UNIVERSITÉ JOSEPH FOURIER

Rapport scientifique présenté pour l'obtention d'une

Habilitation à Diriger des Recherches

Spécialité : « Informatique »

soutenue publiquement par

Fabienne Boyer

le 04 Décembre 2009

Titre :

Gestion de l'adaptabilité dans les applications réparties

JURY

Pr Christine Collet	Président
Mc Françoise Baude	Rapporteur
Pr Laurence Duchien	Rapporteur
Pr Gilles Muller	Rapporteur
Pr Olivier Gruber	Examinateur

Remerciements

Je tiens à remercier vivement Sacha Krakowiak, Professeur à l'Université Joseph Fourier à Grenoble, pour ses relectures attentives ainsi que pour ses commentaires qui m'ont aidé à finaliser ce document.

Je remercie également Jacques Mossière, Professeur à l'Institut Polytechnique de Grenoble et actuellement Directeur de l'ENSIMAG (Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble), pour ses relectures et pour son soutien tout au long de la rédaction de ce document.

J'adresse également de profonds remerciements à Olivier Gruber, Professeur à l'Université Joseph Fourier et leader du projet Synergy dans lequel s'inscrivent mes activités de recherche actuelles au sein de l'équipe Sardes, pour toute l'aide qu'il a apportée au niveau de la rédaction de ce document, ainsi que pour les nombreuses et fructueuses discussions qui en ont découlé.

Je remercie enfin tous les membres de l'équipe Sardes, et en particulier Jean-Bernard Stéfani, notre chef de groupe, pour son soutien et pour m'avoir permis de conserver une cohérence forte dans les activités de recherche que j'ai menées jusqu'à aujourd'hui.

Résumé

Ce mémoire présente une partie de mes travaux de recherche menés dans le domaine des applications réparties. Ces dernières anneés, les modèles à composants ont permis des avancées significatives dans la programmation des applications réparties. Cependant, ces dernières se confrontent aujourd'hui à un problème d'administrabilité. Le caractère distribué et évolutif de l'environnement d'exécution, le nombre de composants logiciels et les propriétés de qualité de service attendues sont autant de critères de complexité qui impactent la phase d'administration. L'un des verrous majeur pour l'administration est l'adaptabilité d'une application, c'est à dire la capacité d'adapter les composants logiciels qui la forment ainsi que leurs dépendances [116]. L'adaptabilité des dépendances dans les applications réparties est le fil directeur de ce document, dans lequel nous abordons dans un premier temps les capacités d'adaptation apportées par les modèles à composants, puis dans un deuxième temps les améliorations permises par le mariage des composants et de la refléxivité.

Mots-clés : Administration, Administration Autonome, Architecture logicielle, Modèle à composants.

Table des matières

Chapitre 1

Introduction

Ce mémoire décrit une partie des travaux de recherche que j'ai menés dans le domaine des applications réparties. Une application est dite répartie lorsqu'elle met en jeu des éléments logiciels coopérants qui s'exécutent sur plusieurs machines reliées par un réseau de communication. Ces dernières années, en raison du succès de l'Internet, ces applications ont connu un essor considérable qui a engendré une complexification des fonctions métier qu'elles sont censées fournir et des propriétés de qualité de service associées.

Ces propriétés de qualité de service peuvent porter sur différents critères tels que la disponibilité des services offerts, la sécurité et l'intégrité des actions effectuées et des données manipulées, les temps de réponse fournis aux usagers, etc. Ainsi par exemple, la plupart des applications de type Systèmes d'Information, qui fournissent des services en ligne, sont maintenant soumises à des contrats de disponibilité de type 7/7-24/24.

Ces attentes en termes de qualité de service impactent et complexifient toutes les phases du cycle de vie des applications réparties. Ce cycle est principalement formé des phases suivantes.

- La phase de conception, qui comprend la définition de l'architecture d'une application sous la forme d'un assemblage de composants logiciels associés à des propriétés de qualité de service diverses.
- La phase de développement, durant laquelle les composants logiciels qui constituent les briques de base de l'application sont mis en œuvre individuellement puis assemblés selon l'architecture préalablement définie.
- La phase de déploiement, généralement pilotée par des administrateurs humains, qui déploient et exécutent l'application dans un environnement d'exécution donné.
- La phase d'administration, durant laquelle l'application en cours d'exécution est supervisée pour satisfaire au mieux les besoins des utilisateurs finaux.

Pour mieux maîtriser la complexité inhérente à toutes ces phases, on a adopté les modèles à composants, qui visent à intégrer toutes les phases du cycle de vie d'une application au travers des deux tendances majeures suivantes.

D'une part, ces modèles minimisent les dépendances entre les phases du cycle de vie, ce qui permet de les associer à des acteurs différents et favorise la réutilisation du logiciel. Pour ce faire, ils séparent la définition des aspects fonctionnels d'une application (c'est-à-dire son code métier),

de ses aspects non fonctionnels tels que ses critères de qualité de service ou sa configuration répartie. Une illustration de ce principe peut être observée dans le fait que l'on cherche à rendre transparente la répartition d'une application, en permettant de programmer les composants logiciels indépendamment de leur configuration répartie future. Autrement dit, on n'impose pas au programmeur de connaître ou de fixer à l'avance la localisation des composants dans l'environnement réparti. Ceci contribue à l'indépendance des phases de programmation et de déploiement, ce qui permet de réutiliser une même application dans différents environnements répartis.

D'autre part, ces modèles cherchent à uniformiser les abstractions manipulées durant les différentes phases du cycle de vie, ce qui facilite l'intégration des différentes phases ainsi que la gestion des retours dans le cycle de vie. Les composants logiciels définis par les architectes lors de la phase de conception représentent la clé de voûte de cette uniformisation. Ils forment les éléments pivots qui sont manipulés à tout instant du cycle de vie applicatif. Chaque composant peut être suivi individuellement depuis sa programmation jusqu'à sa maintenance, facilitant ainsi la gestion de son évolution. [2] évoque cet aspect sous le terme d'absence d'érosion architecturale, signifiant que l'assemblage de composants qui forme une application, une fois défini par les architectes, reste explicite durant la vie de l'application.

Bien que les modèles à composants soient à la source d'avancées significatives dans la maîtrise de la complexité des applications réparties, certains aspects fondamentaux restent partiellement résolus. L'un de ces aspects est la capacité d'adaptation (ou adaptabilité) d'une application répartie au cours de sa vie. L'assemblage de composants qui forme une application doit pouvoir évoluer dans le temps pour s'adapter à des conditions d'exécution changeantes, liées par exemple à des versions nouvelles des composants logiciels, ou à des ajouts/retraits de machines dans l'environnement d'exécution. Le besoin d'adaptabilité est amplifié par la durée de vie généralement longue des applications réparties. En particulier, les fonctions métier et les critères de qualité associés doivent pouvoir évoluer au fur et à mesure de la prise en compte de nouveaux utilisateurs ou de nouveaux besoins.

L'adaptabilité reste un problème complexe car il ne s'agit pas seulement de fournir la mécanique permettant d'adapter une application, en modifiant son architecture par exemple. Il est également nécessaire d'entourer les processsus d'adaptation de critères de robustesse qui garantissent la préservation de l'intégrité de l'application. Le support d'une adaptation contrôlée constitue l'un des enjeux actuels des infrastructures supportant les applications réparties, et représente la problématique dans laquelle s'inscrivent les travaux présentés dans ce document.

1.1 Evolution des approches pour les applications réparties

Les années 80 ont vu l'apparition des systèmes ou des intergiciels répartis, définis comme une couche située entre le système d'exploitation et les applications, mettant en œuvre les fonctions dédiées à la répartition (localisation, communication, etc.). Ces intergiciels ont fourni différents modèles pour la programmation d'applications réparties : communication par messages, modèle client-serveur, mémoire partagée répartie. La communication par messages [83] a marqué les premiers pas de la programmation répartie, en permettant à un processus d'émettre un message vers un processus distant, avec attente ou non de la réception du message par le destinataire. L'abstraction de programmation répartie est alors le message qui est une structure de donnée non typée. Avec ce modèle, la validité de la structuration des données échangées ainsi que du protocole d'échange restent à la charge du programmeur.

Le modèle client-serveur a ensuite rendu disponible l'appel de procédure à distance [17] pour le programmeur d'applications réparties. L'appel de procédure permet à un processus d'exécuter une procédure sur une machine distante. Dans le même temps, sont apparues les mémoires partagées réparties [57][108], qui fournissent l'abstraction d'une mémoire partagée sur des systèmes ayant des mémoires physiquement reparties. Dans les deux cas, des problèmes majeurs liés à la répartition commencent à être pris en compte tels que l'hétérogénéité des machines, l'encodage des informations échangées, etc. Cependant, le niveau de transparence face à la distribution reste limité car la configuration répartie de l'application (en particulier, l'affectation des procédures aux processus) doit être connue et établie par le programmeur, et ne peut pas évoluer.

Dans les années 85-90, un niveau de transparence plus satisfaisant est obtenu avec les langages et modèles à objets répartis. Ceux-ci étendent les concepts des langages à objets au contexte réparti. Ils permettent de structurer une application sous la forme d'objets invocables à distance. Ces langages marquent une avancée majeure en fournissant la notion de référence d'objet réparti, qui est manipulable et transmissible comme une référence centralisée. La répartition devient donc transparente vis-à-vis de l'invocation de méthode, ce qui permet d'offrir une facilité de programmation inégalée jusque là.

En contre partie, certaines applications peuvent voir leurs performances impactées par ce modèle de programmation. Les objets répartis sont en effet transmis par référence, ce qui peut engendrer une augmentation significative du nombre de communications. Un autre manque concernant la plupart des modèles à objets répartis est que les aspects non fonctionnels tels que les transactions, la persistance, la protection, etc., ne sont pas pris en charge de manière unifiée par le modèle de programmation. Par ailleurs, les objets d'une application se référencent directement les uns avec les autres. Ce couplage fort limite les capacités d'adaptabilité de l'application. Finalement, bien que permettant d'améliorer la qualité du code logiciel produit, et en particulier la réutilisation des programmes et leur maintenance, la notion d'objet réparti s'est cependant avérée,à elle-seule, insuffisante. Le manque de support pour les propriétés non fonctionnelles, ainsi que le manque d'adaptabilité de l'application restent les causes principales de ce constat.

Ces difficultés ont motivé l'ajout de la réflexivité dans les langages à objets [44] ou plus généralement dans les intergiciels répartis. Le principe des systèmes réflexifs est d'introduire un niveau supplémentaire d'abstraction (appelé méta-niveau) permettant de contrôler l'exécution des fonctions de base fournies par ce système. Les fonctions de base sont dites réifiées par le méta-niveau. De nombreux prototypes de systèmes et d'intergiciels réflexifs ont été développés [5][61][41][124][42] ainsi que des supports d'exécution pour des langages réflexifs comme Iguana [14] ou ABCL/R3 [52]. Mais la réponse apportée par ces approches reste essentiellement d'ordre mécanique - c'est au programmeur d'exploiter les capacités réflexives pour mettre en place les propriétés non fonctionnelles ou les capacités d'adaptibilité. Le niveau de complexité pour le programmeur reste important.

De manière plus ou moins indépendante des approches réflexives, au départ tout au moins, la programmation par aspects, (AOP, Aspect Oriented Programming [30]), est née pour faciliter l'association de propriétés transverses (typiquement non fonctionnelles) aux objets. Les aspects permettent de programmer des services non fonctionnels indépendamment des objets métiers d'une application, en utilisant par exemple des langages spécialisés. Dans une deuxième étape, sur la base d'un outil de composition (weaver), les aspects non fonctionnels sont combinés et intégrés dans les objets métiers en des points d'interception bien définis, pour produire l'application

finale. L'une des avancées majeures des aspects est qu'ils ont permis de formaliser la définition de points d'interception dans les applications. Ce domaine est encore en plein essor car les problèmes liés à la composition d'aspects restent ardus [116]. Par rapport à l'adaptabilité, il faut noter qu'en l'absence de support particulier, les aspects permettent d'étendre avantageusement le code fonctionnel mais ne fournissent pas une réponse globale au problème d'adaptation des applications réparties (par exemple, il reste difficile de gérer le déplacement d'un objet d'une machine à une autre avec les aspects).

Les modèles à composants [20], incluant parmi eux les technologies actuellement commercialisées telles que JEE [68], .NET [22][110] ou CCM [93], ont été utilisés avec l'objectif de pallier ces limitations. Les composants possèdent des propriétés analogues à celles des objets mais présentent en outre des caractéristiques facilitant leur assemblage et leur association avec des propriétés non fonctionnelles. En particulier, la programmation par composant explicite les dépendances des composants entre eux et vis-à-vis de leur environnement. Ces dépendances sont exprimées en dehors du code des composants, et ne sont donc pas noyées dans le code métier. Parce qu'elles sont explicites, elles peuvent être prises en charge par l'infrastructure d'exécution, qui gère alors la mise en place de l'assemblage entre les composants ainsi que leur connexion avec les services non fonctionnels. L'expression explicite des dépendances des composants permet de gérer l'adaptabilité des applications de manière plus simple et mieux contrôlée.

Le patron de conception principalement utilisé pour permettre l'adaptabilité dans les modèles à composants est l'inversion de contrôle. Ce patron permet de manipuler, à divers moments du cycle de vie de l'application, les dépendances qui relient un composant à d'autres composants ainsi que celles qui le relient aux services de l'environnement. L'inversion de contrôle peut s'appuyer sur différents types de mécanismes sous-jacents, tels que l'interception, le tissage de code ou l'usage d'une structure réflexive. Selon les choix de conception, le niveau d'adaptabilité varie. Il peut recouvrir l'évolution, le remplacement ou la suppression de composants existants, l'introduction de nouveaux composants, la migration de composants d'un site à un autre, la modification des dépendances d'un composant vers les autres composants ou vers les services non fonctionnels.

Bien que les capacités d'adaptabilité fournies par l'inversion de contrôle soient significatives, la réponse apportée par ces solutions n'est pas complètement satisfaisante. Il reste trop complexe, pour un administrateur humain, de définir, déclencher et piloter les processus d'adaptation dans leur totalité. La mouvance actuelle est d'introduire des comportements adaptatifs autonomes pour gérer cette complexité. Le principe est de définir des boucles de commande capables de détecter certains événements se produisant dans le système, et d'y réagir en exécutant un plan d'adaptation particulier. L'approche qui consiste à exploiter pleinement les capacités d'adaptabilité au travers d'un contexte d'administration autonome est, de mon point de vue, très prometteuse. Elle représente un axe de travail en plein essor, qui combine les modèles à composants avec les infrastructures d'administration. C'est dans cette perspective que se situent mes travaux de recherche actuels.

1.2 Mon parcours

Les recherches que j'ai effectuées ont été menées à Grenoble, tout d'abord à l'Unité Mixte de Recherche Bull-IMAG de 1991 à 1995, puis ensuite au sein du Laboratoire Logiciel, Systèmes et Réseaux (LSR) de 1995 à 2007 et enfin au sein du Laboratoire d'Informatique de Grenoble (LIG)

de 2007 à 2009. Durant ces années, le fil directeur de mon parcours a concerné l'adaptabilité des applications réparties.

Au début des années 90, j'ai tout d'abord cherché à exploiter les capacités d'adaptation d'un système dans le but d'en améliorer les performances. Dans le contexte du système à objets répartis Guide [102], j'ai travaillé sur les mémoires réparties et partagées basées sur des architectures de type micro-noyaux [35]. Ces architectures, dites ouvertes, externalisent une partie des services qu'elles fournissent afin de rendre ceux-ci adaptables [8]. J'ai utilisé ces capacités d'adaptation au niveau du service de gestion de la pagination qui met en œuvre une mémoire partagée répartie par duplication de pages. J'ai amélioré ce service en y intégrant une cohérence relâchée de niveau causal, venant s'ajouter à la politique par défaut basée sur une cohérence stricte [33]. L'idée était d'adapter le niveau de cohérence en fonction des catégories d'objets applicatifs. En particulier, les objets Guide non synchronisés étaient gérés selon la cohérence causale, les autres restant sous le contrôle d'une cohérence stricte.

Dans le cadre de mon doctorat, j'ai ensuite poursuivi mes travaux dans le contexte du système Guide et plus largement des applications réparties, en élaborant un service de communication asynchrone basé sur un bus à événements, réalisé sur le système Guide.

À ce point de mon parcours, il m'a semblé important de revenir sur l'adaptabilité des applications réparties, et ce sous un angle plus général, non limité à l'aspect « performances ». L'adoption universelle des protocoles de l'Internet et l'émergence des réseaux sans fils et des ordinateurs portables introduisent des conditions d'exécution changeantes pour les applications (déconnexions temporaires, mobilité des utilisateurs, etc.). L'application doit pouvoir s'adapter en cours d'exécution à ces variations pour continuer à remplir sa fonction.

De multiples exemples illustrent le besoin d'adaptation. Lors de la déconnexion ou de la panne d'une machine par exemple, la configuration répartie de l'application en cours d'exécution doit pouvoir être révisée afin de redéployer les composants impactés par la panne, en minimisant autant que possible les pertes de données ou de traitements en cours. Lors d'une situation de surcharge, il peut être nécessaire de déplacer des composants d'une machine à une autre, ou d'élargir le parc de ressources utilisées par l'application. Par ailleurs, un nombre croissant d'applications utilise différents services interconnectés pour fournir leur fonction métier. Dans un environnement réparti, ces services peuvent s'insérer et se retirer à tout moment, ce qui implique d'adapter l'application en cours pour supporter ces échanges dynamiques de services.

L'adaptation est aussi un moyen pour favoriser la réutilisation de composants logiciels et par suite, diminuer la complexité de programmation des applications réparties. Les composants sont conçus de manière plus ou moins indépendante les uns des autres. Pour les réutiliser dans des environnements d'exécution différents, il faut pouvoir les adapter aux caractéristiques de ces environnements, et en particulier aux services fournis. En d'autres termes, le niveau de réutilisation d'un composant est fortement lié à la capacité de l'adapter à l'environnement dans lequel on souhaite le réutiliser.

Dans les années 1995 à 2000, j'ai focalisé mes activités autour de l'usage des modèles à composants dans le contexte des applications réparties. La description d'architecture explicite et la séparation entre code fonctionnel et non fonctionnel favorisent la réutilisation et l'adaptabilité. Avec Luc Bellissard, nous avons mené des travaux autour de la définition d'un modèle à composants non réflexif nommé Olan [103][71][72], fournissant un langage de définition d'archi-

tecture (en anglais ADL, Architecture Definition Language [87]) exhibant la structure répartie d'une application et les liaisons entre les composants. Nous avons conçu et implanté une machine d'exécution pour ce modèle, prenant en charge les phases de déploiement et d'exécution de l'application. L'adaptabilité fournie par Olan réside principalement dans la capacité d'adapter l'architecture d'une application à l'environnement d'exécution, avant ou au moment du déploiement. Les résultats de ces recherches ont confirmé que la description explicite de l'architecture d'une application est essentielle pour supporter un déploiement adaptatif.

Durant les années 2000 à 2003, je me suis concentrée sur les liaisons entre objets distants, qui constituent un point d'adaptation critique dans une application répartie. J'ai cherché à optimiser les performances d'exécution de ces liaisons en les adaptant aux caractéristiques des objets liés. J'ai plus précisément travaillé sur la conception d'un mécanisme d'invocation adaptable, fournissant deux politiques de gestion des objets répartis : avec ou sans cache [24]. Ce mécanisme utilise des objets d'indirection spécifiques pour la gestion des objets répartis avec cache. Le choix de la politique associée à un objet est réalisé lors de la phase de développement, par l'usage de marqueurs dans le code.

J'ai également choisi d'aborder le problème de l'adaptabilité de manière plus générale, au travers de la réflexivité, dans le cadre du projet Sirac. Un système réflexif possède une représentation de lui-même, peut s'en servir pour raisonner sur lui-même et, si besoin, modifier sa propre interprétation [16]. Lorsque la structure réflexive est présente à l'exécution, elle rend l'application adaptable dynamiquement. J'ai tout d'abord travaillé sur une approche réflexive des liaisons entre objets répartis, dans le contexte de la plateforme répartie Jonathan [13]. J'ai également étudié l'usage des mécanismes réflexifs pour supporter la mobilité des usagers au sein d'un environnement réparti [125]. Dans ces travaux, j'ai appliqué la réflexivité aux objets répartis, en visant essentiellement l'adaptation des invocations de méthodes, pour prendre en compte des contraintes de l'environnement d'exécution ou des aspects non fonctionnels.

Bien que ces travaux aient donné des résultats intéressants, je pense que le potentiel fourni par la réflexivité n'est pas pleinement exploité dans un contexte de programmation à base d'objets. Il me paraît crucial de supporter l'adaptabilité au niveau de l'architecture logicielle d'une application répartie. L'architecture répartie définit les éléments logiciels (objets) et leurs dépendances (délégation, héritage, etc.), ainsi que la projection de ces éléments dans l'environnement d'exécution (répartition des éléments logiciels sur les machines de l'environnement par exemple). L'adaptabilité de l'architecture est indispensable pour pouvoir disposer de fonctions d'administration avancées, telles que la possibilité de remplacer un objet par un autre, ou de déplacer un objet d'un site vers un autre.

En me basant sur l'expérience acquise sur les modèles à composants au travers du projet Olan et sur les techniques réflexives au travers de mes travaux dans Sirac, il devenait naturel d'explorer l'usage des modèles à composants réflexifs pour le support de l'adaptabilité. Les modèles à composants explicitent l'architecture des applications réparties avec l'objectif de faciliter leur administration. La combinaison de ces modèles et de la réflexivité fournit une solution qui, par principe, permet d'inspecter dynamiquement l'architecture d'une application et d'adapter cette architecture à l'exécution. Cette possibilité permet d'administrer une application en se basant sur des mécanismes systématiques. Par exemple, la panne d'un composant peut être traitée en remplaçant ce composant par un autre et en rétablissant ses liaisons avec les autres composants de manière cohérente.

Un atout essentiel de l'adaptabilité dynamique est qu'elle offre les mécanismes permettant d'administrer une application répartie de manière autonome. Cette approche de l'administration a été initialement proposée par les laboratoires de la société IBM (International Business Machine) [55]. Elle est basée sur des boucles de commande fermées réagissant à des événements liés à l'évolution de l'environnement d'exécution ou de l'application. A la suite d'un événement, la boucle de commande procède à une analyse de l'état courant de l'application, dans le but de décider des adaptations qui doivent être effectuées pour traiter l'événement.

Durant les années 2004 à 2008, j'ai mené ces recherches sur l'adaptabilité et son usage dans un contexte autonome dans le cadre du projet Jade [112][115][118], dont l'objectif est de fournir une infrastructure autonome d'exécution et d'administration d'applications réparties. J'ai encouragé l'usage du modèle à composants réflexifs Fractal [29] pour mettre en œuvre Jade, de manière à disposer de capacités de réflexivité. Celles-ci permettent de réifier l'architecture répartie d'une application administrée par Jade, sous la forme d'un graphe de composants représentant les éléments logiciels de l'application ainsi que les éléments matériels de l'environnement d'exécution. Cette réification, présente à l'exécution, permet d'adapter dynamiquement l'application et sa projection dans l'environnement d'exécution. Plusieurs boucles de commande autonomes ont été mises en œuvre, réalisant l'auto-réparation de composants en cas de panne, l'auto-optimisation ou encore l'auto-protection. Ces boucles ont pu être conçues de manière générique, sur la base des fonctions réflexives fournies par le modèle à composants Fractal, indépendamment d'une application particulière. Elles offrent cependant la possibilité d'être spécialisées pour des contextes applicatifs spécifiques.

L'approche que j'ai poussée dans Jade pour l'adaptabilité, basée sur la réification de l'architecture d'une application par réflexivité, a été validée par plusieurs expérimentations dans des contextes applicatifs patrimoniaux tels que JEE [68] ou JMS [96]. Cette approche a également été adoptée par d'autres projets de recherche tels qu'OpenCOM [43] ou Plastik [122]. L'originalité majeure de notre solution se situe au niveau de l'administrabilité de l'infrastructure d'exécution et d'administration elle-même. Nos travaux ont mis en évidence la nécessité de pouvoir adapter, à l'exécution, non seulement l'application, mais également l'infrastructure sous-jacente. Cette dernière est sujette aux évolutions de l'environnement, tout comme l'est l'application administrée. Elle doit donc être adaptée pour préserver son intégrité. J'ai proposé une organisation récursive, dans laquelle l'infrastructure est capable de s'auto-administrer. Cette organisation est le moyen de fournir une infrastructure complètement autonome pour des applications réparties.

1.3 Objectifs et plan du document

Ce document présente les principes des modèles à composants, ainsi que leurs apports pour la mise en œuvre des applications réparties adaptables. Nous abordons ces modèles selon deux niveaux. Tout d'abord, nous nous plaçons au niveau des utilisateurs des modèles à composants, à savoir les architectes d'une application, les programmeurs, et les administrateurs. Nous décrivons quels sont les paradigmes fournis par ces modèles, en accordant une attention particulière aux aspects suivants : la définition des liaisons entre les composants, de leur cycle de vie, et leur association avec des propriétés non fonctionnelles ainsi qu'avec des ressources (par exemple, machines) de l'environnement d'exécution.

Le deuxième niveau que nous considérons est celui de l'infrastructure d'exécution sous-jacente,

supportant les différentes phases du cycle de vie du modèle à composants. Nous nous intéressons aux mécanismes internes de mise en œuvre qui sont utilisés, tels que l'injection de références, l'interception, le tissage de code et la réflexivité. Les critères de qualité que nous valorisons à ce niveau sont les capacités d'adaptation fournies et le moment auquel elles sont disponibles. Nous considérons en particulier l'adaptabilité durant les phases de déploiement (adaptabilité à l'environnement d'exécution) et durant l'exécution (adaptabilité aux conditions d'exécution changeantes).

Le reste de ce document est organisé comme suit. Nous avons choisi de séparer la présentation des modèles à composants élémentaires de ceux basés sur une approche réflexive. Le chapitre 2 traite des principes de base des composants et de leur utilisation pour le développement et l'administration d'applications réparties. Notre contribution à cet axe de travail est récapitulée au travers de la présentation du projet Olan. Le chapitre 3 considère ensuite l'introduction de la réflexivité dans les modèles à composants. Après une présentation générale de ces modèles, nous décrivons les activités que nous avons menées dans le cadre du projet Jade, dont l'objectif est de fournir une infrastructure à composants réflexifs auto-administrés. Ce document se termine par la présentation de nos conclusions et de nos perspectives de recherche autour du support de l'adaptabilité dans les modèles à composants.

Chapitre 2

Modèles à composants

Le passage de la technologie objet à celle des composants a été une évolution majeure dans le domaine des applications réparties, qui a débouché sur des technologies industrielles telles que JEE[68], SPRING[2], CCM[93] ou encore .NET[22][110]. Les composants ont apporté des solutions aux limitations reconnues de l'approche objet. D'une part, le développement de briques logicielles réutilisables a été formalisé et simplifié. D'autre part, l'assemblage de ces briques en un système opérationnel ainsi que l'administration de ce dernier ont été supportés autour du concept d'architecture logicielle.

L'architecture logicielle capture la structure d'une application, en termes des composants qui la forment et de leurs dépendances. Bien que plusieurs définitions existent pour la notion de composant, celle de Szyperski [20] synthétise assez bien notre point de vue.

Szyperski 2002 : A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

L'idée centrale est d'exprimer les dépendances des composants séparément de leur code. L'application de ce principe implique de séparer les aspects fonctionnels (le code métier) des aspects non fonctionnels d'une application. Les aspects non fonctionnels concernent les propriétés de qualité de service requises par les composants ou bien les ressources dont ils ont besoin pour s'exécuter. Ces aspects modélisent donc des dépendances vers des services techniques (par exemple, service de transaction) ou des ressources (par exemple, machines, threads) de l'environnement d'exécution.

La séparation des aspects non fonctionnels du code métier réduit fortement la complexité de développement de ce code. De plus, cette séparation est un prérequis pour permettre de déployer a posteriori un composant dans un environnement d'exécution puisque le composant exprime explicitement ses besoins par rapport aux services et aux ressources qu'il attend de cet environnement. In fine, l'utilisation de composants structure la création et la gestion d'une application répartie avec l'apparition de différents rôles parmi lesquels nous considérons les suivants comme majeurs : l'architecte, le développeur et l'administrateur.

L'architecte est la personne qui produit la description explicite de l'architecture de l'application. Sa responsabilité va au delà d'une seule application, il s'efforce de définir des composants

réutilisables. Cette réutilisabilité s'exprime sur le code métier mais elle est dépendante d'une adaptabilité des composants à leur utilisation dans des contextes potentiellement différents.

Le développeur est chargé de construire et de tester les composants définis dans l'architecture. Sa tâche est simplifiée de par la séparation du code métier de la gestion des dépendances. Les composants peuvent ainsi être développés et testés indépendamment les uns des autres, et assemblés a posteriori. De plus, le développeur peut se focaliser sur le code métier, et laisser la gestion des aspects non fonctionnels à la charge de l'infrastructure qui supporte l'exécution des composants, sous le contrôle de l'administrateur.

L'administrateur a pour rôle de piloter le déploiement et la gestion de l'application dans un environnement d'exécution donné. L'environnement d'exécution dénote l'ensemble des ressources matérielles et logicielles supportant l'exécution de l'application et son infrastructure sous jacente. L'administrateur doit adapter l'application aux caractéristiques de cet environnement. Ainsi, il choisit certaines versions de composants en fonction des capacités de l'environnement (par exemple, mémoire, cpu). De même, il réalise les liens avec les services non fonctionnels disponibles (par exemple, l'usage de fichiers versus l'usage d'une base de donnée pour la persistance).

L'administration est une tâche à long terme car l'environnement d'exécution évolue dans le temps. Cet environnement est généralement constitué d'un ensemble de machines interconnectées. Les évolutions sont de différentes natures comme les pannes temporaires de machines ou bien l'accroissement du parc matériel avec de nouveaux modèles de machines. Pour gérer ces évolutions, l'administrateur doit pouvoir redéployer les composants dynamiquement. Cela nécessite que l'architecture logicielle soit manipulable durant l'exécution et non simplement utilisée pendant les phases de développement et de déploiement. L'architecture logicielle crée alors un lien entre les phases du cycle de vie du logiciel, de son développement jusqu'à son administration.

L'administration d'une application répartie ne se limite pas au redéploiement de certains composants mais couvre l'ensemble des activités d'adaptation qui sont nécessaires au cours de la vie de cette application. Bien souvent, de nouvelles versions de composants devront être installées pour corriger des bogues, améliorer les performances ou même augmenter les fonctionnalités. A nouveau, l'architecture doit être adaptable puisque cela peut nécessiter de changer les contrats des composants et leurs dépendances. Par exemple, un administrateur peut devoir changer de fournisseur pour une fonction métier donnée, changeant les composants utilisés et la répartition des services qu'ils proposent.

Les différents principes énumérés précédemment sont autant de dimensions d'adaptation des applications réparties. Toutes ces formes d'adaptation reposent sur la présence d'une architecture logicielle explicite, et de son exploitation par l'infrastructure qui supporte l'exécution des composants. L'adaptabilité des composants est réalisée via l'inversion de contrôle, un patron de conception qui externalise la gestion des dépendances des composants vers l'infrastructure. Cette externalisation repose sur l'usage de différentes techniques, combinant à la fois du tissage de code et de l'injection de références. C'est cette externalisation qui permet de préserver le code métier tout en offrant l'adaptabilité nécessaire.

Dans ce chapitre, nous allons détailler les fondements de cette adaptabilité. La Section 2.1 présente les principes de base des modèles à composants. La mise en œuvre de l'inversion de contrôle sur la base de ces principes est abordée dans la section 2.2. Avant de conclure, nous illustrons les avantages de l'approche à composants pour l'adaptabilité dans le contexte de nos

travaux dans le projet Olan [103].

2.1 Principes de base des modèles à composants

Historiquement, la notion de composant logiciel a été proposée et explorée dans le domaine du génie logiciel, pour permettre de construire une application par assemblage de briques logicielles. Dans les années 80-90, les idées associées à cette notion ont été reprises dans le contexte des applications réparties et ont débouché sur une offre industrielle importante ainsi que sur un domaine de recherche aujourd'hui très actif. Dans cette section, nous présentons les principes de base des modèles à composants. Dans la section 2.1.1, nous reprenons tout d'abord la vision de Szyperski et le principe sous-jacent d'encapsulation des composants. La notion d'architecture logicielle qui en découle et qui sert de fondation à l'externalisation de la gestion des dépendances est abordée dans la section 2.1.2. La section 2.1.3 présente ensuite les langages de définition d'architectures logicielles (ADL).

2.1.1 Encapsulation

Bien que les modèles à composants existants aient chacun leurs spécificités, ils s'appuient tous sur l'encapsulation des composants qui représente un fondement commun. D'un point de vue général, l'encapsulation d'un composant signifie que les seules interactions possibles entre ce composant et le monde extérieur correspondent à celles définies dans ses dépendances. Les dépendances décrivent les services dont a besoin un composant ainsi que ce qu'il fournit, tant sur le plan fonctionnel que non fonctionnel.

Pour satisfaire la propriété d'encapsulation, les approches à composants séparent la définition d'un service de son implémentation comme le font les approches à objets. L'approche à composants va plus loin cependant car les dépendances considérées définissent non seulement les services fournis mais également ceux requis par les composants. De plus les dépendances peuvent aller au-delà des dépendances fonctionnelles, incluant dans le principe au moins, les aspects non fonctionnels tels que la persistance des composants ou bien les propriétés transactionnelles de leur exécution. Que les dépendances soient fonctionnelles ou pas, la résolution d'une dépendance repose sur la notion de contrat.

Le contrat associé à une dépendance caractérise le service attendu. Autrement dit, il caractérise ce que doit fournir le composant qui viendra « satisfaire » la dépendance. Les services sont à la base spécifiés en termes de leur interface, généralement exprimée dans un langage pivot fortement typé tel qu'un langage de description d'interface (IDL en anglais) [49]. D'après [3], quatre niveaux de contrats peuvent être spécifiés au niveau d'un service. Le niveau 1 s'applique à la définition du type, statiquement vérifiable, du service. Le niveau 2 s'attache à la sémantique de chaque opération fournie par le service. Le niveau 3 permet de spécifier des contraintes de synchronisation entre les opérations, et le niveau 4 tient compte de propriétés non fonctionnelles, liées à la qualité de service fournie.

Ainsi le contrat peut prendre des formes complexes portant sur la nature des interactions ou sur des aspects techniques s'y rattachant. Lorsqu'il s'agit de dépendances non fonctionnelles liant un composant avec un service de l'infrastructure, le contrat est souvent bidirectionnel car l'infrastructure a besoin de manipuler le composant mais le composant veut pouvoir influencer la façon dont il est géré. Cette double interaction met en jeu diverses technologies telles que le tissage

de code et la gestion d'événements ; nous y reviendrons dans la section sur l'inversion de contrôle.

Quelquesoit la nature des contrats supportés, la propriété d'encapsulation demeure la clé de voûte de l'approche à composants. Elle permet d'avoir un déploiement des composants indépendant de leur programmation, et permet par là même leur réutilisation. L'idée est qu'il suffit de satisfaire toutes les dépendances d'un composant pour permettre son fonctionnement dans un environnement d'exécution donné. Cela sous-entend en effet que toutes les dépendances requises par un composant pour assurer son fonctionnement soient explicitées.

Dans les premiers modèles à composants, les dépendances étaient souvent limitées aux aspects fonctionnels, le support de propriétés non fonctionnelles étant laissé à la charge des développeurs. Sans parler de la surcharge de programmation engendrée par la production du code lié aux propriétés non fonctionnelles, cette approche conduit à une perte d'adaptabilité et donc de réutilisabilité importante. Dans les modèles à composants plus récents, l'encapsulation porte sur un ensemble plus large de dépendances qui comprend des dépendances non fonctionnelles. Mais la situation n'est pas encore totalement satisfaisante comme l'illustrent les exemples ci-après.

Par exemple, JEE [68] explicite à la fois les dépendances fonctionnelles des composants ainsi que celles non fonctionnelles vers des services de l'infrastructure tels que les services de persistance ou de gestion de transactions. On ne peut toutefois pas exprimer de dépendances vers l'environnement d'exécution, que cela soit vers le type du Java Runtime Environment (JRE) ou encore vers le type de machines ou de système d'exploitation. S'il est vrai que Java est un langage dont la portabilité n'est plus à démontrer, celle d'une application ou d'un composant est loin d'être acquise. Le monde Java est fragmenté entre différents profils (J2ME, J2SE et JEE) et les versions de Java introduisent maintenant de profondes différences. JEE est un cas typique où l'adaptabilité des composants est bien traitée mais uniquement dans le contexte d'une infrastructure particulière. Autrement dit, les composants JEE sont et restent des composants JEE, ils ne sont pas indépendants de leur environnement et ne peuvent être adaptés vers d'autres infrastructures.

Fractal et son incarnation Julia en Java [29] est un autre exemple. Le modèle Fractal est puissant et général par son extensibilité ; il permet d'exprimer des dépendances fonctionnelles et non fonctionnelles. Ainsi la dépendance non fonctionnelle d'un composant vers une machine de l'environnement d'exécution est exprimée dans l'ADL au travers de la notion de machine virtuelle. Cependant, les dépendances non fonctionnelles considérées restent limitées. En particulier, Julia ne couvre pas les dépendances de types, au sens des interfaces et des classes Java qui sont requises pour pouvoir exécuter un composant. Ces dépendances n'étant pas explicitées, elles ne sont pas gérées par l'infrastructure d'exécution. En conséquence, c'est à l'administrateur humain de placer manuellement les interfaces et les classes requises aux endroits adéquats sur les différentes machines.

Certaines infrastructures à composants se basent sur OSGi [10] pour la gestion des dépendances vers des interfaces et des classes Java. Cependant, ces infrastructures fragmentent la déclaration des dépendances entre OSGi pour les types et le modèle à composants pour les services. C'est en particulier le cas pour la combinaison en vogue entre la plate-forme OSGi et le canevas Spring Framework [59][2].

2.1.2 Architecture logicielle explicite

Les modèles à composants explicitent l'architecture logicielle d'une application, c'est-à-dire qu'ils permettent de manipuler l'assemblage de composants qui forme une application. Cette architecture logicielle explicite s'appuie sur la propriété d'encapsulation des composants et sur la méta-déclaration de leurs dépendances qui en découle. Plus l'encapsulation est complète du point de vue des dépendances considérées, plus l'architecture logicielle est riche d'informations et plus elle représente la réalité de la structure distribuée d'une application. Cependant, il n'y a toujours pas vraiment de consensus sur la notion d'architecture logicielle et les dépendances considérées sont souvent incomplètes comme nous l'avons évoqué précédemment.

Historiquement, et plus particulièrement dans la vision génie logiciel, la description d'une architecture répartie consistait en la description des dépendances fonctionnelles entre des composants représentant des modules de code, au travers de langages qualifiés de MIL (Module Interconnection Languages) [101]. L'objectif était de promouvoir le développement de composants réutilisables. La description d'une architecture en termes d'un assemblage d'instances de composants a été considérée plus tard, au travers de modèles à composants tels que Darwin [63][64], Olan [103], Fractal [29], Spring [59][2], etc. De la même manière, la prise en compte de dépendances non fonctionnelles dans l'architecture n'a pas été immédiate, et reste partiellement couverte par la plupart des modèles à composants existants. À l'exception de [70], peu de modèles explicitent à la fois les besoins des composants en termes de propriétés de qualité de service, ainsi que leurs besoins en termes de ressources matérielles et logicielles.

Il nous paraît important que l'architecture logicielle d'une application répartie capture ces différentes formes de dépendances, illustrées dans la figure 2.1, que nous regroupons sous le terme de *dépendances architecturales*. De plus, nous pensons que celles-ci devraient être unifiées quant à leur définition et à leur prise en charge par l'infrastructure d'exécution. Cette dernière gère les composants tout au long de leur cycle de vie. De plus, elle fournit un ensemble de services non fonctionnels utilisables par les composants pour obtenir des propriétés de qualité de service. Elle peut englober des services non fonctionnels particuliers [68], ou bien fournir simplement la capacité d'intégrer ces services en fonction des besoins applicatifs et des caractéristiques de l'environnement d'exécution sous-jacent [59][2].

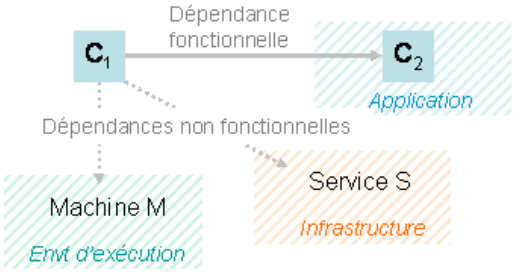

FIG. 2.1 – Types de dépendances architecturales

Dans cette vision unifiée, les **dépendances fonctionnelles des composants** représentent classiquement les services métier qu'ils requièrent pour leur exécution. Par exemple, un composant de commerce électronique ShoppingCart pourra présenter une dépendance fonctionnelle

vers un composant BankAccount fournissant des services d'achat bancaire.

Les **dépendances non fonctionnelles d'un composant vers l'infrastructure d'exécution** concernent majoritairement les besoins du composant en termes de propriétés de qualité de service. Par exemple, un composant peut avoir une dépendance vers un service de persistance, de transactions, ou de sécurité. Comme dit précédemment, ces propriétés sont mises en œuvre par les services hébergés par l'infrastructure.

Enfin, les **dépendances non fonctionnelles d'un composant vers son environnement** expriment ses besoins en termes de ressources, telles que par exemple des machines d'exécution. Ces dépendances englobent des aspects liés à l'administration (description de paramètres de configuration physique tels que localisation/groupement).

L'un des aspects critiques de l'architecture logicielle est qu'elle doit être présente à toutes les phases du cycle de vie-du développement à l'administration. L'implication est que toutes les dépendances mises en jeu lors de ces phases devraient être modélisées dans l'architecture. Par exemple, des dépendances entre des composants et des outils devraient être explicitées, telles que le fait que certains composants requièrent un compilateur de type Ahead Of Time (AOT) pour être déployés. De la même manière, la dépendance entre un composant exécutable et le fichier contenant le code source du composant pourrait être explicitée. Aujourd'hui encore, le cycle de vie global n'est pris en compte que partiellement dans la description d'architecture.

2.1.3 Langage de description d'architectures (ADL)

Dans les années 90 s'est généralisé l'usage des ADL (Architecture Definition Language) [87] pour décrire l'architecture logicielle initiale d'une application répartie, sous la forme d'un assemblage incluant la définition des composants et de leurs dépendances. Le premier langage de ce type (CONIC [62]), écrit en Pascal, a été utilisé dès les années 85.

Les types de dépendances que l'on peut décrire varient selon les ADL [64] et selon le modèle de composants auquel ils sont associés [23]. La plupart permettent de manipuler des dépendances fonctionnelles qui reflètent celles supportées par le modèle à composants sous-jacent (par exemple, délégation, composition). Les dépendances vers les services non fonctionnels sont décrites par certains ADL seulement, sous la forme d'étiquettes associées aux composants (*transactional*, *persistent*, etc.) [68]. Enfin, les dépendances vers l'environnement d'exécution peuvent être exprimées sous la forme de contraintes sur les ressources matérielles qui seront considérées au moment du déploiement [29][23].

Les ADL permettent d'exprimer l'architecture séparément du code métier des composants, ou bien de manière intégrée à ce code sous la forme d'annotations ou d'équivalents [56]. L'exemple donné dans le listing 2.1 montre un extrait de l'ADL Olan [103] définissant deux composants nommés Client et Server, avec leurs dépendances. L'assemblage de ces deux composants forme un troisième composant, nommé ClientServer. L'assemblage est ici défini à l'extérieur du code des composants. En revanche, dans l'exemple JEE donné dans le listing 2.2, les dépendances et l'assemblage sont décrits au travers d'annotations placées dans le code des composants. Ainsi par exemple, l'annotation *@EJB* placée devant la déclaration du champ nommé *server* de type *Server* signifie qu'à l'exécution, toute instance de la classe *Client* sera liée à une instance de la classe *Server*.

Des exemples d'ADL sont Darwin [63][64], xADL [77][78], C2 [88], Olan [103], Wright [111], ou encore [26] qui propose un ADL indépendant d'un environnement d'exécution donné, permettant de déployer des architectures à composants hétérogènes et distribuées. Certains d'entre eux incluent des aspects dynamiques permettant d'exprimer des évolutions de l'architecture à l'exécution, mais ces capacités restent globalement limitées par le caractère déclaratif des ADL et constituent toujours une problématique d'actualité [90].

Listing 2.1 – Exemple d'assemblage dans Olan

```
Composite component ClientServerCmp
  implementation ClientServerCmp uses ClientCmp, ServerCmp{

  // Sub-components
  Client = instance ClientCmp;
  Server = instance ServerCmp;

  // Interconnections between Client and Server sub-components
  Client.service(Request r) => Server.service(r);
  ...
};
```

Listing 2.2 – Exemple d'assemblage dans JEE

```
class Client{
   @EJB Server server ;  // interconnexion with Server component

   public Object service(Request r){
      return server.service(r) ;
   }
   ...
}
```

Les ADLs sont généralement associés à un ensemble d'outils, permettant de vérifier la validité de l'assemblage, et dans certains cas [56][103][77][88] de supporter la phase de déploiement de l'application. Cette phase installe et crée les composants, et finit de résoudre les dépendances qui ne sont pas déjà résolues par les liaisons décrites dans l'assemblage. Certains ADL permettent en effet d'exprimer, au niveau de l'architecture, des dépendances sous une forme non résolue [53][10][46]. Une dépendance non résolue d'un composant C vers un composant C'exprime des contraintes sur C' sans préciser l'identité du composant qui jouera le rôle de C' à l'exécution.

Ainsi une dépendance non fonctionnelle entre un composant et une machine de l'environnement peut être exprimée par une contrainte sur le type d'OS ou sur une capacité mémoire minimale, sans préciser quelle machine sera choisie à l'exécution. De la même manière, une dépendance fonctionnelle d'un composant vers un autre peut être associée à des contraintes sur la qualité du service rendu ou sur la localisation du composant fournissant le service attendu. Prenons l'exemple d'un composant C1 présentant une dépendance fonctionnelle vers un service métier. Au niveau de l'ADL, cette dépendance définit l'interface métier attendue - disons l'interface langage IFoo. Mais dans la mesure où cette dépendance n'identifie pas le composant devant fournir le service, elle est considérée comme non résolue par l'ADL.

Sans résoudre la dépendance au niveau de l'ADL, il est possible de paramétrer cette résolution, par des attributs portant sur la localisation du service demandé, sur sa version ou encore sur le niveau de qualité de service attendu. Tant que le fournisseur du service n'est pas spécifié au niveau de l'ADL, la dépendance est dite non-résolue. À l'exécution, l'infrastructure se chargera de résoudre cette dépendance par un service implémentant l'interface IFoo, quelquesoit l'identité du composant fournissant ce service.

Ainsi l'architecture décrite par un ADL n'est pas forcément opérationnelle si elle contient des dépendances non résolues et présentant un caractère obligatoire pour les composants concernés. Cette résolution des dépendances «non-résolues » au niveau de l'ADL reste généralement à la charge de l'infrastructure, qui est censée résoudre les dépendances durant le déploiement et ensuite durant l'exécution de l'application, au gré des évolutions des services et des ressources présentes. L'infrastructure peut utiliser les principes de services de nommage et de courtage évolués pour ce faire [117][105][48].

La capacité de définir des dépendances fonctionnelles non résolues est l'un des principes des architectures orientées service, qui préconisent un couplage faible entre des composants logiciels fournissant et recquérant des services. Ce couplage repose le plus souvent sur l'usage de Web Services (UDDI), accessibles directement au travers du Web ou via des infrastructures basées sur le standard SCA (Software Component Architecture, http ://osoa.org/display/Main/Home) telles que Tuscany (tuscany.apache.org) ou Frascati (frascati.ow2.org).

Dans tous les cas, la résolution tardive (au déploiement ou à l'exécution) des dépendances des composants implique que ceux-ci autorisent la manipulation de leurs dépendances. Cette capacité est classiquement fournie au travers de l'inversion de contrôle, un patron de conception des modèles à composant qui est décrit dans la section suivante.

2.2 L'inversion de contrôle

L'inversion de contrôle est un patron de conception majeur pour les infrastructures à composants. Ce patron externalise la gestion des dépendances, fournissant ainsi un moyen de les adapter depuis l'extérieur des composants [116]. Sa mise en œuvre peut prendre diverses formes et poursuivre différents objectifs en fonction des dépendances fonctionnelles ou non fonctionnelles que l'on souhaite adapter.

Cette distinction naturelle pour le développeur entre dépendances fonctionnelles et non fonctionnelles est le fil conducteur de notre présentation dans cette section. Après avoir présenté les principes de l'inversion de contrôle, nous nous concentrons sur son application à la gestion des dépendances non fonctionnelles (section 2.2.2), puis à la gestion des dépendances fonctionnelles (section 2.2.3). Pour chaque type de dépendance, nous présentons les principaux mécanismes utilisés pour mettre en œuvre cette inversion de contrôle, à savoir le tissage de code, l'injection de références, l'interception et les callbacks métier.

Notre présentation des mécanismes est associée à une évaluation qui porte d'une part sur les moments du cycle de vie (par exemple, déploiement, exécution) auxquels les dépendances peuvent être adaptées. Nous considérons également comme critère de qualité la transparence fournie au programmeur ainsi que la capacité pour celui-ci de spécialiser la gestion des dépendances. Il est en effet parfois nécessaire de spécialiser la gestion de certaines dépendances pour

prendre en compte les spécificités du code métier.

Enfin, le tissage de code, mécanisme majeur utilisé pour mettre en œuvre l'inversion de contrôle, établit une connexion forte entre l'inversion de contrôle et les techniques issues de la programmation par aspects (AOP / Aspects Oriented Programming) [30]. Nous y consacrons une section séparée (section 2.2.4).

2.2.1 Principe

Dans la section 2.1.2, nous avons vu que la description explicite de l'architecture d'une application au travers d'un ADL met en œuvre une séparation des définitions. Le code métier des composants est séparé de la définition de leur assemblage et de leurs dépendances vers des aspects fonctionnels ou non fonctionnels. Il est utile de conserver cette séparation des aspects à l'exécution, pour accroître l'indépendance entre un composant et les spécificités d'une infrastructure ou d'un environnement d'exécution donné. L'inversion de contrôle appliquée à la gestion des dépendances est un patron de conception qui permet de conserver cette séparation des aspects durant l'exécution. Ce patron permet ainsi d'externaliser la gestion des dépendances au niveau de l'infrastructure d'exécution ; c'est le principe tel que présenté dans [2] : "Ne nous appelez pas, c'est nous (l'infrastructure) qui vous appelons".

Autrement dit, ce ne sont pas les composants qui manipulent leurs dépendances, c'est l'infrastructure qui s'en charge, ce qu'illustre la figure 2.2 ci-dessous. L'infrastructure doit donc connaître les dépendances des composants, et la manière dont celles-ci sont résolues. Ces informations forment l'architecture logicielle de l'application. L'infrastructure peut conserver ces informations dans une structure pivot [103], ou bien les embarquer dans les composants lorsque ceux-ci ont une existence à l'exécution [29].

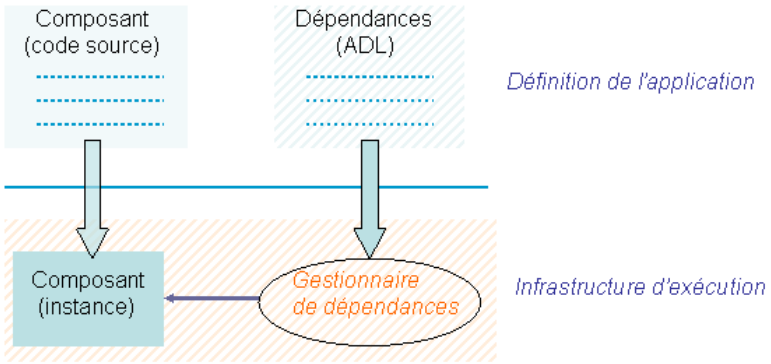

FIG. 2.2 – Principe de l'inversion de contrôle

Dès lors que l'architecture logicielle explicite les dépendances fonctionnelles et non fonctionnelles, l'inversion de contrôle permet de manipuler les deux types de dépendances. Cette vision de l'inversion de contrôle est plus générale que celle communément associée à ce terme, initialement défini par Martin Fowler dans [79], qui considère principalement les dépendances fonctionnelles.

Le principe d'inversion de contrôle est mis en œuvre dans la plupart des environnements existants et commercialisés actuellement, tels que JEE[68], Spring Framework[59][2], ou encore Pico

Container[1]. Spring Framework et Pico Container sont des canevas pour l'inversion de contrôle, qui fournissent des mécanismes de base au sein d'une infrastructure minimale et générique, initialement vide de services non fonctionnels, mais destinée à être spécialisée pour les besoins d'un domaine applicatif particulier. Ainsi, ce type de canevas peut être utilisé pour mettre en œuvre un environnement tel que JEE, qui utilise des mécanismes d'inversion de contrôle au sein d'une infrastructure définissant un ensemble de services non fonctionnels précis.

2.2.2 Le contrôle des dépendances non fonctionnelles

L'inversion de contrôle sur les dépendances non fonctionnelles consiste principalement, pour l'infrastructure, à prendre en charge les appels aux services non fonctionnels utilisés par les composants. Ces appels doivent avoir lieu aux moments opportuns de l'exécution des composants (par exemple, lors de leur création, ou bien lors d'une invocation de méthode métier). Les mécanismes utilisés classiquement pour ce type d'inversion sont le tissage de code, l'interception et les callbacks métier.

Le tissage de code Le tissage de code permet de manipuler le code d'un composant après sa programmation, afin d'y intégrer des aspects de contrôle. Il peut reposer sur l'usage de différentes techniques, permettant de transformer les programmes à différentes étapes du cycle de vie. On différencie principalement les phases avant/après la compilation, pendant le chargement ou encore durant l'exécution.

Le tissage peut ainsi s'appuyer sur l'usage d'outils permettant la modification de code source tel que Velocity (http ://velocity.apache.org/), utilisés généralement avant la compilation. Dans l'environnement Java, d'autres techniques se basent sur les compilateurs ouverts permettant de transformer les programmes juste avant leur compilation. Ainsi Spoon (http ://spoon.gforge.inria) permet de réaliser toutes sortes de transformation sur un programme source après que l'arbre syntaxique a été construit par le compilateur.

Le tissage de code peut également être basé sur la modification de bytecode dans le cas de composants Java au travers d'outils tels qu'ASM (http ://asm.ow2.org), invoqués lors de la compilation ou lors du chargement des classes. L'usage de modifieurs de bytecode permet d'adapter des composants pour lesquels on ne dispose pas forcément du code source. En revanche, la compréhension du code adapté est généralement difficile, car exprimé à un niveau d'abstraction bas (bytecode). Enfin, le tissage peut également avoir lieu durant l'exécution, en se basant sur l'usage de techniques issues de la programmation orientée aspect (AOP) dynamique, comme cela a été fait dans A-TOS [107], ou encore dans JAC [106] qui propose la notion de tissage dynamique pour adapter au cours de l'exécution une application en ajoutant ou en retirant des aspects. Nous revenons plus largement sur le positionnement des techniques de l'AOP par rapport à l'inversion de contrôle dans la section 2.2.4.

Pour le contrôle des dépendances non fonctionnelles, le tissage de code peut être utilisé pour deux objectifs. D'une part, il fournit un moyen d'adapter la gestion du cycle de vie des composants en fonction de leurs dépendances non fonctionnelles. Considérons une dépendance non fonctionnelle liant un composant C à un service non fonctionnel S. Si les changements d'états du cycle de vie de C (création, activation, etc) doivent générer des appels vers S, ces appels peuvent être introduits dans C par tissage.

D'autre part, un tissage de code additionnel peut être nécessaire dans C pour fournir l'interface requise par le service S (voir figure 2.3). Plus précisément, le tissage du contrat requis par une dépendance consiste à ajouter une interface et son implémentation à un composant, dans le but de satisfaire le contrat requis par cette dépendance. Par exemple, ce mécanisme peut ajouter à un composant persistant, les méthodes requises par un service de persistance, telles que *setField()* et *getField()*, permettant de manipuler ses attributs persistants.

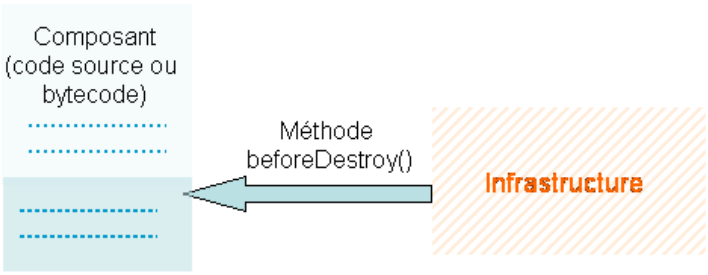

FIG. 2.3 – Tissage du contrat requis par le service S dans un composant

L'interface tissée relève d'un contrat qui est spécifique au service non fonctionnel utilisé. Certaines informations sur le composant peuvent être nécessaires pour pouvoir générer l'implémentation des méthodes correspondant à l'interface tissée (par exemple, la liste des attributs persistants pour un service de persistance). Ces informations sont en général fournies par le programmeur, sous la forme d'annotations [60][68] ou bien en appliquant certaines règles de programmation (par exemple, en supposant que les attributs persistants sont ceux associés à des méthodes de type *setter/getter* définies par le programmeur [68]).

Les règles de programmation sont souvent spécifiques à une implantation donnée d'un service non fonctionnel. Elles influent sur la programmation et doivent donc être connues au moment du développement. Le développeur doit ainsi savoir quelles propriétés non fonctionnelles seront associées aux composants et quelles implantations de ces services seront utilisées.

Les annotations fournissent une meilleure indépendance car elles peuvent éventuellement être ajoutées ou modifiées après la phase de développement. La tendance est de définir ces annotations selon des standards (par exemple, le standard JPA pour la persistance [99], JTA pour les transactions [98]) pour augmenter encore le niveau d'indépendance.

Définir les composants de manière plus indépendante des services utilisés est nécessaire non seulement pour la portabilité des composants mais aussi pour pouvoir adapter dynamiquement les dépendances non fonctionnelles. Durant la phase d'administration, le choix de basculer d'un service de persistance à un autre est alors envisageable. Un tel choix peut être requis pour suivre des évolutions de versions par exemple, ou bien pour s'adapter à des conditions d'exécutions changeantes (par exemple, le passage à un environnement contraint de type PDA).

Les callbacks métier Les *callbacks métier* permettent à l'infrastructure d'émettre des événements vers les composants qui, en retour, exécutent des méthodes métier appelées callbacks. Par ce biais, le développeur peut spécialiser les traitements effectués lors de l'utilisation d'un service non fonctionnel, en associant des callbacks à des événements émis par l'infrastructure.

Par exemple, JEE permet d'ajouter des traitements avant et après l'exécution des méthodes de sauvegarde et de restauration mises en œuvre par le service de persistance. Ainsi, *ejbStore()* et *ejbLoad()* sont des callbacks respectivement exécutées avant une sauvegarde et après un chargement depuis le support de persistance, qui permettent d'effectuer des traitements spécifiques au code métier tel que par exemple la transformation de données non sérialisables ou la vérification de contraintes sur l'état du composant (voir listing 2.3).

Listing 2.3 – Exemple de spécialisation d'un service non fonctionnel

```
// Client component
public class Client {
   ...
   // executed after component is loaded
   public void ejbload(){
   ...
   }
   // executed before the component is stored on DB
   // verify its account number
   public void ejbStore(long id) {
    if (account.length() != 12)
   throw new IllegalArgumentException("Invalid account number");
   ...
}
```

Afin d'assouplir plus encore le contrat entre un composant et l'infrastructure, les méthodes représentant des callbacks métier peuvent être identifiées au travers d'annotations (@postLoad), au lieu d'utiliser une convention de nommage (ejbLoad()). Le tableau ci-après liste les annotations fournies par un service de persistance satisfaisant le standard JPA (Java Persistence API).

Annotation	**Evénement associé**
@PrePersist,@PostPersist	Avant et après la création du composant dans la BD
@PostLoad	Après le chargement du composant depuis la BD
@PreUpdate, @PostUpdate	Avant et après la mise à jour des attributs du composant
@PreRemove, @PostRemove	Avant et après la destruction du composant

L'interception En dehors du tissage de code et des callbacks applicatives, l'autre mécanisme utilisé pour l'inversion de contrôle sur les dépendances non fonctionnelles est le principe d'interception. Ce mécanisme consiste à injecter, au niveau de la chaîne de liaison établie entre deux composants, des objets appelés intercepteurs qui vont effectuer un traitement de part et d'autre de la chaîne de liaison. Cela permet de contrôler les invocations de méthodes métier qu'un composant effectue sur les autres composants de l'application. L'infrastructure peut utiliser ce mécanisme pour gérer les interactions avec les services non fonctionnels qui doivent avoir lieu lors de ces invocations. Un service transactionnel sera par exemple sollicité lors de chaque appel de méthode métier transactionnelle.

L'interception d'une invocation peut reposer sur l'utilisation d'une machine virtuelle spécifique [6], sur la modification du code des composants au moment de leur compilation ou de leur chargement [54], ou enfin sur la modification du code source des composants par un préprocesseur [29]. Un exemple connu d'usage des intercepteurs se trouve dans le Portable Object

Adaptor (POA) de Corba [51][50]. Les intercepteurs, qui ont généralement une existence à l'exécution sous la forme d'objets, sont instanciés à partir de classes générées par l'infrastructure, au moment de la compilation, ou du chargement. Le passage par ces objets a naturellement un surcoût à l'exécution, qui reste cependant acceptable dans le cadre de l'usage qui en est fait dans la plupart des applications réparties [19].

La présence des intercepteurs peut être rendue transparente pour le développeur de l'application répartie, qui utilise alors des fabriques pour créer ses composants. La transparence est également fournie au client d'un composant, les intercepteurs d'appels entrants implémentant les interfaces des composants qu'ils invoquent. En conséquence de cette transparence, il n'est pas toujours possible de spécialiser les traitements effectués au niveau des intercepteurs. En outre, les intercepteurs sont des objets qui, une fois insérés dans la chaîne de liaison liant deux composants, sont souvent non contrôlables à l'exécution dans les modèles non réflexifs. Il n'est pas possible de remplacer un intercepteur par un autre dans une infrastructure telle que JEE.

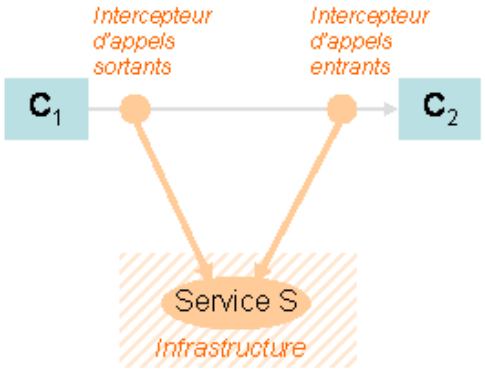

FIG. 2.4 – Intercepteurs d'invocations de méthodes métier

2.2.3 Le contrôle des dépendances fonctionnelles

L'inversion de contrôle sur les dépendances fonctionnelles consiste principalement à déléguer à l'infrastructure la charge d'assembler à l'exécution les composants d'une application. Sa mise en œuvre repose sur trois mécanismes principaux : l'injection de références, le tissage de code et les callbacks métier.

L'injection de références définit le comportement suivant. Lorsqu'un composant C1 doit être lié à un service métier fourni par un composant C2, l'infrastructure se charge d'injecter la référence du service de C2 dans le ou les attributs correspondant de C1. Cette injection, illustrée ci-dessous, est dynamique puisqu'elle nécessite de connaître les identités des composants à lier ainsi que les services qu'ils offrent ou dont ils ont besoin.

L'injection de références touche le cycle de vie des composants, formalisant des patrons existant dans la programmation par objet. En ce qui concerne le cycle de vie, il est habituel de considérer les étapes de création, construction et de destruction des objets. Dans la plupart des langages objets, la création et la construction sont souvent une seule et même phase ; l'objet

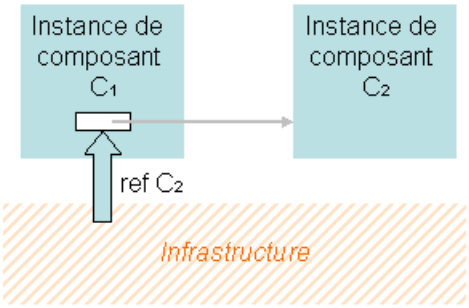

FIG. 2.5 – Injection d'une dépendance vers C2 dans C1

est considéré comme construit dès la fin de l'exécution de son constructeur. Bien que cela soit souvent le cas, il est parfois nécessaire d'injecter des références après la construction d'un objet avant qu'il ne soit entièrement fonctionnel. C'est notamment le cas lorsque l'on construit des structures d'objets avec des cycles.

Il est nécessaire de formaliser ces étapes lorsque l'on considère l'inversion de contrôle puisque l'assemblage des composants n'est plus fait par le développeur des composants mais par l'infrastructure suivant la définition de l'assemblage fournie par l'ADL. On distingue classiquement deux patrons de mise en œuvre pour l'injection de références [59] : l'injection via le constructeur et l'injection par appel de méthodes. Dans les deux cas, le code nécessaire (constructeur ou méthode) peut être injecté dans un composant par tissage de code.

L'injection via le constructeur d'un composant correspond à l'approche la plus proche de la notion de constructeur d'objets. Ce patron consiste à injecter la ou les références nécessaires au moment de l'instanciation d'un composant, en surchargeant, par tissage de code, le constructeur de ce composant. Ce scénario est utilisé dans JEE [68] pour les dépendances fonctionnelles. Une fois affectée, une dépendance fonctionnelle n'est plus modifiable au cours de la vie d'un composant.

L'injection par appel de méthodes permettra au contraire à l'infrastructure de manipuler les dépendances d'un composant plusieurs fois durant sa vie [59]. Les méthodes utilisées pour injecter les dépendances sont de type *setter* et *getter* et permettent l'injection dynamique de références dans le composant. Cette approche nécessite d'informer le composant du moment où ses dépendances sont toutes injectées. Cet aspect est traité au moyen du mécanisme de callbacks métiers, qui permettent à l'infrastructure d'exhiber les événements liés au cycle de vie des composants. Le cycle de vie peut être élémentaire (créé, détruit) ou plus complexe (comprenant des états d'activation et de passivation [68]). L'exemple ci-dessous illustre l'émission de l'événement *afterCreate*, envoyé par l'infrastructure à un composant juste après que sa création et que l'injection de ses références soient terminées.

Par le mécanisme de callbacks, le développeur peut spécialiser les traitements effectués lors des étapes du cycle de vie. Il peut ainsi associer une initialisation métier lors de l'injection de la référence d'un service.

Un exemple connu est le conteneur Enterprise Java Beans (EJB) dans l'environnement JEE.

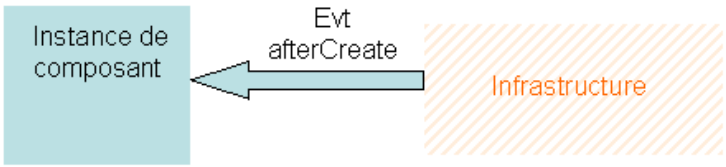

FIG. 2.6 – Exhibition des événements liés au cycle de vie des composants

Ce conteneur propose un cycle de vie complet qui illustre bien l'adjonction d'un comportement spécialisable par le développeur. Il y a principalement quatre étapes du cycle de vie qui sont spécialisables : la création, la destruction, l'activation et la passivation d'un composant. Comme schématisé dans le listing 2.4, la création d'un composant donne lieu à l'appel, par l'infrastructure, de la callback *ejbCreate()* qui permet de finaliser la construction d'un composant une fois qu'il a toutes ses références injectées, avant sa sauvegarde sur un support persistent. La destruction d'un composant est précédée de l'appel à la callback *ejbRemove()* qui permet, comme la méthode *finalize()* en Java pour un objet, de finaliser la destruction d'un composant. Le déchargement et rechargement d'un composant dans le cache mémoire donnent respectivement lieu à l'appel des callbacks *ejbActivate()* et *ejbPassivate()*.

Listing 2.4 – Spécialisation des méthodes du cycle de vie d'un composant dans JEE

```
// Client component
public class Client{
    ...
    void ejbCreate() {...}
    void ejbRemove() {...}
    void ejbActivate() {...}
    void ejbPassivate() {...}
}
```

Au delà du cycle de vie, l'inversion de contrôle sur les dépendances fonctionnelles est totalement transparente pour le développeur. En effet, une référence vers un service d'un composant applicatif peut être utilisée de manière classique pour invoquer une méthode métier.

2.2.4 Le tissage de code et la programmation par aspects

Comme nous l'avons précédemment introduit, le tissage de code permet de manipuler le code d'un composant après sa programmation, afin d'y intégrer des aspects de contrôle. Une possibilité pour mettre en œuvre ce tissage consiste à utiliser la programmation par aspects, ou Aspect Oriented Programming (AOP) [30]. L'AOP est née du constat que si la programmation par objets fournit de bons résultats pour la modélisation des fonctionnalités métier, elle échoue par contre à bien prendre en compte les fonctionnalités transversales, en particulier les dépendances non fonctionnelles. En effet, la mise en œuvre de ces dépendances sans support particulier introduit un phénomène de dispersion (en anglais, scattering), ce qui nuit fortement à l'évolutivité, à la maintenance et à l'efficacité du code. On retrouve aussi un phénomène d'entremélage de code (en anglais tangling) qui concerne des préoccupations diverses. Ces deux phénomènes sont le point de départ du constat dont est issue la programmation par aspects, apparue au milieu des 1990

et issue des travaux menés au Xerox PARC dans l'équipe de Gregor Kiczales.

Les concepts de l'AOP sont stables, à quelques variations près, et acceptés par l'ensemble de la communauté. Un aspect est une entité logicielle qui capture une fonctionnalité transversale à une application. Le code d'un aspect est en fait tissé en des points des classes métiers appelés points de jonction. Un point de jonction (en anglais *join point*) est un point dans l'exécution d'un programme autour duquel un ou plusieurs aspects peuvent être ajoutés. Un aspect est en fait tissé sur un ensemble de points de jonction, que l'on appelle une coupe (en anglais *pointcut*). La coupe est définie avec un langage de patterns qui capture où le tissage doit survenir, alors que le code des aspects capture ce qui doit être tissé. Une coupe fait donc partie de la définition d'un aspect alors qu'un point de jonction est un point dans l'exécution d'un programme. Un même point de jonction peut d'ailleurs appartenir à plusieurs coupes. Le code d'un aspect s'exprime via le concept de code advice.

Au delà du monde des langages, les aspects ont été rapidement adoptés par la communauté des intergiciels afin de profiter d'une adaptabilité accrue. Les intergiciels bénéficient à la fois des approches statiques et dynamiques de la programmation orientée aspect. Les approches statiques, dont celle d'AspectJ [45], fusionnent à la compilation le code des aspects à celui des classes métiers. Pour une adaptabilité dynamique, la seule solution avec une approche statique est donc d'arrêter la plate-forme de l'intergiciel pour retisser des nouveaux aspects ou les tisser en d'autres points. Plus intéressante est l'approche dynamique de l'AOP, apparue vers 1998, qui permet la liaison dynamique entre le code des aspects et le code des classes. Quatre canevas principaux sont JAC [107], AspectWerkz (aspectwerkz.codehaus.org), JBoss AOP (http ://www.jboss.org/jbossaop/) et Spring AOP (static.springframework.org).

Cette démarche de l'AOP ne remet en cause ni l'inversion de contrôle, ni la programmation objet, mais suggère de voir l'AOP comme une modélisation et un outil qui peut simplifier la mise en œuvre de l'inversion de contrôle. Les aspects fournissent en particulier un support systématique pour le tissage de code, permettant de tisser/détisser du code de manière contrôlée. Inversement, pour les supports d'AOP, la structuration en composants d'une application facilite l'association des aspects aux objets et par suite, la maintenance de ces aspects. Des projets comme [89] sont représentatifs de cette synergie.

2.2.5 Bilan sur la mise en oeuvre de l'inversion de contrôle

L'inversion de contrôle reste le pilier central des modèles à composants et il est important de récapituler les principes de sa mise en œuvre dans les environnements à composants considérés.

Nous considérons comme cas d'exemple un composant C1 qui possède une dépendance fonctionnelle D1 vers un composant C2 ainsi qu'une dépendance non fonctionnelle D2 vers un service (par exemple, de persistance) fourni par l'infrastructure, tel qu'illustré dans la figure 2.7.

Dans le tableau ci-après, la colonne intitulée Tissage résume les éléments tissés dans les composants pour la gestion de ces dépendances. La colonne Interactions exprime quelles sont, à l'exécution, les interactions avec C1 initiées par l'infrastructure pour la gestion des dépendances D1 et D2. Enfin, la colonne Spécialisation résume les possibilités dont dispose le programmeur pour spécialiser le traitement effectué par l'infrastructure par rapport à la gestion de ces dépendances.

Dépendance	Tissage	Interactions	Spécialisation
D1 (fonctionnelle)	Tissage dans C1 du constructeur étendu ou du getter/setter	Injection de référence via l'invocation de méthodes tissées Envoi d'événements liés au cycle de vie de C	Callbacks métier
D2 (non fonctionnelle)	Tissage dans C1 de : - contrat requis par D2 - cycle de vie adapté à l'usage de D2 - intercepteurs au niveau de D1	Envoi d'événements liés à l'usage de D2	Callbacks métier

Pour gérer la dépendance fonctionnelle D1, l'infrastructure tisse tout d'abord dans le code de C1, les méthodes permettant de mettre à jour la référence vers C2 (si ces méthodes ne sont pas déjà fournies par le programmeur par des contraintes portant sur la programmation des composants). Ces méthodes peuvent correspondre au constructeur de C1 étendu, ou à des *setter/getter* de la référence vers C2. À l'exécution, l'infrastructure invoque ces méthodes pour injecter la référence vers C2. Elle émet des événements pour informer C1 de son cycle de vie, par exemple pour l'informer du fait que toutes ses références ont été injectées. Le programmeur peut associer des callbacks métier à ces événements pour spécialiser la gestion de son cycle de vie.

Une fois injectée, la dépendance vers C2 peut à nouveau être adaptée durant l'exécution si le tissage est basé sur des méthodes de type *setter/getter*. Notons cependant que, même si ces méthodes fournissent la mécanique permettant d'adapter à nouveau une dépendance en cours d'exécution, cette mécanique n'est exploitable que si l'infrastructure connaît l'ensemble des contraintes associées à la dépendance ainsi que son état dynamique. Rappelons que les contraintes ont été exprimées dans l'ADL, et qu'il n'est pas évident que l'architecture logicielle présente à l'exécution connaisse ces informations.

La gestion de la dépendance non fonctionnelle D2 repose sur le tissage dans C1 du contrat requis par le service de persistance, ainsi que sur l'adaptation de la gestion du cycle de vie de C1 à l'usage de ce service. En particulier, l'infrastructure génère les appels au service de persistance qui doivent avoir lieu lors de certaines transitions dans le cycle de vie de C1. De la même manière, en utilisant des intercepteurs, l'infrastructure tisse les appels au service de persistance qui doivent avoir lieu lorsque C1 effectue des appels de méthode métier vers des composants applicatifs (C2 dans l'exemple).

Le programmeur a la possibilité de spécialiser le traitement effectué pour la gestion de la dépendance non fonctionnelle D2, en associant des callbacks métier à des événements émis par l'infrastructure liés à l'usage du service de persistance. Par exemple, le programmeur peut définir

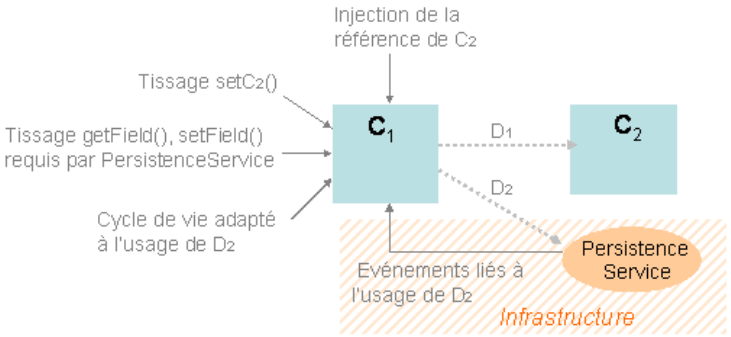

FIG. 2.7 – Scenario récapitulatif pour la gestion des dépendances

une callback activée après chaque mise à jour du composant sur le support de persistance (cf. annotation @PostUpdate dans la figure 2.7).

Avec les mécanismes décrits, la dépendance de C1 vers le service de persistance est mise en place au moment de la compilation et du chargement. Le cycle de vie de C1 est en effet étendu par tissage pour inclure les appels vers le service de persistance. Après ces phases, cette dépendance n'est plus adaptable en l'absence de mécanisme additionnel prévu pour ce faire. L'un des points critiques par rapport à cette possibilité d'adaptation est, comme pour les dépendances fonctionnelles, la connaissance, de la part de l'infrastructure, des caractéristiques associées à la dépendance (par exemple, les critères de qualité de service attendus).

2.3 Le modèle à composants Olan

2.3.1 Objectifs

Durant les années 1995 à 2000, mes idées et mon travail de recherche autour des composants se sont matérialisés via le projet Olan [103][71][72]. Ce projet avait pour objectif de concevoir et mettre en œuvre une infrastructure destinée à la construction, l'exécution et l'administration d'applications réparties mettant en jeu de multiples utilisateurs travaillant de manière coopérative. Des exemples typiques de telles applications sont les téléconférences, les agendas électroniques, les environnements de développements intégrés, ou encore les applications de workflow.

L'approche adoptée par Olan, et suivie par un nombre relativement faible de projets à l'époque dans le contexte des applications réparties [62][101], était d'expliciter l'architecture logicielle des applications pour produire un modèle permettant de contrôler leur assemblage à l'exécution. Olan était plus précisément basé sur la programmation constructive [39], dénommée aussi programmation par composition de logiciels, qui se focalise sur la notion d'architecture des applications. Celle-ci permet de séparer la mise en œuvre des applications en deux niveaux distincts : le premier niveau correspond à la mise en œuvre des briques logicielles de base par les programmeurs. Le second niveau correspond à l'assemblage et à l'intégration de ces briques de base pour former l'application.

Olan définit donc un modèle de composant qui permet de construire des applications par assemblage de briques logicielles patrimoniales. Par principe, ces briques ont été programmées :

(1) plus ou moins indépendamment les unes des autres, (2) indépendamment d'un environnement d'exécution donné et (3) indépendamment d'une configuration répartie donnée.

L'assemblage des briques logicielles formant une application est décrit dans un ADL nommé OCL (Olan Configuration Language). Ce langage exhibe l'architecture d'une application sous la forme de composants et de connecteurs. Les composants correspondent aux briques logicielles définissant le code métier de l'application. Les connecteurs sont des liaisons spécialisables mettant en œuvre un protocole d'échange décrit par un ensemble de propriétés. Par exemple, le connecteur *syncCall* met en œuvre une communication synchrone, qui peut être associée à des opérations de transformations du flot de données échangé entre les composants liés.

Conjointement à l'ADL d'Olan, deux autres langages de même type apparaissent à cette époque dans le contexte des applications réparties : Polilyth [101] et Conic [62]. L'approche proposée par Polilyth sera par la suite étendue au travers du langage Darwin [63][64] pour permettre de décrire une application comme une hiérarchie de composants interconnectés au lieu d'une collection plate de modules. Dans ce contexte, les contributions majeures d'Olan se situent sur la prise en compte du code patrimonial, l'adaptabilité des connecteurs et la prise en charge du déploiement des applications.

La part importante de code patrimonial dans le domaine des applications réparties nous incite à l'époque à fournir un mécanisme d'encapsulation qui permet de représenter ce code par des composants Olan devenant interconnectables avec d'autres. Cette réutilisation de code patrimonial implique de pouvoir adapter les connecteurs aux contraintes et aux spécificités des composants qu'ils lient. L'adaptabilité des connecteurs repose sur un découplage fort entre l'implémentation des composants et la mise en œuvre de leurs interconnexions. Ces dernières sont décrites de manière déclarative, en termes de protocole de synchronisation, de type de communication et de transformation du flot de données échangé.

L'adaptabilité des connecteurs est exploitée par Olan durant la phase de déploiement d'une application. Etant donné la description de l'architecture d'une application, Olan adapte cette architecture à la configuration répartie de l'environnement d'exécution courant (par exemple, projection des composants vers les machines), avant de piloter le déploiement des composants et des connecteurs dans cet environnement.

2.3.2 Le cycle de vie du modèle Olan

La définition d'une application constitue la première phase du cycle de vie Olan. Cette phase met en jeu les deux étapes suivantes :

- La description de l'assemblage des composants utilisés, dans le langage OCL. Cette étape offre une vue architecturale de l'application qui sert de canevas aux phases suivantes.
- L'implémentation des composants. Cette étape peut être effectuée de manière quasi-indépendante de l'application à laquelle les composants vont participer. Ceci facilite la réutilisation du composant dans une autre application.

Une fois ces deux phases réalisées, la description OCL de l'application est compilée dans un environnement d'exécution donné pour générer les classes appropriées de composants et de connecteurs. Le compilateur Olan génère également un script de déploiement qui permet d'effectuer le déploiement initial de l'application dans l'environnement d'exécution, au travers de l'instanciation des objets composants et connecteurs. L'exécution d'une application Olan est

supportée par une infrastructure d'exécution implémentée dans le langage Python au dessus de l'ORB Xerox/ILU [15].

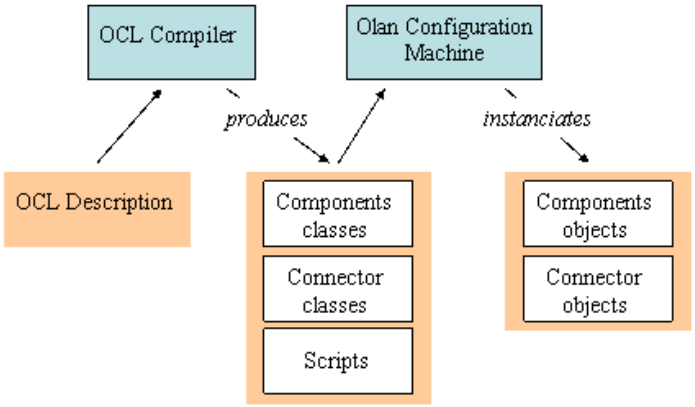

FIG. 2.8 – Cycle de vie d'OLAN

Dans ce cycle de vie, l'adaptabilité fournie par Olan a lieu au moment du déploiement et porte sur trois aspects. D'une part, Olan adapte l'architecture de l'application à l'environnement d'exécution courant, lors de la phase de déploiement. L'architecture est plaquée sur une configuration répartie qui tient compte des machines disponibles et des contraintes de déploiement des composants.

D'autre part, dans cette architecture, Olan adapte les connecteurs aux contraintes des composants qu'ils lient, en agissant sur les flots de données et de contrôle. En particulier, si l'environnement d'exécution le permet, les composants co-localisés seront liés par des connecteurs ne passant pas par les couches réseau.

Enfin, l'infrastructure adapte la mise en œuvre des connecteurs aux services fournis par l'environnement d'exécution. Par exemple, une communication entre des composants hétérogènes pourra être mise en œuvre sur un ORB si l'environnement en fournit un, ou bien directement au dessus de la couche socket.

2.3.3 Le langage OCL

Le langage OCL (Olan Configuration Language), initialement inspiré de l'ADL Darwin, permet de décrire une application comme une hiérarchie de composants interconnectés. À chaque niveau de la hiérarchie correspond une classe de composant, qui encapsule les composants définis au niveau inférieur. Les feuilles de la hiérarchie sont des composants primitifs, qui encapsulent du code patrimonial.

Composants Tout composant est décrit par une interface qui exhibe ses dépendances fonctionnelles avec le monde extérieur, ainsi que par une implémentation qui met en œuvre cette interface. Une interface est exprimée dans le langage de définition d'interface OIL (Olan Interface Language). Une implémentation est programmée dans l'un des langages supportés par Olan (C, C++, Python, Java).

L'interface d'un composant définit les services qu'il fournit aux autres composants, ainsi que ceux qu'il requiert pour son exécution. Les services sont décrits en termes de leur signature et de leur type d'invocation. Ceux devant être invoqués de manière synchrone (étiquettes *Provide* et *Require* dans le tableau ci-après), sont distingués de ceux pouvant être invoqués de manière asynchrone (étiquettes *Notify* et *React*). De plus, un service de type *React* ne peut recevoir que des paramètres envoyés par valeur.

Services	**Flot de données**	**Flot de contrôle**
Provide	in/out	Pas de contrainte particulière
Require	in/out	L'appelant fait un appel synchrone
React	in	Pas de contrainte particulière
Notify	in	L'appelant fait un appel asynchrone

Un exemple simplifié de définition d'interface est présenté ci-après. Il est inspiré d'une expérience de construction d'application répartie de gestion de documents Web avec Olan. L'interface donnée dans le listing 2.5 reflète les notifications et les services fournis par un module implémenté en Python gérant une widget *listbox*.

Listing 2.5 – Exemple de définition d'interface dans Olan

```
typedef sequenxe <*, char> seq ;

   interface listBoxItf {
      provide init() ;
      provide addEntry(in seq EntryContent) ; // add an entry
      provide remEntry(in seq EntryContent) ; // remove an entry
      ..
      notify userAddEntry(); // raised when a user adds an entry
      notify userRemEntry(); // raised when a user removes an entry
      notify userSelectEntry(); // raised when a user selects an entry
      ..
}
```

L'implémentation d'un composant peut être composite ou bien primitive. Une implémentation primitive (voir listing 2.6) encapsule un module logiciel d'un certain type et programmé dans l'un des langages patrimoniaux supportés par Olan (programme source C, module C, librairie, classe Java, exécutable, etc.). Chaque type est associé à un patron d'utilisation mis en œuvre par un composant particulier appelé *wrapper*, qui définit comment utiliser le module encapsulé. Par exemple, un wrapper de librairie appellera directement une fonction fournie par la librairie alors qu'un wrapper de classe instanciera la classe avant d'utiliser ses méthodes. L'implémentation d'un composant primitif définit la localisation des fichiers définissant le module. Ci-après se trouve la définition du composant primitif mettant en œuvre l'interface *listBoxItf*.

Listing 2.6 – Exemple d'implémentation primitive dans Olan

```
// Primitive component ListBoxMod
module « Python » listBoxMod : listBoxItf {
   path : « ${OLAN_SRC} » +  « /examples/URL » ;
   sourceFile : "listBoxMod.py";
   ...
};
```

Les composites sont les unités de structuration et de hiérarchisation d'une application. Une implémentation composite est formée d'une hiérarchie de sous-composants interconnectés, décrite dans le langage OCL. Nous donnons ci-après (voir listing 2.7) un exemple de composant composite (*URLNotifier*) composé de trois sous-composants, connectés au travers du connecteur de type *syncCall* décrit dans la section suivante.

Listing 2.7 – Exemple d'implémentation composite dans Olan

```
// Composite component URLNotifier
implementation URLNotifier uses ListBoxItf, URLMgrItf, URLEditorItf {

   // Sub-components
   URLMgr    = instance URLMgrItf; // List of URL of interest
   ListBox   = instance ListBoxItf; // User Itf
   URLEditor = instance URLEditorItf; //e.g., Netscape browser

   // Interconnections between sub-components
   startURLNotifier()         => URLMgrItf.init() using syncCall() ;
   URLMgr.initGUI()           => ListBoxItf.init()using syncCall() ;
   ListBox.userAddEntry(URL) => URLMgr.addURL(URL) using syncCall() ;
   URLMgr.editURL(URL)        => URLEditor.edit(URL) using syncCall() ;
   ...
};
```

Connecteurs Les connecteurs sont des abstractions du langage OCL qui lient les composants dans une architecture répartie. Ils définissent les types de données échangés, le modèle d'exécution et le protocole de communication.

La table donnée ci-après résume les principaux connecteurs mis à la disposition des architectes, et leur implémentation dans l'infrastructure de base fournie par Olan. Dans cet environnement, les connecteurs synchrones et ceux asynchrones utilisent l'ORB ILU [15] lorsque la communication est distante. Dans un environnement différent, il serait possible de redéfinir l'implantation de ces connecteurs pour utiliser un autre support de communication répartie.

Les connecteurs ont pour rôle d'adapter les communications aux caractéristiques des composants qu'ils lient. Certaines adaptations sont automatiquement générées par le compilateur OCL. Par exemple, le flot de contrôle est adapté (par la création d'un *thread*) lorsque le composant appelant demande une invocation de service asynchrone alors que l'appelé requiert une invocation de service synchrone.

Dans certains cas, le compilateur génère également l'adaptation du flot de données (par exemple, pour passer un paramètre d'un type String à une séquence de caractères). Le langage OIL définit en effet son propre typage de données, permettant de construire une représentation pivot des données transmises ou reçues lors d'une communication. Au moment du déploiement, les traducteurs appropriés (vers les langages C, C++, Java et Python) sont placés dans les connecteurs, ainsi que les convertisseurs permettant de lier des services dont les types ne sont pas directement compatibles.

Nom	**Appelant**	**Appelé**	**Protocole**	**Implémentation**
syncCall 1->1	require	provide	synchrone	Local : appel de méthode IPC : appel de méthode ILU Distant : appel de méthode ILU
asyncCall 1->1	notify	react	asynchrone	Local : appel de méthode asynchrone IPC : appel de méthode asynchrone ILU Distant : appel de méthode asynchrone ILU
asyncToSyncCall 1->1	notify	provide	asynchrone	id. syncCall avec création de thread
RandSyncCall 1->n	require	provide	synchrone	id. syncCall

Dans d'autres cas, l'architecte doit préciser les conversions à appliquer en utilisant un ensemble d'opérateurs prédéfinis par OCL. Dans l'exemple illustré dans le listing 2.8, le compilateur n'autorise la liaison que si l'architecte spécifie les transformations de données à appliquer.

Intégration d'aspects dynamiques Les concepts présentés précédemment permettent de définir l'architecture initiale d'une application, c'est-à-dire celle construite au moment du déploiement. Il est nécessaire de permettre l'évolution dynamique de cette architecture, car les applications visées mettent en jeu des composants et des interconnexions qui varient au fil du temps. Nous avons fourni les fonctions dynamiques suivantes : l'instanciation paresseuse de composants (initialement proposée par Darwin [63]), et les collections.

L'instanciation paresseuse permet de n'instancier un composant qu'au moment où l'un de ses services fournis est invoqué. Ceci permet essentiellement de limiter le nombre de composants déployés durant la vie d'une application.

Listing 2.8 – Exemple d'adaptation du flot de données dans un connecteur

```
implementation URLNotifierImpl .. {
   ...
   ListBox.addEntry(URLDesc) => URLMgr.addURL(URL, pollingPeriod)
      using syncCall()
      do {
         URLDesc = URLDesc.split(' ') ;
         URL = URLDesc[0] ;
         pollingPeriod = URLDesc[1] ;
      } ;
   ...
}
```

Les collections permettent de faire évoluer le contenu d'un composite durant l'exécution, en termes des sous-composants qu'il contient. Une collection est un ensemble de composants de même type dont la cardinalité à l'exécution varie entre une valeur minimale et une valeur maximale données. L'accès aux membres d'une collection peut être effectué en utilisant un service de désignation associative, qui permet de sélectionner les membres qui satisfont certaines propriétés (localisation, valeurs d'attributs, etc.). Ce dernier aspect a été inspiré de [25]. Dans l'exemple donné dans le listing 2.9, une collection est utilisée pour instancier un composant de type *URLPoller* par URL à poller.

Listing 2.9 – Exemple d'usage des collections dans Olan

```
implementation URLMgr ... {
   URLPollerColl = collection URLPoller ;

   // create a new URLPoller instance for each new URL of interest
   addURL(URL, pollingPeriod) =>
      URLPollerColl.addURL(URL, pollingPeriod)
      using createInCollection();

   // change polling period of a particular URLPoller within the
       collection
   changePollingPeriod(URL, period) =>
      URLPollerColl.changePollingperiod(period)
      where URLPollColl.URL == URL
      using aSyncCall();

   // notify all URLPoller instances when the application terminates
   close() => URLPollerColl.close()
      using broadcastSyncCall();
}
```

Intégration des aspects liés à la répartition Les contraintes liées à la répartition sont décrites pour chaque composant au travers de règles portant sur des attributs particuliers appelés *attributs d'administration*. Un attribut d'administration caractérise une ressource de type machine ou utilisateur, comme illustré dans le listing 2.10.

Listing 2.10 – Attributs liés à la gestion de la configuration distribuée dans Olan

```
management attribute Node
    String name ;      // DNS name
    String ipAdr ;     // IP address
    String osType ;    // OS type
    ...
}
management attribute Owner {
    string name ;              // User name
    long uid ;                 // User id
    sequence<long> grpId ;     // User gid
}
```

Les règles décrivant le placement des composants sont formées de prédicats portant sur les attributs d'administration. L'architecte a par exemple la possibilité de co-localiser les composants qui communiquent beaucoup. Dans l'exemple donné dans le listing 2.11, le composant chargé de gérer les URL (*URLMgr*) doit être co-localisé avec celui-gérant l'interface utilisateur (*ListBox*).

Listing 2.11 – Expression de contraintes sur la configuration distribuée dans Olan

```
management URLNotifierMgt : URLNotifierImpl {
 Node.name = « *.inrialpes.fr » ; // all components in domain inrialpes
 URLMgr.Node.name == ListBox.Node.name; // co-locate these components
}
```

2.3.4 L'infrastructure Olan

L'infrastructure Olan est composée d'un ensemble de services réalisant le déploiement et l'exécution des composants et des connecteurs. Un code patrimonial encapsulé dans un composant ne s'exécute cependant pas sur cette infrastructure ; il s'exécute directement sur son propre support d'exécution patrimonial qui est censé être déjà opérationnel ou bien être créé et initialisé dans un composant *wrapper*.

Le service de déploiement fournit les fonctions permettant de créer les composants, les connecteurs, et de lier les composants aux connecteurs. Pour chaque composant décrit dans l'architecture OCL, le service de déploiement cherche une machine correspondant aux contraintes du composant, et capable d'héberger un support d'exécution pour le composant.

Toute entité du modèle Olan (composant, connecteur, collection) est représentée par un ou plusieurs objets à l'exécution. La mise en œuvre des connecteurs utilise deux types d'objets : les adaptateurs de services et les objets de type envoi/réception. Les adaptateurs réalisent les conversions de données générées par le compilateur et/ou spécifiées par l'architecte dans la description OCL. Les objets d'envoi et de réception gèrent le flot de contrôle et les communications effectives entre les composants connectés. Certains objets ne sont présents à l'exécution que si les composants sont distants.

Les objets représentant les composants jouent le rôle d'*intercepteurs*. Ils interceptent les appels entrants et sortants pour les rediriger vers les composants patrimoniaux en fonction de

tables d'indirection appelées *inCall* et *outCall*. Ainsi, comme représenté dans la figure 2.9, tout appel entrant est dirigé via la table *inCall* vers le module qui implémente le service demandé, en passant par un talon qui effectue les conversions nécessaires sur le flot de données entrant, et par un *wrapper* qui encapsule le module invoqué. Talons et *wrappers* sont automatiquement générés par le compilateur OCL, à partir des informations données dans la description ADL de l'application.

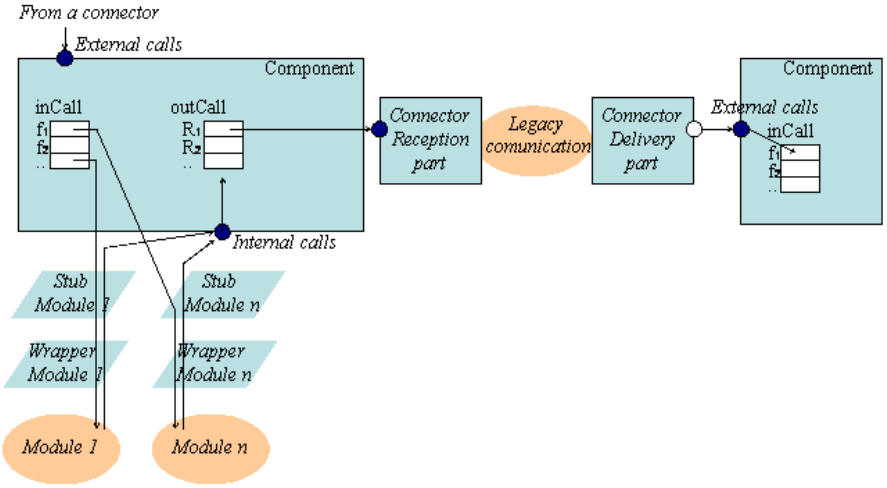

FIG. 2.9 – Cycle de vie applicatif dans Olan

2.3.5 Réflexions sur Olan

Le projet Olan, qui a débuté dans les années 1995, a été précurseur dans l'étude des modèles de construction et de configuration d'applications réparties. A l'époque, seuls les projets Darwin [63][64] et Conic [62] considéraient des objectifs similaires. Les autres réponses au problème d'intégration d'applications réparties étaient majoritairement tournées vers les solutions de type ORB (Object Request Broker). Ces solutions visaient essentiellement à supporter la communication entre applications hétérogènes. La démarche d'Olan a été d'exploiter la notion de composant pour intégrer et assembler du logiciel patrimonial au travers d'une définition explicite de l'architecture répartie globale.

Les résultats d'Olan sont un modèle et une infrastructure associée, permettant d'interconnecter, dans un environnement d'exécution donné, des composants patrimoniaux développés indépendamment les uns des autres. Ce modèle permet l'adaptabilité des dépendances fonctionnelles liant les composants, au travers de la notion de connecteur adaptable. Tout d'abord, les connecteurs sont adaptés aux caractéristiques des composants liés, ce qui permet de connecter des composants hétérogènes présentant des interfaces non conformes sur le plan du protocole de communication, du type des données échangées, ou du modèle d'exécution. Olan adapte également les connecteurs à la configuration répartie de l'application afin d'optimiser les communications.

L'adaptabilité des connecteurs repose sur les mêmes principes que ceux sur lesquels s'appuient les modèles à composants étudiés dans ce chapitre, à savoir : le principe d'encapsulation

des composants, de description explicite de l'architecture d'une application et de l'usage d'un mécanisme d'inversion de contrôle (à base d'intercepteurs).

La description explicite de l'architecture est un point clé d'Olan qui permet d'intégrer les phases de développement et de déploiement. L'architecture, décrite au travers d'un ADL, exprime les caractéristiques des composants par rapport à leurs interfaces, à leur modèle d'exécution et à leur langage d'implémentation. Sur la base des informations exprimées dans cette architecture, Olan génère automatiquement les classes des wrappers qui accéderont au code patrimonial des composants, les classes des intercepteurs qui effectueront le lien entre les *wrappers* et les connecteurs, et les classes des connecteurs. Toutes ces classes seront instanciées par Olan au moment du déploiement.

Au travers de l'inversion de contrôle utilisée par Olan, le code métier d'un composant n'est pas impacté par la gestion des interconnexions dans lesquelles il est impliqué. L'infrastructure a le contrôle complet des interconnexions, à la fois au niveau de leur mise en place et de leur utilisation. Les interconnexions, une fois créées, ne sont cependant plus modifiables durant l'exécution.

Le projet Olan a fait l'objet d'un travail de développement et d'évaluation important, s'étalant sur une durée d'environ quatre ans. Ces évaluations ont mis en lumière l'une des principales faiblesses d'Olan, à savoir le manque de support pour l'adaptabilité dynamique. La modification en cours d'exécution de l'assemblage formant une application n'est pas bien supporté. On ne peut modifier le placement des composants. L'ajout de composants est possible mais il repose sur les mécanismes d'instanciation dynamique et de collections, qui restent contraignants. La modification dynamique des connexions entre les composants n'est pas supportée.

Ce manque d'adaptabilité dynamique nous est apparu crucial par la suite. Les objectifs d'Olan, à travers la proposition d'un langage d'interconnexion de composants et d'une infrastructure associée, s'approchaient des fonctions d'administration des applications réparties. Or le support du cycle de vie complet de ces applications imposait de fournir un support pour reconfigurer dynamiquement les composants et leurs interconnexions. Ce manque d'adaptabilité a été considéré dans une deuxième phase du projet Olan, et a donné lieu a une révision des principes d'utilisation et de mise en œuvre qui ont débouché sur une thèse [85].

2.4 Conclusion

Cette section conclut sur les modèles à composants présentés dans ce chapitre. Cette conclusion porte sur le degré d'adaptabilité fourni par ces modèles à composants, et sur les techniques employées pour obtenir cette adaptabilité.

Nous avons essentiellement considéré l'adaptabilité des dépendances entre les composants, qui représentent des points critiques d'adaptation dans les applications réparties. Nous avons considéré trois types de dépendances de manière unifiée : les dépendances fonctionnelles liant les composants applicatifs entre eux, celles liant les composants avec les services non fonctionnels fournis par l'infrastructure, et enfin celles liant les composants à l'environnement d'exécution (par exemple, machines).

Dans les modèles à composants considérés, l'adaptabilité des dépendances est généralement

mise en œuvre au moment du déploiement. Lors du déploiement d'un composant donné, l'infrastructure choisit les composants qui réaliseront les services métier dont il aura besoin. De même, elle initialise les connections avec les services non fonctionnels que le composant utilisera, et choisit également les ressources matérielles utilisées (par exemple, machine). Ainsi l'infrastructure adapte l'architecture logicielle de l'application à l'environnement courant, en fonction des contraintes des composants.

Ces capacités d'adaptation sont liées à trois principes majeurs des modèles à composants : l'encapsulation, la description explicite de l'architecture et l'inversion de contrôle. L'encapsulation, telle que nous l'avons présentée dans ce chapitre, implique de définir explicitement les dépendances d'un composant, à l'extérieur de son code métier. La description explicite de l'architecture affine cette définition des dépendances en ajoutant des contraintes sur l'assemblage des composants (par exemple, co-localisation de composants). Enfin, l'inversion de contrôle consiste à ne pas gérer les dépendances d'un composant dans son code métier, mais à en donner le contrôle à l'infrastructure, sur la base de la description explicite de l'architecture.

Dans les modèles à composants étudiés, les principes d'encapsulation, de description explicite d'architecture et d'inversion de contrôle sont combinés à trois principaux patrons de mise en œuvre : l'injection de références, le tissage de code et les callbacks métier. L'injection de références permet de contrôler le système de référençage des composants. Les callbacks métier permettent à un composant de spécialiser les traitements effectués par l'infrastructure et par les services non fonctionnels qu'il utilise. Enfin, le tissage de code permet d'étendre l'interface fournie par un composant pour satisfaire les contrats de l'infrastructure et des services non fonctionnels qu'il utilise.

De notre point de vue, le tissage de code dans les composants applicatifs, sans une programmation orientée aspects, reste le mécanisme de mise en œuvre le plus critiquable pour plusieurs raisons. D'une part, il engendre des transformations au niveau du code des composants qui peuvent aller à l'encontre des choix de programmation effectués par les développeurs. Par exemple, il peut ajouter des méthodes publiques de type *setter et getter* pour manipuler les attributs d'un composant, alors que le programmeur avait choisi de définir ces attributs comme privés. D'autre part, ce tissage est souvent mis en œuvre pour pallier la déficience de structure réflexive au niveau de l'infrastructure et des composants. Les composants ne disposant pas d'une structure leur permettant d'inspecter et de manipuler leur état et leur évolution, on ajoute à leur code des méthodes qui réalisent ces tâches de manière non systématique.

La programmation orientée aspect offre une approche mieux structurée et maîtrisée du tissage de code. En particulier pour les intergiciels, les aspects offrent une modularité de la gestion des dépendances non fonctionnelles qui améliore non seulement la transparence pour les développeurs du code métier mais aussi la qualité de l'adaptabilité que l'on peut mettre en œuvre. Les aspects offrent ainsi une intégration non intrusive entre le code métier et les services non fonctionnels de l'infrastructure. En contre partie, l'usage d'aspects, lorsqu'il se base sur des techniques d'interception, peut avoir un impact sur les performances d'exécution, impact qui peut toutefois sembler mineur au regard des avantages obtenus.

Les mécanismes présentés dans ce chapitre fournissent une solution satisfaisante pour adapter les dépendances au moment de la compilation, du chargement ou du déploiement d'une application. L'adaptation des dépendances durant l'exécution d'une application reste difficile. Souvent les dépendances n'ont pas d'existance explicite à l'exécution (elles ont été traitées lors de la

compilation ou du chargement, comme c'est le cas dans JEE). Lorsqu'elles sont représentées à l'exécution, il est nécessaire que cette représentation exhibe suffisamment d'informations pour permettre de les manipuler de manière cohérente. Ces informations incluent les contraintes spécifiées dans l'ADL (par exemple, les contraintes de co-localisation) ainsi que l'état dynamique de l'application (par exemple, la localisation des flôts d'exécution).

L'adaptation des dépendances durant l'exécution d'une application est une capacité qui est requise pour pouvoir supporter pleinement la phase d'administration. En particulier, pour des raisons de réactivité des applications réparties, ou souhaite pouvoir adapter celles-ci sans avoir à les arrêter. Pour gérer efficacement les pannes de composants par exemple, on cherchera à remplacer, durant l'exécution, un composant par un autre. Les dépendances fonctionnelles vers le composant en panne doivent alors être adaptées, pour être redirigées vers le nouveau composant.

Ce type d'adaptation, à chaud, reste un problème complexe, qui va au-delà d'une simple redirection des références contenues dans les composants vers de nouveaux composants, de nouveaux services, ou de nouvelles machines. En effet, nous avons dit qu'une dépendance est associée à un contrat liant les composants de manière réciproque. Lorsqu'une dépendance d'un composant C1 vers C2 est redirigée vers un composant C3, cela peut engendrer une modification du contrat, et éventuellement une évolution du code tissé au niveau de C1. Cette problématique générale d'adaptabilité dynamique des applications à composants est étudiée dans le chapitre suivant, au travers des modèles à composants réflexifs.

Chapitre 3

Modèles à composants réflexifs

Les applications réparties ont une durée de vie généralement longue durant laquelle il faut faire face aux évolutions des besoins des utilisateurs et des variations de l'état de l'environnement d'exécution. Ce contexte changeant suggère des capacités d'adaptation dynamique afin de permettre l'évolution de ces applications sans perturber leur disponibilité.

Ces capacités d'adaptation dynamique peuvent être fournies par l'inversion de contrôle, un patron de conception majeur dans les modèles à composants qui a été présenté dans le chapitre précédent. Le principe est de déléguer la gestion des dépendances entre composants à l'infrastructure d'exécution, qui dispose ainsi de la mécanique permettant de contrôler et d'adapter ces dépendances.

L'inversion de contrôle est utilisée dans des infrastructures à composants actuellement établies commercialement, telles que JEE[68] ou Spring Framework[2]. Cependant, le niveau d'adaptabilité supporté est limité car la mise en œuvre de l'inversion de contrôle se base sur des techniques trop pragmatiques. Ainsi, les dépendances ne sont adaptables qu'au moment du déploiement et non pas tout au long de la vie d'une application.

Une meilleure adaptabilité peut être atteinte en adoptant une approche plus systématique et plus globale au niveau de l'inversion de contrôle. Cela inclut le fait de gérer une architecture logicielle plus riche, incluant toutes les formes de dépendances, y compris celles vers les ressources logicielles et matérielles du système distribué sous-jacent. L'usage d'un modèle à composants réflexifs nous paraît particulièrement intéressant pour atteindre cet objectif [40].

Un système à composant est dit réflexif s'il possède une représentation de lui-même. Cette représentation est appelée *réification*. Lorsque cette représentation est présente à l'exécution, elle permet d'une part d'observer le système, et d'autre part, de l'adapter en le reconfigurant. L'observation et la reconfiguration peuvent porter sur différents aspects, selon le type de réflexivité supporté. On distingue classiquement deux types [60] : la réflexivité structurelle et celle comportementale. Le premier type de réflexivité permet principalement d'agir sur la structure d'une application, telle que les graphes d'héritage. Le deuxième type fournit un moyen de contrôler les mécanismes comportementaux, comme par exemple les invocations de méthodes.

Pour supporter une adaptabilité dynamique complète, il nous faut associer les deux types de réflexivité aux composants. Dans cette association, les composants amènent la définition explicite

de leurs dépendances, conformément à la vision de Szyperski [20]. Sur la base de ces informations, la réflexivité structurelle réifie ces dépendances, qui sont les éléments structurants de l'architecture d'une application. La réflexivité comportementale, quant à elle, fournit le contrôle nécessaire pour adapter le comportement des dépendances.

Si un certain nombre de systèmes et d'intergiciels ont adopté l'usage de la réflexivité comme moyen d'obtenir de l'adaptabilité [5][61][41][124][42], peu de modèles à composants ont exploré cette approche. Parmi ceux-ci, Fractal [29], OpenCOM [43], Dynacomp [46] et K-Component [58] sont des exceptions notables. Fractal et OpenCOM adoptent un modèle à composants réflexifs qui réifie non seulement les dépendances fonctionnelles entre composants applicatifs mais aussi certaines dépendances non fonctionnelles. Dans les deux cas, au travers des fonctions de réflexivité structurelle et comportementale, on dispose d'une mécanique permettant d'adapter dynamiquement des applications.

Sur la base de Fractal et de sa capacité à adapter dynamiquement des applications, Jade [115] est une infrastructure destinée à l'administration autonome d'applications réparties. Jade pousse l'usage de la réflexivité plus loin qu'OpenCOM et Fractal sur deux axes. Le premier est la réification non seulement de l'architecture logicielle de l'application mais aussi de celle du système distribué sur lequel s'exécute cette application. Le second axe est la gestion autonome de l'adaptabilité des applications, c'est-à-dire que celles-ci sont adaptées sans intervention humaine. En effet, l'administration dans Jade est mise en œuvre par des boucles de commande visant à auto-réparer, auto-optimiser et auto-protéger les applications. Ces boucles observent et reconfigurent l'architecture logicielle des applications pour les réparer en cas de panne, les protéger en cas d'intrusion ou encore les optimiser en présence de charges variables dans le temps.

Le reste de ce chapitre est organisé comme suit. Dans la section 3.1, nous revenons sur le concept de système réflexif dans un cadre général, puis nous considérons plus spécifiquement les besoins des modèles à composants adaptables. Les concepts et principes introduits permettent ensuite de décrire et de positionner l'infrastructure à composants réflexifs Jade [115], qui correspond au projet que nous avons mené en collaboration avec Daniel Hagimont et Noël de Palma, de 2003 à aujourd'hui. Après un bilan sur Jade, nous présentons dans la section 3.5 nos conclusions sur l'adéquation des modèles à composants réflexifs pour l'adaptabilité dynamique dans les applications réparties.

3.1 Systèmes réflexifs

Une approche générale d'un système réflexif consiste à l'organiser selon deux niveaux [44]. Le niveau de base comprend les entités manipulées par l'application et fournit les fonctions métier définies par les spécificités de l'application. Le *méta-niveau* utilise une représentation de ces entités pour observer ou modifier le comportement du niveau de base. Il contrôle les mécanismes utilisés pour assurer le fonctionnement des entités du niveau de base.

Le fondement d'une approche réflexive est la réification, un processus qui conduit à matérialiser un concept à l'exécution. Cette matérialisation doit permettre à la fois l'observation et la manipulation pour satisfaire l'appellation d'approche réflexive. Un exemple bien connu est la réification des types d'un langage à l'exécution. Le cas de Java propose une approche réflexive partielle puisque les classes, réifiées comme instance de la class Class, ne sont qu'observables. D'autres langages, comme Smalltalk ou Python par exemple, offrent une approche réflexive com-

plète puisque les classes sont non seulement observables mais aussi modifiables dynamiquement à l'exécution.

La réification ne concerne pas seulement les types des langages de programmation, comme il est souvent d'usage de le croire. Il est possible de réifier tout concept, et en particulier des concepts d'exécution comme l'invocation de méthode dans un langage à objets réflexifs [28] [4]. Il est aussi possible de réifier des éléments d'architecture comme cela peut être le cas avec une architecture distribuée. En modélisant les noeuds d'une grappe de machines, on permet à la fois d'observer le système distribué, avec ses performances dynamiques, et d'en changer l'architecture en modifiant par exemple les tables de routage, les règles des pare-feux ou encore les services qui y sont déployés.

Au delà de la simple réification, le fonctionnement du méta-niveau, et en particulier ses interactions avec le niveau de base, sont spécifiés par un protocole appelé Meta Object Protocol (MOP) [44]. Un MOP peut fournir deux types de réflexivité : structurelle et comportementale.

La réflexivité structurelle permet de réifier la structure d'une application comme les graphes d'héritage et de composition, ou encore les types de données. Cette réflexivité fournit par exemple une description et un accès aux champs d'un objet, à ses méthodes, ainsi qu'à celles héritées.

La réflexivité comportementale réifie les aspects de calcul et de comportement d'une application [80]. Un système de ce type peut permettre de disposer de deux implémentations d'un même module et de basculer de l'une à l'autre. La réification des mécanismes de création d'objets ou de composants, ainsi que des appels de méthodes relève de ce type de réflexivité. En particulier, le début d'une invocation, la terminaison, ou les invocations en attente sont classiquement manipulables au travers de cette réflexivité.

La réflexivité comportementale a été explorée dans le contexte des applications réparties à objets, pour adapter les mécanismes de base du système ainsi que pour enrichir les objets avec des propriétés non fonctionnelles de manière (quasi) indépendante de leur code [27][6][123][54]. Dans EJava [28] par exemple, l'infrastructure, composée d'un noyau minimal, est représentée par un objet du méta-niveau appelé méta-objet. Les fonctions fournies par ce méta-objet permettent d'adapter le noyau pour le rendre opérationnel dans un environnement d'exécution donné, par l'ajout de services et de mécanismes de base tels que des protocoles de communication, des services d'invocation distante ou des services de nommage particuliers.

3.2 Mise en oeuvre des systèmes réflexifs

La façon d'organiser un méta-niveau diffère selon les architectures réflexives. Un élément du méta-niveau peut en effet être mis en œuvre par une classe (appelée méta-classe), par un objet ou par un groupe d'objets.

3.2.1 Par méta-classe

Les modèles réflexifs basés sur des méta-classes tels que CLOS [44] ou OpenCorba [123] rendent les classes manipulables par programme au travers d'une classe de plus haut niveau appelée méta-classe (voir figure 3.1).

Une méta-classe définit un comportement commun à l'ensemble des instances d'une classe donnée. Un exemple de méta-classe est la classe *Class* en Java, qui définit les méthodes *newIns-*

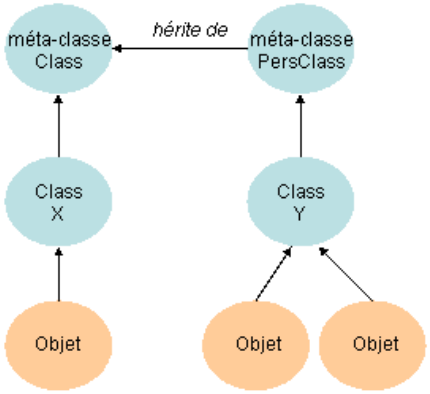

FIG. 3.1 – Structures basées sur des méta-classes

tance() pour créer des instances, *invoke()* pour invoquer des méthodes, ou encore *getAttribute()* pour obtenir les attributs d'une classe. Plusieurs niveaux de méta-classes peuvent être définis (méta-classe, méta-méta-classe, etc.).

Une architecture à méta-classe permet de définir un comportement non-fonctionnel au niveau d'une méta-classe qui s'appliquera donc à l'ensemble des instances des classes elles-mêmes instances de cette méta-classe. Par exemple, la gestion d'objets persistants peut s'appuyer sur la définition d'une méta-classe spécifique *persistentMetaClass*. Ainsi, toute classe définie comme une instance de *persistentMetaClass* aura des instances persistentes.

Un avantage de ce type de structure réside dans sa facilité d'utilisation puisque les méta-classes sont rattachées aux classes par un lien d'instanciation. En revanche, puisque ce lien d'instanciation est unique, il est impossible d'associer des comportements différents à des objets d'une même classe. Par ailleurs, en l'absence d'héritage multiple, il faut définir autant de méta-classes que de combinaisons de comportements non-fonctionnels, ou bien définir une méta-classe générique qui serait capable d'interpréter des besoins non-fonctionnels méta-déclarés par classe ou par instance de classe.

3.2.2 Par méta-objet

Les structures basées sur des méta-objets [80][14][6] mettent en œuvre le méta-niveau avec des objets qui contrôlent tout ou partie des aspects réifiés pour les objets du niveau de base. Ces aspects peuvent relever d'une réflexivité comportementale ou structurelle.

La configuration des relations entre les méta-objets et les composants du niveau de base peut se fonder sur différentes techniques. Dans DART [94], lors de la programmation d'une classe applicative, le programmeur spécifie à l'aide d'étiquettes la composition des méta-objets qui sera, par défaut, affectée aux instances de cette classe, ainsi que les méthodes dont l'appel doit engendrer un transfert au méta-niveau. D'autres approches cherchent à séparer complètement le code métier du code non-fonctionnel, et donc de l'expression des liens avec le méta-niveau. Dans ReflectiveJava [54] ainsi que dans Fractal [29] par exemple, la configuration des méta-objets associés aux objets de base est décrite dans des fichiers de configuration. Enfin, dans [28], des annotations sont insérées dans les programmes applicatifs.

Organisation des méta-objets On distingue deux types d'organisations à base de méta-objets, selon que les méta-objets sont partagés ou non entre plusieurs composants du niveau de base. Dans les organisations basées sur des méta-objets *propres*, un méta-objet gère le comportement propre à un objet, tel que par exemple une politique de synchronisation ou de sécurité particulière, dépendante des méthodes exportées par cet objet. A l'inverse, un méta-objet *partagé* gère un comportement commun à un groupe d'objets du niveau de base.

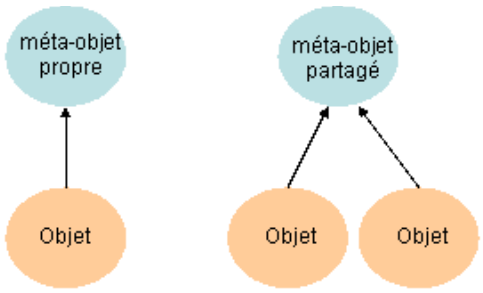

FIG. 3.2 – Structures basées sur des méta-objets

Certaines approches définissent une bijection entre méta-objets et objets du niveau de base, ce qui revient à centraliser dans un méta-objet propre tous les aspects réifiés. Sont structurés de cette manière ReflectiveJava [54] qui est une extension réflexive du langage Java, et CAOLAC[75] qui définit un MOP pour des applications réparties à objets, permettant de gérer des aspects relatifs à la répartition, à la concurrence et à la synchronisation des objets. Dans CAOLAC, le méta-objet est plus précisément réalisé comme une classe dont hérite un objet du niveau de base.

Les structures qui associent plusieurs méta-objets à un objet de base permettent, dans le principe, de gérer les aspects réifiés de manière plus indépendante et plus dynamique. Par exemple, cette organisation permet d'associer un méta-objet spécifique à chaque aspect de contrôle d'un objet, tels que la gestion de son cycle de vie, de ses dépendances, de sa structure interne, etc. Elle permet également d'ajouter et de retirer des méta-objets dynamiquement. Bien que ce type d'organisation paraisse avantageux d'un point de vue génie logiciel, la mise en œuvre des méta-objets reste complexe car les aspects réifiés sont rarement indépendants les uns des autres. En particulier, lorsque le méta-niveau est utilisé pour gérer les dépendances non fonctionnelles (c'est-à-dire pour gérer les interactions avec les services implantant les propriétés visées), se pose le problème de combiner ces propriétés de manière cohérente. Certaines infrastructures réflexives proposent pour cela un modèle de composition de méta-objets dont nous parlons ci-après.

Composition des méta-objets Un modèle de composition de méta-objets fournit un cadre pour combiner des aspects de contrôle selon un schéma de coopération précis. [28][80][5] proposent une composition en pile. À tout niveau, un méta-objet a le choix d'intervenir ou non dans le traitement d'un aspect réifié, avant de déléguer le traitement au méta-objet qui lui succède dans la pile. La figure 3.3 montre une pile composée de deux méta-objets redéfinissant le comportement de l'invocation de méthode : le méta-objet *monitoring* est chargé de réaliser des observations ; le méta-objet *persistence* provoque une sauvegarde sur un support persistant avant et après tout appel au composant de base.

D'autres projets permettent de composer des méta-objets comme des paires d'objets liés par une relation orientée d'agrégation ou de spécialisation [123]. Comme pour l'organisation précé-

dente, cette solution impose une conception particulière des méta-objets qui prévoit leur future composition.

Enfin, des modèles de composition plus complexes ont été proposés. Dans Guarana[6], on trouve un modèle de composition en arbre, dans lequel à chaque niveau, un méta-objet appelé *composer* contrôle les appels sur les méta-objets de niveau inférieur et peut introduire du parallélisme dans ces appels.

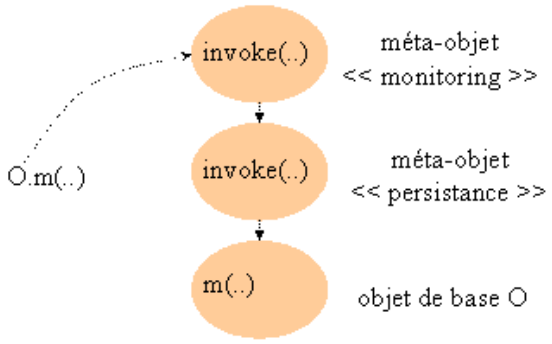

FIG. 3.3 – Exemple d'organisation en pile du méta-niveau

Citons finalement le projet DART [94], qui fournit une infrastructure adaptable pour des applications réparties dans laquelle des propriétés non fonctionnelles peuvent être dynamiquement attachées à des objets de base. DART utilise la notion de méta-espace, correspondant à un ensemble plat de méta-objets qui sont tous invoqués séquentiellement lorsque le méta-espace est sollicité. Un méta-espace peut être partagé par un groupe d'objets de base.

Approche mixte On trouve également des structures mixtes, dans lesquelles le méta-niveau est formé de méta-objets propres et de méta-objets partagés, afin de profiter des avantages respectifs de chacune des approches.

Dans l'ORB Flexinet [5], un *cluster* correspond à un ensemble d'objets liés fonctionnellement. Les méta-objets associés à un cluster gèrent les propriétés non fonctionnelles telles que la persistance ou les transactions. Ils supportent un ensemble évolutif de propriétés non fonctionnelles. En revanche, la gestion des liens entre les objets est contrôlée par des méta-objets propres.

Dans Guarana [6] et MetaXa [80], chaque méta-objet partagé gère une propriété non fonctionnelle donnée. Un même objet de base peut être sous le contrôle de plusieurs méta-objets qui peuvent évoluer dynamiquement. La souplesse supportée par cette structure est associée à une forte complexité de mise en œuvre des méta-comportements. Pour réduire cette complexité, Guarana associe un méta-objet propre *primaire* à tout objet de base, qui se charge d'appeler les méta-objets *secondaires* (gérant par exemple des propriétés non fonctionnelles), selon un schéma de coopération prédéfini appelé méta-configuration.

3.3 Systèmes réflexifs à composants

L'usage de fonctions réflexives dans les modèles à composants couvre des objectifs plus larges que ceux initialement ciblés dans les modèles à objets réflexifs, qui visaient principalement l'association de propriétés non fonctionnelles aux objets. Avec les composants, on souhaite pouvoir introspecter et adapter toutes les formes de dépendances architecturales d'une application, de manière uniforme et à tout moment du cycle de vie.

Les dépendances peuvent être fonctionnelles, liant les composants de l'application entre eux, ainsi que non fonctionnelles, liant les composants aux services de l'infrastructure et aux ressources de l'environnement d'exécution (voir section 2.1.2). L'objectif est de réifier l'ensemble des dépendances, fonctionnelles et non fonctionnelles, afin d'obtenir un moyen homogène d'adapter celles-ci.

3.3.1 Types de réflexivité

L'adaptation des dépendances entre composants relève d'une réflexivité structurelle, car ces dernières sont des éléments structurants de l'architecture d'une application. Cependant, il est nécessaire de combiner la réflexivité structurelle avec celle comportementale, car la mise en œuvre de la première forme de réflexivité requiert de disposer de fonctions fournies par la seconde. En particulier, pour manipuler les dépendances, il est nécessaire d'adapter les opérations de gestion du cycle de vie et d'invocation des composants, comme cela a été montré dans le chapitre précédent (voir section 2.2). L'adaptation de ces opérations relève d'une réflexivité comportementale.

Les modèles à composants réflexifs fournissent donc une réflexivité mixte qui s'applique sur un niveau de base formé par les composants applicatifs. Ces composants possèdent une structure de données interne, qui peut être manipulée par des méthodes métier définies par l'interface fonctionnelle qu'ils exportent. Le méta-niveau réifie la structure interne ainsi que les aspects non fonctionnels des composants, c'est-à-dire la sémantique des opérations de contrôle telles que l'invocation de méthode, les opérations de gestion du cycle de vie, d'assemblage, etc.

On trouve ce type de réflexivité mixte dans les systèmes administrables qui utilisent la réflexivité comme support pour l'adaptabilité [115][29][43][58]. Dans Fractal [29], par exemple, la réflexivité structurelle permet de contrôler tout type de dépendance liant un composant à un autre, telle qu'une relation de composition. Ainsi, il est possible d'ajouter ou de retirer un sous-composant ou une relation de délégation. La réflexivité comportementale dans Fractal permet de contrôler la gestion du cycle de vie d'un composant ainsi qu'une partie de sa structure interne, exposée comme un dictionnaire d'attributs.

OpenCOM [43] associe un groupe de méta-objets (appelés *méta-modèles*) à chaque composant du niveau de base. Ces méta-objets gèrent de manière complémentaire des aspects structurels (*interface* et *architecture*) et comportementaux (*interception* et *ressources*). Le méta-objet *interface* réifie les interfaces métier fournies et requises par un composant alors que le méta-objet *architecture* réifie sa structure interne en termes d'un assemblage de sous-composants et des contraintes qui y sont associées. Le méta-objet *interception* permet d'ajouter des traitements lors des invocations des méthodes métier d'un composant. Enfin, le méta-objet *ressources*, qui constitue un élément différenciant du modèle OpenCOM, permet de gérer les ressources supportant l'exécution d'un composant, telles que les threads, la mémoire allouée, ou la taille des buffers utilisés. Ce méta-objet est partagé entre les composants d'un même espace d'adressage.

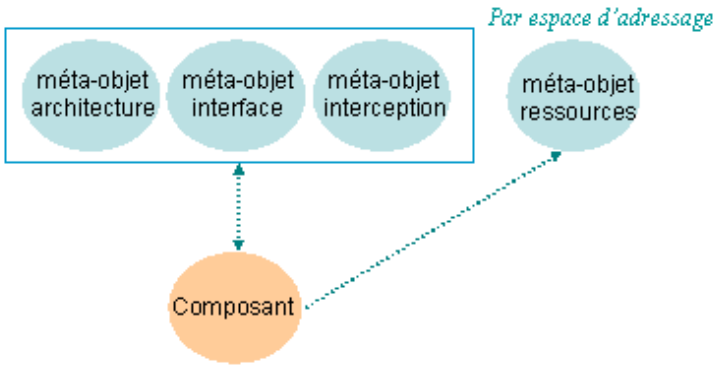

FIG. 3.4 – Structure du méta-niveau dans OpenCOM

3.3.2 Aspects réifiés

L'adaptabilité des dépendances repose donc sur une combinaison des fonctions de réflexivité structurelle et comportementale. Pour obtenir un niveau d'adaptabilité dynamique, il faut également considérer quels sont les aspects réifiés au niveau de chaque type de réflexivité.

En particulier, la réflexivité structurelle doit fournir des capacités d'introspection avancées au niveau des dépendances. Pour toute dépendance, cette réflexivité doit permettre de connaître les propriétés et les contraintes qui la définissent. Si l'on considère par exemple une dépendance non fonctionnelle liant un composant à un service de l'infrastructure, la structure réflexive doit permettre de connaître la fonction attendue au niveau de ce service (par exemple, sécurité), le contrat permettant de l'utiliser, la version utilisée, etc. Ces informations sont requises pour mettre en œuvre une dépendance ainsi que pour gérer son évolution dynamique (par exemple remplacer les éléments liés, ou faire évoluer le contrat qui les lient) de manière cohérente.

Au niveau de la réflexivité comportementale, le cycle de vie des composants doit être réifié de manière à disposer des méta-opérations permettant de stopper temporairement un composant, pour le redémarrer ultérieurement, après avoir modifié certaines de ses dépendances. En particulier, pour pouvoir manipuler dynamiquement les dépendances d'un composant C tout en préservant la cohérence d'une application, il est nécessaire d'attendre que toutes les transactions et/ou invocations métier impliquant C soient terminées.

Ce problème de synchronisation est abordé en détails dans [62], qui définit différentes notions dont celles d'*état passif* et d'*état gelé*. Un composant dans l'état passif peut finir de traiter les transactions et/ou invocations en cours mais n'en accepte pas et n'en initie pas de nouvelles. Dans l'état gelé, un composant ne reçoit plus de requêtes, n'en initie plus et n'est pas en train d'en servir une. [62] propose un algorithme permettant de geler tout composant préalablement à sa reconfiguration dynamique, sur la base de deux méta-opérations (*activate()* et *passivate()*) forçant un composant à passer respectivement dans l'état actif ou dans l'état passif. Dans certains modèles à composants réflexifs [29], ces méta-opérations sont confondues avec celles permettant de démarrer et d'arrêter un composant (par exemple, *start()* et *stop()*). Nous donnons une vision plus concrète de ces aspects dans la section suivante, au travers de la description de l'infrastructure Jade.

3.4 L'infrastructure à composants Jade

Jade [112][115][118] est une infrastructure à composants pour l'administration autonome d'applications réparties. Cette infrastructure est basée sur le modèle à composants réflexifs Fractal [29]. Elle supporte les phases de développement, déploiement, exécution et administration des applications réparties. L'approche adoptée dans Jade est de fournir une administration basée sur l'architecture des applications. Ce type d'administration exploite la présence d'une architecture logicielle explicite et manipulable pour mettre en œuvre des services d'administration évolués. Cette approche permet de construire des services d'administration autonomes, capables d'exécuter des tâches administratives sans intervention humaine.

Les applications administrées peuvent être construites directement avec le modèle Fractal, ou peuvent être patrimoniales. Dans le premier cas, les composants Fractal portent à la fois le code fonctionnel des applications et les opérations d'administration élémentaires qui définissent une interface d'administration minimale et uniforme. Cette interface d'administration correspond en effet à l'interface réflexive du modèle Fractal, qui fournit des fonctions d'introspection et d'intercession minimales pour d'administration. Lorsque les éléments administrés sont patrimoniaux, ils sont encapsulés dans des composants qui réifient leurs interfaces d'administration spécifiques dans l'interface d'administration uniforme.

La réflexivité du modèle Fractal est à la fois structurelle et comportementale. La réflexivité structurelle réifie l'architecture logicielle des composants, en termes de leur structure interne et de leurs dépendances. Tout lien entre composants, de type délégation ou composition, est réifié au méta-niveau. La réflexivité comportementale réifie principalement la gestion du cycle de vie et les invocations de méthodes métier. Cette double réflexivité permet de connaître et de manipuler, en cours d'exécution, l'état courant de l'architecture d'une application. L'infrastructure a ainsi la possibilité de connaître les liens mettant en œuvre les dépendances d'un composant donné. Elle peut également agir sur les composants et sur leurs liens, en modifiant la topologie de l'architecture ou en agissant sur le comportement des composants par le contrôle de leur cycle de vie.

La suite de cette section est organisée comme suit. La section 3.4.1 présente le modèle à composants Fractal qui est utilisé à la fois pour mettre en œuvre l'infrastructure Jade et pour construire les applications administrées avec Jade. La section 3.4.2 présente les principes de conception de l'infrastructure Jade, qui s'appuient sur une organisation récursive permettant d'obtenir des propriétés d'auto-administrabilité. Nous passons ensuite à la mise en œuvre de l'infrastructure Jade (section 3.4.3), et montrons comment l'inversion de contrôle est réalisée sur la base des capacités réflexives fournies par Fractal. La section 3.4.4 montre que les capacités d'adaptation obtenues sont suffisantes pour mettre en œuvre un service d'auto-réparation capable de réparer, sans intervention humaine, tout composant tombant en panne dans le système, y compris lui-même. Un bilan récapitule les points forts de Jade ainsi que les aspects qui mériteraient d'être améliorés dans le futur.

3.4.1 Modèle à composants

Jade s'appuie sur le modèle Fractal [29] qui est un modèle à composants réflexifs destiné à la construction d'applications réparties complexes. Ce modèle capture à la fois la dimension fonctionnelle des composants au travers des concepts d'interfaces et de dépendances mais aussi la dimension réflexive qui fournit la base de la réification de l'architecture nécessaire à Jade.

Le modèle Fractal est hiérarchique, c'est-à-dire qu'il permet de définir des composants primitifs et composites. Un composant primitif contient du code exécutable. Un composant composite est construit comme un assemblage de sous-composants, dont certains peuvent être partagés entre plusieurs composants composites. Les composants primitifs ou composites exhibent des interfaces fonctionnelles clientes et serveurs, qui peuvent être liées pour créer des assemblages de composants. Le modèle est typé, de manière à fournir des garanties de compatibilité lorsque les composants sont assemblés. Le type d'un composant est défini par ses dépendances fonctionnelles requises et fournies.

En plus de ses interfaces fonctionnelles, un composant possède un ensemble extensible d'*interfaces de contrôle* qui définissent ses capacités réflexives. Ces interfaces permettent d'inspecter et de manipuler les aspects réifiés suivants : son état, son cycle de vie, ses dépendances fonctionnelles, et ses invocations entrantes et/ou sortantes (par interception). Ces aspects relèvent d'une réflexivité structurelle (état, dépendances) et comportementale (invocations, cycle de vie).

Ces concepts sont illustrés dans la figure ci-après qui représente un assemblage de deux composants C1 et C2. Au niveau de base, le composant C1 est lié par une dépendance fonctionnelle au composant C2. Au niveau méta, le système maintient une représentation de cette architecture qui réifie les composants C1 et C2 ainsi que leurs dépendances. Cette représentation est causalement connectée au niveau de base, car les opérations de contrôle effectuées sur le méta-niveau, comme le changement d'une dépendance par exemple, sont automatiquement répercutées sur le niveau de base.

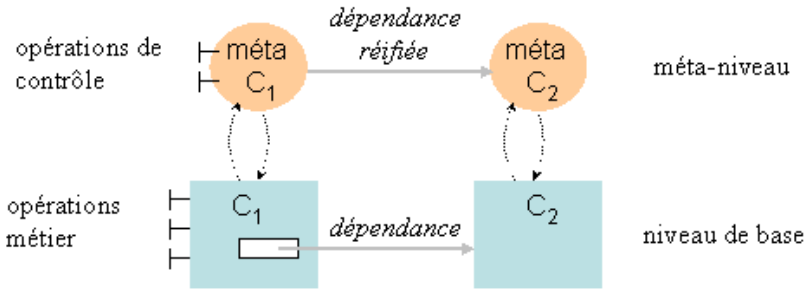

FIG. 3.5 – Méta-niveau des composants Fractal

Le méta-niveau est structuré grâce à des objets de contrôle, appelés contrôleurs. Un contrôleur réifie un aspect donné du composant fonctionnel, selon une certaine politique. Ces objets de contrôle sont propres à chaque composant, comme illustré dans la figure 3.6.

Bien que le modèle soit extensible et qu'il permette l'ajout de nouvelles interfaces de contrôle et des contrôleurs associés, Jade ne dépend que des contrôleurs suivants.

- *AttributeController*. Les attributs sont des propriétés configurables qui forment l'état administrable d'un composant. Le contrôleur d'attributs implante une interface qui fournit les méthodes (getter/setter) permettant de manipuler ces attributs.
- *LifecycleController*. Ce contrôleur permet d'appeler les opérations élémentaires de gestion du cycle de vie, comprenant au minimum le démarrage et l'arrêt d'un composant.
- *BindingController*. Ce contrôleur contrôle les dépendances d'un composant vers d'autres composants. Il implante une interface permettant de consulter et de modifier les liens qui, à l'exécution, résolvent les dépendances du composant.

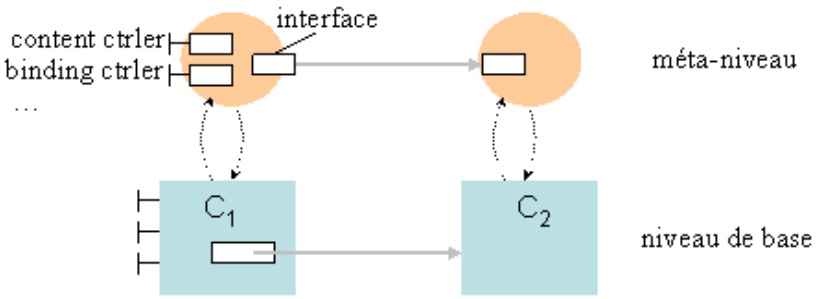

FIG. 3.6 – Organisation du méta-niveau dans Fractal (en complément à la figure précédente)

– *ContentController*. Ce contrôleur permet d'observer, d'ajouter, de retirer des sous-composants dans un composite.

Les contrôleurs sont sollicités de manière explicite, à l'exception des contrôleurs de type *intercepteurs* qui réifient les invocations de méthodes métier. Un intercepteur est mis en œuvre par un objet (appelé *proxy*) qui implémente une interface métier fournie ou requise par un composant. Cet objet a la possibilité d'ajouter des traitements exécutés avant et/ou après les invocations de l'interface métier qu'il représente puisque toute invocation sur l'interface métier passe par lui.

Dans la mise en œuvre Julia de Fractal [29], la composition des contrôleurs repose sur une organisation en pile de *mixins* inspirée de JAM (www.disi.unige.it/person/LagorioG/jam). Cette organisation est illustrée dans la figure ci-après, pour le cas du contrôleur de liaisons (BindingController). Un mixin est un objet Java respectant un certain patron de programmation, qui lui permet d'être inséré à tout endroit dans une pile protocolaire.

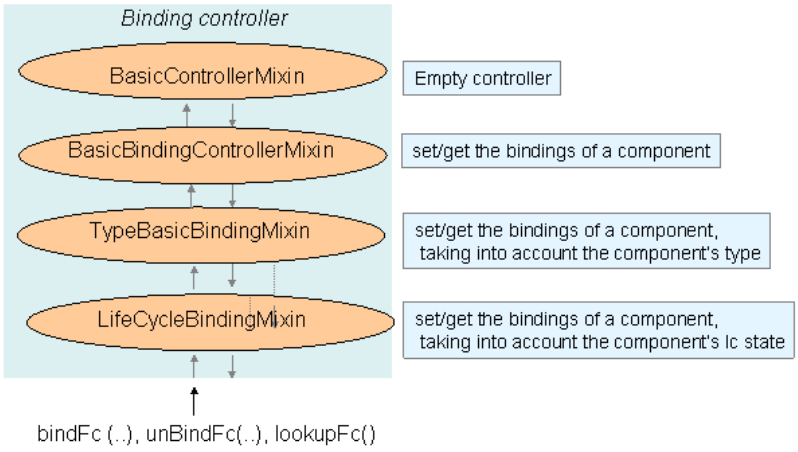

FIG. 3.7 – Organisation des contrôleurs Fractal en pile protocolaire

Ainsi, l'interface *BindingController* est composée de trois méthodes de contrôle : *bindFc()*

pour établir une liaison, *unBindFc()* pour détruire une liaison, et lookupFc() pour obtenir la liste des liaisons du composant courant. Chacune de ces méthodes est définie au niveau de chaque mixin de manière à avoir la capacité d'invoquer systématiquement la même méthode au niveau du mixin suivant dans la pile.

Par exemple, lors de l'invocation de la méthode *bindFc()*, le mixin *LifeCycleBindingMixin* est invoqué en premier (selon la figure ci-après, qui est une simplification de la pile protocolaire réelle). Ce mixin va vérifier que le composant est dans un état du cycle de vie qui lui permet d'établir une liaison. Le mixin *TypeBasicBindingMixin* va ensuite vérifier que la liaison à établir est conforme aux contraintes du système de types. Enfin, le mixin *BasicBindingController* va établir la liaison effective.

Dans cette approche à pile de mixins, chaque interface de contrôle est donc générée par assemblage de mixins. Ainsi, chaque contrôleur peut intervenir dans toute interface de contrôle et non seulement dans la sienne. Le contrôleur de liaisons peut mixer du code à la fois dans les méthodes de l'interface *BindingController* mais aussi dans les autres interfaces de contrôle comme *AttributeController* ou *LifecycleController*. Cela revient donc à voir un contrôleur comme un ensemble de mixins qui sont composés pour générer le code des interfaces de contrôle, ces mixins pouvant mettre en œuvre la fonction de contrôle attendue ou bien gérer la coordination avec un autre aspect de contrôle.

L'organisation à base de pile de mixins fournie par Fractal est optimisée, à l'exécution, de manière à fusionner les objets et à réduire le nombre d'invocations. Cette optimisation repose sur la construction de classes par fusion des méthodes provenant de différentes classes.

3.4.2 Architecture réifiée

Le choix de représenter les applications administrées par des composants Fractal nous a permis de placer la notion d'architecture logicielle au centre de la conception de Jade. Cette architecture est définie par le méta-niveau des composants Fractal, qui réifie l'architecture logicielle effective de l'application répartie. Elle a donc une existence explicite à l'exécution.

Ce choix s'est révélé essentiel pour atteindre nos objectifs d'adaptabilité dynamique, mais non suffisant. Pour assurer une adaptabilité dynamique complète, il nous a fallu étendre cette architecture réifiée pour capturer non seulement l'assemblage des composants applicatifs, mais également les services de l'infrastructure et les ressources de l'environnement d'exécution. Cela permet en effet de prendre en considération l'ensemble des dépendances architecturales, comme identifié au chapitre précédent (section 2.1.2).

La figure 3.8 illustre l'ensemble des dépendances d'un composant C1 qu'il est important de réifier dans l'architecture réflexive. Tout d'abord, nous y trouvons les dépendances fonctionnelles entre composants, représentées pas une dépendance fonctionnelle liant C1 au composant C2. Nous y trouvons aussi les dépendances non fonctionnelles liant C1 à l'infrastructure et à l'environnement, toutes les deux réifiés par des composants. Ici, nous avons deux dépendances non fonctionnelles, l'une sur un composant S représentant un service fourni par l'infrastructure et l'autre sur un composant R représentant une ressource fournie par l'environnement.

Le choix de représenter l'infrastructure et l'environnement par des composants permet d'unifier la gestion des dépendances fonctionnelles avec celles non fonctionnelles. En effet, ces dépendances étant mises en œuvre par des liens entre composants, elles sont automatiquement

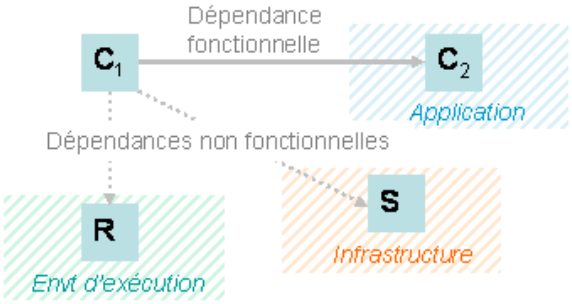

FIG. 3.8 – Dépendances réifiées dans Jade

réifiées dans l'architecture, par les mêmes mécanismes du modèle Fractal que pour les composants applicatifs. En d'autres termes, par les contrôleurs Fractal, ces dépendances deviennent introspectables et manipulables, et par conséquent adaptables.

Illustrons cela tout d'abord sur le cas d'une dépendance non fonctionnelle sur l'environnement d'exécution. Dans Jade actuel, un composant applicatif peut définir une dépendance sur un noeud virtuel d'exécution et non sur une machine spécifique. Cette dépendance peut décrire les caractéristiques attendues de la machine d'exécution. Il sera de la responsabilité du service de déploiement dans Jade de choisir le noeud du système distribué qui deviendra la machine physique d'exécution et d'établir le lien (par exemple, le *binding* Fractal) entre le composant applicatif et celui représentant la machine d'exécution.

Cela présuppose que Jade modélise l'architecture matérielle du système distribué sous-jacent. L'approche choisie est de modéliser cette architecture comme une composition hiérarchique de composants. Ainsi, une grappe de machines est un composite Fractal qui groupe des composants représentant les machines, eux-mêmes composites regroupant les composants actuellement déployés sur ces machines. Déployer un composant C sur une machine représentée par un composant M revient donc tout simplement à ajouter le composant C comme sous-composant de M.

La figure 3.8 illustre aussi des dépendances non fonctionnelles sur des services de l'infrastructure, également modélisés par des composants. Ces services peuvent être des fournisseurs de propriétés non fonctionnelles (par exemple, un service de réplication, de transaction, ou de sécurité), ou des services d'administration (déploiement, réparation, etc.). L'infrastructure Jade fournit en effet une structure d'accueil pour ces deux catégories de services.

Deux exemples importants sont les services d'auto-protection et d'auto-réparation. L'auto-protection a été expérimentée dans le cadre d'une grappe de machines [12]. Ce service s'appuie sur l'architecture réifiée pour connaître les canaux de communication qui sont légitimes entre les composants. Dans cette architecture, les liens mettant en œuvre les dépendances réifient les informations nécessaires telles que les adresses IP et les ports utilisés pour établir des connections TCP/IP entre des composants distants. Il est alors possible de configurer des pare-feux, un sur chaque noeud, pour détecter les tentatives de communication au travers de canaux non déclarés dans l'architecture.

Tout noeud étant la source d'une telle tentative est considéré comme contaminé et sera isolé par le service d'auto-protection. Cette isolation s'obtient par une restructuration architecturale,

coupant tous les liens sortant des composants déployés sur le noeud contaminé. On voit là un autre usage de la réification de l'architecture distribuée de la grappe sous-jacente, ce qui illustre la puissance d'une approche uniforme. L'auto-réparation illustre cette même puissance puisqu'elle s'appuie aussi sur l'architecture afin de reconstruire celle-ci à l'identique après une panne de noeud. L'auto-réparation est le cas pratique étudié en détail à la fin de ce chapitre.

La gestion des dépendances unifiées est un choix majeur dans Jade, qui permet d'obtenir une structure d'administration récursive, dans laquelle par principe tout service de l'infrastructure peut s'appliquer à tout composant. Cette structure est illustrée dans la figure 3.9. Les services s'appliquent aux composants applicatifs, c'est leur fonction première. Mais ils s'appliquent aussi aux composants qui mettent en œuvre les services (non fonctionnels) de l'infrastructure et de l'environnement d'exécution. Par exemple, l'auto-protection peut s'appliquer au service d'auto-réparation pour le protéger d'éventuelles attaques. Inversement, l'auto-réparation s'applique au service d'auto-protection afin de pouvoir le réparer s'il est endommagé par une panne.

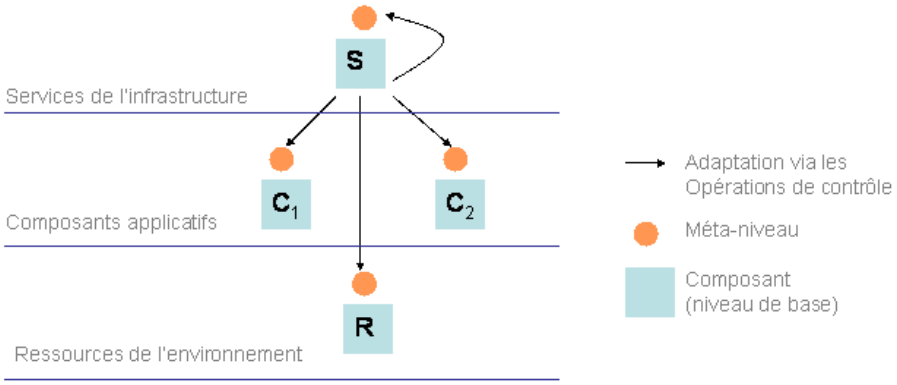

FIG. 3.9 – Structure récursive dans Jade

En poursuivant cette logique, les services peuvent s'appliquer à eux-mêmes puisqu'ils s'appliquent aux composants présents dans l'architecture réifiée et qu'étant eux-mêmes mis en œuvre par des composants, ils apparaissent dans cette architecture. Ainsi, le service d'auto-protection se retrouve à devoir s'auto-proteger d'attaques et le service d'auto-réparation à devoir s'auto-réparer. Cette récursion vient de l'uniformité de l'approche où les services de l'infrastructure sont mis en œuvre par des composants. Elle préserve la minimalité des concepts et la facilité de mise en œuvre. La possibilité de récursion infinie est stoppée au deuxième niveau par l'usage de contrôleurs spécifiques.

3.4.3 Adaptabilité dynamique

L'adaptabilité dynamique des applications administrées par Jade s'appuie sur l'inversion de contrôle. D'une part, l'infrastructure prend en charge, au travers du service de déploiement, la création des composants, la mise en place de leurs dépendances initiales, et leur démarrage à partir de la description d'architecture exprimée dans l'ADL Fractal. D'autre part, les composants et les dépendances, une fois créés, sont automatiquement administrés par l'infrastructure, de manière à garantir leur bon fonctionnement.

L'inversion de contrôle mise en œuvre par Jade a donc lieu d'une part au moment du déploiement d'une application pour produire un assemblage de composants conforme à la description donnée dans l'ADL, et d'autre part durant l'exécution pour adapter l'application aux évolutions de l'environnement et de ses utilisateurs.

Le patron de mise en œuvre pour cette inversion de contrôle repose sur les interfaces de contrôle fournies par les composants Fractal. D'une part, ces interfaces sont capables de manipuler le cycle de vie des composants, ce qui permet de spécialiser les actions effectuées lors des changements d'états du cycle de vie, en particulier pour effectuer des appels aux services non fonctionnels. Ces interfaces permettent également de gérer la passivation d'un composant avant de manipuler ses dépendances, afin de préserver la cohérence de l'application en cours d'exécution (voir section 3.3.2).

D'autre part, les interfaces de contrôle permettent d'inspecter les dépendances des composants pour obtenir les caractéristiques qui y sont associées. Ces caractéristiques peuvent correspondre à des contraintes de co-localisation, de version, ou de qualité de service portant sur le fournisseur d'un service donné. En l'état actuel de Jade, les informations qui réifient une dépendance sont minimales. Elles donnent accès au type d'un service requis ou fourni, ainsi qu'à l'identité des composants liés par la dépendance. Ce niveau de réification est cependant suffisant pour mettre en œuvre une inversion de contrôle dynamique, permettant d'adapter l'application en cours d'exécution à la suite de certains événements tels que des pannes survenant dans l'environnement d'exécution. Cet aspect est étudié plus en détail dans la section suivante.

Le dernier aspect lié à l'inversion de contrôle porte sur la capacité, pour l'infrastructure, de gérer les communications depuis un composant vers les services non fonctionnels qu'il requiert. Cette capacité repose sur l'organisation du méta-niveau de Fractal en un ensemble de méta-objets (contrôleurs) propres à chaque composant. Cet ensemble est extensible, afin de permettre d'intégrer des propriétés non fonctionnelles additionnelles, en définissant de nouveaux contrôleurs qui gèrent les liens avec les services non fonctionnels concernés. Par exemple, un composant persistant (resp. transactionnel) est alors un composant contenant un contrôleur de persistance (resp. transaction). [1]. Cette capacité permet de réaliser l'équivalent du tissage du contrat requis par une dépendance non fonctionnelle (voir section 2.2.2) par l'ajout d'un nouveau contrôleur. L'exploitation de l'organisation à base de contrôleurs pour le support de propriétés non fonctionnelles est abordée de manière approfondie dans [76], qui repose sur les techniques issues de la programmation par aspects pour combiner des aspects de contrôle dans des composants Fractal.

L'infrastructure pilote donc le cycle de vie des composants et la mise en place de leurs dépendances, à la fois au moment du déploiement et ensuite durant l'exécution de l'application. Ce pilotage est transparent pour le développeur de l'application, même si Fractal offre à ce dernier la possibilité de spécialiser, par surcharge, les traitements effectués au niveau des contrôleurs. Cette spécialisation s'appuie sur un système qui notifie à un composant les invocations des opérations de contrôle le concernant. Ce mécanisme permet de réaliser l'équivalent des *callbacks* métier présentées dans le chapitre précédent (voir section 2.2.2). A chaque invocation d'une méthode de contrôle, le code correspondant défini par le méta-niveau de Fractal est exécuté. Si le composant implémente l'interface correspondante à l'opération de contrôle, alors l'appel lui est notifié.

En sus du caractère spécialisable évoqué précédemment, l'inversion de contrôle mise en œuvre

[1] Il faut noter que dans la version actuelle de Jade, le support de propriétés non fonctionnelles est très limité et ne comprend que la gestion de la réplication

dans Jade intègre un caractére *autonomique*, car elle est capable de réagir à des événements affectant les dépendances sans nécessiter d'intervention humaine. Ce dernier point, qui constitue la contribution majeure de Jade, rend nécessaire l'usage d'une architecture réifiée riche. Une gestion autonome des dépendances signifie qu'à la suite de certains événements, Jade dispose de la connaissance suffisante pour analyser les impacts de l'événement sur l'état courant de l'architecture, puis décider d'un plan de rétablissement des dépendances affectées. Pour ce faire, il est tout autant nécessaire de pouvoir inspecter l'état courant de l'architecture que de pouvoir adapter cet état. En outre, les éléments réifiés dans cet état courant doivent refléter une connaissance minimale sur l'application administrée, permettant une prise de décision pertinente. Par exemple, lors d'une panne de machine, Jade doit pouvoir, par introspection du composant représentant la machine défaillante, obtenir la liste des composants qui s'exécutaient sur cette machine préalablement à la panne.

Les travaux que nous avons menés jusqu'à maintenant ont montré que la réification de l'ensemble des dépendances architecturales, du cycle de vie et de l'état des composants sont suffisants pour fournir des dépendances auto-protégées et auto-réparables. Durant la phase d'exécution, toute dépendance associée à des contraintes de protection est dynamiquement adaptée en fonction des événements signalant des intrusions dans le système [12].

De manière similaire, une dépendance peut défaillir à la suite de la panne d'un composant. Son rétablissement peut engendrer la création de nouveaux composants venant remplacer ceux tombés en panne et leur déploiement sur de nouvelles machines. Ainsi une dépendance liant un composant C1 à un autre composant C2 peut, à la suite d'une défaillance impactant C2, être automatiquement redirigée vers un composant C3 si C3 satisfait les contraintes associées à la dépendance. Le cas de l'auto-réparation des dépendances est traité plus en détail dans la section suivante.

3.4.4 Étude de cas

Notre étude de cas sera le déploiement d'une application répartie décrite au travers de l'ADL Fractal et l'application de l'auto-réparation à cette application. L'objectif d'un tel service est de détecter l'occurrence de pannes dans l'environnement d'exécution et d'en corriger les effets afin de conserver le même niveau de disponibilité pour l'application administrée. Les pannes que nous avons traitées jusqu'à présent sont les pannes franches de noeuds. La réparation d'une telle panne consiste à manipuler l'état courant de l'architecture pour la ramener à un état opérationnel. Différentes politiques de réparation peuvent être considérées, dont l'une consiste à ramener l'architecture à un état équivalent à celui précédant la panne.

Un service d'auto-réparation ne peut fonctionner que si les dépendances architecturales sont adaptables dynamiquement. Les manipulations effectuées au niveau de l'architecture consistent en effet principalement, en le remplacement de composants, de services ou de ressources défaillants par d'autres non défaillants. Tout élément remplacé nécessite l'adaptation des dépendances dans lesquelles il intervenait, afin de garantir la cohérence de la nouvelle architecture. Dans la figure ci-après, le remplacement de C2 par C3 implique de supprimer le lien de C1 vers C2, pour le remplacer par un lien de C1 à C3.

La suite de cette section décrit tout d'abord la définition d'une application Jade, montrant l'utilisation de l'ADL Fractal pour définir les composants et leur assemblage initial. Puis nous présentons les principes de mise en œuvre du processus d'auto-réparation dans Jade. Nous ter-

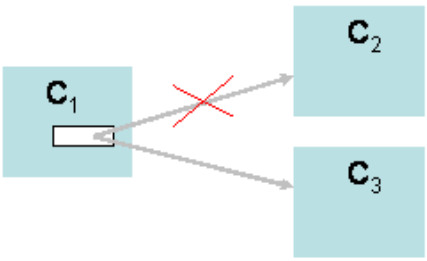

FIG. 3.10 – Exemple d'adaptation d'une dépendance

minons sur l'autonomie du service d'auto-réparation, qui est capable de s'auto-réparer.

Création de l'application La création d'une application avec Jade commence par la définition des composants puis de leur assemblage initial, le tout au travers de l'ADL Fractal, basé sur XML.

Listing 3.1 – Exemple de définition d'un composant dans Fractal

```
<component name="client">
  <interface name="r" role="server" signature="java.lang.Runnable" />
  <interface name="s" role="client" signature="Service" />
  <content class="ClientImpl" />
</component>
```

Listing 3.2 – Exemple de définition d'architecture dans Fractal

```
<definition name="HelloWorld">

  <interface name="r" role="server" signature="java.lang.Runnable"/>

  <component name="client">
    <interface name="r" role="server" signature="java.lang.Runnable"/>
    <interface name="s" role="client"  signature="Service"/>
    <content class="ClientImpl"/>
    <virtual-node name="remote-node1"/>
  </component>

  <component name="server">
    <interface name="s" role="server" signature="Service"/>
    <content class="ServerImpl"/>
     <virtual-node name="remote-node1"/>
  </component>

  <binding client="this.r" server="client.r"/>
  <binding client="client.s" server="server.s"/>
</definition>
```

La définition d'un composant explicite ses dépendances. Les dépendances fonctionnelles sont

exprimées par des balises nommées *interface*. Dans l'exemple donné dans le listing 3.1, un composant nommé client est défini. Il fournit une interface fonctionnelle de type *java.lang.Runnable*, et requiert une interface fonctionnelle de type *Service* pour fonctionner. Le composant client est primitif, et il est implanté par la classe *ClientImpl*.

L'architecture initiale d'une application Jade est également définie dans l'ADL Fractal, comme illustré dans le listing 3.2. Cet ADL définit une application composée de trois composants (nommés *helloworld*, *client* et *server*) liés à l'exécution par deux dépendances (*bindings*).

La description d'une architecture explicite également certaines dépendances non fonctionnelles. Celles vers les ressources de l'environnement sont exprimées, de manière simple, à l'aide d'étiquettes dédiées. Par exemple, il est possible de contraindre deux composants à être co-localisés en les associant à un nom de machine identique, éventuellement non résolu. Cet aspect est illustré dans l'exemple considéré, où les composants *client* et *server* doivent être déployés sur un noeud virtuel de nom *remote-node1* (voir la figure 3.2.). Les dépendances vers les services de l'infrastructure restent non explicitées en l'état actuel de Jade, mais devraient l'être dans des travaux futurs.

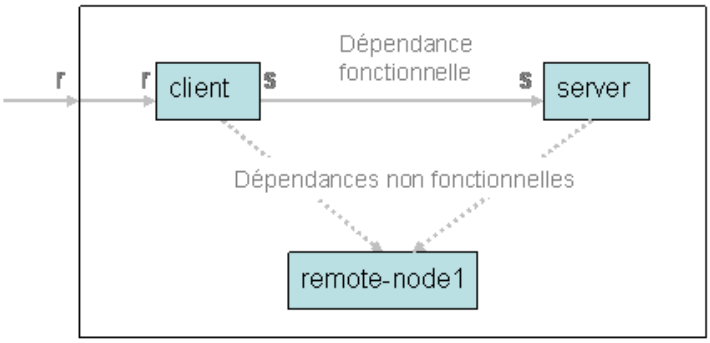

FIG. 3.11 – Architecture de l'application Fractal décrite dans le listing 3.2

L'auto-réparation Le processus de réparation exécuté après la détection d'une panne comprend deux étapes principales : l'analyse de l'état courant de l'architecture, et la réparation de cet état.

L'analyse de l'état courant de l'architecture identifie les éléments en panne et récupère les informations sur leur état et sur les liens mettant en œuvre leurs dépendances. Cette étape consiste en une introspection des composants de l'architecture concernés par la panne. Sont considérés le composant représentant la machine défectueuse ainsi que ceux correspondant aux composants qui s'exécutaient sur cette machine avant la panne. Les informations qui doivent être récoltées durant l'analyse sont fournies par l'architecture réifiée, c'est-à-dire par les méta-données des composants : liens, état du cycle de vie, valeurs des attributs, etc.

Pour pouvoir réaliser cette étape d'analyse, les méta-données des composants doivent rester disponibles après l'occurrence de pannes. Celles-ci étant co-localisées avec les composants, il y a besoin d'un mécanisme de type *checkpointing* qui fournisse, à tout instant, la dernière vue cohé-

rente des méta-données des composants. Ceci nous a conduit à une architecture à deux couches, comme illustré ci-après dans la figure 3.12.

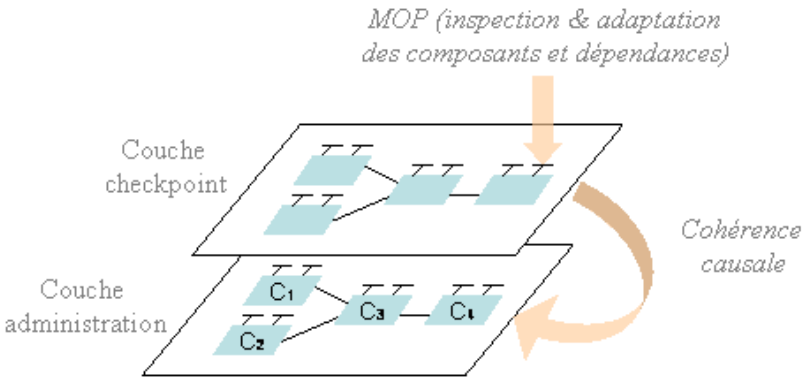

FIG. 3.12 – Architecture d'administration

Dans cette figure, la couche *administration* contient les composants applicatifs ainsi que ceux représentant les ressources de l'environnement d'exécution et les services de l'infrastructure. Les composants de la couche *checkpoint* fournissent une duplication du méta-niveau des composants de la couche administration. L'invocation de toute méta-opération sur un composant (par exemple, *addSubComponent(..)*) passe par la couche *checkpoint*, qui le répercute ensuite sur la couche *administration* (en établissant la relation de composition correspondante), afin de garantir une cohérence entre les deux niveaux.

A l'aide des méta-données récoltées durant la phase d'analyse, la phase de réparation substitue les composants en panne par des nouveaux composants puis résout les dépendances vers ces nouveaux composants. Cette phase consiste en une manipulation des composants et de leurs dépendances au niveau de la couche *checkpoint*, avec répercussion sur la couche *administration*. Tous les types de liens entre les composants sont considérés. Si un composant en panne avait un lien vers un autre composant, ce lien doit être fermé et reconstruit avec le nouveau composant venant substituer celui en panne. De la même manière, lorsqu'un composant en panne était un sous-composant d'autres composants, la relation de composition doit être reconstruite en conséquence.

Le principe de la réparation d'un composant est donc de le remplacer par un composant de même type, venant prendre la place du défaillant dans l'assemblage global. L'hypothèse qui est faite dans Jade est en effet que les contraintes associées aux dépendances restent satisfaites si l'on remplace le composant cible d'une dépendance par un autre de même type. Cette hypothèse est réaliste, mais peut être restrictive car il n'est pas toujours possible de recréer un nouveau composant de même type. C'est le cas lorsque le composant défaillant est un noeud, et que tous les noeuds de même type sont déjà utilisés dans le système. C'est également le cas lorsque le composant représente un service de l'infrastructure qui n'est plus disponible pour des raisons diverses, liées par exemple à des capacités mémoire insuffisantes. Dans un contexte futur, il serait nécessaire de revisiter cette hypothèse et de considérer que la réparation des dépendances peut mettre en jeu des processus plus complexes de résolution des contraintes associées aux dépendances.

L'autonomie de l'auto-réparation Un service d'auto-réparation complètement autonome doit non seulement réparer les pannes impactant l'application administrée, mais également ses propres pannes. Pour mettre en œuvre un service de réparation capable de s'auto-réparer, nous avons du rendre hautement disponibles d'une part le processus de réparation, et d'autre part les méta-données manipulées par ce processus.

Une solution classique pour la haute disponibilité est d'utiliser de la redondance au niveau des données et des traitements. En répliquant à la fois le processus de réparation et la couche checkpoint, un service de réparation tolérant les fautes peut être construit. Toutefois, le nombre de pannes toléré reste limité par la cardinalité des répliques. Cette limitation n'est pas acceptable pour un service d'auto-réparation, car à chaque panne elle implique l'intervention d'un administrateur humain pour ré-établir cette cardinalité.

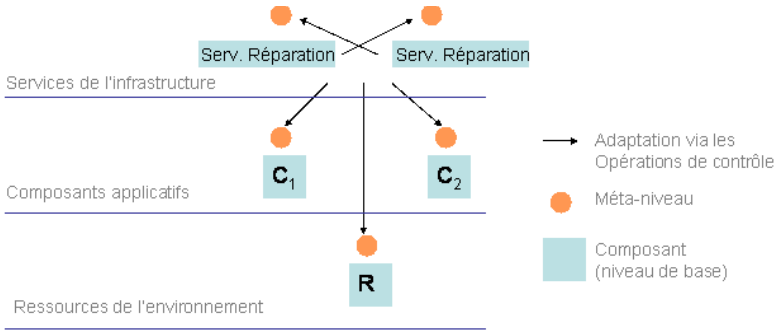

FIG. 3.13 – Structure répliquée du self-repair dans Jade

Nous avons adopté une approche dans laquelle chaque réplique du service d'auto-réparation est un composant. De ce fait, chaque réplique est visible au niveau de l'architecture réifiée, et par suite, automatiquement réparée par le service d'auto-réparation en cas de panne, comme illustré dans la figure 3.13. Ce choix permet de se ramener à une implémentation non récursive, dans laquelle chaque réplique du service de réparation s'applique aux autres répliques à l'exception d'elle-même.

Les composants qui correspondent à des répliques du service d'auto-réparation ainsi que ceux qui forment la couche checkpoint sont représentés dans l'architecture réifiée, qui délimite la zone sur laquelle s'applique le service d'auto-réparation. Cette zone est représentée par la couche *administration* dans la figure 3.14. Chaque réplique du service d'auto-réparation applique l'algorithme de détection-réparation. Par conséquent, chaque réplique détecte et répare les pannes impactant tout composant représenté dans l'architecture réifiée, y compris les autres répliques du service d'auto-réparation.

Revenons sur les répliques des composants de la couche checkpoint qui sont visibles dans la couche administration. Comme la couche checkpoint reflête la couche administration, les répliques de la couche checkpoint sont donc visibles dans la couche checkpoint. Pour prévenir une

récursion infinie, les méta-opérations de la couche checkpoint ont été spécialisées pour ne pas être répercutées sur une couche checkpoint de niveau supérieur (checkpoint du checkpoint).

Pour gérer la création de composants dupliqués, nous avons choisi d'utiliser des composants *usine* dupliqués. Autrement dit, nous avons appliqué la duplication sur un composant usine de façon à créer une usine dupliquée. Cette usine est alors mise en œuvre par un ensemble de copies réparties sur un ensemble de machines prédéfinies dans Jade.

Pour supporter les communications entre composants répliqués, nous avons défini et mis en œuvre différentes formes de liaisons totalement fiables (one-to-many, many-to-one, many-to-many), sur la base d'un protocole de communication de groupe uniforme et atomique (JGroups, http ://www.jgroups.org/javagroupsnew/docs/index.html). Une description complète de ce travail peut être trouvée dans [38]. Une approche plus générale pour la construction de liaisons complexes entre composants peut être trouvée dans [32].

Sur la base de cette architecture répliquée, l'algorithme de réparation exécuté par le service d'auto-réparation après la détection d'une panne est schématisé dans le listing 3.3. Nous détaillons dans la suite chaque étape de cet algorithme afin de montrer que les opérations de contrôle fournies par les composants permettent de réaliser les adaptations nécessaires pour réparer une panne.

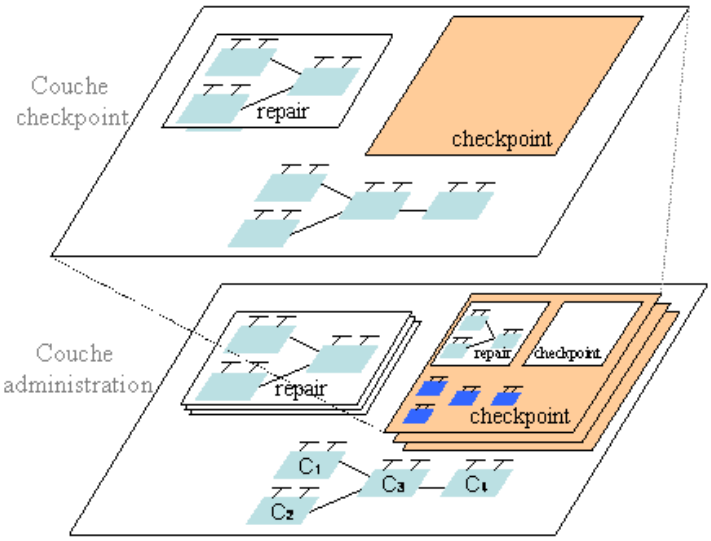

FIG. 3.14 – Architecture d'administration avec composants répliqués

L'analyse de la panne et la construction du plan de réparation utilisent la couche checkpoint pour introspecter l'état courant de l'architecture, et insérer dans le plan de réparation les références des composants en panne.

Le nettoyage du système implique de localiser et de détruire toutes les références vers des composants en panne, dans l'ensemble des composants du système. Cette étape est réalisée en introspectant le méta-niveau des composants en panne, afin de déterminer les identités des composants source pour chaque lien dont la destination est un composant en panne.

La destruction des liens est réalisée par l'utilisation de l'opération *unbind()* fournie par le méta-niveau de Fractal. Ces opérations sont spécialisables en fonction du code métier des composants, ce qui permet de réaliser les actions additionnelles qui seraient nécessaires (telles que la fermeture de sockets par exemple).

Listing 3.3 – Algorithme d'auto-réparation

```
1. Analyse the failure and build a repair plan
2. Clean up the global system (remove failed components)
3. Execute the repair plan (substitute failed components by newly ones)
```

Listing 3.4 – ETAPE 1 de l'algorithme d'auto-réparation

```
OBJECTIVE : Analyse the failure and build a repair plan

REQUIRE
    FailedNode : The reference of the component representing the failed
    node.

ENSURE
    FailedCmps : The references of the failed components
    RepairPlan : A repair plan composed of the description of the
                 components to repair (the defaut policy defines
                 the components to repair as those that were running
                 on the failed node)

ALGO
    FOR ALL cmp in FailedNode.getSubComponents()
        FailedCmps.addCmp(cmp)
        RepairPlan.addCmp(cmp)
    ENDFOR
END
```

Finalement, la dernière étape consiste à remplacer les composants défaillants par de nouveaux composants. Les caractéristiques architecturales des composants défaillants ont été enregistrées dans le plan de réparation. Ces informations décrivent les dépendances (liaisons), état du cycle de vie, valeurs des propriétés, etc. Elles permettent de substituer chacun de ces composants par un nouveau ayant des caractéristiques équivalentes. Ainsi, tout nouveau composant se voit automatiquement attribuer les liaisons du composant défaillant qu'il remplace, et vient donc "prendre sa place" dans l'architecture réifiée.

Listing 3.5 – ETAPE 2 de l'algorithme d'auto-réparation

```
OBJECTIVE : Clean up the global system

REQUIRE
    FailedCmps: The references of the failed components

ENSURE
    All relationships (binding, containment) involving a failed
    component are closed and removed

ALGO
    FOR ALL cmp in FailedCmps
        // for all alive component, remove a failed binding
        FORALL itf in cmp.getServerInterfaces()
            FORALL clientItf in itf.getBindingSources()
                IF clientItf.owner() not in FailedCmp
                    clientItf.unbind()
            ENDFOR
        ENDFOR
        // for all alive component, remove a failed child
        FORALL parentCmp in cmp.getParentCmps()
            IF parentCmp not in FailedCmp
                parentCmp.removeSubComponent(cmp)
        ENDFOR
        // for all alive component, remove a failed parent
        FORALL subCmp in cmp.getSubComponents()
            IF subCmp not in FailedCmp
                subCmp.removeParent (cmp)
        ENDFOR
   ENDFOR
END
```

Listing 3.6 – ETAPE 3 de l'algorithme d'auto-réparation

```
OBJECTIVE : Execute the repair plan

REQUIRE
    RepairPlan : a repair plan as returned by Etape 1,
    FailedNode : the reference of the component representing the
                 failed node
ENSURE
    The failed components are replaced by newly ones having the
    same management state
ALGO
    newNode = NodeAllocator.replace(FailedNode)
    FORALL cmp in RepairPlan
        // Create an equivalent component (same attributes,
        // interfaces, relationships and life cycle state)
        // and deploy it on newNode
    ENDFOR
END
```

3.4.5 Réflexions sur Jade

Dans cette section, nous avons présenté l'infrastructure Jade, conçue et développée au sein de l'équipe Inria/Sardes de 2004 à 2009. Cette infrastructure vise à supporter le développement, le déploiement et l'administration d'applications réparties programmées dans le modèle à composants Fractal.

Les applications patrimoniales peuvent également bénéficier des fonctions fournies par Jade, à condition d'envelopper leurs éléments logiciels dans des composants Fractal. Les applications gérées par Jade sont initialement décrites comme un assemblage de composants dans un ADL. Celui-ci permet d'expliciter les dépendances fonctionnelles qui lient les composants entre eux, ainsi que certaines dépendances non fonctionnelles telles que la co-localisation de certains composants.

À l'exécution, ces dépendances sont contrôlées par l'infrastructure Jade au travers du principe d'inversion de contrôle, décrit dans le chapitre précédent. La mise en œuvre de ce principe repose sur les fonctions réflexives fournies par le modèle à composants Fractal. Grâce à ces fonctions, l'adaptation des dépendances est possible à l'exécution. Une dépendance peut en effet être manipulée à tout moment de la vie d'une application, puisque les caractéristiques associées à cette dépendance sont accessibles et modifiables dynamiquement. Cet aspect met en contraste Jade avec des infrastructures plus classiques telles que JEE, qui ne manipulent les dépendances qu'au moment du déploiement d'une application.

Les capacités d'adaptation dynamiques des dépendances sont intéressantes mais restent difficiles à exploiter par un administrateur humain. Pour diminuer cette complexité, Jade place les dépendances sous le contrôle de services autonomes, basés sur des boucles de commande. Parmi les différents services proposés, nous pouvons citer le service de déploiement qui se charge de résoudre les dépendances, le service d'optimisation qui optimise les dépendances des composants vers les ressources de l'environnement d'exécution ou encore le service de réparation qui répare les dépendances défaillantes.

Plus largement, cette gestion autonomique des composants et de leurs dépendances a été l'une des contributions majeure et précurseure de Jade au domaine de l'administration autonome [11]. Le point central est l'exploitation de la connaissance de l'architecture pour mettre en œuvre des fonctions d'administration autonomes. D'autres projets tels que Rainbow [120], Darwin [63][64], xADL [78], poursuivent également cet objectif. A la différence de Jade, ces projets se basent sur des modèles à composants non réflexifs, et utilisent des mécanismes plus ou moins ad hoc pour introspecter et manipuler l'assemblage des composants. L'usage de techniques réflexives dans le contexte de l'administration autonome est considéré dans des projets comme OpenORB [43] ou Plastik [122]. Toutefois, contrairement à Jade, l'infrastructure d'administration fournie par ces projets n'est pas capable de s'auto-administrer. De ce fait, la présence d'un administrateur humain reste indispensable pour configurer, déployer et superviser l'infrastructure d'administration.

Pour fournir une infrastructure capable de s'auto-administrer, les services de l'infrastructure Jade (déploiement, optimisation, réparation, etc.), sont représentés par des composants Fractal. Ils obtiennent dès lors les capacités d'introspection et d'intercession fournies par Fractal. Ceci permet de définir une organisation récursive de Jade dans laquelle chaque service bénéficie des fonctions fournies par les autres services. Chaque service peut donc être auto-déployé, auto-optimisé, auto-réparé, etc. Afin d'assurer la haute disponibilité de l'infrastructure Jade, le

service de réparation est conçu de manière à pouvoir s'appliquer récursivement à lui-même. Le principe est de dupliquer ce service, en garantissant que chaque réplique est capable de réparer toute autre réplique. Cette structure de conception récursive, qui marque l'une des spécificités de Jade, a été validée par des expérimentations poussées dans des contextes applicatifs réels tels que JEE [68] ou JMS [96].

Bien que ces expérimentations aient conforté nos choix de conception, elles ont également mis en évidence certains manques qui devraient être considérés dans le futur de Jade. Une première extension, indispensable pour conforter l'utilisabilité de Jade dans un contexte réel, concerne la coordination des adaptations qui sont déclenchées par l'exécution de différents services d'administration. Actuellement, ceux-ci peuvent en effet exécuter des actions conflictuelles mettant en péril la cohérence de l'architecture. L'usage de mécanismes de synchronisation basiques, tels que l'exclusion mutuelle entre l'exécution des processus d'adaptation, n'est pas pertinent car certains processus tels que l'auto-réparation sont prioritaires par rapport à d'autres.

Les autres manques que nous identifions concernent la gestion des dépendances. Tout d'abord, les dépendances des composants vers des services non fonctionnels indépendants des aspects purement administratifs, tels qu'un service de transaction, de persistance ou de duplication n'ont pas été considérées jusqu'à présent. Ce manque incombe aux objectifs initiaux de Jade, qui étaient d'administrer des applications patrimoniales embarquant leurs propres services non fonctionnels. Ces objectifs ont ensuite été élargis au support d'applications à composants (basées sur Fractal). De plus, comme nous l'avons vu dans le présent chapitre, Jade est conçu à base de composants qui requièrent certains services non fonctionnels. C'est par exemple le cas du service d'auto-réparation qui a besoin d'un service de duplication. La gestion des dépendances entre les composants qui forment les services de l'infrastructure devrait être supportée par Jade, et ce au travers des mêmes mécanismes que ceux gérant les dépendances au niveau des composants applicatifs.

Un autre aspect qui n'a pas été considéré jusqu'à présent concerne la portée des services d'administration fournis par l'infrastructure, tels que le service d'auto-protection, celui d'auto-réparation, etc. Par défaut, ceux-ci s'appliquent à tous les composants applicatifs et à leurs dépendances. Nous pensons que, lorsqu'un composant est soumis au contrôle d'un service d'administration donné, ce contrôle devrait être modélisé par une dépendance au niveau de l'ADL. Cette modélisation permettrait à l'architecte de sélectionner, au niveau architectural, quels services d'administration sont requis par un composant.

Concernant les dépendances non fonctionnelles des composants vers les ressources de l'environnement, nous pensons que le support actuel est trop faible. Il se limite à la description de dépendances de type *co-localisation* entre composants. Il faudrait permettre d'exprimer des dépendances sous une forme plus générale, portant par exemple sur les caractéristiques des machines pouvant héberger un composant.

Plus globalement, l'ADL actuel ne permet pas d'exprimer des contraintes d'ordre sémantique sur les dépendances. Seul le niveau 1 des quatre niveaux de contrats proposés dans [3] est pour l'instant considéré. Il n'est par exemple pas possible d'exprimer des contraintes portant sur les versions des services fournis, ou sur la qualité de service assurée, telle que la fiabilité, les performances ou la sécurité. Or l'expression de contraintes de ce type est nécessaire pour supporter les applications qui intègrent un ensemble de fonctions sans connaître, a priori, les éléments logiciels qui fourniront ces fonctions. Des avancées en ce sens ont été proposées dans le projet ProActive

[70], en particulier pour prendre en compte des contraintes de sécurité.

Enfin, il est important de noter que les informations supplémentaires que nous proposons d'ajouter à l'ADL pour améliorer la définition des dépendances doivent être réifiées à l'exécution. La réification actuelle des dépendances est trop restreinte, et ne couvre pas l'ensemble des informations exprimées aujourd'hui dans l'ADL. Par exemple, les contraintes de co-localisation ne sont considérées qu'au moment du déploiement et ne sont pas enregistrées dans les contrôleurs. En conséquence, lorsque dans un assemblage donné, un composant C1 est co-localisé avec un composant C2 qui devra être remplacé par un autre composant C3 durant l'exécution, rien n'empêchera C3 d'être distant de C1.

3.5 Conclusion

Ce chapitre a présenté les principes des systèmes réflexifs et leur application dans le cadre des modèles à composants dans le but d'obtenir des applications adaptables dynamiquement.

Le principe des systèmes réflexifs est d'introduire un niveau supplémentaire d'abstraction (appelé méta-niveau) permettant de contrôler les concepts de base fournis par un système. Par un processus appelé réification, ces concepts peuvent recevoir une réalisation concrète qui permet de les inspecter et de les manipuler. Lorsque la réflexivité est dynamique, ils peuvent être inspectés et manipulés durant l'exécution.

Selon la nature des concepts réifiés, la réflexivité fournie est qualifiée de structurelle ou comportementale. Une réflexivité structurelle permet de manipuler la structure d'une application telle que les graphes d'héritage ou de composition. La réflexivité comportementale s'attache aux aspects de calculs et de comportements, tels que les invocations de méthodes.

Dans le contexte des modèles à composants, l'usage d'une réflexivité dynamique structurelle permet de réifier les dépendances, dans le but de pouvoir les inspecter et les adapter durant l'exécution. Pour obtenir un niveau de contrôle suffisant sur les dépendances, la réflexivité structurelle est combinée avec une réflexivité comportementale, qui permet de contrôler les opérations de gestion du cycle de vie et d'invocation des composants.

Un aspect important pour obtenir l'adaptabilité souhaitée est d'utiliser un modèle à composants réflexifs pour représenter non seulement les composants applicatifs mais également les ressources de l'environnement d'exécution et les services de l'infrastructure. Dans ce cas, les dépendances réifiées par le méta-niveau du modèle à composants sont aussi bien fonctionnelles, liant les composants applicatifs entre eux, que non fonctionnelles, liant les composants aux services de l'infrastructure ainsi qu'aux ressources de l'environnement d'exécution.

L'usage d'un modèle à composants réflexifs permet d'adapter les dépendances dynamiquement. En complémentarité avec les capacités de modification d'une dépendance fournies par la couche réflexive, la possibilité de connaître dynamiquement ses caractéristiques par introspection est indispensable pour résoudre celle-ci à nouveau lorsqu'elle devient inopérante ou défaillante. Si l'on considère par exemple une dépendance fonctionnelle liant un composant C1 à un autre composant C2, la structure réflexive doit permettre de connaître quel est le service attendu au niveau de C2, avec quelles propriétés (par exemple, version), quelles contraintes de localisation, etc. Sur la base de ces informations, l'infrastructure ou l'administrateur a la capacité de modifier

cette dépendance de manière cohérente, en remplaçant C2 par un composant C3 qui satisfait les caractéristiques demandées.

Les capacités d'adaptation fournies par une structure réflexive restent cependant d'ordre mécanique. L'exploitation de ces capacités par un administrateur humain est difficile, car celui-ci doit savoir quelles adaptations doivent être appliquées pour chaque type d'événements pouvant se produire dans le système. À cet égard, le couplage des mécanismes réflexifs avec des boucles de commande autonomes paraît tout à fait pertinent. Les capacités d'introspection permettent en effet de fournir la connaissance requise par les boucles, et celles d'intercession fournissent les leviers permettant d'agir sur l'état du système.

Par cette association, les dépendances architecturales, qui représentent des points d'adaptation critiques dans l'administration d'une application, sont gérées de manière autonome par les boucles de commande. Dans Jade par exemple, nous avons placé sous le contrôle d'une boucle d'auto-optimisation les dépendances non fonctionnelles liant les composants aux ressources de l'environnement. Nous avons également défini une boucle d'auto-réparation qui se charge de réparer une dépendance défaillante, fonctionnelle ainsi que non fonctionnelle.

L'approche autonome n'est valide que si les composants et les dépendances gérés de manière autonome sont non seulement ceux de niveau applicatif, mais également ceux mettant en œuvre les services formant l'infrastructure. Dans le cas contraire, l'infrastructure représente un point de défaillance potentiel qui peut amener le système à un état incohérent. La survenance brutale d'une panne au niveau d'un service impose alors la présence d'un administrateur pour redémarrer ce service, le reconfigurer et rétablir ses dépendances conformément au précédent service.

Lorsque les services de l'infrastructure sont représentés par des composants, on obtient la capacité de les gérer de manière autonome. Dans cette approche, les boucles de commande sont considérées comme des services de l'infrastructure, qui s'auto-administrent. Les travaux que nous avons menés sur Jade ont permis de valider cette approche, dans le contexte d'applications Internet de type e-commerce ou forums. Nous considérons que le domaine d'utilisation de cette approche devrait être élargi pour la valider plus amplement, en considérant d'autres contextes comme les applications pair à pair, les applications de calcul sur les grilles ou encore le domaine de l'embarqué.

Chapitre 4

Conclusions et perspectives de recherche

Dans ce document, nous avons présenté nos réflexions sur l'adaptabilité des applications réparties. Celles-ci s'appuient sur les travaux de recherche que nous avons menés, dont une partie sont présentés dans les chapitre 2 et 3, une liste plus exhaustive étant fournie dans un document séparé. Les classes d'applications réparties que nous avons considérées recouvrent toutes celles qui nécessitent d'être adaptées, durant leur cycle de vie, à des besoins applicatifs évoluant et à des conditions d'exécution changeantes.

Bien que nous nous soyons plus particulièrement focalisés sur les applications de type entreprise dans nos travaux de recherche, la problématique de l'adaptabilité s'applique à beaucoup d'autres domaines comprenant les applications mobiles, les applications s'exécutant sur grille ou sur réseaux pair à pair, ou encore les systèmes ubiquitaires et les applications embarquées. Les réflexions présentées dans ce document, portant sur les principes d'administration basée sur l'architecture, doivent être considérées comme pouvant s'appliquer en tout ou partie à ces différents domaines.

Les applications réparties sont aujourd'hui soumises à un besoin d'adaptabilité qui conditionne leur degré de réutilisation ainsi que leur capacité à être administrées. L'adaptabilité concerne principalement la possibilité de faire évoluer l'architecture d'une application répartie, à n'importe quel moment du cycle de vie (par exemple, lors du développement, du déploiement, ou de l'exécution). Nous avons abordé le concept d'architecture sous un angle général, qui englobe les éléments logiciels et matériels relevant à la fois de l'application, de l'infrastructure et de l'environnement d'exécution. De manière plus précise, l'architecture réifie l'assemblage de tous ces éléments au travers des différents types de relations qui les lient. Ainsi, elle donne une vision globale des éléments qui contribuent à la mise en œuvre et à l'exécution d'une application. C'est cette vision globale qui permet d'atteindre l'adaptabilité requise par les applications réparties.

L'adaptabilité, telle que nous la considérons dans ce document, correspond à la capacité de manipuler aussi bien l'architecture d'une application répartie que celle de l'infrastructure distribuée qui l'héberge. Cette adaptabilité s'exprime donc par la reconfiguration de l'architecture, sans impacter le code métier de l'application, c'est-à-dire sans avoir à revenir sur les composants déjà programmés. Ces reconfigurations architecturales peuvent être faites statiquement ou dynamiquement. Statiquement, l'adaptabilité nécessite l'arrêt de tout ou partie de l'application

et de l'infrastructure, ce qui est plus simple si l'on peut se permettre un tel arrêt. Cependant, la tendance va vers une disponibilité accrue qui suggère d'adopter une approche dynamique. L'adaptabilité est dite dynamique lorsque les transformations d'architecture peuvent être mises en œuvre durant l'exécution de l'application, sans avoir à stopper ni l'application, ni l'infrastructure.

Ainsi, durant toute la durée de vie de l'application, l'adaptabilité sera utilisée pour conserver un certain niveau d'exigence portant sur les aspects fonctionnels et non fonctionnels de l'application et de l'infrastructure. Ce besoin d'adaptabilité s'explique par l'évolution constante des conditions opératoires, des solutions et des besoins.

L'évolution des conditions opératoires dénote les changements portant sur les éléments matériels et logiciels du système global. Par exemple, l'administration de l'infrastructure impose retraits et ajouts de machines. Les pannes matérielles et logicielles modifient aussi les conditions opératoires de façon impérieuse. Sans aller jusqu'à la panne, les surcharges impactent fortement la qualité de service et ce totalement dynamiquement. Face à ce type d'évolution, l'adaptabilité doit permettre de reconfigurer l'architecture pour garantir le respect des critères techniques de l'application, tels que le niveau de disponibilité ou les performances d'exécution attendus.

L'évolution des solutions et des besoins peut se situer au niveau fonctionnel ainsi qu'au niveau non fonctionnel. Au niveau fonctionnel, on veut pouvoir intégrer dans l'architecture de nouveaux éléments logiciels contribuant à la mise en œuvre de l'application. Ces éléments peuvent venir remplacer certains déjà présents, comme par exemple lors de l'évolution de versions, ou ils peuvent correspondre à des éléments additionnels répondant à des besoins nouveaux des utilisateurs. L'évolution des aspects non fonctionnels concerne plus particulièrement l'infrastructure qui joue un double rôle : de médiateur entre les éléments logiciels et matériels, et de fournisseur de services non fonctionnels communs. A ce double titre, elle est confrontée à l'apparition de nouvelles technologies ou de nouveaux services. L'adaptabilité doit permettre de les intégrer de manière à en faire bénéficier les applications en cours.

Dans la suite de cette conclusion, nous revenons sur les principes des solutions permettant de répondre à ce besoin d'adaptabilité, puis nous résumons nos contributions dans ce domaine. Nous concluons enfin avec nos perspectives de recherche.

4.1 Evolution des solutions pour l'adaptabilité

Si l'on considère l'adaptabilité comme la capacité d'agir sur l'architecture d'une application répartie sans modifier son code métier, alors nous devons constater que cette capacité a été exploitée dès les premières avancées dans le domaine des applications réparties.

Les environnements à objet répartis, qui ont permis de structurer une application sous la forme d'objets invocables à distance, ont marqué une étape majeure dans la programmation des applications réparties. En effet, ces environnements ont permis de rendre la distribution transparente au niveau programmatique, au travers de la notion de référence d'objet réparti qui est manipulable et transmissible comme une référence centralisée. La mise en œuvre de cette notion repose sur l'exploitation de techniques d'adaptabilité, car l'architecture logicielle effective de l'application est en effet adaptée par l'infrastructure, qui insère au sein de cette architecture des objets agissant, à l'exécution, comme des représentants locaux d'objets distants.

Une généralisation de cette notion de représentant a ensuite été introduite sous la forme d'intercepteurs, correspondant à des objets insérés dans l'architecture et chargés d'adapter une invocation de méthode pour l'enrichir de propriétés non fonctionnelles ou bien pour considérer certaines spécificités de l'objet demandeur ou de l'objet serveur. Le mécanisme d'intercepteurs de la technologie Corba [47], développée sous l'égide de l'OMG, en est un exemple. Les intercepteurs sont des objets qui sont insérés dans l'architecture au moment de la compilation ou du chargement. Durant l'exécution, s'ils permettent d'adapter le comportement de l'application, ils ne permettent pas, en tant que tels, de reconfigurer son architecture.

Les intercepteurs ont cependant été utilisés comme un moyen de mettre en œuvre une réflexivité dynamique, l'une des techniques de base pour obtenir des capacités d'adaptabilité durant l'exécution. Les environnements à objets réflexifs permettent de programmer le code fonctionnel des objets, puis d'adapter leur gestion et leurs propriétés non fonctionnelles au niveau réflexif, séparément de leur code métier. Ces environnements sont basés sur un principe d'organisation logicielle à deux niveaux (base et méta), dans laquelle le niveau méta permet de consulter et de modifier la structure et/ou le comportement des objets du niveau de base, correspondant aux objets métier. La puissance fournie par les techniques réflexives reste cependant difficile à exploiter dans un cadre général, en l'absence de support fournissant une vision claire de l'architecture de l'application que l'on souhaite adapter.

Il faut noter que l'usage de techniques réflexives pour obtenir des capacités d'adaptabilité va au-delà du domaine des applications réparties. Ainsi les approches fondées sur les systèmes d'exploitation réflexifs tels que Flexinet [5] et Apertos [61] exploitent ces techniques. La réflexivité fournit par ailleurs une base de mise en œuvre pour la programmation par aspects [30] qui est une autre approche pour l'adaptabilité, axée sur la programmation par séparation des préoccupations et permettant d'enrichir une application avec des aspects non fonctionnels a posteriori par rapport à la phase de programmation.

Parallèlement aux techniques réflexives et à la programmation par aspects, les environnements à composants se sont imposés pour la construction et la mise en œuvre des applications réparties. Les composants sont des unités de programmation, d'assemblage et de déploiement qui explicitent leurs dépendances vis-à-vis des autres composants ainsi que vis-à-vis des services non fonctionnels qu'ils requièrent. Les dépendances explicites permettent d'avoir une vision claire de l'architecture logicielle d'une application. Sur la base de cette architecture, la plupart des environnements à composants existants et établis commercialement (JEE[68], Spring[59], PicoContainer[1], etc.) adaptent une application au moment de son déploiement, pour tenir compte de la configuration de l'environnement d'exécution réparti.

Le principe de conception utilisé pour obtenir cette adaptabilité est l'inversion de contrôle, qui signifie que les dépendances des composants sont contrôlées par l'infrastructure et non par eux-mêmes. Le mécanisme de mise en œuvre de l'inversion de contrôle dans les environnements à composants établis repose cependant trop souvent sur une approche pragmatique qui utilise des intercepteurs et des techniques élémentaires de tissage de code. Plus récemment, la programmation par aspects est venue offrir une meilleure modélisation de l'inversion de contrôle. Avec une bonne pratique, elle permet un tissage puissant de code modulaire et minimal dont la fonction est d'adapter de manière transparente le code métier aux services de l'infrastructure.

Le premier grand défi qui demeure reste le besoin d'adaptabilité dynamique. Que ce soit

par l'approche pragmatique ou par les aspects, l'adaptabilité dynamique n'a que peu de réalité aujourd'hui même si certaines solutions techniques existent. L'une des causes est le manque de garanties pour manipuler l'architecture dynamique de manière cohérente, sans interférer avec les exécutions métier en cours. Par ailleurs, la connaissance des propriétés et des contraintes de qualité de service de l'application doit être suffisamment riche pour empêcher la dégradation des fonctions métier fournies lors des adaptations. Toutes ces contraintes et propriétés doivent être explicitées au niveau de l'architecture par un mécanisme systématique.

Cette approche systématique peut être abordée par l'usage combiné des composants et de la réflexivité dynamique. En effet, cette réflexivité fournit un moyen pour réifier l'architecture de l'application à l'exécution, permettant à la fois d'introspecter et de manipuler cette architecture. Si cette architecture est suffisamment riche, elle fournit la connaissance requise pour adapter l'application de manière cohérente. Nous tombons alors dans le domaine de l'*administration basée sur l'architecture*, dont le principe est d'exploiter la présence d'une architecture explicite et manipulable pour administrer une application répartie dans un environnement d'exécution donné.

Le second grand défi de l'adaptabilité reste sa complexité croissante. Toute la puissance des composants réflexifs et de l'administration basée sur l'architecture reste en effet difficile à exploiter par des administrateurs humains, de par la quantité et la variété des événements nécessitant d'opérer des adaptations architecturales. De ce fait, la tendance est de compléter l'action des administrateurs humains par des processus d'adaptation autonomes. Ceux-ci déclenchent de manière automatique les adaptations nécessaires au niveau de l'architecture, en réaction à certains événements affectant l'application, l'infrastructure ou l'environnement d'exécution.

Du point de vue de l'adaptabilité, la complémentarité des composants, de la réflexivité et de l'approche autonome reste très prometteuse et source de perspectives de recherche intéressantes. Dans cette combinaison, la réflexivité enrichit les composants de capacités d'adaptabilité, ce qui fournit une base pour la mise en place de processus d'administration autonomes qui, en retour, permettent de mieux maîtriser la puissance de ces capacités.

4.1.1 Contributions

Mes travaux de recherche se sont majoritairement inscrits dans le cadre de la gestion de l'adaptabilité des applications réparties. J'ai plus précisément travaillé autour de l'adaptabilité des applications à base d'objets répartis au travers du projet Guide [102][35][33]. Mes travaux se sont ensuite tournés vers les modèles à composants, dans le cadre du projet Olan [71][72], dont les contributions principales sont rappelées ci-après.

L'adaptabilité apparaît alors comme une problématique cruciale dans le domaine des applications réparties, et j'ai cherché à évaluer l'usage de techniques réflexives pour répondre à cette problématique, au sein du projet Sirac [114][37][36][24][125]. Enfin, sur la base de l'expérience acquise dans le domaine des objets répartis, des composants et de la réflexivité, j'ai souhaité aborder, au travers du projet Jade [112][115][118], le problème de l'adaptabilité dans les applications réparties sous un angle plus large, plus systématique, et associé à l'usage de techniques d'administration autonomes.

Je récapitule ci-après les contributions des travaux liés aux projets Olan et Jade, que j'ai considérés comme les plus significatifs de mon parcours. Les contributions des autres travaux sont résumées dans un document séparé.

Composants (Olan) Dans le projet Olan, nous avons conçu un modèle d'interconnexion de composants patrimoniaux développés indépendamment les uns des autres. Ce modèle est basé sur les principes d'encapsulation et de description explicite de l'architecture d'une application. Nous avons également mis en œuvre une infrastructure supportant ce modèle d'interconnexion, gérant l'adaptabilité des interconnexions au moment du déploiement. Les interconnexions sont adaptées aux contraintes des composants liés (par exemple, langage de programmation, synchronisation), ainsi qu'à la configuration répartie de l'application.

En termes de contributions, Olan a fait partie des premiers projets axés sur l'usage des composants logiciels dans le domaine des applications réparties, et en ce sens a il été précurseur au niveau des concepts et des outils associés aux composants, qui étaient encore peu adoptés à l'époque. En sus des aspects d'encapsulation et de dépendances explicites, nous avons défini un ADL pour décrire les interconnexions de composants.

Par ailleurs, Olan a montré l'importance et la complexité inhérentes à la gestion des interconnexions, qui représentent des points d'adaptation critiques dans les applications réparties. Pour gérer ces interconnexions au niveau de l'infrastructure, nous avons utilisé un mécanisme relevant de l'inversion de contrôle, basé sur des intercepteurs.

Enfin, l'adaptabilité à la configuration répartie d'une application, qui permet de déployer celle-ci avec différentes topologies distribuées, et ceci de manière totalement indépendante du code métier, a également représenté une contribution majeure. Cette adaptabilité a permis de déployer une même application dans des environnements d'exécution différents, composés de machines d'exécution dont le système fournissait des services non fonctionnels (par exemple, services de communication) différents d'un environnement à l'autre.

Dans une deuxième étape, à la suite d'un retour sur l'expérience acquise et mettant en évidence le manque d'adaptabilité dynamique des interconnexions supportées par Olan, nous avons revisité la conception de ce système et adopté une approche réifiant l'architecture répartie pour répondre à ce besoin. Ces capacités d'adaptation dynamiques ont permis de fournir un support pour la phase d'administration des applications [85].

Composants réflexifs (Jade) Nous avons proposé dans le projet Jade une infrastructure pour l'administration des applications réparties. Notre objectif était de fournir un support d'administration performant, permettant d'adapter les applications à chaud pour répondre aux besoins de disponibilité et de réactivité. L'approche que nous avons choisie pour l'adaptabilité est basée sur l'utilisation de composants réflexifs.

L'une des particularités de Jade est d'utiliser les composants réflexifs non seulement pour programmer le niveau applicatif, mais également pour modéliser l'infrastructure et l'environnement d'exécution sous-jacent. L'architecture réifiée reflète ainsi tous les types de dépendances liant les composants, c'est-à-dire celles de type fonctionnel liant les composants applicatifs entre eux, celles liant les composants applicatifs aux services non fonctionnels fournis par l'infrastructure, et enfin celles liant les composants aux ressources (par exemple, machines) fournies par l'environnement d'exécution. Autrement dit, l'architecture réifiée de Jade donne une vision globale et unifiée de tous les éléments qui contribuent à la mise en œuvre d'une application, allant de la couche applicative à la couche matérielle.

Cette vision unifiée de l'architecture est à la base de l'introduction de processus d'admi-

nistration autonomes, représentant une contribution importante de Jade. Ces processus ont la capacité de raisonner sur l'architecture réifiée et d'adapter celle-ci pour répondre aux exigences de l'application et de ses utilisateurs. Nous avons expérimenté avec succès l'approche autonome pour l'auto-déploiement, l'auto-réparation, l'auto-protection et l'auto-optimisation. Nous avons appliqué ces processus autonomes dans un contexte de grappes pour l'administration de serveurs JEE d'applications Web et d'intergiciels orientés message (en anglais Message-Oriented Middleware).

C'est au travers de cette expérience riche que l'organisation récursive du système Jade s'est imposée comme essentielle. Cette organisation, qui représente une autre originalité de nos travaux, intègre les processus d'administration autonomes à l'infrastructure. Ainsi l'ensemble des éléments administrés et des services administrants sont représentés dans l'architecture réifiée. Cela ouvre la possibilité d'une administration récursive puisque les services d'administration s'appliquent alors non seulement aux éléments administrés mais également à eux-mêmes, sans fracture technologique aucune.

Dans cette approche, l'infrastructure se présente comme une plate-forme distribuée adaptée au développement de services autonomes d'administration. Elle prône le même paradigme de composants réflexifs pour la conception des services d'administration que pour celle des éléments administrés. Ceci permet aux processus autonomes de s'auto-administrer avec les mêmes algorithmes que ceux employés sur les composants administrés.

Un exemple de cette organisation récursive est le service d'auto-réparation. À un premier niveau, ce service surveille les composants administrés et est chargé de réparer ceux qui tombent en panne. Il fournit ainsi un comportement autonome d'auto-réparation des composants administrés. À un deuxième niveau, ce service d'auto-réparation est lui-même développé sous la forme de composants et il peut donc s'appliquer à lui-même. Pour ce faire, nous avons adopté une mise en œuvre répliquée qui permet à chaque réplique de surveiller et de réparer les autres répliques. Nous obtenons ainsi, via l'organisation récursive de Jade, la capacité de résister à toute panne de composant, qu'il s'agisse d'un composant applicatif ou d'infrastructure, et ce de manière totalement autonome.

Plus globalement, Jade a été précurseur dans le fait de combiner les composants, la réflexivité et l'administration autonome. Les retours sur cette expérience sont largement positifs et ils ont mis en évidence le rôle majeur joué par l'architecture dans le succès de cette triple association. Ils ont également mis en lumière le fait que la richesse et la précision des concepts réifiés sont des critères majeurs pour permettre la définition de processus autonomes travaillant à grain fin.

4.2 Perspectives

Mes perspectives se situent dans le prolongement de mes travaux sur les modèles à composants adaptables. Elles visent, à moyen terme, la recherche de solutions abouties pour l'adaptation des applications réparties. En effet, l'évolution des systèmes répartis montre une importance croissante de l'adaptabilité dont la complexité de mise en œuvre reste problématique.

Il me paraît important de poursuivre dans la voie d'une adaptabilité contrôlée par des processus autonomes, mis en œuvre par des boucles de commandes manipulant l'architecture d'une application. Ce type d'adaptabilité va de pair avec l'usage de modèles à composants réflexifs qui

rendent explicite et manipulable l'architecture d'une application au travers d'une architecture réifiée.

Dans ce contexte, nous en sommes aux balbutiements des processus autonomes qui sont encore souvent abordés par le biais de solutions pragmatiques, fournissant une adaptabilité limitée. Nous pensons que l'approche actuelle pour le développement des boucles de commande est en cause, avec une programmation souvent spécifique s'appuyant sur une logique procédurale de bas niveau. Le premier défi, abordé en 4.2.1, est donc de permettre une expression plus aisée d'une expertise autonome.

Le second défi, décrit en 4.2.2, concerne la robustesse des adaptations mises en œuvre. Tout d'abord, il faut s'assurer qu'une adaptation laisse l'architecture dans un état cohérent. Il faut bien sûr se protéger des pannes habituelles telles que les pannes de machine ou de réseau. Il est également nécessaire de maîtriser les violations de contraintes d'intégrité complexes au sein même de l'architecture, ce qui peut demander de faire échouer tout ou partie d'une adaptation. Parallèlement à la préservation de la cohérence de l'architecture, il est tout aussi important d'assurer la disponibilité fonctionnelle du système que l'on adapte.

4.2.1 Vers l'expertise autonome de l'administration

Il semble inévitable que l'administration autonome évolue vers l'expression d'une expertise autonome de haut niveau, se rapprochant des systèmes experts et s'éloignant de la programmation procédurale de boucles de commande spécifiques.

Un premier facteur de complexité dans la définition de boucles autonomes vient du fait qu'elles sont écrites dans les langages de programmation classiques des modèles à composants et que les opérations réflexives fournies par ces modèles sont de bas niveau.

Un second facteur de complexité provient du fait que les événements susceptibles de déclencher des adaptations sont nombreux et concernent des aspects très variés du système surveillé. Bien que qu'une l'architecture réifiée fournisse un cadre uniforme pour observer ces événements, elle reste toutefois insuffisante face à l'hétérogénéité sémantique de ces événements.

Il est d'usage d'approcher cette hétérogénéité via le principe de la séparation des aspects, c'est-à-dire que chaque boucle de commande autonome est censée gérer une catégorie particulière d'événements. Cependant, ces catégories sont rarement orthogonales, ce qui suggère une coopération entre les boucles de commande qui reste difficile à exprimer et à mettre en place.

Bien que le domaine de l'expertise autonome soit récent, ces deux types de complexité ne le sont pas. Elles sont assez classiques des systèmes décisionnels complexes et ont trouvé des solutions en intelligence artificielle, en particulier dans les systèmes experts. Un défi majeur est donc la recherche d'un langage déclaratif de haut niveau permettant une expression aisée des réactions autonomes à des événements variés. Les événements ne sont pas figés et tous connus, il faut donc un moyen d'en définir de nouveaux et d'avoir un pont efficace entre le système sous-jacent qui génère ces événements et ce langage déclaratif où ils doivent être pris en compte.

Ce langage doit permettre d'exprimer des réactions en termes d'opérations de plus haut niveau que celles fournies par les modèles à composants réflexifs. L'intérêt de ces dernières est certainement dans leur minimalité, offrant un cadre uniforme pour l'introspection et la reconfiguration de

l'architecture d'une application. Mais la complexité d'expression des réactions autonomes suggère de pouvoir définir des opérations à plus gros grain, dont certaines mettent en œuvre une sémantique ensembliste. Nous proposons donc d'évoluer vers des composants dont le contrôle peut être spécialisé, à la manière de la spécialisation des méta-classes dans certains langages orientés objets.

Ainsi par exemple, certains composants métier peuvent être associés à une opération de haut niveau permettant de les déplacer d'une machine à une autre, alors que d'autres interdisent ce type d'opération. Ces opérations de contrôle de haut niveau sont spécialisées dans leur implémentation. Ainsi, le retrait d'un composant métier n'a que les effets de bords habituels alors que le retrait d'un composant réifiant une machine s'accompagne soit du retrait en cascade des composants qui y sont déployés ou bien de leur migration préalable sur d'autres machines.

Ces opérations de reconfiguration de haut niveau sémantique peuvent en outre capturer des contraintes architecturales complexes. Ainsi, par exemple, une opération de déploiement d'un composant sur une machine pourra n'être valide que si la machine est disponible et que ses quotas ne sont pas dépassés.

L'introduction d'opérations ensemblistes est également primordiale car les adaptations concernent fréquemment des ensembles de composants. Par exemple, un changement de version de composant peut s'exprimer par une opération ensembliste qui s'applique sur l'ensemble des composants d'un type donné. De même l'arrêt, ou tout autre opération, sur un composant répliqué est de fait une opération ensembliste sur l'ensemble des réplicas. Le mécanisme de composites hiérarchiques tel que celui fourni par Fractal est certainement un premier pas vers la définition d'opérations ensemblistes, mais son usage est encore limité.

Via l'accroissement du niveau sémantique des opérations de reconfiguration, il devient possible d'exprimer plus succintement une expertise autonome plus complexe. Cette expertise s'exprime en termes de règles, associant des réactions autonomes à des événements variés. L'approche à base de règles permet d'aborder la coopération entre réactions autonomes qui sont potentiellement antinomiques, au travers d'heuristiques et de priorités.

L'usage de règles place l'architecture réifiée au centre de la coopération des réactions autonomes. En effet, les règles ont besoin de s'appliquer sur une même architecture partagée et évolutive. Cela suggère que cette architecture réifiée soit découplée du système exécutoire pendant la phase de planification des adaptations. Puis, une fois acquise la convergence des adaptations qui s'appliquent, nous obtenons une architecture cible. A partir de celle-ci, nous pouvons déduire un plan de reconfiguration du système exécutoire de son architecture actuelle vers celle ciblée. Il est clair que l'exécution de ce plan incombe à l'infrastructure d'administration sous-jacente à l'application et non pas individuellement aux boucles de commande comme c'est le cas aujourd'hui dans beaucoup d'infrastructures. C'est au travers de cette exécution répartie qu'apparaît le besoin de robustesse, le second défi abordé dans mes perspectives et le sujet de la prochaine section.

4.2.2 Vers une robustesse accrue des architectures adaptables

Assurer la robustesse de l'architecture d'une application face à ses adaptations est un défi qui concerne différents aspects. Il s'agit tout d'abord de garantir que les adaptations font passer l'architecture d'un état cohérent à un autre. La cohérence des adaptations est du domaine de l'expertise, et se traduit par la production de plans de reconfigurations qui sont supposés être

corrects. Une fois cette hypothèse faite, il est essentiel de considérer l'exécution robuste de ce plan. Cela suggère une propriété d'atomicité qui s'appuie a priori sur une isolation des reconfigurations avec les exécutions en cours de fonctions métier.

Les systèmes transactionnels sont des exemples classiques d'environnements où l'isolation et l'atomicité sont combinées pour garantir la robustesse de l'exécution des transactions, qui définissent des transformations cohérentes de la base de données. On retrouve ici les classiques propriétés ACID (Atomicity, Consistency, Isolation, et Durability), où la durabilité est la propriété de non perte des reconfigurations commises même en cas de pannes.

Cependant, les systèmes transactionnels ont des limites bien connues. Tout d'abord, l'isolation à base de verrous ne fonctionne bien qu'avec peu de conflits et avec peu d'écritures. De ce point de vue, le domaine qui nous intéresse ici est problématique, car l'exécution d'une transaction de reconfiguration est totalement en écriture puisqu'elle modifie l'architecture de l'application en cours. Il est donc crucial de jouer sur la notion de conflit et d'assouplir au maximum le critère de sérialisation recherché.

Une approche que je souhaite investiguer consiste à augmenter l'architecture réifiée d'informations sur les exécutions métier, permettant d'optimiser la gestion des transactions de reconfiguration. Il est en effet peu probable que tous les composants métier d'une application s'exécutent à un instant donné. Si l'architecture capture assez d'information sur ces composants métier, il devient envisageable de paralléliser à grain fin certaines opérations de reconfiguration et certaines fonctions métier. L'idée est donc d'exploiter une architecture réifiée riche pour jouer sur l'ordonnancement entre opérations métier et opérations de reconfiguration.

Par ailleurs, pour satisfaire le besoin de disponibilité croissant des applications, il est crucial d'éviter une reprise complète des transactions de reconfiguration en cas de conflit en proposant des mécanismes de transactions incrémentaux [119]. Cela est problématique puisque cela rejoint une autre limite des systèmes transactionnels : leur vivacité dans un contexte distribué, où l'occurrence de pannes ou de délais empêche souvent la complétion des opérations atomiques.

Une direction qui semble prometteuse est la possibilité de définir des opérations inverses. Il se trouve que, d'après notre expérience, toutes les opérations de reconfiguration de bas niveau ont des opérations inverses. C'est au moins le cas dans le système Jade où nous utilisons déjà les opérations inverses pour l'atomicité. Des travaux de recherche sont cependant nécessaires pour étudier la faisabilité de cette approche lors du passage à des opérations de reconfiguration plus complexes.

4.2.3 Vers une nouvelle orientation de recherche

Ces perspectives suggèrent d'approcher l'administration des applications selon de nouvelles bases, fournissant un socle plus adéquat pour garantir la robustesse des adaptations. Cette démarche évoque un retour à la proposition originale de Paul Horn, initialement directeur mondial de la recherche chez IBM. En effet, la proposition initiale était de revoir la conception de nos systèmes informatiques en profondeur, à l'image de l'auto-réparation et de l'auto-régulation des systèmes biologiques.

Dans notre contexte, l'exploitation d'une architecture réifiée pour assurer la robustesse du système adapté requiert que cette architecture soit suffisamment riche, capturant non seule-

ment les composants et leurs dépendances mais également des aspects dynamiques associés (par exemple, les flots d'exécution ou les dissémination de références). Un exemple typique est celui des références d'objets qui s'échappent d'un composant pour aller dans un autre, sans qu'aucune dépendance ne réifie cet échappement. Bien que cette possibilité soit en principe interdite par la propriété d'encapsulation des composants, il n'en reste pas moins que la majorité des supports pour les modèles à composants ne vérifient pas cet aspect. En conséquence, toute migration d'un composant ayant laissé échappé de telles références devient alors source d'incohérences majeures dans l'application.

Un autre verrou qu'il nous faut aborder est que l'architecture réifiée doit être *causalement connectée* au système adapté, c'est-à-dire qu'elle doit capturer la structure *effective* de ce système. Cette propriété reste généralement partiellement satisfaite dans les environnements actuels, car elle implique un traçage trop coûteux à l'exécution.

Le défi est donc double : il faut enrichir l'architecture réifiée et garantir son exactitude. Ces deux besoins nous mènent naturellement à considérer l'architecture réifiée comme devant intégrer les couches *systèmes* de nos environnements d'exécution, au lieu de relever du niveau *intergiciel* où l'information et le contrôle sont limités pour répondre aux défis ciblés. En d'autres termes, nous considérons qu'une partie de la gestion des propriétés autonomes doit intégrer la conception de nos systèmes d'exploitation. A ce titre, j'ai rejoint un nouveau projet dont l'approche originale est adaptée à ces remarques.

Le projet revient sur la conception à base de machine virtuelle des couches les plus basses de nos systèmes d'exploitation, étudiant une nouvelle architecture logicielle à base d'objets et de composants. Le constat est que les abstractions des années soixante-dix, telles que les processus, les threads, les fichiers, ou encore la mémoire virtuelle, ne sont plus adaptées aux besoins des nouvelles applications. En effet, ces abstractions sont encore majoritairement conçues pour des applications indépendantes et non adaptables.

Le défi relevé par ce nouveau projet est la recherche de nouvelles abstractions qui soient adaptés au développement de logiciels coopératifs et adaptables. Le but est d'assainir nos architectures, évitant l'empilement de couches logicielles dont le manque de performance et la fragilité ne sont plus à démontrer. Le fondement est la redéfinition d'une Instruction Set Architecture (ISA) qui ne soit pas le reflet d'une architecture matérielle mais qui soit une fondation logicielle solide à base d'une machine virtuelle. Une façon simple de le présenter est la fusion entre une approche micronoyaux et une approche machine virtuelle telle que la machine virtuelle de Java ou C#, dans lesquelles serait intégré au plus bas niveau le concept de composant.

Dans cette approche, les notions de composants et d'architecture réifiée deviennent des briques de base de la machine virtuelle. Celle-ci fournit en retour un contrôle à grain fin permettant par exemple de tracer les invocations de méthodes, de réordonner celles-ci pour des besoins de reconfiguration ou encore de contrôler les fuites de références pour disposer de mécanismes d'isolation robustes. C'est à ce niveau que j'envisage d'intégrer les mécanismes de base permettant de supporter les propriétés d'adaptabilité autonomes, telles que l'auto-déploiement, l'auto-protection, et l'auto-réparation.

Bibliographie

[1] PicoContainer (2004). The PicoContainer Project, 2004.

[2] Spring (2006). The Spring Framework. 2006.

[3] Beugnard A., Jezequel J.-M., Plouzeau N., and Watkins D. Making components contract aware. *Computer*, 32(7) :38–45, Jul 1999.

[4] Goldberg A. and Robson D. *Smalltalk-80 : the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[5] Hayton R.and Herbert A. and Donaldson D. FlexiNet—a flexible component oriented middleware system. In *EW 8 : Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 17–24, New York, NY, USA, 1998.

[6] Oliva A. and Buzato L. The Design and Implementation of Guarana. In *Proceedings of COOTS'99 USENIX Conference on Object-Oriented Technologies*, may 1999.

[7] Wollrath A., Riggs R., and Waldo J. A Distributed Object Model for the Java System. *USENIX Computing Systems*, 9, 1996.

[8] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach : A New Kernel Foundation for UNIX Development. In *USENIX Conference Proceedings*, pages 93–112, 1986.

[9] Birrel A.D. and Nelson B.J. Implementing Remote Procedure Call. *ACM Transactions on Computer Systems*, 2(1), february 1984.

[10] OSGI Alliance. The OSGi Service Platform Release 4, 2005.

[11] Tanenbaum A.S. and Van Steen M. *Distributed Systems : Principles and Paradigms*. Prentice Hall, 2001.

[12] Claudel B., De Palma N., Lachaize R., and Hagimont D. Self-protection for distributed component-based applications. In *Symposium on Stabilization, Safety, and Security of Distributed Systems (formely Symposium on Self-stabilizing Systems), (SSS'06)*, Dallas, TX, USA, november 2001.

[13] Dumant B., Horn F., Tran F. D., and Stefani J.-B. Jonathan : an Open Distributed Processing Environment in Java. In Springer-Verlag, editor, *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, The Lake District, UK, 2007.

[14] Gowing B. and Cahill V. Meta-Object protocols for C++ : the Iguana Approach. In *Reflection'96*, San Francisco, United State, 1996.

[15] Janssen B. and Spreitzer M. ILU 2.0 alpha10 Reference Manual. In *Xerox Corporation Palo Alto CA*, february 1996.

[16] Smith B. Reflection and Semantics in a Procedural Language. *PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology*, 1982.

[17] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1) :39–59, 1984.

[18] Fabienne Boyer, Sébastien Chassande-Barrioz, Didier Donsez, David Féliot, and Sacha Krakowiak. GICOM : un atelier pour l'expérimentation des technologies de systèmes distribués d'entreprise. In *TICE 2002*, Lyon, November 2002.

[19] Marchetti C. CORBA Request Portable Interceptors : A Performance Analysis. In *DOA '01 : Proceedings of the Third International Symposium on Distributed Objects and Applications*, page 208, Washington, DC, USA, 2001. IEEE Computer Society.

[20] Szyperski C. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[21] CCM. The CORBA Component Model. Object Management Group Specification, http ://www.omg.org/technology/documents/formal/components.htm.

[22] Microsoft Corporation. .NET, http ://www.microsoft.com/net.

[23] Garlan D., Monroe R., and Wile D. Acme : An Architecture Interchange Language. In *GASCON'97*, pages 169–183, Toronto, Ontario, november 2007.

[24] Hagimont D. and Boyer F. A Configurable RMI Mechanism for Sharing Distributed Objects. *IEEE Internet Computing*, 5(1) :36–43, 2001.

[25] Luckham D.C. and Vera J. An Event-Based Architecture Definition Language. *IEEE Trans. Softw. Eng.*, 21(9) :717–734, 1995.

[26] G. Dufrene and Seinturier L. Un ADL pour les Architectures Distribuées à Composants Hétérogènes. In *Conference Francophone sur les Architectures Logicielles (CAL'08)*, 2008.

[27] Bergmans E. and Aksit M. *Constructing reusable components with multiple concerns using composition filters*. Software Architectures and Component Technology : The State of the Art, in Research and Practice, M. Aksit (ed.), Kluwer Academic Publishers, 2000.

[28] Bruneton E. Un support d'exécution pour l'adaptation des aspects non fonctionnels des applications réparties. In *PhD Thesis*, Institut National Polytechnique de Grenoble, 2001.

[29] Bruneton E., Coupaye T., Leclercq M., Quéma V., and Stefani J.B. The FRACTAL component model and its support in Java : Experiences with Auto-adaptive and Reconfigurable Systems. *Software Practice and Experience*, 36(11-12) :1257–1284, 2006.

[30] Kiczales G. et al. Aspect-Oriented Programming. In *11th European Conference on Object-Oriented Programming (ECOOP'97), LNCS 1241*, Jyaskyla, Finland, june 1997.

[31] Shaw M. et al. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4), 1995.

[32] Baude F., Caromel D., Henrio L., and Morel M. Collective Interfaces for Distributed Components. In *Seventh IEEE International Symposium on Cluster Computing and the Grid, CCGRID 2007*, pages 599–610, May 2007.

[33] Boyer F. A Causal Distributed Shared Memory based on External Pagers. In *USENIX Mach Symposium*, Monterey, december 1991.

[34] Boyer F. Coordinating Software Development Tools with Indra. In IEEE, editor, *International Conference on Software Engineering Environments (SEE'95)*, Noordwijkerhout (Netherlands), april 1995.

[35] Boyer F., Cayuela J., Chevalier P-Y., and Hagimont D Freyssinet A. Supporting an object-oriented distributed system : experience with Unix, Chorus and Mach. In *24th IEEE Symposium on Experience with Distributed and Multiprocessor Systems*, Atlanta, March 1991.

[36] Boyer F. and Charra O. Utilisation de la reflexivité dans les plate-formes adaptables pour applications réparties. In *Conférence sur les NOuvelles TEchnologies de la REpartition (NOTERE'2000)*, Paris, november 2000.

[37] Boyer F., Charra O., and Senart A. *Réflexivité pour des environnements adaptables.* Chapitre 3 du livre Les Intergiciels, I. Demeure and E.. Najm (éditeurs), Hermes Lavoisier Science, pp 73-91, april 2002.

[38] Boyer F., Gruber O., de Palma N., Sicard S., and Stefani J.B. *Self-Repair of Distributed Applications.* book chapter of Architecting Dependable Systems, Edited by R. de Lemos and J.C. Fabre, Springer, 2009.

[39] DeRemer F. and Kron H.H. Programming-in-the-large vs. Programming-in-the-small. *IEEE Transaction on Software Engineering*, 2(2), june 1976.

[40] Eliassen F., Goebel V., Kristensen T., Plagemann T., Andersen A., Rafaelsen Hans O., Yu W., Blair G., Costa F., Coulson G., Saikoski K. B., and Hansen O. Next Generation Middleware : Requirements, Architecture, and Prototypes. In *FTDCS '99 : Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 60–65, Washington, DC, USA, 1999. IEEE Computer Society.

[41] Kon F., Roman M., Liu P., Mao J., Yamane T., Magalha C., and Campbell Roy H. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *Middleware '00 : IFIP/ACM International Conference on Distributed systems platforms*, pages 121–143, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[42] Ogel F., Folliot B., and Piumarta I. On Reflexive and Dynamically Adaptable Environments for Distributed Computing. In *ICDCSW '03 : Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 112, Washington, DC, USA, 2003. IEEE Computer Society.

[43] Coulson G., Blair G. S., Clarke M., and Parlavantzas N. The Design of a Configurable and Reconfigurable Middleware Platform. *Distributed Computing*, 15(2) :109–126, 2002.

[44] Kiczales G., des Rivieres J., and Bobrow D.G. The Art of the Metaobject Protocol. In *MIT Press 345*, 1991.

[45] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., and Griswold W.G. An Overview of AspectJ. In *ECOOP '01 : Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[46] Paller G. Building Reflective Mobile Middleware Framework on Top of the OSGi Platform. In *Lecture Notes in Computer Science, Volume 4039/2006,*, 2006.

[47] Object Management Group. The Common Object Request Broker Architecture. 1995.

[48] Object Management Group. Trading object service v1.0. Specification. 2000.

[49] Object Management Group. The Common Object Request Broker : Architecture and Specifications. In *Revision 1.1/Omg Document No 91.12.1*, 2001.

[50] Object Management Group. Interceptors Published Draft with Corba 2.4+ Core Chapters. In *Document number ptc/2001-03-04*, 2001.

[51] Object Management Group. Corba 3.0, a Portable Interceptor Chapter.formal/020657. june 2002.

[52] Masuhara H., Matsuoka S., Watanabe T., and Yonezawa A. Object-oriented concurrent reflective languages can be implemented efficiently. In *OOPSLA '92 : ACM Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 127–144, New York, NY, USA, 1992.

[53] Georgiadis I., Magee J., and Kramer J. Self-organising Software Architectures for Distributed Systems. In *First Workshop on Self-Healing Systems*, Charleston, South Carolina, 2002.

[54] Welch I. and Stroud R. Dalang, A Reflective Java Extension. In *Reflection '99, LNCS 1616*, 1999.

[55] IBM. Autonomic Computing : the IBM Perspective on the State of Information Technology. *http ://www-1.ibm.com/industries/government/doc/content/resource/thought/278606109.html*, 2001.

[56] Aldrich J., Chambers C., and Notkin D. ArchJava : Connecting Software Architecture to Implementation. In *International Conference on Software Engineering (ICSE '02)*, may 2002.

[57] Chase J., Amador F., Lazowska E., Levy H., and Littlefield R. The Amber system : parallel programming on a network of multiprocessors. In *SOSP '89 : Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 147–158, New York, NY, USA, 1989.

[58] Dowling J. and Cahill V. The K-Component Architecture Meta-model for Self-Adaptive Software. In *Third international Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Lecture Notes In Computer Science, Springer-Verlag editor*, London, UK, september 2001.

[59] Dubois J., Retaillé J.P., and Templier T. Spring par la pratique. *Ed. Eyrolles*, 2006.

[60] Ferber J. Computational reflection in class based object-oriented languages. In *OOPSLA '89 : ACM Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 317–326, New York, NY, USA, 1989.

[61] Itoh J., Lea R., and Yokote Y. Using Meta-objects to support optimization in the Apertos operating system. In *USENIX Conference on Object Oriented Technologies (COOTS'95)*, june 1995.

[62] Magee J. and Sloman M. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6) :663–675, 1989.

[63] Magee J., Dulay N., and Kramer J. Structuring Parallel and Distributed Programs. *Software Engineering Journal*, 8 :73–82, 1993.

[64] Magee J., Dulay N., and Kramer J. Specifying distributed software architectures. In *5th European Engineering Conference (ESEC'95), volume 989 of Lecture Notes in Computer Science*, Berlin, Germany, 1995.

[65] MCAffer J. Meta-Level Architecture Support for distributed Objects. In *Proceedings of the 4th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS '95)*, page 232, Washington, DC, USA, 1995.

[66] Miller J. and Mukerji J. Model Driven Architecture (MDA). In *Object Management Group, OMG TC Document ormsc/2001-07-01*, july 2001.

[67] Poole J. Model-Driven Architecture : Visions, Standards and Emerging Technologies. In *Workshop on MetaModeling and Adaptive Object Models, European Conference on Object Oriented Programming (ECOOP'01)*, june 2001.

[68] J2EE. The Java 2, Enterprise Edition. *Sun Microsystems (http ://-java.sun.com/products/j2ee)*.

[69] Purtilo J.M. The POLYLITH Software Bus. *ACM Transactions on Programming Languages and Systems*, 16 :151–174, 1994.

[70] Baduel L., Baude F., Caromel D., Contes A., Huet F., Morel M., and Quilici R. *Grid Computing : Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[71] Bellissard L., Boyer F., and Riveill M. Construction and Management of Cooperative Distributed Applications. In *IEEE International Workshop on Object Orientation in OperatingSystems (IWOOOS'95)*, University of Lund, Sweden, august 1995.

[72] Bellissard L., Atallah S.B., Boyer F., and Riveill M. Distributed application configuration. In *Proceedings of the 16th International Conference on Distributed Computing Systems,*, pages 579–585, May 1996.

[73] Bellissard L., Atallah S.B., Boyer F., and Riveill M. Distributed Application Configuration. pages 579–585, May 1996.

[74] Seinturier L., Duchien L., and Florin G. A Meta-Object Protocol for Distributed OO Applications. In *Proceedings of TOOLS*, 1997.

[75] Seinturier L., Duchien L., and Florin G. CAOLAC : un protocole à méta-objets pour la synchronisation d'objets concurrents. *L'objet*, 4(3) :241–272, 1998.

[76] Seinturier L., Pessemier N., Duchien L., and Coupaye T. A Component Model Engineered with Components and Aspects. *Component-Based Software Engineering*, Lecture Notes on Computer Sciences, HAL - CCSD, Volume 4063/2006 :139–153, 2006.

[77] Dashofy E. M., Van der Hoek A., and Taylor R. A Highly-Extensible, XML-Based Architecture Description Language. In *WICSA '01 : Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, page 103, Washington, DC, USA, 2001.

[78] Dashofy E. M., Van der Hoek A., and Taylor R. An infrastructure for the rapid development of XML-based architecture description languages. In *ICSE '02 : ACM Proceedings of the 24th International Conference on Software Engineering*, pages 266–276, New York, NY, USA, 2002.

[79] Fowler M. Inversion of Control Containers and the Dependency Injection Pattern. http ://www.martinfowler.com/articles/injection.html., 2004.

[80] Golm M. Meta XA and the Future of Reflection. In *OOPSLA Workshop onReflective Programming in C++ and Java*, Toronto, Ontario, october 1998.

[81] Riveill M., Balter R., and Boyer F. Communication synchrone entre programme : principes et réalisations. *Traité Informatique, Techniques de l'Ingénieur*, 2000.

[82] Riveill M., Balter R., and Boyer F. Communication synchrone entre programme par appel de procédure (RPC) ou appel de méthode (RMI) - Principes et réalisation. *Techniques de l'ingénieur, Informatique*, march 2001.

[83] Rozier M., Abrossimov V., Arm F., Boule I., Gien M., Guillemont M., Herrmann F., Kaiser C., Langlois S., Leonard P., and Neuhauser W. Overview of the Chorus Distributed Operating System. *Computing Systems*, 1 :39–69, 1988.

[84] Microsoft. Microsoft DCOM, 1998.

[85] De Palma N. Services d'Administration d'Applications Réparties. In *PhD Thesis*, Université Joseph Fourier, Grenoble, october 2001.

[86] De Palma N., Bouchenak S., Boyer F., Hagimont D., Sicard S., and Taton C. Jade : un environnement d'administration autonome. In *Technique et Science Informatique (TSI)*, 2007.

[87] Medvidovic N. and Taylor R. N. A framework for classifying and comparing architecture description languages. In *ESEC '97/FSE-5 : Proceedings of the 6th European SOFTWARE ENGINEERING*, pages 60–76, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[88] Medvidovic N. and Taylor R. Exploiting Architectural Style to Develop a Family of Applications Abstract– Reuse of. In *Software Engineering IEEE Proceedings*, volume 144, pages 237–248, 1997.

[89] Pessemier N., Seinturier L., Duchien L., and Coupaye T. A Component-Based and Aspect-Oriented Model for Software Evolution. *International Journal of Computer Applications in Technology (IJCAT)*, 31(1-2) :94–105, 2008.

[90] Barais O., Lawall J., Le Meur A.F., and Duchien L. Software Architecture Evolution. *Software Evolution, Tom Mens and Serge Demeyer (eds), Springer Verlag*, pages 233–262, 2008.

[91] Brand P., Hoglund J., Popov K., de Palma N., Boyer F., N. Parlvanzas, Vlassov V., and Al-Shishtawy A. The Role of Overlay Services in a Self-Managing Framework for Dynamic Virtual Organizations. In *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, Heraklion - Crete, Greece, june 2007.

[92] Maes P. Concepts and experiments in computational reflection. In *ACM Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 147–155, New York, NY, USA, 1987. ACM.

[93] Merle P., Sevilla-Ruiz D., Bohme H., Leblanc S., Ritter T. Vadet M., and Scott Evans. Corba Component Model Tutorial. In *OMG, 2002, Document ccm/02-06-01*, june 2001.

[94] Raverdy P. and Rodger L. DART : A Distributed Adaptative Run-time. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, 1998.

[95] David P.C. and Ledoux T. Towards a Framework for Self-Adaptive Component-Based Applications. In Isabelle Demeure Jean-Bernard Stefani and editors Daniel Hagimont, editors, *the 4th IFIP WG6.1 International Conference on Distributed Applications and Interoperable Systems, (DAIS 2003)*, Paris, France, november 2003.

[96] Sun Microsystems Press. Java Messaging Service API Specification 1.0. 1999.

[97] Sun Microsystems Press. Jini Architecture Specification. 1999.

[98] Sun Microsystems Press. Java Transaction API Specification 1.0.1. 2001.

[99] Sun Microsystems Press. Java Persistence API Specification 1.0.1. 2003.

[100] James M. Purtilo. Module Interconnection Languages. *Journal of Systems and Software*, 6(4), 1986.

[101] James Marvin Purtilo. *A software interconnection technology to support specification of computational environments (polylith, module interconnection language).* PhD thesis, Champaign, IL, USA, 1986.

[102] Balter R., Bernadat J., Decouchant D., Duda A., Freyssinet A., Krakowiak S., Meysembourg M., Le Dot P., Nguyen H., Paire E., Riveill M., Roisin C., Rousset de Pina X., Scioville R., and Vandome G. Architecture and implementation of Guide, an object-oriented distributed system. *Computing Systems*, 4(1), 1989.

[103] Balter R., Bellissard L., Boyer F., Riveill M., and Vion-Dury J.Y. Architecturing and Configuring Distributed Applications with Olan. In *IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England, september 1998.

[104] Marvie R., Merle P., and Geib J.M. Separation of Concerns in Modeling Distributed Component-Based Architectures. In *Proceedings of the Sixth International Enterprise*

Distributed Object Computing Conference (EDOC'02), year = 2002, isbn = 0-7695-1656-4, pages = 144, publisher = IEEE Computer Society, address = Washington, DC, USA,.

[105] Marvie R., Merle P., Geib J.M., and Leblanc S. TORBA : trading contracts for CORBA. In *COOTS'01 : Proceedings of the 6th conference on USENIX Conference on Object-Oriented Technologies and Systems*, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[106] Pawlak R., Duchien L., Florin G., Martelli L., and Seinturier L. Distributed Separation of Concerns with Aspect Components. In *IEEE Proceedings of the Technology of Object-Oriented Languages and Systems TOOLS (33)*, pages 276–287, 2000.

[107] Pawlak R., Seinturier L., Duchien L., and Florin G. JAC : A Flexible Solution for Aspect-Oriented Programming in Java. In *Reflection 2001, Lecture notes in computer science, Springer Berlin*, volume 2192, pages 1–24, 2001.

[108] Rashid R., Baron R., Forin A., Golub D., Jones M., Julin D., Orr D., and Sanzi R. Mach : A Foundation for Open Systems. In *Proceedings of the Second Workshop on Workstation Operating Systems(WWOS2)*, september 1989.

[109] Reussner R., Poernomo I., and Schmidt H. Reasoning about Software Architectures with Contractually Specified Components,. *LNCS Springer Berlin / Heidelberg*, Volume 2693/2003, 2003.

[110] Weatherley R. and Gopal V. Design of the Portable .NET Interpreter DotGNU. In *Linux.conf.au*, Perth (Australia), january 2003.

[111] Allen R.J., Douence R., and D Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Conference on Fundamental Approaches to Software Engineering (FASE '98)*, march 1998.

[112] Bouchenak S., Boyer F., Hagimont D., Krakowiak S., de Palma N., Quema V., and J.-B. Stefani. Architecture-Based Autonomous Repair Management : Application to J2EE Clusters. In *Proceedings of the Second International Conference on Autonomic Computing, (ICAC'05)*, pages 369–370, June 2005.

[113] Bouchenak S., Boyer F., de Palma N., Gruber O., Sicard S., and Taton C. Jade : A Framework to Build Autonomic Systems. *accepted (under major revision) to ACM Transactions on Autonomous and Adaptative Systems*, october 2009.

[114] Bouchenak S., Boyer F., Depalma N., and Hagimont D. Can Aspects Be Injected ? Experience with Replication and Protection. In Springer-Verlag, editor, *International Symposium on Distributed Objects and Applications (DOA' 03)*, Catania, Italy, 2003.

[115] Bouchenak S., Boyer F., Depalma N., and Hagimont D. Component-based Autonomic Management System. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, Orlando, FL, USA, october 2005.

[116] Krakowiak S. Intergiciel et Construction d'Applications Réparties. In *Chapter 2, Patrons et canevas pour l'intergiciel, http ://sardes.inrialpes.fr/ krakowia/MW-Book/*, 2008.

[117] Leblanc S., Merle P., and Geib J.M. Torba : vers des composants de courtage. In *6ème Conférence Internationale sur les Langages et Modèles à Objets*, Montpellier, january 2002.

[118] Sicard S., Boyer F., and de Palma N. Using Components for Architecture-Based Management : the Self-Repair case. In *International Conference on Software Engineering (ICSE'08)*, may 2008.

[119] M. Saheb, R. Karoui, and S. Sédillot. Open Nested Transaction : A Support for Increasing Performance and Multi-tier Applications. In *Lecture Notes in Computer Sciences, Transactions and Database Dynamics, Springer Berlin*, volume 1773/2000, pages 167–192, 2000.

[120] Cheng S.W., Garlan D., Schmerl B., Sousa J.P., Spitznagel B., and Steenkiste P. Using Architectural Style as a Basis for Self-repair. In *3rd Working IEEE/IFIP Conference on Software Architecture*, Montreal, Canada, 2002.

[121] Abdellatif T., Boyer F., Kornas J., and Stefani J.B. Administration fondée sur l'architecture des serveurs d'applications J2EE patrimoniaux . In *5ème Conférence Française sur les Systèmes d'Exploitation (CFSE-5)*, Perpignan, France, october 2006.

[122] Batista T., Joolia A., and Coulson G. Managing Dynamic Reconfiguration in Component-based Systems. In *European Workshop on Software Architectures*, Pisa, Italy, June 2005.

[123] Ledoux T. Réflexion dans les systèmes répartis : application à Corba et Smalltalk. In *PhD thesis*, Université de Nantes, Ecole des Mines de Nantes, 1998.

[124] Ledoux T. and Rue Alfred Kastler. OpenCorba : a Reflective Open Broker. In *Reflection'99, Lecture Notes on Computer Sciences, Springer Verlag*, volume 1616, Saint-Malo, 1999. Springer-Verlag LNCS, Vol. 1616.

[125] Marangozova V. and Boyer F. Using reflectives features to support mobile users. In *Workshop on Reflection and Metalevel Architectures*, Nice - France, june 2000.

[126] Zhixue W. Reflective Java and a reflective component-based transaction architecture. In *Workshop on Reflective Programming in C++ and Java*, 1998.