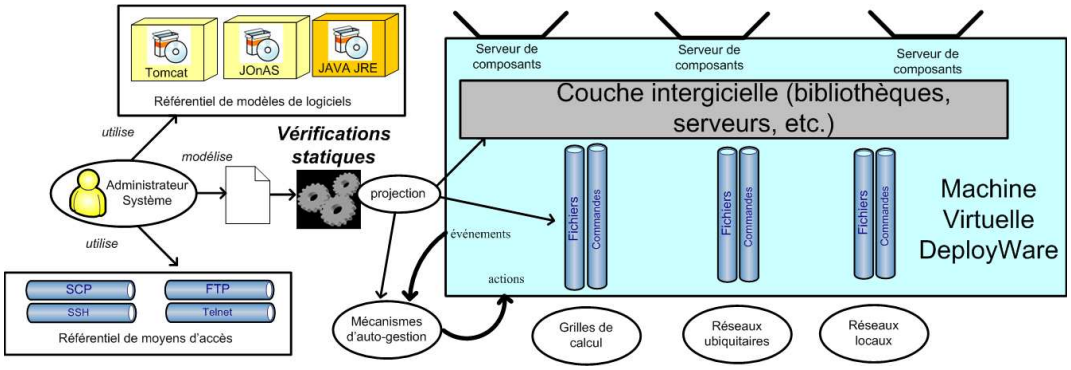

FIG. 3.4 – Rôle de l'administrateur système

sants pour la couche métier du logiciel. En effet, ce paradigme est parmi l'un des plus utilisés pour la construction d'applications réparties. Le travail de l'architecte métier repose sur celui de l'administrateur système. En effet, dans un premier temps, l'administrateur système définit le déploiement de tous les logiciels nécessaires au démarrage de serveurs de composants. Dans un second temps, l'architecte métier décrit la structure de son application métier, ainsi que l'affectation de chacun des composants de son application métier sur les serveurs démarrés par l'administrateur système. La structure de l'application métier est transformée en description d'architecture persistante pendant l'exécution. Ainsi, comme l'administrateur système, l'architecte métier peut définir pour son architecture des politiques d'autonomie, dans le but d'adapter dynamiquement le déploiement de composants sur les machines joignant le domaine de déploiement au fil du temps.

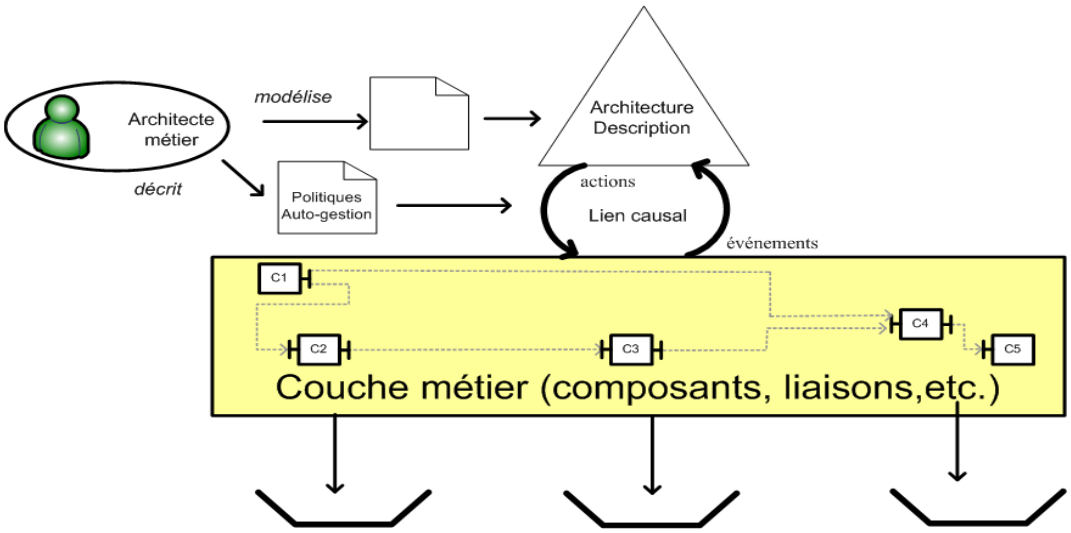

FIG. 3.5 – Rôle de l'architecte métier

3.5 Contribution : DeployWare / DACAR

Notre contribution consiste en une approche orientée modèles, pour les quatre rôles que nous venons de définir. Le schéma global de notre contribution est représenté sur la figure 3.6. Cette approche se décompose en deux grandes parties : pour le déploiement de l'architecture métier et pour le déploiement de la couche intergicielle. La première partie, DeployWare, est une approche à base de modèles pour la description de la couche intergicielle, c'est-à-dire des bibliothèques de base jusqu'aux serveurs de composants d'un système logiciel. Cette partie concerne les experts réseaux et logiciels ainsi que l'administrateur système. La seconde partie, DACAR (pour *Distributed Autonomic Component-based ARchitectures*), est une approche à base de modèles pour applications auto-gérées à base de composants répartis. Cette seconde partie est à l'usage des architectes métier.

Dans un premier temps, nous proposons un méta-modèle DeployWare qui définit des concepts pour l'expression du déploiement de la couche intergicielle. Ce méta-modèle permet d'exprimer **l'intégralité des étapes du cycle de vie** de chaque logiciel qui compose cette couche intergicielle. Le contenu, en terme d'actions élémentaires, de chacune de ces étapes sera également exprimable à l'aide du méta-modèle, et ce de manière générique. Même si la tâche de l'expert réseau reste essentiellement technique par définition, l'utilisation de ce méta-modèle propose une **montée en abstraction** pour l'expert logiciel ainsi que l'administrateur système. Le découpage du processus de déploiement des logiciels en actions élémentaires génériques rend possible la description de n'importe quel logiciel, et sur n'importe quel type de machines. L'**hétérogénéité** des machines hôtes et des logiciels de la couche intergicielle est ainsi assurée à l'aide de ce méta-modèle générique.

En outre, la définition de la structure du langage à l'aide d'un méta-modèle rend possible la **vérification statique** de propriétés des logiciels et systèmes ainsi décrits par les experts logiciels et les administrateurs systèmes. En effet, nous ajoutons un certain nombre de constructions dans le méta-modèle qui permettent notamment de lier des actions élémentaires de déploiement antagonistes, ou encore de lier des actions élémentaires aux logiciels qui les fournissent. La première vérification rendue possible est celle qui consiste à vérifier que le déploiement de tout logiciel est bel et bien réversible intégralement, ce qui signifie que toutes les actions entreprises pour déployer le logiciel sont bien annulables. Le second type de vérification statique consiste à vérifier la cohérence des dépendances d'un logiciel. Cela signifie vérifier que toutes les dépendances déclarées pour un logiciel, fûrent-elles techniques ou métier, sont pertinentes —*i.e.* le logiciel à déployer dépend il réellement d'un autre logiciel ? À l'inverse, il s'agit de vérifier que si un logiciel B fournit une action présente dans le protocole de déploiement d'un logiciel A, alors B est bien présent dans les dépendances de ce logiciel A. Les deux derniers types de vérification que nous mettons en œuvre concernent les systèmes logiciels décrits par l'administrateur système. La première consiste à vérifier qu'avant de déployer un logiciel sur une machine hôte, les dépendances de ce logiciel ont bien été déployées en amont. Enfin la dernière vérification consiste à vérifier, pour deux logiciels installés sur une même machine hôte, que les ressources utilisées par ces logiciels n'interfèrent pas (par exemple, il s'agit de vérifier que deux logiciels n'attendent pas de connexions sur le même port). L'ensemble de

ces vérifications, rendues possibles par le méta-modèle, permet de détecter statiquement et de prévenir un grand nombre d'inconsistences dans la description du déploiement de la couche intergicielle.

Enfin, une plate-forme intergicielle pour le déploiement de la couche intergicielle des systèmes distribués a été implémentée sous forme d'une machine virtuelle DeployWare à base de composants. Cette plate-forme prend en charge l'**automatisation** de la procédure de déploiement de chacun de logiciels. Elle permet également de déployer automatiquement les logiciels conformément à l'ordre qu'imposent leurs dépendances respectives. Cette machine virtuelle repose sur une architecture de composants Fractal. Le choix du paradigme de composants pour réaliser la machine virtuelle s'explique par la séparation claire entre interfaces et implémentations de composants, ce qui permet de prendre en charge facilement l'hétérogénéité des systèmes en faisant varier les implémentations d'un composant pour une interface donnée. La seconde raison de ce choix réside dans le fait que certains modèles de composants existants, notamment Fractal, proposent une démarche reconfiguration dynamique des architectures. Cette reconfiguration dynamique de la machine virtuelle DeployWare est notamment utile pour la prise en charge des environnements ouverts.

Des composants de la machine virtuelle font qu'elle permet de prendre en charge le déploiement de la couche intergicielle des systèmes en **environnements ouverts distribués**. En effet il est possible d'implémenter des politiques d'auto-gestion conformes aux principes de l'*informatique auto-gérée*, exprimées à l'aide du paradigme *Événement-Condition-Action*. Ce paradigme permet de réaliser des modifications dynamiques du déploiement automatiquement, en prenant en compte des événements extérieurs, comme les arrivées de machines hôtes dans le domaine de déploiement par exemple. En outre, la machine virtuelle DeployWare permet de prendre en charge des **déploiements à grande échelle** en distribuant le processus de déploiement sur plusieurs machines afin de multiplier les ressources disponibles pour le déploiement. Des mécanismes pour exprimer la répétition du déploiement de logiciels de même type sur un grand nombre de machines ainsi que la possibilité d'effectuer le déploiement de logiciel sur un grand nombre de machines en parallèle permet de décrire facilement des couches intergicielles pouvant couvrir un très grand nombre de machines hôtes.

Une fois la couche intergicielle déployée, l'*architecte métier* en charge du déploiement des composants métiers peut recourir à deux manières différentes pour déployer son architecture de composants en fonction de ses besoins. La première possibilité est de fournir à l'administrateur système, un descripteur de type ADL, ainsi que le code conditionné dans des paquetages, le tout conformément à une technologie métier donné (*i.e.* un modèle de composants donné). En effet, de nombreux modèles de composants fournissent un outil de déploiement dédié à leur technologie (*e.g.* OpenCCM DCI, OSGi/OBR, Fractal ADL). Les experts logiciels de cette technologie encapsulent alors cet outil de déploiement dans un type de logiciel DeployWare. L'administrateur système est alors capable d'inclure le déploiement de l'architecture métier à gros grain dans la description de son système. Aucune visibilité et donc aucune dynamicité dans ces architectures n'est dans ce cas envisageable.

Pourtant, dans un contexte d'environnements ouverts, il est probable que l'architecte métier veuille inclure des politiques d'auto-gestion dans l'architecture métier, afin de décrire le déploiement de nouveaux composants sur les machines hôtes faisant leur apparition dans le domaine de déploiement dynamiquement. Pour cela, nous proposons DACAR, une approche à base de modèles pour le déploiement distribué auto-géré d'architectectures de composants. Cette approche repose également sur un **méta-modèle générique** de composants générique qui repose sur le méta-modèle de configuration et de déploiement d'applications à base de composants génériques OMG D&C discuté dans la section 2.3.3. À ce méta-modèle nous ajoutons les concepts pour exprimer des politiques d'autonomie sur le déploiement de ces architectures. DACAR représente le dernier maillon pour couvrir l'intégralité du déploiement des logiciels qui composent un système logiciel. En utilisant un méta-modèle de composants génériques, nous maintenons l'hétérogénéité à tous les niveaux de description d'un système logiciel.

3.6 Plan du document

Dans le chapitre 4, nous présentons notre méta-modèle DeployWare pour la description du déploiement de la couche intergicielle d'un système. Ce méta-modèle propose des concepts pour l'expert logiciel ainsi que l'administrateur système. Les vérifications qui sont possibles pour les modèles DeployWare sont également présentées dans ce chapitre. Dans le chapitre 5, nous présentons la machine virtuelle DeployWare à base de composants Fractal, pour l'exécution automatique du déploiement de la couche intergicielle. Les mécanismes pour l'exécution de politiques d'auto-gestion, ainsi que l'utilisation à grande échelle de cette machine virtuelle sont également présentés. Dans le chapitre 6, nous présentons DACAR, notre cadre de conception pour le déploiement d'architectures auto-gérées à base de composants. Nous présentons les éléments de modélisation en marge du méta-modèle OMG D&C permettant l'expression des politiques d'auto-gestion. Le maintien dynamique du lien causal entre ce modèle d'architecture et l'assemblage de composants à déployer, ainsi que l'exécution des politiques d'auto-gestion sont également discutés. Nous illustrons enfin notre approche sur deux exemples concrets de déploiement de systèmes distribués dans le chapitre 7.

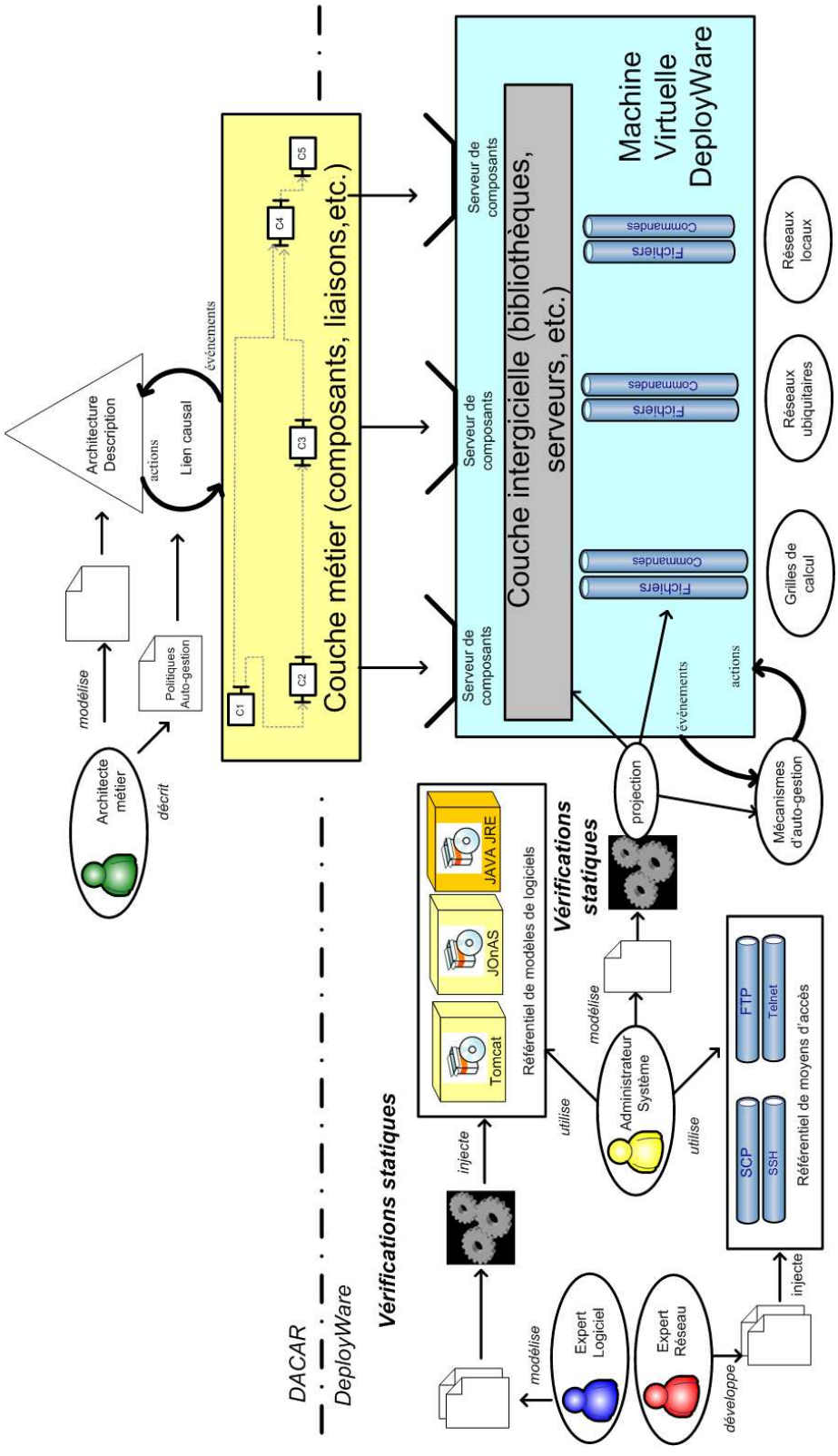

FIG. 3.6 – Vue d'ensemble de notre contribution

Chapitre 4

Méta-modèle générique DeployWare pour le déploiement de systèmes logiciels

Sommaire

Dans ce chapitre, nous allons détailler l'une des grandes lignes de notre contribution. Cette première étape de notre travail consiste à définir un méta-modèle générique pour le déploiement des logiciels de la couche intergicielle d'un système logiciel réparti. Ce langage doit fournir des concepts permettant, dans un premier temps, à l'expert logiciel d'exprimer les processus de déploiement des logiciels qu'il maîtrise. Cette description du protocole de déploiement de ces logiciels doit se faire indépendamment des spécificités des machines hôtes. Dans un second temps, ce langage permet aux administrateurs système de composer les modèles de déploiement des logiciels créés par les experts logiciels, afin de construire l'architecture de la couche intergicielle, et de déterminer de cette manière le protocole de déploiement de l'ensemble de la couche intergicielle.

Comme nous l'avons observé dans l'état de l'art, la possibilité d'effectuer des vérifications statiques sur les descripteurs de déploiement est une propriété importante pour le déploiement de logiciels. Nous souhaitons donc mettre en œuvre un certain nombre de vérifications comportementales statiques. Ce type de vérification permet d'aller au delà des simples vérifications de type et de syntaxe. Elles permettent de vérifier la cohérence globale du processus de déploiement décrit. C'est l'expressivité du méta-modèle DeployWare qui rend possible ces vérifications. Dans ce chapitre, nous spécifions formellement un certain nombre de propriétés à vérifier sur les modèles DeployWare, ainsi que leur mise en œuvre concrete dans notre approche.

Dans ce document, le méta-modèle est présenté sous une forme graphique pour des raisons de lisibilité. Néanmoins, ce méta-modèle, ainsi que les vérifications de ceux-ci, ont été réalisés à l'aide de Kermeta, le langage de méta-modélisation créé par l'équipe Triskell de l'IRISA [50]. Le langage Kermeta permet de spécifier non seulement la structure d'un méta-modèle, mais également son comportement. Kermeta est compatible avec la syntaxe EMOF [1], puisqu'il est construit comme une extension d'EMOF. Kermeta est intégré à l'IDE Eclipse en tant que plugin, et est librement disponible à l'adresse `http://www.kermeta.org`.

4.1 Méta-modèle DeployWare

4.1.1 Vue d'ensemble

Dans cette section, nous présentons notre méta-modèle pour décrire le processus de déploiement d'un système logiciel. Les concepts de ce méta-modèle sont découpés en deux paquetages, pour les deux rôles impliqués dans la modélisation du déploiement, c'est-à-dire l'expert logiciel et l'administrateur système. L'expert réseau possède un rôle plus programmatif que nous détaillons dans le chapitre 5. Quant à l'architecte métier, il utilise le méta-modèle d'architecture DACAR présenté dans le chapitre 6. Les paquetages de concepts pour l'expert logiciel et l'administrateur système sont tous deux représentés sur la figure 4.1. Il est à noter que l'administrateur doit utiliser les types de logiciels définis par les experts logiciels ; le paquetage *SystemAdmin* doit donc importer *SoftwareExpert*.

4.1.2 Expert logiciel

L'expert logiciel fait partie des concepteurs d'un logiciel donné. Il a, dans le cadre du déploiement, la tâche de fournir la spécification du déploiement de son logiciel. Nous proposons un méta-modèle, un langage dédié donc, pour lui permettre d'écrire la spécification complète du déploiement de son logiciel. À la fin de sa tâche, le logiciel est prêt à être instancié et déployé sur les machines hôtes par l'administrateur système.

Cette partie du méta-modèle offre des concepts utiles à la description du protocole de déploiement pour un logiciel donné. Ces concepts, représentés sur la figure 4.2 (paquetage *SoftwareExpert*), sont destinés au rôle d'Expert Logiciel. Il s'agit des concepts nécessaires et suffisants pour la définition précise du protocole de déploiement d'un logiciel. Pour définir ce langage générique de déploiement, nous avons étudié une multitude de paradigmes et de technologies différents. Parmi les paradigmes étudiés, on trouve des technologies dédiées 1) aux

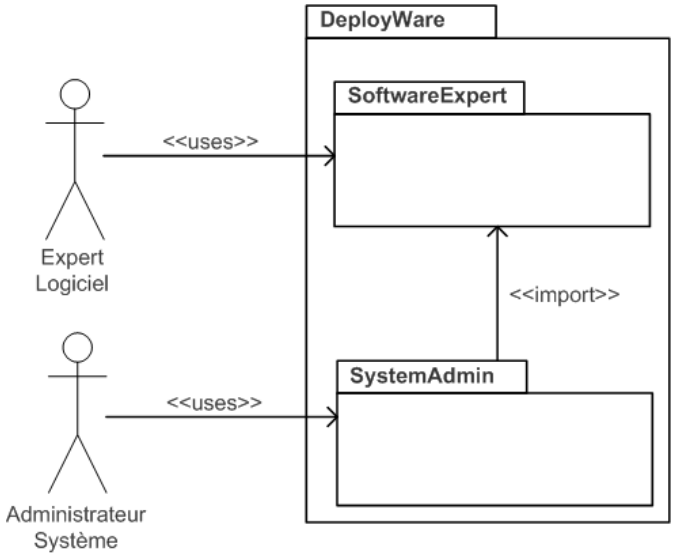

FIG. 4.1 – Vue d'ensemble des paquetages DeployWare

architectures orientées services (SOA) comme ActiveBPEL[1], Orchestra[2], Apache Tuscany [3], OW2 PEtALS [4], 2) à JEE comme les serveurs d'applications OW2 JOnAS [5], JBoss [6], Apache Geronimo [7], Sun GlassFish[8], 3) aux systèmes à base de CORBA comme OW2 OpenCCM[9]. De cette étude, nous avons créé un ensemble de concepts génériques permettant d'exprimer le déploiement de systèmes logiciels.

4.1.2.1 Définition des concepts

Personnalité Le concept de personnalité (**Personality**) représente un ensemble de logiciels communs à une technologie donnée. La notion de personnalité peut être assimilée à celle d'un paquetage, il s'agit d'un espace de nommage commun à un ensemble de logiciels. À titre d'exemple, on peut citer la personnalité JOnAS qui contient non seulement le logiciel serveur JEE, mais également les applications métiers déployées par dessus (*e.g.* archives EAR, WAR, JAR). Le concept de personnalité possède un attribut *name* de type EString qui représente son nom. Une personnalité possède une relation multiple *software* vers le concept de type de logiciel.

[1] http://www.activevos.com/community-open-source.php
[2] http://orchestra.ow2.org
[3] http://incubator.apache.org/tuscany
[4] http://petals.ow2.org
[5] http://jonas.ow2.org
[6] http://www.jboss.org
[7] http://geronimo.apache.org
[8] https://glassfish.dev.java.net/
[9] http://openccm.ow2.org

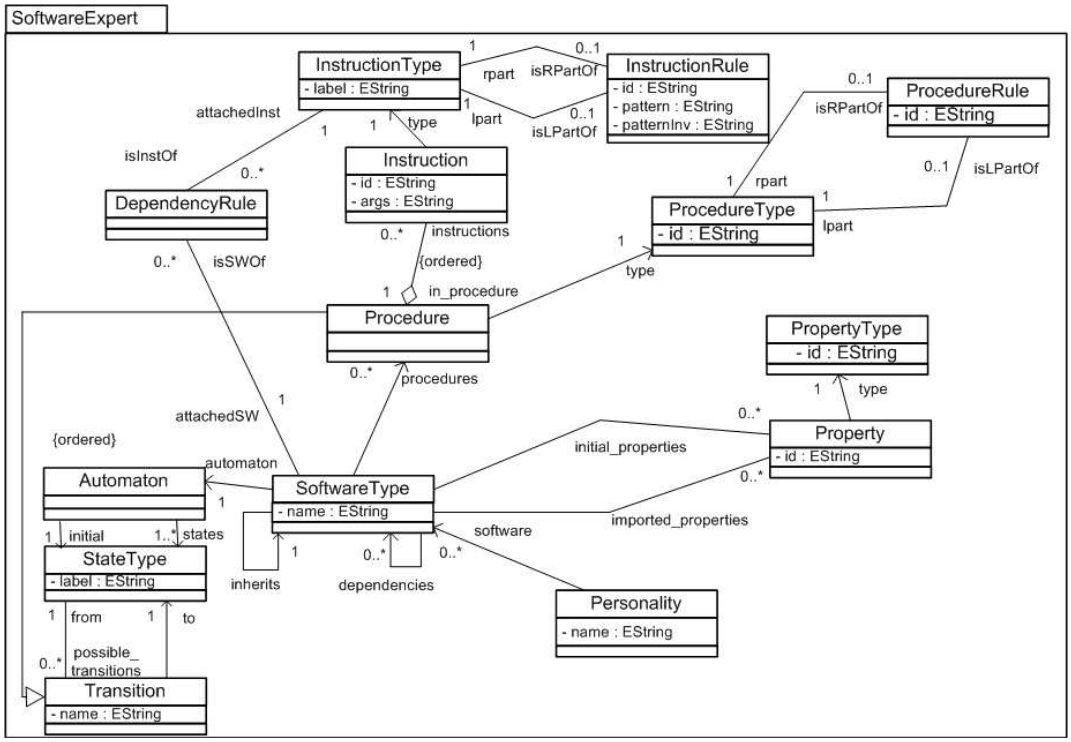

FIG. 4.2 – Paquetage de l'expert logiciel

Type de logiciel La notion centrale de ce paquetage du méta-modèle est celle de type de logiciel (**SoftwareType**) qui réifie n'importe quelle entité logicielle mise en œuvre dans la couche intergicielle d'un système logiciel. Un logiciel de n'importe quelle technologie, quelle que soit sa place dans la pile logicielle (hormis la couche métier, gérée par l'architecte métier) peut être représenté par un type de logiciel. Une type de logiciel possède un attribut *name* de type EString qui représente son nom unique au sein de sa personnalité. Un type de logiciel possède une référence multiple *dependencies* vers lui-même afin de représenter les dépendances logicielles. Un logiciel, pour pouvoir être déployé, peut nécessiter que d'autres logiciels soient déployés en amont. Créer une association entre un logiciel et ses dépendances signifie donc que ces logiciels doivent être déployés avant d'exécuter la procédure de déploiement de ce logiciel. Par exemple, n'importe quel serveur JEE, comme tout logiciel Java, nécessite qu'un *Java Runtime Environment* (JRE) soit déployé sur la machine où l'on souhaite déployer ce serveur. Dans DeployWare, JOnAS et le JRE sont tous deux représentés comme des types de logiciel, et le JRE sera ajouté à la liste des dépendances de JOnAS. Cela implique deux choses : premièrement, dans un système donné, si une instance de JOnAS est déclarée, une instance de JRE doit l'être également, et deuxièmement, lorsque l'on déploie JOnAS, la première chose à faire est de déployer le JRE décrit.

Un type de logiciel possède une référence multiple *procedures* vers le concept de procédure afin de matérialiser les différentes étapes du cycle de vie du déploiement du logiciel décrit. Un

type de logiciel possède une référence multiple *isSWOf* vers les éventuelles règles de dépendances qui le concerne. Un type de logiciel possède un automate (*automaton*) qui définit l'état du logiciel ainsi que l'enchainement possible d'appels de procédures en fonction de l'état. Enfin, un type de logiciel possède un certain nombre de propriétés. Ces propriétés peuvent être de natures différentes. La première catégorie de propriétés sont les propriétés que l'on peut qualifier d'inhérentes au logiciel. Par exemple, on peut citer le port sur lequel un serveur se met en attente de requêtes. Ce type de propriétés peut concerner aussi bien des propriétés de fonctionnement du logiciel que des propriétés de déploiement. On peut donc représenter sous cette forme le chemin de l'archive du logiciel, ou encore le chemin dans le système de fichiers où ce logiciel doit être installé, etc.. Ces propriétés font partie de la définition du logiciel concerné. On appelle ces propriétés *initiales* et dans le méta-modèle, elles sont définies par l'association multiple *initial_properties* vers le concept de propriété. L'autre type de propriétés concerne des propriétés qui sont importées d'autres logiciels. En effet, il est possible qu'un logiciel ait besoin de détenir une information provenant d'un autre logiciel duquel il dépend, pour fonctionner correctement. Par exemple, un serveur JOnAS dépend d'un serveur de bases de données. Pour que les requêtes soient correctement acheminées au serveur de bases de données, le serveur JOnAS doit être configuré avec le numéro de port du serveur de bases de données. Cette propriété ne caractérise pas directement le serveur JOnAS mais plutôt le serveur de bases de données, cette propriété est néanmoins utilisée pour la configuration de JOnAS. Il s'agit donc d'une propriété dite *importée*, et le logiciel qui l'importe possède une référence (*imported_properties*) sur cette propriété (qui est une propriété initiale d'un autre logiciel).

Un type de logiciel peut également hériter d'un autre type de logiciel. Cet héritage est matérialisé dans le méta-modèle par l'association simple *inherits* vers un autre type de logiciel. Cela signifie qu'il hérite de toutes les propriétés et procédures définies pour ce logiciel. Pour des raisons de simplicité, et éviter les problèmes de conflits lors de schémas d'héritages complexes, nous décidons d'interdire l'héritage multiple de types de logiciels.

Propriétés Le concept de propriété (**Property**) représente une variable d'un logiciel, c'est-à-dire un paramètre qui peut prendre une valeur différente pour chaque instance du logiciel. Il s'agit d'un champ configurable du logiciel. Une propriété possède un attribut *id* de type EString pour représenter son identifiant unique au sein du logiciel décrit. Une propriété possède une relation unique *type* vers le concept de type de propriété.

Type de propriétés Le type de propriété permet de représenter une classe de propriété donnée. Il permet d'ajouter de l'information supplémentaire de typage à un champ du logiciel décrit. Ce concept possède un attribut *id* de type EString afin de déterminer l'identifiant du type de propriété.

Procédures Il existe dans ce méta-modèle des concepts permettant de décrire les instructions élémentaires qui composent le protocole de déploiement d'un logiciel donné. Le concept central de cette partie est la **Procedure** qui représente un ensemble d'opérations ayant pour but

de remplir l'une des étapes du cycle de vie de déploiement d'un logiciel. Une procédure possède un attribut *name* de type EString pour désigner son nom unique à l'intérieur du logiciel décrit. Elle possède un type (*type*), ce qui signifie qu'elle fournit les opérations dans un but précis qui est celui du type en question. Une procédure possédant le type *Installation* contient donc un ensemble d'opérations que visent à installer le logiciel concerné. Ce cycle de vie est configurable. En effet, un certain nombre de type de d'étapes de cycle de vie peuvent être ajoutées. Par exemple, un serveur JEE possède une interface web d'administration. Une action lors du déploiement de ce serveur peut consister à ouvrir cette interface pour interagir avec elle. Cette opération de *management* n'est pas une opération du cycle de vie du déploiement tel que nous l'avons défini, mais il convient néanmoins de l'ajouter dans les procédures. Le concept de procédure hérite de celui de transition, car l'appel d'une procédure représente le passage entre deux états possibles du logiciel. Une procédure possède une référence multiple ordonnée *instructions* vers le concept d'instructions, afin de décrire le comportement de cette procédure en terme d'actions élémentaires sur la machine hôte.

Type de procédure Un type de procédure (**ProcedureType**) est le concept qui permet de modéliser l'intention d'une procédure. Ce concept possède un attribut *id* de type EString qui permet d'exprimer l'identifiant unique du type défini. En effet, l'installation d'un logiciel, par exemple, représente une intention qui peut être réalisée par une ou plusieurs procédures. Un type de procédure peut être associé à son type de procédure antagoniste, c'est-à-dire un type de procédure qui manifeste l'intention inverse (*e.g.* installation/désinstallation, ou démarrage/arrêt). Dans ce cas, il possède une référence *isRPartOf* ou *isLPartOf* vers une règle de procédure. Ces règles étant binaires, le type de procédure considéré peut être membre *gauche* ou *droit* de la règle.

Règle de procédure Ce concept permet d'associer dans un modèle DeployWare deux types de procédures à intention antagoniste. Ainsi ce concept de règle de procédure (**ProcedureRule**) possède deux références uniques distinctes *rpart* et *lpart* vers deux types de procédures à comportement antagonistes. L'utilisation de ce concept concerne les algorithmes de vérification et sera décrite dans la section 4.2.

Instruction Les instructions (**Instruction**) représentent des actions élémentaires de déploiement à exécuter sur une machine hôte distante (*e.g.* télécharger un fichier, fixer une variable d'environnement, lancer un exécutable, etc.). Les instructions possèdent un attribut *label* de type EString pour représenter un identifiant unique au sein d'une procédure. Elles possèdent également un attribut de type EString *args* qui représente sous forme textuelle des arguments passés à l'instruction. Elles possèdent une référence unique *type* vers un type d'instruction, une référence unique *in_procedure* pour référencer la procédure dans laquelle elle est contenue.

Type d'instruction Un type d'instruction (**InstructionType**) est le concept qui permet de modéliser l'intention d'une instruction. Ce concept possède un attribut *id* de type EString qui permet d'exprimer l'identifiant unique du type défini. Par exemple, le positionnement

d'une variable d'environnement représente une intention qui peut être réalisée par une instruction ou plusieurs instructions différentes. Un type d'instruction peut-être associé à son type d'instruction antagoniste, c'est-à-dire un type d'instruction qui manifeste l'intention inverse (*e.g.* positionner/supprimer une variable, ou démarrer/tuer un processus). Dans ce cas, il possède une référence *isRPartOf* ou *isLPartOf* vers une règle d'instruction. Ces règles étant binaires, le type d'instruction considéré peut être membre *gauche* ou *droit* de la règle. Un type d'instruction possède également une référence multiple *isInstOf* vers les éventuelles règles de dépendance qui le concernent. Il est important de noter que ces éléments de modélisation doivent représenter l'action à effectuer de manière abstraite et indépendante de toute considération technique sous-jacente. Par exemple, on peut créer un type d'instruction `SetVariable` pour fixer une variable d'environnement, mais c'est une erreur de créer un type d'instruction `ExportVariable` car la commande `export` est dépendante d'un shell et d'un système d'exploitation donné (le Bourne Shell du système GNU/Linux).

Règle d'instruction Ce concept permet d'associer dans un modèle DeployWare deux types d'instruction à intention antagoniste. Ainsi ce concept de règle d'instruction (**InstructionRule**) possède deux références uniques distinctes *rpart* et *lpart* vers deux types d'instruction à comportements antagonistes. L'utilisation de ce concept sera décrite dans la section 4.2. Une règle d'instruction possède un attribut de type EString pour représenter son identifiant unique au sein du logiciel modélisé. Elle possède également deux attributs de type EString qui représentent deux expressions régulières auxquelles doivent correspondre les arguments des deux instructions antagonistes.

Règle de dépendance Ce concept de règle de dépendance (**DependencyRule**) permet d'associer dans un modèle DeployWare un type d'instruction et le type de logiciel qui le fournit. Ce concept possède donc une référence unique *attachedInst* vers le type d'instruction concerné, et une référence unique *attachedSW* vers le type de logiciel qui fournit ce type d'instruction.

Automate L'automate (**Automaton**) représente l'ensemble d'appels possibles de procédures pour un type de logiciel en fonction de l'état dans lequel il se trouve. Un automate possède une relation multiple *states* vers les différents types d'états qu'il possède. Il possède une seconde relation simple *initial* vers le type d'état initial (fixé par défaut aux instances de ce logiciel) d'un type de logiciel.

Transition Une transition (**Transition**) représente le passage entre deux types d'état du système. Une transition possède deux références simples vers le concept de type d'état, qui matérialisent le type d'état à partir duquel on peut déclencher cette transition (*from*), et le type d'état vers lequel l'automate arrive après l'exécution de cette transition (*to*). Une transition possède également un attribut de type EString *name* qui représente l'étiquette de la transition.

Type d'état Un type d'état (**StateType**) représente l'état d'avancement du déploiement d'un type de logiciel donné. Un changement d'état est provoqué par l'appel d'une transition de déploiement pour le logiciel considéré. Le fait qu'un logiciel soit dans un état donné conditionne

Identifiant	Description
Installation	Type de procédure pour décrire une procédure visant à installer un logiciel.
Configure	Type de procédure pour décrire une procédure visant à configurer un logiciel, avant son lancement.
Start	Type de procédure pour décrire une procédure visant à mettre un logiciel en état de fonctionnement.
Stop	Type de procédure pour décrire une procédure visant à arrêter le fonctionnement d'un logiciel.
Uninstallation	Type de procédure pour décrire une procédure inverse à celle de l'installation qui vise donc à désinstaller un logiciel.

TAB. 4.1 – Types de procédures peuplant la bibliothèque de base DeployWare

le nombre de transitions qui peuvent être appelées, par le biais de transition. Un type d'état possède donc une référence multiple *possible_transitions* afin de décrire les transitions déclenchables à partir de l'état. Un type d'état peut également être étiqueté à l'aide de l'attribut *label* de type EString.

4.1.2.2 La bibliothèque initiale pour l'expert logiciel

Pour offrir un langage de déploiement directement utilisable, un certain nombre d'éléments sont requis. Parmi ces éléments, nous avons notamment besoin d'un certain nombre de types de procédures, ainsi qu'un certain nombre de types d'instructions ou encore de types de propriétés. Il nous faut donc disposer d'un certain nombre d'éléments de modèles génériques insérés dans la bibliothèque de base de DeployWare, pour servir de point de départ au travail des experts logiciels. Dans le chapitre 3, nous reviendrons sur cette bibliothèque et ce qu'implique de définir de nouveaux types. Nous allons tout d'abord définir un certain nombre de types de procédures, décrits dans le tableau 4.1. Quelques exemple de types de propriété sont représentés sur le tableau 4.2. Enfin, des types d'instructions génériques sont présentés sur le tableau 4.3.

Il est bien sûr possible d'étendre cette bibliothèque afin d'ajouter de nouveaux types d'instruction, de procédure ou de propriétés.

Nous ajoutons également, pour simplifier la description de modèles de logiciels dans la suite du document, un type générique *abstrait* de logiciel dit *Installable*, qui factorise des propriétés et des opérations communes à un grand nombre de logiciels. Un diagramme d'instance informel du type de logiciel `Installable` est représenté sur la figure 4.3.

Ce modèle de logiciel définit le type de logiciel `Installable` qui définit deux propriétés : `archive` de type paramètre d'archive, et `home` de type paramètre de chemin d'installation. Deux procédures *install* de type `Installation`, et *uninstall* de type `Uninstallation` sont déclarées dans ce type de logiciel. Les instructions de la procédure *install* consistent à d'abord charger l'archive sur la machine hôte distante (instruction *upload*), ensuite à se déplacer dans le répertoire choisi pour l'installation du logiciel (instruction *cd-home*), puis à extraire l'archive du logiciel (instruction *unarchive*), et enfin supprimer l'archive une fois son contenu extrait (instruction *remove*). La procédure *uninstall* contient une instruction *remove-home* qui

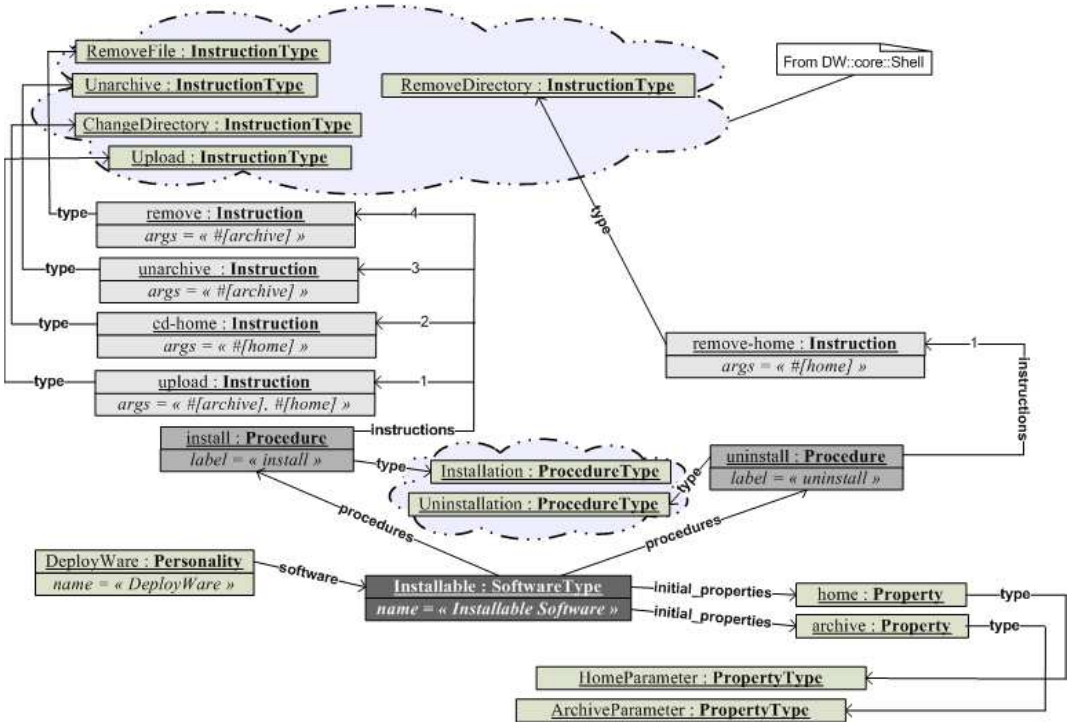

FIG. 4.3 – Modèle générique de logiciel Installable

supprime le répertoire `home` défini pour le logiciel. En effet, toutes les actions de la procédure `install` ont eu pour seul effet de créer ce répertoire (l'archive a été supprimée dans la procédure `install`). La simple action de supprimer ce répertoire annule donc l'installation.

4.1.2.3 Un exemple de modélisation de logiciel : JOnAS

Dans cette section, nous illustrons l'utilisation de notre méta-modèle pour modéliser deux types de logiciels. Nous choisissons d'étudier la modélisation du type de logiciel JOnAS, un serveur JEE distribué dans le cadre du consortium OW2, ainsi que d'un composant à déployer pour ce type de serveur, *i.e.* un Enterprise Java Bean (EJB). Ces modèles sont représentés sous

Identifiant	Description
HomeParameter	Type de propriété standard permettant de décrire le chemin du répertoire d'installation d'un logiciel.
ArchiveParameter	Type de propriété permettant de décrire l'emplacement d'une archive de logiciel.
PortParameter	Type de propriété permettant de décrire un port réseau sur une machine.
NameParameter	Type de propriété permettant de décrire le nom d'un logiciel.

TAB. 4.2 – Quelques types de propriétés peuplant la bibliothèque de base DeployWare

Identifiant et arguments	Description
AddVariable (var_name,path)	Ajouter un séparateur puis un chemin à la variable d'environnement spécifiée.
AddToPath (path)	Spécialisation de AddVariable pour la variable PATH.
ChangeDirectory (path)	Changer le répertoire courant.
CopyFile (file_path,dest_path)	Copier un fichier de la machine hôte vers un autre emplacement (toujours sur la machine hôte).
Execute (exec_path)	Lancer un exécutable.
Fork (exec_path)	Lancer un processus en tâche de fond.
LaunchProcess (exec_path,pid)	Lancer un exécutable en tâche de fond et enregistrer son identifiant de processus (le PID).
KillProcess (pid)	Tuer un exécutable en tâche de fond en utilisant son identifiant de processus (le PID).
MakeDirectory (dir_path)	Créer un répertoire.
RemoveDirectory (dir_path)	Supprimer un répertoire.
RemoveFile (file_path)	Supprimer un fichier.
RenameFile (file_path, new_name)	Renommer un fichier.
SetVariable (var_name,var_path)	Affecter une valeur à une variable d'environnement.
Unarchive (file_path)	Commande générique pour décompresser une archive.
Upload (file_path, dest_path)	Commande générique pour charger un fichier sur une machine hôte.
UnsetVariable (var_name)	Commande générique pour supprimer une variable d'environnement.
UploadGenFile (file_name,dir_path)	Charger un fichier de configuration sur une machine hôte.

TAB. 4.3 – Types d'instruction peuplant la bibliothèque de base DeployWare

forme de diagrammes d'instances informels représentés sur les figures 4.4 et 4.5.

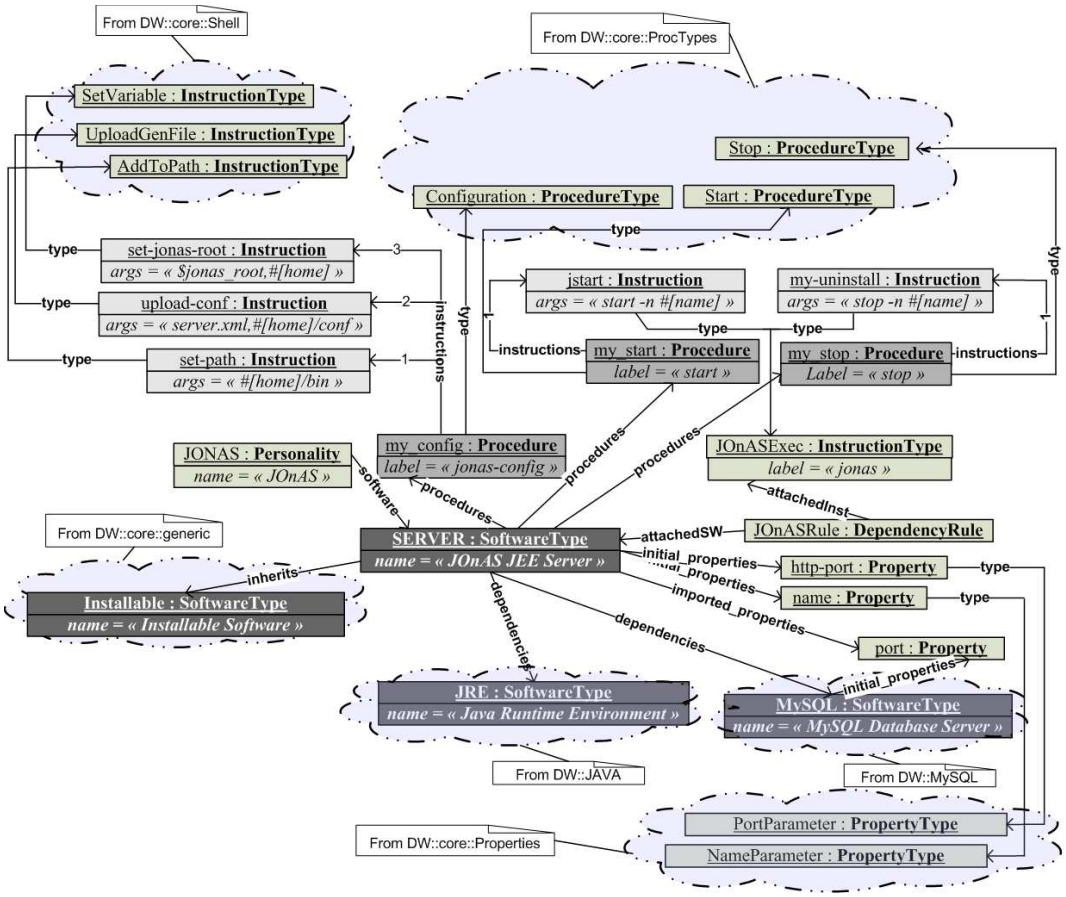

FIG. 4.4 – Modélisation de logiciel avec DeployWare : le serveur JEE JOnAS

Plaçons nous dans la perspective de l'expert logiciel qui a pour tâche de définir le processus de déploiement du serveur JEE JOnAS. Il définit donc en premier lieu une nouvelle personalité `JONAS` dans laquelle il introduit la définition d'un nouveau type de logiciel `SERVER`. Il définit ensuite les propriétés intrinséques du serveur JOnAS à savoir le port (propriété **http-port** de type **PortParameter**) sur lequel il propose entre autres un accès d'administration Web, un identifiant (propriété **name** de type **NameParameter**). Le serveur JOnAS nécessite d'être connecté à une base de données pour pouvoir fonctionner correctement. L'expert logiciel définit donc une première dépendance vers le type de logiciel MySQL. D'ailleurs, le logiciel `JOnAS.SERVER` possède également une référence vers la propriété *port* du logiciel `mysql1` en tant qu'*imported_properties*. En outre, JOnAS nécessite comme tout logiciel Java, une machine virtuelle Java. C'est la raison pour laquelle une deuxième dépendance vers le type de logiciel JRE est définie. Son type de logiciel hérite de `Installable`, afin d'hériter des tâches d'installation et désinstallation génériques, ainsi que des propriétés pour le répertoire d'installation (propriété **home**) et de l'archive contenant la distribution (propriété **archive**). Il définit ensuite trois procédures : `my_config` de type `Configuration`, `my_start` de type

`Start` et `my_stop` de type `Stop`.

La procédure de configuration contient 3 instructions : pour fixer la variable $JONAS_ROOT nécessaire au démarrage de JOnAS, pour charger un fichier de configuration rempli avec les propriétés du logiciel, et enfin pour ajouter l'exécutable de JOnAS dans la variable d'environnement $PATH. Il définit ensuite un nouveau type d'instruction `JOnASExec` qui réifie l'utilisation de la commande `jonas`. Ce nouveau type d'instruction est utilisé dans les procédures `my_start` et `my_stop` afin de respectivement exécuter les commandes pour démarrer (commande `jonas start`) et arrêter (commande `jonas stop`) le serveur.

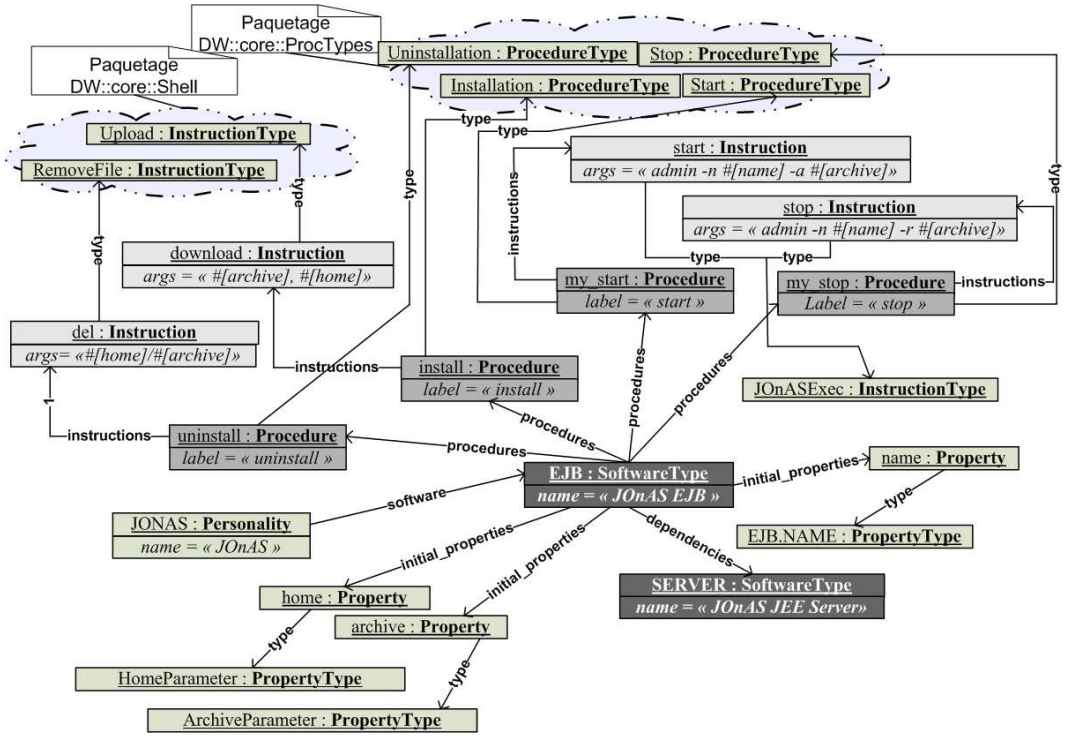

FIG. 4.5 – Modélisation de logiciel avec DeployWare : un EJB

L'expert logiciel va ensuite définir le modèle de déploiement d'un EJB. Il définit de la même manière les propriétés **name**, **home** et **archive**. Ce logiciel n'hérite pas du logiciel `Installable` car il n'est entre autres pas nécessaire de décompresser l'archive d'un EJB pour le déployer. Ainsi une procédure d'installation contient une instruction de téléchargement de l'archive, une procédure de démarrage utilise l'instruction JOnASExec pour déployer l'archive sur le serveur, une procédure d'arrêt replie l'archive EJB, et enfin une procédure de désinstallation supprime l'archive.

4.1.3 Administrateur système

Une fois les personnalités modélisées par les experts logiciel, l'administrateur système construit des instances de logiciels définis dans ces personnalités, configure ces instances et ainsi décrit l'assemblage global de ces instances. L'administrateur système modélise ainsi le déploiement de l'ensemble de la couche intergicielle de son système logiciel.

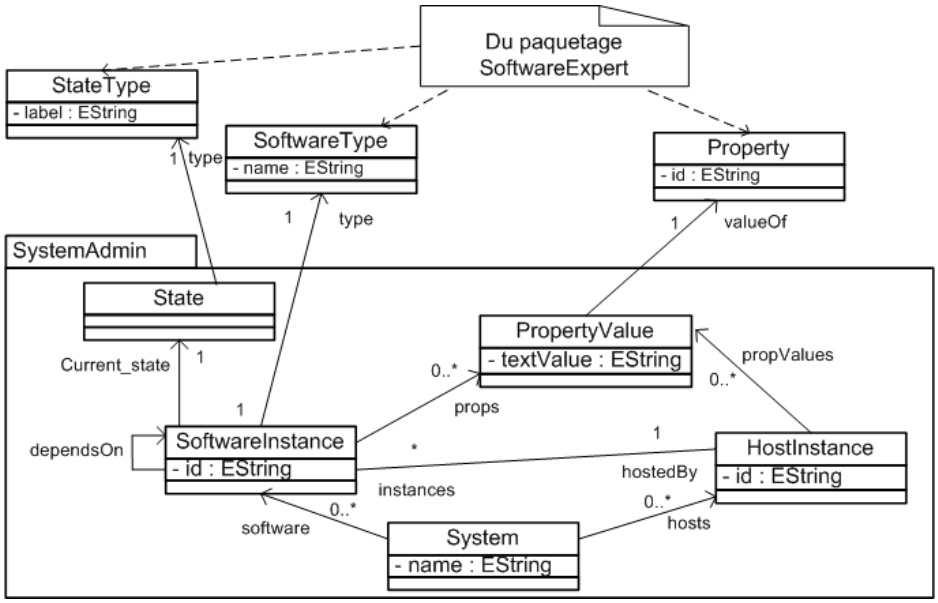

FIG. 4.6 – Paquetage de l'administrateur système

4.1.3.1 Définition des concepts

Un deuxième paquetage dans notre méta-modèle, appelé *SystemAdmin* et représenté sur la figure 4.6, offre les concepts permettant à l'administrateur de composer la couche intergicielle de son système logiciel.

Système La notion de système (**System**) représente l'ensemble des instances de logiciels et des machines hôtes impliquées dans le déploiement modélisé. Ce concept possède un attribut *name* de type EString qui représente l'identifiant du système. Il possède une référence multiple *software* vers les instances de logiciels à déployer, et une référence multiple *hosts* vers les machines hôtes impliquées.

SoftwareInstance Le concept d'instance de logiciel (**SoftwareInstance**) reflète la description du déploiement d'une instance d'un type de logiciel. Ce concept possède un attribut *id* de type EString qui décrit le nom de l'instance. Une instance de logiciel possède une référence simple *type* vers le concept de type de logiciel du paquetage de l'expert logiciel. Ce concept

possède également une référence multiple *props* vers le concept de valeur de propriétés, afin de décrire les valeurs que prennent les différentes propriétés définies pour le type du logiciel décrit. Enfin, une instance de logiciel possède une référence simple *hostedBy* vers le concept de machine hôte afin de définir la machine qui héberge l'instance décrite.

HostInstance Le concept d'instance d'hôte (**HostInstance**) représente une machine hôte du domaine de déploiement. Une machine hôte possède un attribut de type EString pour représenter le nom de la machine décrite. Ce concept possède une référence multiple *propValues* vers le concept de valeur de propriété, afin de représenter les valeurs pour les propriétés de communication de la machine hôte décrite. Enfin, une instance de machine hôte possède une référence multiples *instances* vers les différentes instances de logiciels qu'elle héberge.

Valeur de propriété Ce concept (**PropertyValue**) réifie la valeur pour une propriété d'un logiciel ou une machine hôte donnés. L'expert logiciel a défini des propriétés pour le type de logiciel qu'il modélise. L'administrateur système assigne ensuite des valeurs à ces propriétés via le concept de valeur de propriété. De la même manière l'administrateur système affecte des valeurs pour les propriétés de la machine hôte[10]. Ce concept possède donc une référence unique *valueOf* vers la propriété du type de logiciel, ou de la machine hôte concernée. En outre, ce concept possède un attribut *textValue* de type EString qui représente la valeur textuelle de cette propriété.

4.1.3.2 Bibliothèque initiale pour l'administrateur système

Un certain nombre de propriétés doit être fixé à propos des machines, afin d'en définir les moyens d'accès. Nous allons définir un certain nombre de propriétés à usage général pour les machines hôtes, ainsi que nous l'avons fait pour les types de procédures et d'instructions.

Le nom d'hôte Le nom d'hôte ou l'adresse IP de la machine. Une propriété de ce type doit posséder un argument correspondant à la chaine de caractères encodant l'adresse.

L'utilisateur Les paramètres pour se connecter sur la machine en tant qu'un utilisateur. Une propriété de ce type doit posséder en argument les identifiants, mots de passe et/ou clés privées d'authentification si un tel mécanisme d'authentification est accessible sur la machine.

Transfert de fichiers Le protocole de transfert de fichiers utilisé pour télécharger des fichiers sur la machine. Une propriété de ce type ne possède pas d'argument. Il existe une valeur de propriété par protocole existant. Parmi les protocoles existants, on trouve entre autres SCP et FTP.

Protocole d'accès à distance Le protocole de communication disponible sur la machine pour envoyer des commandes. Une propriété de ce type ne possède pas d'argument. Il existe une valeur de propriété par protocole existant. Parmi les protocoles existants, on trouve entre autres SSH et Telnet.

[10] Ces propriétés sont définies par l'administrateur système, et non par l'expert logiciel, celui-ci n'ayant pas de connaissances sur les machines hôtes du domaine de déploiement.

Shell Le langage d'interpréteur de commandes disponible sur la machine. Une propriété de ce type ne possède pas d'argument. Il existe une valeur de propriété par langage existant. Parmi les langages existants, on trouve entre autres le Bourne Shell, le C-Shell ou encore le langage d'interpréteur de commande de Windows.

Il est également possible d'ajouter des propriétés aux machines hôtes si le besoin s'en fait sentir. Ces propriétés peuvent être strictement documentaires, ou alors servir à l'adaptation des mécanismes de déploiement.

4.1.3.3 Exemple de modélisation de système à base de JEE

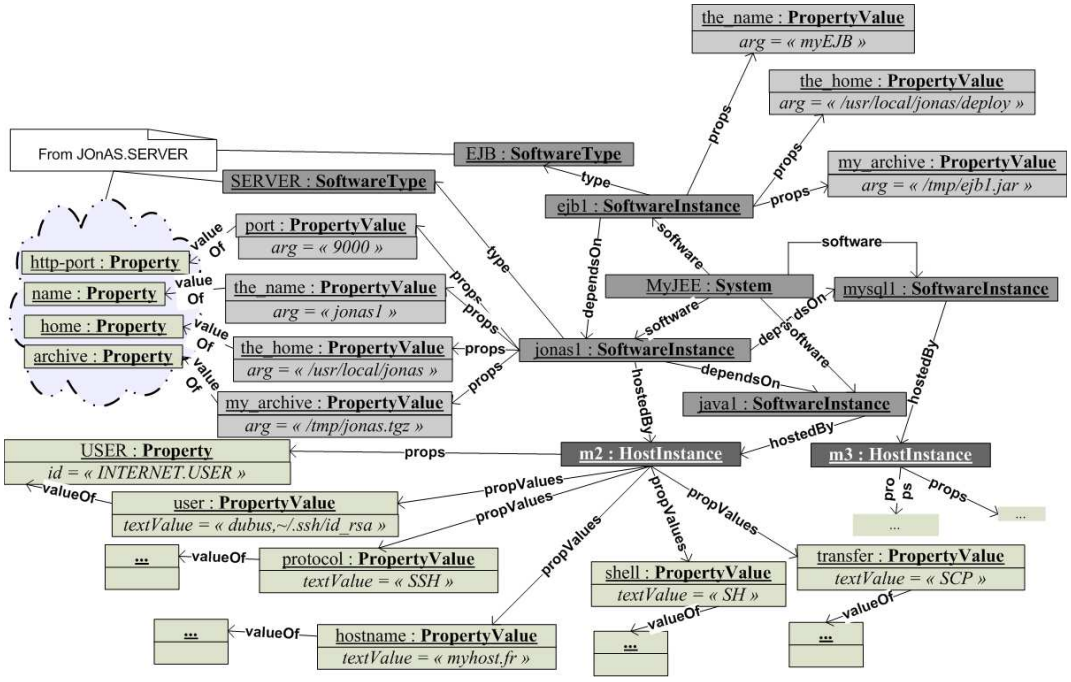

FIG. 4.7 – Système JEE à l'aide de la personnalité JOnAS

Un exemple de modèle de système logiciel tel que l'administrateur système peut concevoir, en l'occurrence une architecture JEE qui utilise les deux logiciels modélisés précédemment, est représenté sur la figure 4.7.

Nous nous plaçons donc cette fois dans la perspective de l'administrateur système. Nous supposons que les experts logiciels JOnAS et MySQL ont défini un certain nombre de types de logiciels, qui vont être réutilisés ici.

Tout d'abord l'administrateur configure ensuite la machine hôte devant accueillir le logiciel serveur JEE. Il fixe notamment les propriétés relatives au protocole (*protocol*), au shell (*shell*), au transfert de fichiers (*transfer*), au nom d'hôte (*hostname*) et aux informations de connexion (*user*) pour la machine choisie. L'administrateur va ensuite instancier des logiciel `JRE` (*java1*) et `MySQL` (*mysql1*), que nous ne détaillons pas ici. Puis il va définir une instance de logiciel `JOnAS.SERVER` au sein d'un système *MyJEE*. Cette instance de logiciel dépend de *java1* et

de *mysql1*. Il affecte ensuite une valeur aux différentes propriétés de son logiciel via les valeurs de propriétés *port*, *the_name*, *the_home* et *my_archive*. Enfin, il définit une instance de logiciel `EJB` configure les propriétés, fixe de la même manière les dépendances (un EJB dépend d'un serveur `JOnAS`, donc il ajout *jonas1* aux dépendances de *ejb1*). De cette manière, il a décrit l'ensemble du processus de déploiement d'un EJB et de l'infrastructure logicielle sous-jacente.

4.2 Vérifications statiques sur les modèles DeployWare

4.2.1 Vue d'ensemble du processus de vérification

Pour renforcer la généricité du méta-modèle DeployWare, ainsi que pour accroître son utilisabilité, nous précisons le sens exact de chaque concept du langage, et la manière dont ces concepts doivent être employés. En spécifiant cette sémantique, nous restreignons notre langage à une manière unique d'utilisation, et empêchons toute ambiguité dans l'utilisation du langage. En effet, nous avons pu constater, lors de l'étude de l'existant, que de nombreux langages de déploiement, dits génériques, péchaient finalement par leur manque de sémantique, ce qui restreint non seulement une éventuelle automatisation de la projection des modèles indépendants de la plate-forme vers les modèles spécifiques, mais surtout qui ne permet à aucun programme de vérifier statiquement la cohérence du déploiement décrit.

Par exemple, les diagrammes de déploiement d'UML sont principalement utilisés pour documenter le déploiement du système modélisé. En revanche, il est très difficile de projeter automatiquement ces modèles vers des modèles spécifiques, précisément à cause du manque de sémantique de ce langage. En effet, ne sachant pas à quoi correspondent exactement les concepts, il est difficile de les projeter vers des concepts concrets des langages cibles. En outre, il est impossible d'effectuer une vérification statique de la cohérence du déploiement décrit. Par exemple, si l'on modélise le déploiement d'une application JEE avec un diagramme de déploiement UML de base, il est impossible de vérifier que les composants déployés sur le serveur JEE sont bien tous des composants JEE (EJB ou Servlets), et non des composants JBI ou encore CCM.

Certaines autres approches effectuent néanmoins quelques vérifications sur les descripteurs de déploiement. Ces vérifications sont généralement au mieux *dynamiques directes*[11] (l'erreur, exprimée clairement, n'apparait qu'une fois le déploiement commencé) comme dans GoDIET, mais la plupart du temps elles sont *dynamiques indirectes* (une fois le déploiement commencé, l'erreur apparait sous une forme indirecte, et la vraie cause de l'erreur reste à décrypter). Cela signifie que l'administrateur en charge du déploiement ne peut constater l'échec de son déploiement seulement que lorsque celui-ci est en train de s'exécuter. De plus, il doit souvent décrypter les messages d'erreurs qu'il obtient afin de déterminer la cause exacte de l'erreur. Dans les déploiements à grande échelle principalement, ce constat est inacceptable, car la totalité du déploiement doit alors être rejouée, après avoir corrigé l'erreur. En outre, effectuer un déploiement erroné dans des environnements ubiquitaires, où des machines apparaissent et

[11] Ces types de vérification sont détaillés dans la section 2.1.5

disparaissent dynamiquement, peut avoir des conséquences dommageables sur les terminaux concernés.

Nous proposons donc d'établir formellement la sémantique de certains concepts de notre langage. Cette sémantique nous offre la possibilité de vérifier statiquement les modèles DeployWare décrits par les différents acteurs du déploiement. Nous proposons donc d'utiliser la sémantique formelle de nos modèles pour pouvoir les vérifier **statiquement** et ainsi assurer que ces modèles décrivent un déploiement *complet*, *cohérent* et *réversible*.

4.2.2 Définition des propriétés à vérifier

Un modèle de déploiement est dit *complet* lorsque toutes les dépendances et toutes les propriétés requises pour les logiciels impliqués dans le déploiement sont fixées. Vérifier qu'un modèle de déploiement est *cohérent* consiste à vérifier que les propriétés et les dépendances définies pour une instance de logiciel sont bien fixées de manière conforme à la description établie par l'expert logiciel, et qu'il n'y a donc pas d'incohérences dans le modèle complet défini. Un modèle de déploiement est enfin considéré comme étant *réversible* s'il existe une ou plusieurs opérations capables d'annuler chaque opération de déploiement d'une des entités déployées. Nous caractérisons dans la suite, de manière formelle à l'aide d'une logique de premier ordre, les propriétés à remplir pour un modèle et nous étudions également les validateurs réalisés pour les modèles DeployWare.

Nous mettons en œuvre cinq vérifications sur les modèles de déploiement DeployWare, que nous classons en trois catégories (complétude, cohérence, et réversibilité), développées dans les sections 4.2.4, 4.2.5, et 4.2.6. Cette liste de cinq vérifications n'est pas exhaustive. Elle constitue la première étape d'une démarche de vérification statique des modèles qui doit être approfondie dans les travaux futurs. Néanmoins, cette liste de vérifications ainsi que la démarche utilisée pour effectuer les vérifications sur les modèles représente une preuve de concept convaincante quant à la faisabilité de la vérification statique du déploiement de systèmes logiciels.

Tout d'abord, le premier type de vérification que nous mettons en œuvre concerne les dépendances d'un type de logiciel défini par l'expert logiciel. Nous souhaitons vérifier la pertinence des déclarations de dépendance. Une dépendance est déclarée pertinente pour un type de logiciel donné si au moins une instruction d'au moins une procédure est liée au logiciel déclaré comme dépendance. Cette vérification permet d'évaluer la complétude du type de logiciel modélisé et est détaillée dans la section 4.2.4.

Ensuite, nous souhaitons effectuer deux types de vérifications qui concernent l'administrateur système. La première d'entre elles concerne la vérification des dépendances d'une instance de logiciel. Il s'agit de vérifier que si le déploiement d'une instance de logiciel est modélisé, alors les dépendances de ce logiciel doivent être présentes également dans le modèle. La seconde vérification concerne les valeurs de propriétés d'une instance de logiciel. Cette vérification en deux temps consiste à vérifier qu'une instance de logiciel possède bien les valeurs pour chaque propriété nécessaire. Dans un second temps, il s'agit de vérifier que les ressources utilisées par une instance de logiciel sur une machine ne le sont pas également par une autre instance de logiciel déployée sur la même machine. Ces deux propriétés permettent d'assurer la

cohérence du processus de déploiement modélisé par l'administrateur système, et sont définies dans la section 4.2.5.

Enfin, nous souhaitons vérifier que le déploiement d'un type de logiciel est réversible. Cela signifie que ce logiciel peut être déployé sur une machine mais également replié de manière complète. Ainsi pour un type de logiciel défini, il s'agit de vérifier que chacune des procédures est bien réversible à l'aide d'une autre procédure dite inverse. Ensuite, pour aller plus loin dans la vérification de la réversibilité, la seconde vérification consiste à vérifier qu'au sein d'un modèle de type de logiciel, deux procédures inverses possèdent bien un comportement inverse en terme d'instructions. Il s'agit donc de vérifier que l'action de chaque instruction d'une procédure est annulée par l'instruction inverse dans la procédure inverse. Ces deux vérifications permettent d'évaluer la réversibilité du déploiement. Elles sont présentées dans la section 4.2.6.

Nous définissons nos vérifications en deux temps dans la suite de ce chapitre. Dans un premier temps, nous présentons de manière formelle, à l'aide de la logique de premier ordre, les différentes propriétés à vérifier. Dans un second temps, nous détaillons les algorithmes des programmes de vérification. Ces programmes ont été écrits à l'aide de Kermeta, mais par souci de clarté, nous présentons les algorithmes en pseudo-code. En effet, le méta-modèle DeployWare étant réalisé à l'aide de Kermeta, il est facile de développer des programmes Kermeta capable d'introspecter un modèle conforme à ce méta-modèle. De cette manière, il est possible d'implémenter les vérifications et de pouvoir vérifier n'importe quel modèle automatiquement en le passant en entrée des programmes validateurs. La version Kermeta des programmes de validation présentés dans la suite sont disponibles en téléchargement à l'adresse suivante : `http://www.lifl.fr/~dubus/DeployWare`.

4.2.3 Spécification formelle des propriétés

Nous décidons donc d'établir des prédicats de la logique du premier ordre pour spécifier des éléments de sémantique pour le langage de déploiement DeployWare. Avant de pouvoir définir les prédicats du langage DeployWare, il convient d'établir la grammaire du calcul des prédicats employée. Cette grammaire est définie par les quatre catégories suivantes.

- Les **variables** sont les symboles utilisés. L'ensemble de variables peut être infini ou restreint. Nous choisissons un ensemble infini de variables.
- Les **constantes** sont les valeurs existantes pour les variables définies. Nous considérons comme constante l'ensemble des instances de logiciels définies pour un système logiciel donné, représenté dans la suite par le symbole Ψ.
- Les **Prédicats** représentent les formules atomiques de notre langage. Nous définissons trois prédicats binaires : l'appartenance à un ensemble $\in$, l'égalité $=$ de termes, la différence $\neq$ de termes.
- Les **Fonctions** représentent le calcul d'un ensemble particulier. Nous définissons un certain nombre de fonctions dans notre langage, et ce afin de pouvoir exprimer la structure des logiciels dans nos prédicats. La spécification de ces fonctions est représentée sur le tableau 4.4.

Identifiant	Arité	Argument(s)	Description
type	Unaire	s : `SoftwareType` ou p : `Property`	Renvoie la valeur du type de logiciel/propriété en argument
inverse	Unaire	p : `ProcedureType` ou i : `InstructionType`	Renvoie la valeur inverse du type passé en argument (via les concepts de de Instruction/ProcedureRule)
procedures	Unaire	s : `SoftwareType`	Renvoie l'ensemble des procédures définies pour le type de logiciel en argument.
instructions	Unaire	p : `Procedure`	Renvoie l'ensemble des instructions définies pour la procédure en argument.
properties	Unaire	s : `SoftwareType`	Renvoie l'ensemble des propriétés définies pour le type de logiciel en argument.
typeDependences	Unaire	s : `SoftwareType`	Renvoie l'ensemble des types de logiciel dont dépend le type de logiciel en argument.
dependences	Unaire	s : `SoftwareInstance`	Renvoie l'ensemble des logiciels dont dépend le logiciel en argument.
instProvided	Unaire	s : `SoftwareInstance`	Renvoie l'ensemble des instructions fournies par le logiciel en argument.
importedProps	Unaire	s : `SoftwareType`	Renvoie l'ensemble des propriétés importées pour le logiciel en argument.
propValues	Unaire	s : `SoftwareInstance`	Renvoie l'ensemble des valeurs de propriété pour le logiciel en argument.
textValue	Unaire	pv : `PropertyValue`	Renvoie la valeur textuelle de la valeur de propriété.
host	Unaire	s : `SoftwareInstance`	Renvoie l'instance d'hôte de sur lequel le logiciel est déployé.

TAB. 4.4 – Fonctions définies pour la vérification de logiciels

4.2.4 Complétude au niveau des dépendances

La première vérification que nous allons détailler concerne les modèles définis par l'expert logiciel. En effet dans cette partie, lors de la création de nouveaux types de logiciels, une première difficulté consiste à définir la liste des dépendances d'un logiciel. Commettre une erreur dans cette liste de dépendances pose inévitablement un problème. Par exemple, oublier une dépendance technique ou métier d'un logiciel dans la déclaration du type d'un logiciel fera inévitablement échouer le déploiement de chaque instance de ce logiciel. À l'inverse, déclarer une dépendance inutile entrainera, pour chaque instance du logiciel déployé, le déploiement d'un logiciel sans utilité sur la machine. Cela peut avoir des conséquences désastreuses sur le temps de déploiement dans des contextes à très larges échelles. De plus, dans le cadre d'un déploiement sur terminaux mobiles, déployer des logiciels inutilement peut amener à saturer rapidement la mémoire déjà limitée des terminaux. Ainsi il est bénéfique pour l'administrateur qu'une vérification de la pertinence des dépendances soit effectuée au niveau des modèles de l'expert logiciel. Le critère que nous choisissons pour déterminer la pertinence d'une dépendance va concerner les instructions utilisées et les propriétés importées par le logiciel étudié. Pour juger de la pertinence d'une dépendance, nous allons étudier les instructions qui composent les procédures des logiciels, ainsi que les propriétés importées d'autres logiciels. Certains types d'instructions peuvent être fournis par des logiciels. Par exemple l'instruction `JavaExec` consistant à lancer un processus `java` représente une commande fournie par le type de logiciel JRE. Ainsi, on peut considérer qu'un logiciel A dépend d'un logiciel B si A utilise une instruction qui est fournie par le logiciel B. Aussi une dépendance entre deux logiciels peut s'exprimer par l'importation d'une propriété du logiciel duquel ce logiciel dépend. De cette manière, un logiciel A dépend de B si A importe une propriété initiale de B. La première contrainte à poser alors est que pour chaque logiciel de la base DeployWare, chaque dépendance doit être justifiée par l'utilisation, dans les procédures de ce logiciel, d'une instruction fournie par cette dépendance, ou d'une propriété importée. Dès lors cette contrainte peut s'exprimer comme suit :

$$\begin{aligned}
&\forall s \in \Psi, t = type(s), \forall d \in typeDependences(t) \\
&\quad \exists p \in procedures(t), \exists i \in instructions(p)(i \in providedInst(d)) \\
&\qquad \vee \quad \exists p_{imp} \in importedProps(t), \exists p_{init} \in properties(d)(p_{imp} = p_{init})
\end{aligned}$$

Cette formule traduit la vérification suivante : pour chaque logiciel de la base DeployWare, et pour chaque dépendance déclarée pour ce logiciel, soit il existe une instruction dans l'une des procédures de ce logiciel qui est liée à cette dépendance, à l'aide d'une règle de dépendance (`DependencyRule`), soit il existe une propriété importée par ce logiciel, qui appartient aux propriétés initiales du logiciel en dépendance. Par exemple, tout logiciel qui définit une dépendance avec la machine virtuelle Java doit posséder une instruction fournie par ce logiciel comme par exemple une instruction de type `JavaExec` qui correspond à l'instruction de lancement d'une JVM, `JarExec` qui correspond à l'utilisation de l'utilitaire de compression (*resp.* décompression) d'archive Java. Autre exemple, le logiciel du serveur JOnAS dépend du logiciel MySQL, car il importe une des propriétés de ce serveur de bases de données —*i.e.* le numéro du port de la base de données.

À l'inverse, pour chaque instruction composant les procédures d'un logiciel, le type de logiciel qui fournit cette instruction doit figurer parmi les dépendances déclarées pour ce type de logiciel. On exprime cette contraposée de la manière suivante.

$$\forall s \in \Psi, \forall p \in procedures(type(s)), \forall i \in instructions(p)$$
$$\exists s_{dep} \in typeDependences(type(s))(i \in providedInst(s_{dep}))$$

Cette formule traduit la vérification suivante : dans un type de logiciel, pour chaque procédure, la présence d'une instruction doit être appuyée par l'ajout dans les dépendances du logiciel qui fournit ce type d'instruction. Cette propriété ne s'applique néanmoins pas aux instructions du Shell fournies directement par la machine hôte.

De la même manière, pour chaque propriété importée par un logiciel, le logiciel pour lequel cette propriété est initiale doit être déclaré dans les dépendances. On peut exprimer cela de la manière suivante.

$$\forall s \in \Psi, \forall importedP \in importedProperties(type(s))$$
$$\Rightarrow \exists s_{dep} \in typeDependences(type(s))(importedP \in properties(s_{dep}))$$

Cette formule traduit la vérification suivante : dans un type de logiciel, la présence d'une propriété importée doit être appuyée par l'ajout dans les dépendances du logiciel pour qui cette propriété est initiale.

Dans notre méta-modèle, il est permis de définir de nouvelles instructions (*e.g.* la définition d'un type d'instruction `JavaExec` pour le lancement de JVM) et d'associer ces instructions aux logiciels qui fournissent ces instructions sur la machine (par exemple associer le type d'instruction `JavaExec` au type de logiciel `JAVA.JRE`, puisque ce logiciel fournit cette instruction). Cette association est matérialisée par le concept `DependencyRule` du méta-modèle, qui référence à la fois le type d'instruction concernée et le type de logiciel qui la fournit. De cette manière, il est possible de vérifier qu'un logiciel qui utilise dans ses procédures une instruction de type `JavaExec` dépende bien du logiciel `JAVA.JRE` qui fournit cette instruction, sinon il est incorrect. À l'inverse si l'on considère un logiciel qui dépend de `JAVA.JRE`, mais qui n'utilise pas d'instruction de type associé à `JAVA.JRE` dans ses procédures, alors une incohérence est détectée.

Dans ce cas précis, il est impossible d'affirmer avec exactitude qu'il s'agisse d'une erreur. En effet, on ne peut pas affirmer que cette dépendance est inutile car la dépendance d'un logiciel envers un autre n'implique pas toujours l'utilisation d'une instruction. Par exemple, un serveur JEE JOnAS n'utilise pas directement l'exécutable `java` (*i.e.* `JavaExec`) dans sa procédure de déploiement. Néanmoins sans une JVM installée, JOnAS ne peut pas être déployé. Cette vérification sert donc d'avertissement pour l'expert logiciel, mais ne relève pas une erreur.

Le listing 4.1 donne l'extrait en pseudo-code de l'algorithme permettant de vérifier si un type de logiciel est correct au niveau de ses dépendances, en utilisant la structure du méta-modèle DeployWare. La première opération `vérifierDépendancesRequises` vérifie les dépendances manquantes, pour un type de logiciel donné. La deuxième opération `vérifier-DépendancesInutiles` permet de vérifier si toutes les dépendances sont justifiées pour un type de logiciel donné.

```
FONCTION vérifierDépendancesRequises(soft_type: SoftwareType)
{
  POUR CHAQUE Procedure p APPARTENANT À soft_type.procedures
    POUR CHAQUE Instruction i APPARTENANT À p.instructions
      SI i.type.attachedInst != référence_nulle
      ALORS
        required_soft = i.type.attachedInst.isSWOf
        SI (required_soft N'APPARTIENT PAS À soft_type.dependences) ET NON
            dépendanceDeclaréeDans(soft_type.inherits)
        ALORS
          AFFICHER(Dépendance required_soft requise par instruction i.id mais non
              présente)
        FIN_SI
      FIN_SI
    FIN_POUR
  FIN_POUR

  POUR CHAQUE Property prop APPARTENANT À soft_type.imported_properties
    BOOLEEN resultat = FALSE
    POUR CHAQUE SoftwareType dep APPARTENANT À soft_type.dependences
      SI prop APPARTIENT À dep.initial_properties
      ALORS
        resultat = TRUE
      FIN_SI
    FIN_POUR
    SI resultat = FALSE
    ALORS
      AFFICHER(La propriété prop.id est importée sans que sa dépendance ne soit
          déclarée)
    FIN_SI
  FIN_POUR
}

FONCTION vérifierDépendancesInutiles(soft_type: SoftwareType)
{
  POUR CHAQUE SoftwareType soft_dependences APPARTENANT À soft_type.dependencies
    BOOLEEN resultat_instruction = FALSE
    POUR CHAQUE Procedure p APPARTENANT À ( (soft_type.procedures) UNION (soft_type.
        inherits.procedures) )
      POUR CHAQUE Instruction i APPARTENANT À p.instructions
        SI i.type.isInstOf != référence_nulle
        ALORS
          SI i.type.isInstOf.isSWOf == soft_dependences
          ALORS resultat_instruction = TRUE
          FIN_SI
        FIN_SI
      FIN_POUR
    FIN_POUR

    POUR CHAQUE Property p APPARTENANT À soft_type.imported_properties
      BOOLEEN resultat_propriete = FALSE
      POUR CHAQUE SoftwareType dep APPARTENANT À soft_type.dependencies
        SI p APPARTIENT À dep.initial_properties
        ALORS
          resultat_propriete = TRUE
        FIN_SI
      FIN_POUR
    FIN_POUR
    SI non(resultat_instruction) ET non(resultat_propriete)
    ALORS
      AFFICHER(Dépendance soft_dependances.name non justifiée !)
```

```
    FIN_SI
  FIN_POUR
}
```

Listing 4.1 – Extrait de pseudo-code pour la vérification des dépendances d'un type de logiciel

4.2.5 Cohérence et conformance des instances par rapport aux types

Une autre vérification à faire lors du déploiement d'un système distribué consiste à vérifier que les dépendances d'une instance de logiciel sont bien résolues. En effet, deux types de dépendances existent : les dépendances techniques et les dépendances liées au métier de l'application. Si un logiciel est déployé sans que l'une de ses dépendances techniques ne soit installée, le déploiement échouera immédiatement. En revanche dans le cas d'une dépendance métier non résolue, le logiciel sera correctement déployé, mais l'exécution du métier de l'application sera source d'erreurs. Ainsi lorsqu'un système est composé de logiciels, il faut vérifier que les dépendances techniques requises par les types de ces logiciels sont bien résolues, ainsi que les dépendances métier.

Cette constrainte sur la pertinence des dépendances au niveau des types de logiciel peut s'exprimer formellement comme suit :

$$\forall s \in \Psi, \forall d \in typeDependences(type(s)), \exists dep \in dependences(s)(type(dep) = d)$$

En langage naturel, cela veut dire que pour toute instance de logiciel à déployer, pour tous les types de dépendances déclarés à l'aide l'association *dependencies* pour ce type du logiciel, il doit exister, dans les dépendances déclarées à l'aide de l'association *dependsOn* de cette instance, un logiciel de chaque type. Par exemple, le type de logiciel `JOnAS.SERVER` déclare une dépendance vers les types de logiciels JRE et MySQL. Cela signifie que chaque instance de logiciel `JOnAS.SERVER` doit déclarer une instance de JRE et une instance de MySQL dans les dépendances *dependsOn*. On renforce cette contrainte avec la suivante :

$$\forall s \in \Psi, \forall dep \in dependences(s), \exists d \in typeDependencies(type(s))(d = type(dep))$$

Ce qui signifie que si une dépendance est déclarée pour une instance du logiciel, il faut que le type du logiciel que l'on ajoute dans les dépendances fasse partie des types de logiciels dont dépend le logiciel en question, sinon la dépendance est inutile. Par exemple, ajouter une dépendance vers une instance du logiciel Ant dans l'association *dependsOn* de JOnAS.Server n'est pas autorisé, car le type JOnAS.SERVER ne déclare pas de dépendances vers ce type de logiciel Ant.

Le listing 4.2 donne l'extrait de pseudo-code permettant de vérifier si un type de logiciel est correct au niveau de ses dépendances. Un algorithme similaire peut être appliqué sur les valeurs de propriétés pour vérifier que chaque valeur de propriété correspond bien à une propriété requise dans la définition du type de logiciel correspondant. Nous ne détaillons pas cet algorithme qui est très proche de celui présenté sur 4.2. L'opération `vérifierDépendances-Résolues()` permet de vérifier que les dépendances déclarées pour une instance de logiciel, passée en paramètre, sont bien conformes aux dépendances déclarées dans le type du logiciel étudié.

```
FONCTION vérifierDépendancesRésolues (soft_instance : SoftwareInstance)
{
  SoftwareType si_type = soft_instance.type
  POUR CHAQUE SoftwareType depType APPARTENANT À si_type.dependencies
    BOOLEEN resultat = FAUX
    POUR CHAQUE SoftwareInstance depDeclared APPARTENANT À soft_instance.dependsOn
      SI depDeclared.type == depType
      ALORS
        resultat = VRAI
      FIN_SI

      SI depDeclared.type N'APPARTIENT PAS À si_type.dependencies
      ALORS
        AFFICHER(depDeclared.id de type depDeclared.type parait etre une dépendance
            inutile)
      FIN_SI
    FIN_POUR
    SI non(resultat)
      ALORS AFFICHER (Cette instance de logiciel nécessite une dépendance de type
          depType.name)
    FIN_SI
  FIN_POUR
}
```

Listing 4.2 – Extrait de pseudo-code de vérification des dépendances d'une instance de logiciel

Nous souhaitons également effectuer des vérifications des valeurs de propriétés des logiciels qui sont déployés sur la même machine hôte. En effet, il faut vérifier que, sur une machine donnée, les valeurs de propriétés de même type, quelle que soit l'instance de logiciel qu'elles concernent, soient différentes. Dans le cas contraire, il serait possible de modéliser deux serveurs quelconques déployés sur une même machine qui attendraient des connexions sur le même port par exemple. Autre exemple, deux logiciels installés au même endroit et possédant des fichiers aux noms identiques (*e.g.* des fichiers tels les `build.xml` de Ant sont très fréquents).

Ainsi cette propriété peut être modélisée à l'aide de la formule suivante.

$$\forall s_1, s_2 \in \Psi, s_1 \neq s_2, \forall pValue_1 \in propValues(s_1), \forall pValue_2 \in (propValues(s_2)),$$
$$type(prop(pValue_1)) = type(prop(pValue_2) \wedge host(s_1) = host(s_2)$$
$$\Rightarrow textValue(pValue_1) \neq textValue(pValue_2)$$

En langage naturel, cela signifie que toutes les valeurs de propriétés de même type sur des logiciels différents déployés sur la même machine doivent être différentes également. Sur le listing 4.3 figure le pseudo-code du programme effectuant cette vérification. L'opération `vérifierInterférencePropriétés` permet de vérifier sur une machine hôte, donnée en paramètre, que tous les logiciels déployés sur cette machine utilisent des ensembles de valeurs de propriétés de même types disjoints.

```
FONCTION vérifierInterférencePropriétés (host: HostInstance)
{
  TABLE_HACHAGE mémoire = (vide) ;

  POUR CHAQUE SoftwareInstance si APPARTENANT À host.instances
    POUR CHAQUE PropertyValue pv APPARTENANT À si.props
```

```
      SI non(mémoire.CONTIENT_CLE(pv.type))
      ALORS
        mémoire.ajouter(pv.type,pv.textValue)
      SINON
        SI mémoire.obtenir(pv.type) == pv.textValue
          ALORS AFFICHER (Attention : la propriété pv.type.name
                          du logiciel si.id possède la meme
                          valeur et le meme type qu'une autre)
        FIN_SI
      FIN_SI
    FIN_POUR
  FIN_POUR
}
```

Listing 4.3 – Extrait de pseudo-code Kermeta pour la vérification de non-interférence des propriétés de logiciels

4.2.6 Réversibilité des procédures et instructions

Un déploiement ne peut être valide que s'il possède la particularité d'être intégralement réversible. Par intégralement réversible, nous entendons que chaque logiciel composant le système logiciel soit repliable [12], et donc le système doit pouvoir être replié complètement et laisser le domaine de déploiement dans l'état initial. Cette propriété est particulièrement importante dans le cadre des environnements ouverts distribués. En effet, si l'on considère le déploiement d'un système logiciel dans un environnement de type informatique ubiquitaire, il est dangereux de ne pas assurer que toute action effectuée sur un terminal mobile n'est pas annulable. Par exemple, lorsque le terminal mobile entre dans le domaine de déploiement, plusieurs logiciels vont être déployés sur celui-ci. Si lorsqu'il quitte le domaine de déploiement, tous les logiciels ne sont pas repliés, alors le terminal se trouve pollué avec des éléments de logiciels dont il n'a plus besoin. En outre si un même terminal entre dans le domaine de déploiement, le quitte, puis recommence cette manipulation un grand nombre de fois, il risque de se trouver confronté à un problème de saturation de mémoire. Il existe un *effet de bord* indésirable du déploiement pour ces machines hôtes. Dans un contexte de grille de calcul, le problème est également important car la *pollution* des machines par effet de bord atteint des milliers de machines.

Un moyen de vérification que chaque logiciel est repliable consiste tout simplement à veiller à ce que les procédures de déploiement d'un logiciel soient bien associées deux à deux : une pour le déploiement avec une pour le repliement. Un exemple simple est le fait de vérifier qu'une procédure de type *démarrage* est bien associée dans le logiciel à une procédure de type *arrêt*. Un tel type de contrainte s'exprime de cette manière.

$$\forall s \in \Psi, s_{type} = type(s), \forall p \in procedures(s_{type})$$
$$\exists p' \in procedures(s_{type})(type(p') = inverse(type(p)))$$

En langage naturel, cela signifie que pour chaque procédure modélisée dans un type de logiciel, une procédure représentant l'intention inverse, donc de type inverse à la procédure initiale, doit être également présente.

[12] replier étant l'action symétrique de déployer.

Dans le méta-modèle DeployWare, les moyens d'opérer de telles vérifications nous sont offerts par le biais du concept **ProcedureRule** qui permet d'associer deux **ProcedureType**. Si au niveau de l'expert logiciel on définit qu'une procédure de type *start* est associée à une procédure inverse de type *stop*, on pourra vérifier que tout logiciel comprenant une procédure de type *start* contient bel et bien la procédure inverse de type *stop*.

Le listing 4.4 donne l'extrait de pseudo-code permettant de vérifier si un type de logiciel est correct en terme de réversibilité du déploiement. L'opération `vérifierProcédure-Inverses` vérifie que les procédures définies dans un type de logiciel, donné en paramètre, sont bien réversibles.

```
FONCTION vérifierProcédureInverses(soft_type : SoftwareType)
{
  POUR CHAQUE Procedure p1 APPARTENANT À soft_type.procedures
    BOOLEAN resultat = FAUX
    SI p1.type.isRPartOf != référence_nulle
    ALORS
      POUR CHAQUE Procedure p2 APPARTENANT À soft_type.procedures
        SI p2.type.isLPartOf == p1.type.isRPartOf
        ALORS
          resultat = VRAI
        FIN_SI
    FIN_SI

    SI p1.type.isLPartOf != référence_nulle
    ALORS
      POUR CHAQUE Procedure p2 APPARTENANT À soft_type.procedures
        SI p2.type.isRPartOf == p1.type.isLPartOf
        ALORS
          resultat = VRAI
        FIN_SI
      FIN_POUR
    FIN_SI

    SI non(resultat)
    ALORS
      AFFICHER(Procedure p1.name ne possède pas son inverse)
    FIN_SI
  FIN_POUR
}
```

Listing 4.4 – Extrait de pseudo-code pour la vérification de la réversibilité du déploiement

Pour deux procédures déclarées comme étant de types inverses, il faut vérifier que les instructions qu'elles exécutent représentent effectivement des actions qui s'annulent pour éviter les effets de bord. Toutes les instructions dans une procédure doivent donc trouver leur instruction inverse dans la procédure inverse. De cette manière, on peut assurer que les procédures de déploiement écrites sont symétriques. Grâce au méta-modèle DeployWare, il est possible d'effectuer cette vérification statiquement sur le même principe que dans les procédures.

La représentation formelle de cette contrainte est la suivante :

$$\begin{aligned}&\forall s \in \Psi, \forall p \in procedures(type(s)), \forall i \in instructions(p),\\ &\quad \exists p_{inv} \in procedures(type(s)), \exists i_{inv} \in instructions(p_{inv})\\ &\quad (type(p_{inv}) = inverse(type(p)) \wedge type(i_{inv}) = inverse(type(i)))\end{aligned}$$

En langage naturel, cela revient à vérifier que pour n'importe quel logiciel, si une instruction est modélisée dans une procédure, alors une instruction à l'intention inverse (donc de type inverse) doit être modélisée dans la procédure de type inverse à la procédure initiale.

L'expert logiciel qui définit des instructions attachées à un logiciel peut associer à l'aide d'une **InstructionRule** deux **InstructionTypes** comme étant des types d'instruction inverses, et pour chaque couple de procédures inverses d'un logiciel, il suffit de vérifier que si l'une de ces instructions est utilisée dans une procédure, l'instruction inverse est présente dans la procédure inverse.

Pour vérifier l'exacte opposition de ces instructions, la seule comparaison des types n'est pas suffisante. Il est également nécessaire de comparer les arguments des instructions. Par exemple, l'instruction `SetVariable(PATH,/home/user/.../)` n'est pas annulée par l'instruction `UnsetVariable(JONAS_HOME)`, qui est pourtant du type inverse, mais bel et bien par l'instruction `UnsetVariable(PATH)`. Il est donc nécessaire de vérifier les arguments des instructions. Néanmoins, cette vérification est spécifique aux types d'instructions concernés. Pour l'exemple de `SetVariable` la vérification consiste à assurer que le premier paramètre de `SetVariable` est identique au paramètre de `UnsetVariable`. Mais cette contrainte peut changer en fonction des types d'instruction. La spécification de la contrainte sur les arguments se fait donc au niveau de l'`InstructionRule` correspondante. Le champ *pattern* de l'`InstructionRule`, dans le cas de `SetVariable` est positionné à la valeur `@X@,@Y@`. Cela signifie que l'ensemble des arguments de l'instruction est considéré comme deux blocs textuels (identifiés par les variables X et Y) séparés par une virgule. Le label inverse *patternInv* est positionné à la valeur `@X@`, ce qui signifie que l'ensemble des arguments de l'instruction `UnsetVariable` doit correspondre exactement au bloc X de `SetVariable`.

Le listing 4.5 donne l'extrait de pseudo-code permettant de vérifier si deux instructions sont bel et bien inverses. L'opération `vérifierInstructions` vérifie que les instructions définies dans deux procédures supposées inverses, données en paramètres, sont bien inverses en terme d'opération élémentaires effectuées.

```
FONCTION vérifierInstructions(proc1,proc2 : Procedure)
{
  POUR CHAQUE Instruction i1 APPARTENANT À proc1.instructions
    BOOLEAN resultat = FALSE
    SI (i1.type.IsRPartOf == référence_nulle) ET (i1.type.IsRPartOf ==
        référence_nulle)
    ALORS resultat = TRUE
    SINON
      SI i1.type.isRPartOf /= référence_nulle
      ALORS
        POUR CHAQUE Instruction i2 APPARTENANT À proc2.instructions
          InstructionRule iRule = i1.type.isRPartOf
          SI i2.type.isLPartOf == i1.type.isRPartOf ET EVALUER_PATTERN_MATCHING(iRule
              ,i1.label,i2.label)
          ALORS
            resultat = TRUE
          FIN_SI
        FIN_POUR
      SINON
        SI i1.type.isLPartOf /= référence_nulle
        ALORS
          POUR CHAQUE Instruction i2 APPARTENANT À proc2.instructions
            InstructionRule iRule = i1.type.isLPartOf
```

```
            SI i2.type.isRPartOf == i1.type.isLPartOf ET EVALUER_PATTERN_MATCHING(
                iRule,i1.label,i2.label)
            ALORS
              resultat = TRUE
            FIN_SI
          FIN_POUR
        FIN_SI
      FIN_SI
    SI non(RESULTAT)
    ALORS
      AFFICHER (Instruction i1.id ne possède pas d'intruction inverse dans proc2.name
          )
    FIN_SI
  FIN_POUR
}
```

Listing 4.5 – Extrait de code Kermeta de vérification d'instructions inverses

4.3 Conclusion/Synthèse

Dans ce chapitre, nous avons détaillé la première partie de notre contribution, pour la modélisation du déploiement de la couche intergicielle d'un système logiciel réparti. Cette contribution repose sur le méta-modèle DeployWare.

Ce méta-modèle est découpé en deux paquetages, ce qui permet de renforcer la séparation des préoccupations entre administrateur système et expert logiciel. Ce méta-modèle permet aux administrateurs systèmes (*resp.* experts logiciels) de monter en abstraction dans la description du déploiement de la couche intergicielle (*resp.* des logiciels de leur technologie) tout en utilisant un langage dédié pour les rôles respectifs du déploiement. Les concepts de ce méta-modèle nous ont permis de valider le méta-modèle en modélisant une multitude de logiciels de technologies différentes représentées sur le tableau 4.5.

Le travail de l'état de l'art se rapprochant le plus de notre contribution est l'approche à base de méta-modèle proposée dans ORYA. Notre approche se démarque en définissant une sémantique plus précise des concepts de notre méta-modèle. Ainsi une démarche de transformation de modèles est rendue possible, et les modèles peuvent être vérifiés statiquement.

En outre, notre méta-modèle permet d'exprimer les concepts de n'importe quel logiciel, et indépendamment des spécificités de la machine sur laquelle sera déployé ce logiciel. Ainsi l'hétérogénéité des technologies et des supports matériels est prise en charge dans notre approche.

Enfin, l'ajout de concepts tels que les règles de dépendance, ou règles de procédure/instruction, rendent possible un certain nombre de vérifications statiques. Il s'agit d'une contribution significative dans notre domaine. En effet, dans la plupart des modèles de déploiement existants, aucune vérification n'est possible en dehors des vérifications structurelles. Nous avons défini un certain nombre de propriétés que les modèles doivent remplir, et nous avons implémenté les programmes de vérifications pour chacune de ces propriétés. Cinq propriétés sont identifiées dans cette thèse, et le détail des programmes de vérifications ont été décrits. Néanmoins cela constitue un point de départ pour la démarche. D'autres vérifications peuvent être

Systèmes à base d'architectures orientées services	Processus BPEL, Moteurs ActiveBPEL et Orchestra, Applications SCA, Apache Tuscany SCA, Composants et Assemblages JBI, Conteneur JBI PEtALS
Systèmes à base de JEE	Composants EAR JAR WAR et RAR, Serveurs d'applications Apache Geronimo, JOnAS, JBoss, Sun Glassfish, systèmes auto-gérés JADE/JASMINe
Systèmes Web	Démon HTTP Apache, Conteneur de servlet Tomcat
Systèmes à base de CORBA	Intergiciels CORBA, OpenCCM et applications
Systèmes à base de Fractal	Julia, Fractal ADL et RMI
Systèmes à base de Java	Java Runtime Environnement (JRE), JamVM JRE pour PDA Linux
Systèmes de bases de données	Serveur de bases de données MySQL
Services réseaux	Registre OpenLDAP
Systèmes d'exploitation	Tous type d'OS virtuels à l'aide de QEMU

TAB. 4.5 – Tableau des différents types de logiciels modélisés avec DeployWare

implémentées, et le principal apport de notre travail pour ce point réside dans la définition d'une démarche de vérification.

Dans le chapitre suivant, nous étudions comment ces concepts peuvent être projetés vers une machine virtuelle de déploiement capable d'exécuter automatiquement le déploiement de la couche intergicielle définie par l'administrateur système et l'expert réseau. Même si cette projection reste à ce jour informelle, nous allons voir comment concevoir une plate-forme intergicielle de déploiement qui prend en entrée des descripteurs très proche des modèles DeployWare.

Chapitre 5

Machine virtuelle DeployWare

Sommaire

Dans ce chapitre, nous détaillons la machine virtuelle d'exécution vers laquelle sont projetés les modèles DeployWare présentés dans le chapitre précédent. En effet, une fois que l'administrateur a modélisé puis effectué les vérifications sur son système logiciel, il procède au déploiement effectif du système décrit. Dans le cadre d'une démarche d'ingénierie dirigée par les modèles telle que nous la mettons en œuvre, il est nécessaire de définir une plate-forme d'exécution du déploiement, ainsi qu'un modèle de cette plate-forme. En effet, les PIM définis conformément à notre spécification du chapitre 4 doivent être ensuite projetés vers des modèles spécifiques à une plate-forme d'exécution donnée, afin d'être exécutés.

Pour cela, nous conçevons une plate-forme intergicielle générique de déploiement de systèmes distribués à base de composants Fractal. Un langage pour exprimer des déploiements sur cette plate-forme est également présenté.

Une vue d'ensemble du fonctionnement de la machine virtuelle que nous proposons est donnée sur la figure 5.1. Une machine virtuelle est définie en entrée par un descripteur de déploiement. Une machine virtuelle représente donc un déploiement de système logiciel. Ces machines sont constituées des composants de logiciels correspondants. Ces composants encapsulent le protocole de déploiement des logiciels à déployer ; ils utilisent la couche de composants d'accès aux machines hôtes pour exécuter à distance les instructions élémentaires de déploiement. Les composants de logiciels sont ensuite administrables soit de manière autogérée, soit de manière manuelle à l'aide d'un explorateur dédié.

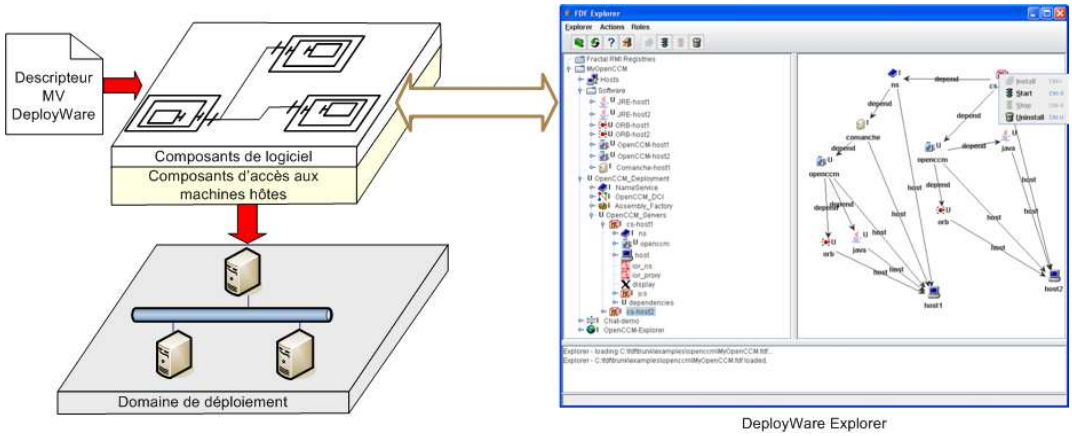

FIG. 5.1 – Vue d'ensemble de la machine virtuelle DeployWare

Le but initial d'une machine virtuelle est de fournir un cadre d'exécution pour les processus de déploiement décrits à l'aide de modèles DeployWare. Ainsi une projection des concepts du méta-modèle DeployWare vers le format d'entrée de la machine virtuelle doit être établi. À ce jour, cette projection reste manuelle. Néanmoins les concepts de la machine virtuelle restent très proches de ceux du méta-modèle.

Dans un premier temps, nous présentons l'architecture d'un composant réifiant le processus de déploiement d'un logiciel ainsi que le composant réifiant l'accès aux machines hôtes. Cette architecture correspond donc à la projection dans la machine virtuelle du concept de type de logiciel. Enfin, nous montrons que l'utilisation de motifs architecturaux tels que définis dans [61] rend la construction d'une machine de déploiement de systèmes distribués très simple, et nous définissons alors un langage simple et très proche des concepts du méta-modèle pour la construction des machines de déploiement DeployWare.

Le langage de description d'architectures Fractal ADL, utilisé pour décrire des architectures à base de composants Fractal, ne nous parait pas approprié pour la construction de machines virtuelles. En effet, utiliser un langage généraliste de description d'architectures pour construire une machine virtuelle de déploiement contraint le concepteur à avoir de solides

connaissances dans ce paradigme. Nous souhaitons donc simplifier l'utilisation de ce langage, pour le restreindre à la seule description de la composition des logiciels d'un système.

Nous donnerons également les détails sur les composants de la machine virtuelle qui rendent les déploiements sensibles aux environnements ouverts distribués. Ces composants sont en charge de l'exécution des politiques d'autonomie. Puis les améliorations de la machine virtuelle permettant d'effectuer des déploiements à large échelle seront également détaillées.

5.1 Composants de la machine virtuelle

Dans cette section, nous étudions la réalisation dans la machine virtuelle de deux concepts principaux de DeployWare, à savoir les logiciels et les machines hôtes. En effet, dans le méta-modèle, un système logiciel réparti correspond à un ensemble de machines hôtes et un certain nombre de logiciels devant être déployés sur ces machines. Dès lors, nous nous penchons dans un premier temps sur l'implémentation de la couche d'accès aux machines hôtes. Cette couche d'accès, implémentée à l'aide de composants, doit être réalisée par l'expert réseau. Ensuite, nous nous penchons sur l'implémentation et l'assemblage de composants représentant le protocole de déploiement de logiciels. Ces composants doivent permettre d'exécuter les différentes procédures et instructions de déploiement des logiciels telles qu'elles sont décrites dans le méta-modèle DeployWare.

Nous avons fait le choix d'utiliser le paradigme de programmation par composants pour construire les machines virtuelles DeployWare. Ce choix est justifié par la volonté de séparer au maximum les différentes fonctionnalités de la machine virtuelle. Comme nous l'exposons en détails dans la suite, nous souhaitons séparer l'accès aux machines hôtes des protocoles de déploiement des logiciels du système. Le paradigme de programmation par composants représente la meilleure alternative à ce jour pour maintenir une séparation claire des préoccupations au sein de la machine virtuelle DeployWare. L'utilisation d'interfaces génériques, réalisées par plusieurs implémentations, est intensive dans DeployWare [1]. En effet, il est possible en utilisant cette technique, de rendre possible la généricité pour les protocoles d'accès aux machines, des protocoles de transfert de fichiers, et des différents interpréteurs de commandes par exemple. Par exemple, le mécanisme de protocole de transfert de fichier va être représenté par un composant aux interfaces bien définies en terme d'opérations, et chaque implémentation existante contiendra le code capable d'envoyer la requête à la machine hôte conformément au protocole concerné. De plus, une préoccupation essentielle pour la construction de machines virtuelles est l'extensibilité de la liste des différents moyens d'accès implémentés. De nouvelles technologies apparaissent constamment, et il est important que DeployWare puisse rester utilisable au fil des avancées technologiques. En utilisant une architecture à composants pour la conception de ces machines virtuelles, il est possible d'ajouter de nouveaux protocoles, de nouveaux interpréteurs de commandes, etc. simplement en développant de nouveaux types de composants et en les intégrant à la bibliothèque de composants de DeployWare. Enfin, les applications à base de composants sont souvent facilement reconfigurables à l'exécution, grâce à leur structure explicitement identifiée à l'exécution, ainsi qu'à la séparation claire des préoccupations entre les composants. Cette possibilité de reconfiguration des machines virtuelles est une propriété

[1] Ce mécanisme est illustré dans la section 5.1.1.

indispensable pour la construction de machines virtuelles de déploiement en environnements ouverts. Le déploiement se poursuivant au cours du temps, la machine de déploiement doit être étendue, donc reconfigurée dynamiquement.

Nous avons fait le choix du modèle de composants Fractal et de Julia, son implémentation de référence en Java, comme canevas de développement de la machine virtuelle DeployWare.

La première raison est que Fractal est un canevas générique et extensible qui permet de concevoir de manière modulaire une application, quelle que soit sa nature. En effet le modèle Fractal a d'ores et déjà fait ses preuves en étant l'outil de développement de nombreux logiciels au sein du consortium Objectweb notamment. Parmi ces logiciels on trouve le canevas de communication DREAM, le gestionnaire transactionnel GoTM, ou encore le serveur d'application ESB PEtALS. Le serveur JEE JOnAS possède également quelques modules développés à l'aide du modèle Fractal.

La seconde raison qui motive notre choix repose sur le caractère réflexif dynamique du modèle Fractal. En effet une application à base de composants Fractal est entièrement reconfigurable pendant son exécution, ce qui est une propriété indispensable pour envisager des reconfigurations dynamiques, et un déploiement en environnements ouverts distribués.

5.1.1 Couches d'accès aux machines hôtes

Dans la définition des rôles que nous avons introduite dans le chapitre 4, il reste un rôle dont les fonctions n'ont pas encore été présentées : il s'agit de celui d'**expert réseau**. Ce dernier est chargé d'implémenter les moyens d'accès aux machines et de les rendre disponibles pour l'administrateur système en charge de déployer des logiciels sur ces machines. Les composants d'accès aux machines doivent être génériques et réutilisables, afin de pouvoir construire des couches d'accès en assemblant les différents composants. Nous définissons donc un cadre de conception de composants réutilisables pour implémenter les moyens d'accès aux machines hôtes. Dans la suite, nous appelons ces composants d'accès aux machines des *adaptateurs*. Pour un type d'accès donné (transfert de fichiers, commandes à distance, etc.) un composant adaptateur fournit toujours les mêmes interfaces et les mêmes opérations, qui correspondent aux opérations génériques proposées par le type d'accès. Par exemple l'interface d'un composant d'accès de type transfert de fichiers possède deux opérations : `download(source,destination)` et `upload(source,destination)`, qui correspondent respectivement au téléchargement d'un fichier en provenance de la machine distante, et au chargement d'un fichier local sur la machine distante.

En revanche, il *adapte* l'appel de l'opération aux spécificités du type d'accès disponible sur la machine (des requêtes FTP ou SCP seront exécutées par exemple lors de l'appel de l'opération `upload()` du composant de transfert d'un fichier). En assemblant des instances de ces composants, l'administrateur système est capable de construire une couche de communication complète (*i.e.* envoi de commandes, téléchargement de fichiers, etc.) avec les machines sur lesquelles il souhaite déployer des logiciels. Les composants de logiciel, décrits dans la suite, vont utiliser cette couche de communication avec la machine pour exécuter les instructions des procédures de déploiement.

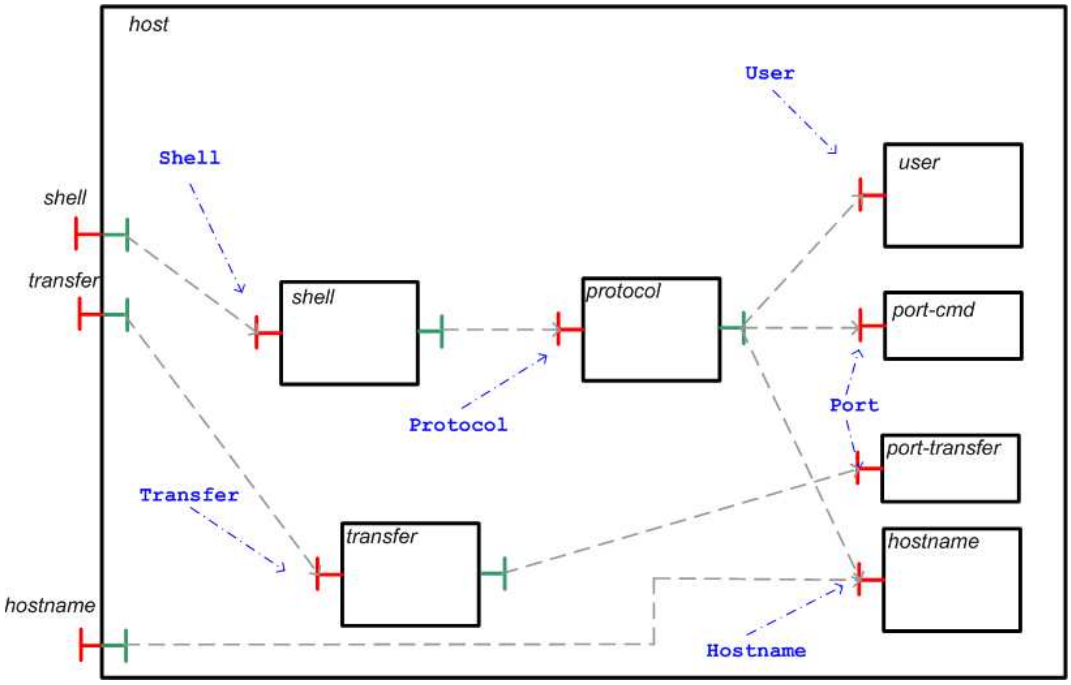

FIG. 5.2 – Composants d'abstraction des machines hôtes de la machine virtuelle DeployWare

Les instructions représentent des commandes à envoyer sur la machine hôte distante. Cependant, ces commandes doivent être envoyées en utilisant différents protocoles selon l'accès que fournit la machine (SSH, Telnet ou autre) et les commandes sont formattées en fonction du Shell proposé par la machine (de type SH, CSH ou le Shell de Microsoft Windows). De la même manière, le téléchargement des fichiers sur la machine doit se faire conformément au protocole de transfert de fichiers supporté par la machine (SCP, FTP, etc.). Ce sont les interfaces des composants, indépendamment de leur implémentation, qui vont nous permettre de gérer de manière transparente cette adaptation à l'environnement sous-jacent. En effet, les intentions vis-à-vis de la machine sont toujours les mêmes (*i.e.* créer un répertoire, naviguer dans le système de fichier, lancer un exécutable, envoyer un fichier). Ainsi ce sont ces opérations génériques qui vont être regroupées dans les interfaces des composants d'accès et différentes implémentations existent pour chaque interface d'accès. Cette architecture est représentée sur la figure 5.2. Un composant de machine hôte est composé de cinq composants importants. Le composant *shell* est responsable du formattage des commandes. Ces commandes sont ensuite déléguées au composant *protocol* chargé de les exécuter sur la machine distante de manière adéquate. Le nom d'hôte est encapsulé dans le composant *hostname*, et le nom de connexion (*i.e.* le login) est contenu dans le composant *user*. Enfin, le composant *transfer* encapsule la logique de transfert de fichiers vers ou en provenance de la machine hôte.

Les API des différents types de composants de la couche d'accès sont représentés sur la figure 5.3.

Nous avons identifié trois opérations fournies par l'interface `Shell`. La première opé-

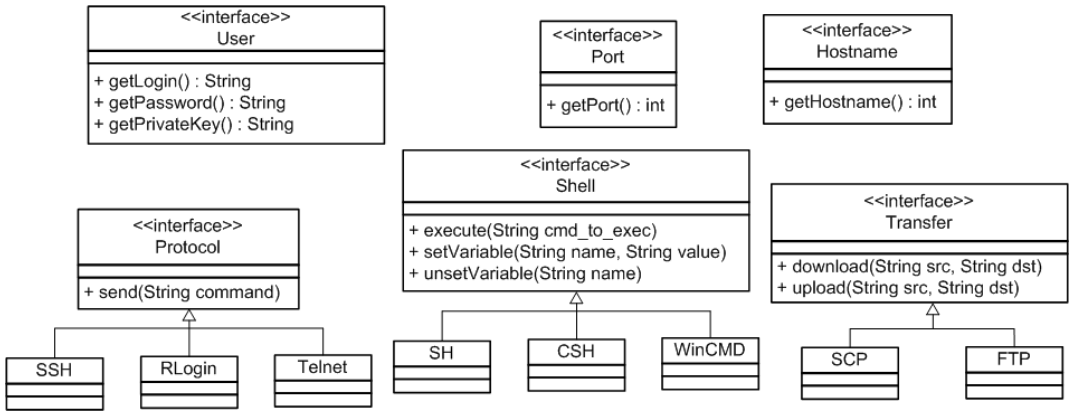

FIG. 5.3 – Interfaces des composants de machines hôtes

ration correspond à l'appel d'une commande (`execute(String cmd_to_exec)`) qui prend comme paramètre la commande à lancer. La deuxième opération correspond au positionnement d'une variable d'environnement (`setVariable(String name, String value)`) qui prend comme paramètres l'identifiant de la variable ainsi que la valeur à attribuer à cette variable. Enfin, la troisième opération correspond à l'opération inverse, c.-à-d. la suppression d'une variable d'environnement (`unSetVariable(String name)`). Notons qu'il n'existe pas d'opération pour tuer un processus, car cette opération consiste à exécuter une commande (typiquement `kill` en Bourne Shell) dont l'action est de tuer un processus. Cette opération est donc un type particulier d'appel de l'opération `execute()`. Il existe plusieurs implémentations de l'interface `Shell` dans la machine virtuelle DeployWare, par exemple pour le Bourne Shell (SH) ou l'invite de commandes de Windows.

Nous avons identifié une seule opération fournie par l'interface `Protocol` Cette opération `send(String cmd)` correspond à l'envoi d'une commande à la machine en utilisant le protocol adéquat (*i.e.* SSH, Telnet, etc.). Il existe plusieurs implémentations de l'interface `Protocol` dans la machine virtuelle DeployWare, par exemple pour le protocole SSH ou Telnet.

Nous avons identifié deux opérations fournies par l'interface `Transfer`. La première opération correspond au téléchargement d'un fichier de la machine hôte distante vers la machine locale (`download(String source, String destination)`) qui prend comme paramètres le chemin du fichier à télécharger sur la machine hôte distante, et le chemin local du répertoire dans lequel sauvegarder le fichier téléchargé. La seconde opération correspond au chargement d'un fichier de la machine locale vers la machine hôte (`download(String source, String destination)`) qui prend comme paramètres le chemin du fichier à charger sur la machine hôte distante, et le chemin local du répertoire dans lequel chercher le fichier à charger. Il existe plusieurs implémentations de l'interface `Transfer` dans la machine virtuelle DeployWare, par exemple pour le protocole FTP.

Enfin nous avons identifié trois opérations fournies par l'interface `User` servant à fournir les informations de connexion distante sur une machine pour un utilisateur donné. Cette interface contient donc trois opérations : `getLogin()` pour obtenir le nom de connexion,

`getPassword()` pour obtenir le mot de passe si c'est ce mode d'authentification qui vaut pour la machine, ou `getPrivateKey()` pour obtenir la clé de connexion si c'est ce mode d'authentification qui vaut pour la machine.

Enfin l'interface `Port` permet d'identifier le port ouvert pour la connexion à distance, ou le transfert de fichiers. Cette interface contient donc une unique opération `getPort()` qui renvoie la valeur numérique du port matérialisé par le composant en question.

L'expert réseau est en charge d'implémenter chaque opération de chaque interface en fonction du mécanisme réseau qu'il réalise (accès, transfert de fichier, etc.). Les implémentations qu'il réalise pour chaque composant sont ensuite réutilisables *sur l'étagère*. L'administrateur système n'a ensuite qu'à décrire les machines du domaine de déploiement concerné en terme d'information de connexion (nom d'utilisateur, mot de passe), les ports sur lesquels sont ouverts les canaux de communication, les protocoles d'accès à distance et de transfert de fichier, et le Shell. Les informations fournies par l'administrateur système restent donc purement descriptives, il choisit uniquement des implémentations existantes sur l'étagère pour chacun des composants d'accès à la machine hôte.

À titre d'exemple, sur le listing 5.1 figure l'implémentation de l'interface `Shell` pour le Bourne Shell (SH). Cette classe implémente l'interface `Shell` de la figure 5.2, en formattant les commandes conformément au langage SH. L'implémentation des composants Fractal de la machine virtuelle sont écrits à l'aide du framework d'annotations Fraclet [62]. Ce framework permet l'utilisation d'annotations Java pour faciliter le développement de composants Fractal : `@Component` permet de déclarer qu'une classe Java est l'implémentation d'un type de composant donné, `@Requires` permet de définir une référence vers une interface fournie par un autre composant, `@Provides` permet une interface fournie par le composant, et `@Attribute` permet d'identifier un attribut de la classe Java comme étant un attribut de composant. L'implémentation des interfaces `Protocol` et `Transfer` se fait de manière similaire. L'implémentation consiste à définir respectivement le mode d'envoi des commandes formattées par le composant Shell (*e.g.* via SSH, Telnet), et le mode d'envoi de fichiers binaires (*e.g.* via SCP, FTP). Par exemple, une implémentation existante du protocole SSH repose sur l'API JAVA JSch. Une implémentation qui existe pour l'interface Transfer repose sur l'API Commons Net FTP du consortium Apache.

```
@Component public class SH implements Shell
{
  // L'interface 'protocol' requise par ce composant.
  @Requires protected Protocol protocol;
  public void execute (final String command) {
    // Envoyer la commande, donc son identifiant, via le protocole.
    protocol.send(command);
  }
  public void setVariable (final String name, final String value) {
    // Formatter la commande à la syntaxe shell Unix.
    protocol.send("export " + name + "=" + value);
  }
  public void unsetVariable (final String name) {
    protocol.send("unset " + name);  }  }
```

Listing 5.1 – Implémentation SH de l'interface Shell

5.1.2 Composants de logiciels

Les composants de logiciel contiennent, sous une forme exécutable, la définition des protocoles de déploiement d'un logiciel donné. Une représentation schématique d'un tel logiciel est donnée sur la figure 5.4.

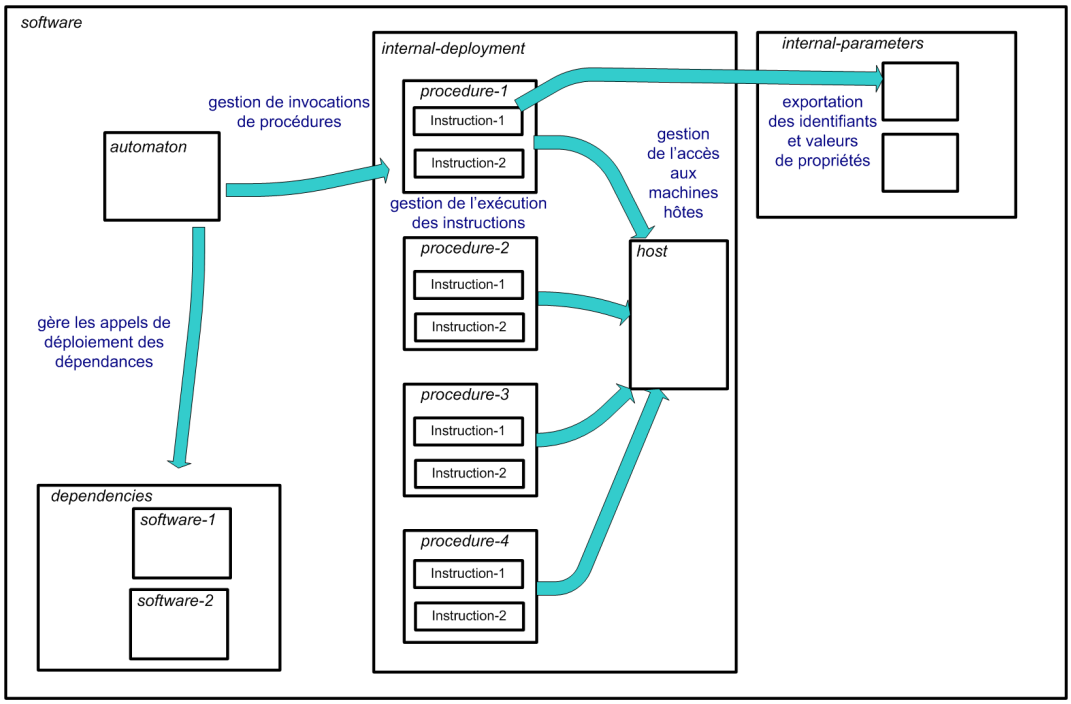

FIG. 5.4 – Composant représentant un type de logiciel dans la machine virtuelle DeployWare

L'architecture d'un composant de logiciel est composé de quatre grandes parties dont nous allons détailler le contenu dans les paragraphes suivants. Tout d'abord une partie de l'architecture est dédiée à la définition des propriétés des logiciels (*internal-parameters*). Une autre partie de l'architecture est quant à elle dédiée à la définition de l'automate de déploiement du logiciel (*automaton*). Ensuite, un composant de logiciel comporte une partie pour la définition des différents logiciels dont dépend le logiciel décrit (*dependencies*). Enfin, les procédures de déploiement sont décrites dans la quatrième partie de l'architecture (*internal-deployment*).

Dans le suite, nous décrirons également les différentes interactions entre ces quatre parties, et l'interaction entre les composants de logiciels et le composant d'accès à la machine hôte sur laquelle le déploiement doit être effectué.

Paramètres de logiciels Dans un composant de logiciel, les paramètres du logiciel concernés sont regroupés dans un composite spécial appelé `internal-parameters`. Le concept de *paramètre* est une projection immédiate du concept de *Property* dans le méta-modèle DeployWare. L'architecture de ce composite est représentée sur la figure 5.5.

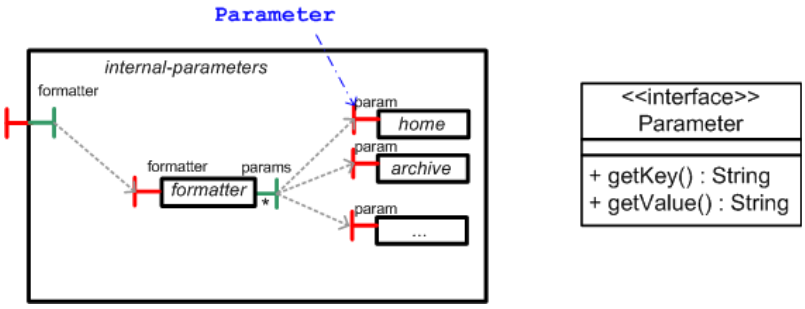

FIG. 5.5 – Composant représentant les paramètres d'un type de logiciel

Ce composant possède d'abord un certain nombre de *composants de propriété*. Un composant de propriété est un composant primitif possédant une seule interface, qui doit offrir des opérations permettant : 1) de connaître l'identifiant de la propriété (`getKey()`) 2) de connaître la valeur actuelle de la propriété (`getValue()`). Le composant `internal-parameters` contient également un composant générique appelé `formatter`. Le rôle de ce composant est de formater une expression contenant des identifiants de paramètres, c.-à-d. remplacer les symboles correspondant à des variables par les valeurs actuelles des paramètres. Par exemple, si un logiciel possède un paramètre `home`, et que l'une de ses instructions représente l'instruction BASH `export HOME=#{home}`[2], alors formater cette commande signifie remplacer `#{home}` par la valeur du paramètre correspondant à cet identifiant. Un tel mécanisme permet de factoriser l'affectation d'une valeur de paramètre : la valeur du paramètre est positionnée une fois, et dans les procédures ou instructions du logiciel, on emploie l'identifiant du paramètre qui est ensuite formatté, non directement sa valeur. L'interface du formatter est également fournie au niveau du composite `internal-parameters` pour être utilisable par les autres sous-composants du composant de logiciel. Les appels sur cette interface sont ensuite délégués directement au formatter.

Pour la gestion de propriétés importées d'autres logiciel (la relation *imported_props* du méta-modèle DeployWare), il est possible d'utiliser le partage de composants Fractal, pour partager les composants de paramètres du logiciel concerné, dans le logiciel courant. Ainsi ce composant peut également être lié au formatter du logiciel courant.

Gestion des dépendances Les dépendances d'un composant de logiciel sont regroupées dans un composite dénommé `dependencies`, dont la structure est représentée sur la figure 5.6.

Les sous-composants de ce composite sont des références vers les différents composants de logiciel correspondant aux dépendances du logiciel décrit, qu'elles soient techniques ou métiers. Un composant particulier, l'`internal-dispatcher`, est en charge de l'orchestration d'appels de procédures sur les dépendances, en accord avec l'automate défini pour le logiciel défini. Ce composant possède une interface cliente multiple sur chacune des inter-

[2]La notation `#{param}` représente une référence vers la valeur du paramètre d'identifiant `param`.

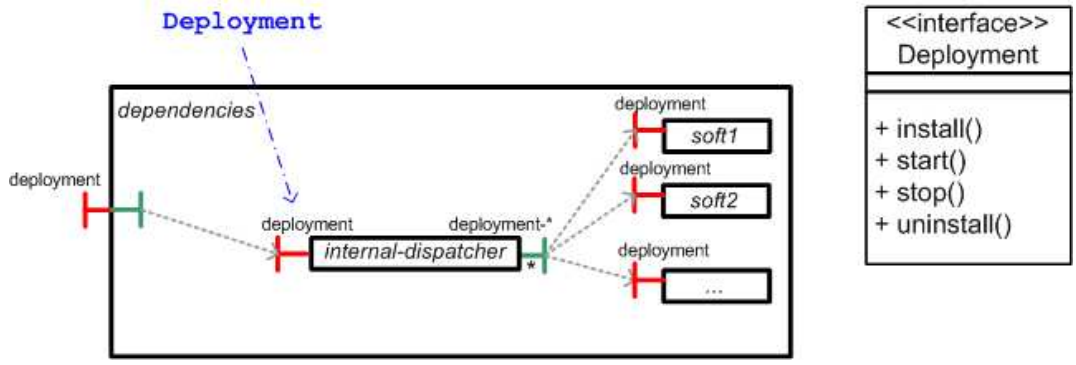

FIG. 5.6 – Composant de gestion des dépendances d'un type de logiciel

faces `deployment` des logiciels présents dans les dépendances. L'interface `deployment` est l'interface qui contient toutes les opérations comprises dans le cycle de vie d'un logiciel, donc l'installation, la configuration, le démarrage etc.. L'implémentation du composant `internal-dispatcher` renferme la logique d'orchestration des dépendances.

Procédures et instructions de déploiement Les procédures de déploiement sont regroupées dans un composant nommé `internal-deployment`. Ce composant est représenté sur la figure 5.7.

Ce composant exporte une interface de type `deployment` afin de piloter le déploiement du logiciel, donc d'appeler les procédures adéquates en fonction de l'opération de l'interface qui est appelée.

À l'intérieur du composant `internal-deployment`, on trouve un composant nommé `internal-status` qui est un automate interne. Cet automate, dont l'interface `deployment` est liée à celle du composite englobant, pilote l'exécution des différentes procédures présentes dans le composite. Cela signifie que l'implémentation de cet automate renferme l'ordre d'exécution des procédures définies pour ce logiciel, comme nous le verrons dans le paragraphe suivant. Ces procédures sont représentées elles-mêmes sous forme de composants composites, exposant une interface de type `java.lang.Runnable`, qui fournit donc une opération `run()` permettant de lancer l'exécution de la procédure. Ces procédures contiennent des composants d'instruction qui eux également possèdent une interface de type `java.lang.Runnable`. Les composants d'instruction sont librement implémentés par l'expert réseau ou par l'administrateur système, en cas de besoin d'étendre l'ensemble des instructions incluses dans la bibliothèque de base de DeployWare. Selon les besoins et les choix de conception qui sont pris, ces composants peuvent être primitifs ou composites.

Un grand nombre de liaisons est nécessaire pour le bon fonctionnement des procédures et de ses instructions.

Par exemple pour que les instructions puissent être exécutées, il faut qu'elles soient reliées à plusieurs autres composants. Il faut qu'elles soient reliées aux interfaces d'accès aux machines en premier lieu, afin d'obtenir les informations de connexion à la machine hôte,

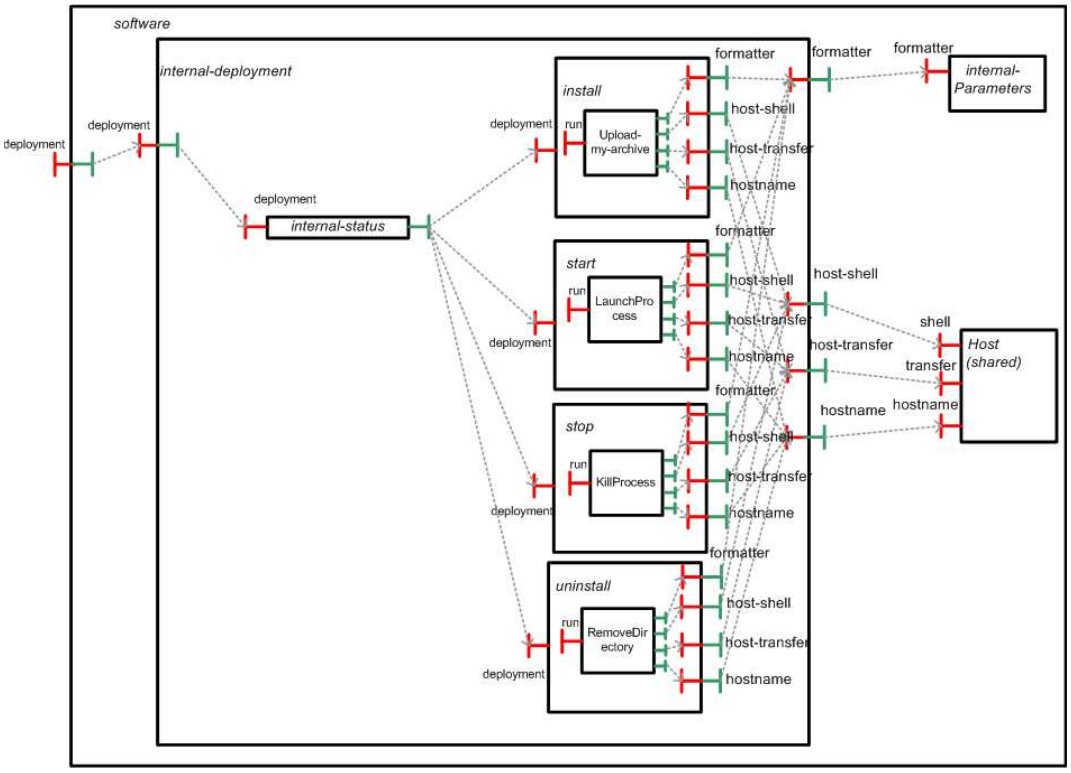

FIG. 5.7 – Composant de procédures d'un type de logiciel

d'envoyer des commandes en utilisant le bon protocole et le bon langage, ou d'envoyer des fichiers. Pour cela, le composant d'accès à la machine hôte concerné doit tout d'abord être partagé dans le composant de logiciel. Ensuite chacune de ses interfaces doit être importée dans `internal-deployment`, puis importée également dans chacune des procédures, afin de pouvoir finalement lier les instructions à ces interfaces d'accès, et rendre l'envoi des commandes et/ou fichiers possible. L'interface `formatter` du composant `internal-parameters` doit également être importée jusqu'au niveau de chaque instruction. Une fois liée à cette interface, les instructions peuvent formatter leurs commandes afin d'importer les valeurs des propriétés du logiciel.

Le listing 5.2 montre un exemple d'implémentation possible pour un composant d'instruction, en l'occurrence `SetVariable`, l'instruction qui permet de positionner une variable d'environnement.

```
@Component
public class SetVariable extends AbstractRunner
{
  /**
   * Référence vers l'interface shell du composant d'accès
   */
  @Requires
  protected Shell shell;
```

```
 /**
   * Référence vers l'interface formatter du composant
   * formatter dans internal-parameters
   */
  @Requires
  protected Formatter formatter;


  /**
   * Le nom de la variable
   */
  @Attribute
  protected String name;

  /**
   * La valeur à affecter à la variable
   */
  @Attribute
  protected String value;

  // ----------------------------------------------------------------------
  // Implementation of the Runnable interface
  // ----------------------------------------------------------------------

  public void run () {
    shell.setVariable(formatter.format(name), formatter.format(value));
  }
}
```

Listing 5.2 – Implémentation du composant d'instruction `SetVariable`

On remarque que cette instruction récupère son interface cliente `Shell`, qui doit être liée à l'interface fournie équivalent sur un composant Shell, récupère également l'interface `formatter` et invoque l'opération `setVariable()` sur l'interface `Shell` en ayant pris soin de formatter les arguments pour résoudre toute référence éventuelle vers un paramètre du logiciel.

Automate L'automate de déploiement d'un type de logiciel est représenté sous la forme d'un composant primitif. Ce composant primitif expose une interface serveur de type *deployment*, qui regroupe toutes les opérations du cycle de vie du déploiement. Chacune de ces opérations s'inscrit dans un enchaînement global. L'exemple classique d'un tel enchaînement est la séquence `install`,`start`,`stop`,`uninstall`. Le code de ce composant permet notamment de gérer le nombre d'occurrences d'appel des différentes procédures. Pour le démarrage par exemple, si `start` est appelée, le compteur est incrémenté, et il est décrémenté en cas d'appel à la procédure inverse de `start`, c.-à-d. `stop`. De cette manière, il est possible d'empêcher les procédures en question d'être exécutées mal à propos (*e.g.* appeler `start` sur un logiciel déjà démarré). Grâce aux compteurs, il est également possible de déterminer à quel moment un logiciel n'est plus utilisé par aucun autre, auquel cas on peut l'arrêter voire le désinstaller. Par exemple, si la procédure de démarrage d'un logiciel SW_0 est appelé N fois, cela signifie que N-1 logiciel $SW_1 ... SW_{N-1}$ au minimum[3] dépendent de ce logiciel, et qu'au moment d'appeler la procédure de démarrage sur ces logiciels, un appel récursif de la procédure

[3] Au minimum, car l'un des appels peut avoir été manuel.

de démarrage de SW_0 a été effectué à chaque fois. Au moment d'arrêter ces logiciels, de la même manière, les procédures d'arrêt des logiciels $SW_1 ... SW_{N-1}$ appelent récursivement la procédure d'arrêt de SW_0. À ce moment, si le compteur de SW_0 tombe à zéro, cela signifie qu SW_0 n'a jamais été appelé manuellement, mais seulement par les logiciels qui dépendent de lui. SW_0 est donc inutile et peut à son tour être automatiquement arrêté. Ce raisonnement est également valable avec l'installation et la désinstallation.

Le code d'un composant d'automate définit principalement les choses suivantes. La procédure d'installation d'un logiciel n'est exécutée que lorsque le logiciel est dans l'état *désinstallé*. La procédure de démarrage d'un logiciel n'est exécutée que lorsque le logiciel est dans l'état *installé* ou *désinstallé*. Dans le deuxième cas, on lance d'abord la procédure d'installation, puis la procédure de démarrage. la procédure d'arrêt d'un logiciel n'est exécutée que lorsque le logiciel est dans l'état `démarré`. On ne lance la procédure de désinstallation que lorsque le logiciel est dans l'état *installé* ou *démarré*. Dans le deuxième cas on lance d'abord la procédure d'arrêt, puis la procédure de désinstallation.

Cet ensemble de clauses, auxquelles on ajoute la tenue des différents compteurs, constitue la définition d'un automate d'appels des procédures suffisant pour un type de logiciel. Il est possible d'implémenter un automate plus complexe selon les besoins de l'expert logiciel.

5.1.3 DeployWare Explorer

Un explorateur graphique a été réalisé pour administrer manuellement les systèmes déployés avec DeployWare. Une capture d'écran de cet explorateur est représentée sur la figure 5.8. Sur cette figure, les caractères *U* représente l'état actuel du logiciel décrit : en l'occurrence *Uninstalled*. Les autres états sont représentés par la *I* pour *Installed*, un petit pictogramme représentant un feu de signalisation vert pour *Started*. Ensuite les flèches étiquetées *host* représentent le lien entre un logiciel et la machine qui l'héberge, alors que les flèches étiquetées *depend* représentent les liens de dépendances entre logiciels.

Cet explorateur permet de visualiser tous les logiciels du système à déployer. Il est également possible de visualiser (sur le panneau de droite) le graphe de dépendances ainsi que la machine d'hébergement de chaque logiciel. Il est possible en utilisant cet explorateur d'invoquer dynamiquement les procédures définies pour les logiciels du système et de visualiser les valeurs définies pour les propriétés des logiciels.

5.1.4 Correspondance avec le méta-modèle DeployWare

Dans le cadre d'une ingénierie dirigée par les modèles, il est essentiel de définir une correspondance entre les concepts du PIM, dans notre cas le méta-modèle DeployWare, et le PSM qui dans notre cas est le modèle de description de la machine virtuelle DeployWare.

À ce jour, la transformation d'un modèle DeployWare vers une instance de la machine virtuelle DeployWare reste manuelle. Cela signifie que l'expert logiciel (*resp.* l'administrateur système) doit, d'abord modéliser sa personnalité (*resp.* son système logiciel), effectuer les vérifications adéquates, puis traduire son modèle en langage DeployWare. Notre volonté a été de faire en sorte que ces deux formes d'expression du déploiement de systèmes distribués restent

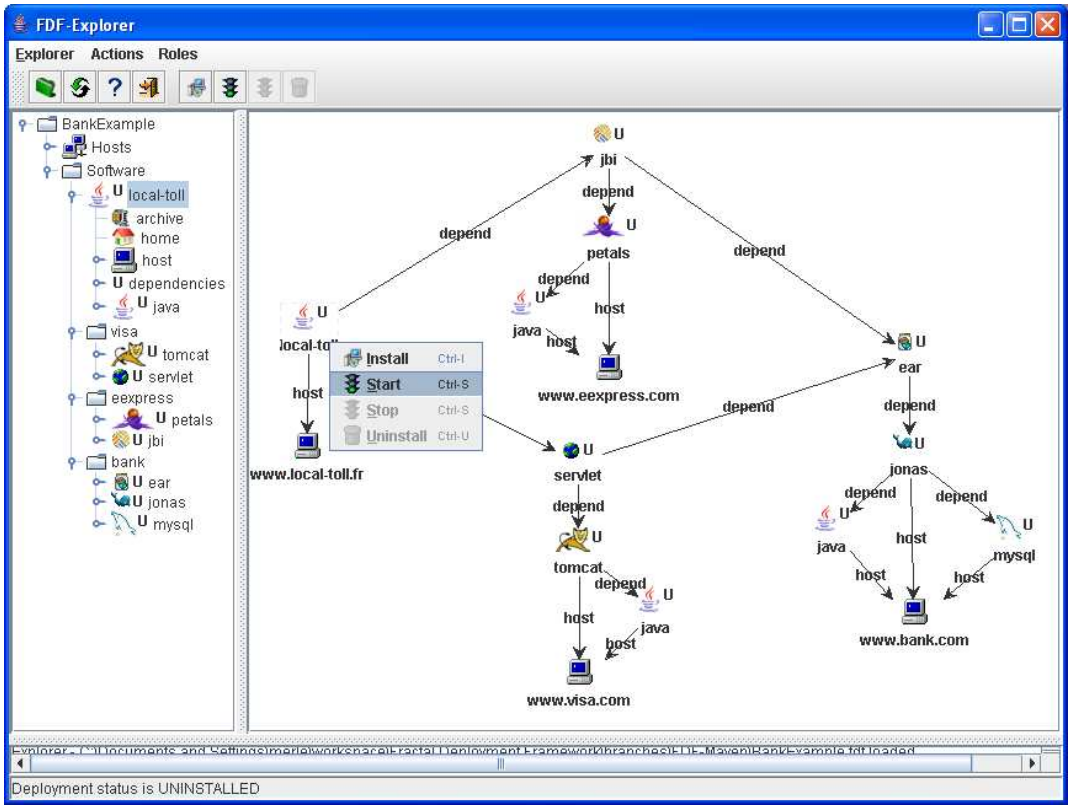

FIG. 5.8 – Capture d'écran de la console graphique de DeployWare

très proches, afin de faciliter au maximum la transformation automatique d'une forme vers l'autre. C'est là une des raisons de l'utilisation de motifs architecturaux Fractal, et donc de la définition du langage DeployWare pour la machine virtuelle décrits dans la section 5.2.

Ainsi le concept de personnalité DeployWare correspond à un paquetage de définitions Fractal. Le concept de machine hôte dans le méta-modèle correspond à la définition, par l'administrateur système, d'un composant d'accès à la machine hôte correspondante dans la machine virtuelle. Enfin les concepts de type de logiciel, dépendances, propriétés (qu'elles soient initiales ou importées), procédures et instructions, et instances de logiciels, correspondent respectivement aux notions de composants de logiciels, de composants de dépendances, de composants de paramètres (partagés lorsqu'il s'agit de propriétés importées), de composants de procédure, et de composants d'instruction.

Ainsi le travail de projection du PIM vers le PSM est très simple à réaliser. De plus, le langage de description de machines virtuelles DeployWare améliore nettement la concision des descriptions, et rapproche un peu plus encore les concepts du méta-modèle des concepts de la machine virtuelle.

5.2 Le langage DeployWare basé sur Fractal ADL

De récentes améliorations du modèle Fractal nous permettent de faciliter grandement l'expression d'architectures de machines de déploiement DeployWare, telles que décrites dans la section précédente. Nous détaillons ces extensions du modèle dans la sous-section 5.2.1, avant de démontrer l'apport de ces extensions dans la construction d'architecture de déploiement DeployWare.

5.2.1 Motifs architecturaux en Fractal ADL

Les travaux de R. Rouvoy et al. [61] sur les motifs architecturaux dans les architectures à base de composants Fractal offrent des perspectives nouvelles pour le modèle.

En effet, un nouvel élément est ajouté à la syntaxe de Fractal ADL, l'élément `apply`. Cet élément permet d'exprimer la répétition d'un motif d'architecture dans différents contextes.

Cet équivalent de la boucle `for` des langages de programmation permet de reproduire un schéma d'architecture en faisant varier un certain nombre de paramètres. La variable de la boucle peut représenter n'importe quel élément du modèle Fractal : un composant, une interface, ou même une chaine de caractères représentant le nom de l'interface d'un composant. Pour exprimer l'ensemble des valeurs que prendra la variable, les auteurs proposent d'utiliser une requête FScript [25].

Ainsi, il est possible en utilisant la boucle `apply` d'exprimer des motifs d'architecture génériques réutilisables parmi les applications. Nous allons expliciter un certain nombre de ces motifs génériques. Tout d'abord, le motif d'*exportation automatique des interfaces serveur d'un composant* (`AutoExport`) sous forme de définition Fractal ADL consiste dans un premier temps, pour un composant (`component`) d'une définition (`definition`) donnée, à insérer le composant dans le composite qui hérite de ce motif. Dans un second temps, pour chaque interface fournie par le composant inséré, il s'agit de déclarer la même interface au niveau du composite englobant, et ensuite d'établir une liaison entre l'interface de component et l'interface identique au niveau de la définition. L'illustration d'un tel mécanisme est représentée sur la figure 5.9.

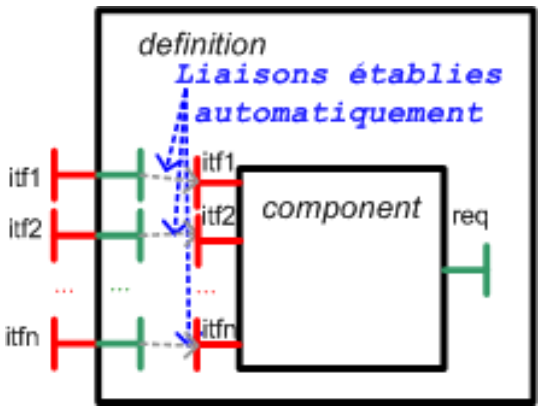

FIG. 5.9 – Composite Fractal représentant le motif d'exportation automatique

À l'inverse, le motif `AutoImport` (qui prend deux paramètres : le nom de l'interface à créer, et la signature de l'interface) a pour but de créer une interface cliente dans un composite, et de connecter la facette interne serveur de cette interface à tous les sous-composants possédant une interface cliente compatible. Ce motif est graphiquement illustré sur la figure 5.10 appliqué à l'interface `logger`.

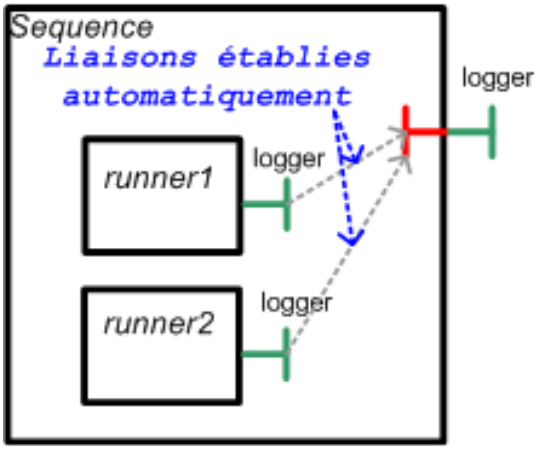

FIG. 5.10 – Composite Fractal représentant le motif d'importation automatique

Un motif générique consiste également à partager automatiquement un composant dans un composite avec tous les sous-composants compatibles (la compatibilité dans ce cas correspond à l'existence dans l'arborescence du composite d'un composant portant le nom passé en paramètre (`name`). Ce motif *AutoSharing* est graphiquement représenté sur la figure 5.11.

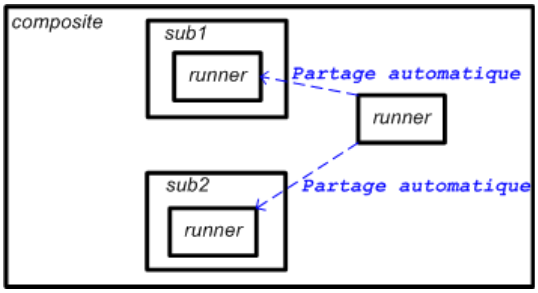

FIG. 5.11 – Composite Fractal représentant le motif de partage automatique

Enfin, le motif `AutoBinding` consiste à lier une interface d'un composant d'un type donné à toutes les interfaces compatibles, exceptées les propres interfaces serveurs du composant. Le composant, le nom de l'interface et le type de cette interface sont donnés en paramètre de ce motif dont une représentation schématique est donnée sur la figure 5.12.

Ces quatre motifs génériques d'architecture rendent l'utilisation de Fractal ADL encore plus aisée et attrayante. Dans la section suivante, nous démontrons qu'utiliser les quatre motifs d'architecture que nous venons d'exposer simplifiement drastiquement l'expression de composants de déploiement de la machine virtuelle DeployWare.

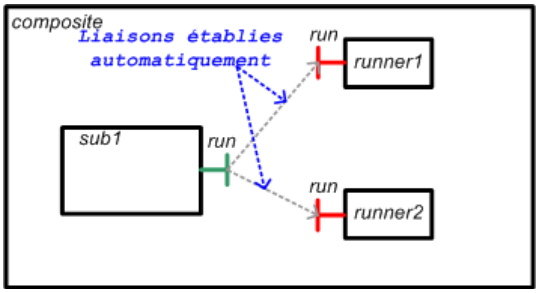

FIG. 5.12 – Composite Fractal représentant le motif de liaison automatique

5.2.2 Utilisation des motifs architecturaux pour la construction de la machine virtuelle DeployWare

Comme nous l'avons vu dans la présentation des composants de la section précédente, les architectures de composants de logiciel DeployWare comportent un grand nombre d'instances de composants, mais également un grand nombre de liaisons entre composants, et d'importations ou d'exportations d'interfaces de composants. C'est la raison pour laquelle nous choisissons d'utiliser les motifs d'architecture définis dans la section 5.2 pour simplifier l'expression de composants d'accès aux machines hôtes ou de logiciel.

Décrire un composant de logiciel DeployWare est à la fois très verbeux, mais également aussi très répétitif. En fait, chaque liaison des composants que l'on a exposés dans la section précédente peut être inférée. Les liaisons entre composants ne servent jamais à exprimer une particularité du logiciel. Ces liaisons sont inhérentes aux choix de conception de la plate-forme d'exécution du déploiement, mais pas à la conception du logiciel à déployer. La même remarque est valable pour les exportations ou importations d'interfaces d'un composant à un autre, et également les partages de composants.

Nous souhaitons donc utiliser les motifs d'architecture Fractal ainsi que le mécanisme d'héritage entre composants, de manière à simplifier au maximum la définition de logiciel. Cela doit nous permettre de factoriser un maximum de constructions de la machine virtuelle, et laisser à l'expert logiciel le soin de décrire uniquement ce qui caractérise son logiciel, c'est-à-dire les paramètres, dépendances, procédures et instructions de son type de logiciel.

Tout d'abord pour le composant d'accès à la machine hôte (page 127), on peut appliquer le motif `AutoExport` aux interfaces `shell`, `transfer`, et `hostname`, afin de déléguer ces interfaces sur le composite englobant du composant d'accès. Nous définissons ensuite un type de composant abstrait, dans lequel nous spécifions une implémentation par défaut pour les composants `shell`, `protocol`, `transfer` et `hostname`, ainsi que les liaisons qui étaient définies sur la figure 5.2 de la page 127 et enfin l'application du motif AutoExport sus-citée. De ce fait, pour définir un host particulier, l'administrateur système n'a plus qu'à préciser quelle implémentation il veut pour les composants `shell`, `protocol`, `transfer`, `user`, ainsi que le nom d'hôte de la machine. Les informations Fractal liées à la machine virtuelle sont contenues dans cette définition abstraite, et aucune liaison n'est à exprimer par l'administra-

teur système. Ensuite, pour la définition de composants de logiciel, il est possible d'utiliser un certain nombre de motifs architecturaux. Le motif `AutoExport` est utilisé pour exporter l'interface `deployment` du composant `internal-deployment` (figure 5.7 de la page 133) vers le composite global du composant de logiciel. Ce même motif est utilisé pour exporter l'interface `deployment` du composant `internal-status` vers son composite englobant `internal-deployment`.

Le motif `AutoImport` est utilisé pour importer les interfaces `host-shell`, `host--transfer` et `hostname` dans `internal-deployment`, puis également pour importer ces interfaces de `internal-deployment` vers chacun des composants de procédure.

Le motif d'`AutoBinding` est utilisé entre autres afin de connecter le composant *formatter* à chacun des composants de propriété présents dans le composite *internal-parameters*, quel que soit le nombre de paramètres présents dans le composant *internal-parameters*. De la même manière qu'avec les composants d'accès aux machines, nous factorisons un ensemble de constructions (*i.e.* les définitions des composants `internal-parameters`, `internal--deployment`, `dependencies` et un composant d'automate de base) dans un composant de logiciel générique dans lequel nous établissons toutes les connexions qui peuvent l'être. À l'aide de motifs, nous déterminons des importations/exportations d'interface, ainsi que des liaisons automatiques qui seront établies au chargement d'un composant de logiciel. L'expert logiciel qui souhaite définir un nouveau type de composant n'a qu'à choisir, paramétrer et déclarer des instances de composants de propriétés, de procédures, de dépendances et d'instructions, qui seront automatiquement intégrés dans le composant de logiciel à l'aide des motifs et de l'héritage.
Enfin, pour affecter une machine hôte à un logiciel, il faut partager le composant de machine hôte à la racine du composant de logiciel correspondant. L'application des importations et liaisons automatiques assure l'interaction entre composant d'accès à la machine hôte partagé et le composant de logiciel.

Par exemple, dans le type de logiciel *Installable* déjà évoqué dans cette thèse dans la section 4.1.2, on définit deux propriétés obligatoires pour tout logiciel de ce type qui sont `archive` et `home`. Dans la définition du type abstrait, on déclare alors deux composants `home` et `archive` à la racine du logiciel, et on utilise le motif *AutoSharing* pour préciser que la définition de ces composants doit être automatiquement incluse dans le `internal-parameters`. On définit ainsi des propriétés globales pour le logiciel.

5.2.3 Le langage DeployWare

L'utilisation de motifs architecturaux pour définir un composant de logiciel réalise automatiquement la totalité des liaisons, exportation/importation d'interfaces. La définition générée à partir du modèle de l'expert logiciel ne consiste qu'à définir quelles sont les propriétés du logiciel, quelles sont ses dépendances, ses procédures et leurs instructions respectives. Finalement, même au niveau de la machine virtuelle, il n'y a pas de nécessité de connaître le modèle de composants Fractal. Toutes les constructions propres à Fractal sont inférées en fonction du

```
<definition name="MyDefinition" extends
    ="MySuperDefinition">
  <component name="a" definition="
      MyAComponent(a_value)"/>
  <component name="b">
    <component name="c" definition="
        MyCComponent"/>
    <component name="a" definition="./a
        "/>
  </component>
</definition>
```

Listing 5.3 – Définition en Fractal ADL

```
MyDefinition = MySuperDefinition
{
  a = MyAComponent(a_value) ;
  b {
    c = MyCComponent ;
    a = ./ ;
  }
}
```

Listing 5.4 – Définition en langage DeployWare

plan de composant défini. Nous avons donc créé un langage propre à la construction de type de logiciel et d'instances de ces logiciels. Ce langage ne fait en fait que cacher les notions Fractal complètement inutiles à la création de logiciel pour la machine virtuelle. La syntaxe XML de Fractal ADL est abandonnée au profit d'une syntaxe déclarative moins verbeuse, en enlevant même le mot clé `component`. Les principaux éléments de ce langage sont les suivants :

- Le mot clé `definition` disparait. Le nom de la définition est le premier identifiant du fichier.
- L'héritage de définitions reste possible, mais le mot clé `extends` disparait au profit du symbole =.
- La description du contenu d'un composant se fait entre accolades.
- Au sein d'accolades, la notation «a = B(c,d)» signifie qu'une instance de composant, nommée a, est de type B avec c et d en paramètre.
- Au sein d'accolades, la notation «a = ./b/» signifique que dans ce contexte le composant A est le même (utilisation du mécanisme de partage) que celui à l'intérieur du composite nommé b à la racine de la définition (dénotée par le point).

Sur les listings 5.3 et 5.4 figure, à gauche, un exemple de définition Fractal ADL, et à droite, son équivalent en langage DeployWare.

En utilisant ce langage, on crée des logiciels qui vont *contenir* des propriétés, des procédures et des dépendances. Les conventions de nommage des composants servent de repère pour l'écriture de fichiers de définition de type ou d'instance de logiciel. Sur le listing 5.5 est représenté un exemple de définition d'un type de logiciel par l'expert logiciel, et sur le listing 5.6 sont représentées une machine hôte et la configuration du déploiement d'une instance du type de logiciel décrit dans le listing 5.5 [4]. Le parser, très simple, chargé de lire les fichiers conformes à la syntaxe DeployWare, et de générer des descripteurs Fractal ADL classiques est réalisé à l'aide du framework JavaCC[5].

```
MyPersonality.MySoftware
= software.Installable(mysoftware,MYSOFTWARE)
{
  # archive is required.
  archive = MYSOFTWARE.ARCHIVE(UNDEFINED);
```

[4] Par convention, les majuscules sont utilisées pour les types du langage, alors que les minuscules sont utilisées pour les identifiants

[5] `https://javacc.dev.java.net/`

```
  # home is required.
  home = MYSOFTWARE.HOME(UNDEFINED);

  # Implementation of the JRE software.

  my-software {
    internal-parameters {
       server-name = MyPersonality.MYSOFTWARENAME(Default_Name);
    }
    internal-deployment {

      # The 'configure' procedure
      configure {
        set-home-variable = SHELL.SetVariable(...);
      }
      # The 'start' procedure.
      start {
  ...
      }
      # The 'stop' procedure.
      stop {
  ...
      }
      # The 'unconfigure' procedure
      unconfigure {
        ...
      }
}}}
```

Listing 5.5 – Exemple de fichier de définition de type de logiciel dans la VM DeployWare

```
my-system{
 host1 = INTERNET.HOST {
      hostname = INTERNET.HOSTNAME(hostname);
      user     = INTERNET.USER(username,password,privateKey);
      transfer = TRANSFER.JSCP;
      protocol = PROTOCOL.OpenSSH;
      shell    = SHELL.SH;
    }

 soft = MyPersonality.MySoftware {
          archive = MYSOFTWARE.ARCHIVE(Archive_File_URI);
          home    = MYSOFTWARE.HOME(Java_Home_Directory_URI);
    host = /host1 ;
          internal-parameters {
            server-name = MyPersonality.MYSOFTWARENAME(mysoft_instance) ;
          }
        }
}
```

Listing 5.6 – Exemple de fichier de définition d'instance de logiciel dans la VM DeployWare

5.3 Composants spécifiques au déploiement en environnements ouverts

Dans cette section, nous injectons des mécanismes d'autonomie dans la machine virtuelle DeployWare, afin de rendre possible le déploiement des systèmes logiciels dans des environnements ouverts distribués. La particularité des environnements ouverts distribués est que le nombre de machines hôtes est inconnu et imprévisible au fil du temps. Lorsqu'il s'agit de réseaux ubiquitaires faisant intervenir des terminaux mobiles, le déploiement de systèmes logiciels est continu à travers le temps, et doit s'effectuer sur les machines qui entrent ou sortent du réseau. Un administrateur système humain, en charge de ce déploiement, ne peut décemment pas surveiller les allées et venues de terminaux mobiles afin de déployer les logiciels adéquats sur les terminaux mobiles entrant dans le réseau. Il faut alors envisager un déploiement *auto-géré* qui s'adapte aux fluctuations de l'environnement sous-jacent.

Ainsi, notre objectif est d'appliquer une approche d'informatique auto-gérée au déploiement de systèmes logiciels. Pour cela, nous ajoutons dans la machine virtuelle DeployWare des *boucles de contrôle* comme nous les avons décrites dans la section 2.5.

Nous implémentons l'auto-gestion dans DeployWare en plusieurs étapes. Tout d'abord nous identifions les deux environnements de la boucle de contrôle comme étant le domaine de déploiement (pour l'*environnement d'exécution*) et l'architecture de la machine virtuelle DeployWare (pour la *connaissance*). Ensuite, nous définissons un type de logiciel DeployWare particulier capable de prendre en charge l'exécution de boucles de contrôle pour adapter la machine virtuelle aux fluctuations de l'environnement. Des boucles de contrôle exprimées à l'aide du paradigme *Evénement-Condition-Action* [22] peuvent être injectées dans ce logiciel, qui gère un ensemble de composants de logiciels DeployWare ou de composants d'accès aux machines hôtes. Il est capable d'écouter des événements, de tester si ces événements déclenchent l'une des boucles de contrôle, et le cas échéant, lance l'exécution de la reconfiguration sur les composants de la machine virtuelle, ce qui implique une reconfiguration du système déployé par la machine virtuelle.

5.3.1 Application de l'informatique auto-gérée pour le déploiement

Dans notre cas, l'environnement d'exécution est le domaine de déploiement, donc l'ensemble des machines hôtes, et les différents logiciels qui y sont déployés. Le partie connaissance est quant à elle représentée par l'architecture de composants de la machine virtuelle DeployWare. En effet, cette architecture représente une abstraction adéquate, représentant les éléments essentiels du système logiciel, sous forme d'assemblage de composants. Le lien causal de la connaissance vers l'environnement d'exécution est d'ailleurs déjà existante dans l'un des sens, car la reconfiguration de l'architecture de composants DeployWare, si elle est couplée à des appels de procédure, entraine la modification souhaitée sur l'environnement d'exécution. L'autre sens du lien causal sera à assurer par les politiques définies par l'administrateur système : c'est lui doit implémenter les mécanismes de surveillance de l'environnement d'exécution, notamment en terme d'arrivées ou de départ des machines, et décrire les mises à jour de la machine virtuelle en conséquence (ajout d'un composant d'accès à la machine hôte détectée). Cette architecture, par les capacités d'introspection et de reconfiguration dynamique du

modèle de composants Fractal, représente un support pour l'application de la planification des reconfigurations de la boucle de contrôle.

L'administrateur système doit également disposer d'un langage pour exprimer ses politiques d'auto-gestion sous forme de boucles de contrôle. Les éléments de ces politiques comportent trois éléments. Dans la mesure où s'adapter signifie s'ajuster à une fonction ou à une circonstance particulière, cette circonstance propice à l'adaptation doit figurer dans la politique. Il s'agit du *stimulus*. Par ailleurs, l'adaptation en réponse à un stimulus peut dépendre d'un ensemble de propriétés subsidiaires nécessaires au déclenchement des actions d'adaptation. C'est le *contexte* du stimulus. Enfin, quand un stimulus intervient dans un certain contexte propice à l'adaptation, l'*exécution* de la politique doit être déclenchée. Le paradigme de règle *Événement-Condition-Action*, largement étudié dans la communauté des bases de données actives [22], peut nous permettre d'exprimer ces différentes politiques. En effet, le stimulus étant modélisé dans la partie événement de la politique, le contexte étant décrit comme un ensemble de conditions, il est ainsi possible de décrire l'exécution de la politique dans la partie action de la règle.

Nous voyons donc, dans la suite, comment les mécanismes d'auto-gestion ont été apportés à la machine virtuelle DeployWare à l'aide d'un type de logiciel donné, l'`Autonomic.MANAGER` qui utilise le langage ECA, des logiciels sondes et un composant d'action pour implémenter les boucles de contrôle.

5.3.2 Logiciel DeployWare d'auto-gestion

L'architecture du composant de logiciel pour l'auto-gestion dans DeployWare est représenté sur la figure 5.13.

L'élément de plus haut niveau est le *domaine d'auto-gestion*. Il est représenté par un composite contenant un composant `Autonomic.MANAGER`, un ou plusieurs composants `Autonomic.SENSOR`, un composant `Autonomic.ACTUATOR` et un ensemble de composants de logiciel ou de machines hôtes contrôlés par les politiques d'autonomie.

Les composants `Autonomic.SENSOR` représentent les *sondes*, c.-à-d. une entité chargée de surveiller l'état de l'environnement d'exécution, donc dans notre cas du domaine de déploiement, et de faire remonter des changements notoires sous forme d'événements. Ces événements étant purement informatifs, nous imposons juste qu'ils soient représentés par des objets quelconques. L'expert réseau est en charge du développement des composants de type `Autonomic.SENSOR`. Il s'agit de composants Fractal possédant une interface client `event` qui correspond à un puits d'événements, c'est-à-dire une interface pour communiquer les événements observés depuis l'extérieur. Le code de ce composant est ensuite librement réalisé afin de surveiller l'environnement afin de notifier le `Autonomic.MANAGER` d'un éventuel changement significatif.

Les composants `Autonomic.ACTUATOR` représentent le moteur d'exécution des reconfigurations. Ce composant fait partie de la bibliothèque DeployWare, et expose une interface *action* offrant de nombreuses opérations de reconfiguration dans le domaine d'auto-gestion dans lequel il est inséré. L'opération `addSoftware` consiste à insérer un nouveau logiciel dans le domaine d'auto-gestion. Cette opération prend en paramètre la description du logiciel dans le

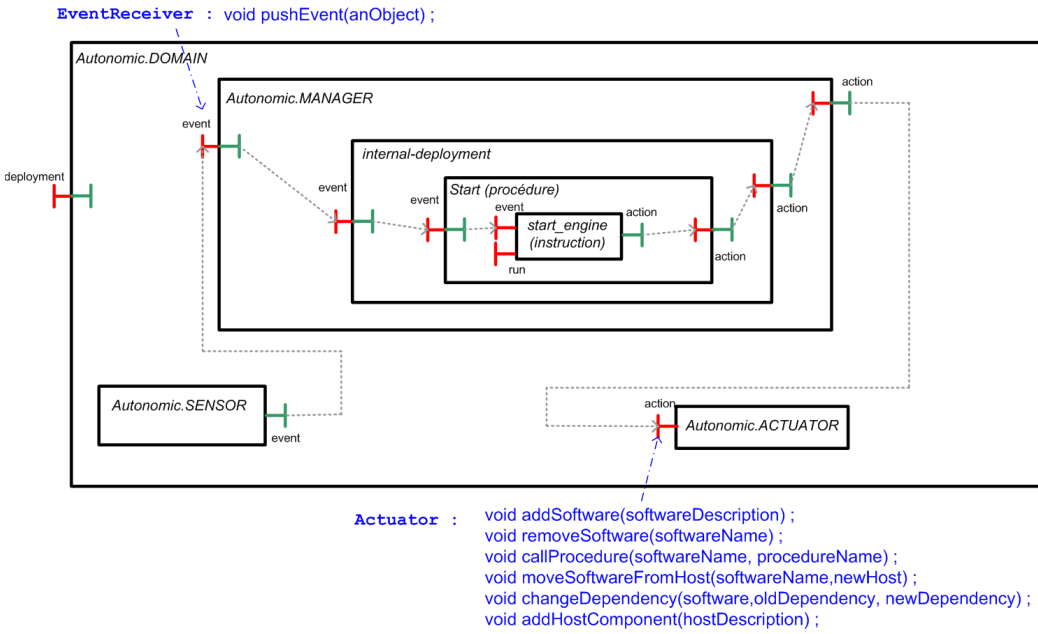

FIG. 5.13 – Architecture du composant de logiciel pour l'auto-gestion

langage DeployWare que nous avons défini dans la section 5.2. La description de ce nouveau logiciel peut faire référence à une machine hôte déjà existante dans la machine virtuelle, ou un autre logiciel. L'opération `removeSoftware` consiste à supprimer l'occurrence d'un composant de logiciel dont le nom (*i.e.* le nom du composant le réifiant) est passé en paramètre. Cette opération ne replie pas le logiciel, elle ne fait que supprimer le composant qui réifie le logiciel. L'opération `callProcedure` consiste à invoquer l'exécution d'une procédure de l'un des logiciels. L'identifiant du logiciel, ainsi que l'identifiant de la procédure à invoquer sont passés en paramètre. L'opération `moveSoftwareFromHost` permet de redéployer un logiciel sur une autre machine. Cette migration de logiciel reste simpliste et ne permet pas de gérer la migration des états. Une telle opération permet par exemple de procéder facilement au redéploiement d'un logiciel provenant d'une machine en panne. L'opération `changeDependency` permet de redéployer un logiciel en modifiant l'une de ses dépendances. Cette opération prend trois paramètres : le logiciel à redéployer, l'ancienne dépendance, et la nouvelle dépendance. Les deux dépendances doivent naturellement posséder le même type de logiciel. Enfin, l'opération `addHostComponent` permet d'ajouter une nouvelle instance de composant d'accès à une machine hôte. De la même manière que pour l'opération d'ajout de composant de logiciel, cette opération prend en paramètre un fichier comprenant la description du composant hôte dans la syntaxe DeployWare.

5.3.3 Implémentation du composant de décision

Le composant de décision est au centre de l'auto-gestion dans DeployWare. Il s'agit du composant devant recevoir d'un côté les événements, et de l'autre exécuter les opérations de reconfiguration en accord avec les politiques d'auto-gestion qui ont été définies pour le domaine administré par ce composant.

Nous avons réalisé deux implémentations de ce composant de décision.

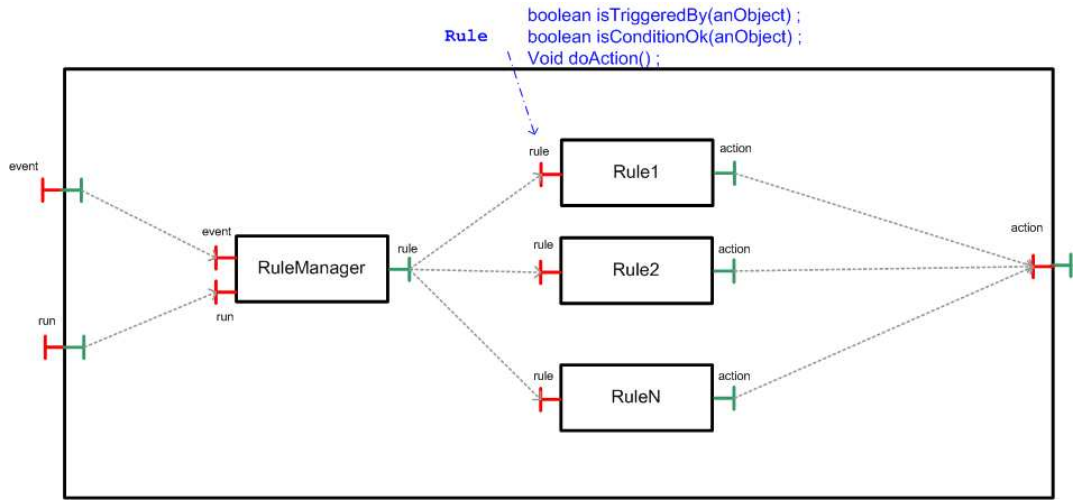

FIG. 5.14 – Moteur de règles à base de composants Fractal

Tout d'abord, nous avons réalisé un prototype de moteur de règles à base d'architecture Fractal. Ce moteur est représenté sur la figure 5.14. Il est composé d'un composant `RuleManager` qui reçoit les évenements, et cherche les règles déclenchables par cet événement. L'API des composants de règles permet de calculer si un événement déclenche cette règle (opération `isTriggeredBy()`. Si c'est le cas on confronte l'événement en question aux conditions qui doivent être remplies (opération `IsConditionOk()`). Dans le code de l'opération `IsConditionOk()`, il est possible d'introspecter la machine virtuelle à l'aide de l'API Fractal, et donc possible d'évaluer des conditions dans l'ensemble de la machine virtuelle et pas uniquement sur l'événement déclencheur. Enfin si les conditions sont remplies, on peut lancer l'exécution des actions, qui seront autant d'appels sur l'interface `action` à laquelle est reliée la règle. Un code de composant de règle est représenté sur le listing 5.7. La règle `NewPDARule` consiste, lorsqu'un événement de type `deployware.events.NewHostEvent` survient, à vérifier que la machine hôte qui vient d'entrer dans le réseau est un PDA, et si c'est le cas, invoque le composant `Actuator` pour instancier un nouveau composant d'accès à la machine hôte de type PDA correspondante.

```
package myrules;
@Component
public class NewPDARule implements Rule {

  @Requires(name="action")
  public Actuator domain ;
```

```
  public MyRule(String n) {
    this.eventName = n;
  }

  public boolean isTriggeredBy(Object o) {
    return o instanceof deployware.events.NewHostDeclaredEvent ;
  }

  public boolean isConditionOk(Object o) {
    deployware.events.NewHostEvent event = (deployware.events.NewHostEvent) o ;
    return event.hosttype.equals("PDA") ;
  }

  public void doAction(Object o) {
    deployware.events.NewHostEvent event = (deployware.events.NewHostEvent) o ;
    // GenericPDAHostFileDescription est une classe utilitaire capable de générer un
        fichier
    // de description DeployWare représentant un hôte de type PDA
    File dw_file = new GenericPDAHostFileDescription(event.hostname) ;
    this.domain.addHostComponent(dw_file) ;
  }
}
```

Listing 5.7 – Code d'un composant Fractal de règle d'auto-gestion

Les avantages de ce moteur de règles résident dans la liberté d'implémentation des différentes règles. Il est en effet possible, grâce à ce canevas, de bénéficier des nombreuses constructions (*e.g.* les boucles) du langage Java pour articuler les différentes actions à effectuer. En outre, il est possible dans ces règles d'utiliser l'API Fractal pour introspecter le domaine d'auto-gestion et implémenter des conditions de déclenchement plus fines. Néanmoins ce moteur de règles nécessite une connaissance du modèle de composants Fractal, ce qui peut être redhibitoire pour l'administrateur système.

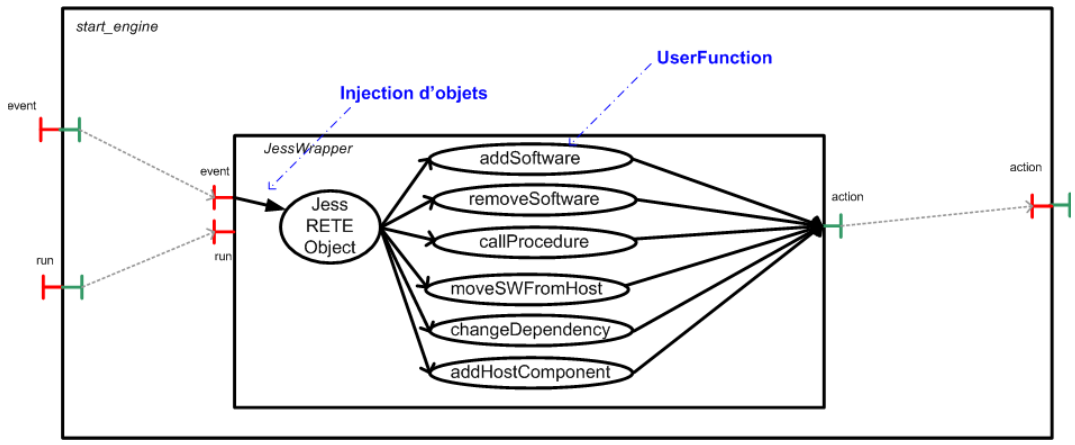

FIG. 5.15 – Moteur de règles basé sur JESS

Le second moteur de règles repose sur JESS, un moteur d'inférence en Java[6]. L'architecture de ce moteur de règles est schématisée sur la figure 5.15. Ce moteur de règles d'infé-

[6]http ://herzberg.ca.sandia.gov/

rence est hautement personnalisable. Il permet notamment l'injection d'objets Java directement dans le moteur, confrontés aux règles présentes dans le moteur. Il permet d'appeler des actions personnalisées créées en utilisant le mécanisme d'*UserFunction*. Ainsi pour définir une nouvelle action dans le langage de Jess, il faut créer une classe qui implémente l'interface `UserFunction`. Cette interface définit des opérations pour obtenir le nom de la commande afin de reconnaître le UserFunction adéquat lors du parsing, et définit une opération qui correspond à l'invocation de l'action à effectuer en cas d'appel à cette nouvelle commande (une commande définie par une UserFunction peut posséder des paramètres). Un *UserFunction* représentant une seule action, nous avons implémenté autant d'*UserFunctions* qu'il y a d'opérations dans l'interface `action` du type de composant `Actuator`. Le listing 5.8 représente la même règle que celle présentée pour le moteur de règles Fractal, mais en utilisant la syntaxe Jess.

```
#La règle qui déclare la correspondance entre l'objet d'événement
#et le type Jess NewPDAEvent
(deftemplate NewPDAEvent (declare (from-class deployware.events.NewPDAEvent)))

(defrule newPDA
"Add new DeployWare host component when it is detected"
#Si l'événement NewPDAEvent apparait (on le stocke dans une variable event)
?event <- (NewPDAEvent {hosttype = "PDA"})
=>
(printout t "#JESS# -- Rule triggered : New PDA appears !!!" crlf)

#On considère ici que le code du UserFunction addHostComponent génère
#le fichier description correspondant avant d'effectuer l'appel sur
#l'interface action
#(Jess utilise la reflexion Java pour récupérer la valeur de l'attribut hostname de
    event)

(addHostComponent PDAComponent event->hostname)
)
```

Listing 5.8 – Exemple de règle pour le moteur DeployWare/Jess

Le principal avantage de ce moteur de règles est que la syntaxe pour écrire les politiques d'autonomie utilise la syntaxe de Jess, facile à prendre en main. De cette manière, il est envisageable de configurer le composant `start_engine` de l'architecture d'auto-gestion à l'aide d'une propriété de logiciel correspondant à un fichier de règles Jess. Le principal inconvénient réside dans le fait que si l'on souhaite injecter des conditions plus complexes dans les politiques (*e.g.* des conditions sur l'état actuel du domaine d'auto-gestion) il faut injecter les informations sous forme de types Jess, et donc traduire l'architecture de la machine virtuelle dans le langage de Jess.

5.4 Utilisation de la machine virtuelle DeployWare pour déploiements sur grille

Parmi les environnements ouverts, les grilles de calcul [33] soulèvent également des défis, et plus généralement le déploiement à très large échelle. En effet, le nombre de machines hôtes impliquées dans le déploiement sur grilles de calcul est très grand. Le nombre de logiciel à

déployer grandit donc proportionnellement au nombre des machines, et il en va de même pour les dépendances à gérer. De plus, tout déploiement sur une grille de calcul nécessite de réserver à l'avance les ressources nécessaires. Cela ajoute une étape au travail de l'administrateur. Ce dernier doit : effectuer des réservations de machines sur la grille, récupérer les adresses de ces machines, décrire le déploiement d'un grand nombre de logiciels. Cette dernière tâche peut d'ailleurs s'avérer répétitive. Par exemple, si l'administrateur souhaite déployer une application à base de composants CORBA sur mille machines, il va devoir déployer 1000 fois le serveur de composants CORBA, ce qui représente une tâche répétitive et ennuyeuse pour l'administrateur système. De plus, dans la machine virtuelle DeployWare de base, le déploiement se fait de manière séquentielle. Or, dans un environnement tel que les grilles de calcul, il est impensable d'envisager un déploiement entièrement séquentiel, sous peine de faire face à des temps de déploiement redhibitoires. Le premier défi posé par le déploiement sur grilles de calcul consiste à trouver un moyen d'encapsuler dans la machine virtuelle DeployWare un mécanisme de réservation de machines, et de décrire de la manière la plus naturelle possible les différents logiciels à installer sur ces machines. À ce défi s'ajoute le besoin de paralléliser l'exécution de la machine virtuelle, et ce sans corrompre l'ordre d'exécution des dépendances.

De plus, le déploiement de logiciels sur grilles de calcul se heurte également à des problèmes techniques. En effet en utilisant la machine virtuelle DeployWare pour déployer un grand système logiciel sur grille, des problèmes de ressources se posent si la taille du domaine de déploiement devient trop grande. En effet, si un grand nombre de logiciels doivent être déployés en parallèle, la machine depuis laquelle la machine virtuelle DeployWare est démarrée ne possèdera pas assez de ressources pour déployer des logiciels en parallèle. Cette limitation de ressources peut concerner le nombre de processus parallèles (les *threads*) pouvant coexister, la mémoire vive disponible, ou encore le nombre de connexions réseau (les *sockets*). Le second défi du déploiement sur grilles de calcul consiste à optimiser le fonctionnement de la machine virtuelle en terme d'utilisation des ressources système.

Notre proposition pour que la machine virtuelle DeployWare passe à l'échelle consiste en deux points visant à relever chacun des deux défis soulevés. Le premier élément consiste à concevoir un nouveau type de logiciel, spécifique à la grille, capable d'encapsuler les mécanismes de réservation de machines pour la grille. En complément de ce logiciel, l'utilisation de la construction `foreach` de l'extension de Fractal ADL présentée dans 5.2 permet d'exprimer la répétition du déploiement d'un logiciel sur plusieurs nœuds. Cela a pour but de rendre plus concis les descripteurs de déploiement d'application à large échelle.

Le second élément que nous proposons consiste à distribuer le fonctionnement de la machine virtuelle DeployWare elle-même. En effet en distribuant la machine virtuelle sur plusieurs machines, il est possible de mutualiser les ressources utilisées par la machine virtuelle. Nous évaluerons le gain, en terme de temps de déploiement, d'une distribution de la machine virtuelle.

5.4.1 Réservation et parallélisation du déploiement

La grille de calcul étant un réseau partagé par plusieurs utilisateurs, le déploiement d'une application dans ce genre d'environnements nécessite d'effectuer une réservation préalable de

ressources. Divers outils existent pour gérer les réservations de ressources sur grilles de calcul. Parmi ceux-ci, OAR [7] est l'outil utilisé pour gérer les ressources de la grille de calcul expérimentale française *GRID'5000* [12]. Cet outil permet de lancer en ligne de commandes une requête de réservation de ressources. Dans cette requête, il est possible de fixer le nombre de machines désirées, ainsi que le délai de reservation, c.-à-d. le durée durant laquelle la réservation sera valide. En retour de cette commande est créé un fichier contenant la liste des noms d'hôtes des machines réservées.

Dans le cadre d'un déploiement à l'aide de la machine virtuelle de base DeployWare, l'administrateur système doit d'abord effectuer lui-même la réservation, puis doit exploiter lui-même cette liste de machines, pour ensuite écrire son descripteur de déploiement, et notamment la description des machines hôtes, dans le langage DeployWare. De plus, il est rare qu'un administrateur qui effectue un déploiement sur 500 machines doive déployer 500 logiciels différents. Il est même probable qu'il doive lancer quelques processus serveurs identiques sur chacune des machines, avant de déployer son application métier (serveurs de composants, serveurs d'application, démons, etc.). Ainsi il va devoir, dans son descripteur DeployWare, recopier 500 fois le même logiciel en modifiant la machine hôte.

Tout d'abord, nous ajoutons trois nouveaux mécanismes dans la bibliothèque DeployWare. En premier lieu, nous introduisons un nouveau type de logiciel, le composant `GRID5000.OAR`, qui représente un logiciel qui lance une requête de réservation, et mémorise le résultat dans un fichier donné en propriété du logiciel. Le deuxième mécanisme est un composant `Compute-Hostname` capable de lire le fichier généré par `GRID5000.OAR` et de renvoyer un à un les noms d'hôtes des machines réservées. Le troisième mécanisme que nous ajoutons est le mécanisme de `DynamicHost` qui est un type de machine hôte particulier, qui ne possède pas de nom d'hôte fixé statiquement. Ce composant d'accès dynamique à la machine hôte prend en paramètre un composant de calcul `ComputeHostname` qui lui envoie un nom de machine hôte disponible et non encore attribué à un autre `DynamicHost`.

Ainsi en utilisant le mécanisme d'itération `apply` de Fractal ADL, il est possible de décrire de manière succinte un ensemble de machines réservées sur la grille. Une telle déclaration est représentée sur le listing 5.9, qui encapsule la réservation de cinquante nœuds sur Grid5000 et définit autant de machines hôtes DeployWare correspondant aux nœuds réservés.

```
GRID.TEST  {
  oar = GRID5000.OARGRID(oar_args n=50, /tmp/nodes);
  compute  = INTERNET.COMPUTEHOSTNAME(/tmp/nodes);

  /* The declaration of nodes */
  Hosts = GRID5000.G5K_NETWORK {

  g5k-nodes {
    apply ForEachInIntegerRange(i,1,50) {
      node-%{i} = GRID5000.DYNAMICNODE {
        user      = INTERNET.USER(jdubus, ,~/.ssh/id_rsa);
        compute  = /;
}}}}
```

Listing 5.9 – Extrait de définition DeployWare qui déclare 50 nœuds réservés sur la grille

[7] `http://oar.imag.fr`

À ces constructions, nous ajoutons également un nouveau type de composants rendant possible le déploiement d'un certain nombre de logiciels en parallèle. Ce nouveau type de composant, appelé `FDF.PARALLEL-COLLECTION`, est en fait un composite abstrait qui contient un composant gestionnaire, chargé de créer des threads pour déployer les logiciels contenus dans le composite. Dans ces différents threads sont invoquées les opérations de l'interface `deployment` des différents logiciels présents dans le composite. Le déploiement de ces logiciels se fait alors en parallèle.

Le listing 5.10 montre un ensemble de logiciels déployés en parallèle sur les noeuds réservés dans le listing 5.9. Les logiciels déployés en parallèle dans cet exemple sont des machines virtuelles Java.

```
GRID.TEST  {
 // ...
  fc-servers-cluster = FDF.PARALLEL-COLLECTION {
        apply ForEachInIntegerRange(i,1,50) {
            jvm-%{i} = JAVA.JRE {
                home = JAVA.HOME(/tmp/fractal-server);
                archive = JAVA.ARCHIVE(/tmp/archives/my_jre.tar.gz);
                host = Hosts/g5k-nodes/node-%{i};
}}}}
```

Listing 5.10 – Extrait de définition DeployWare qui décrit le déploiement en parallèle de JVM

5.4.2 Distribution de la machine virtuelle DeployWare

Lorsque l'administrateur effectue un déploiement sur la grille, il peut maintenant utiliser le logiciel DeployWare encapsulant les requêtes pour réservation de ressources. Néanmoins, au delà d'un certain nombre de machines, le déploiement ne croît plus de manière linéaire, comme nous le montre la courbe en figure 5.16.

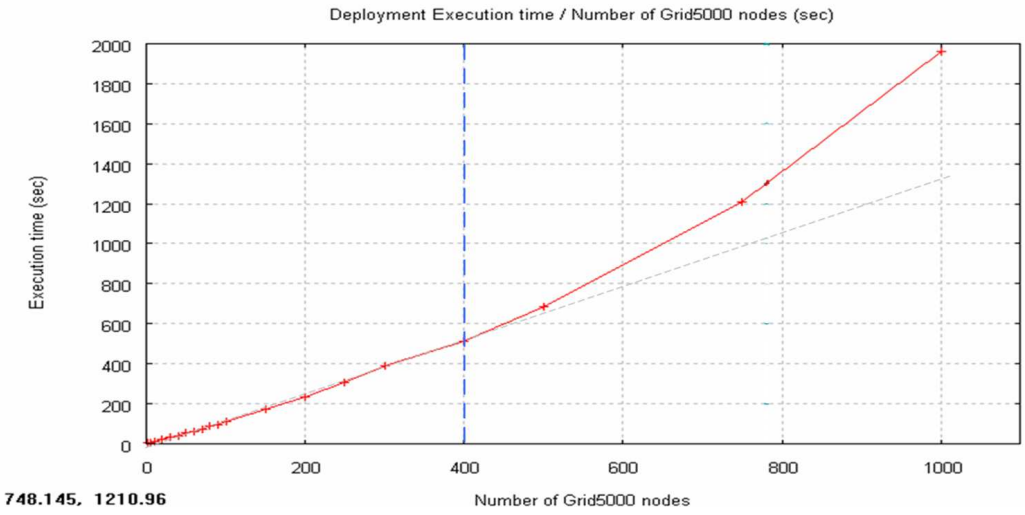

FIG. 5.16 – Mesures du temps de déploiement avec un serveur DeployWare

Les chiffres de ce diagramme représentent le temps de déploiement (en ordonnée) en fonction du nombre de serveurs de composants CORBA déployés (en abscisse), à raison d'un serveur par nœud. Cette courbe montre que le temps de déploiement est linéaire jusqu'à 400 nœuds. Cette limite représente la limite de ressources que nous avons énoncé précédemment. A partir de 800 nœuds en parallèle, le système se retrouve saturé en nombre d'ouverture de sockets. Il doit donc attendre que certaines sockets se libèrent avant de pouvoir poursuivre le déploiement. Le problème est de plus en plus important à mesure que le nombre de logiciels à déployer grandit, et le temps de déploiement croît exponentiellement.

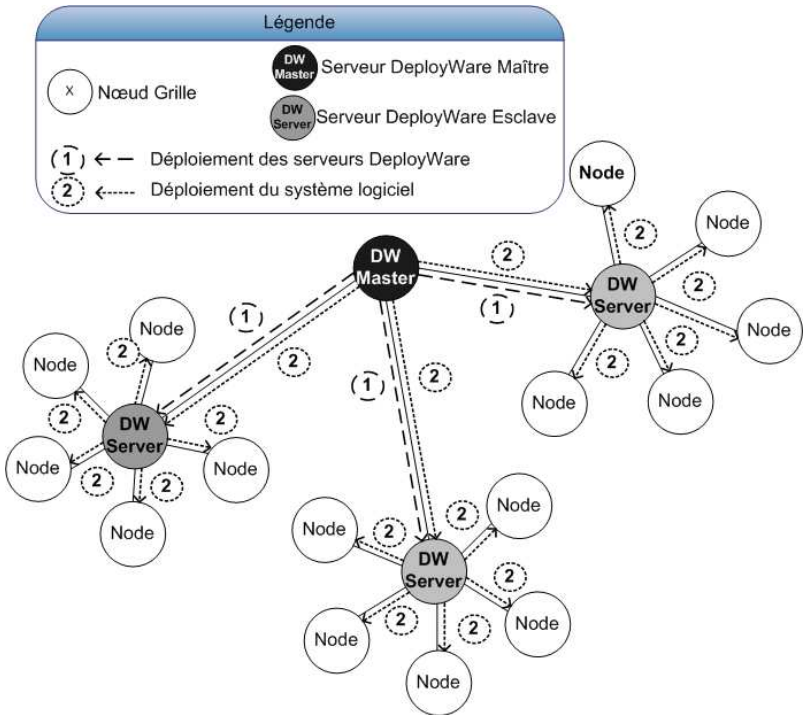

FIG. 5.17 – Architecture distribuée de la machine virtuelle pour déploiements à larges échelles

Pour remédier à ce problème, nous avons mis en place un mécanisme de distribution de la machine virtuelle DeployWare. L'architecture de cette distribution est représentée sur la figure 5.17. Cette architecture se compose d'un serveur DeployWare central (`DW Master`) qui dans un premier temps déploie les serveurs auxiliaires (`DW Server`). Dans un second temps, le serveur central distribue les tâches de déploiement aux auxiliaires. En utilisant la bibliothèque Fractal RMI, il est possible de distribuer le composite global de la machine virtuelle DeployWare sur plusieurs machines. Le nombre de sockets disponibles pour le déploiement (ou toute autre ressource système) est alors multiplié par le nombre de machines sur lesquelles le composite DeployWare est distribué. Les mêmes mesures ont été reproduites en distribuant la machine virtuelle sur la courbe de la figure 5.18. Ces courbes montrent clairement que le temps de déploiement décroit nettement au fur et à mesure que l'on distribue la machine virtuelle sur plusieurs machines.

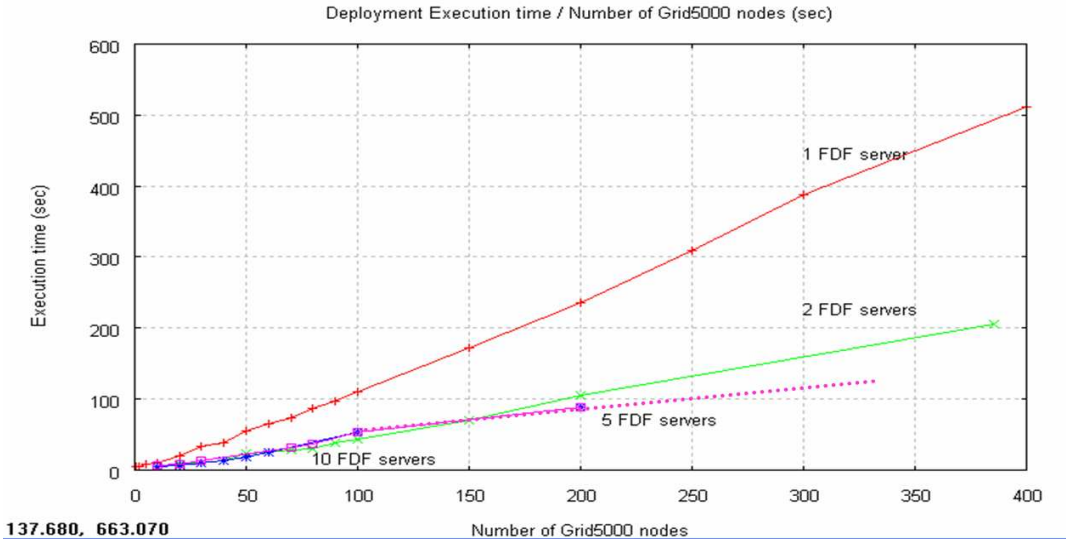

FIG. 5.18 – Mesures du temps de déploiement en distribuant le serveur DeployWare sur plusieurs machines

Enfin, puisqu'elle est distribuée, se pose le problème du déploiement de la machine virtuelle de déploiement elle-même. En effet, cette machine virtuelle étant réalisée en Fractal, des serveurs de composants Fractal doivent être démarrés sur les machines sur lesquelles s'étend la machine virtuelle, puis la description de la machine virtuelle doit être déployée via Fractal ADL sur les nœuds Fractal distants. Pour remédier à ce problème, nous avons réalisé un type de logiciel `DeployWare`, qui permet de déployer automatiquement la machine virtuelle elle-même sur plusieurs machines.

5.5 Conclusion/Synthèse

L'exécution du déploiement des systèmes logiciels répartis dans notre approche est réalisée à l'aide d'une approche à base de composants Fractal. L'architecture de la machine virtuelle est très similaire d'un déploiement à un autre, et l'utilisation des motifs architecturaux de Fractal ADL rend la définition d'un déploiement très aisée. Nous avons donc simplifié son expression à l'aide d'un langage avec peu de concepts, qui permet de décrire de manière concise le déploiement de n'importe quel système. Ce langage est très proche du méta-modèle DeployWare ce qui facilite la projection des concepts du méta-modèle définis dans le chapitre 4. À ce jour, cette projection reste manuelle, uniquement par manque de temps pour réaliser la transformation de modèles automatiquement. Néanmoins, la principale limitation de cette approche réside dans la relative difficulté d'étendre l'ensemble d'instructions de la bibliothèque DeployWare, à moins de posséder des connaissances dans le modèle de composants Fractal.

La description des systèmes auto-gérés en environnements ouverts distribués est également rendue possible grâce à des composants de la machine virtuelle dédiés à la description de boucles de contrôle. À l'aide du paradigme ECA, il est possible de modifier dynamiquement le déploiement du système en fonction d'événements tels que l'arrivée de nouvelles machines hôtes. Pourtant l'expression des boucles de contrôle ne permet pas de faire l'analyse de l'ensemble du système afin de prendre une décision, à moins d'opter pour le moteur de règles en Fractal, et donc de posséder une bonne connaissance, là encore, du modèle de composants Fractal.

Enfin un certain nombre d'améliorations, telles que l'encapsulation des mécanismes de réservation, l'utilisation de l'itération `foreach` de Fractal ADL, ou encore la distribution de la machine virtuelle DeployWare elle-même, ont été apportées afin de rendre la machine virtuelle DeployWare entièrement adéquate pour un déploiement sur des grilles de calcul à très large échelle.

La machine virtuelle DeployWare est disponible sous licence LGPL à l'adresse suivante : `http ://fdf.gforge.inria.fr`. Les composants pour le déploiement d'un grand nombre de technologies sont déjà disponibles. Dans la bibliothèque de personnalités DeployWare on trouve des technologies à base de services (*e.g.* PEtALS JBI, ActiveBPEL, Tuscany SCA), à base de JEE (*e.g.* JOnAS, JBoss, Geronimo), ou de composants (*e.g.* OpenCCM). Des implantations de composants pour l'accès aux machines sont également disponibles : SH, CSH et WinCommand pour le shell, SSH et Telnet pour le protocole, FTP et SCP pour le transfert.

Enfin, la machine virtuelle DeployWare a été intégrée dans les serveurs JOnAS, PEtALS et JASMINE du consortium OW2, comme outil de déploiement.

Chapitre 6

Déploiement auto-géré d'architectures de composants avec DACAR

Sommaire

6.1 Introduction

Dans ce chapitre, nous présentons notre contribution pour le quatrième et dernier rôle de notre découpage de la procédure de déploiement : le rôle d'architecte métier. Une fois la couche intergicielle déployée par l'administrateur système, un ensemble de *noeuds d'exécution —i.e.* des serveurs offrant le support d'exécution aux applications métier— est prêt à recevoir les applications. C'est à ce moment que l'architecte métier intervient. Son rôle est de concevoir et déployer l'application métier devant s'exécuter sur les noeuds d'exécution en utilisant un paradigme approprié.

Nous choisissons de poser l'hypothèse selon laquelle l'architecte métier utilise le paradigme de programmation à base de composants pour réaliser son application. En effet, il s'agit d'un paradigme de mieux en mieux accepté du monde industriel, et le sujet de toujours autant

d'études de recherche. Le paradigme de services Web [55] reste néanmoins souvent opposé à celui de composants. Cependant, de plus en plus de travaux tendent à rapprocher ces deux paradigmes, et notamment l'approche *Service Component Architecture* du consortium OpenSOA [54]. Cette approche propose, entre autres, d'utiliser un langage de description d'architecture semblable à ceux utilisés pour les composants pour décrire des architectures de services. Ceci légitime un peu plus notre choix de se limiter pour l'instant au paradigme de programmation par composants.

6.1.1 Contexte

L'approche dite *constructive* est employée pour assembler et configurer une application répartie à base de composants. Le concepteur définit une architecture logicielle à partir d'un assemblage de composants logiciels qu'il prend *sur l'étagère*. Ensuite, il configure cet assemblage afin de procéder à la projection de celui-ci vers un système concret sur une plate-forme d'exécution choisie. Aujourd'hui, cette approche de programmation à base de composants s'est imposée dans de nombreux environnements, y compris les applications sur grilles de calcul [34] et l'informatique ubiquitaire [29], c'est-à-dire les *environnements ouverts distribués*. L'approche constructive pour assembler des applications à base de composants est supportée par des langages dédiés à la description d'architectures, les *Architecture Description Languages* (ADL) [49] que nous avons étudiés dans le chapitre 2. De tels langages servent à décrire de manière déclarative l'assemblage de composants et sa projection vers la couche intergicielle du système logiciel déployé par l'administrateur système.

Dans ce chapitre, nous appelons *architecture abstraite* l'assemblage tel qu'il est décrit dans un ADL, et *architecture concrète* l'assemblage d'instances de composants en train de s'exécuter sur une machine hôte. Classiquement, le processus de déploiement d'une application métier à base de composants consiste en la projection de l'architecture abstraite vers l'architecture concrète. Néanmoins, dans un environnement ouvert distribué, il n'est pas possible de déclarer exhaustivement cette projection puisqu'il est impossible de déterminer ni de prévoir par avance le domaine de déploiement, c.-à-d. l'ensemble des machines disponibles. À ce jour, aucun ADL ne permet de prendre en compte le caractère ouvert de l'environnement sur lequel est déployée une application, comme nous l'avons vu dans la section 2.2.

6.1.2 Problématique

Lors de la conception, l'architecte définit l'architecture abstraite de son application en utilisant un ADL. Cette description est transformée en l'assemblage de composants physiques devant être déployé sur la couche intergicielle, c.-à-d. l'architecture concrète. Mais, une fois ce déploiement effectué, l'architecture abstraite est déconnectée de l'architecture concrète, et l'utilité de l'ADL s'arrête là. La figure 6.1 schématise ce processus classique de déploiement d'architecture à composants : l'architecture abstraite est transformée en architecture concrète. Après la phase de déploiement, cette architecture concrète n'est plus reliée à l'architecture abs-

traite. Ainsi, l'architecture abstraite est déconnectée des changements survenant au niveau du domaine de déploiement ou de la couche intergicielle : elle n'est plus exploitable.

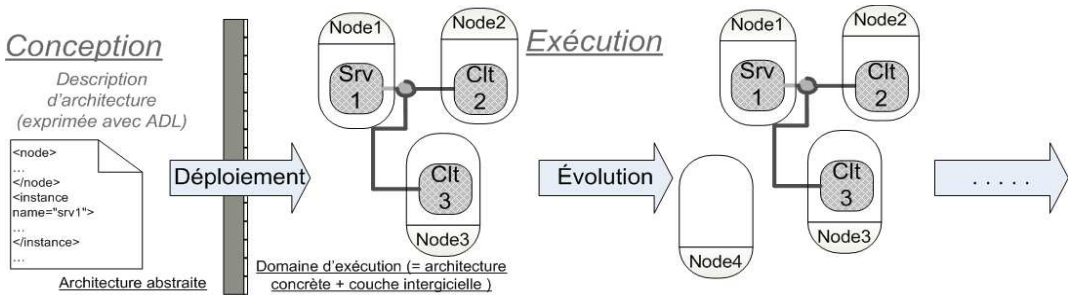

FIG. 6.1 – Après le déploiement, l'architecture abstraite n'est plus exploitée

À l'instant T du déploiement, l'architecture abstraite, et donc la surcouche d'abstraction de l'application métier, est perdue. La vision modulaire à un haut niveau d'abstraction de l'application métier laisse place aux composants et à leurs liaisons, assez techniques à manipuler quelle que soit la technologie employée. Toute opération sur l'architecture une fois le déploiement effectué doit être exprimée à l'aide de concepts très techniques et très bas niveau.

6.1.3 Motivations

Notre premier objectif consiste à maintenir une couche d'abtraction disponible pendant l'exécution de l'application métier. De cette manière, il est possible de continuer à administrer l'application en utilisant les mêmes concepts, au même niveau d'abstraction, que ceux utilisés pour la description du déploiement de l'architecture. En effet, dans notre vision, le déploiement au sens transformation d'architecture abstraite vers architecture concrète, représente une étape de l'administration de ce logiciel. Les autres étapes d'administration comme la reconfiguration dynamique ou la mise à jour doivent être exprimées avec le même langage, les mêmes concepts.

Notre second objectif consiste à injecter de manière transverse de nouveaux concepts afin d'exprimer des politiques d'autogestion de l'architecture pendant son exécution. De cette manière, il est possible d'adresser le déploiement des architectures en environnements ouverts distribués. Nous souhaitons que ces politiques soient exprimées au même niveau que le déploiement de l'architecture, car il s'agit d'informations relatives au déploiement, en l'occurrence le déploiement dynamique. Ainsi le besoin de maintenir une couche d'abstraction pour représenter l'architecture avec des concepts de haut niveau est renforcé par la volonté d'injecter ces mécanismes d'expression de l'auto-gestion censés utiliser une couche de connaissance abstraite.

6.1.4 Lien causal

Pour maintenir une couche d'abstraction exploitable à l'exécution, il est nécessaire d'instaurer un *lien causal* entre la couche d'abstraction et l'assemblage de composants représenté. Le terme lien causal représente une liaison à double sens entre la couche d'abstraction d'une application et l'application elle-même. L'objectif de ce lien est d'assurer qu'à tout moment la couche d'abstraction représente effectivement l'assemblage de composants qui s'exécute sur le domaine de déploiement. Le premier sens de cette liaison, de l'application vers l'abstraction, est appelée l'*observation*. Ce sens assure que toute modification survenant au niveau de l'architecture concrète est bien reportée au niveau de l'architecture abstraite, c.-à-d. la couche d'abstration est mise à jour en temps réel en fonction des évènements survenant au niveau de l'architecture concrète. Le second sens du lien causal, de l'architecture abstraite vers l'architecture concrète, est appelée le *déploiement*[1]. Ce lien consiste à assurer que toute modification au niveau de l'architecture réifiée est effectivement appliquée au niveau de l'architecture concrète.

Afin de prendre en charge les environnements ouverts, nous proposons d'injecter des mécanismes d'auto-gestion dans le processus classique de déploiement d'architectures à composants. Pour cela, l'architecture abstraite doit être réifiée en un modèle en mémoire persistant à l'exécution, qui matérialise la partie *connaissance* de la boucle de contrôle[2]. Ainsi, nous définissons également le terme *architecture réifiée* comme étant la réification en mémoire manipulable dynamiquement de l'architecture abstraite. Nous définissons également le *domaine d'exécution* comme étant l'association de la couche intergicielle et de l'architecture concrète déployée dessus. Dans le cadre de l'application d'une démarche d'informatique auto-gérée, le lien causal doit être maintenu dynamiquement entre le domaine d'exécution et l'architecture réifiée pendant le déploiement, mais également au delà. Cela signifie qu'une modification dans le domaine d'exécution doit déclencher la modification équivalente dans l'architecture réifiée, et *vice versa*. L'architecture globale de notre contribution est représentée sur la figure 6.2 et comporte deux grandes parties.

1. **Architecture réifiée** : Il s'agit de la réification sous forme d'un graphe d'objets, des concepts exprimés dans les descripteurs d'architectures. Nous imposons qu'un découpage (plus ou moins explicite) soit possible au sein de cette réification :

   La partie *Type* La première composante contient la définition des types et implantations de composants ainsi que des assemblages constituant l'application. Cette partie permet de factoriser la définition des types de composants. Ainsi, au moment de la définition des instances, le type de l'instance ne doit pas être redéfini exhaustivement, mais pointe juste sur la définition du type correspondant.

   La partie *Domaine* La seconde contient les informations descriptives sur le domaine de déploiement de l'application considérée. Il s'agit donc de la déclaration des noeuds disponibles, des différentes interconnexions entre ces noeuds, et les propriétés qui

[1] Déploiement prend ici le sens de la projection des concepts de l'architecture abstraite vers l'architecture concrète.

[2] ce concept a déjà été détaillé dans la section 2.5

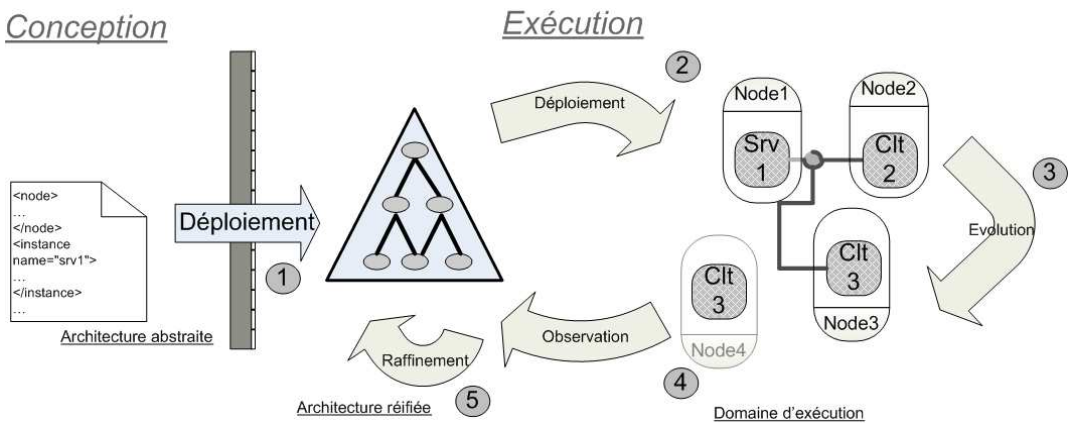

FIG. 6.2 – Durant l'exécution, l'architecture réifiée est causalement liée à l'architecture concrète

se rapportent à cet environnement. Cette partie est indispensable dans la perspective d'une gestion des environnements ouverts. L'architecture réifiée représentant la connaissance, les informations sur les nœuds se doivent d'être présents.

La partie *Plan* Enfin, cette partie décrit le plan de déploiement, c.-à-d. l'affectation d'instances de composants définis dans la partie *Type* sur les noeuds définis dans la partie *Domaine*. Elle comprend également la description des liaisons entre ces différentes instances afin que l'application soit correcte.

2. **Domaine d'exécution :** Il s'agit de l'architecture concrète et de l'environnement dans lequel elle évolue, c.-à-d. la plate-forme à composants employée. Elle comprend les artefacts relatifs à l'application (composants à l'exécution, liaisons entre composants, etc.), que l'on appellera *architecture concrète* dans la suite, ainsi que les différentes variables du contexte de l'application (noeuds d'accueil, capacité de calcul des noeuds, etc.)

D'abord, le processus *classique* de déploiement a lieu. Ce processus consiste à transformer l'architecture abstraite en architecture réifiée (phase 1 de la figure 6.2). Puis, cette architecture réifiée est transformée en architecture concrète (phase 2). Ensuite, le domaine d'exécution, qui comprend potentiellement un environnement ouvert distribué, évolue en termes d'entrées et sorties de nouveaux nœuds (phase 3). Ces évolutions sont observées et l'architecture réifiée est mise à jour avec les changements (phase 4). Enfin, suite à certains changements de l'architecture réifiée, des raffinements de cette dernière peuvent être exécutées sous forme de politiques d'auto-gestion (phase 5).

6.1.5 L'architecture de notre contribution : DACAR

DACAR est notre contribution pour réaliser le déploiement et l'autogestion des architectures à base de composants. Cette contribution consiste en une architecture générique, comprenant trois grands acteurs : l'architecture réifiée, le moteur de politiques et le domaine d'exécu-

tion. Nous proposons également un certain nombre d'API afin de définir la manière d'interagir avec ces différents acteurs. Cette architecture globale est représentée sur la figure 6.3.

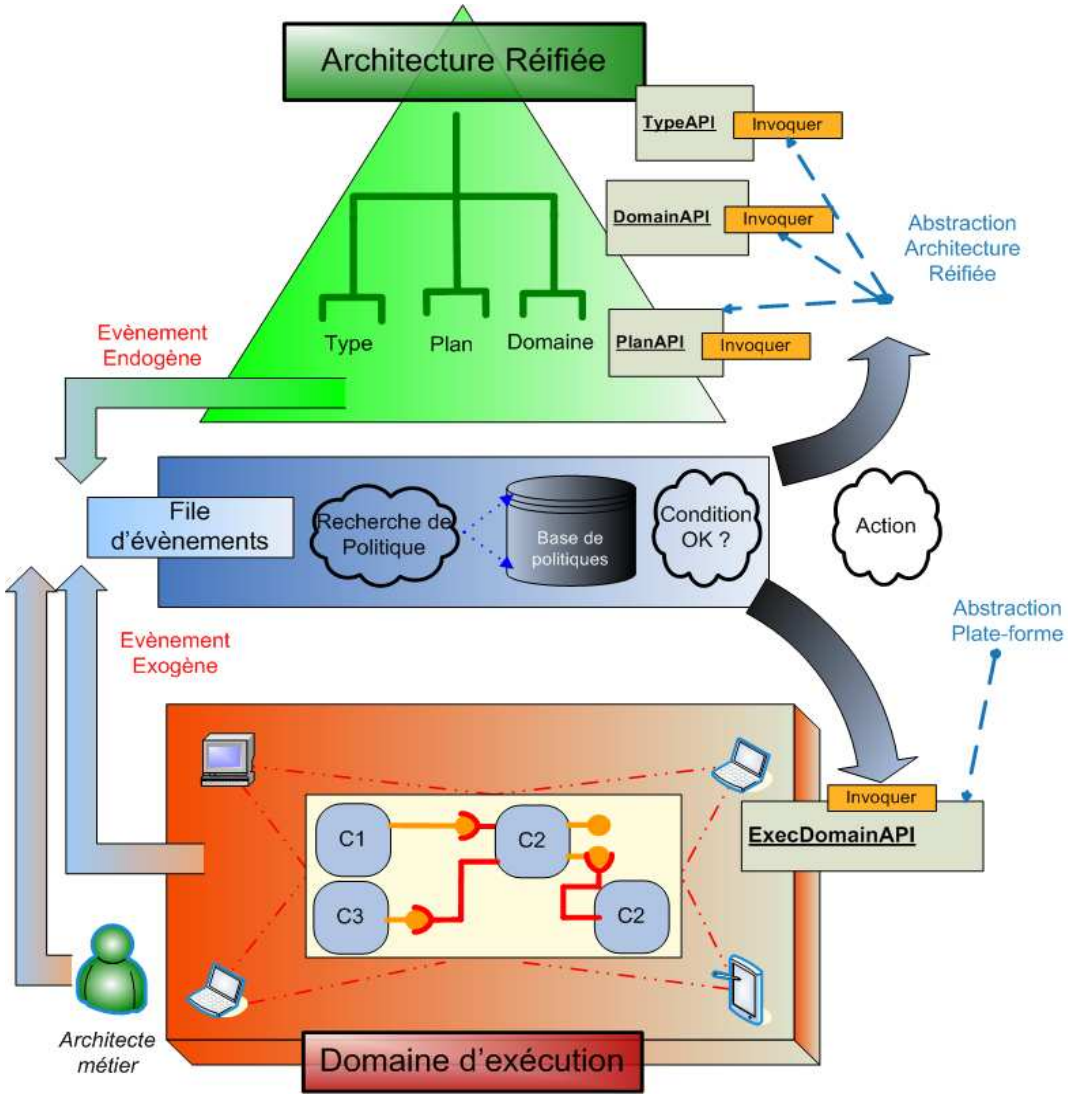

FIG. 6.3 – Vue d'ensemble du prototype DACAR

En premier lieu, nous définissons un certain nombre d'API afin de posséder un moyen uniforme pour reconfigurer l'architecture réifiée ou le domaine d'exécution. En effet, ces deux ensembles doivent pouvoir être modifiés dynamiquement, ce qui implique que des opérations de modification soient implémentées. Ces API sont représentées sous forme UML sur la figure 6.4.

Ainsi l'API définie pour la partie `Domain` de l'architecture réifiée offre des opérations pour ajouter la déclaration d'un nouveau noeud d'exécution (opération `addNode()`), et pour supprimer des déclarations de noeuds d'exécution (opération `removeNode`). L'API définie

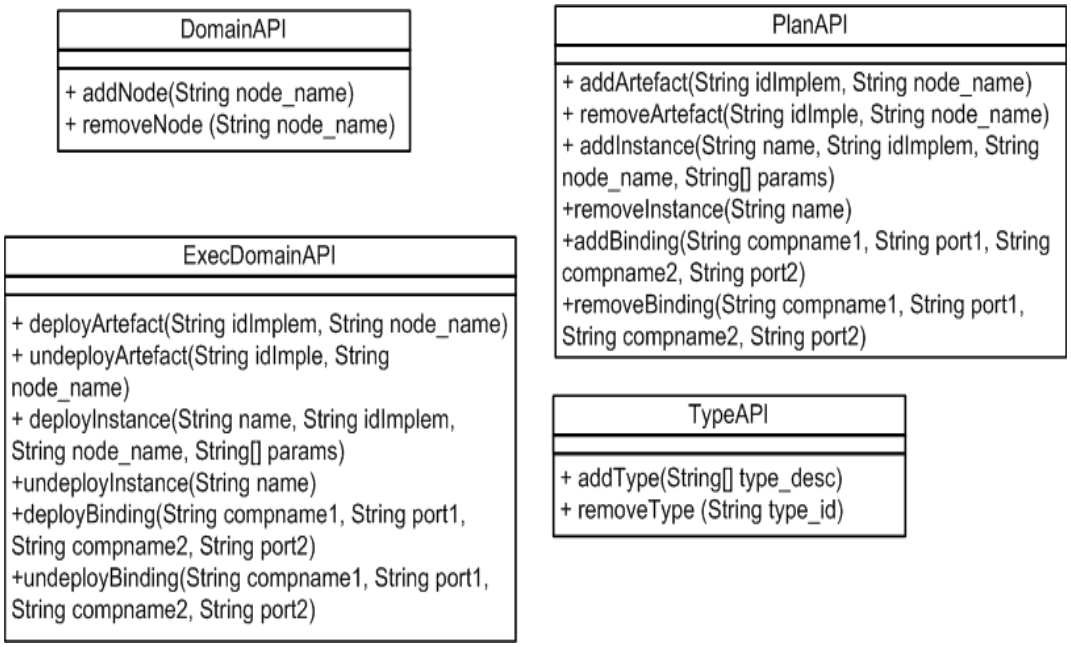

FIG. 6.4 – API des différentes parties de la contribution

pour la partie `Type` permet d'ajouter (opération `addType()`) ou retirer (opération `removeType()`) des types de l'application. L'API définie pour la modification du plan de l'architecture réifiée contient elle davantage d'opérations. Une première opération `addArtefact` (*resp.* `removeArtefact`) permet d'ajouter (*resp.* supprimer) la déclaration d'une fabrique de composant sur une machine donnée. Il est à noter que ce concept de fabrique de composants n'est pas obligatoire dans tous les modèles de composants, mais il existe certains modèles comme le modèle de composants CORBA pour lesquels l'installation d'une fabrique (la *home*) est obligatoire sur un noeud. Une seconde opération `addInstance` (*resp.* `removeInstance`) permet d'ajouter (*resp.* retirer) la déclaration d'une instance de composant d'un certain type, sur un certain noeud, avec un certain nombre de paramètres. Enfin, une troisième opération `addBinding` (*resp.* `removeBinding`) permet d'ajouter (*resp.* retirer) la déclaration d'une liaison entre le port requis d'un composant en paramètre, et le port fourni d'un autre composant, également en paramètre.

Dans un second temps, nous proposons également d'implémenter cette architecture d'environnement de conception et de déploiement d'applications auto-gérées à base de composants répartis. Cela implique trois grands difficultés à résoudre. Le premier défi à résoudre consiste à trouver un langage générique de description d'architecture à base de composants, afin que tout architecte logiciel soit en mesure de manipuler ce langage pour construire son application. Le deuxième défi à relever consiste à trouver un langage pour exprimer les politiques d'autogestion de l'application, notamment pour prendre en charge l'arrivée ou le départ de nouvelles machines dans le domaine de déploiement (donc de nouveaux serveurs de composants potentiels dans la couche intergicielle). Enfin le troisième défi à relever consiste à concevoir un pro-

totype de moteur de déploiement capable 1) d'interpréter le langage d'architecture générique et de traduire ce langage en appels sur l'API du modèle de composants choisi, 2) de maintenir le lien causal entre architecture réifiée et domaine d'exécution et 3) d'observer les événements survenant au niveau du domaine d'exécution et d'appliquer les politiques d'auto-gestion le cas échéant.

Dans la section 6.2, nous présentons le langage de description d'architecture générique que nous utilisons dans notre approche et nous motivons ce choix. Dans la section 6.3, c'est le langage d'expression des reconfigurations qui est présenté et motivé. La section 6.4 est dédiée à DACAR notre prototype de moteur de déploiement prenant en entrée une description d'architecture auto-gérée et qui la déploie selon un modèle de composants donné, en assurant le lien causal entre architecture réifiée et architecture concrète, et appliquant les règles d'auto-gestion décrites. Enfin, la section 6.5 présente un exemple simple afin d'exposer la méthodologie à adopter pour mettre en œuvre la démarche que nous mettons en place.

6.2 Modèle d'architecture métier générique

Dans notre contribution, nous ne souhaitons pas restreindre le modèle de composant utilisé par l'architecte métier. En effet, notre motivation est de maintenir globalement l'hétérogénéité des technologies dans le processus de déploiement. Notre environnement DeployWare répond déjà à cette propriété, et nous souhaitons poursuivre dans cette voie avec DACAR. Ainsi le langage que l'architecte métier doit utiliser pour concevoir son application doit être générique.

Plutôt que de se pencher sur l'élaboration d'un nouveau langage d'architecture générique, nous préférons réutiliser un modèle existant. En effet, certains travaux se concentrent désormais sur des langages de description d'architecture générique comme xADL [23] ou encore ACME [35]. Néanmoins, nous souhaitons utiliser la spécification définie par l'OMG *Deployment and Configuration of Distributed Component-based Applications* [52]. En effet, suite aux lacunes sur le déploiement de la spécification du modèle de composants CORBA, l'OMG a décidé de se pencher sur un méta-modèle complet pour le déploiement et la configuration d'applications à base de composants. Si cette spécification reste facilement projetable vers le modèle CCM, elle reste néanmoins suffisamment générique pour exprimer un assemblage de composants de n'importe quelle technologie.

Cette spécification, nous l'avons vu dans la section 2.3.3, est découpée en deux grandes parties : la première spécifie la structure des données pour représenter le déploiement d'une application à base de composants (c'est le *data model*), la seconde spécifie les interfaces des différents acteurs du déploiement (c'est le *target model*). Nous nous intéressons uniquement à la partie données de ce méta-modèle, puisque nous cherchons un langage pour la description d'architecture.

Les raisons de notre choix d'OMG D&C sont multiples. Tout d'abord, il s'agit d'un des seuls standards de langage d'architecture générique. En effet, d'autres langages sont dits génériques mais ne sont reconnus par aucun organisme de standardisation. Le modèle OMG D&C

est soutenu par l'OMG qui est l'un des organismes de standardisation les plus reconnus. Beaucoup des spécifications issus des groupes de travail de l'OMG sont encore largement acceptées dans la communauté du génie logiciel (*e.g.* les spécifications CORBA, UML, MDA ou encore MOF sont autant de références dans le monde du génie logiciel). De plus, certains ADL génériques restent purement descriptifs, ou se concentrent sur des parties plus formelles de la description d'architecture logicielle. La spécification OMG D&C est, par définition, orientée vers le déploiement et configuration, ce qui fait que ce méta-modèle contient beaucoup de concepts pour spécifier au mieux les méta-informations utiles pour un moteur de déploiement chargé d'interpréter ces fichiers.

Enfin, la partie *data model* de la spécification correspond parfaitement à nos besoins car elle est en effet scindée en trois parties. La première partie est dédiée à la définition des types de composants et des assemblages. La seconde partie est quant à elle dédiée à la description des noeuds d'exécution chargés d'héberger les composants logiciels durant l'exécution. La troisième partie est dédiée à la description d'instances de composants, la description de l'affectation de ces instances sur les noeuds d'exécution définis dans la seconde partie, et enfin la description des interconnexions entre ces différentes instances.

Ces trois parties correspondent rigoureusement aux trois grandes lignes requises pour notre langage d'architecture. Dans la suite, nous appelons ces parties `Type`, `Domain` et `Plan`. Ces trois grandes parties représentent en effet les données nécessaires et suffisantes pour effectuer le déploiement, la configuration et la reconfiguration d'une application métier à base de composants.

La spécification OMG D&C se matérialise sous la forme d'un méta-modèle qui est également fourni par l'OMG sous plusieurs formes (dont un schéma XML). Ainsi, l'architecte logiciel dispose d'un premier paquetage qui lui permet d'exprimer les types des différents composants qu'il met en jeu dans son application. Il peut également, le cas échéant, décrire les différents nœuds d'exécution dont il possède statiquement la connaissance (la prise en charge des environnements ouverts intervient ultérieurement). Enfin il possède les concepts nécessaires à la description d'instances de ces composants, ainsi que de leurs interconnexions.

6.3 Modèle d'expression des politiques d'auto-gestion

Pour mettre en place notre approche, nous avons besoin d'étendre le langage d'architecture choisi afin que l'architecte métier puisse décrire les politiques d'auto-gestion qui concernent son architecture.

Pour mener à bien cette démarche d'auto-gestion, trois types de reconfiguration dynamique doivent être mis en œuvre. Tout d'abord, il faut assurer le lien causal entre l'architecture réifiée et le domaine d'exécution. Cela signifie que quand un événement provient du domaine d'exécution, cet événement doit être reporté dans l'architecture réifiée. Il faut également assurer l'autre sens du lien causal, c.-à-d. assurer qu'un événement provenant de l'architecture réifiée est répercuté au niveau du domaine d'exécution. Enfin, quand un événement, comme

un nouveau noeud d'exécution est démarré, survient, il faut appliquer la règle d'auto-gestion existante pour cet événement.

Le concept d'événement est commun à ces trois types de reconfiguration dynamique. Ainsi nous définissons un événement comme étant la réification d'un changement survenu dans l'une des deux composantes du modèle défini précédemment. Cet événement peut provenir de l'architecture elle-même, il est alors dit *endogène* (*exemple* : une nouvelle instance est ajoutée dans la partie Plan). Mais il peut aussi être déclenché par le domaine d'exécution voire par l'administrateur de l'application, et il est *exogène* (*exemple* : un nouveau noeud est démarré sur la plate-forme). Un événement doit contenir les informations pertinentes à propos du changement occasionné, afin de pouvoir effectuer la reconfiguration adaptée.

Notre contribution consiste à décomposer la politique d'adaptation dynamique de l'architecture sous formes de politiques qui suivent le paradigme *Évènement-Condition-Action (ECA)* [22]. Une politique est composée de trois parties. Une partie événement qui désigne le type d'événement qui doit déclencher cette politique. La seconde partie regroupe les conditions que les propriétés de l'événement doivent remplir afin de valider le déclenchement de la politique. Enfin une partie action définit les différentes opérations de reconfiguration à effectuer, ainsi que la cible sur laquelle opérer ces reconfigurations.

Nous définissons dans notre modèle deux types d'événements et deux domaines d'action pour les politiques, il y a donc quatre types de politiques possibles. Nous choisissons cependant de séparer nos politiques en trois catégories :

- Les *politiques de déploiement*. Elles sont déclenchées par un événement endogène. Elles opèrent une reconfiguration au niveau de la plate-forme ou de l'architecture concrète. Elles projettent les concepts de l'architecture réifiée vers une architecture concrète. *Exemple* : *Lorsque la description d'une instance est ajoutée à la partie Plan de l'architecture réifiée (sans condition supplémentaire), alors déployer l'instance décrite sur la plate-forme*. Ces politiques assurent ainsi l'un des sens du lien causal entre architecture réifiée et domaine d'exécution.
- Les *politiques d'observation*. Elles sont déclenchées par un événement exogène. Elles opèrent une reconfiguration au niveau de l'architecture réifiée. Elles ont pour but de mettre à jour l'architecture réifiée afin de la notifier d'éventuels changements au niveau de la plate-forme ou de l'application déployée. *Exemple* : *Lorsqu'un noeud est démarré sur la plate-forme, alors ajouter la déclaration de ce noeud dans la partie Domaine de l'architecture réifiée*. Ces politiques assurent l'autre sens du lien causal entre domaine d'exécution et architecture réifiée.
- Les *politiques architecturales*. Elles sont déclenchées par un événement endogène. Elles opèrent une reconfiguration au niveau de l'architecture réifiée elle-même. Elles offrent un moyen d'assurer des propriétés architecturales qui dépendent d'un changement provenant de l'architecture. Un exemple est de vouloir assurer que toutes les instances de composants `MyComponent` doivent avoir une liaison avec un composant `Logger` connu : *Lorsqu'une instance de composant* C *de type* `MyComponent` *est déclarée dans la partie Plan de l'architecture réifiée, alors déclarer une liaison entre* C *et* Logger *dans la partie Plan*.

Le dernier type de politique consiste à opérer une reconfiguration directement au niveau de la plate-forme en réponse à un événement exogène provenant de celle-ci. Nous préconisons de découper cette politique en une succession de politiques d'observations, suivies d'une série éventuelle de politiques architecturales, et enfin une série de politiques de déploiement. De cette manière, l'architecture réifiée est consciente de tout changement dans l'architecture concrète, et les deux couches restent causalement liées.

Les politiques doivent être les plus élémentaires possible. En effet, il est possible avec ce paradigme de spécifier des politiques à granularité très fine qui vont favoriser la réutilisation et la maintenance. Il est préférable de chaîner les différentes actions en fonction des événements produits par les actions antérieures. Les déclenchements des politiques se font alors *en cascade*, les actions des politiques restent indépendantes les unes des autres, et donc réutilisables et maintenables séparément.

Ainsi, nous proposons un cadre de conception afin que l'architecte logiciel puisse exprimer ces politiques dans un langage donné, de manière transverse à la définition d'architecture. Nous voyons dans la prochaine section comment l'architecte métier peut exprimer ses politiques d'auto-gestion à l'aide de composants Fractal, dans le cadre de notre prototype de moteur de déploiement d'architectures génériques auto-gérées : DACAR.

6.4 Prototype DACAR de déploiement d'architectures métiers auto-gérées

Dans cette section, nous présentons quelques détails techniques de la réalisation de notre prototype DACAR pour réaliser le déploiement d'architectures auto-gérées décrites de manière générique grâce au méta-modèle OMG D&C. Nous nous focalisons également dans cette section sur l'approche mise en œuvre pour que l'architecte métier soit capable d'exprimer ses règles de reconfiguration.

Réaliser notre approche consiste à implémenter les différentes interfaces des API pour les différentes technologies choisies. Concernant l'architecture réifiée, le modèle OMG D&C étant générique, ces API sont implantées une fois pour toutes. En revanche, concernant l'API du domaine d'exécution, celle-ci se doit d'être implémentée par un expert de la technologie en question. Pour les besoins du prototypage, deux implémentations de cette API ont été écrites : l'une pour la plate-forme à composants CORBA OpenCCM[3] et l'autre pour la plate-forme à composants Fractal.

Ce prototype a lui-même été implémenté à l'aide d'une architecture à base de composants Fractal. L'architecture est représentée sur la figure 6.5.

Pour la construction de l'architecture réifiée, il faut écrire un programme capable de lire en entrée une architecture exprimée dans le langage OMG D&C, afin d'en construire une graphe

[3] `http://openccm.ow2.org`

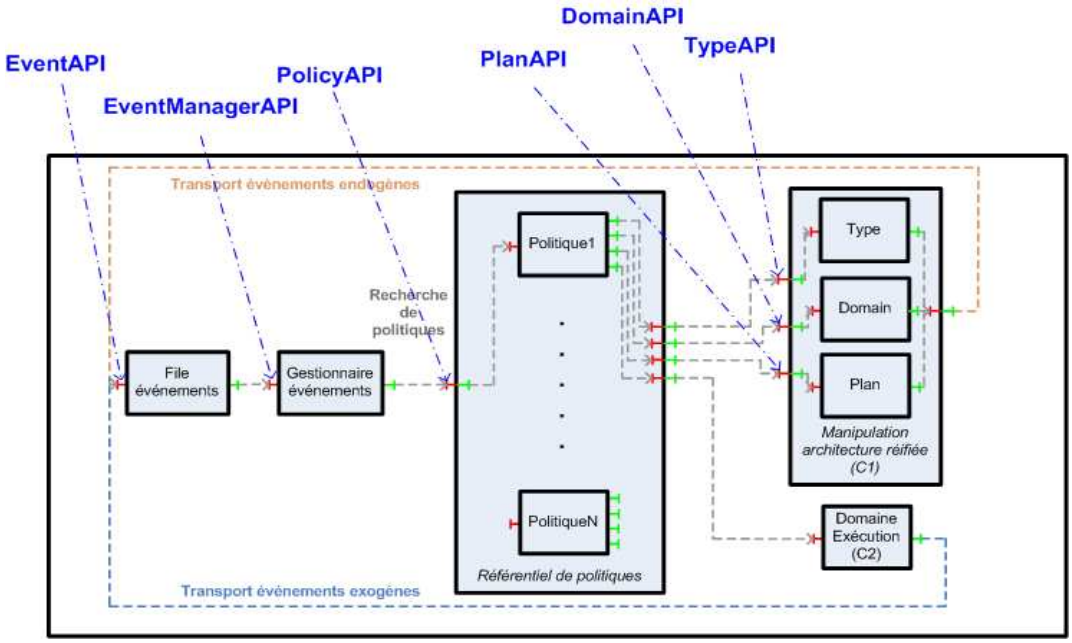

FIG. 6.5 – Architecture Fractal du prototype DACAR

d'objets correspondant à la réification des concepts de la spécification. Pour cela, nous choisissons d'exploiter le schéma XML fourni avec la spécification.

Le canevas JAXB[4] défini par Sun permet de générer un ensemble de classes Java correspondant aux concepts définis dans un schéma XML. Ce canevas génère également les outils de transformation d'un fichier XML, conforme au schéma XML, vers un graphe d'instances des classes générées. Ainsi l'utilisation de JAXB nous permet : 1) de générer un *parser* de fichiers OMG D&C automatiquement à partir du schéma XML, et 2) de disposer de l'API générée par le biais des classes pour manipuler un graphe d'objets OMG D&C dynamiquement.

Un premier composant (C1) qui encapsule l'architecture réifiée a donc été réalisé en Fractal. Il utilise notamment l'API générée par JAXB pour manipuler le graphe d'objets durant l'exécution, sans jamais violer les règles de structure sur ce graphe imposées par le schéma XML d'OMG D&C. Ce composant possède également une interface requise de type `EventAPI` qui permet d'envoyer les événements endogènes vers le moteur d'exécution des politiques d'auto-gestion que nous détaillons dans la suite.

Le composant (C2) qui encapsule le domaine d'exécution possède quant à lui deux implémentations : la première pour implémenter les opérations de l'API pour le déploiement de composant CCM, et la deuxième pour le modèle de composants Fractal. Ces deux implémentations reposent sur l'un de nos travaux antérieurs : le cadre de conception à base de composants

[4] `http://java.sun.com/developer/technicalArticles/WebServices/jaxb/`

pour les machines de déploiement de composants répartis présenté dans [31] et [27]. Le code des opérations de l'API peut également manipuler directement l'API de déploiement de la technologie choisie. Il est à noter que ce composant d'encapsulation du domaine d'exécution possède également une interface requise de type `EventAPI`. Cette interface est utilisée pour acheminer les événements exogènes vers le moteur d'exécution des politiques d'auto-gestion.

Il reste donc à concevoir les composants responsables de l'application des politiques d'auto-gestion. L'API des différentes interfaces des composants de politiques sont représentées sur la figure 6.6.

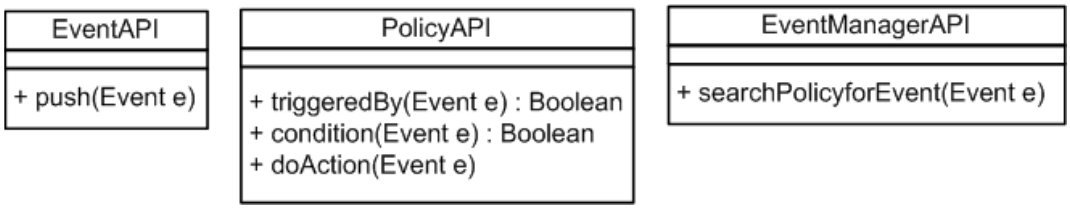

FIG. 6.6 – API des composants de gestion de politique

Ces politiques d'auto-gestion sont à la fois responsable du lien causal entre l'architecture réifiée et le domaine d'exécution, au travers des politiques de *déploiement* et d'*observation*. Le troisième type de politique d'auto-gestion sert à reconfigurer l'architecture réifiée en réponse à des changements provenant de l'architecture réifiée elle-même, c'est-à-dire des politiques d'auto-gestion spécifiques à une application donnée. Chacun des types de composants existants doit posséder une interface fournie de type `PolicyAPI` et doit être connectée à l'interface cliente de même type du gestionnaire d'événements. Sinon cette politique ne pourra jamais être déclenchée. Pour cela, nous réutilisons le moteur de règles présenté dans le chapitre 5, en l'adaptant.

Nous définissons tout d'abord un composant chargé d'enregistrer les événements, qu'ils soient exogènes ou endogènes. Ce composant renferme la logique de gestion des événements : elle peut être séquentielle ou parallèle selon les besoins et la quantité d'événements susceptibles de survenir dynamiquement.

Le deuxième composant pour la gestion des politiques est le gestionnaire d'événements. Son rôle est de chercher, pour un événement donné, parmi les règles existantes celles qui sont déclenchées par ce type d'événements. Si les conditions énoncées sont remplies, alors ce composant lance l'exécution de l'action de reconfiguration de cette politique.

Les politiques de *déploiement* sont des composants possédant une interface requise de type `ExecDomainAPI` puisque leur action se situe au niveau du domaine d'exécution ; les opérations qui peuvent être appelées par les politiques de déploiement se situent exclusivement au niveau du domaine d'exécution. Les politiques d'*observation* sont des composants possédant trois interfaces requises de type `DomainAPI`, `PlanAPI` et `TypeAPI` puisque leur action se situe au niveau de l'architecture réifiée ; les opérations qui peuvent être appelées par les politiques d'observation se situent exclusivement au niveau de l'architecture réifiée. Enfin les

politiques *architecturales* sont des composants possédant également trois interfaces requises de type `DomainAPI`, `PlanAPI` et `TypeAPI` puisque leur action se situe au niveau de l'architecture réifiée ; les opérations qui peuvent être appelées par les politiques architecturales se situent exclusivement au niveau de l'architecture réifiée.

Ainsi, pour réaliser les politiques d'auto-gestion de son application, l'architecte métier doit développer les composants Fractal correspondants. Il existe néanmoins une bibliothèque DACAR contenant un certain nombre de règles prêtes à être réutilisées. Notamment, les règles d'observation comme par exemple la déclaration, dans l'architecture réifiée, d'un nouveau nœud d'exécution apparu dynamiquement sont présentes dans la bibliothèque. Le code de cette règle est représenté sur le listing 6.1.

```
package org.objectweb.mdploy.archRuleEngine.rules;

import dacar.archRuleEngine.events.Event;
import dacar.archRuleEngine.events.NodeAddingExt;
@Component
public class DeclareNewNodeRule implements PolicyAPI{

  /**
   * Annotation Fraclet pour marquer le champ comme
   * étant une interface requise dont le nom est
   * ${name} dans le fichier ADL
   */
  @Requires (name="type_description_action")
  protected TypeAPI type_modif ;

  /**
   * Annotation Fraclet pour marquer le champ comme
   * étant une interface requise dont le nom est
   * ${name} dans le fichier ADL
   */
  @Requires (name="plan_description_action")
  protected PlanAPI plan_modif ;

  /**
   * Annotation Fraclet pour marquer le champ comme
   * étant une interface requise
   * ${name} dans le fichier ADL
   */
  @Requires (name="domain_description_action")
  protected DomainAPI domain_modif ;

  public boolean checkEvent(Event e) {
    // L'interface Event définit une opération qui permet
    // de récupérer le type de l'événement
    return e.getType().equals("NodeAddingExt") ;
  }

  public boolean checkCondition(Event e) {
    // Dans cette règle, aucune condition n'est mise
    // mais il est envisageable de raffiner cette règle, pour
    // par exemple, limiter l'ajout de machines seulement aux PDA
    return true ;
  }

  public void doAction(Event e) {
    // Ici seulement la partie domaine de l'architecture réifiée
```

```
    // est utilisée pour la mise à jour
    this.domain_modif.addNode(((NodeAddingExt)e).node_name()) ;
  }
}
```

Listing 6.1 – Code du composant de règle d'observation d'un nouveau noeud d'exécution

De la même manière, certaines règles de déploiement basiques, comme la règle qui consiste à exécuter le déploiement d'une instance de composant déclarée dans l'architecture réifiée, sont présentes dans la bibliothèque DACAR. Le code du composant Fractal implantant cette règle de déploiement d'une instance de composant est donné sur le listing 6.2

```
package dacar.archRuleEngine.rules;

import dacar.archRuleEngine.events.Event;
import dacar.archRuleEngine.events.InstanceDescriptionAdding;
@Component
public class DeclareNewInstanceDeploymentRule implements PolicyAPI {
  /**
   * Annotation Fraclet pour marquer le champ comme
   * étant une interface requise
   * ${name} dans le fichier ADL
   */
  @Requires (name="platform_action")
  protected ExecDomainAPI execDomain ;

  public boolean checkEvent(Event e) {
    return e.getType().equals("InstanceDescriptionAdding") ;
  }

  public boolean checkCondition(Event e) {
    return true ;
  }

  public void doAction(Event e) {
    // On connait le type de l'événement déclenchant
    // on peut donc caster l'objet afin d'utiliser son API
    InstanceDescriptionAdding ida = (InstanceDescriptionAdding)e ;
    this.execDomain.deployInstance(ida.getName(),ida.getSource(),ida.getNode(),ida.
        getParams());
  }
}
```

Listing 6.2 – Code du composant de règle de déploiement d'une nouvelle instance de composant

6.5 Exemple illustratif

Dans cette section, nous illustrons l'utilisation de notre prototype DACAR dans le cadre d'un exemple d'application à base de composants. Pour la lisibilité de l'exemple, nous exprimons nos politiques sous forme textuelle simple, proche du langage naturel. Cette application est destinée à être exécutée dans un environnement ouvert, en l'occurrence un environnement ubiquitaire. L'application que nous choisissons de déployer est une application à base de composants réalisée dans le cadre du projet RNTL AMPROS [57]. Le but de cette application, appelée *Plan Rouge* est de mettre en place une infrastructure de communication ubiquitaire

pour la gestion de crise en cas d'incident de grande ampleur, ou de catastrophe naturelle. Un diagramme d'une version simplifiée (nécessaire à l'étude des politiques DACAR) de l'architecture de l'application est représentée en figure 6.7. Trois types de composants sont impliqués. Le type `VictimManager` qui correspond au composant central chargé de maintenir la liste des blessés, il propose deux ports fournis *collector* pour venir mettre à jour la liste des blessés, et *browser* pour consulter cette liste. Le type `GlobalView` qui correspond à un composant de visualisation avancée de chaque blessé. Il propose un port requis *browser* pour se connecter sur le port fourni équivalent du `VictimManager`. Enfin le type `Rescuer` correspond à un type de composant pour la création de fiches pour chacune des victimes du sinistre. Ces fiches sont ensuite soumises au `VictimManager` en utilisant l'interface requise `collector`.

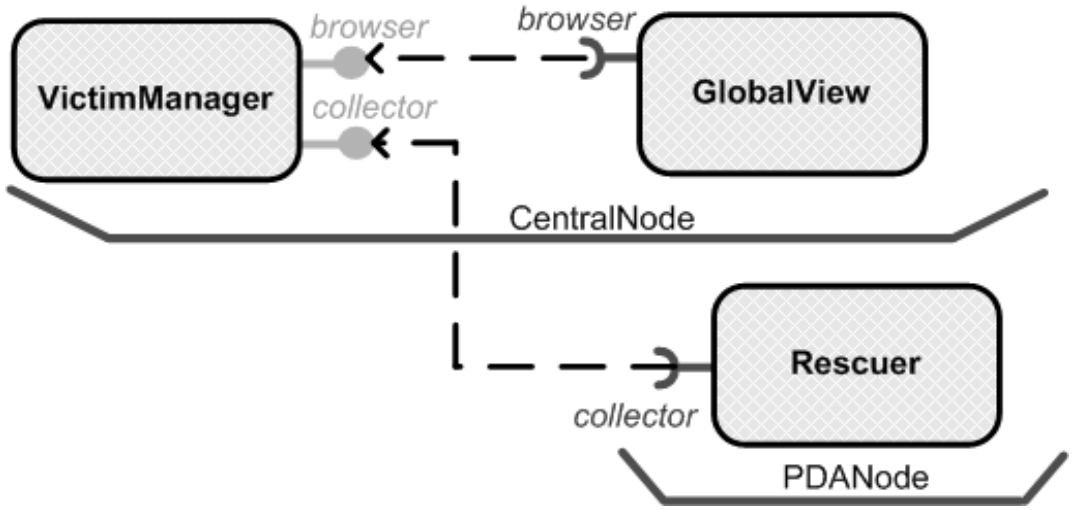

FIG. 6.7 – Architecture de l'application plan rouge

Dans ce scénario de gestion de crise, les secours médicaux sont organisés autour de trois zones sur le lieu de l'accident : le poste médical avancé (PMA), le centre de commandement et la zone du sinistre proprement dite. Tous les personnels sont équipés de PDA et les victimes sont progressivement installées dans des sarcophages (sorte de sacs de couchage chauffants) dotés de capteurs médicaux reliés à un équipement embarqué communicant. Les pompiers sont organisés en noria pour la collecte des victimes : installation des sarcophages, saisie des fiches médicales de l'avant et évacuation vers le PMA. Soit sur le terrain soit dans le PMA, les médecins complètent les fiches médicales de l'avant avec un diagnostic pour les décisions d'orientation et d'évacuation.

Le scénario de déploiement est le suivant : les deux composants `VictimManager` et éventuellement `GlobalView` (nous discuterons l'utilité de ce composant plus loin), sur une machine centrale, supposée fixe. Chaque pompier entrant en intervention, allume un PDA qui entre dans le domaine de déploiement[5]. Un composant `Rescuer` est alors automatiquement déployé sur ce PDA puis automatiquement connecté au `VictimManager`.

[5] On omet ici volontairement l'installation du serveur de composants

Un nouveau pompier arrivant sur le lieu de la catastrophe se voit attribuer un PDA. L'installation du composant `Rescuer` est géré avec les deux règles qui suivent. Le première politique, représentée sur le listing 6.3, déclenche l'ajout d'une instance `Rescuer` dans la partie Plan lorsqu'un nouveau terminal est ajouté dans la partie *Domaine*. Il s'agit d'une politique architecturale, qui est déclenchée à la suite de l'exécution de la règle d'observation énoncée dans la section précédente (listing 6.1). La seconde politique, représentée sur le listing 6.4, choisit l'implantation en fonction du profil du terminal. Lorsque le même pompier quitte le plan rouge, il rend le PDA ; le repliement des composants applicatifs est de la même manière automatisé.

```
*************
* EVÈNEMENT *
*************
Noeud N déclaré dans Domaine

***********
* ACTION  *
***********
plan.declareInstance(type=Rescuer},name=P_N},hostnode=N )
```

Listing 6.3 – Politique architecturale pour le déploiement d'une instance de `Rescuer`

```
*************
* EVÈNEMENT *
*************
Instance I de type Rescuer déclarée dans Plan sur Noeud N

*************
* CONDITION *
*************
I.getImplementation() == null

**********
* ACTION *
***********
I.setImplementation(getAdequateImplForProfile(N,'Rescuer'));
```

Listing 6.4 – Politique architecturale pour la sélection d'une implémentation de `Rescuer`

Le placement de certains composants de la partie fixe de l'application peut être auto-géré également, de manière à ce que les capacités de la machine hôte soient optimisées. Ainsi une politique de placement auto-géré de composant peut s'exprimer avec le prototype DACAR. Sur le listing 6.5, figure une règle qui déploie le composant `VictimManager` sur un nœud possédant une puissance de calcul suffisante et hébergeant une base de données. Pour la simplicité de l'expression de cette règle, on considère qu'une opération `getNodeWithPrefs()` est disponible afin d'introspecter la partie domaine de l'architecture réifiée, et y trouver un nœud qui satisfasse les contraintes.

```
*************
* EVÈNEMENT *
*************
Instance VM de type='VictimManager' déclarée dans Plan

*************
* CONDITION *
```

```
*************
VM.getNode() == null

**********
* ACTION *
**********
Map prefs;
prefs.put('processor','1Ghz');
prefs.put('memory','512Mo');
prefs.put('database','true');
VM.setNode(getNodeWithPrefs(prefs));
```

Listing 6.5 – Exemple de politique de placement d'instance de composant

Des composants dans l'architecture du scénario plan rouge peuvent être optionnels. Par exemple, le composant `GlobalView` , qui correspond à un composant d'interface graphique évoluée, et donc couteux en terme de ressources, n'est pas indispensable à l'application. Dès lors, il est intéressant de pouvoir écrire une politique (représentée sur le listing 6.6) qui permet d'exprimer que le déploiement du composant GlobalView ne peut se faire que si la machine possède suffisament de mémoire.

```
*************
* EVÈNEMENT *
*************
Instance GV type='GlobalView' déclarée dans Plan sur noeud N

*************
* CONDITION *
*************
N.getFreeSpace() < thisRule.getImplementationRepository().get(GlobalView).getSize()

**********
* ACTION *
**********
plan.removeInstance(name='GV')
```

Listing 6.6 – Exemple de règle de déploiement conditionnel en fonction des ressources

L'utilisation des règles de type ECA permet non seulement de rendre possible l'adaptation auto-gérée de l'architecture, mais aussi de simplifier l'expression de la politique d'adaptation.

6.6 Conclusion

Dans ce chapitre, nous avons présenté l'approche que nous préconisons pour l'architecte métier. Cette approche a été implémentée par le prototype DACAR.

Dans un premier temps, ce prototype propose de modéliser l'application à base de composants indépendamment de toute technologie en utilisant le méta-modèle OMG D&C pour modéliser son architecture. Il existe à ce jour peu de plates-formes de déploiement capables de prendre en charge un format standardisé de description d'architecture. Après diverses expérimentations de notre prototype pour diverses applications, il apparait que le choix du méta-modèle de l'OMG présente quelques limitations. En effet, comme nous l'avons noté dans la section [**?**], ce méta-modèle comporte un pléthore de concepts parmi lesquels seulement certains d'entre eux se sont avérés utiles. En outre, cette spécification semble aujourd'hui finalement assez peu acceptée, en dehors de la communauté CORBA. Ainsi, l'apprentissage du

langage OMG D&C pour l'architecte métier peut s'avérer aussi contraignant que pour une technologie dédiée, car le langage est très verbeux comme en atteste l'extrait en annexe A, et le standard assez peu reconnu. Enfin les concepts eux-mêmes sont parfois peu clairs, et il est souvent nécessaire de chercher longuement dans la spécification afin de trouver la matérialisation en OMG D&C d'un concept basique des modèles à composants. Enfin le profondeur des concepts est bien souvent rédhibitoire pour le concepteur (la hauteur de l'arbre XML présenté dans l'annexe A le montre clairement). Un travail ultérieur pourra consister à établir un méta-modèle minimaliste comprenant les concepts strictement nécessaires à la définition d'une architecture logicielle, quelle que soit le modèle de composants choisi.

Dans un deuxième temps, le prototype DACAR permet à l'architecte système d'exprimer des politiques pour 1) maintenir comme il le souhaite le lien causal entre architecture réifiée et domaine d'exécution et 2) définir des politiques d'auto-gestion pour son architecture, pouvant se baser sur le caractère ouvert des environnements. Parmi les approches existantes pour la reconfiguration dynamique des architectures à base de composants, il en existe peu capable de s'adapter aux fluctuations d'un environnement ouvert distribué. De plus, non seulement la description de l'architecture, mais également la description des politiques d'auto-gestion est faite de manière totalement transparente de la technologie choisie. Néanmoins la réalisation de ces règles est programmative et impose à l'architecte métier la connaissance du modèle de composants Fractal. La réalisation d'un *Domain Specific Language* est une perspective évidente de ce travail.

Ce constat renforce le besoin d'établir, dans un travail futur, un méta-modèle de politique d'auto-gestion. Ce méta-modèle, outre les concepts de description des politiques, pourra contenir des concepts orientés vérification permettant de lier certains concepts, comme nous l'avons fait dans le méta-modèle DeployWare. Il serait envisageable ainsi d'écrire des programmes de validation pour les politiques d'auto-gestion. En effet, il existe un certains nombres d'erreurs qui peuvent survenir à l'exécution, à cause des politiques d'auto-gestion, notamment lorsque le paradigme ECA est utilisé comme dans DACAR. Certaines de ces erreurs sont dûes à l'interaction entre plusieurs politiques d'auto-gestion. Ces types d'interaction entre politiques réactives ont été identifiées dans le monde des bases de données actives [59]. Par exemple, le déclenchement en chaîne de politiques, connu sous le nom de *Sequential Action Interaction*, peut être toléré dans l'exécution d'un système autonome, mais la séquence est malgré tout à surveiller pour qu'elle n'aboutisse pas à une *Looping Interaction*, c.-à-d. un cycle dans l'exécution des politiques. Cependant, aucun des travaux sur l'informatique auto-gérée ne se préoccupe aujourd'hui de la fiabilité à l'exécution de telles reconfigurations dynamiques. Outre la détection de cycles, il y a un certain nombre de propriétés qu'un système auto-géré peut fournir comme la convergence (c.-à-d. tout ensemble de politiques exécutées dans un ordre quelconque, aboutit à un même état du système), le déterminisme de l'application des politiques (c.-à-d. l'application d'une même politique aboutit toujours au même état du système).

Ainsi, sans que cela ait été implémenté dans le prototype DACAR, nous proposons, dans le futur, de définir un méta-modèle de politiques d'auto-gestion, contenant un certain nombre de concepts servant à la vérification statique des politiques [28]. Le comportement d'auto-gestion d'une architecture est défini en utilisant des politiques ECA dans DACAR. Nous proposons

d'introduire le concept de *règle d'intention* pour rendre possible les vérifications. Ce concept permet de définir, pour une action donnée, l'événement résultant susceptible d'être produit. De cette manière, l'architecte logiciel exprime ses intentions dans une politique : il précise quel changement est susceptible de survenir lors de l'exécution d'une action précise. En utilisant cette association, il est possible de construire un graphe d'exécution des politiques, en fonction des changements existants déclarés par l'architecte métier. La figure 6.8 décrit comment utiliser ce graphe pour déceler d'éventuels cycles, qui peuvent conduire le système dans un état incohérent, et en avertir l'architecte avant d'effectuer le déploiement. À partir de la liste des événements existants, il est possible d'obtenir les différentes actions qui vont être exécutées (par les politiques déclenchées par l'événement). Grâce au concept de règle d'intention, il est également possible de calculer les différents événements produits après exécution de ces règles. Ainsi en reproduisant le processus, il est possible d'obtenir le graphe global et de détecter les cycles.

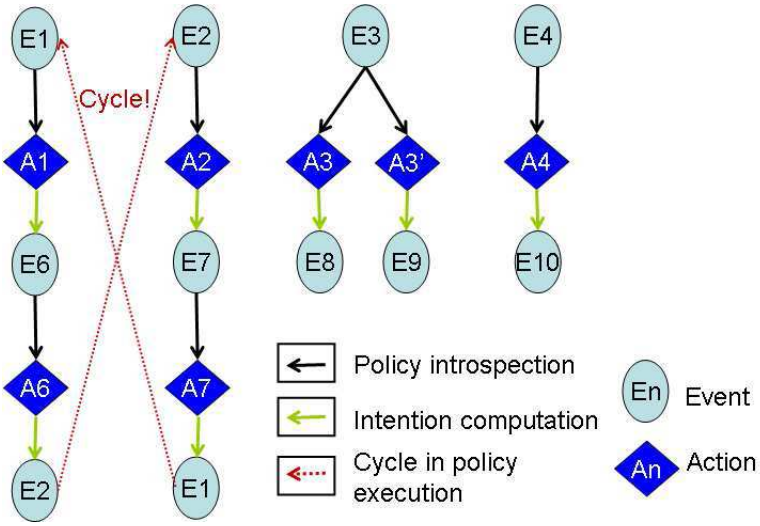

FIG. 6.8 – Détection de cycles à l'aide du graphe établi avec les règles d'intention

Le prototype de DACAR est disponible à l'adresse suivante : `http://www.lifl.fr/~dubus/dacar`.

Quatrième partie

Validation de la contribution

Chapitre 7

Cas d'étude

Sommaire

Dans ce chapitre, nous allons valider notre approche en la confrontant à des problèmes de déploiement en grandeur réelle. Ce chapitre est également l'occasion d'exposer la manière d'utiliser DeployWare et plus principalement sa machine virtuelle, ainsi que DACAR, pour effectuer des déploiements de systèmes logiciels dans différents contextes. Dans ce chapitre, nous n'étudions pas les instances de modèles DeployWare, mais leur traduction équivalente en langage de machine virtuelle DeployWare. Nous étudions deux cas, qui vont permettre de valider DeployWare dans plusieurs cas de figure.

Tout d'abord, nous étudierons un premier exemple, l'agence de voyages, mettant en jeu un très grand nombre de technologies différentes. En effet, ce scénario applicatif, basé sur une architecture orientée services, présente une pléthore de technologies différentes et d'applicatifs à déployer sur ces technologies. En outre, le schéma de dépendances entre les différents logiciels de ce scénario est très complexe, et l'ordre de déploiement des logiciels est peu évident à établir pour un administrateur système. Cet exemple nous permettra notamment de valider la pertinence des concepts de DeployWare pour faire face à un grand nombre de technologies et à un schéma de dépendances complexe.

Le deuxième exemple que nous étudierons, un serveur d'horaires de train pour terminaux mobiles dans une gare, est un exemple de déploiement en environnement ouvert distribué. En effet, dans ce scénario applicatif, le nombre de machines hôtes entrant dans le réseau est

inconnu et imprévisible. Nous verrons comment l'utilisation de DeployWare et de DACAR permet de prévoir et d'étendre le déploiement de l'application et de ses dépendances, à chaque terminal entrant sur le domaine de déploiement.

7.1 Agence de voyages

Dans cette section, nous présentons l'agence de voyages à base de services distribués, puis nous décrivons la description et l'exécution de ce déploiement à l'aide de la machine virtuelle DeployWare. Nous nous pencherons sur la définition de composants de logiciels DeployWare à l'aide du langage dédié, présenté dans le chapitre 5.

7.1.1 Scénario

L'agence de voyages possède des partenariats avec une compagnie aérienne et une chaine hôtelière comme représenté sur la figure 7.1. Chaque partenaire est localisé dans son propre domaine, en utilisant des technologies et des serveurs indépendants, conformément au paradigme orienté services [10], qui standardise non pas les serveurs d'applications, mais les interactions entre composants. L'objectif pour l'agence de voyage est d'offrir à ses clients une application de réservation en ligne. Cette application Web doit permettre à un client de sélectionner une date ainsi qu'une destination pour son voyage. L'application ensuite effectue la réservation pour un vol et une chambre d'hôtel, conformément aux données renseignées par le client. Pour effectuer un tel scénario, l'agence de voyages se doit de contacter la compagnie aérienne et la chaine hôtelière afin d'effectuer les réservations. Comme les partenaires exposent déjà leurs services, l'agence de voyages a l'opportunité d'ouvrir son système d'information afin d'intégrer ces services externes. Ainsi, l'agence de voyages peut développer sa propre application Web en orchestrant différents services existants.

Une approche SOA de ce type nécessite une infrastructure. La technologie Entreprise Service Bus (ESB) est un choix plausible pour réaliser une telle architecture. L'ESB PEtALS[1] est un conteneur JBI distribué (Java Business Integration[38], le standard Java pour l'intégration de services). PEtALS est un unique bus de communication, qui est distribué et qui doit donc être déployé sur chaque machine où un composant participant à l'application est déployé. Des composants tels que des connecteurs SOAP [73], connecteurs JMS[67], ou un moteur de workflow y sont ensuite installés. Afin de se connecter aux applications existantes ou de définir une procédure d'orchestration, ces composants doivent être configurés à l'aide de *service assemblies* JBI, qui sont également déployés sur PEtALS.

La compagnie aérienne est donc exposée via un service web. Ainsi un nœud PEtALS doit être installé dans le domaine de la compagnie aérienne, avec un composant SOAP, configuré avec un service assembly JBI qui référence l'URL du Web Service.

[1] Disponible librement sous Licence LGPL à l'adresse `http://petals.ow2.org`

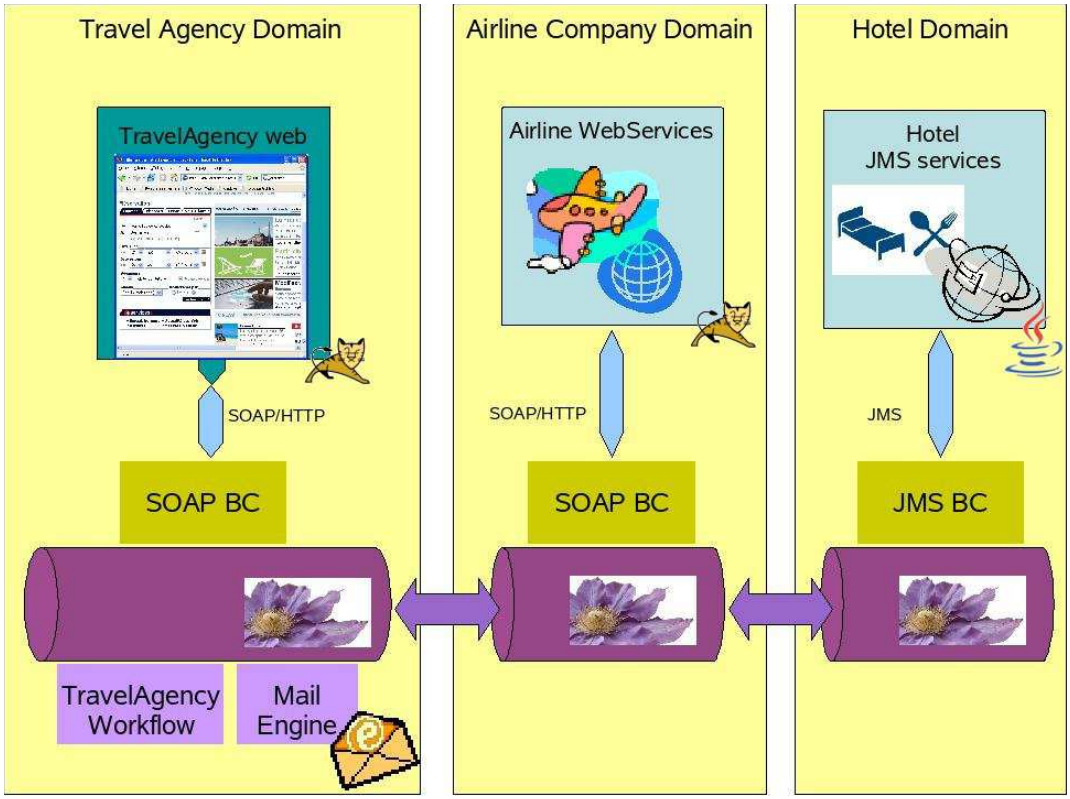

FIG. 7.1 – Architecture du scénario de l'agence de voyages

Les services du groupe hôtelier sont quant à eux accédés via le protocole JMS. Ainsi un nœud PEtALS doit être installé dans le domaine de la chaine hôtelière, configuré avec un autre service assembly JBI qui référence la destination JMS utilisée pour échanger de l'information avec le groupe hôtelier.

Finalement, l'agence de voyages héberge une application Web destinée à l'utilisateur final, qui orchestre le processus global de réservation. Un troisième nœud PEtALS est donc installé dans le domaine de l'agence de voyages. Ce nœud est connecté à l'application Web à l'aide d'un autre composant SOAP configuré. Il héberge également un moteur de workflow et un serveur de mail. Les trois nœuds PEtALS sont connectés les uns aux autres, assurant la communication entre chaque application. Ainsi, chaque service hebergé sur un container particulier est accessible aux autres de manière transparente.

Voyons maintenant le processus global de réservation. Un utilisateur se connecte sur l'application Web de l'agence de voyages. Il choisit une date et une destination, saisit des données personnelles, comme son adresse email par exemple. Puis, il soumet sa réservation et du point de vue utilisateur, le processus s'arrête là. L'application effectue une requête SOAP (contenant toutes les informations de l'utilisateur) au composant SOAP PEtALS. Comme ce composant est configuré pour envoyer des requêtes de l'application web vers le moteur de workflow, ce moteur est appelé et démarre le processus de réservation. Ce processus consiste en l'extraction de la requête des informations pertinentes pour la réservation de vol, puis invoque le service JBI de la compagnie aérienne, en utilisant PEtALS. Le deuxième composant SOAP (qui est connecté au Web service de la companie aérienne) reçoit la requête. Il effectue une reqûete SOAP vers le service web de la compagnie aérienne, récupère la réponse, et envoie le résultat en retour au moteur de workflow, toujours en utilisant PEtALS. Le moteur de workflow effectue le même traitement pour la réservation de chambre d'hôtel en envoyant une requête JBI au composant JMS. Ce composant publie une requête JMS au serveur JMS de la chaine hôtelière puis récupère la réponse. Il envoie la réponse au moteur de workflow, en utilisant JBI. Le moteur de workflow calcule les deux réponses afin d'extraire l'information de confirmation sous forme d'identifiant de réservation. Il produit un message textuel qui est envoyé via JBI au serveur de mail, en utilisant l'adresse fournie plus tôt par l'utilisateur. Pour compléter le processus, le serveur de mail qui reçoit la reqûete envoie le mail à l'utilisateur.

Le code source de ce scénario est disponible librement à l'adresse `http://svn.forge.objectweb.org/cgi-bin/viewcvs.cgi/petals/tags/petals-travelagency-*-1.2` et le lecteur peut se rapporter à [46] pour davantage de détails sur la manière d'implémenter des composants de service JBI ainsi que des service assemblies. L'ensemble des logiciels à installer ainsi que les dépendances entre ces logiciels est représentée sur la figure 7.2. Ce graphe nous fait prendre conscience de la complexité de l'orchestration d'un tel exemple, ainsi que de la multitude de technologies à déployer. Dans la section suivante, nous voyons comment prendre en charge le déploiement d'un tel système avec DeployWare.

7.1.2 Agence de voyages avec DeployWare

Dans cette section, nous illustrons concrètement comment utiliser DeployWare pour déployer un système logiciel respectant le paradigme d'architectures orientées service, comme notre agence de voyages. À l'aide du langage de la machine virtuelle DeployWare décrit dans

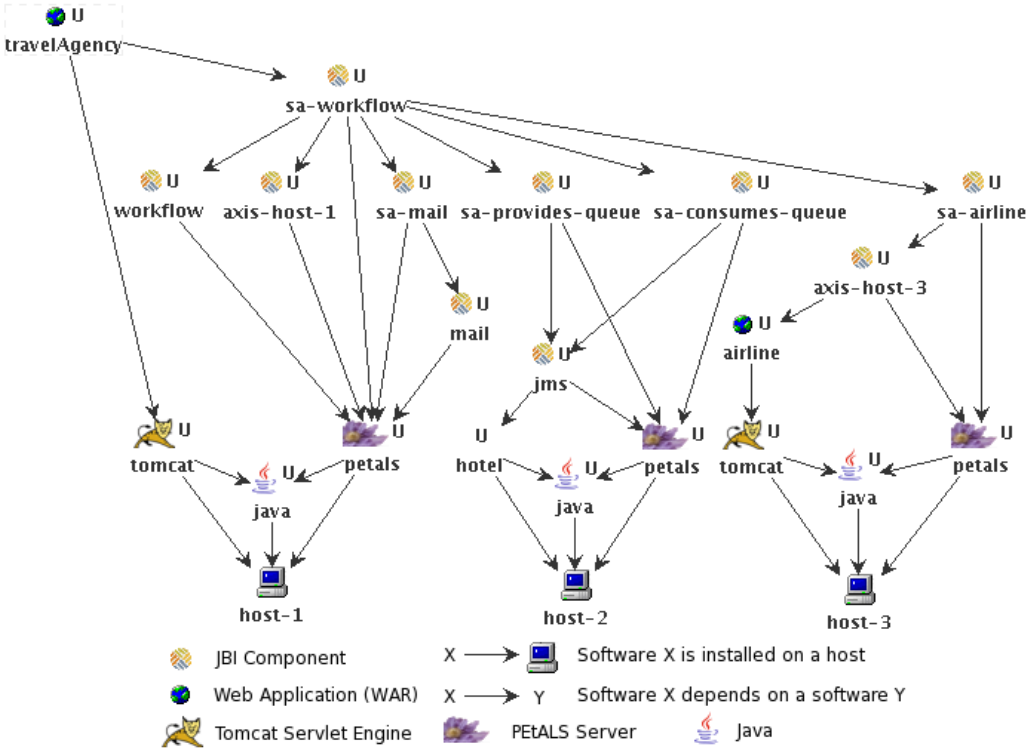

FIG. 7.2 – Graphe de dépendances entre les logiciels impliqués dans l'agence de voyage

le chapitre précédent, nous allons présenter les personnalités à décrire par l'expert logiciel et le système global décrit par l'administrateur système.

7.1.2.1 Conception des personnalités DeployWare

Avant de laisser l'administrateur système décrire son système logiciel global, nous allons nous pencher sur la définition de personnalités par les experts logiciels concernés.

```
JAVA.JRE = software.Installable
{
  archive = JAVA.ARCHIVE(UNDEFINED);
  home    = JAVA.HOME(UNDEFINED);

  internal-deployment {
    start {
      SHELL.SetVariable(JAVA_HOME,#[home]);
      SHELL.SetVariable(PATH,%PATH%:%JAVA_HOME%/bin);
    }
    stop {
      SHELL.RemoveFromVariable(PATH,%JAVA_HOME%/bin);
      SHELL.UnsetVariable(JAVA_HOME);
    }
  }
}
```

Listing 7.1 – Type de logiciel DeployWare pour Java

Le listing 7.1 déclare la personnalité DeployWare pour le Java Runtime Environment (JRE). Ce logiciel hérite de `software.Installable`, le type abstrait de logiciel que nous avons évoqué dans le chapitre 4, qui capitalise les procédures communes à tout logiciel à installer sur une machine hôte. Les valeurs des propriétés `archive` et `home` sont ensuite fixées par l'administrateur système comme nous le verrons dans le section suivante. Ensuite, dans le type de logiciel `JRE` est défini le corps des procédures `start` (l. 6–9) et de sa procédure inverse `stop` (l. 10–13), qui représentent les instructions à exécuter séquentiellement lorsque l'on démarre ou stoppe le logiciel. Ici, le démarrage (*resp.* l'arrêt) consiste à positionner (*resp.* supprimer) les variables du shell de la machine hôte.

```
JAVA.ApplicationServer = software.Installable,
  software.Hosting, JAVA.DependOn,
  software.Manageable(http://#[hostname]:#[http-port])
{
  properties {
    http-port = HTTP.PORT(0);
  }
}
```

Listing 7.2 – Type de logiciel DeployWare pour les serveurs d'applications à base de Java

Le listing 7.2 définit un type de logiciel abstrait pour n'importe quel type de serveur d'application reposant sur Java. Typiquement, un `JAVA.ApplicationServer` est un `software.Installable` (l. 1) hébergeant d'autres composants de logiciel (*e.g.*, WAR, JBI, EJB, OSGi bundles, etc.) (`software.Hosting` – l. 2) qui possède une dépendance vers un JRE (`JAVA.DependOn` – l. 2), et est administrable via une interface Web (`software.Manageable` – l. 3).

```
PEtALS.SERVER = JAVA.ApplicationServer
```

```
{
  archive = PEtALS.ARCHIVE(UNDEFINED);
  home = PEtALS.HOME(UNDEFINED);

  properties {
    http-port     = HTTP.PORT(7080);
    jmx-user      = JMX.USER(admin);
    jmx-password = JMX.PASSWORD(admin);
    jmx-port      = JMX.PORT(7700);
    # Other configurable properties.
  }
  internal-deployment {
    configure {
      TRANSFER.UploadGeneratedFile(PEtALS/server.properties,#[home]/conf);
    }
    start {
      SHELL.Fork(java -Xmx1024m -jar #[home]/bin/server.jar);
      PEtALS.Wait(#[hostname],#[jmx-port],#[jmx-user],#[jmx-password]);
    }
    stop {
      SHELL.Execute(java -Xmx1024m -jar #[home]/bin/server.jar stop);
    }
  }
}
```

Listing 7.3 – Type de logiciel représentant un serveur PEtALS

Le listing 7.3 définit le type de logiciel pour le conteneur JBI PEtALS. Après avoir installé le serveur mais avant de le démarrer, la procédure `configure` est appelée pour générer puis télécharger le fichier de configuration du serveur (l. 15) afin entre autres de fixer les paramètres des protocoles HTTP et JMX nécessaire à l'administration à distance du serveur PEtALS. La procédure `start` consiste à lancer le processus du serveur sur la machine hôte choisie (l. 18) et attendre que le serveur soit prêt à recevoir des requêtes (l. 19). La procédure `stop` tue le processus lancé dans le `start` (l. 22). On justifie le nombre différent d'instruction des procédures `start` et `stop`, par le fait que le type d'instruction `PEtALS.Wait` ne possède pas d'inverse : en effet ce type d'instruction possède une action limitée dans le temps, elle s'annule donc d'elle-même.

```
TOMCAT.SERVER = JAVA.ApplicationServer
{
  archive = TOMCAT.ARCHIVE(UNDEFINED);
  home = TOMCAT.HOME(UNDEFINED);

  properties {
    http-port     = HTTP.PORT(8080);
    http-user     = HTTP.USER(admin);
    http-password = HTTP.PASSWORD(admin);
  }
  internal-deployment {
    configure {
      SHELL.SetVariable(CATALINA_HOME,#[home]);
      TRANSFER.GenerateThenUpload(TOMCAT/server.xml,#[home]/conf);
      TRANSFER.GenerateThenUpload(TOMCAT/tomcat-users.xml,#[home]/conf);
    }
    start {
      SHELL.Fork(%CATALINA_HOME%/bin/startup.sh);
      HTTP.Wait(http://#[hostname]:#[http-port]/manager/html,#[http-user],#[http-
          password]);
    }
```

```
21      stop {
          SHELL.Execute(%CATALINA_HOME%/bin/shutdown.sh);
23      }
    }
25  }
```

Listing 7.4 – Type de logiciel DeployWare pour le conteneur de Servlet Tomcat

Le listing 7.4 définit le type de logiciel DeployWare pour le moteur de servlet Tomcat. Ce type de logiciel est structurellement similaire à `PEtALS.SERVER` mais encapsule des propriétés et des procédures d'administration propres à Tomcat comme les scripts `startup` et `shutdown`. Les procédures de démarrage (*resp.* arrêt) sont donc relativement similaires.

```
1   TOMCAT.WAR = software.Hosted(tomcat),
      software.Manageable(http://#[hostname]:#[http-port]/#[name])
3   {
      archive  = TOMCAT.WAR.ARCHIVE(UNDEFINED);
5     name     = TOMCAT.WAR.NAME(UNDEFINED);
      internal-deployment {
7       install {
          TRANSFER.Upload(#[archive],#[home]/#[archive-filename]);
9         TOMCAT.Manage(deploy,war=file://#[home]/#[archive-filename]);
          SHELL.RemoveFile(#[home]/#[archive-filename]);
11      }
        start {
13        TOMCAT.Manage(start);
        }
15      stop {
          TOMCAT.Manage(stop);
17      }
        uninstall {
19        TOMCAT.Manage(undeploy);
        }
21    }
    }
23
    TOMCAT.Manage(command, argument=) =
25  HTTP.GetURL(http://#[hostname]:#[http-port]/manager/${command}?
    path=/#[name]&amp;${argument},#[http-user],#[http-password])
27  {}
```

Listing 7.5 – Type de logiciel pour une application Web pour Tomcat

Le listing 7.5 représente le type de logiciel DeployWare pour une application Web hébergé sur un moteur Tomcat. Chaque Moteur Tomcat fournit une servlet web pour le déploiement d'autres applications Web. Ainsi, le type de logiciel `TOMCAT.WAR` ne fait qu'encapsuler les appels à cette servlet. Cela est effectué grâce à l'instruction `TOMCAT.Manage` (l. 13, 16, 19) définie dans le but d'envoyer des commandes à la servlet via le protocol HTTP.

7.1.2.2 Définition du système logiciel

Pour décrire le scénario de déploiement de l'agence de voyages, l'administrateur système doit décrire l'ensemble des machines hôtes du domaine de déploiement, puis l'ensemble des logiciels devant être déployés, leurs propriétés, ainsi que leurs dépendances. Les trois listings suivants ne se focalisent que sur la partie déploiement du domaine de l'agence de voyages, représenté sur la figure 7.1.

```
1 host-1 = INTERNET.HOST {
    hostname = INTERNET.HOSTNAME(host-1.travel-agency.com);
3   user     = INTERNET.USER(login,...);
    transfer = TRANSFER.SCP;
5   protocol = PROTOCOL.OpenSSH;
    shell    = SHELL.SH;
7   }
  }
```

Listing 7.6 – Description d'une machine hôte du domaine

Le listing 7.6 illustre la déclaration d'une machine hôte par l'administrateur système. `Host-1` est une instance du type d'hôte `INTERNET.HOST` de nom d'hôte `host-1.travel-agency.com` (l. 2) accessible via le protocole SSH (l. 5) à l'aide du login `login` (l. 3) pour le shell SH (l. 6), et accessible en transfert à l'aide du protocole SCP (l. 4).

```
  java-1 = JAVA.JRE {
2   archive = JAVA.ARCHIVE(archives/jre-1.5.0.tgz);
    home = JAVA.HOME(/home/jre-1.5.0) ;
4   host = /host-1;
  }
6 petals-1 = PEtALS.SERVER {
    archive = PEtALS.ARCHIVE(archives/petals-standalone-1.2.1.zip);
8   home    = PEtALS.HOME(/home/petals-1.2);
    host    = /host-1;
10  dependencies {
      java = /java-1 ;
12  }
  }
14 tomcat-1 = TOMCAT.SERVER {
    archive = TOMCAT.ARCHIVE(archives/apache-tomcat-5.5.23.zip);
16  home    = TOMCAT.HOME(/home/apache-tomcat-5.5.23);
    host    = /host-1;
18  dependencies {
      java = /java-1 ;
20  }
  }
```

Listing 7.7 – Description d'instances de JVM et de serveurs PEtALS et Tomcat

Le listing 7.7 déclare à la fois une machine virtuelle Java (l. 1–5), un serveur JBI PEtALS (l. 6–13), et un serveur de servlet Tomcat (l. 14–21). Étant tous trois des logiciels installables, les propriétés `home` et `archive` sont positionnées, ainsi qu'une référence sur la machine hôte sur lesquelles ces logiciels doivent être installés.

Dans notre cas, la JVM et les deux serveurs seront déployés sur la même machine, `host-1` décrite plus haut.

```
workflow = PEtALS.JBI {
  archive = PEtALS.JBI.ARCHIVE(archives/workflow-engine.zip);
  petals  = /petals-1;
}
axis-host-1 = PEtALS.JBI {
  archive = PEtALS.JBI.ARCHIVE(archives/petals-bc-axis2-1.2.zip);
  petals  = /petals-1;
}
mail = PEtALS.JBI {
  archive = PEtALS.JBI.ARCHIVE(archives/petals-bc-mail-1.2.zip);
  petals  = /petals-1;
```

```
}
sa-mail = PEtALS.SA {
  archive = PEtALS.JBI.ARCHIVE(archives/mail-service-assembly.zip);
  petals  = /petals-1;
  dependencies {
    /mail;
  }
}
sa-workflow = PEtALS.SA {
  archive = PEtALS.SA.ARCHIVE(archives/workflow-service-assembly.zip);
  petals  = /petals-1;
  dependencies {
    /workflow;
    /axis-host-1;
    /sa-mail;
  }
}
travelAgency = TOMCAT.WAR {
  archive = TOMCAT.WAR.ARCHIVE(archives/travelAgency.war);
  name    = TOMCAT.WAR.NAME(TravelAgency);
  tomcat  = /tomcat-1;
  dependencies {
    /sa-workflow;
  }
}
```

Listing 7.8 – Description de différentes instances de logiciels impliqués dans l'agence de voyages

Enfin, le listing 7.8 déclare tous les composants métiers de l'application. L'instance `workflow` représente le déploiement du service JBI d'orchestration. L'instance `axis-host-1` représente le *binding component* JBI PEtALS implémenté avec Axis. L'instance `mail` représente le JBI PEtALS du serveur de messagerie. L'instance `sa-mail` représente le *service assembly* JBI du serveur mail. L'instance `sa-workflow` représente le *service assembly* pour le moteur d'orchestration. L'instance `travelAgency` représente enfin l'application Web finale pour l'utilisateur. Les composants JBI et les *service assemblies* associés doivent être déployées sur le même serveur PEtALS, lequel est fixé dans la définition des composants concernés. De plus les dépendances métiers entre composants sont fixées par l'administrateur système afin de guider la machine virtuelle sur l'ordre d'exécution de déploiement.

Une fois que l'administrateur a complètement décrit son système à déployer, il est capable de charger sa définition dans la console graphique DeployWare Explorer. La console graphique est représentée sur la figure 7.3.

7.1.3 Conclusion

Ce premier cas d'étude permet de mettre en exergue l'apport de DeployWare pour décrire le déploiement d'un système distribué fortement hétérogène à base de services. Le langage DeployWare (et à plus forte raison le méta-modèle) permet d'exprimer l'essentiel des informations concernant le déploiement de différentes technologies. En effet, via cet exemple, ce langage apparait très peu verbeux, et se concentre sur les parties réellement utiles, aussi bien pour l'expert logiciel que pour l'administrateur réseau. La syntaxe reste assez facile à prendre

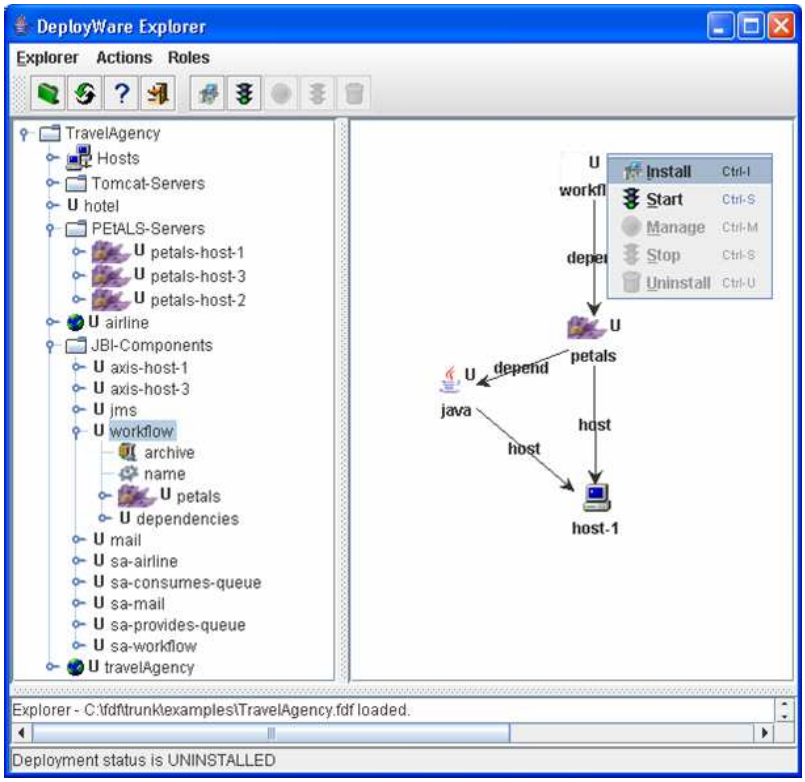

FIG. 7.3 – Console graphique DeployWare Explorer

en main, une fois les quelques types d'instructions et de procédures maîtrisés. En outre, le descripteur DeployWare pour déployer l'agence de voyages peut être très facilement adapté en cas de changement, qu'il soit minime ou important, dans l'architecture. Par exemple, en utilisant des scripts pour déployer ce même exemple, une modification sur un logiciel peut en entrainer plein d'autres en cascade. Enfin, l'orchestration globale du déploiement du système est effectuée de manière automatique par la machine virtuelle. En effet, l'expert logiciel déclare les dépendances pour chaque logiciel, l'administrateur système résoud les dépendances pour chacune de ses instances de logiciel. En revanche, l'administrateur système ne doit à aucun moment définir l'orchestration globale, laquelle est calculée automatiquement en fonction des dépendances de logiciel.

À ce jour, aucune approche ne permet de décrire l'architecture d'un système distribué complexe et hétérogène, et également d'automatiser l'exécution de ce déploiement (*i.e.* calcul de l'ordre de déploiement, calcul du format des commandes à envoyer, etc.). De plus la syntaxe pour décrire un tel système reste très succinte et à la portée de n'importe quel administrateur système.

Au delà des avantages de description proposés par DeployWare, ce cas d'étude ouvre de nouvelles perspectives, ou de nouvelles difficultés à surmonter. Cet exemple illustre notamment le déploiement d'une architecture orientée services complète reposant sur des infrastructures très diverses en termes de technologies. Or, le principe essentiel des services web est de faire communiquer les systèmes d'informations d'entreprises différentes. Les services web d'une entreprise donnée sont donc déployés dans des réseaux différents administrés par des personnes différentes. La faisabilité d'un tel déploiement d'architecture orientée services de manière globale parait donc compromise. Ce constat nous amène notamment à considérer une perspective de travail discutée dans le chapitre 8 : le déploiement incrémental de systèmes logiciels inter-domaines. En outre dans le cadre d'une architecture inter-entreprise, les problèmes de sécurité liés aux pares-feux notamment, font que les communications entre logiciels ne peuvent se faire qu'en utilisant le protocole HTTP et son port associé (le port 80). Il est ainsi difficilement envisageable de disposer d'un accès à distance (SSH, Telnet ou autre) dans chacun des domaines. Cela représente là encore un frein pour un déploiement global d'architecture orientée services.

En outre ces expériences ont montré que certaines erreurs, non détectées statiquement, persistent à empécher le déploiement de s'effectuer correctement. Ces erreurs, parfois minimes comme l'absence d'une archive de logiciel à l'endroit indiqué, font parfois échouer un grand déploiement à un stade avancé. Le système se retrouve donc partiellement déployé sur un sous-ensemble des machines du domaine. Cela nous amène à deux remarques. La première est qu'il est raisonnable d'envisager d'intégrer les vérifications d'existance des fichiers d'archive (ou autres chemins) des logiciels installables. Cette vérification n'est pas faisable uniquement à partir de la structure du méta-modèle, mais reste néanmoins vérifiable statiquement. La seconde remarque est que le déploiement d'un système logiciel doit être atomique. Cela signifie que soit l'intégralité du système est déployé, soit rien n'est déployé. Il pourrait être judicieux de s'inspirer par exemple des mécanismes mis en place dans GoDIET. Ainsi en cas d'erreur à n'importe quel moment du déploiement, il serait possible de défaire toutes les actions effectuées jusque là.

En conclusion cet exemple illustre parfaitement l'aisance de mise en œuvre du déploiement d'un système logiciel multi-technologies avec DeployWare, mais certaines améliorations

doivent être apportées à DeployWare pour pouvoir parfaitement gérer tous les paramètres du déploiement d'architectures orientées service inter-entreprise.

7.2 Gare du futur

7.2.1 Scénario

Le second exemple que nous étudions illustre l'utilisation conjointe de DeployWare et DACAR, dans le contexte d'un environnement ouvert distribué. Ce cas d'étude, ainsi que l'utilisation de DACAR pour résoudre le problème, ont été développés dans le cadre du projet CPER MOSAIQUES [72][29].

Dans une gare ferroviaire, on décide de mettre en œuvre un logiciel capable de fournir les informations concernant les horaires de départ ou d'arrivées des trains dans la gare. Ce logiciel pourra également fournir d'autres fonctionnalités, comme la possibilité de réserver un billet de train directement dans la gare sans faire la queue au guichet. L'ambition de cette application étant de libérer l'utilisateur du guichet, on souhaite que ces fonctionnalités soient disponibles directement sur le PDA ou le *smartphone* des personnes entrant dans la gare. Les composants pour réaliser cette application existent déjà, et l'architecture de l'application est représentée sur la figure 7.4. Cette architecture peut être scindée en deux parties que nous appelons dans la suite la *partie fixe* de l'architecture, et la *partie mobile*.

La partie fixe regroupe l'ensemble des composants capables de fournir le service. Ces composants doivent être déployés sur une machine hôte devant être fixe —*i.e.* qui ne disparait à aucun moment du domaine de déploiement. Deux types de composants sont nécessaires pour réaliser cette partie fixe de l'architecture. Le composant `DatabaseTrainSchedule` est un composant qui englobe une base de données, et qui expose un port fourni contenant les opérations d'accès en lecture aux données de cette base. Le composant `TrainStation` est le composant qui représente les fonctionnalités offertes aux utilisateurs de la gare. Il est donc connecté au port *data* du composant de base de données, afin d'avoir accès aux données ferroviaires qu'elle contient. Ce composant fournit également un port *the_service* qui expose les services offerts aux utilisateurs de la gare.

La partie mobile de l'application regroupe l'ensemble des composants capables d'offrir une interface graphique à l'utilisateur, sur son terminal mobile, afin d'accéder aux données, et éventuellement d'effectuer des actions (*e.g.* réservation/échange/annulation de billets). Cette partie est dite mobile car elle doit être déployée autant de fois qu'il y a de terminaux mobiles dans le domaine de déploiement. Il n'est donc pas possible statiquement de connaitre le plan de déploiement exact de la partie mobile, ni même de le prévoir. Dans notre cas, cette partie mobile de l'architecture contient un composant. Ce composant `TrainGUI` est un composant qui contient une interface graphique permettant à l'utilisateur de 1) visualiser les informations courantes sur les trains et 2) interagir avec la partie fixe pour obtenir des informations plus précises, ou réserver un billet etc.. Ce composant possède un port requis *the_service* devant être connecté au composant de service (*i.e.* `TrainStation`) de la gare.

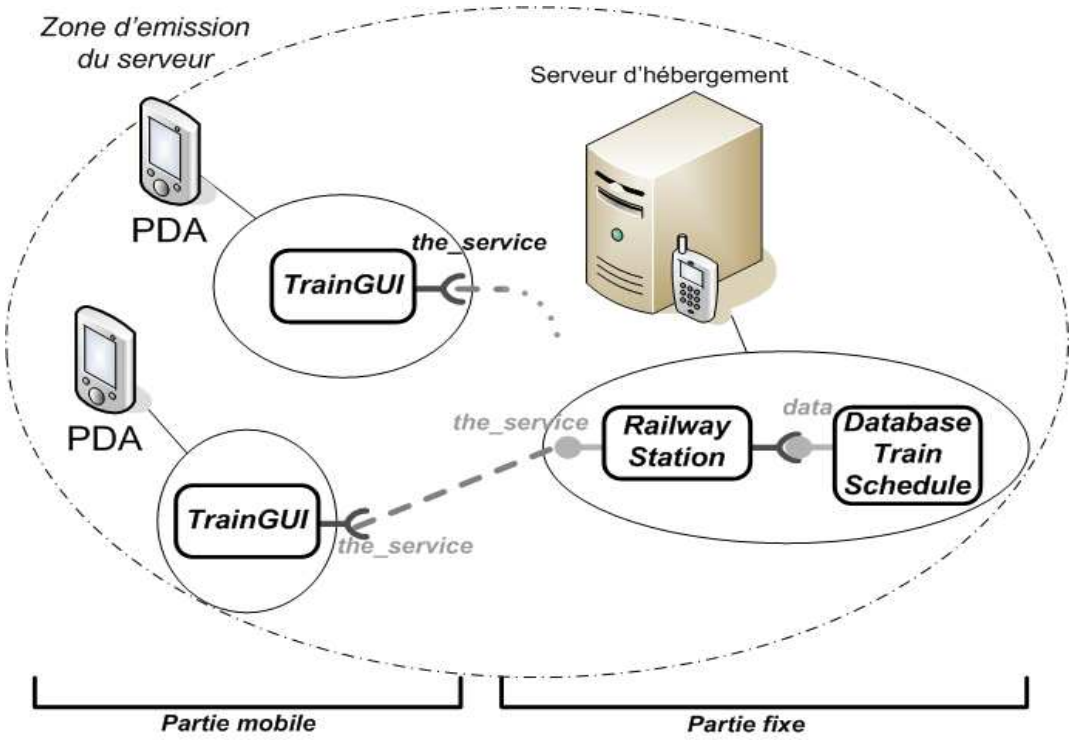

FIG. 7.4 – Architecture globale de l'application de la gare du futur

Nous souhaitons mettre en œuvre cette application en utilisant le modèle de composants CCM, et plus particulièrement l'implémentation libre en Java de ce modèle réalisée dans notre équipe, OpenCCM. Pour mener à bien ce déploiement, il faut tout d'abord mettre en œuvre le déploiement de la partie fixe de l'application qui comprend 1) la couche intergicielle composée des différents logiciels nécessaires pour exécuter une application OpenCCM, 2) l'architecture métier en terme de composants OpenCCM et leurs interactions. Ensuite il faut mettre en œuvre le déploiement de la partie mobile. Ce déploiement se faisant dans un environnement ouvert distribué, un certain nombre de politiques d'auto-gestion vont devoir être écrites, aussi bien pour la couche intergicielle que pour l'architecture métier. En effet, aucun prérequis ne peut-être supposé sur les terminaux entrant dans le domaine de déploiement. Ainsi le déploiement dynamique de la couche intergicielle, comme celui des composants métiers doit être effectué sur ces éventuels terminaux.

7.2.2 Gare du futur avec DeployWare/DACAR

7.2.2.1 Conception des personnalités DeployWare

Tout d'abord, nous allons traiter ce scénario du point de vue de la couche intergicielle. Cela revient à définir dans un premier temps quels types de logiciels doivent être conçus par l'expert logiciel. Dans un second temps, il convient de définir l'architecture de la couche intergicielle à l'aide du langage de l'administrateur système.

Tout d'abord, le canevas OpenCCM est réalisé en Java, donc pour l'utiliser, il est nécessaire de disposer de la personnalité Java. Cette personnalité ayant été écrite dans la section 7.1, on peut raisonnablement considérer qu'elle fait maintenant partie de la bibliothèque de base DeployWare. Nous ne la détaillerons donc pas et la réutiliserons directement.

La personnalité OpenCCM comporte un certain nombre de logiciels, correspondant tous à une fonctionnalité du canevas. Le premier type de logiciel que nous allons étudier et dont nous allons réaliser la description DeployWare est l'*Object Request Broker* (ORB). En effet, n'importe quel produit CORBA repose sur un bus de communication IIOP, lequel est implémenté dans un ORB. Le type de logiciel représentant un ORB donné (n'importe lequel en fait) est représenté sur le listing 7.9. Le processus de déploiement d'un ORB est assez similaire à celui d'une JVM, vu dans la section précédente. Ce processus consiste essentiellement à charger une archive sur la machine hôte, et à fixer la variable d'environnement pointant sur le repertoire des bibliothèques de l'ORB. Ainsi le type de logiciel `CORBA.ORB` hérite de la définition `Installable` et fournit des procédures pour positionner (*resp.* supprimer) la variable d'environnement correspondante.

```
CORBA.ORB = software.Installable
{
  orb {
    archive = CORBA.ARCHIVE(UNDEFINED);
    home    = CORBA.HOME(UNDEFINED);
    internal-deployment{
      start {
        SHELL.SetVariable(ORB_HOMEDIR,#[home]);
      }
      stop {
        SHELL.UnsetVariable(JAVA_HOME);
      }
```

```
    }
  }
}
```

Listing 7.9 – Description en langage DeployWare du type de logiciel pour un ORB

Le type de logiciel OpenCCM est quant à lui représenté sur le listing 7.10. Le processus de déploiement du logiciel OpenCCM consiste à charger l'archive, fixer les variables d'environnement `OPENCCM_HOMEDIR` et `OPENCCM_CONFIGDIR` nécessaires au fonctionnement du logiciel[2]. Il faut ensuite ajouter, dans la variable d'environnement `PATH`, le répertoire des binaires d'OpenCCM. Toutes ces opérations sont définies dans la procédure `install` (l. 17–21). Comme ce logiciel hérite du type `Installable`, les positionnements de ces trois variables sont ajoutés à la suite de celles définies dans le type abstrait. La procédure de démarrage (l. 22–24) consiste à lancer l'exécutable `ccm_install` présent dans le répertoire des binaires d'OpenCCM. Les procédures d'arrêt (l. 25–27) et de désinstallation (l. 28–32) effectuent les instructions inverses des procédures de démarrage et d'installation.

```
1  OpenCCM.OpenCCMSoftware
   = software.Installable, JAVA.DependOn(openccm), ORB.DependOn(openccm)
3  {
     openccm {
5      home = OpenCCM.HOME(UNDEFINED);
       archive = OpenCCM.ARCHIVE(UNDEFINED);
7
       internal-deployment {
9        install {
           set-openccm_home = SHELL.SetVariable(OpenCCM_HOMEDIR,#[home]);
11         set-openccm_config_dir = SHELL.SetVariable(OpenCCM_CONFIG_DIR,#[home]/
               OpenCCM_CONFIG_DIR);
           set-path = SHELL.AddPath(#[home]/bin);
13       }
         start {
15         openccm-start = SHELL.Execute(ccm_install);
         }
17       stop {
           openccm-stop = SHELL.Execute(ccm_deinstall);
19       }
         uninstall {
21         unset-path = SHELL.RemovePath(#[home]/bin);
           unset-openccm_config_dir = SHELL.UnsetVariable(OpenCCM_CONFIG_DIR);
23         unset-openccm_home = SHELL.UnsetVariable(OpenCCM_HOMEDIR);
         }
25     }
     }
27 }
```

Listing 7.10 – Type de logiciel DeployWare pour OpenCCM

Pour déployer un serveur de composant OpenCCM, après avoir installé le canevas, il existe plusieurs logiciels à démarrer. Ces logiciels, décrits dans la suite, ne sont pas à installer car ils sont en fait fournis par le canevas. Néanmoins nous définissons un type de logiciel DeployWare pour chacune de ces fonctionnalités, afin de pouvoir déléguer la gestion de l'orchestration entre ces tâches à la machine virtuelle DeployWare. Dans les listings 7.11, 7.12 et 7.13 sont représentés respectivement les types de logiciels pour démarrer le service de nommage

[2]Pour davantage de détails sur le fonctionnement d'OpenCCM, le lecteur se réfèrera à [30].

d'OpenCCM, démarrer l'infrastructure de déploiement réparti OpenCCM *Distributed Computing Infrastructure* [14], et enfin démarrer un serveur de composant OpenCCM. Le type de logiciel `OpenCCM.NS` dépend du type de logiciel `OpenCCM.OpenCCMSoftware`. Le type de logiciel `OpenCCM.DCI` dépend lui des types de logiciel `OpenCCM.OpenCCMSoftware` et de `OpenCCM.NS`. Enfin le type de logiciel `OpenCCM.NM` dépend du type de logiciel `OpenCCM.DCI`.

```
OpenCCM.NS = OpenCCM.DependOn(ns)
{
  ns {
    internal-deployment {
      start {
        ns-start = SHELL.Execute(ns_start #{ns_args});
      }
      stop {
        ns-stop = SHELL.Execute(ns_stop);
      }
    }

    internal-parameters {
      ns_args = OpenCCM.TEXTUALPARAM( );
      ns_ior = CORBA.IOR(No IOR Defined);
    }
  }
}
```

Listing 7.11 – Type de logiciel DeployWare pour le NameService OpenCCM

```
OpenCCM.DCI = NS.DependOn(dci)
{
  dci {
    internal-deployment {
      start {
        set_ns_ior = SHELL.Execute(ccm_set_ior NameService #[ns_ior]);
        dci-start = SHELL.Execute(dci_start #{dci_args} #{dci_name});
      }
      stop {
        dci-stop = SHELL.Execute(dci_stop #{dci_name});
      }
    }

    internal-parameters {
      dci_name = OpenCCM.TEXTUALPARAM(DefaultDCIName) ;
      dci_args = OpenCCM.TEXTUALPARAM( ) ;
      dci_ior = CORBA.IOR (No IOR defined) ;
      ns_ior = ./dci/dependencies/ns/internal-parameters/ior_ns ;
    }
  }
}
```

Listing 7.12 – Type de logiciel DeployWare pour OpenCCM DCI

```
OpenCCM.NM = DCI.DependOn(nm) {
  nm {
    internal-deployment {

      start {
        set_ns_ior = SHELL.Execute(ccm_set_ior NameService #[ns_ior]);
        set_dci_ior = SHELL.Execute(ccm_set_ior DCI #[dci_ior]);
```

```
        nm-start = SHELL.Execute(node_start #{node_args} #{node_name});
      }
      stop {
        nm-stop = SHELL.Execute(node_stop #{node_name});
      }
    }
    dependencies {
      dci = OpenCCM.DCI ;
      ns = OpenCCM.NS ;
    }

    internal-parameters {
        node_name = OpenCCM.TEXTUALPARAM(DefaultName);
        node_args = OpenCCM.TEXTUALPARAM( );
        ns_ior = ./nm/dependencies/ns/internal-parameters/ior_ns ;
        dci_ior = ./nm/dependencies/dci/internal-parameters/ior_dci ;
    }
  }
}
```

Listing 7.13 – Type de logiciel DeployWare pour le serveur de composants OpenCCM

7.2.2.2 Architecture de la couche intergicielle

Une fois les différents types de logiciel définis, nous nous intéressons maintenant au travail de l'administrateur système dans sa description du déploiement de la couche intergicielle aussi bien fixe que mobile. L'architecture de la couche intergicielle est schématisée sur la figure 7.5.

La partie fixe de la couche intergicielle est déployée sur une machine `m1`. Cette partie est composée d'une instance de logiciel de type `JAVA.JRE` appelée `Java-m1` et d'une instance de logiciel de type `CORBA.ORB` appelée `ORB-1`. Ensuite une instance de logiciel de type `OpenCCM.OpenCCMSoftware` appelée `CCM-1` est créée. Cette instance dépend de `Java-m1` et de `ORB-1`. Ensuite une instance de logiciel de type OpenCCM.NS appelée `NS-1` est créée. Cette instance dépend de `CCM-1`.

Une instance de logiciel de type OpenCCM.DCI appelée `DCIManager1` est créée. Cette instance dépend de `NS-1` et de `CCM-1`.

Enfin, une instance de logiciel de type OpenCCM.NM est créée. Cette instance dépend de `DCIManager1`.

Concernant la partie mobile, il faut déployer une architecture quasi-similaire, l'instance de `OpenCCM.DCI` et le `OpenCCM.NS` en moins, puisque ces deux logiciels doivent être installés une seule fois pour toute l'architecture. Des instances de `JAVA.JRE`, `CORBA.ORB`, `OpenCCM.OpenCCMSoftware` et `OpenCCM.NM` (qui doit invariablement dépendre de l'instance `NS-1`) doivent être déployées sur chacun des terminaux mobiles entrant dans le réseau.

Pour réaliser ce déploiement dynamiquement, nous devons écrire une politique d'autogestion prenant en charge ce déploiement.

Le listing 7.14 représente la description dans le langage DeployWare du système à déployer.

```
GareDuFutur
{
  # Description des machines hôtes
  Hosts = INTERNET.NETWORK {
    host-domain = Autonomic.DOMAIN {
      m1 = INTERNET.HOST {
```

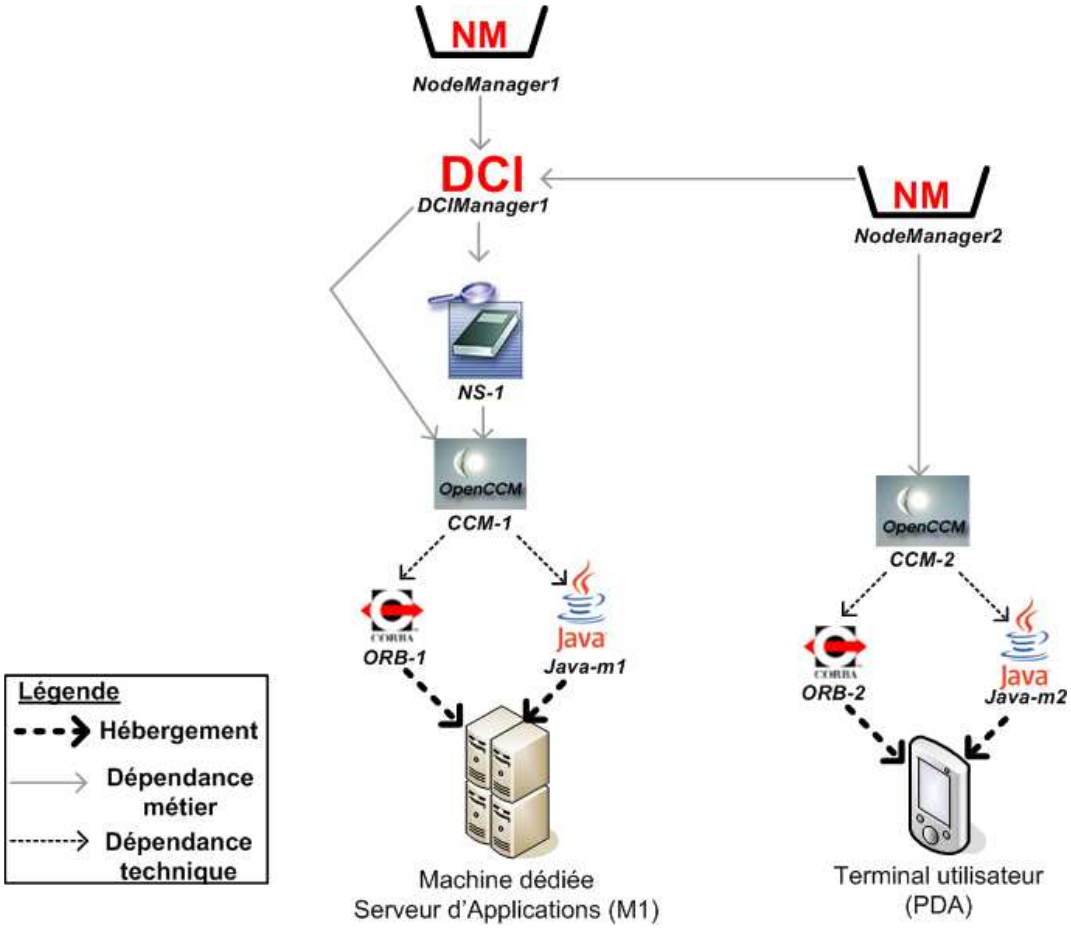

FIG. 7.5 – Architecture de la couche intergicielle de l'application de la gare du futur

```
        hostname = INTERNET.HOSTNAME(myhost.mycompany.com);
        user     = INTERNET.USER(root,,/cygdrive/c/cygwin.home/dubus/.ssh/id_rsa);
        transfer = TRANSFER.SCP;
        protocol = PROTOCOL.OpenSSH;
        shell    = SHELL.SH;
      }

      sensor = Autonomic.PDADetectorSensor ;
      actuator = Autonomic.DefaultDeployWareActuator ;
      internal-deployment {
        start {
          start-engine = Autonomic.JessRuleEngine(/home/dacar/TrainStationHostRule.
              clp) ;
        }
      }
    }
  }

  # Description of software, services and application servers

  Software {
    Java-m1 = JAVA.JRE {
      archive = JAVA.ARCHIVE(Archive_File_URI);
      home    = JAVA.HOME(Java_Home_URI);
      host = Hosts/host-domain/m1;
    }
    ORB-1 = CORBA.ORB {
        archive = CORBA.ARCHIVE(Archive_File_URI);
        home = CORBA.HOME(Corba_Home_URI);
        host = Hosts/host-domain/m1;
    }

    CCM-1 = OpenCCM.SERVER {
        archive = OpenCCM.ARCHIVE(Archive_File_URI);
        home = OpenCCM.HOME(OpenCCM_Home_URI);
        orb = Software/ORB-1;
        java = Software/Java-m1;
        host = Hosts/m1;
    }

    NS-1 = OpenCCM.NS {
        host = Hosts/m1;
        ns = CORBA.IOR(http://host1:8080/NameService.IOR);
        openccm = ./Software/CCM-1 ;
    }

    DCIManager1 = OpenCCM.DCI(DefaultDCI) {
        ns = OpenCCM_Deployment/NameService;
        host = Hosts/m1;
        dci_ior = OpenCCM.IOR(http://host1:8080/DCI.IOR);
        dci_name = OpenCCM.TEXTUALPARAM(DCIManager1);
    }
    NodeManager1 = OpenCCM.NM(DefaultDCI) {
        host = Hosts/m1;
  node_name = OpenCCM.TEXTUALPARAM(StationCentralNode) ;
        dci = ./Software/DCIManager1 ;
    }

    autonomic-domain-soft = Autonomic.MANAGER {
      sensor = Autonomic.DWHostDetectorSensor ;
      actuator = Autonomic.DefaultDeployWareActuator ;
      internal-deployment {
```

```
68          start {
              start-engine = Autonomic.JessRuleEngine(/home/dacar/
                  TrainStationSoftwareRule.clp) ;
70          }
          }
72      }
    }
74
    }
76  }
```

Listing 7.14 – Description du déploiement de la couche intergicielle pour l'exemple du train

Dans la description du déploiement de cette couche intergicielle, on remarque la définition de deux domaines d'autonomie. Le premier, dans `Hosts`, contient un senseur `PDADetectorSensor` capable de détecter l'entrée des terminaux dans le domaine de déploiement. Ce domaine contient également l'actuateur générique qui permet d'appeler dynamiquement les opérations définies dans l'API définie dans la section 5.3 du chapitre 5. Le fichier de définition de politiques `TrainStationHostRule.clp`, contient la politique suivante (en langage naturel pour simplifier l'expression et la compréhension) : *Pour chaque événement de type 'Nouveau terminal entrant', il faut appeler l'opération addNode avec les paramètres correspondants sur l'actuateur*. Cela assure la causalité entre le domaine de déploiement et la machine virtuelle.

Le second domaine, dans `Software`, contient un senseur `DWHostDetectorSensor` capable de détecter l'apparition de nouvelles définitions de machines hôtes à l'intérieur de la machine virtuelle DeployWare. Ce second domaine contient lui aussi l'actuateur générique. Le fichier de description de politiques `TrainStationSoftwareRule.clp` contient la politique suivante : *Pour chaque événement de type 'Nouveau host entrant' il faut appeler l'opération addSoftware() quatre fois pour déployer le JRE, l'ORB, OpenCCM et le NodeManager sur le host entrant*.

Ces politiques assurent le déploiement dynamique de la couche intergicielle quelle que soit le nombre de machines entrant sur le réseau.

7.2.2.3 Architecture métier

Maintenant que le déploiement dynamique de la couche intergicielle a été décrit par l'administrateur système, il reste à l'architecte métier de décrire son architecture. Cette tâche est finalement assez simple, compte tenu de la faible complexité de l'application. Ainsi l'administrateur doit décrire, à l'aide de langage OMG D&C, la partie fixe de l'architecture métier. Cette partie est assez simple à décrire puisqu'il s'agit uniquement d'instancier les composants `RailwayStation` et `DatabaseTrainSchedule` sur le nœud *StationCentralNode* (*i.e.* le nœud déployé à l'aide du logiciel `NM-1`). Pour décrire la partie mobile de l'architecture métier, l'architecte va écrire la simple politique d'auto-gestion dont le code du composant de politique est donné sur le listing 7.15

```
@Component
public class NewHostThenTrainGUI implements PolicyAPI {

  public String eventName = "deployware.events.NewHostDeclaredEvent";

 /**
```

```
   * Annotation Fraclet pour marquer le champ comme
   * étant une interface requise dont le nom est
   * \${name} dans le fichier ADL
   */
  @Requires(name="plan_description_action")
  protected PlanAPI plan_modif ;

  /**
   * Annotation Fraclet pour marquer le champ comme
   * étant une interface requise
   * \${name} dans le fichier ADL
   */
  @Requires(name="domain_description_action")
  protected DomainAPI domain_modif ;

  public boolean isTriggeredBy(Object o) {
    String eventClass = o.getClass().getName();
    return eventClass.equals(eventClassName) ;
  }

  public boolean checkCondition(Event e) {
    NewHostDeclaredEvent nhde = (NewHostDeclaredEvent)e ;
    return nhde.getType().equals("PDA") ;
  }

  public void doAction(Event e) {
    NewHostDeclaredEvent nhde = (NewHostDeclaredEvent)e ;
    HashMap param = new HashMap() ;
    param.add(...) ;
    String instance_name = nhde.name().toLowerCase()+"_trainGUI" ;
    this.plan_modif.addInstance(instance_name,"TrainGUIImpl",nhde.name(),param) ;
    this.plan_modif.addBinding("RailwayStation","the_service",instance_name,"
        the_service") ;
  }
}
```

Listing 7.15 – Politique architecturale pour le déploiement dynamique du composant `TrainGUI`

Cette simple politique architecturale, associée aux politiques de déploiement et d'observation existantes dans la bibliothèque de DACAR, permet d'assurer que pour chaque serveur de composant déployé sur chacun des PDA entrant dans la couche intergicielle, un composant de type `TrainGUI` est déployé et relié à au composant central de type `RailwayStation`.

7.2.3 Conclusion

Ce second cas d'étude permet quant à lui d'évaluer l'apport des politiques d'auto-gestion de DeployWare et de DACAR dans une application à base de composants à déployer en environnement ouvert ubiquitaire. À ce jour, aucune approche de déploiement d'architecture métier, quel que soit le paradigme employé, ne permet de prendre dynamiquement en compte l'entrée ou la sortie de terminaux mobiles. L'approche DeployWare permet, en plus de la description du déploiement de la couche intergicielle, de programmer des politiques d'auto-gestion pour étendre dynamiquement cette couche intergicielle aux nouvelles machines entrant sur le réseau. L'approche DACAR quant à elle permet d'effectuer la même démarche, mais au niveau de l'architecture métier. Le paradigme à base d'événements utilisé pour décrire les politiques

d'auto-gestion s'est avéré assez intuitif. Un ADL générique (en l'occurrence OMG D&C) est proposé pour décrire une éventuelle partie fixe dans l'architecture, et un langage de règles assez simple permet également de définir le déploiement de la partie mobile de l'application —*i.e.* les composants devant être déployés dynamiquement sur les machines qui entrent sur le réseau. Cet exemple met en exergue la grande complémentarité entre les approches DACAR et DeployWare, qui permet de décrire le déploiement de l'intégralité d'un système logiciel, tout en injectant de l'auto-gestion dans ce déploiement, afin de prendre en charge l'entrée de nouvelles machines dans le réseau.

Néanmoins, cette expérience a également mis en exergue certaines limitations pour ces deux approches et leur utilisation en environnement ouvert. La première concerne la réalisation des politiques d'auto-gestion dans DeployWare. En effet l'expression de ces politiques se fait de manière séparée par rapport à la description du système logiciel à déployer. La deuxième limitation concerne l'utilisation de DACAR et plus précisément de OMG D&C, son langage de description d'architecture associé. Ce langage s'est révélé très verbeux, et la description d'une architecture logicielle, aussi simple soit-elle, est très longue et donc rédhibitoire pour l'architecte métier. De plus comme pour DeployWare, le découplage entre description de l'architecture métier et des politiques d'auto-gestion s'avère peu pratique pour l'architecte métier. Une unification des différents éléments de description est à envisager fortement.

Cinquième partie

Conclusion

Chapitre 8

Conclusion et travaux futurs

Sommaire

8.1 Bilan sur l'approche proposée

Nous discutons ici des différents apports de cette thèse ainsi que les travaux à mener dans la suite.

Nous avons présenté DeployWare/DACAR, une approche basée sur l'ingénierie dirigée par les modèles, pour décrire et exécuter le déploiement de systèmes logiciels répartis. Le déploiement est pris en charge dans sa totalité, depuis l'installation de librairies sur les machines hôtes jusqu'à la configuration de déploiement de composants métiers relatifs à une technologie donnée sans oublier la renconfiguration dynamique et l'extension du déploiement sur les machines entrant dynamiquement sur le réseau, et ce sans aucun pré-requis sur les machines concernées. Le premier apport majeur de ce travail est d'avoir su unifier et abstraire une phase du cycle de vie du logiciel qui est une des plus techniques. En effet, une grande majorité des travaux qui s'adresse au déploiement logiciel se cantonne à la couche des composants logiciels. Le déploiement de la couche intergicielle sous-jacente doit être déployée manuellement, après avoir étudié les documentations des technologies concernées, ce qui représente une charge de travail conséquente et très sujette à erreurs, surtout dans des environnements comprenant un grand nombre de machines. L'expression du déploiement, aussi bien des composants logiciels que de leur support d'exécution et bibliothèques associées, à un haut niveau d'abstraction, permet d'exprimer plus clairement et de manière plus lisible et transparente la procédure de déploiement d'un système. L'utilisation des concepts de l'ingénierie dirigée par les modèles permet

non seulement de monter en abstraction lors de la description du déploiement, mais également de séparer les préoccupations au sein du processus de déploiement.

Un langage d'expression du déploiement purement descriptif, de haut niveau, est donc proposé dans cette thèse, tant pour décrire le déploiement de la couche intergicielle (méta-modèle DeployWare) que l'architecture à composants (partie modèle d'architecture de DACAR). Les concepts de ce méta-modèle sont séparés selon le rôle qu'ils concernent : l'expert logiciel décrit le protocole de déploiement d'un type de logiciel donné, l'administrateur système décrit l'architecture de la couche intergicielle de son système et enfin l'architecte métier décrit l'architecture de composants métiers qui composent son application. Ces paquetages reposent les uns sur les autres, de manière à emboiter les différents rôles entre eux dans le cadre du déploiement global. Le principal atout des différents paquetages de méta-modèle que nous proposons dans cette thèse repose sur la généricité des concepts employés. En effet, chaque rôle utilise un langage dépourvu de considérations techniques telles que la syntaxe des commandes à taper, la commande utilisée pour envoyer des fichiers, etc. Les différents concepts des méta-modèle DeployWare et DACAR ont été validés à l'aide d'exemples concrets issus d'applications du monde industriel. La modélisation de ces différents scénario a mis en exergue la généricité des méta-modèles et l'extensibilité de l'approche.

Le deuxième apport de l'utilisation de DeployWare/DACAR concerne la capacité de vérification. En effet, la méta-modélisation permet de définir la structure du langage de déploiement choisi. Nous avons exploité cette définition de structure pour y ajouter des concepts permettant de relier des concepts entre eux (*e.g.* les types d'instructions antagonistes). Ce supplément de méta-information attaché aux concepts du méta-modèle permet de vérifier certaines propriétés statiquement, comme la réversibilité, la complétude et la cohérence du déploiement. Ces vérifications, et le fait qu'elles soient statiques, renforce la pertinence de l'utilisation de modélisation pour décrire le déploiement. Les modèles ne servent pas uniquement à abstraire la description du déploiement, ils offrent un cadre de conception complet aidant les experts logiciels et administrateurs systèmes (dans nos travaux nous avons appliqué les vérifications pour ces deux rôles uniquement) à réaliser leur tâche de manière optimale, en fournissant des outils d'analyse. En outre, des vérifications telles que nous les mettons en œuvre rendent plus facile le déploiement dans de larges environnements, en minimisant les erreurs de conception, et évitant par là même de nombreuses itérations de déploiements dûes à des erreurs à répétition, coûteuses en temps dans ces environnements.

La gestion du déploiement en environnements ouverts distribués est également adressée dans notre approche. Cette gestion ne concerne que deux rôles : l'administrateur système et l'architecte métier. Des concepts inspirés de l'informatique auto-gérée ont été ajoutés à ce méta-modèle. Il est alors possible de spécifier des politiques d'auto-gestion exprimées à l'aide d'un paradigme naturel et facile à prendre en main, basé sur *événement-condition-action*. L'administrateur système peut étendre le déploiement de la couche intergicielle du système sur de nouvelles machines hôtes entrées dynamiquement dans le domaine de déploiement. L'architecte logiciel peut quant à lui, dans DACAR, étendre le déploiement de l'architecture logicielle à base de composants sur de nouvelles machines entrées dynamiquement. L'intégration de la prise en charge d'événements dynamiques rend possible la gestion des environnements ouverts

distribués.

Enfin, la machine virtuelle DeployWare proposée dans cette thèse offre une plate-forme vers laquelle il est possible de projeter les modèles DeployWare. Le format d'entrée de cette machine virtuelle est très simple et surtout très proche des concepts du méta-modèle DeployWare ce qui augmente la capacité de transformation automatique. Le point fort de cette plate-forme repose principalement sur son extensibilité extrême, rendue possible grâce à une utilisation intensive des composants à très fine granularité pour représenter chaque mécanisme de cette machine virtuelle. Cette machine virtuelle offre également les API pour que l'expert réseau développe intuitivement les composants adaptateurs pour l'accès aux machines et le transfert de fichiers. La machine virtuelle DeployWare est actuellement intégrée et utilisée comme outil de déploiement dans des distributions de serveurs d'applications tels que JOnAS ou encore PEtALS et JASMINe.

8.2 Perspectives

De nombreuses perspectives aussi bien scientifiques que techniques peuvent être envisagées suite à ce travail. Ces perspectives concernent les différents concepts de méta-modélisation choisis, mais également les plates-formes mises en œuvre dans notre approche.

8.2.1 Méta-modèle DeployWare/DACAR

Tout d'abord, les premières perspectives de notre travail concernent la définition des méta-modèles pour le déploiement de la couche intergicielle (DeployWare) et de la couche métier (DACAR). Il s'avère que certains logiciels proposent certaines subtilités difficilement modélisables à l'aide des concepts DeployWare. Par exemple, le serveur ESB PEtALS propose dans sa version 2.1.2 la notion de *domaine*. Un domaine PEtALS représente de manière abstraite un ensemble d'ESB qui partagent des propriétés. Par exemple, ces ESB sont configurés avec une adresse de *broadcast* à utiliser pour la diffusion des messages. Cette notion sert également à définir une portée à certains paramètres passés dans les messages qui transitent. Ces paramètres n'ont une valeur qu'à l'intérieur du domaine PEtALS pour lequel ils ont été définis. D'une manière plus générale, il est courant de ressentir le besoin de définir un ensemble de logiciels possédant les mêmes propriétés. Ainsi une perspective consiste à généraliser l'utilisation du domaine (déjà utilisé dans le cadre de l'application de politiques d'autonomie) afin de pouvoir factoriser une propriété commune à un ensemble d'instances de logiciels. L'utilisation à travers le temps du méta-modèle DeployWare permettra d'identifier de nouveaux concepts à injecter dans ce méta-modèle.

Concernant le méta-modèle d'architectures métiers, l'une des principales perspectives consiste à l'étendre à d'autres paradigmes que celui de composants. En effet, une architecture à base de services web par exemple ne peut être exprimée en terme de composants et de liaisons entre ces composants. Dans ces architectures par exemple, il est question de services, d'interfaces et d'orchestration. D'une manière générale, il peut être intéressant, même si cela constitue un travail de recherche à part entière, de se pencher sur la définition d'un méta-modèle générique d'applications métiers. Il s'agit là d'un problème difficile mais qui permettrait, dans

le cadre de DACAR, de pouvoir représenter n'importe quelle application, et d'y injecter des mécanismes d'auto-gestion.

Une perspective pour le méta-modèle a également été identifiée suite aux expériences menées dans le chapitre 7. Cette perspective consiste à rendre possible la description du déploiement d'un système sur plusieurs domaines. Cette problématique concerne principalement les systèmes logiciels répartis sur plusieurs domaines administrés par des personnes différentes, comme les architectures orientées service par exemple. Une perspective pourrait être de créer un référentiel de modèles DeployWare à l'exécution qui correspondraient aux différents *morceaux* du système déployés. Un administrateur qui viendrait alors se brancher sur le système existant pourrait charger et importer un modèle du référentiel pour venir l'enrichir avec les éléments de son domaine à déployer, qui dépendent éventuellement des éléments du modèle importé.

Enfin, une perspective importante pour l'approche consiste à unifier les méta-modèles employés. En effet, à ce jour les méta-modèle DeployWare et DACAR sont totalement indépendants, un choix qui se justifie par le peu d'interaction entre l'architecture métier et l'administrateur système. Pourtant, certaines vérifications statiques pourraient être effectuées entre les modèles de l'architecte métier et ceux de l'administrateur système (de la même manière qu'entre ceux de l'administrateur système et l'expert logiciel, comme les vérifications de cohérence). Par exemple, vérifier qu'un composant logiciel utilisant un service intergiciel donné est bien déployé sur un serveur de composants proposant ce service. Un autre exemple, certains intergiciels comme CORBA, permettent de faire interagir des composants implémentés dans différents langages de programmation. Il peut alors être utile d'assurer que le serveur d'un composant écrit en Java est bien déployé lui-même sur une machine offrant les bibliothèques nécessaires pour exécuter Java.

Il est également important d'envisager un langage spécifique graphique ou textuel pour certains rôles identifiés dans la procédure de déploiement. Cela aurait pour effet de rendre plus conviviale la description du déploiement pour chacun des rôles mentionnés. Par exemple, l'expert logiciel pourrait bénéficier d'un langage textuel pour définir les différentes instructions qui composent les procédures de déploiement du logiciel dont il a l'expertise. L'administrateur et l'architecte métier, quant à eux, pourraient bénéficier d'un formalisme graphique, plus adapté à leur rôle. Ces perspectives sont rendues possible par l'outillage autour des méta-modèles notamment fourni par Eclipse, comme le *Graphical Modeling Framework* [71], des approches spécifiques à Kermeta comme Sintaks [2], ou encore le canevas TOPCASED[1].

8.2.2 Vérifications statiques

Des travaux peuvent également être menés afin d'approfondir les vérifications statiques possibles à l'aide du méta-modèle. Actuellement, les vérifications mises en œuvre dans DeployWare sont uniquement structurelles. Ces vérifications utilisent la structure dédiée du méta-modèle pour vérifier la cohérence du déploiement ainsi décrit. Cela permet, statiquement, de détecter des erreurs potentielles durant le processus de déploiement. Ces erreurs auraient mis le déploiement en péril, et ainsi gaché les ressources potentiellement acquises avec un temps

[1] `http://gforge.enseeiht.fr/topcased/`

limité (réservations sur grille par exemple). D'autres problèmes, qui ne sont pas des erreurs à proprement parler, peuvent géner le déploiement. Il s'agit du temps de déploiement d'une architecture. En effet dans un contexte de très large échelle, c'est-à-dire plusieurs millions de machines, une préoccupation essentielle est que le temps mis à déployer les logiciels d'un système reste raisonnable, sous peine de voir sa réservation de ressources s'achever avant la fin du déploiement. Ainsi une perspective pourrait être d'optimiser la parallélisation des actions de déploiement, afin d'optimiser le temps de déploiement du système. Une première piste à explorer pourrait être d'utiliser la méta-modélisation exécutable (avec des framework tels que Kermeta), pour pouvoir simuler le déploiement en *exécutant les concepts du méta-modèle*. Cette simulation pourrait produire, dans un formalisme à déterminer, une analyse de la parallélisation, et ainsi fournir un support pour augmenter la parallélisation du déploiement à certains endroits —paralléliser le déploiement de logiciels indépendants qui était effectué séquentiellement par exemple. Ainsi, statiquement, il serait possible d'ajuster le déploiement de la couche intergicielle d'un système logiciel afin qu'il soit optimal en terme de temps d'exécution.

8.2.3 Environnements ouverts distribués

Des perspectives importantes sont à envisager pour la gestion des environnements ouverts. La principale limitation de l'approche DACAR réside dans le fait que la gestion des politiques d'auto-gestion est centralisée. Cela pose tout d'abord des problèmes d'efficacité lors du passage à l'échelle, mais cela pose d'autres problèmes, notamment dans les environnements ubiquitaires. En effet, la gestion des déconnexions est impossible à l'aide d'une gestion centralisée. Si un PDA rejoint le réseau, qu'il reçoit un certain nombre de logiciels, et qu'il quitte précipitamment le réseau, ou qu'il est déconnecté, il est impossible, pour une politique de déploiement d'agir sur lui, puis qu'il ne possède plus de connexion réseau le reliant à l'infrastructure de DACAR. De ce fait, l'exécution du moteur de politiques de DACAR pourrait être distribuée afin d'éviter, en premier lieu, qu'il devienne un goulet d'étranglement. De plus cela pourrait permettre de *déployer* l'auto-gestion sur les machines concernées, afin d'exécuter des politiques *locales*. Par exemple lorsqu'un terminal mobile entre sur le réseau, un micro-moteur DACAR pourrait y être déployé avec une politique à portée locale, qui se déclenche lorsque la connectivité est perdue, pour alors replier tous ce qui avait été déployé lors du passage dans le réseau. De cette manière, le terminal n'est pas pollué par les actions de déploiement et la réversibilité du déploiement est préservée.

8.3 Publications

Cette section recense les différents articles publiés au cours de cette thèse, au sujet du travail présenté.

– Jérémy Dubus et Philippe Merle : Vers l'auto-adaptabilité des architectures logicielles dans les environnements ouverts distribués. *Première Conférence Francophone sur les Architectures Logicielles (CAL'2006)*, Nantes, France. Septembre 2006. `http://www.sciences.univ-nantes.fr/lina/cal2006/`. Hermès Science. ISBN : 2-7462-1577-2. p. 13–29

- Jérémy Dubus et Philippe Merle : Applying OMG D&C Specification and ECA Rules for Autonomous Distributed Component-based Systems. *Workshop Models @ Runtime, in conjunction with MoDELS / UML 2006*, Gênes, Italie. Octobre 2006. `http://www.comp.lancs.ac.uk/computing/users/bencomo/MRT06/`.
- Jérémy Dubus et Philippe Merle : Autonomous Deployment and Reconfiguration of Component-Based Applications in Open Distributed Environments. Poster. In *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA'06)*, Montpellier, France. Novembre 2006. `http://www.cs.rmit.edu.au/fedconf/index.html?page=doa2006cfp`. On the Move to Meaningful Internet Systems 2006 : OTM 2006 Workshops. ISBN : 978-3-540-48269-7. Volume : 4277/2006. p. 26–27. Springer Verlag
- Jérémy Dubus et Philippe Merle : Towards Model-Driven Validation of Autonomic Software Systems in Open Distributed Environments. *Workshop M-ADAPT, in conjunction with ECOOP 2007*, Berlin, Allemagne. Juillet 2007. `http://www.comp.lancs.ac.uk/~bencomo/M-ADAPT07/index.html`.
- Areski Flissi, Jérémy Dubus, Nicolas Dolet et Philippe Merle : Deploying on the grid with DeployWare. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'08)*, Lyon, France. Mai 2008. `http://ccgrid2008.ens-lyon.fr/`. IEEE. p. 177–184.
- Jérémy Dubus, Areski Flissi, Nicolas Dolet et Philippe Merle : Une démarche orientée modèle pour déployer des systèmes logiciels répartis. *Revues des Sciences et Technologies de l'Information - Numéro spécial "Architectures Logicielles" de la revue L'Objet*. Juillet 2008. p. 35–59.

Annexe A

Descripteur OMG D&C d'une architecture de composants CORBA

Voici un exemple de descripteur de déploiement qui décrit la partie fixe (*i.e.* sur le nœud `CentralNode`) de l'architecture de l'application Plan Rouge décrite à la page 170

```
<?xml version="1.0"?>

<Deployment:Deployment
xmlns:Deployment="http://www.omg.org/Deployment"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
<Deployment:PackageConfiguration>
  <basePackage>
   <UUID>VictimManagerImpl</UUID>
   <label> VictimManager </label>
   <realizes>
    <port>
     <name>browser</name>
     <specificType>IDL:planrouge/Browser:1.0</specificType>
     <supportedType>IDL:planrouge/Browser:1.0</supportedType>
     <provider>true</provider>
     <exclusiveProvider> false </exclusiveProvider>
     <exclusiveUser>false</exclusiveUser>
     <optional>false</optional>
     <kind>Facet</kind>
    </port>

    <port>
     <name>collector</name>
     <specificType>IDL:planrouge/Collector:1.0</specificType>
```

```
      <supportedType>IDL:planrouge/Collector:1.0</supportedType>
      <provider>true</provider>
      <exclusiveProvider>false</exclusiveProvider>
      <exclusiveUser>false</exclusiveUser>
      <optional>false</optional>
      <kind>Facet</kind>
     </port>
    </realizes>

    <implementation>
     <name>VictimManager.car</name>
     <referencedImplementation>
      <label>victimmanager.jar</label>
      <monolithicImpl>
       <primaryArtifact>
       <name>VictimManagerImpl</name>
       <referencedArtifact>
        <label>victimmanager_jar</label>
        <UUID>VictimManagerImpl</UUID>
        <location>file:/C:/temp/archives/victimmanager.car</location>
        <execParameter>
         <name>entrypt</name>
         <value>
         <type>
           <value/>
         </type>
         <value>
           <string>org.objectweb.ccm.prototest.cif.
                   VictimManagerHomeImpl.create_home</string></value>
         </value>
        </execParameter>
        </referencedArtifact>
       </primaryArtifact>
     </monolithicImpl>
    </referencedImplementation>
   </implementation>
  </basePackage>
</Deployment:PackageConfiguration>

<Deployment:PackageConfiguration>
   <basePackage>
    <UUID>GlobalViewImpl</UUID>
    <label> GlobalView </label>
    <realizes>
```

```
     <port>
      <name>browser</name>
      <specificType>IDL:planrouge/Browser:1.0</specificType>
      <supportedType>IDL:planrouge/Browser:1.0</supportedType>
      <provider>false</provider>
      <exclusiveProvider> false </exclusiveProvider>
      <exclusiveUser>false</exclusiveUser>
      <optional>false</optional>
      <kind>SimplexReceptacle</kind>
     </port>
    </realizes>

    <implementation>
     <name>GlobalView.car</name>
     <referencedImplementation>
      <label>globalview.jar</label>
      <monolithicImpl>
       <primaryArtifact>
       <name>GlobalViewImpl</name>
       <referencedArtifact>
        <label>globalview_jar</label>
        <UUID>GlobalViewImpl</UUID>
        <location>file:/C:/temp/archives/globalview.car</location>
        <execParameter>
         <name>entrypt</name>
         <value>
         <type>
           <value/>
         </type>
         <value>
           <string>org.objectweb.ccm.prototest.cif.
                   GlobalViewHomeImpl.create_home</string></value>
         </value>
        </execParameter>
        </referencedArtifact>
       </primaryArtifact>
      </monolithicImpl>
     </referencedImplementation>
   </implementation>
  </basePackage>
</Deployment:PackageConfiguration>


<Deployment:Node>
```

```
 <name>CentralNode</name>
 <label>CentralNode</label>
</Deployment:Node>

<Deployment:DeploymentPlan>
 <artifact>
  <name>VictimManager</name>
  <location>file:/C:/temp/archives/victimmanager.car</location>
  <node>CentralNode</node>
  <source>GlobalViewImpl</source>
 </artifact>

 <artifact>
  <name>GlobalView</name>
  <location>file:/C:/temp/archives/globalview.car</location>
  <node>CentralNode</node>
  <source>GlobalViewImpl</source>
 </artifact>

 <instance>
  <name>vm</name>
  <node>CentralNode</node>
  <source>VictimManagerImpl</source>
  <implementation>
   <name>VictimManager.jar</name>
  </implementation>
  <configProperty>
   <name>name</name>
   <value>
    <type><boundedString/></type>
    <value><string>VM1</string></value>
   </value>
  </configProperty>
 </instance>

 <instance>
  <name>gv</name>
  <node>CentralNode</node>
  <source>GlobalViewImpl</source>
  <implementation>
   <name>GlobalView.jar</name>
  </implementation>
  <configProperty>
   <name>name</name>
```

```
    <value>
     <type><boundedString/></type>
     <value><string>GV1</string></value>
    </value>
   </configProperty>
  </instance>

  <connection>
   <name>connection1</name>
   <internalEndpoint>
    <portName>browser</portName>
    <provider>true</provider>
    <instance>
     <name>vm1</name>
    <source>VictimManagerImpl</source>
   </instance>
   </internalEndpoint>
   <internalEndpoint>
    <portName>browser</portName>
    <provider>false</provider>
    <instance>
     <name>gv1</name>
    <source>GlobalViewImpl</source>
    </instance>
    </internalEndpoint>
  </connection>
</Deployment:DeploymentPlan>

</Deployment:Deployment>
```

Bibliographie

[1] Meta Object Facility (MOF) Core Specification. Rapport technique, Object Management Group, Janvier. Version 2.0 - formal/06-01-01, 2006.

[2] Pierre ALAIN, Franck FLEUREY et Jean Marc JÉZÉQUEL : Model-Driven Analysis and Synthesis of Concrete Syntax. Rapport technique, Octobre 2006. Kermeta Workshop, Rennes, France.

[3] Robert ALLEN : *A Formal Approach to Software Architecture* . Thèse de doctorat, University of Carnegie Mellon, Pittsburgh, USA, Janvier 1997. Paru également en tant que Rapport Technique CMU-CS-97-144.

[4] OSGi ALLIANCE : Open Services Gateway Initiative Service Platform Specification. Rapport technique, Mai. Version 4.1, 2007.

[5] AS-2 EMBEDDED COMPUTING SYSTEM COMMITTEE SAE : Architecture Analysis and Desing Language (AADL). SAE standards nAS5506, Novembre 2004.

[6] Laurent BADUEL, Françoise BAUDE, Denis CAROMEL, Arnaud CONTES, Fabrice HUET, Matthieu MOREL et Romain QUILICI : *Grid Computing : Software Environments and Tools*, chapitre Programming, Deploying, Composing, for the Grid. Springer-Verlag, Janvier 2006.

[7] Olivier BARAIS : *Construire et Maîtriser l'évolution d'une architecture logicielle à base de composants*. Thèse de doctorat, Université des Sciences et Technologies de Lille - LIFL, Lille, France, Novembre 2005.

[8] Olivier BARAIS et Laurence DUCHIEN : *SafArchie Studio : An ArgoUML Extension to Build Safe Architectures* , pages 85–100. Springer, 2005. ISBN : 0-387-24589-8.

[9] Meriem BELGUIDOUM : *Conception d'une infrastructure pour un déploiement sûr et flexible des composants logiciels*. Thèse de doctorat, Ecole Nationale Supérieure des Télécommunications de Bretagne, Rennes, France, Mars 2008.

[10] Norbert BIEBERSTEIN, Sanjay BOSE, Marc FIAMMANTE, Keith JONES et Rawn SHAH : *Service-Oriented Architecture (SOA) Compass : Business Value, Planning, and Enterprise Roadmap*. IBM Press, Octobre 2005.

[11] Barry W. BOEHM : *Software Engineering Economics*. Prentice-Hall, 1981.

[12] Raphaël BOLZE, Eddy CARON et Michel Daydé et AL. : Grid'5000 : A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, Novembre 2006.

[13] Sara BOUCHENAK, Noël De PALMA, Daniel HAGIMONT et Christophe TATON : Autonomic Management of Clustered Applications. *In IEEE International Conference on Cluster Computing*, Barcelone, Espagne, Septembre 2006.

[14] Frédéric BRICLET, Christophe CONTRERAS et Philippe MERLE : OpenCCM : une infrastructure à composants pour le déploiement d'applications à base de composants CORBA. *In Actes de la 1ère Conférence Francophone sur le Déploiement et la (Re)Configuration des Logiciels (DECOR'04)*, pages 101–112, Octobre 2004. Grenoble, France.

[15] Laurent BROTO, Daniel HAGIMONT, Estella ANNONI, Benoit COMBEMALE et Jean-Paul BAHSOUN : A Model Driven Autonomic Management System. *In 5th International Conference on Information Technology - New Generations*, Las Vegas, Etats-Unis, Avril 2008.

[16] Laurent BROTO, Daniel HAGIMONT, Noël de PALMA et Suzy TEMATE : Autonomic Management Policy Specification in Tune. *In 23rd Annual ACM Symposium on Applied Computing*, Fribourg, Suisse, Mars 2008.

[17] Laurent BROTO, Patricia STOLF, Jean-Paul BAHSOUN, Daniel HAGIMONT et Noël de PALMA : Spécification de politiques d'administration autonome avec Tune. *In 6ème Conférence Française sur les Systèmes d'Exploitation (CFSE-6)*, Fribourg, Suisse, Février 2008.

[18] Éric BRUNETON, Thierry COUPAYE, Matthieu LECLERCQ, Vivien QUÉMA et Jean-Bernard STEFANI : The FRACTAL Component Model and Its Support in Java. *Software Practice and Experience – Special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12):1257–1284, août 2006.

[19] Éric BRUNETON, Thierry COUPAYE et Jean-Bernard STEFANI : The Fractal Component Model. OW2 consortium Specification Draft - 2.0-3, Février 2004.

[20] Eddy CARON, Chouhan Pushpinder KAUR et Dail HOLLY : GoDIET : A Deployment Tool for Distributed Middleware on Grid'5000. *In* IEEE, éditeur : *EXPGRID workshop. Experimental Grid Testbeds for the Assessment of Large-Scale Distributed Applications and Tools. In conjunction with HPDC-15.*, pages 1–8, Paris, France, Juin 2006.

[21] Benoît CLAUDEL, Noël DE PALMA, Renaud LACHAIZE et Daniel HAGIMONT : Self-protection for Distributed Component-Based Applications. *In 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (formerly Symposium on Self-stabilizing Systems) (SSS 2006)*, Dallas, TX, USA, Novembre 2006.

[22] Thierry COUPAYE et Christine COLLET : Denotational Semantics for an Active Rule Execution Model. *2nd International Workshop on Rules in Database Systems, Lecture Notes In Computer Science*, 985:36–50, 1995. Londres, Royaume-Uni.

[23] Eric M. DASHOFY, André van der HOEK et Richard N. TAYLOR : An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. *In Proceedings of the 24th International Conference on Software Engineering (ICSE2002)*, pages 67–82, Orlando, USA, Mai 2002.

[24] Pierre-Charles DAVID : *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation* . Thèse de doctorat, École des mines de Nantes et Université de Nantes, Nantes, France, Juillet 2005.

[25] Pierre-Charles DAVID et Thomas LEDOUX : Safe Dynamic Reconfigurations of Fractal Architectures with FScript. *In Proceedings of Fractal CBSE Workshop, ECOOP'06*, Nantes, France, Juillet 2006.

[26] Gan DENG, Jaiganesh BALASUBRAMANIAN, William OTTE, Douglas C. SCHMIDT et Aniruddha S. GOKHALE : DAnCE : A QoS-Enabled Component Deployment and Configuration Engine. *In Component Deployment*, pages 67–82, Grenoble, France, Novembre 2005.

[27] Jérémy DUBUS : Modélisation et génération automatique d'infrastructures de déploiement d'applications réparties. Mémoire de D.E.A., Laboratoire d'Informatique Fondamentale de Lille (LIFL), Lille, France, Juin 2005.

[28] Jérémy DUBUS et Philippe MERLE : Towards Model-Driven Validation of Autonomic Software Systems in Open Distributed Environments . *In Proceedings of the ECOOP Workshop on Model-Driven Adaptation*, pages 132–141, Berlin, Germany, Juillet 2007.

[29] Areski FLISSI, Christophe GRANSART et Philippe MERLE : Une infrastructure composants pour des applications ubiquitaires. *In 2nde conférence Ubiquité et Mobilité (UbiMob 2005)*, Grenoble, France, Juin 2005.

[30] Areski FLISSI et Philippe MERLE : Getting started with OpenCCM. Tutorial - Final Draft 1.0, Objectweb, Novembre 2002.

[31] Areski FLISSI et Philippe MERLE : Une démarche dirigée par les modèles pour construire les machines de déploiement des intergiciels à composants. *In Actes de la conférence Langages et Modèles à Objets (LMO 2005)*, Berne, Suisse, Mars 2005. Hermès Sciences.

[32] Jacqueline FLOCH, Svein HALLSTEINSEN, Erlend STAV, Frank ELIASSEN, Ketil LUND et Eli GJORVEN : Using Architecture Models for Runtime Adaptability. *IEEE Software*, 23(2):62–70, 2006.

[33] Ian FOSTER et Carl KESSERMAN : *The Grid : Blueprint for a New Computing Infrastructure*. 2004. ISBN : 1-55860-933-4.

[34] Nathalie FURMENTO, Anthony MAYER et Stephen McGough et AL. : An Integrated Grid Environment for Component Applications. *In Grid Computing - Grid 2001, Second International Workshop*, pages 26–37, Denver, USA, Mars 2001. LNCS Vol. 2242.

[35] David GARLAN, Robert T. MONROE et David WILE : Acme : An Architecture Description Interchange Language. *In Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, Novembre 1997.

[36] Aniruddha GOKHALE, Balachandran NATARAJAN, Douglas C. SCHMIDT et AL. : CoSMIC : An MDA Generative Tool for Distributed Real-time and Embedded Component Middleware and Applications. *In Actes du ACM OOPSLA 2002 Workshop on Generative Techniques in the Context of the Model Driven Architecture*, Novembre 2002. Seattle, États Unis.

[37] Richard S. HALL, Dennis HEIMBIGNER et Alexander L. WOLF : A Cooperative Approach to Support Software Deployment Using the Software Dock. *In ICSE '99 : Proceedings of the 21st International Conference on Software engineering*, pages 174–183, Los Alamitos, États-Unis, 1999. IEEE Computer Society Press.

[38] JAVA COMMUNITY PROCESS : Java(tm) Business Integration (JBI) 1.0 - Final Release. JSR 208, Sun Microsystems, Inc., août 2005.

[39] Jeffrey O. KEPHART et David M. CHESS : The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, 2003.

[40] Gregor KICZALES et Mira MEZINI : Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. *In* Andrew P. BLACK, éditeur : *ECOOP 2005*, volume 3586 de *LNCS*, pages 195–213. Springer, 2005.

[41] J. KRAMER, J. MAGEE et A. FINKELSTEIN : A Constructive Approach to the Design of Distributed Systems. *In Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS)*, pages 580–587, Washington, USA, 1990. IEEE Computer Society.

[42] Sébastien LACOUR : *Contribution à l'automatisation du déploiement d'applications sur des grilles de calcul* . Thèse de doctorat, Institut de Recherche en Informatique et Systèmes Aléatoires , Rennes, France, Décembre 2005.

[43] Sébastien LACOUR, Christian PÉREZ et Thierry PRIOL : Generic Application Description Model : Toward Automatic Deployment of Applications on Computational Grids. *In 6th IEEE/ACM International Workshop on Grid Computing (Grid2005)*, Seattle, WA, USA, Novembre 2005. Springer-Verlag.

[44] Vincent LESTIDEAU : *Modèles et environnement pour configurer et déployer des systèmes logiciels*. Thèse de doctorat, Université de Savoie, LISTIC/LSR-ADELE, Annecy, France, Décembre 2003.

[45] Martin LIPPERT : OSGi and Eclipse Equinox Explained. Rapport technique, Avril 2007. EclipIST 2007 Turkish Eclipse Submit.

[46] Adrien LOUIS : Build an SOA Application from Existing Services - Put Heterogeneous Services Together with the Petals ESB. *JavaWorld.com*, octobre 2006. http ://www.javaworld.com/javaworld/jw-10-2006/jw-1011-jbi.html.

[47] J. MAGEE, N. DULAY, S. EISENBACH et J. KRAMER : Specifying Distributed Software Architectures. *In* W. SCHAFER et P. BOTELLA, éditeurs : *Actes de 5th European Software Engineering Conférences (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, Septembre 1995. Springer-Verlag.

[48] Douglas MCILROY : Mass Produced Software Components. *In Proceedings of the NATO Software Engineering Conference*, pages 138–155, Octobre 1968.

[49] Nenad MEDVIDOVIC et Richard N. TAYLOR : A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26, issue 1:70–93, Janvier 2000. ISSN : 0098-5589.

[50] Pierre-Alain MULLER, Franck FLEUREY et Jean-Marc JÉZÉQUEL : Weaving Executability into Object-Oriented Meta-Languages. *In Actes de MODELS/UML'2005*, pages 264–278, Octobre 2005. Montego Bay, Jamaique.

[51] OBJECT MANAGEMENT GROUP : CORBA Components Specification. Available Specification formal/2006-04-01, Avril 2006.

[52] OBJECT MANAGEMENT GROUP : Deployment and Configuration of Component-based Distributed Applications. Available Specification formal/2006-04-02, Avril 2006.

[53] OBJECT MANAGEMENT GROUP : Unified Modeling Language : Superstructure. Available Specification, Version 2.1.2 formal/2007-11-02, Novembre 2007.

[54] OPEN SOA : SCA Service Component Architecture - Assembly Model Specification. SCA Version 1.0, Open SOA, Mars 2007.

[55] Michael PAPAZOGLOU : *Web Services : Principles and Technology*. Prentice-Hall, 2007.

[56] Klaus POHL, Günter BÖCKLE et Frank J. van der LINDEN : *Software Product Line Engineering — Fondations, Principles and Techniques*. Springer Link, 2005.

[57] AMPROS PROJECT : Example Applications and the Related Services, Version 1.0. Deliverable Report of WP4, novembre 2005.

[58] Vivien QUÉMA : *Vers l'exogiciel : une approche de la construction d'infrastructures logicielles radicalement configurables*. Thèse de doctorat, Institut National Polytechnique de Grenoble - LSR/IMAG, Grenoble, France, Décembre 2005.

[59] Stephan REIFF-MARGANIEC et Kenneth J. TURNER : Feature Interaction in Policies. *Computer Networks 45*, pages 569–584, Mars 2004. Department of Computing Science and Mathematics, University of Stirling, United Kingdom.

[60] Romain ROUVOY : *Une démarche à granularité extrêmement fine pour la construction de canevas intergiciels hautement adaptables : application aux services de transactions* . Thèse de doctorat, Université des Sciences et Technologies de Lille, LIFL/INRIA, Lille, France, Décembre 2006.

[61] Romain ROUVOY et Philippe MERLE : Description et vérification de motifs d'architecture avec FRACTAL ADL. *In 13ème Conférence Francophone sur les Langages et Modèles à Objets (LMO)*, L'Objet, pages 49–64, Toulouse, France, Mars 2007. Hermès Science.

[62] Romain ROUVOY, Nicolas PESSEMIER, Renaud PAWLAK et Philippe MERLE : Using Attribute-Oriented Programming to Leverage Fractal-Based Developments. *In 5th International ECOOP Workshop on Fractal Component Model*, Nantes, France, Juillet 2006.

[63] Davide SANGIORGI et David WALKER : *The Pi-Calculus - A Theory of Mobile Processes*. Cambridge University Press, 2003.

[64] Douglas C. SCHMIDT : Model Driven Engineering. *IEEE Computer Society*, pages 25—31, Février 2006. Vanderbilt University, Nashville, USA.

[65] Sylvain SICARD, Noël DE PALMA et Daniel HAGIMONT : J2EE Server Scalability through EJB Replication. *In 21st ACM Symposium on Applied Computing (SAC'06)*, Dijon, France, Avril 2006.

[66] Marc SNIR et Steve OTTO : *MPI-The Complete Reference : The MPI Core*. MIT Press, Cambridge USA, 1998.

[67] SUN MICROSYSTEMS : Java Message Service. Final Release version 2.1, Décembre 2003.

[68] SUN MICROSYSTEMS INC. : Java Enterprise Edition 5 Specification. JSR-000244 2.6, Novembre 2003.

[69] Clemens SZYPERSKI : *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Professional, Décembre 1997.

[70] Christophe TATON, Sara BOUCHENAK, Noël DE PALMA, Daniel HAGIMONT et Sylvain SICARD : Self-Optimization of Clustered Databases. *In 2nd International IEEE WoWMoM Workshop on Autonomic Communications and Computing (ACC 2006)*, Niagara-Falls, Buffalo-NY, Juin 2006.

[71] Artem TIKHOMIROV et Alexander SHATALIN : OSGi and Eclipse Equinox Explained. Rapport technique, Mars 2008. EclipseCON Conference, Santa Clara, USA.

[72] USTL/UVHC/EMD/INRETS : Projet CPER MOSAIQUES — MOdèles et infraStructures pour Applications ubIQUitairES. Rapport technique, CPER, Février 2005 au Juin 2007, Juin. Thème 2.1 Logiciel pour la communication — Rapport Final.

[73] W3C CONSORTIUM : Simple Object Access Protocol. W3C Recommendation version 1.2, Avril 2007.

[74] Gregory ZELESNIK : The UniCon Language Reference Manual. Manual, Carnegie Mellon School of Computer Science, Mai 1996.