Fédération : une architecture logicielle pour la construction d'applications dirigée par les modèles

Anh Tuyet Le

UNIVERSITE JOSEPH FOURIER – GRENOBLE I

THESE

pour obtenir le grade de

DOCTEUR de l'Université Joseph Fourier de Grenoble

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

Discipline : Informatique

présentée et soutenue publiquement par

LE Anh Tuyet

le 23 janvier 2004

Fédération : une Architecture Logicielle pour la Construction d'Applications Dirigée par les Modèles

Directeur de thèse :

Jacky ESTUBLIER

JURY :

F. Ouabdesselam, Professeur à l'Université Joseph Fourier	(Président)
L. Duchien, Professeur à l'Université de Lille	(Rapporteur)
C. Godart, Professeur à l'Université HENRI Poincaré - Nancy 1	(Rapporteur)
J-B. Stefani, Directeur de recherche à l'INRIA	(Examinateur)
F. Bernigaud, Industriel, Société Actoll	(Examinateur)
J. Estublier, Directeur de recherche au CNRS	(Directeur de thèse)

Thèse préparée au sein du Laboratoire LSR – Equipe ADELE

À ma mère

Tặng mẹ yêu dấu của con

Ce travail de thèse ne serait pas le même sans le soutien et l'aide de plusieurs personnes.

Je tiens à remercier les différents membres du jury qui m'ont fait l'honneur de participer à ma soutenance de thèse. Je souhaite mentionner F. Ouabdesselam, Professeur à l'Université Joseph Fourier de Grenoble, pour avoir présidé mon jury ainsi que mes deux rapporteurs L. Duchien, Professeur à l'Université de Lille, et C. Godart, Professeur à l'Université HENRI Poincaré de Nancy, pour avoir accepté de lire et d'évaluer mon travail de thèse. Je remercie aussi J-B. Stefani, Directeur de Recherche à l'INRIA pour sa participation au jury.

Je remercie la société Actoll qui a financé cette thèse, et plus particulièrement mon responsable F. Bernigaud.

Je tiens à exprimer mes plus sincères remerciements à J. Estublier, pour son rôle de directeur de thèse, pour ses conseils et pour sa disponibilité. Je lui suis très reconnaissante du point de vue professionnel.

Merci à Rémy, German, et Laurent qui ont passé un temps important pour lire le document et qui m'ont aidé à l'améliorer.

Merci à tous les membres de l'équipe Adèle du LSR.

Merci à mes amis Benoît, Céline, Lizbeth, Quynh, Hoang, Trinh, Vincent et quelques autres pour leur sympathie et l'aide qu'ils m'ont apportée au cours de ma thèse à Grenoble.

Un merci spécial à Jorge pour ses conseils et pour le partage très enrichissant de ses connaissances.

Une pensée chaleureuse à Sonia, avec qui j'ai partagé des moments de joie et de tristesse personnels et professionnels. Sans toi, ces quatre ans à Grenoble m'auraient paru plus long et plus difficile.

Et finalement, je voudrais adresser ma gratitude à ma famille, surtout à ma mère, pour son soutien. Merci à Cuong, mon copain, pour l'amour inconditionnel et la confiance qu'il m'a offert tout au long de ces années.

Résumé

La construction d'applications par composition d'éléments existants permet de réduire le temps de développement des applications ; pour cette raison, c'est un thème central du génie logiciel. On peut trouver quatre types de composition proposés par les approches actuelles : l'intégration, la connexion, la coordination et l'orchestration. Tandis que l'intégration et la connexion se basent sur les liens directs liant les éléments à composer, la coordination et l'orchestration utilisent une architecture à deux niveaux dans laquelle la logique de composition est exprimée séparément.

Cette thèse propose un modèle de composition, basé sur la coordination, où la réutilisation, l'évolution et l'adaptation sont les objectifs premiers. Le concept de domaine est proposé pour représenter un groupe d'éléments coordonnés fournissant une fonctionnalité clairement identifiée par un modèle conceptuel. La composition de domaines est effectuée en établissant des relations entre les concepts provenant des modèles conceptuels de ces domaines.

Une fédération est introduite comme une architecture logicielle permettant la structuration des domaines et la composition de domaines. La composition de domaines est réalisée par la synchronisation de domaines à travers un espace partagé et contrôlé (univers commun) contenant la matérialisation des modèles conceptuels des domaines composés ainsi que des relations liant leurs concepts.

Les environnements pour la conception, le développement, l'exécution et l'administration d'une fédération ont été réalisés, et plusieurs applications ont été construites afin de valider l'approche proposée dans cette thèse.

Mots clés : Composition, coordination, modèle de composants, programmation par aspects, guidage par les procédés, ingénierie dirigé par les modèles, domaine, modèle conceptuel, fédération.

Federation : a Software Architecture for Model - Driven Construction of Applications

Abstract

Building applications by composing existing elements allows a reduction in development time; for this reason, this is an important topic of software engineering. Current approaches propose four types of composition, which include integration, connection, coordination and orchestration. While integration and connection are based on direct links between the elements that are composed, coordination and orchestration are based on a two-level architecture where composition logic is represented and managed separately.

This thesis proposes a composition model based on coordination, where re-use, evolution and adaptation are the primary objectives. The concept of domain is proposed to represent a group of elements that are coordinated in order to provide a functionality that is clearly identified by a conceptual model. Composition of domains is carried out by establishing relationships between concepts that belong to the conceptual models of these domains.

A federation is introduced as a software architecture that allows domains to be structured and composed. Composition of domains is ensured by synchronizing domains through a controlled and shared space (common universe) that contains the conceptual models of composed domains as well as the relationships between their concepts.

The design, development, execution and management environment of a federation were developed, and several applications were built in order to validate the approach proposed in this thesis.

Keywords: Composition, coordination, component models, aspect oriented programming, process driven, model driven engineering, domain, conceptual model, federation.

Table des matières

Chapitre III - Composition d'éléments logiciels par coordination

Chapitre IV - Domaines et composition de domaines

Chapitre V - Construction des domaines génériques

Chapitre VI - Expérimentations et validation

Chapitre VII - Conclusions et perspectives

Chapitre I

Introduction

1. Le problème

Aujourd'hui, les applications sont de complexité croissante, leur réalisation nécessite des investissements toujours plus importants et pourtant le délai de mise sur le marché (*time to market*, en anglais) est un des facteurs déterminant de succès. Dans l'objectif de réduire le temps de développement des applications, la composition et la réutilisation représentent des thèmes de recherche importants pour les organisations.

On entend par *composition*, une manière de structurer et de construire des applications à partir d'éléments logiciels. Avec la composition, la profession d'informaticien évolue de plus en plus depuis le statut de « programmeur » d'applications vers le statut d'« intégrateur » d'éléments logiciels.

On entend par *réutilisation,* la construction des applications en réutilisant des éléments logiciels existants. La réutilisation nous permet de profiter les fonctionnalités qui sont déjà écrites, déboguées et maintenues par d'autres.

De cette manière, il est très intéressant d'avoir un modèle permettant de construire des applications à partir de la composition d'éléments logiciels en réutilisant le plus possible des éléments existants.

Il est fréquent que les besoins des clients évoluent avec le temps, ce qui implique de pouvoir modifier facilement les applications en conséquence. Désormais, l'évolution et l'adaptation des applications sont des critères importants pour distinguer différents modèles pour la composition [AFGK02, GKAF01].

Les approches actuelles proposent des solutions assez limitées en ce qui concerne la réutilisation des éléments logiciels existants ainsi que l'évolution et l'adaptation de l'application résultante. La programmation orientée objets ne propose qu'une composition entre classes : une classe hérite ou appelle des méthodes d'une autre classe. Naturellement, ces deux classes sont fortement liées. Le fait de changer l'implémentation de la classe appelée a de fortes chances d'impacter la classe appelante [LVE03, Mey00, Vil03].

Les composants ont été définis afin d'améliorer cette situation [Vil03, BBB00, Szy02, HC01]. Un composant demande les services, fournis par un autre composant, à travers une interface. Ceci permet au composant appelant (i.e. le composant client) d'ignorer

l'implémentation du composant appelé (i.e. le composant serveur). Cependant, le composant appelant doit être développé avec une connaissance précise de l'interface proposée et des nombreuses contraintes imposées par le composant appelé. Par conséquent, la probabilité de composer deux composants développés indépendamment, est très faible.

L'approche EAI (*Enterprise Application Integration*) [Lin00, LSH03, Lutz00] utilise le concept d'adaptateur permettant d'ajuster l'appelant et l'appelé afin de les connecter alors qu'ils n'étaient pas, originalement, conçus pour cela. Cependant, les connexions directes entre eux provoquent des difficultés pour faire évoluer et adapter l'application résultante [LR97, AFGK02].

Le BPM (*Business Process Management)* propose une architecture de composition à deux niveaux [LR97, BPM02, Mann99, LRS02] dans laquelle la logique de la composition est représentée explicitement par un procédé. Actuellement, le BPM est surtout utilisé dans le contexte des services Web. Des nombreux standards de BPM existent pour orchestrer les services Web tels que BPEL4WS (*Business Process Execution Language for Web Services*) [BPEL4WS], BPML (*Business Process Modelling Language)* [BPML], etc.

Ces approches ont leurs propres caractéristiques et spécificités. Néanmoins, elles ne peuvent être utilisées que dans des conditions spécifiques. Ceci est dû au fait qu'elles considèrent que les éléments à composer sont de même nature ou bien que les éléments ont été construits pour être utilisés dans une composition.

Une application, telle que livrée par un intégrateur, doit pouvoir être constituée d'éléments logiciels qui apportent des fonctionnalités souhaitées quelle que soit leur nature. Ces éléments peuvent être une application patrimoine (*legacy*), un composant sur étagère (*COTS - Commercial Off The Shelf*) [Vig98, BA99, Car97, VGD98], un composant d'un modèle quelconque, du code développé pour compléter des fonctionnalités manquantes, etc.

Nous nous sommes intéressés dans cette thèse aux problèmes posés dans ce contexte de composition et en particulier, aux solutions de composition dans lesquelles les éléments logiciels seront guidés pour travailler ensemble. Ceci permet d'éviter des connexions directes entre eux. Nous appelons *composition par coordination* une telle composition.

Nous pensons qu'il est important d'isoler des groupes d'éléments logiciels. Ceci permet de réutiliser non seulement un élément logiciel, mais aussi un groupe d'éléments. Les impacts, causés par l'évolution et l'adaptation d'un groupe, seront minimisés.

Nous pensons aussi qu'il est nécessaire de pouvoir réfléchir, travailler et valider une composition à un haut niveau d'abstraction (e.g. au niveau des modèles) en gardant, à tout moment, la vision globale abstraite du domaine du problème à résoudre. De ce point de vue, notre approche est conforme à l'approche MDA (*Model Driven Architecture*), proposé par l'OMG (*Object Management Group*) [BB02a, BB02b, Dsouza01, MDA01, MDA03, Bez01, Bez et el. 03].

2. Cadre du travail

Cette thèse a été réalisée dans le cadre d'une convention CIFRE dans laquelle se déroule le projet Centr'Actoll [Actoll]. Ce dernier est une collaboration entre plusieurs sociétés : ACTOLL, AREA [Area], INOVATEL, ASK [Ask], et le laboratoire LSR (Logiciels, Systèmes et Réseaux) de l'université de Grenoble [Lsr]. Cette collaboration a pour vocation de partager des connaissances théoriques et techniques :

- D'une part, le LSR désire développer des recherches dans un contexte industriel avec des contraintes réelles ;
- D'autre part, Actoll souhaite appliquer le résultat des recherches du LSR concernant la composition d'éléments logiciels existants.

Dans le cadre de cette collaboration, il a été convenu que trois applications seraient construites :

- Un système informatique dédié à la gestion des péages des réseaux de transport urbain et interurbain en utilisant des tickets *multi-modaux*. Avec un tel ticket, un client peut payer non seulement l'autoroute ou le parking, mais aussi d'autres types de transport (e.g. SNCF) et les services culturels tels que le cinéma, la cantine, la bibliothèque, etc. Pour ce faire, il est indispensable de faire inter-opérer les systèmes d'information des différentes sociétés proposant les services intégrés dans le ticket *multi-modal* ;
- Un environnement permettant aux membres d'un groupe de travailler collectivement sur un document. Cet environnement doit être construit en faisant inter-opérer les outils liés au processus de production coopérative d'un document tels que des éditeurs de document, des gestionnaires de versionnement des documents, des outils de transfert de document, des gestionnaires de tâches, des gestionnaires de ressources humaines, etc. ;
- Un système de déploiement automatique de logiciels sur des réseaux étendus et complexes. Ce système de déploiement est composé d'un outil de gestion de procédé servant à automatiser le processus de déploiement et un ensemble d'outils pour analyser et transférer l'application à déployer.

Notre laboratoire intervient dans ce projet en fournissant une approche de composition qui permet de faire inter-opérer des systèmes d'information provenant des différentes sociétés. Il est clair que chaque société a sa propre technologie pour construire ses systèmes. Pour les raisons de confidentialité et de fiabilité, nous ne pouvons pas accéder au code source des systèmes à composer ; ils sont ainsi, pour nous, des boîtes noires, hétérogènes et autonomes.

3. Objectifs de la thèse

Cette thèse a les quatre objectifs concrets suivants :

- Etudier en détails la composition utilisée par les approches actuelles. Concrètement, il faut identifier leurs manières de définir des unités de composition ainsi que leur opérateur de composition. Il est également important d'analyser leurs avantages et leurs inconvénients ;

- Etudier le problème de la composition, par coordination et dirigée par les modèles, d'éléments logiciels de différentes natures. A partir de cette étude, l'objectif est d'introduire une architecture logicielle ouverte, aisément adaptable à divers besoins et contextes, qui permet de construire les applications à partir de la composition d'éléments logiciels en tenant compte de :

    - Les éléments logiciels qui apportent les spécificités souhaitées à l'application à construire quelle que soit leur nature ;
    - L'évolution et l'adaptation de l'application résultante ;
    - La réutilisation de l'application résultante ou d'une partie de cette application.

  Nous appellerons *fédération* cette architecture logicielle.

- Construire une plate-forme complète pour supporter une fédération. Cette plate-forme doit contenir des outils permettant de définir des éléments d'une fédération, d'exécuter une fédération et d'administrer son exécution. Etant donné que nous sommes dans le cadre d'une collaboration industrielle, cette plate-forme doit supporter tous les aspects pratiques importants (e.g. la distribution, sécurité, hétérogénéité, etc.) et en plus, être flexible et performante ;

- Développer des applications réelles afin d'évaluer notre approche. D'une part, ceci nous aide à valider notre plate-forme développée, mais surtout, à étudier les aspects méthodologiques liés à la construction des applications par l'architecture de fédération. D'autre part, ceci nous permet d'accomplir notre participation au projet Centr'Actoll.

4. Contributions de la thèse

Les contributions principales de cette thèse peuvent être résumées comme suit :

- Nous étudions en détail les différentes manières de composer des éléments qui sont utilisées par les approches actuelles (cf. chapitre 2). En se basant sur ces observations, nous identifions deux axes de composition : (1) composition au niveau modèle, proposée par l'approche MDA et (2) composition au niveau code, utilisée par d'autres approches. Nous identifierons également quatre opérateurs de composition : (1) composition par intégration, (2) composition par connexion, (3) composition par coordination et (4) composition par orchestration. Les avantages, les inconvénients et le contexte d'utilisation de ces quatre opérateurs de composition seront aussi analysés ;

- En prenant la coordination comme une manière de composer, nous proposerons une fédération comme étant une architecture logicielle, ouverte et dynamique, permettant de composer les éléments logiciels de diverses natures en préservant leur autonomie et en assurant la cohérence des concepts partagés entre eux (cf. chapitre 3) ;

- En se basant sur l'idée de travailler au niveau modèle de MDA, nous proposerons d'utiliser un modèle conceptuel pour représenter un groupe d'éléments logiciels qui se coordonne. Un tel groupe d'éléments sera appelé *domaine*. La composition de domaines sera basée sur la composition de leur modèles conceptuels. Le concept de domaine et la composition de domaines permet de réutiliser un groupe d'éléments logiciels avec la sémantique de coordination entre éléments constituants. En plus, les impacts causés par l'évolution et l'adaptation d'un domaine seront minimisés (cf. chapitre 4) ;

- En s'inspirant la généricité proposée par les *frameworks* de type boîte blanche de la programmation orientée objet (*white-box framework)* [Mat00] et des *templates* de C++ [Lan03, DGD] et d'UML [UML01], nous proposerons une manière de réaliser des domaines génériques pouvant être utilisés dans plusieurs contextes d'utilisation. Avec un tel domaine générique, même un utilisateur final est capable de définir ses contextes d'utilisation pour lesquels ce domaine peut être utilisé.

- Nous construisons une plate-forme de support pour notre architecture de fédération. Cette plate-forme peut être utilisée pour supporter les applications à base de composants dynamiques ainsi que les applications orientées aspects.

5. Plan de la thèse

Outre l'introduction, ce rapport de thèse est divisé en trois grandes parties :

- Partie I : Etude des approches de composition existantes.

  Cette partie est composée d'un seul chapitre : le chapitre 2. Dans ce chapitre, nous clarifions les différentes approches de composition actuelles. Nous analysons, en particulier, les avantages, les inconvénients de ces approches afin de pouvoir bénéficier de leurs avantages dans notre approche de composition.

- Partie II : Etude de la composition, par la coordination et dirigée par la composition de modèles conceptuels exécutables.

  Cette partie est composée de trois chapitres. Le chapitre 3 présente l'architecture de la fédération. Le chapitre 4 étudie le concept de domaine et la composition de domaines. Le chapitre 5 étudie la manière de construire un domaine générique dans l'objectif de pouvoir l'utiliser dans des contextes similaires.

- Partie III : Présentation de nos résultats et de nos expérimentations.

  Cette partie est composée d'un seul chapitre : le chapitre 6. Ce chapitre présente les expérimentations que nous avons réalisées. Ces expérimentations sont constituées

d'une plate-forme de support et de l'ensemble des applications réelles que nous avons développées tout au long de cette thèse.

Enfin, le chapitre 7 propose une synthèse du travail effectué et présente des perspectives de travail futur.

Chapitre II

Etat de l'art

1. Introduction

La structuration et la construction des applications, par la composition d'élément logiciels de haut niveau d'abstraction, sont depuis longtemps le rêve des programmeurs informaticiens. Quels sont les mécanismes de composition qui existent aujourd'hui ?

Ce chapitre tente de répondre à cette question. Pour ce faire, il propose de parcourir les différentes approches actuelles pour la construction des applications, afin d'étudier en détail leurs mécanismes de composition. Pour chaque mécanisme de composition, les points suivants seront considérés :

- Quels sont les éléments de composition (i.e. les unités de composition) ? Comment sont-ils spécifiés ? ;
- Quel est le couplage entre les éléments de composition ? Ceci correspond à étudier la dépendance, l'interférence entre les éléments composés ;
- Quel est l'opérateur de composition ? Ceci correspond à la manière de lier les éléments de composition pour construire une application ;
- Est-il simple de faire évoluer et d'adapter l'application qui résulte de la composition ? Pour nous, l'évolution est liée aux changements des besoins fonctionnels de l'application (i.e. changement du problème à résoudre ou de la solution du problème à résoudre) ; l'adaptation est liée aux changements des besoins non-fonctionnels de l'application (e.g. changement de plate-forme d'exécution).

Cinq approches principales seront étudiées dans ce chapitre. Ces approches peuvent être divisées en deux grands axes.

Le premier axe est lié aux approches de programmation actuelles. Il s'agit de la programmation à base de composants, de la programmation par aspects – l'AOP (*Aspect Oriented Programming*) – et de l'ingénierie dirigée par des modèles – le MDA (*Model Driven Architecture*).

Le deuxième axe est lié à la construction des applications à partir d'éléments logiciels existants. Deux approches représentatives sont : l'approche EAI (*Enterprise Application Integration*) et l'approche BPM (*Business Process Management*).

Ce chapitre est organisé de la manière suivante. La section 2 présente la notion de composant et la composition de composants. La composition des services non-fonctionnels, proposée par les conteneurs, est également étudiée dans cette section. La section 3 se concentre sur l'AOP et la composition d'aspects. La section 4 étudie le MDA, proposé par l'OMG (*Object Management Group*), où la composition est faite au niveau des modèles. Les mécanismes consacrés à composer des éléments logiciels existants, proposés par l'approche d'EAI et de BPM, seront étudiés dans la section 5.

La section 6 fait une synthèse sur ces divers mécanismes de composition et retient des points positifs de chaque approche.

2. Programmation à base de composants

2.1. Motivation

Née au début des années 80, l'approche objet a démontré qu'elle facilitait la production des logiciels. D'une part, elle permet d'organiser le processus de production en plusieurs étapes, de l'analyse à l'implémentation, autour des entités, classes et objets, représentant des concepts du monde réel. A travers ces étapes, les mêmes concepts sont représentés par différents formalismes et à différents niveaux d'abstraction. D'autre part, diverses facilités tels que des patrons de conception (*pattern*), des canevas (*framework*), etc. sont proposés dans l'objectif de faciliter la programmation orientée objets (POO) [Mey96, Mey97, GHJ95, Mat00].

Cependant, la POO a des limitations importantes [Vil03, Mey97, Fin98] :

- Le mécanisme de composition est assez limité. Il s'agit d'une composition entre deux classes : une classe hérite ou invoque les méthodes d'une autre classe. Cette composition, faite à l'étape de codage, est exprimée à travers le langage de programmation. Désormais, les classes sont étroitement liées, ce qui empêche de séparer le processus de développement en deux étapes : la construction (codage) des éléments composables et l'assemblage de ces éléments ;
- La réutilisation à grande échelle s'est heurtée aux difficultés d'une réutilisation de type « boîte blanche » fondée sur l'héritage où les implémentations des entités réutilisées (i.e. des objets) sont visibles et modifiables ;
- Les caractéristiques non-fonctionnelles (NF) ne sont pas natives dans le modèle à objet original.

  Pour surmonter cette limitation, certains modèles à objet ont été étendus afin de supporter quelques caractéristiques NF, en particulier la distribution, tels que RMI (*Remote Method Invocation*) [Gro01, PM01], CORBA (*Common Object Request Broker Architecture*) [OH98], etc. Néanmoins, dans ce cas, il faut ajouter du code visant à gérer des propriétés NF dans les méthodes des classes. Par exemple, dans RMI, pour qu'une classe puisse être accédée depuis un client distant, il faut qu'elle hérite de la classe *UnicastRemoteObject* et déclare toutes les méthodes distantes dans

une interface héritant de l'interface *Remote*. Par conséquent, le code de cette classe est le résultat du mélange de la partie fonctionnelle et de la partie NF. Ceci empêche l'adaptation de l'application : le fait de changer de technologie (e.g. passer de RMI à CORBA) peut impliquer la réécriture de plusieurs méthodes.

Ces limitations ne sont pas négligeables dans le contexte de la programmation actuelle où, d'une part, les applications sont de plus en plus complexe et d'autre part, leur évolution et leur adaptation aux nouveaux besoins doivent être facilitées. Pour cela, il faut disposer d'un nouveau paradigme de programmation permettant de :

- Construire des applications par l'assemblage d'unités indépendantes ;
- Faciliter la gestion des caractéristiques non-fonctionnelles (NF).

Concernant le premier point, il est important de disposer de langages et de mécanismes de haut niveau d'abstraction permettant, d'abord, de définir les unités à composer, et ensuite de spécifier les relations entre elles, en termes de connexions, dans l'objectif de définir une application. La réutilisation des unités composables afin de construire des applications différentes est ainsi sollicitée.

Concernant les caractéristiques NF, elles sont d'une part difficiles à construire et d'autre part, indispensables pour les grandes applications. Une solution intéressante est de séparer le code lié aux caractéristiques NF du code métier. Ceci aide non seulement à faciliter la construction du codé lié aux caractéristiques NF, mais aussi à maximiser la réutilisation de ce code. Ce dernier sera géré par l'infrastructure supportant le code métier réalisé par les unités composables.

2.2. Composant comme élément d'assemblage

Les problèmes évoqués ci-dessus sont à la base d'un nouveau paradigme de programmation : la programmation à base de composants, dans laquelle les unités de composition réutilisable sont appelées des composants.

Actuellement, il n'y pas de consensus sur la définition d'un composant. Nous citons ici deux définitions représentatives :

- *"A software component is a unit of composition with contractually specified and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition"* [Clemens Szyperski - Szy02];

- *"A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard"* [HC01].

Au premier abord ces définitions ne sont pas les mêmes. Cependant, elles permettent de se mettre d'accord sur le fait qu'un composant est une unité de composition réutilisable et qu'un composant est bien défini grâce à un modèle. Ce dernier permet de spécifier l'interdépendance d'un composant par rapport à un contexte d'utilisation concret.

En effet, les composants représentent une évolution des objets en proposant des solutions aux limitations concernant la réutilisation et la composition. Du point de vue structurel, un composant propose une vision abstraite d'un groupe d'objets. Cette vision abstraite possède un certain degré d'opacité permettant de cacher la structure interne du composant et d'isoler le groupe d'objets, ce qui fonde le composant, en spécifiant les relations du composant, en terme de responsabilités et de besoins, par rapport au reste de l'application. L'ensemble des responsabilités et des besoins d'un composant représente son contrat de collaboration avec les autres. Ceci permet au composant d'évoluer plus aisément en respectant son contrat de collaboration.

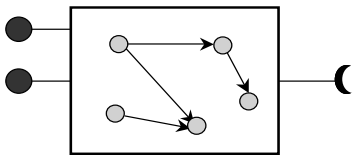

Figure 1. Un composant cache sa structure interne (d'après [Vil03])

Du point de vue méthodologique, un composant est construit pour être assemblé avec d'autres composants (i.e. pour être utilisé dans une composition). Par conséquent, il est conscient de sa collaboration avec ces composants dans l'objectif de construire une application. Cependant, un composant peut être utilisé dans n'importe quelle composition tant que son contrat de collaboration est assuré.

De cette manière, avec les composants, on peut distinguer clairement deux étapes dans le processus de production une application : (1) la réalisation des composants et (2) l'assemblage des composants en respectant leurs contrats de collaboration. Les décisions concernant la composition des composants sont repoussées à la phase d'assemblage de ceux-ci.

2.2.1. Modèle de composants

Pour supporter les composants et les applications à base de composants, une technologie de composants concrète doit fournir plusieurs éléments dont les plus importants sont : (1) un modèle de composant et (2) un ensemble d'outils – appelé le *framework* – permettant de supporter le modèle de composant, depuis la construction des composants, l'assemblage de composants pour construire une application jusqu'à l'exécution de cette application.

Le modèle de composants est la base d'une technologie de composants. Il consiste en un ensemble de conventions à respecter dans la construction et l'utilisation des composants. L'objectif de ces conventions est de permettre de définir et de gérer, d'une manière uniforme, les composants.

Les conventions couvrent toutes les phases dans le cycle de vie d'une application à base de composants : la conception, l'implémentation, l'assemblage, le déploiement et l'exécution. Concrètement, plusieurs aspects doivent être considérés : la manière de définir les composants à partir d'objets, les relations entre les composants, les propriétés

NF pour chaque composant, l'assemblage de composants, le déploiement et l'exécution d'une application à base de composants, etc.

Dans cette section, nous nous intéressons particulièrement à la définition et l'assemblage des composants. Ces deux aspects nous permettent de savoir comment les composants sont définis et composés. Ils correspondent respectivement à la vision externe et à la vision interconnexion d'un modèle de composant. D'autres visions d'un modèle de composant peuvent être trouvés dans [Vill03].

2.2.2. Vision externe d'un composant

Comme son nom l'indique, cette vision est destinée à spécifier la frontière entre un composant et le reste de l'application. Cette frontière est définie en terme de contrat de collaboration ; un contrat représente des responsabilités et des besoins du composant vis-à-vis des autres composants.

Pour ce faire, le concept de **port** a été ajouté dans l'objectif de spécifier une relation d'un composant avec les autres. Différents types de ports ont été définis afin d'exprimer différents types de relation.

Pour exprimer les responsabilités d'un composant par rapport aux autres, il y trois types de ports (cf. Figure 2) :

- **Interface fournie IF**. Il s'agit d'un port, caractérisé par une interface, contenant les services fournis par le composant ;

- **Facette F**. Il s'agit d'un port, caractérisé par un nom et une interface, contenant des services fournis par le composant. Ainsi, il est possible qu'un composant implémente plusieurs fois la même interface mais via des facettes différentes, ce qui est impossible de faire par des interfaces fournies. Notons que certains modèles de composant, tels que CCM [CCM02], utilisent des facettes. La plupart des autres, dont COM [COM] et OSGI [OSGI], proposent des interfaces fournies ;

- **Source S**. C'est un port, caractérisé par un type d'événement, désignant l'événement que le composant va émettre afin d'annoncer que quelque chose vient de se passer dans le composant. Les événements représentent un moyen de coordonner, de façon asynchrone, des composants.

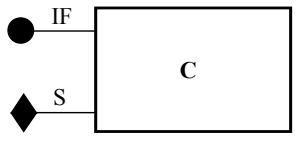

Figure 2. Représentation des responsabilités d'un composant

De la même manière, trois types de port ont été définis afin d'exprimer les besoins des composants les uns par rapport aux autres :

- **Interface requise IR**. Il s'agit d'un port, caractérisé par une interface, contenant des services dont le composant a besoin pour accomplir ses tâches ;
- **Réceptacle R**. Il s'agit d'un port, caractérisé par une interface et un nom, contenant des services dont le composant a besoin pour accomplir ses tâches. Similaire aux facettes, un composant peut demander la même interface mais sous différents noms et donc différents réceptacles. Ce concept est dual avec le concept de facette ;
- **Puits P**. C'est un port, caractérisé par un type d'événement, désignant un événement que le composant attend d'un autre composant afin de se synchroniser.

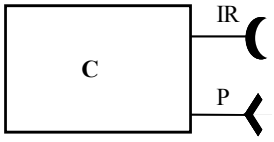

Figure 3. Représentation des besoins d'un composant.

Il est important de noter que la cardinalité peut être spécifiée aux ports de type *Réceptacle, Interface Requise* et *Source* afin de désigner le nombre de ports qui peuvent connecter avec ces ports.

Grâce à ces ports, la frontière, exprimé en terme des responsabilités et des besoins, d'un composant par rapport aux autres est clairement définie. Les composants sont ainsi prêts à être composés pour construire une application à base de composants.

2.2.3. Vision de composition (inter-connexion entre des composants)

Avant d'avancer dans cette section, il est nécessaire de remarquer que l'on ne peut composer que des composants qui suivent un même modèle de composants.

La vision de composition permet de spécifier la manière dont les composants sont connectés. La connexion entre deux composants se fait en connectant leurs ports. Il y a quelques règles pour connecter deux ports :

- Une interface fournie IF peut se connecter à une interface requise IR ou un réceptacle s'ils sont compatibles ;
- Une facette F peut se connecter à un réceptacle R ou une interface fournie IF s'ils sont compatibles ;
- Une source S ne peut être connectée qu'à un puits P s'ils sont compatibles ;
- Deux ports sont compatibles si les interfaces qui les caractérisent sont les mêmes ou bien l'une est sous-type de l'autre.

A côté des règles ci-dessus, il est nécessaire que les connexions respectent la cardinalité définie sur les ports. Si l'un réceptacle est déclaré comme multiple, il peut être connecté aux plusieurs facettes à la fois. De la même manière, si une source est multiple, elle peut envoyer un événement vers plusieurs puits.

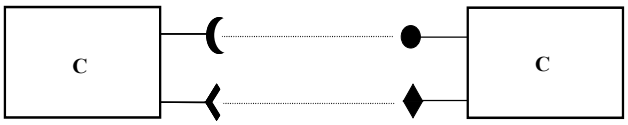

Figure 4. Composition de deux composants par connexion de leurs ports.

Suivant le modèle de composant, les connexions entre composants peuvent être spécifiées dans un langage dédié (i.e. un langage d'assemblage). Cependant, elles seront décrites de manière statique (i.e. avant l'exécution de l'application). La machine d'exécution utilise cette information pour connecter les instances de ces composants lors de l'exécution.

Il faut aussi noter que la connexion entre deux composants représente une dépendance fonctionnelle entre eux. Ceci veut dire qu'un composant déclare ses besoins fonctionnels sans prendre en compte qui les implémentera et comment ils seront connectés physiquement. Cela fait la différence avec la composition entre deux objets dans laquelle la dépendance vers une classe particulière est déclarée.

2.3. Les conteneurs et la gestion des services non-fonctionnels

2.3.1. Objectifs des conteneurs

Le concept de conteneur (*container*, en anglais) est proposée par le modèle de composant EJB [Rom99]. Pour savoir ce qu'est un conteneur, il est important de dire que, classiquement, dans le code d'un composant, on peut trouver deux parties [Rom99, Duc02]:

- Le code fonctionnel qui implémente les fonctionnalités fournies par le composant ;
- Le code non-fonctionnel (NF) qui implémente les caractéristiques NF du composant et qui permet d'adapter le composant à un environnement d'exécution spécifique.

Les conteneurs sont apparus pour faciliter la gestion du code NF. L'idée est d'exclure le code NF des composants et de le déléguer aux conteneurs. Autrement dit, les conteneurs sont chargés du code NF et de le proposer aux composants sous forme des services NF.

Les services NF dont un composant a besoin sont déclarés dans son *descripteur de déploiement*, grâce un langage déclaratif, lors de la phase de déploiement.

Le concept de conteneur apporte plusieurs avantages :

- La programmation des composants est « facilitée » par l'utilisation de conteneurs. Les développeurs de composants peuvent désormais se concentrer sur la partie fonctionnelle des composants ;
- Les services NF seront gérés par le conteneur de manière uniforme et quasiment transparente pour les composants ;
- Réutilisation des services NF par la réutilisation des conteneurs ;

- La séparation des domaines d'expertise : la programmation des composants, celle des conteneurs, l'assemblage, le déploiement, etc.

Il est important de noter que le conteneur n'est présent que dans certains modèles de composant tels que le modèle EJB [Rom99], CCM (*Corba Component Model*) [CCM02]. Puisque le modèle de conteneur de CCM est inspiré de celui d'EJB ; les services NF proposés et même leur fonctionnement sont proches de ceux des EJB. Dans la partie qui suit, nous étudions le principe des conteneurs.

2.3.2. Coordination de services non-fonctionnels

Pour la gestion des services NF, ce que les conteneurs font est de coordonner les fonctionnalités des composants avec les services NF.

Puisque les conteneurs sont basés sur l'hypothèse que les services NF sont orthogonaux (i.e. il n'existe pas d'interférence entre eux), la coordination est traduite tout simplement par l'ordonnancement des services NF avec les fonctionnalités des composants. Par conséquent, les services NF peuvent être exécutés avant et/ou après une fonctionnalité des composants (cf. Figure 5).

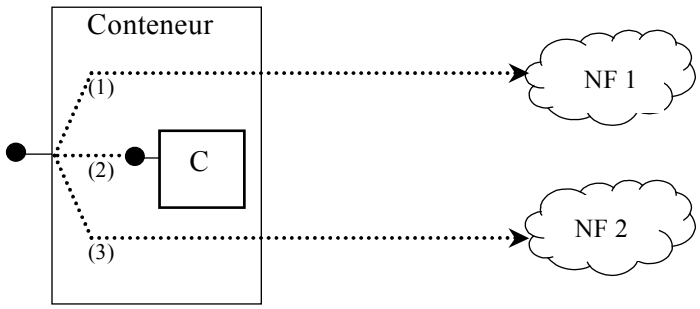

Figure 5. Le conteneur coordonne les services NF et fonctionnels

Cependant, pour certains services NF, la transaction par exemple, l'ordonnancement n'est pas suffisant. Dans ce cas, un composant doit collaborer avec son conteneur afin de gérer ces services. Par conséquent, il faut fournir les moyens pour que le conteneur et le composant puissent se coordonner et s'échanger des informations :

- Le conteneur doit définir des interfaces pour que le composant puisse informer de son état ou bien consulter le contexte d'exécution. Le conteneur CCM, par exemple, fournit ces interfaces à travers les interfaces internes (*Internal*) ;
- Le composant doit définir des interfaces de rappel (*callback)*, pour que le conteneur puisse contrôler la participation du composant dans la gestion des services NF. Les interfaces à implémenter dépendent des services NF qu'utilise le composant. Afin gérer les transactions, un composant CCM, par exemple, doit implémenter les méthodes *after_begin(), before_completion()* et *after_completion ()*.

La Figure 6 montre la relation entre un composant et son conteneur dans l'objectif de collaborer pour la gestion de certains services NF. Le conteneur gère les liens vers les

services NF à la place du composant. Néanmoins, il y a des interactions entre le conteneur et le composant pour accomplir la gestion de services NF.

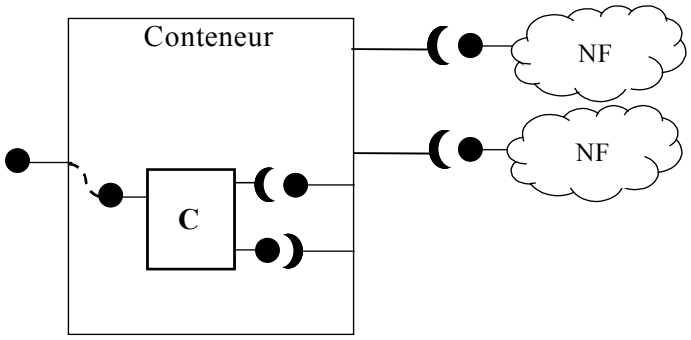

Figure 6. Relation entre le composant et son conteneur (d'après [Vil03])

2.3.2.1. Services proposés par un conteneur CCM

Plusieurs types de conteneurs sont utilisés pour accueillir différents types de composants [CCM02]. Il y a deux grandes catégories de conteneur CCM : (1) conteneurs orientés services qui accueillent des composants dont la fonctionnalité correspond à un service et (2) conteneurs de type *Entity* et *Process* qui accueillent des composants gérant des données persistantes.

Les services NF qu'un conteneur CCM propose dépendent de son type. Les services NF principaux sont :

- **Persistance**. Ce service est proposé par les conteneurs de type *Entity* et *Process* afin de gérer la persistance des données d'un composant. Pour ce service, on a deux choix de politique : (1) la persistance est gérée par le conteneur et (2) la persistance est gérée par le composant lui-même. Dans le premier cas, les méthodes *ccm_store()* et *ccm_load()* sont générées automatiquement. Dans le deuxième cas, il est à la charge du programmeur d'implémenter ces méthodes pour gérer lui-même la sauvegarde et la restauration de l'état du composant ;

- **Transaction**. Un conteneur CCM est capable de gérer des transactions pour les composants. Plusieurs options sont proposées pour différents niveaux ; le niveau de transaction souhaité doit être précisé à travers le descripteur de déploiement [CCM02] ;

- **Sécurité**. Ce service permet de gérer les accès sécurités aux fonctionnalités offertes par les composants. Ce service est applicable à toute les sortes de composant CCM ;

- **Distribution**. Ce service permet la distribution des composants CCM. Cependant, la gestion de ce service est différente de celles des autres ; elle n'est pas paramétrée à partir d'un descripteur mais elle est une conséquence directe du modèle CCM. Un composant CCM est avant tout un objet CORBA dont la distribution est gérée directement par la couche ORB (*Object Request Broker*) ;

- **Activation/ Passivation** : Le conteneur gère la présence en mémoire des diverses instances de composants, les désactive quand il le juge nécessaire et les réactive lors de requêtes des clients.

Il s'agit des services les plus couramment utilisés dans les grandes applications. Il est certain que la délégation de la gestion de ces services aux conteneurs simplifie la programmation et la réutilisation des composants.

2.3.3. Limitation des conteneurs actuels

La gestion des services NF proposée par les conteneurs est assez limitée car [Duc02, Bar et al. 01] :

- Il ne peut travailler qu'avec des services NF orthogonaux ;
- Le nombre de services NF fournis est limité et fixé. Il est impossible d'ajouter, ni de modifier ces services ;
- La manière de coordonner plusieurs services NF avec les fonctionnalités d'un composant est pré-déterminée et pré-calculée dans les conteneurs ;
- La gestion des services NF n'est pas tout à fait transparente aux composants. En réalité, on doit implémenter les interfaces de rappel (*callback*) afin de coordonner composant et conteneur lors de la gestion des services NF. Autrement dit, il faut prévoir, dès la phase de codage, quels sont les services NF dont ce composant aura besoin, et non pas seulement dans la phase de déploiement où l'on déclare des services NF souhaités dans le descripteur du déploiement.

2.4. Discussions

La programmation à composants propose un paradigme de composition très intéressant. Pour résumer ce paradigme, nous proposons de suivre quatre points que nous avons énoncés au début de ce chapitre :

- **Eléments de composition** : Ce sont des composants qui proviennent d'un même modèle de composant.

  Dans les modèles de composant à base de conteneurs, les composants ne contiennent que le code fonctionnel. Le code non-fonctionnel est délégué aux conteneurs ;

- **Manière de composer** : Il y a deux mécanismes de composition proposés: (1) la composition par connexion et (2) la composition par coordination.

  La composition par connexion est utilisée pour connecter les composants en liant leurs ports. Les ports connectés doivent être compatibles. Ceci implique le fait que le composant appelant (i.e. le composant client) dépend de la signature des méthodes fournies par le composant appelé (i.e. le composant serveur). Autrement dit, le composant client doit être développé avec une connaissance précise du composant

appelé. Par conséquent, la probabilité de composer deux composants, qui sont développés indépendamment, est très faible ;

La composition par coordination est utilisée par les conteneurs. Ce dernier gère des services NF orthogonaux en les coordonnant avec les fonctionnalités des composants. Pourtant, la coordination est assez simple ; elle est plutôt un ordonnancement entre les services NF et les fonctionnalités du composant ;

Néanmoins, comme le nombre des services NF est fixé, l'ordonnancement est déjà pré-calculé pour tous les « mariages » possibles entre ces services. Le fait de paramétrer un conteneur ne permet que de sélectionner un mariage pré-calculé des services NF proposés par ce conteneur ;

- **Couplage entre les éléments à composer** : Le composant appelant ne dépend que des interfaces du composant appelé et ignore complètement l'implémentation de ces interfaces. On est donc libre de sélectionner un composant quelconque qui implémente ces interfaces ; le couplage entre les composants est faible ;

- **Adaptation et évolution de l'application résultante** : A propos de l'adaptation, elle est plus simple à faire pour les modèles de composant à base de conteneur. Toutefois, les conteneurs sont nés afin de faciliter les changements non-fonctionnels des applications à base de composants (i.e. changement de l'environnement d'exécution). Cependant, à cause de la collaboration entre les composants et leur conteneur dans l'objectif de gérer certains services NF (i.e. les interfaces de rappel et les interfaces internes), il n'est pas si clair de faire adapter les composants dans des nouveaux environnements ;

  A propos de l'évolution d'une application, la capacité de supporter l'évolution d'une application dépend des modifications. Si une modification demande un changement au sein des composants, les modèles de composants la supportent bien car les composants sont déjà isolés ;

  En revanche, si une modification demande des changements en ce qui concerne les connexions entre les composants, il est très difficile de prédire les conséquences d'une telle modification ;

- **Réutilisation** : Un composant peut être réutilisé dans plusieurs compositions à condition que son contrat de collaboration soit assuré. En plus, avec l'aide des conteneurs, les composants ne contiennent que le code fonctionnel. La réutilisation est plus simple.

3. Programmation par aspects

"Separation of concerns is a key guiding principle of software engineering. It refers to the ability to identify encapsulate and manipulate only those parts of software that are relevant to a particular concept goal or purpose. Concerns are the primary criteria for decomposing software into smaller more manageable and comprehensible parts that have meaning to software engineer" [Elr et al.]

Depuis son apparition en 1995, la séparation des préoccupations s'affirme comme étant un nouveau paradigme de programmation. Parmi les travaux qui se concentrent sur ce paradigme, on peut citer la programmation au niveau Méta [KRB91, Kic92], les filtres de composition [Aks et al. 93, Aks et al. 94], la programmation par aspects l'AOP (*Aspect Oriented Programming*) [Kic et al.], etc. Cette dernière en est la plus représentative. C'est pour cette raison que l'on va étudier, dans cette section.

3.1. Motivation

La définition d'applications complexes dans différents domaines a rencontré un problème récurrent : l'existence de fonctionnalités qui sont disséminées dans différents endroits d'une application et qui ne bénéficient pas d'une encapsulation adéquate tant au niveau des modèles de conception que des langages de programmation. Une telle fonctionnalité est appelée une préoccupation. Elle peut être liée à des caractéristiques fonctionnelles mais surtout à des caractéristiques NF de l'application.

Cette faiblesse dans l'expression des préoccupations est manifeste dans la réalisation du serveur Apache, telle qu'elle est décrite par G.Kiczales dans son tutorial sur AspectJ. Certaines préoccupations telles que l'analyse syntaxique de termes XML ou la reconnaissance de motifs dans des URL sont bien encapsulées dans une seule classe. Par contre, le code implémentant la notion de session de connexion, de trace, d'exception et de synchronisation, etc. sont disséminées dans plusieurs classes [AJ, Kic et al.].

Le fait d'être disséminées à différents endroits provoque la répétition de code, ce qui rend complexe la lecture de code source et surtout la maintenance de l'application.

La programmation par aspects a été proposée comme une technique de programmation permettant la séparation des préoccupations dans une application [LH95]. Chaque préoccupation est conçue comme étant un aspect. L'idée de l'AOP est de pouvoir identifier, encapsuler et développer des aspects séparément et indépendamment. Ensuite, on les intègre aux « bons endroits » dans l'application pour qu'elle prenne en compte un nouveau but, un nouveau service, un nouveau besoin, etc.

3.2. Modèle de l'AOP

Le modèle de l'AOP fonde la définition d'une application sur un programme principal, appelé le programme de référence, et un ensemble d'aspects indépendants. Le programme de référence détermine la sémantique métier de l'application. Les aspects sont *tissés* à ce programme afin de l'étendre ou de l'adapter pour un usage particulier.

La Figure 7 montre le principe de base de l'AOP. Le programme de référence et les aspects sont, en théorie, développés séparément. Les aspects sont, ensuite, tissés au programme de référence à l'aide d'un outil spécifique – le tisseur d'aspects (*aspect weaver*). Ce dernier produit un nouveau programme qui contient les fonctionnalités de référence et celles fournies par les aspects.

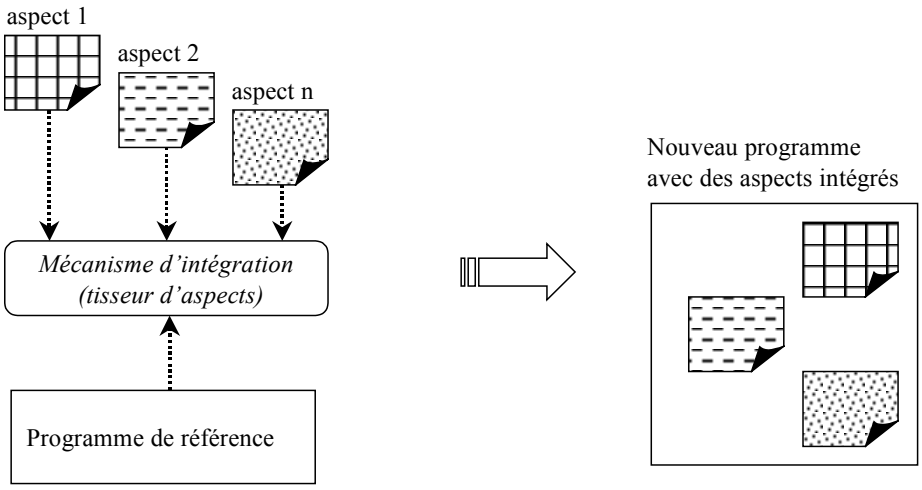

Figure 7. Modèle général de l'AOP.

Le programme de référence est défini dans un langage de programmation traditionnel (un langage à objets, par exemple). Il sert de référentiel pour introduire les points servant à ancrer les aspects. Il peut ainsi être considéré comme étant l'environnement ou bien le contexte d'exécution des aspects attachés.

Puisque les unités de composition sont des aspects, nous regardons ci-après en détails les mécanismes proposés par l'AOP pour :

- Définir les aspects ;
- Composer les aspects et le moment que l'on réalise effectivement la composition;
- Coupler les aspects et le programme de référence ;
- Faire évoluer et adapter le programme qui résulte de la composition des aspects au sein du programme de référence.

3.2.1. Définition des aspects

En schématisant, on peut dire que pour l'AOP, un aspect est caractérisé par deux éléments : (1) le code ajouté et (2) la localisation :

- Le **code apporté** constitue ce que l'aspect apporte au programme de référence. Ceci est appelé également l'action ou bien la méthode de l'aspect ;
- La **localisation** exprime l'endroit, dans le programme de référence, où l'aspect doit être attaché.

Un système AOP peut fournir un ou plusieurs langages généralistes ou dédiés qui sont destinés à définir des aspects. Le fait de définir un aspect consiste à spécifier ses deux éléments ci-dessus. En général, les deux éléments d'un aspect sont exprimés en terme d'éléments du programme de référence.

A titre d'exemple, on peut citer AspectJ qui est une implémentation représentative de l'AOP [AJ, Kic et al.]. AspectJ propose un langage à base de `Java` pour définir les

aspects. Deux éléments ci-dessus d'un aspect sont définis respectivement par un point de jonction (*Joint point*) et une méthode d'aspect (*advice*) et dans un même code.

Contrairement à AspectJ, JAC (*Java Aspect Components*) n'est pas un langage pour l'AOP, mais un *framework* proposé par [JAC, PSD03]. Ce *framework* offre la gestion dynamique et flexible des aspects pour une application. Avec JAC, la localisation et le code ajouté d'un aspect sont défini séparément. Ces deux éléments peuvent être définis sans référencer à un programme de référence comme dans l'AspectJ.

AspectJ et JAC prend les méthodes comme étant la granularité des points de jonction. Trois possibilités d'exécuter une méthode d'aspect à un point de jonction sont proposées : *before, after, around* pour désigner le fait que cette méthode d'aspect soit exécutée avant, après ou à la place de la méthode associée au point de jonction.

Puisque les aspects représentent la sémantique additionnelle au programme de référence, les aspects ne peuvent pas s'exécuter indépendamment du programme de référence. En plus, il est nécessaire d'avoir un mécanisme permettant de passer des paramètres entre le programme de référence et des aspects. AspectJ propose de s'appuyer sur les paramètres définis dans les points de jonction. Ce mécanisme n'est pas très intuitif, mais s'avère intéressant et relativement performant [Duc02].

3.2.2. Mécanisme de composition

A propos du mécanisme de composition de l'AOP, nous distinguons deux facettes principales :

- la composition des aspects au sein du programme de référence et
- la composition des aspects attachés à la même localisation dans le programme de référence.

Concernant la composition des aspects au sein du programme de référence, nous pouvons dire que l'AOP est une approche de bas niveau. Ceci, parce que l'AOP travaille directement au niveau du code source. Aucune vision de haut niveau d'abstraction n'est utilisée pour spécifier les aspects ainsi que le programme de référence. Nous considérons une telle composition comme *intégration*.

De cette manière, on ne compose pas un ensemble de boîtes noires (e.g. des composants) avec une frontière fonctionnelle bien définie, mais on tisse ou bien intègre des lignes de codes des boites blanches – des aspects – au sein d'un programme de référence. La structure interne du programme de référence est complètement visible aux aspects. Par conséquent, il est difficile de faire évoluer le programme de référence car un changement mineur de ce programme peut perturber la localisation des endroits où il faut attacher les aspects. Ceci peut être vu clairement dans AspectJ où les méthodes du programme de référence sont utilisées pour définir les endroits où attacher les aspects.

Concernant la composition d'aspects attachés au même point dans le programme de référence, nous pouvons distinguer deux manières :

- Par l'ordonnancement des aspects, on ne peut pas avoir un contrôle très fin sur la composition des aspects. Ceci doit se baser sur l'hypothèse qu'on n'a pas besoin de coordonner les aspects et que les aspects sont orthogonaux ;

- Par le tissage, la composition d'aspects peut être réalisée en entremêlant des instructions de ces aspects. Ceci permet un contrôle très fin sur la coordination des aspects. Il est donc possible de travailler avec des aspects non-orthogonaux.

AspectJ s'intéresse trop peu aux problème de composition des aspects. C'est pour cela qu'il prend l'ordonnancement comme étant le moyen de composer les aspects attachés à un même point de jonction. Pour ordonner les aspects, AspectJ propose quelques règles d'ordonnancement assez simples [AJ, Kic et al.]. Ceci n'est pas suffisant car il est fréquent que les aspects ne soient pas orthogonaux. Par exemple, un aspect de persistance peut être lié à un aspect de transaction [Duc02].

Des travaux qui s'intéressent à la composition des aspects non-orthogonaux peuvent être trouvés dans [KYX03].

3.2.3. Mécanisme d'implémentation.

Le mécanisme d'implémentation distingue le moment où l'intégration des aspects au sein du programme de référence, doit être accomplie. Deux politiques peuvent être utilisées :

- L'intégration **statique** consiste à utiliser un compilateur qui génère le nouveau programme en insérant les aspects au programme de référence. De cette manière, le nouveau programme contient les aspects intégrés qui sont exprimés en termes d'instructions additionnelles dans les méthodes du programme de référence.

  Cette politique offre une exécution performante mais peu flexible. Elle peut travailler en manipulant le code source et même en code binaire (*bytecode,* dans le cas de `Java`) du programme de référence. L'intégration statique est utilisée par AspectJ ;

- L'intégration **dynamique** consiste à utiliser un interpréteur pour ajouter dynamiquement, lors de l'exécution, les aspects au programme de référence. Pour ce faire, l'interpréteur doit instrumenter l'exécution du programme de référence afin de capturer les points où il faut ajouter les aspects et ensuite, exécuter ces aspects.

  Il est clair que l'intégration dynamique est plus flexible que l'intégration statique puisque les aspects peuvent être ajoutés/enlevés à tout moment. Cependant, le temps d'exécution est plus important.

  L'intégration dynamique peut être appliquée au code binaire du programme de référence. L'intégration dynamique est utilisée par JAC.

3.2.4. Découplage

Ce problème est lié à la relation et l'interférence entre un programme de référence et les aspects visant à étendre ce programme.

Il est clair que les développeurs qui écrivent les aspects doivent connaître le fonctionnement et la structure interne du programme de référence. Lorsque les aspects sont complètement orthogonaux, ils peuvent être définis séparément. Lorsqu'il y a interférence entre eux, il est nécessaire de considérer leurs relations.

Quant aux développeurs écrivant le programme de référence, doivent-ils être conscients de l'intégration des aspects dans certains points de son programme ? Quelles sont les préparations nécessaires dans ce programme afin de pouvoir accueillir les aspects ?

L'approche POA ne précise pas les décisions concernant les sujets ci-dessus. En effet, il s'agit de questions difficiles à répondre dans le cas général. Nous pensons qu'il n'est pas simple d'intégrer des aspects à un programme de référence quelconque pour un usage particulier. Ceci, parce qu'il est difficile d'intervenir dans des points où l'état du programme de référence n'est pas encore stable ou bien dans des points au milieu d'une méthode. Il est clair que la structuration des méthodes à étendre d'un programme de référence est important afin de pouvoir ajouter facilement des sémantiques additionnelles.

3.2.5. Réutilisation des aspects

Pour un aspect, ce qui est réellement intéressant de réutiliser le code apporté (i.e. la méthodes d'aspect, *advice*). La possibilité de réutiliser d'un aspect sera maximale lorsque le code apporté et la localisation sont définies séparément.

Avec AspectJ, le code apporté et la localisation d'un aspect sont définis dans le même code. En plus, la définition de la location fait référence toujours à la structure interne d'un programme de référence auquel attaché l'aspect. Ceci empêche de réutiliser un aspect avec des programmes de référence différents.

Pour surmonter cet inconvénient, AspectJ propose le concept d'aspect abstrait et d'héritage d'aspect. Un aspect abstrait contient la définition d'une localisation abstraite et du code apporté. Par l'héritage, cet aspect abstrait sera concrétisé pour être utilisé dans un programme de référence concret ; la définition de la localisation abstraite sera complétée en référençant à ce programme. Des détails concernant les aspects abstraits et l'héritage d'aspect peuvent être trouvés dans [AJ, Kic et al.].

Pour JAC, la réutilisation des aspects sera favorisée car il existe une séparation nette entre le code apporté et la localisation. Cette dernière est, en plus, configurable ; elle peut être donc adaptée à différents programmes de référence.

3.3. Discussions

Tout d'abord, en ce qui concerne la composition proposée par l'AOP, nous faisons un résumé en suivant les points que nous avons énoncés au début de ce chapitre :

- **Eléments de composition :** Ce sont les aspects et un programme de référence. L'idée principale de l'AOP est de tisser les aspects à un programme de référence afin de l'étendre ou de l'adapter pour un usage particulier ;

- **Manière de composer** : L'AOP utilise l'intégration de code comme étant la manière de tisser des aspects au sein du programme de référence. Ainsi, le programme de référence et les aspects sont des boîtes blanches lors de l'intégration ;

- **Couplage entre les éléments à composer** : Les aspects dépendent fortement, au niveau syntaxique et lexical, de la structure interne du programme de référence. De plus, ils ne peuvent être définis indépendamment que s'ils sont orthogonaux ;

- **Adaptation et évolution de l'application résultante** : Les aspects sont conçus afin de faire évoluer et adapter un programme (i.e. le programme de référence) pour un usage spécifique. Cependant, une fois que les aspects sont attachés, il est difficile de faire évoluer ce programme. Même un changement mineur, tel que le renommage d'une méthode ou la modification d'une classe sur laquelle sont attachés les aspects, peut demander de localiser des aspects et éventuellement de changer la façon d'utiliser et d'implémenter ces aspects ;

- **Réutilisation** : la réutilisation des aspects est assez limitée puisque la définition des aspects dépend de la structure interne du programme de référence.

Concernant l'approche elle-même, il reste beaucoup de problèmes à considérer :

- Le problème d'interférence entre des aspects n'est pas traité. On ne peut donc travailler qu'avec des aspects orthogonaux. Or, il y a souvent un risque d'interférence entre les aspects ;

- Les mécanismes destinés à tolérer et gérer les exceptions dans le programme final sont trop limités. Plusieurs emplacements doivent être vérifiés pour chercher une faute. Celle-ci peut être causée par le programme de référence, l'aspect inséré, par les interférences entre le programme de référence et l'aspect ou bien par des interférences entre des aspects [Elr et al.] ;

- L'assurance de la cohérence du programme résultat lors de l'ajout les aspects au sein du programme de référence. Comment peut-on assurer que l'apparition des aspects ne remet pas en cause le fonctionnement initial du programme de référence ?

- Il est difficile de définir des méthodologies pour travailler avec l'AOP car tout dépend de la structure du programme de référence et en conséquence, il n'y a aucune façon générale de raisonner et de valider;

Bien que l'AOP ne puisse pas être utilisé directement pour notre problématique, son idée de faire étendre un programme en utilisant des aspects est très intéressante ; des fonctionnalités peuvent être ajoutées, à un programme de référence, au fur et à mesure. Ce qu'il faut faire est de mettre l'AOP dans un cadre conceptuel afin de faciliter la validation et le contrôle du programme résultant. Dans ce cas, nous pouvons construire un noyau de fonctionnalité minimale, et d'autres fonctionnalités peuvent être ajoutées au fur et à mesure en fonction des besoins.

4. Ingénierie à base de modèles

Cette section étudie le MDA (*Model Driven Architecture*) qui est une nouvelle orientation de programmation proposée par l'OMG (*Object Management Group*) [BB02a, BB02b, Dsouza01, MDA01, MDA03, AK03, Bez01, Bez et al 03]. Comme le MDA s'oriente vers des modèles, nous allons étudier la construction d'applications par la transformation de modèles et la composition d'applications dirigée par la composition de modèles.

4.1. Motivation

En novembre 2000 à Orlando, l'OMG a proposé sa nouvelle orientation – le MDA – comme une réponse aux diverses difficultés du génie logiciel :

- Il y a trop de technologies et de plate-forme à l'heure actuelle, telles que EJB, CCM, .NET, etc. et le rythme d'arrivée des nouvelles plates-formes s'accélère. Ceci implique un empilement de technologies pour lesquelles il est nécessaire de conserver dans l'entreprise des compétences diverses et variées pour des raisons de maintenance des applications ;
- Les applications sont de plus en plus complexes. Le coût de migration d'une application d'une plate-forme à une nouvelle plate-forme devient rédhibitoire ;
- Les applications ont besoin d'interopérer. Or, on ne considère pas crédible qu'une seule plate-forme (e.g. Corba) puisse s'imposer. Chaque plate-forme a ses propres spécificités et qualités, et l'hétérogénéité n'est pas un phénomène temporaire. Il est donc nécessaire de trouver un moyen d'interopérer à un niveau plus abstrait.

La solution que le MDA propose pour surmonter ces problèmes consiste à utiliser des modèles dans toutes les étapes du processus de développement et d'interopérabilité des applications. Concrètement, le MDA suggère de :

- Séparer les éléments métiers qui ne sont liés qu'à la logique métier des applications, et ceux spécifiques à une plate-forme concrète. Ces éléments forment respectivement les modèles métiers et les modèles spécifiques à cette plate-forme ;
- Se concentrer sur l'élaboration de modèles métiers ;
- Favoriser des approches de transformation paramétriques des modèles métiers vers des plates-formes techniques concrètes (cf. Figure 8);

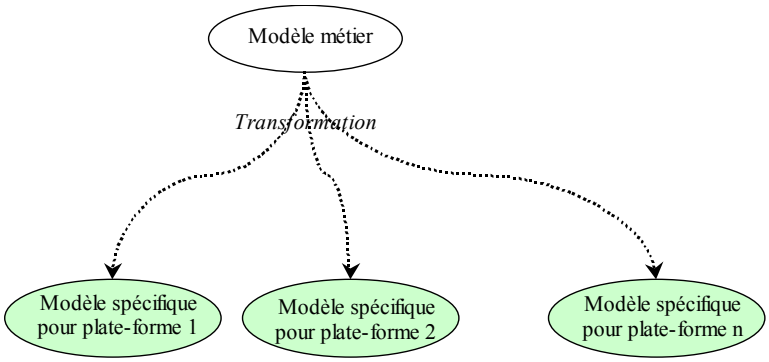

Figure 8. Transformation des modèles

Il est clair que le MDA se positionne à un niveau supérieur d'abstraction en se concentrant sur les modèles métiers. Avec ceci, un grand chantier s'ouvre avec plusieurs avantages :

- Un modèle métier fournit une spécification formelle de la structure et des fonctions de l'application en éliminant les détails techniques ;
- Il est plus facile de valider, de faire évoluer, de maintenir et de réutiliser des modèles métiers ;
- Des implémentations sur des plates-formes différentes peuvent être construites à partir d'un même modèle « métier », grâce à la transformation de modèles. Ceci facilite la portabilité des applications ;
- L'intégration et l'interopérabilité des applications peuvent être prévues à travers des modèles métiers et remettre à plus tard l'implémentation grâce à la transformation des modèles.

En schématisant, on pourrait dire que la solution de MDA inclut des changements sur plusieurs facettes du génie logiciel :

- Changement de paradigme : des objets aux modèles ;
- Changement de stratégie : de la composition d'objets à la transformation des modèles.

4.2. Définitions de base

4.2.1. Modèles

Une application est construite afin de résoudre un problème dans un contexte spécifique. Par conséquent, la description de cette application est fortement liée à la description du contexte et au problème qu'elle tente de résoudre.

Une application peut être décrite par plusieurs représentations. D'une part, ces représentations peuvent concerner différents points de vue dont chacun se concentre sur une préoccupation particulière de l'application. D'autre part, elles peuvent être liées à un même point de vue, mais à différents niveaux d'abstraction.

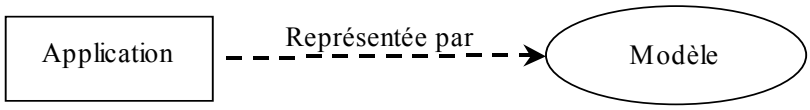

Figure 9. L'application est représentée par un modèle

Chaque représentation est appelée un modèle de l'application. Ce dernier permet de représenter l'application selon un point de vue et à un niveau d'abstraction donné, en ignorant ou en supprimant délibérément les détails qui ne correspondent pas à ce point de vue et/ou à ce niveau d'abstraction. Cependant, il doit être suffisamment explicite et précis pour permettre d'analyser et de vérifier la solution proposée par l'application, en l'absence

d'ambiguïté. Un modèle est plus simple à comprendre que l'application qu'il représente. Un modèle bien structuré et conçu peut rendre compréhensible une application complexe.

Un modèle *m* est exprimé dans un formalisme (ou bien un langage). Ce dernier est constitué d'un ensemble de signes, de symboles et de termes. Ceux-ci sont des représentations graphiques ou textuelles des primitives de modélisation dont la sémantique est clairement définie par le méta-modèle sur lequel se base ce formalisme. Si nous connaissons le méta-modèle, nous sommes capables de comprendre, de raisonner et d'interpréter le modèle *m* conformant à ce méta-modèle.

Par exemple, le méta-modèle UML propose des primitives de modélisation tels que *Class, Association*, *Contraint*, etc. Ces primitives sont transformées en boîte, flèche dans le formalisme graphique UML et en contraintes OCL. Le formalisme UML est utilisé pour représenter un modèle UML d'une application quelconque.

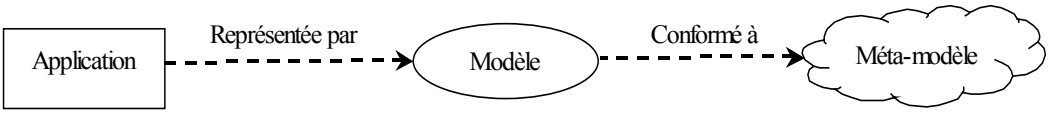

Figure 10. Méta-modèle et modèle

4.2.2. Types de modèles

Le MDA définit deux types de modèles :

- Les PIM (*Platform Independent Model*) qui sont indépendants de toute plate-forme ;
- Les PSM (*Platform Specific Model*) qui sont dépendants d'une plate-forme spécifique.

Pour faire cette distinction, le MDA est basé sur la notion de plate-forme. Nous étudions d'abord ce qu'est une plate-forme du point de vue de MDA.

Plate-forme

Une plate-forme est un ensemble de sous-systèmes/technologies qui fournissent un ensemble cohérent de fonctionnalités à travers des interfaces et des patrons d'utilisation spécifiques. Une application quelconque qui dépend de cette plate-forme peut utiliser ces fonctionnalités sans connaître les détails concernant la façon dont ces fonctionnalités sont fournies par la plate-forme [MDA03].

Il y a plusieurs types de plates-formes : (1) générique tels que la plate-forme de lot (*batch*, en anglais) qui supporte une série de programmes indépendants dont chacun finit son exécution avant que le prochain commence, (2) spécifique lié à une technologie particulière tels que EJB, CCM, J2EE et (3) spécifique lié à un vendeur particulier tels que J2EE de IBM *WebSphere*, etc.

Une plate-forme a son modèle. Ce dernier fournit un ensemble de concepts qui représentent les éléments faisant partie de la plate-forme et des services fournis par la

plate-forme. Il spécifie, en plus, les conditions concernant l'utilisation des parties de la plate-forme et la connexion d'une application à la plate-forme [MDA03].

Par exemple, *EntityComponent*, *SessionComponent*, *Facet, Receptacle*, etc. sont les concepts principaux pour le modèle de plate-forme CCM. Ces concepts permettent de spécifier l'usage de la plate-forme pour une application basée sur CCM. L'OMG a proposé également un profil UML CCM comme étant le langage pour définir une telle application.

PIM – Platform Independent Model

Un modèle PIM d'une application est un modèle qui n'a pas de dépendance vers des plates-formes concrètes. Il ne contient que des entités fonctionnelles avec leurs interactions, exprimées uniquement en termes de la logique métier de l'application [BB02a, MDA01, MDA03].

PSM – Platform Specific Model

Un modèle PSM d'une application est un modèle qui a des dépendances vers une plate-forme concrète. Il contient les informations du PIM correspondant et également des détails spécifiant comment l'application utilise cette plate-forme. Il sert essentiellement de base à la génération de code exécutable sur cette plate-forme [BB02a, MDA03].

Un modèle PSM est exprimé en terme du modèle de plate-forme correspondante. Par exemple, un modèle PSM lié à CCM sera exprimé avec l'aide du profil UML CCM.

4.2.3. Transformation de modèles

Pour le MDA, les modèles sont le référentiel dans le processus de construction des applications. En conséquence, ils doivent être manipulables. Plusieurs opérations destinées à manipuler les modèles ont été identifiées dans [BB02b]. Parmi ces opérations, nous nous intéressons particulièrement à l'opération de transformation de modèles. Cette opération est considérée comme la base du MDA.

L'objectif de la transformation de modèles est de produire un modèle à partir d'un autre modèle. Prenons la notation *m(A)/Mm* pour désigner le modèle *m* d'une application *A* exprimé par le méta-modèle *Mm*. Une transformation T permettant la projection depuis *m1(A)/Mm1* à *m2(A)/Mm2* est exprimée comme suivant :

$T(m1, m2) : m1(A)/Mm1 \rightarrow m2(A)/Mm2$

Il est important de caractériser les différents types d'opérations de transformation. La connaissance des méta-modèles sources et cible peut être utilisée dans cet objectif. Dans le cas où les deux méta-modèles sont identiques, c'est à dire que $Mm1 = Mm2$, on a une transformation directe. Sinon, on a une transformation indirecte.

Les transformations directes, tels que le raffinement ou l'optimisation, sont faites, en général, de façon manuelle. Par contre, des transformations indirectes où $Mm1 \neq Mm2$,

dont la transformation PIM à PSM et PSM à PIM sont des exemples, peuvent parfois être réalisées automatiquement en appliquant des règles de transformation.

Il y a plusieurs manières d'appliquer une règle de transformation : (1) sur un modèle particulier ou (2) sur tous les modèles issus d'un méta-modèle particulier.

- Dans le premier cas, une règle de transformation spécifie la manière de transformer un (un groupe) élément d'un modèle source en un autre (des autres) éléments dans le modèle cible. Un modèle (*template,* en anglais) peut être utilisé pour exprimer une telle règle [MDA03] ;

- Dans le deuxième cas, une règle de transformation est basée sur les méta-modèles source et cible en spécifiant comment un concept de méta-modèle source sera transformé en un autre du méta-modèle cible. La transformation sera désormais prédéterminée pour tous les modèles exprimés par le méta-modèle source.

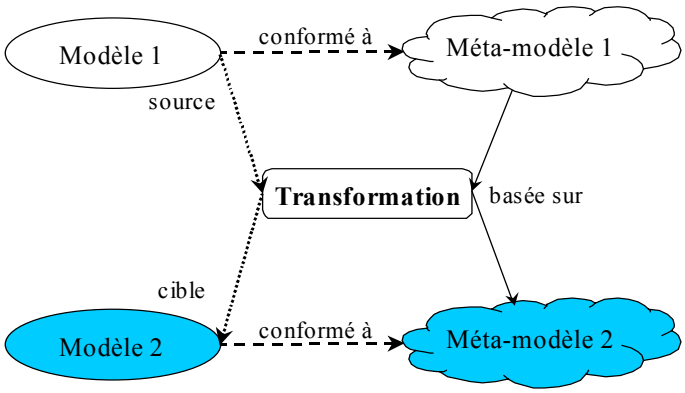

Figure 11. Transformation indirecte, basée sur les méta-modèles

La transformation de modèles est actuellement un thème de recherche actif. Dans la suite de cette section, nous étudions quelques approches de transformation automatique de modèles.

4.2.3.1. XSLT (Extensible Stylesheet Language Transformation)

XSLT est un langage de transformation proposé par le W3C en 1999 [XSLT]. Ce langage permet de définir des règles de transformation destinées à transformer un document XML bien formé (i.e. il est valide pour une DTD – *Data Type Definition* – particulière) en un autre.

Une feuille de style XSLT regroupe un ensemble de règles de transformation dont chacune définit des traitements opérationnels à effectuer sur un nœud de l'arbre formé par le document source. Une telle règle est composée de balises dont la première est `<xsl:template>` et la dernière est `</xsl:template>`. La première est complétée par un attribut `'match'` dont la valeur permet de définir le ou les nœuds du document XML source sur lesquels s'applique la transformation. La localisation des nœuds et des informations contenues dans ces nœuds est faite avec l'aide de XPATH [XPATH].

```
<xsl:template match="Entity1">
    <xsl:if test="att operator value">
        <xsl:element name="Entity2">
            <!--attributes mapping-->
            <xsl:apply-templates
            select="./child::node()" mode="Entity1"/>
        </xsl:element>
    </xsl:if>
</xsl:template>
```

Figure 12. Un exemple d'une règle avec XSLT

Dans l'exemple de la Figure 12, la règle sera appliquée sur les nœuds nommé « Entity1 » avec la condition 'att operator value' et a pour objectif de transformer ces nœud en « Entity2 ».

XSLT peut être utilisé pour définir des règles de transformation qui sont appliquées au niveau modèle. Cependant, comme [PBG01] l'a énoncé, XSLT a plusieurs inconvénients qui sont, entre autres :

- Les règles de transformation de XSLT sont de bas niveau. Elles sont définies en se basant sur la structure des documents (i.e. des modèles) source et cible mais non pas sur les méta-modèles source et cible ;
- L'écriture d'une feuille de style XSLT est longue et pénible. Elle demande d'avoir une bonne connaissance de la structure des documents source et cible ;
- Il est difficile de lire et d'interpréter une feuille XSLT.

4.2.3.2. MTRANS

MTRANS est un langage proposé par France Télécom R&D [PBG01, BP02]. Il s'agit d'un langage déclaratif permettent de définir des règles de transformation de haut niveau qui s'appuient sur les méta-modèles source et cible, à condition qu'ils soient conformes au MOF (*Meta Object Facility*) [MOF00].

La grammaire des règles de MTRANS peut être trouvée dans [PBG01]. Leurs règles représentent un mélange de l'approche déclarative et procédurale. Une règle exprime explicitement d'une part, un élément cible et un élément source et d'autre part, la manière de transformer l'élément source en l'élément cible.

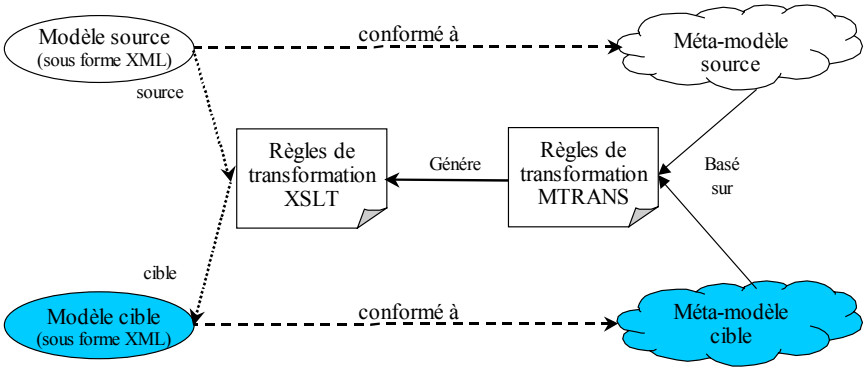

Figure 13. Transformation par MTRANS

Des règles de transformation XSLT, appliquées aux modèles, sont générées à partir d'une règle de haut niveau de MTRANS. La règle XSLT de la Figure 12 est exprimée en MTRANS de la façon suivante :

```
Entity1 [att operator value] Entity2
```

4.2.3.3. Transformations par graphe

Varro et al [VVP02, Var02] décrivent un système destiné à la transformation de modèles par graphe. Dans leur approche, une transformation se compose d'un ensemble de règles combinées avec des opérateurs, tels que *forall*, *try*, *loop,* etc., déterminant la manière et l'ordre d'exécution des règles. Une règle de transformation est exprimée sous la forme `Rule = (Bg, Cd, Ag)` dans laquelle `Bg` est le graphe original (i.e. avant la transformation), `Cd` est optionnel et représente la condition d'application de la règle, `Ag` est le graphe résultant de la transformation.

Cette approche est non-déterministe dans le choix de la règle provenant d'une transformation, et dans le choix du sous-graphe sur lequel doit être appliquée une règle. En outre, une transformation appliquée peut impliquer une série d'application des règles. Il est donc nécessaire de faire attention à la répétition des règles et à l'ordre d'application des règles afin d'assurer la terminaison de la transformation.

4.2.3.4. Transformations relationnelles

La transformation relationnelle [HK03, AK02a] est utilisée plutôt pour exprimer la projection (*mapping*) entre deux méta-modèles et par conséquent, entre deux modèles provenant de ces deux méta-modèles. Pour cela, elle définit, tout d'abord, les relations entre deux concepts provenant du méta-modèle source et cible. Ensuite, pour chaque relation, plusieurs paires sont établies entre des éléments qui sont des instances des concepts concernés par cette relation et qui viennent des modèles source et cible. Des contraintes de type OCL peuvent être utilisées pour exprimer les conditions nécessaires lors de l'établissement les paires d'instances.

Figure 14. Relation et paire

De cette manière, la transformation relationnelle représente une transformation bidirectionnelle tandis que les autres transformations sont unidirectionnelles. Elle permet de tracer la transformation de modèles, entre un modèle source et un modèle cible ; L'ingénierie inverse est donc possible.

4.2.3.5. Discussions

Les approches de transformation de modèles, que nous venons de présenter, permettent de construire un modèle à partir d'un autre modèle. Néanmoins, elles n'abordent pas la

validation d'un modèle transformé. Nous distinguons deux niveaux de validation d'une transformation de modèles :

- La transformation doit être correcte du point de vue syntaxique. Ceci signifie que le modèle résultat doit être bien formé (*well-formed,* en anglais) dans le langage cible (i.e. le méta-modèle cible) ;
- La transformation doit assurer quelques critères sémantiques : (1) la transformation doit terminer, (2) la transformation doit donner un résultat unique, (3) la transformation doit préserver l'équivalence sémantique entre les modèles source et cible.

Pour l'instant, les approches actuelles pour la transformation de modèles ne permettent qu'une validation syntaxique. En ce qui concerne la vérification au niveau sémantique d'une transformation de modèles, quelques initiatives ont été commencées dans [Var02].

4.3. Construction des applications par transformation de modèles

L'approche MDA se base sur le constat que les modèles existent partout, depuis l'analyse jusqu'à l'exécution, dans le processus de construction des applications. Des exemples sont le modèle d'analyse, de conception, de test, de qualité de service, de plate-forme, de déploiement, d'exécution, etc. Ces modèles peuvent être classés en deux groupes :

- **Groupe 1** contient des modèles qui représentent l'application à différentes étapes de construction tels que le modèle d'analyse, le modèle de conception, le code exécutable, etc. ;
- **Groupe 2** contient des modèles servant à vérifier ou contrôler l'application dans les différentes étapes de son cycle de vie. Le modèle de test, par exemple, propose des cas de test qui sont destinés à vérifier la correction de l'application. Le modèle de déploiement, à son tour, spécifie comment il faut déployer l'application. Chacun de ces modèles se concentre sur une préoccupation spécifique liée au processus de construction et d'exécution des applications.

L'approche MDA tente de travailler ces deux groupes de modèles. Concernant les modèles du premier groupe, le MDA se base sur le constat que ces modèles représentent l'application à différentes étapes à différents niveaux d'abstraction et/ou à différents points de vue. Parmi ces modèles, il y a des modèles de type PIM et d'autres qui sont de type PSM. Ce qui est intéressant est qu'un modèle peut être construit à partir d'un autre, grâce à la transformation de modèles.

En se basant sur ces observations, le MDA propose de considérer le processus de construction d'une application comme une chaîne de transformation de modèles à partir du modèle métier qui est le modèle de plus haut niveau d'abstraction. Cette chaîne, illustrée dans la Figure 15, contient des transformations directes qui s'appliquent sur un modèle PIM ou sur un modèle PSM, tels que :

- L'optimisation qui sert à améliorer un modèle selon des règles de conception dans l'objectif d'en augmenter la lisibilité, l'efficacité ou bien la réutilisabilité ;

- Le raffinement qui consiste à ajouter d'autres informations dans un modèle pour qu'il devienne plus complet et plus détaillé. Par exemple, un objet « abstrait » peut être raffiné en un ensemble d'autres objets de granularité plus fines.

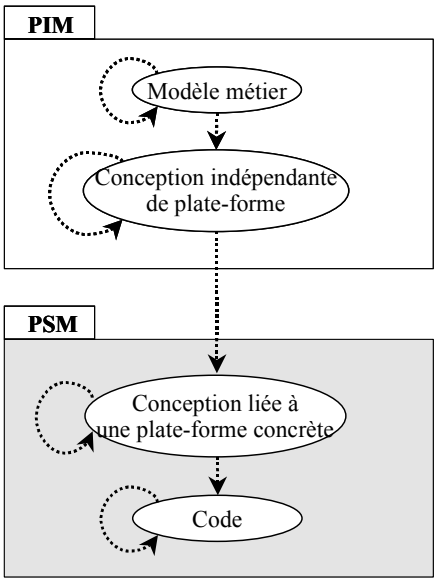

Figure 15. Chaîne de transformation de modèles pour construire une application

Des transformations indirectes peuvent également être trouvées dans cette chaîne de transformation. Il s'agit de la transformation de PIM vers PSM ou de PSM vers code. La transformation de PIM vers PSM est effectuée lorsque le modèle PIM est satisfaisant et que l'on veut utiliser ce modèle dans une plate-forme particulière. Cette transformation consiste à ajouter des informations liées à la plate-forme afin de préparer une transformation PSM vers du code exécutable sur cette plate-forme. Ces deux transformations indirectes, de PIM vers PSM et de PSM vers code, ont pour but d'être fortement automatisées.

Les modèles du deuxième groupe, liés au contrôle et à la cohérence de l'application, sont actuellement développés, documentés et maintenus en dehors du code. Ceci implique un risque d'incohérence entre ces modèles et l'application à construire. C'est pourquoi le MDA souhaite les générer à partir des modèles du premier groupe. De cette façon, la cohérence sera mieux assurée.

L'idée proposée par le MDA est simple et intuitive. Cependant, pour que cela devienne réaliste, il est indispensable de créer des moyens permettant de :

- Représenter des modèles PIMs avec suffisamment de détails pour être immergés dans une plate-forme particulière ;
- Représenter des modèles PSMs ;
- Transformer automatiquement des modèles PIMs vers PSMs et des PSMs vers le code.

Il est important de savoir que le succès du MDA dépend fortement de ces moyens. Pour l'instant, la modélisation et l'échange de modèle s'effectuent sur des standards reconnus

de l'OMG bien qu'ils soient en cours d'évolution afin de couvrir divers domaines d'application et de contenir suffisamment d'informations. Des exemples de ces standards sont le MOF (*Meta Object Facility*) [MOF00, Lem00], le XMI (*XML Metadata Interchange*) [XMI00], l'UML (*Unified Modeling Language*) [UML01], etc. Quant aux techniques de transformation de modèle, comme étudié précédemment, elles ne sont pas encore maîtrisées à l'heure actuelle.

A côté de ces moyens techniques, il est nécessaire de considérer également les problèmes de fond de l'approche MDA. Parmi ces problèmes, on peut citer entre autres :

- La migration des applications patrimoine vers le MDA ;
- La maintenance des applications développées par le MDA. Est-il obligatoire et possible de remonter au modèle PIM pour le faire ?
- L'application de la séparation des préoccupations de l'AOSD (*Aspect Oriented Software Development*) au MDA. En réalité, un seul modèle PIM n'est pas suffisant pour décrire toutes leurs préoccupations d'une application complexe. Il sera ainsi intéressant de pouvoir utiliser plusieurs modèles dont chacun exprime une facette de l'application ;
- La relation du MDA avec les technologies de composant, de services Web, etc. ;
- L'interopérabilité et la composition d'applications écrites dans des langages de programmation différents et s'exécutant sur des plates-formes différentes.

Le dernier problème représente un point important qui a motivé le MDA. En effet, dans [Soley01], Soley a écrit :

"It's about integration. It's about interoperability…. The need for integration remains strong – we need to build on the success of UML and CORBA … to provide the model-based standards that are necessary to extend integration beyond the middleware approach."

Puisque la composition est le sujet de ce chapitre, nous allons, dans la suite, imaginer la composition d'applications basée sur la composition de modèles de MDA.

4.4. Composition d'applications à base de modèles

4.4.1. Composition de modèles

La raison principale pour laquelle le MDA travaille avec des modèles est de cacher tous les détails techniques liés à l'implémentation. Désormais, pour le MDA, la composition d'applications s'appuiera sur la composition de modèles PIMs de ces applications.

De cette manière, nous ne pouvons composer que des applications dont le modèle PIM est disponible et pour lesquelles la transformation PIM vers le code exécutable – l'application elle-même – est faisable. Ceci implique que nous ne pouvons pas appliquer cette composition pour des applications existantes puisque la transformation bidirectionnelle, entre une application existante et son modèle PIM, n'est pas disponible.

Etant donné deux modèles PIMs, la composition de ces deux modèles consiste à construire un modèle, appelé le *'modèle de relation'*, servant à exprimer et à maintenir les relations, les corrélations, etc. entre les éléments provenant de ces modèles. Plusieurs types de relation peuvent être envisagés :

- La relation de correspondance entre deux éléments pour signifier le fait que ces éléments représentent un même concept, mais modélisés différemment par chaque modèle ;
- L'association entre deux éléments pour désigner une connexion inter-modèles.

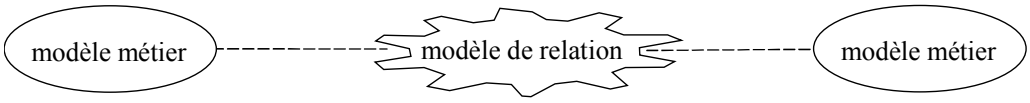

Figure 16. Construire le modèle de relation

En appliquant l'opération de transformation de modèles sur les modèles PIM et le modèle de relation, la composition de modèles est transformée en composition d'applications car :

- Les modèles PIMs sont transformés en les applications à composer ;
- Le modèle de relation sera transformé en un médiateur faisant le pont entre ces applications. L'objectif principal du médiateur est d'assurer la connexion et la communication entre ces applications, ce qui consiste à surmonter l'incompatibilité concernant les données, le protocole de communication, etc.

Cette transformation est illustrée graphiquement par la figure ci-dessous.

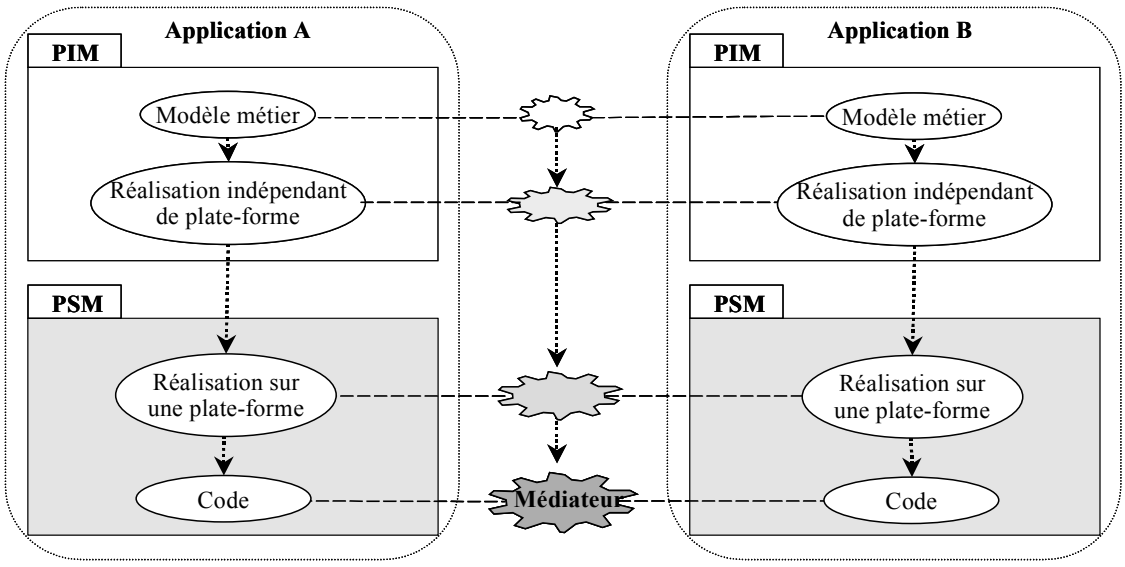

Figure 17. Transformer le modèle de relation

Le principe de cette transformation est simple et intuitif. Cependant, la réalisation n'est pas triviale. Pour illustrer les difficultés de cette transformation, nous proposons de la regarder plus en détail. Supposons que l'on compose deux applications A et B dont les

modèles sont respectivement *m1(A)/MmA1* et *m2(B)/MmB*. Supposons que le modèle de relation entre ces deux modèles soit désigné par *m1(R)/MmR1*. Plusieurs questions doivent être posées à propos de ce modèle de relation :

- Comment concevoir le méta-modèle *MmR1* par rapport à *MmA1* et *MmB1* ?

  Cette question peut être ignorée lorsque l'on travaille au niveau des modèles PIMs car dans ce cas, *MmA1= MmB1 = MmR1 ;* c'est le méta-modèle UML. Dans les autres cas, *MmR* peut être considéré comme étant le résultat d'une combinaison de *MmA1* et *MmB1* auquel est ajouté des concepts pour exprimer des types de relation entre des concepts provenant de ces deux méta-modèles.

- Comment le modèle de relation *m1(R)/MmR1* est-il réalisé ?

  En ce qui concerne cette question, dans [Dzouza01], Dzouza a identifié que les techniques de modélisation actuelles doivent évoluer afin de permettre de:

  - Importer les deux modèles à composer dans le modèle de relation ;
  - Etablir des relations de différents types entre des éléments provenant de ces deux modèles ;
  - Ajouter des attributs et/ou fonctions afin de gérer ces relations, ce qui consiste à gérer les éléments qui participent aux relations.

Lorsqu'on réalise une transformation des modèles de A et de B, le modèle de relation doit, lui aussi, être transformé, afin de pouvoir mettre en relation les nouveaux modèles de A et B. Nous avons donc à faire trois transformations T1, T2 et T3 (cf. Figure 18). La transformation T2 qui transforme *m1(R)/MmR1* $\rightarrow$ *m2(R)/MmR2* doit tenir compte de T1 et de T3 puisque :

- Le méta-modèle *MmR2* doit être construit par rapport à *MmA2* et *MmB2 ;*
- Le modèle *m2(R)/MmR2* doit être construit en fonction des modèles *m2(A)/MmA2* et *m2(B)/MmB2*. En effet, le modèle *m2(R)/MmR2* doit maintenir les relations entre des éléments venant du *m2(A)/MmA2* et *m2(B)/MmB2* comme elles sont définies avant la transformation entre des éléments venant du *m1(A)/MmA1* et *m1(B)/MmB1*. Lorsque ces éléments sont transformés, les relations devraient être transformées pour pouvoir s'adapter à eux.

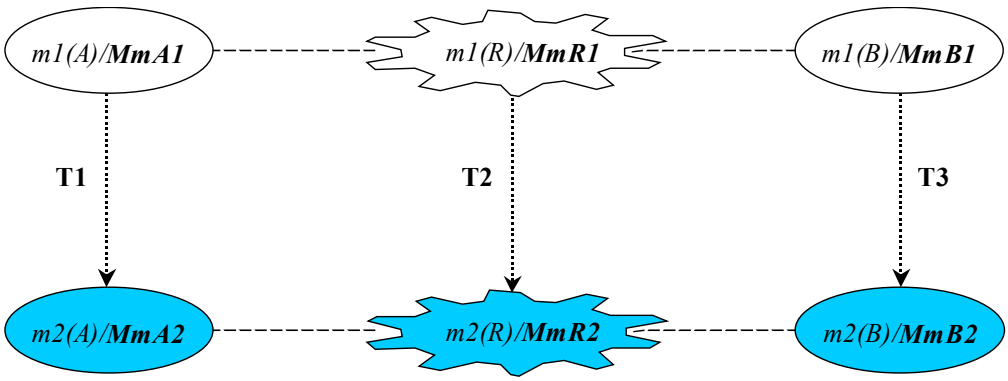

Figure 18. Transformation des modèles

Les problèmes ci-dessus restent encore des points ouverts. Cependant, pour que la composition d'applications basée sur la composition de modèles devienne réalisable, des technologies de modélisation et de transformation de modèles doivent être beaucoup plus évoluées.

Il y a actuellement quelques initiatives autour de la composition de modèles telles que [Cla02, CS02]. Cependant, ces travaux sont un peu différents car leur objectif est de fusionner, par la composition, des modèles afin de créer un modèle unique. Ainsi, à la fin, nous avons une application unique au lieu de plusieurs d'applications qui s'exécutent sur des plates-formes différentes et qui inter-opèrent. De ce fait, ces travaux s'intéressent plutôt sur la séparation des préoccupations dans le MDA.

4.4.2. Vision d'architecture au niveau code

Supposons qu'il soit possible d'établir les modèles de relation entre des applications et de réaliser le processus de transformation de ces modèles. Dans ce cas, pour chaque modèle de relation entre deux modèles PIMs, un médiateur est généré pour assurer les connexions entre deux applications correspondantes. Ce médiateur offre toutes les facilités pour aider ces applications à se coordonner comme spécifié dans le modèle de relation.

De cette manière, l'application résultante contiendra les applications correspondantes aux modèles PIMs composés et un ensemble de médiateurs faisant le lien entre ces applications. Ceci est illustré graphiquement dans la Figure 19.

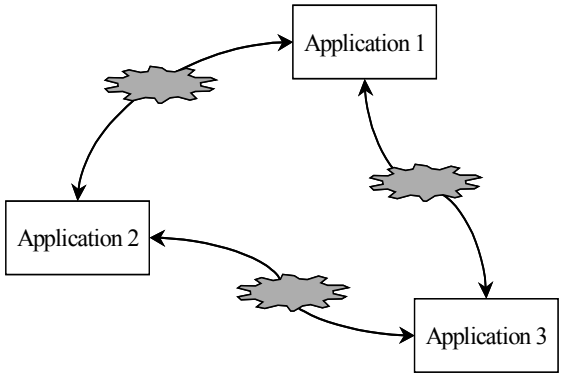

Figure 19. Médiateurs entre applications composées

De manière simple, on peut s'apercevoir que la composition d'applications basée sur la composition de modèles, telle que proposée par le MDA, ne promet pas d'être une solution captivante. Au contraire, il existe plusieurs inconvénients qui sont entre autres :

- La composition de modèles, traduite par la construction du modèle de relation, est difficile avec les techniques de modélisation qui existent actuellement [Dsouza01] ;
- La transformation des modèles (modèles des applications à composer et modèle de relation), et la génération des médiateurs n'est pas un travail trivial ;
- A propos de l'évolution l'application résultante, chaque fois que l'on change une application composée, que ce soit des changements de sa plate-forme d'exécution ou

de ses fonctionnalités, il est nécessaire de re-générer le médiateur assurant les relations entre cette application avec les autres applications ;

- L'ensemble de tous les médiateurs forme un graphe difficile à comprendre, à gérer et à contrôler.

4.5. Discussions

L'idée du MDA de travailler sur des modèles représente une bonne direction pour le génie logiciel du futur. En effet, il est indispensable de pouvoir réfléchir et travailler au niveau du modèle du domaine du problème, en gardant à tout moment une vision globale abstraite. Cette idée sera reprise dans notre approche de composition, mais de façon différente de celle du MDA.

Concernant la manière de travailler avec les modèles du MDA, il reste encore des difficultés concernant :

- La transformation automatique de modèles. Ce n'est pas un travail simple surtout pour valider et assurer la cohérence, la correction, etc. de la transformation. La plupart des techniques actuelles de transformation automatique ne proposent qu'une transformation syntaxique des modèles ;
- La maintenance et l'évolution de l'application. Ceci implique d'avoir une transformation bidirectionnelle des modèles. De cette manière, à tout moment, on peut revenir en arrière pour effectuer des modifications ;
- La composition de modèles dont chacun modélise une préoccupation d'une application. L'idée de cette composition est de fusionner les modèles composés afin de recevoir le modèle complet de l'application. Les transformations sont appliquées sur ce modèle afin de construire l'application ;
- La composition de modèles dont chacun représente une application à inter-opérer. L'idée de cette composition n'est pas de fusionner des modèles mais d'établir le modèle de relation entre les modèles composés. La réalisation de ce processus a beaucoup de limitations que nous avons étudiées précédemment (cf. section 4.4).

A propos du mécanisme de composition proposée par l'approche MDA, nous faisons un résumé en respectant les points que nous avons énoncés au début de ce chapitre comme suit :

- **Elément de composition** : Ce sont les modèles ;
- **Manière de composer** : Lorsque les modèles représentent différentes préoccupations d'une application, la composition est faite par tissage de modèle ;

  En revanche, lorsque les modèles représentent des applications à inter-opérer, la composition est faite par l'établissement de relations entre éléments de ces modèles ;

- **Couplage entre les éléments à composer** : Ce sujet n'est pas abordé dans MDA ;

- **Adaptation et évolution de l'application résultante** : Le MDA fait hypothèse que l'adaptation et l'évolution se font au niveau modèle PIM. L'adaptation et l'évolution d'une application peuvent être faite assez simplement lorsqu'il y a, à tout moment, une correspondance sémantique entre le(s) modèle(s) PIM(s) et l'application ;
- **Réutilisation** : Un modèle PIM peut être réutilisé pour générer des variantes d'une application sur différentes plates-formes.

5. Composition d'éléments logiciels existants

5.1. Motivation

Bien que les approches de construction d'applications présentées précédemment proposent des mécanismes de composition intéressants, elles ne permettent pas de construire des applications à partir d'éléments logiciels existants tels que :

- Des applications patrimoines ou issues du marché ;
- Des composants sur étagère (i.e. des COTS[1]) ;
- Des services Web qui sont des applications autonomes, indépendantes, accessibles à travers l'Internet.

En général, ces éléments sont de bonne qualité, fournissent des fonctionnalités puissantes, couvrent une vaste gamme de besoins fonctionnels et sont peu coûteux. Il est donc important de disposer d'environnements permettant de les composer, dans l'objectif de construire des nouvelles applications fournissant une fonctionnalité plus complexe. Dans la suite de ce chapitre, on va appeler ces applications des **applications globales** ; les éléments logiciels qui participent à la réalisation de l'application globale seront appelés des **participants**.

Dans ce contexte de composition, il est nécessaire de considérer plusieurs problèmes difficiles :

- Les participants sont souvent des boîtes noires. Ceci veut dire que l'on ne peut pas accéder à leur code source ;
- Il est probable que différents participants offrent des fonctionnalités similaires, mais avec des caractéristiques différentes. Le problème posé est de pouvoir sélectionner un bon élément en fonction du contexte d'exécution ;
- Il est très probable que les participants partagent des concepts liés au métier de l'application à construire. Il est, dans ce cas, important d'assurer la cohérence des concepts partagés entre les différents participants ;

[1] Le terme anglophone est *'Commercial Off The Shelf'*. Une étude détaillée de COTS peut être trouvée dans [Vig98, BA99, Car97, VGD98].

- Chaque participant peut avoir sa propre syntaxe, sa propre structure, sa propre manière pour représenter, interpréter et utiliser les concepts partagés. Par conséquent, il est indispensable de résoudre l'hétérogénéité entre les participants en ce qui concerne la syntaxe, la structure et la sémantique des concepts partagés ;
- Les participants peuvent être autonomes et actifs (i.e. ils peuvent changer leur état interne par leur propre initiative). Les applications interactives sont des exemples typiques de participants actifs. Le fait de changer l'état interne d'un participant actif peut imposer aux autres participants de se synchroniser, si le changement est lié aux concepts partagés entre eux ;

Ces problèmes ont été identifiés dans [Car97, Lut00]. Ils sont dus au fait que les participants ne sont pas conçus pour être utilisés dans une application spécifique, mais pour être utilisés de façon autonome. Par conséquent, ils peuvent avoir des fonctionnalités similaires et utiliser des concepts similaires ; ils peuvent aussi avoir des interfaces pour supporter les interactions humaines.

Il est clair que les mécanismes de composition présentés précédemment ne permettent pas de répondre à ces problèmes. L'objectif de cette section est donc d'étudier des mécanismes permettant de tenir comptes de participants ayant les caractéristiques exprimées ci-dessus.

5.2. Contrôle de composition

Comme l'application globale est construite à partir de participants, son comportement sera déterminé d'une part, par les fonctionnalités fournies par les participants et d'autre part, la manière dont les participants sont reliés ensemble. En effet, ce sont des fonctionnalités qui fondent l'application globale mais seulement lorsque la collaboration est établie entre elles [AFGK02, GKAF01]. Par conséquent, la collaboration entre les participants doit être établie d'une certaine façon afin de résoudre le problème de l'application globale.

Il est important de noter que, pour des applications globales, les changements les plus fréquents sont susceptibles de ne pas se produire au niveau des entités métiers du domaine (e.g. la notion d'un compte bancaire) mais plutôt au niveau des règles métiers ou bien des protocoles qui déterminent comment les entités métiers agissent les uns sur les autres (e.g. comment les clients agissent avec leurs comptes). Autrement dit, les changements impliquent moins les participants que la façon de collaborer [AFGK02].

Si l'on appelle **contrôle** ce qui est chargé de : (1) imposer un objectif global à l'ensemble de participants et (2) guider des participants pour travailler ensemble, on peut distinguer deux approches :

- Le **contrôle implicite** dans lequel le contrôle n'est pas séparé des participants. Autrement dit, il est à la charge de tous les participants de savoir comment il faut collaborer avec les autres dans l'objectif de réaliser l'application globale ;
- Le **contrôle explicite** dans lequel le contrôle est séparé des participants. Ce contrôle orchestre et pilote explicitement ce que chaque participant doit faire afin de parvenir

à l'objectif global. Nous considérons cette composition comme *composition par orchestration*.

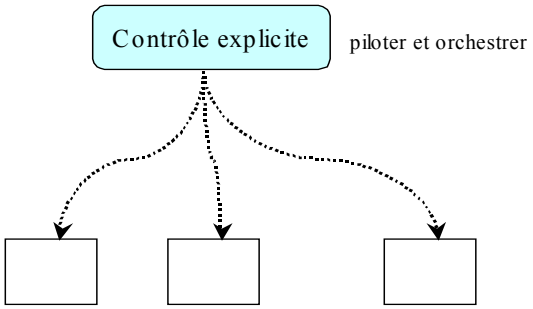

Figure 20. Le contrôle explicite

Le tableau suivant résume les caractéristiques de ces deux approches de composition avec le contrôle implicite et explicite.

Contrôle implicite	Contrôle explicite
Le contrôle est dispersé dans tous les participants	Le contrôle est séparé des participants.
Les participants sont chargés de collaborer entre eux.	Le contrôle explicite est chargé de piloter et orchestrer les participants afin de parvenir à l'objectif global.
Les participants doivent « se connaître ». Le couplage entre participants est fort.	Les participants ne se connaissant pas. Le couplage entre les participants est minimisé.
Composition par intégration.	Composition par orchestration.

Tableau 1. Contrôle implicite vs. contrôle explicite

L'approche EAI (*Enterprise Application Integration)* [Lin00, Lutz00, Mann99, Pol01, LSH03] se base sur le contrôle implicite pour la composition d'éléments logiciels existants. En revanche, l'approche BPM *(Business Process Management)* [CSC02, WC02, LRS02] est une approche représentative qui utilise le contrôle explicite.

Dans la suite de cette section, nous explorons le contrôle implicite à travers l'approche EAI. Pour le contrôle explicite, nous présenterons d'abord la caractérisation du contrôle explicite et ensuite, l'approche BPM. L'utilisation du BPM pour la composition des services Web sera étudiée à la fin de cette section.

5.3. Le contrôle implicite dans l'approche EAI

Mis dans le contexte de réutilisation des applications d'entreprise existantes, l'EAI tente de fournir des moyens permettant, de manière assez simple, de connecter des applications existantes ou à venir dans l'objectif de construire une application globale [Lin00].

5.3.1. Le concept d'adaptateur

Afin de faire travailler ensemble des applications existantes, hétérogènes et autonomes, l'approche EAI se base sur des adaptateurs. En mettant ces applications en collaboration, des adaptateurs sont chargés, entre autres, de :

- Fournir un moyen de transférer le contrôle et les données ;
- Fournir un moyen de concilier l'hétérogénéité au niveau du format de données et du protocole de communication ;
- Assurer la sémantique de la coordination entre ces applications.

Supposons que l'on veuille réaliser une connexion de l'application 1 vers l'application 2 comme l'illustre la Figure 21. Dans le pire des cas où ces deux applications sont existantes et autonomes, il faut créer deux adaptateurs entre elles. L'adaptateur 1 est considéré comme l'adaptateur source, il:

- Observe l'application 1 afin de capturer des actions auxquelles l'application 2 s'intéresse ;
- Signale à l'adaptateur 2 lorsqu'une telle action survient.

L'adaptateur 2 joue le rôle d'adaptateur cible. Il est chargé de :

- Recevoir une demande de l'adaptateur 1 et les données associées ;
- Transformer ces données au format utilisé par l'application 2 ;
- Passer la demande et les données à l'application 2 en respectant son protocole de communication.

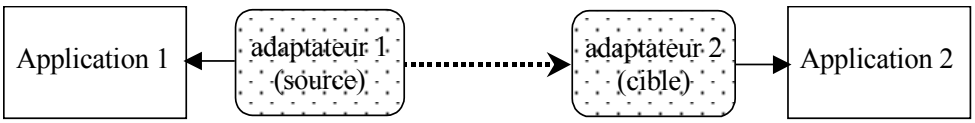

Figure 21. Lier des applications par des adaptateurs

Ces deux adaptateurs doivent jouer à la fois ces deux rôles lorsqu'il s'agit d'une connexion bi-directionnelle entre deux applications.

De cette manière, ce sont les adaptateurs qui assurent la coordination entre les deux applications. Ce sont eux qui décident quand et comment il faut synchroniser les applications. Ils sont donc fortement liés l'un à l'autre en raison des collaborations nécessaires entre les deux applications correspondantes.

Une question est de savoir comment des adaptateurs interviennent dans les applications ?

En général, les applications sont structurées en trois couches : la couche de présentation qui contient l'interface de manipulation (GUI – *Graphical User Interface*), la couche de la logique métier qui implémente la fonctionnalité fournie par l'application et la couche de données où sont gardées les données utilisées par l'application. De cette manière, nous

pouvons décider de faire intervenir des adaptateurs à chacune de ces couches. Ceci nous permet de distinguer trois types d'EAI en se basant sur la couche où intervient l'adaptateur : (1) EAI au niveau données, (2) EAI au niveau métier et (3) EAI au niveau interface d'utilisateur [Lin00].

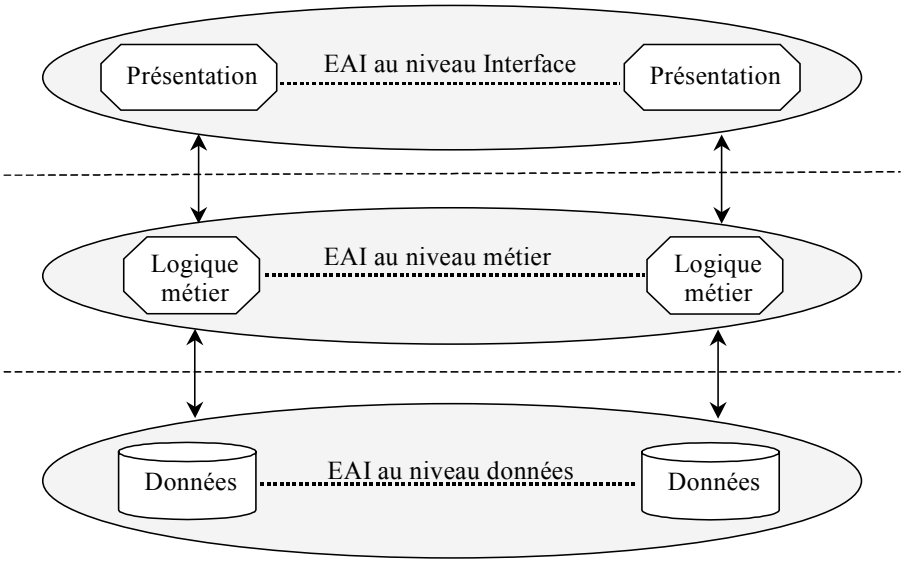

Figure 22. Trois types d'EAI

La réalisation des adaptateurs pour une couche peut être plus simple que pour d'autres couches. Ceci dépend des caractéristiques de chaque application. Ainsi, il faut bien étudier les caractéristiques des applications avant de décider à quelle couche on va réaliser les adaptateurs.

5.3.2. Topologie d'interconnexion

La topologie d'interconnexion représente le plan d'interaction entre les applications participantes. Deux applications ont besoins d'être interconnectées s'il y a un échange de données ou de service entre elles [Vig98].

Pour l'EAI, la topologie d'interconnexion joue un rôle important. Elle permet de structurer les adaptateurs. Le nombre de connecteurs, la construction, le fonctionnement et même le couplage entre les connecteurs dépendent fortement de la topologie d'interconnexion utilisée. Deux types de topologie sont utilisés par l'EAI :

- Topologie point à point ;
- Topologie structurelle.

Avec la topologie point à point, deux applications ayant besoin de communiquer sont liées par une connexion directe entre elles. Cette connexion peut être réalisée d'une façon synchrone par l'invocation de méthode ou bien asynchrone par des événements. Des protocoles de communication synchrone tels que RPC (*Remote Procedure Call*) ou bien asynchrone tel qu'une queue de message, etc. peuvent être adoptés.

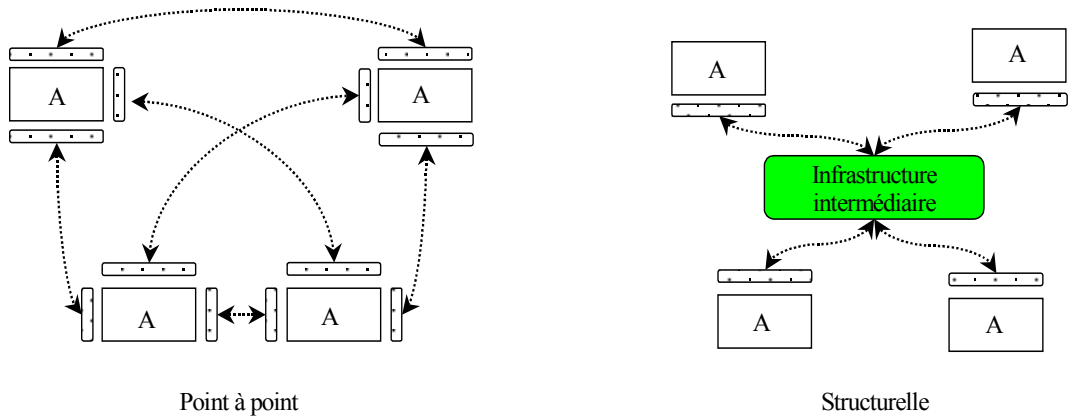

Figure 23. Intégration point à point et structurelle

Cependant, les connexions directes présentent plusieurs inconvénients :

- Une application participante a besoins d'autant d'adaptateurs que le nombre d'applications avec lesquelles elle veut communiquer. Désormais, le nombre total des adaptateurs croît exponentiellement avec le nombre des applications participantes ;
- L'ensemble des connexions entre les applications participantes forme un graphe complexe et implicite qu'il est difficile de gérer ;
- Il est difficile de maintenir et de faire évoluer l'application globale. Le remplacement d'une application participante par un autre, par exemple, demande de reconstruire et de retester toutes les connexions liées avec cette application.

Ainsi, cette topologie n'est satisfaisante que dans les cas où le nombre d'applications à intégrer n'est pas important. De plus, l'application globale ne doit pas évoluer fréquemment.

Avec la topologie structurelle, les applications ne sont pas connectées directement mais à travers une infrastructure intermédiaire. C'est l'intermédiaire qui est chargé de passer le contrôle et les données entre les applications. Par conséquent, les applications participantes ne doivent connaître que l'intermédiaire. Le couplage entre elles sera réduit ; il sera donc plus simple de faire évoluer et de maintenir l'application globale.

Il y a plusieurs façons de réaliser cet intermédiaire ; il peut être :

- Un bus de message (de type publications et souscriptions) qui permet à une application d'envoyer, de façon asynchrone, des messages à plusieurs applications qui ont déclaré s'intéresser à ces messages. Le MOM (*Message Oriented Middleware*) est un exemple typique de ce type de bus ;
- Un bus logiciel qui permet de transmettre des invocations de méthodes entre des applications. L'ORB de CORBA, par exemple, est un bus à objets qui aident à passer des invocations de méthodes entre des applications à objets ;
- Une base de données partagée où une application peut écrire de l'information qu'une autre va lire. C'est grâce à cela qu'elles se coordonnent.

Pour chaque application participante, il est nécessaire de créer seulement un adaptateur afin d'établir la collaboration entre elle et l'intermédiaire. Toutefois, l'adaptateur doit être construit en respectant le mode de réalisation de l'intermédiaire (i.e. communication par l'échange des messages, par l'invocation des méthodes, etc.).

Il est important de noter que l'intermédiaire est au même niveau d'abstraction que les applications participantes. Il n'est pas chargé de guider la coordination entre les participants. Il ne propos qu'un moyen de transférer le contrôle et des données entre les participants en minimisant le couplage entre eux. Ceci différencie de l'approche contrôle explicite que nous allons présenter dans la section 5.4.

Concernant le choix d'une topologie, il faut bien identifier les caractéristiques de l'application globale et aussi celles des applications participantes afin de déterminer une topologie ainsi qu'une infrastructure de communication convenable.

5.3.3. Patrons d'adaptateur d'EAI

Les patrons d'adaptateurs d'EAI ne sont rien d'autre qu'une capitalisation des expériences dans la construction des adaptateurs. Chaque patron est utilisé dans un cas spécifique. Il permet d'optimiser le nombre d'adaptateurs à réaliser et la réalisation de ces adaptateurs.

Nous montrons brièvement cinq patrons qui ont été recensés dans [Lutz 2000] :

- **Messager** qui joue le rôle d'intermédiaire pour passer l'information et le contrôle entre des applications ;
- **Adaptateur** qui est destiné à convertir l'interface d'une application existante en une interface "souhaitée" par d'autres applications ayant besoins d'utiliser cette interface ;
- **Façade** qui fournit une interface simplifiée de l'application afin de minimiser les dépendances entres une application cliente et des applications serveurs ;
- **Médiateur** qui décrit une approche architecturale où la logique d'interaction entre des applications est encapsulée et découplée de ces applications. Ainsi, la maintenance sera plus simple car toute la logique d'interaction est centralisée et gérée par le médiateur ;
- **Automate du procédé** qui tente d'exprimer et d'automatiser la logique d'interaction entre les applications participantes par un procédé.

Parmi ces cinq patrons, certains sont destinés à la topologie point à point tels que l'adaptateur et la façade. Par contre, les autres utilisent la topologie structurelle. De plus, les patrons de médiateur et d'automate du procédé tentent de séparer la logique d'interaction entre des applications participantes afin de mieux les gérer.

Récemment, certains produits d'EAI ont ajouté la capacité d'exprimer et de gérer la logique d'interaction entre des applications participantes par un *workflow* (voir la définition dans la section 5.4.1) tels que NetEAI [NetEAI], Tibco [Tibco], [Mann99] [Van

et al. 01], etc. La gestion de *workflow* dans NetEAI, par exemple, est assez simple. C'est simplement un module qui permet de gérer le flux d'un message entre des applications au sein d'une entreprise à travers un bus de message MOM. Ceci représente certainement une évolution du contrôle implicite vers explicite.

5.3.4. Discussions

Le contrôle implicite en général, et l'EAI en particulier, est une solution simple et naturelle. Lorsque des applications hétérogènes ne sont pas construites pour travailler ensemble, la façon la plus intuitive est de créer des adaptateurs afin de pouvoir les connecter directement. Celle-ci ne peut s'appliquer que pour des cas simples.

Nous faisons un petit résumé sur l'EAI en respectant les points énoncés dans le début de ce chapitre :

- **Eléments de composition** : Ce sont les applications d'entreprise existantes ;
- **Manière de composer** : EAI se base sur le concept d'adaptateur afin d'ajuster les applications existantes dans l'objectif de les faire travailler ensemble. Ce sont les adaptateurs qui sont chargés de faire coordonner les applications participantes. La logique de la composition est dispersée dans les adaptateurs ;
- **Couplage entre les éléments à composer** : Avec la topologie point à point, les adaptateurs (et les applications participantes, en conséquence) sont fortement liés. En revanche, avec la topologie structurelle, ils sont moins couplés ;
- **Adaptation et évolution de l'application résultante** : Comme la logique de la composition est dispersée et implicite dans les adaptateurs des applications participantes, il est difficile de faire évoluer et d'adapter l'application résultante ;
- **Réutilisation** : C'est l'objectif d'EAI que de travailler avec les applications existantes.

Ce qui est important à noter est que l'EAI ne propose que des solutions implicites pour les problématiques que nous avons énoncées dans la section 5.1 tels que le partage des concepts, l'hétérogénéité, l'initiative des applications, etc. Ceci parce que les solutions pour ces problèmes sont cachées dans les adaptateurs. Puisque tout est implicite, il est très difficile de proposer, de valider et de faire évoluer ces solutions.

5.4. Le contrôle explicite

5.4.1. Caractérisation du contrôle

Comme nous l'avons présenté dans la section 5.2, le contrôle explicite est chargé de piloter et d'orchestrer les applications participantes en imposant un objectif global. Pour cela, il y a deux aspects importants qui sont à la charge du contrôle :

- Exprimer, d'une manière explicite, l'objectif global sous forme de la coordination entre des participants ;
- Piloter les participants en imposant l'objectif global.

Il y a plusieurs formalismes possibles pour exprimer la logique de coordination tels que les grammaires [FHS92], les réseaux de Pétri [VM94], etc. Cependant, ils ne sont ni intuitifs, ni simple à visualiser.

Récemment, il est apparu l'idée d'utiliser des *workflows* pour réaliser le contrôle explicite [Mann99, LR97, VK02]. L'objectif de cette section est d'étudier les *workflows* et également les systèmes de gestion de *workflow* (WfMS – *Workflow Management System*) afin de savoir pourquoi et comment ils peuvent être un candidat approprié et performant pour réaliser le contrôle explicite.

Nous proposons de commencer par un historique sur les *workflows*.

5.4.1.1. Un peu d'histoire

Au début des années 80, il y a eu une révolution concernant le travail coopératif. Il s'agissait d'utiliser des technologies informatiques afin de gérer la coopération entre les membres d'un groupe qui ne travaillent pas forcement en même temps et au même endroit [CVK99].

Cette révolution a fait naître la notion de travail coopératif géré par l'ordinateur (*CSCW – Computer Supported Cooperative Work*, en anglais) et la notion de *groupware* désignant des systèmes informatiques construits pour assurer les objectifs du CSCW.

La coopération peut être décomposée en trois axes : la communication, la collaboration et la coordination. La **communication** concerne l'échange des informations entre les membres d'un groupe. La **collaboration** est considérée comme étant le partage d'un espace où les membres peuvent stocker, traiter et partager un ou plusieurs documents communs. Par contre, la **coordination** consiste à ajuster la contribution des membres au travail commun. Pour cela, la coordination décompose ce dernier en un ensemble de tâches. Elle établit ensuite des règles d'assignation de tâches aux membres ainsi que l'enchaînement de ces tâches afin d'assurer le fonctionnement correct du groupe.

Les systèmes de *groupware* ont été crées pour répondre à ces trois axes de coopération. Les groupwares de communication explorent des capacités de communication personnelle et collective. Des exemples de cette catégorie sont les courriers électroniques, la vidéo conférences, etc. Par contre, les systèmes de gestion de configuration ou bien les éditeurs coopératifs sont des exemples de groupwares de collaboration.

Les systèmes de gestion de *workflow* – WfMSs – ont été crées afin de gérer la coordination. Pour ce faire, ils permettent de capturer, de modéliser et d'exécuter un *workflow* qui représente la coordination entre les membres d'un groupe. A un moment donné, un membre peut recevoir des indications (*guidelines*, en anglais), et même des données concernant la tâche à faire.

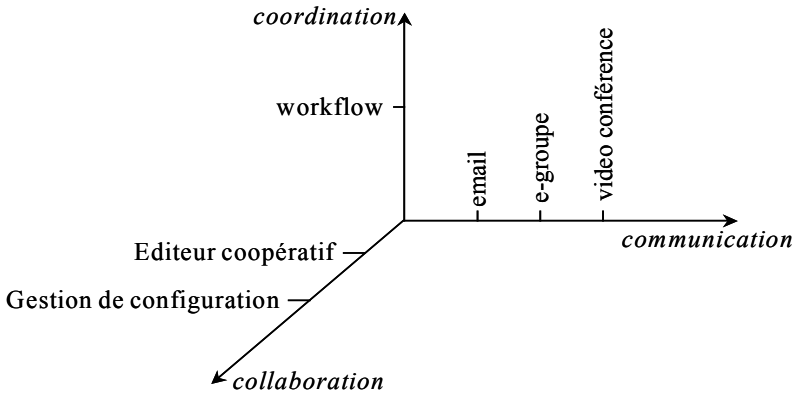

Figure 24. Des groupwares de trois axes de coopération

Initialement, les WfMSs ne considèrent que des humains pour réaliser des tâches. Ils jouent donc plutôt le rôle de planification ou bien de gestion de projet. Aujourd'hui, il est possible d'intégrer des applications au sein des WfMSs afin d'automatiser certaines tâches. Ils dépassent donc le domaine du CSCW et sont partiellement ou totalement automatisés.

5.4.1.2. Systèmes de gestion de workflow

Un **procédé** désigne l'ensemble des activités faisant intervenir des personnes ou bien des applications dans l'objectif de réaliser un but. Un modèle de procédé contient donc la description des activités, des produits, des prescriptions et des contraintes. Un **workflow** est l'automatisation d'un procédé dans lequel des produits et des activités sont enchaînées entre des participants – ceux qui sont chargés de réaliser des activités – en suivant des prescriptions et des contraintes définies dans le modèle de ce procédé [WfMC96].

Un WfMS est un système qui permet de définir, de créer et de gérer l'exécution des *workflows* en utilisant des moteurs qui sont capables d'interpréter le modèle de procédé et d'interagir avec les participants [WfMC96].

Bien que différentes solutions aient été proposées par différents WfMSs, le processus de gestion d'un workflow peut se résumer en trois grandes étapes :

- Modélisation du procédé ;
- Implémentation du *workflow* ;
- Exécution du *workflow*.

Modélisation du procédé

L'objectif de cette étape est de construire un modèle de procédé. Ce dernier permet de représenter une planification pour réaliser un but.

Un modèle de procédé est souvent défini en se basant sur trois concepts fondamentaux : activité, produit et ressource. Avec ces trois concepts, on peut distinguer trois approches principales de modélisation des procédés : (1) orientée activité, (2) orientée produit et (3)

orientée ressource. Parmi ces trois approches, l'approche orientée activité fournit une vision globale et claire des procédés. Elle est donc la plus souvent utilisée.

Dans l'approche orientée activité, le concept d'activité est le concept central qui représente une étape (une unité de travail, un pas, etc.) contribuant à réaliser un but. Un procédé est donc décomposé en activités, interconnectées séquentiellement ou parallèlement, dont une initiale et une (ou plusieurs) finale. Un exemple de modèle de procédé est illustré dans la Figure 25 avec une activité initiale (activité 1) et deux finales (activité 5 et 6) qui représentent deux possibilités de terminer le procédé.

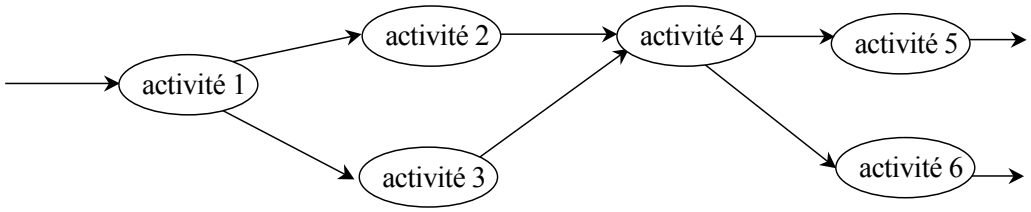

Figure 25. Un modèle de procédé orienté activité

Les interconnexions entre les activités permettent de spécifier le flot de contrôle et/ou le flot de données. Le flot de contrôle représente l'enchaînement et l'ordonnancement des activités. Par contre, le flot de données représente l'échange des produits ou des données entre elles. Un produit est une information qui peut être créée, transformée ou consommée par une activité. Une activité peut demander en entrée un ou plusieurs produits et produire en sortie un ou plusieurs produits. Selon le WfMS, les produits sont typés ou pas.

Chaque activité peut exiger une ressource (humaine ou matérielle). Cette ressource représente ce qui réalise le travail matérialisé par l'activité. La plupart des WfMSs actuels établissent statiquement l'association entre les activités et leurs ressources lors de la phase de modélisation.

Implémentation du procédé

L'objectif de cette étape est d'établir physiquement l'association entre des activités ayant besoins d'une ressource matérielle, et leur ressource comme spécifié dans le modèle de procédé.

Pour cela, il est nécessaire de traiter le problème de l'interopérabilité entre le moteur de *workflow* et l'ensemble des applications (i.e. les ressource matérielles) utilisées pour implémenter des activités du procédé. Ensuite, il faut établir physiquement les liens entre eux pour que le moteur puisse interagir avec ces applications.

Exécution du *workflow*

L'exécution d'un *workflow* commence par l'activité initiale et doit se terminer par une activité finale, en passant par les activités intermédiaires et en respectant les règles d'ordonnancement définis dans le modèle de procédé. Pour ce faire, le moteur de *workflow* doit assurer :

- L'invocation de chaque application participante lorsque l'activité correspondante est démarrée ;
- L'enchaînement des produits entre les activités comme défini par le flot de données dans le modèle de *workflow*. Ceci implique le passage des données entre applications participantes ;
- La cohérence et la fiabilité de l'exécution du *workflow* ;
- La surveillance de l'exécution et la gestion des erreurs du *workflow*.

5.4.1.3. Réalisation du contrôle explicite par un WfMS

L'utilisation d'un WfMS pour réaliser le contrôle explicite peut apporter plusieurs avantages :

- Un modèle de procédé peut être utilisé pour représenter l'objectif global sous forme de la coordination entre des participants. Les modèles de procédé sont, en général, assez intuitifs et simples à comprendre. Ils fournissent, par conséquent, un langage de composition intéressant ;
- Un moteur de *workflow* est capable d'interpréter un modèle de procédé et d'invoquer des participantes en respectant le modèle de procédé. Un moteur de *workflow* est tout à fait adéquat pour jouer le rôle d'un contrôle explicite. La Figure 26 montre le guidage des participants par un moteur de *workflow* ;
- L'application globale sera séparée clairement en deux couches : (1) les fonctionnalités de base et (2) la coordination de ces fonctionnalités. La première couche est assurée par les participants tandis que la coordination est déléguée à un WfMS. A chaque étape dans la phase de construction de cette application globale, il faut tenir compte de la relation entre ces deux couches ;
- Un WfMS est un outil générique qui peut être utilisé dans des domaines d'application différents.

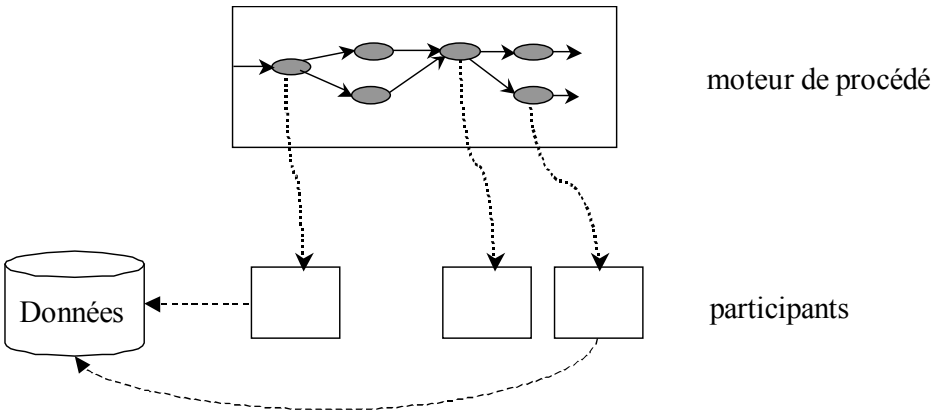

Figure 26. Le moteur de *workflow* guide les participants à travailler ensemble

5.4.2. La gestion des procédés d'entreprise – le BPM

5.4.2.1. Qu'est ce que le BPM ?

Le BPM s'intéresse à la manière dont des entreprises offrent leurs services métiers (*business service*) aux clients. Puisque les entreprises se basent sur leurs éléments logiciels existants tels que des applications, des bases de données, des services Web, etc., il est certain que ces éléments ne peuvent pas rester isolés, ils doivent, au contraire, collaborer afin d'accomplir le service métier de l'entreprise [CSC02].

Le BPM travaille donc sur la composition des éléments logiciels existants en respectant les règles du jeu d'un service métier d'une entreprise. Le résultat de cette composition est considéré comme le procédé d'entreprise (*business process*).

Aujourd'hui, comme les entreprises ont besoin de collaborer avec des partenaires afin de mieux servir leurs clients, les procédés d'entreprise deviennent de plus en plus complexes. Ces procédés peuvent :

- Etre très grand en dépassant la frontière de l'entreprise, c'est-à-dire qu'ils couvrent non seulement des activités réalisées au sein de l'entreprise mais aussi chez des partenaires. Des chaînes logistiques (*supply-chain*) dans laquelle le procédé couvre plusieurs fournisseurs, l'entreprise et des clients sont des exemples typiques de procédés d'entreprise [CE93] ;
- Etre une collaboration entre des procédés indépendants s'exécutant simultanément dans des environnements différents, possiblement hétérogènes. Ces procédés peuvent appartenir à des partenaires différents ;
- Etre très dynamiques afin de mieux répondre aux besoins spécifiques des clients et aux changements des relations avec des partenaires ;
- Mettre très longtemps à s'exécuter et pouvoir échouer partiellement.

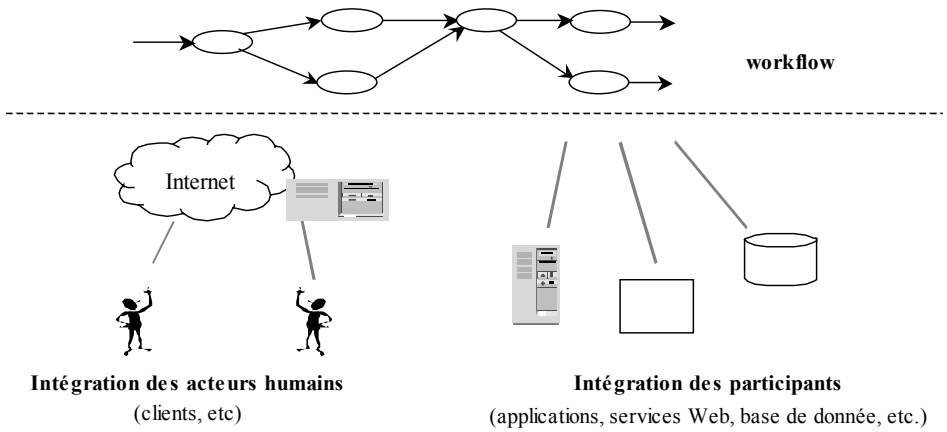

Figure 27. Procédé d'entreprise avec BPM

Pour supporter un procédé d'entreprise, ce que le BPM doit considérer peut être traduit principalement par :

- La gestion du *workflow* représentant la vision globale du procédé d'entreprise ;
- L'intégration des acteurs humains ainsi que des éléments logiciels existants et hétérogènes au *workflow* afin d'implémenter le procédé d'entreprise.

Il est certain que, ceci correspond exactement au paradigme de composition par contrôle explicite dans lequel le modèle de procédé sert, en même temps, à :

- Exprimer la manière dont l'entreprise propose le service métier aux clients ;
- Exprimer la logique de coordination entre des participants afin d'accomplir le service métier de l'entreprise.

Le *workflow* est, sans doute, l'entité de première classe du BPM. Ceci explique pourquoi la plupart des efforts du BPM se concentrent sur la définition de langages destinés à définir et à exécuter des procédés d'entreprise. Evidemment, ces langages doivent faciliter l'intégration des participants aux *workflows*.

Il est important de noter que le BPM n'impose aucune contrainte sur le type de participant à utiliser. Toutes sortes d'élément logiciel, existant ou à venir, peuvent être utilisés pour implémenter le procédé d'entreprise. Pourtant, les services Web sont particulièrement visés. Nous allons étudier ce cas particulier dans la partie 5.4.3.

5.4.2.2. Améliorations des WfMS traditionnels

Les WfMSs « traditionnels » sont assez limités pour faire face aux caractéristiques des procédés d'entreprise :

- Chaque activité ayant besoin d'une ressource matérielle, est associée à une application qui l'implémente. Cette association est souvent déterminée par des contraintes définies lors de la définition du modèle de procédé. Puisque les contraintes d'association font partie du modèle de procédé, la flexibilité et la réutilisation des modèles de procédés sont donc limités ;
- L'association entre une activité et l'application qui l'implémente est figée à l'exécution. Ceci implique des difficultés concernant l'utilisation des WfMSs dans un environnement dynamique comme dans le cas des services Web ;
- L'interopérabilité entre des WfMSs est assez limitée malgré des efforts de l'association de coalition de workflow – WfMC (*Workflow Management Coalition*) ;
- La capacité d'intégration des applications hétérogènes aux *workflows* est assez limitée et manque de flexibilité.

Par conséquent, il est nécessaire d'améliorer des WfMSs traditionnels afin de pouvoir les utiliser dans le contexte de BPM. Les améliorations principales consistent à faciliter : (1) la définition et la réutilisation des modèles de procédé et (2) l'adaptation et l'évolution des procédés d'entreprise.

En ce qui concerne la définition et la réutilisation des modèles de procédé, il est nécessaire de définir un langage permettant de :

- Définir des modèles de procédé complexes pouvant contenir et/ou référencer d'autres modèles ;
- Faciliter l'intégration des participants au sein du procédé d'entreprise, même à l'exécution et en fonction de la disponibilité des participants. Ceci est surtout important pour la composition des services Web ;
- Eliminer les dépendances vers les participants implémentant les activités du procédé. Ceci permet de mieux réutiliser le modèle de procédé dans des contextes différents.

Actuellement, plusieurs langages ont été proposés, tels que BPML[2] [BPML], BPEL4WS[3] [WC02, BPEL4WS], etc. visant à définir des modèles de procédé. Cependant, ces langages sont strictement appliqués aux procédés d'entreprise qui n'utilisent que des services Web comme participant. C'est pour cette raison que nous présenterons ces langages après avoir présenté la composition des services Web.

En ce qui concerne l'adaptation et l'évolution des procédés d'entreprise, il est nécessaire d'augmenter la flexibilité, la dynamicité et l'adaptabilité des WfMSs à propos de :

- L'association entre une activité et le participant qui l'implémente. Ceci doit permettre de retarder le plus tard possible le choix des participants et même de changer des participants à l'exécution du procédé d'entreprise ;
- La modification du modèle de procédé, même en l'exécution. Ceci permet de changer le modèle de procédé afin de mieux servir des clients particuliers ou de répondre à une erreur inattendue.

La Figure 28 montre deux endroits où des améliorations peuvent être offertes par les WfMSs afin d'augmenter la flexibilité des procédés d'entreprise.

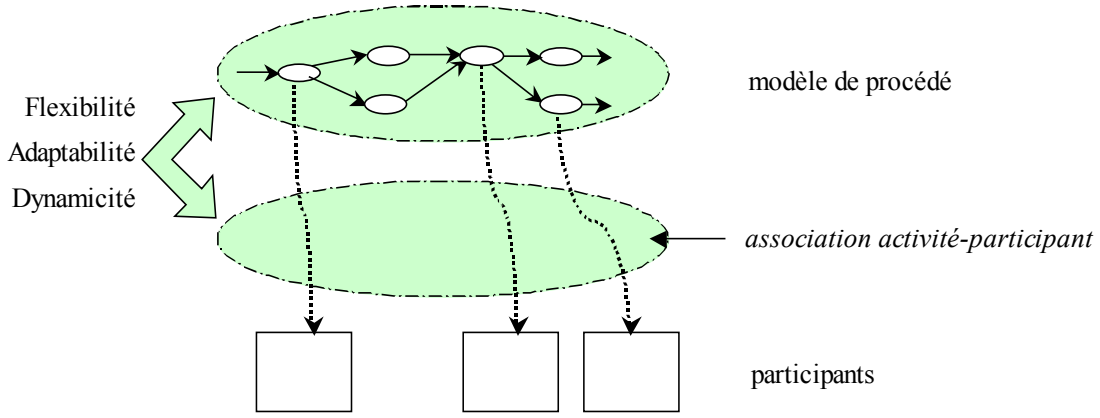

Figure 28. Amélioration des WfMSs

Il y a beaucoup de travaux qui s'intéressent à ces améliorations tels que [WSR99, HSW01, YP02, Cas et al.00, HG03]. Parmi ces travaux, il y en a qui considèrent des applications

[2] Business Process Modelling Language

[3] Business Process Execution Language for Web Service

comme des participants, et d'autres qui utilisent des services Web comme participants. Nous en étudions deux dans la section de composition de services Web (cf. section 5.4.3.3).

Pour la suite, nous allons regarder un cas spécifique de l'approche BPM dans lequel les participants sont des services Web. Nous allons regarder également les langages de définition des procédés d'entreprise fondés sur des services Web et les améliorations pour augmenter la flexibilité et l'adaptabilité des WfMSs.

5.4.3. Composition de services Web

Depuis ces dernières années, le Web devient la plate-forme par laquelle beaucoup d'entreprises communiquent avec leurs partenaires et proposent leurs services aux clients. Par conséquent, les services Web, qui sont des applications indépendantes pouvant être publiées, localisées et accédés à travers le Web [YP02, WSDL], sont très utilisés. Les exemples de services Web sont des applications pour réserver un hôtel, acheter un ticket d'avion, visiter un monument, etc. à travers le Web.

Individuellement, chaque service fournit une fonctionnalité limitée. Ainsi, apparaît le besoin de composer des services Web afin de construire un service composite proposant une fonctionnalité de plus haut niveau d'abstraction. Le service composite '*planifier un voyage*', par exemple, composé des services ci-dessus, proposera aux clients des services complets concernant un voyage. Ce service permet d'une part, de rendre des services plus adaptés aux besoins des clients et d'autre part, d'étendre et de spécialiser les services Web existants.

Il y a deux types principaux de composition de services Web :

- **Composition statique**. Il s'agit un type de composition dans lequel les services Web à composer sont pré-calculés avant qu'un client ne fasse une requête du service composite. Ce type de composition peut être appliqué dans des environnements « stable » où les services Web participants sont toujours disponibles et où le comportement du service composite est le même pour tous les clients ;
- **Composition dynamique**. Les services Web à composer sont déterminés lors de l'exécution de la requête d'un client. Ils peuvent être déterminés selon les besoins, les contraintes de chaque client, la disponibilité des services Web, etc.

La composition dynamique apparaît la plus intéressante. D'une part, elle promet d'être capable de faire face à un environnement très dynamique dans lequel des services apparaissent et disparaissent rapidement. Ces services peuvent également être des services similaires. D'autre part, elle permet de mieux satisfaire les besoins de chaque client.

La plupart des travaux actuels se concentrent sur la composition dynamique des services Web [YP02, MSZ01, Cas et al.00, WC02, ZBNN01]. Dans la suite de cette section, nous regardons rapidement cette composition.

5.4.3.1. Difficultés de la composition dynamique de services Web

Contrairement à la composition d'applications existantes, où les applications à composer sont disponibles d'une manière prévisible et stable, celle de services Web doit faire face à un environnement dynamique. Ceci implique de considérer plusieurs points :

- La découverte de « bons » services pour composer selon des contraintes et des besoins.
- La dynamicité et la flexibilité dans la composition de services Web.

La découverte des services Web

Cette tâche joue un rôle très important pour la composition dynamique des services Web. Elle est liée essentiellement à : (1) la représentation, d'une manière uniforme, des services Web et (2) la découverte des services en fonction des contraintes et des besoins sans distinction de leurs moyens d'accès. Naturellement, le deuxième point est fortement lié au premier.

Actuellement, il y a deux approches principales qui sont basées respectivement sur : la syntaxe et la sémantique pour décrire et découvrir des services Web.

Lié à la première approche, le langage WSDL (*Web Service Description Language*) permet de décrire les fonctionnalités, les données (sous forme de messages) et également le protocole d'accès des services Web [WSDL]. Le standard UDDI (*Universal Description, Discovery and Integration)*[UDDI] est un registre de service permettant d'identifier et de localiser un service Web en se basant sur le WSDL.

En ce qui concerne la deuxième approche, DAML-S (*DARPA Agent Markup Language*) [DAMLS] permet de décrire la sémantique des services Web en se basant sur des ontologies [MSZ01]. DAML-OIL [DAML-OIL] est un exemple de formalisme permettant de définir des ontologies. Cependant, ceux ci sont hors de la portée de cette thèse car nous nous intéressons essentiellement à la composition, y compris la composition dynamique des services Web.

Mécanisme de composition

Les services Web étant dynamiques, le paradigme de composition par le contrôle explicite sera le plus adapté puisque :

- On ne peut pas construire les adaptateurs car l'on ne sait pas avec quel service Web et comment il faut adapter ;
- On ne peut pas laisser aux services Web la responsabilité liée aux décisions de coordination car ils ne savent pas avec qui il faut coordonner, quand et comment.

Il est donc nécessaire de disposer d'un contrôle explicite pour tenir compte de l'objectif du service composite et pour aider les services à se coordonner afin de parvenir à cet objectif.

La plupart des solutions pour la composition dynamique de services Web se fondent sur le contrôle explicite, mais avec des réalisations différentes. Le [YP02], par exemple, se base

sur son propre formalisme – WSCL (*Web Service Composition Language*). Ce dernier est similaire avec des formalismes issus du domaine des procédés. Néanmoins, des *workflows* sont les plus utilisés [Cas et al.00, CS02, HG03]. Ceci est prouvé par l'apparition de plusieurs langages permettant de définir et de gérer des procédés d'entreprise qui s'appliquent à la composition de services Web.

5.4.3.2. Quelques langages de définition des procédés

A l'heure actuelle, il y a quelques langages qui sont destinés à la définition et l'exécution des procédés d'entreprise qui sont implémentés par des services Web. Des exemples de ces langages sont le WSFL, le BPML, le BPEL4WS, WSCI, etc. Les objectifs principaux de ces langages sont de :

- Proposer un langage formel de définition des procédés ;
- Décrire le comportement visible des procédés d'entreprise de façon indépendante de l'implémentation. Pour ce faire, ces langages proposent de définir des procédés en se basant sur des ports et des messages des services Web qui sont potentiellement utilisés pour les implémenter ;
- Prendre en compte des besoins principaux des procédés d'entreprise : temps d'exécution potentiellement long, gestion des transactions, gestion des erreurs et d'exceptions, etc.

BPEL4WS	**BPML**
Langage de définition de procédé exécutable basé sur XML. Ile est crée par IBM, Microsoft, et BEA en fondant sur XLANG et WSFL.	Langage de définition de procédé exécutable basé sur XML. Il est défini par l'association BPMI *(Business Process Management Initiative)*
Ne supporte pas la définition de procédés imbriqués. Mais, supporte la composition « récursive » en considérant qu'un service composite est un service.	Supporte la composition récursive par l'usage des procédés imbriqués qui sont liés au contexte de définition (i.e. le procédé père).
Gère des compensations et des exceptions.	Gère des compensations, des exceptions, des délais de gardes.
Procédé orienté activité : • activité simple pour référencer une action; • activité structurée pour structurer des activités dans un groupe.	Procédé orienté activité : • activité simple correspond à une action; • activité complexe désigne une structure des sous-activités ou bien un sous-procédé.
Actions possibles pour des activités simples : *receive, reply, invoke, assign, compensate, delay, empty, throw, terminate*.	Actions possibles pour des activités simples : *call, action, assign, compensate, wait, empty, throw/catch*
Pour structurer des activités définies dans un groupe d'activité : *sequence, switch, while, flow, pick*.	Pour structurer des activités dans une activité complexe : *all, choice, foreach, sequence, switch, until, while.*
Pour spécifier un service attendu en se basant sur WSDL : serviceLinkType, partners.	Attribut de l'activité utilisé pour localiser le service utilisé, si nécessaire : locate.

Tableau 2. BPEL4WS et BPML

Le tableau ci-dessus propose une comparaison entre ces deux langages de définition de procédés d'entreprise.

5.4.3.3. Quelques systèmes de composition dynamique de services Web

Dans cette partie, nous nous concentrons principalement sur quelques manières d'améliorer des WfMSs afin de mieux s'adapter à la composition dynamique des services Web.

SMWM - Service Mediating Workflow Management

Il s'agit d'une étude de l'université de Twente, Pays-Bas, qui tente d'améliorer l'association entre le *workflow* et des services Web en ajoutant un médiateur entre ces deux couches [HG03].

Le médiateur permet de dissocier le lien direct entre le *workflow* et les services Web. Il est donc chargé d'établir dynamiquement l'association entre chaque activité et son service Web correspondant en exécution. Pour ce faire, il est basé sur deux informations :

- Des **contrats d'association** qui spécifient la correspondance entre chaque activité et un ensemble de service qui sont capables de l'implémenter. Ces contrats sont établis lors de la phase de définition de modèle de procédé ;
- Des **politiques de sélection** qui permettent de choisir, à l'exécution, puis d'invoquer le service le plus approprié pour une activité, dans le cas où il y aurait plusieurs candidats possibles.

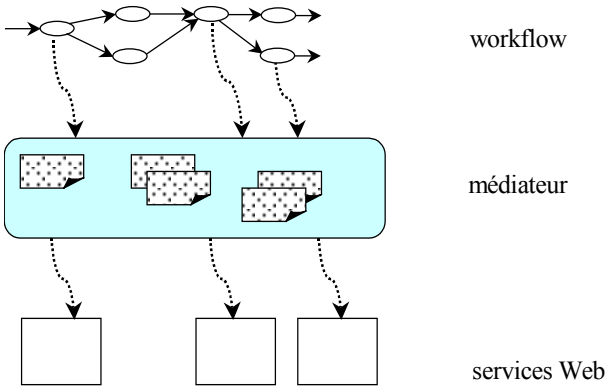

Figure 29. Médiateur dans SMWM

Grâce au médiateur, la définition des modèles de procédé ne contient plus d'information concernant des services utilisés. Ceci rend des *workflows* plus indépendant et plus flexible dans l'établissement de l'association avec ces services.

eFlow

Il s'agit d'un système de composition de services Web issue de HP (*Hewlett-Packard)* [Cas et al.00] qui a pour objectif d'augmenter la flexibilité et la dynamicité des services composites à travers :

- La capacité de changer l'implémentation d'un service composite afin de s'adapter à l'environnement avec peu ou pas d'intervention humaine. Ceci correspond, en effet, au fait de varier l'association entre une activité et les services Web qui implémentent cette activité.
- La capacité de modifier le modèle de procédé à l'exécution afin de répondre à un besoin spécifique d'un client ou bien une situation exceptionnelle inattendue.

Pour le premier point, *l'eFlow* propose :

- Des règles de sélection dynamique d'un service pour implémenter une activité ;
- Des activités multiples dont le nombre d'instance est déterminé en exécution. Par conséquent, ces activités peuvent être associées à autant de services Web que leur nombre d'instances ;
- Des activités génériques dont chacune peut être remplacée par un modèle de procédé. Ces modèles, à leur tour, peuvent être implémentés par différents services Web.

Pour le deuxième point, *l'eFlow* offre des moyens pour changer le modèle de procédé pour une ou plusieurs instances de *workflow* en respectant des règles de cohérence et en suivant le protocole de migration depuis l'ancien vers le nouveau modèle de procédé. Le détail de règles peuvent être trouvées dans [Cas et al.00].

5.4.4. Discussions

Tout d'abord, nous faisons un résumé sur le BPM et la composition de services Web en respectant les points énoncés au début du chapitre comme suit :

- **Eléments de composition** : Ce sont les éléments logiciels existants, surtout des services Web ;
- **Manière de composer** : Dans BPM, l'orchestration est utilisée comme étant le moyen de composer les éléments logiciels.

  Pour ce faire, BPM sépare l'application globale en deux couches : (1) les éléments logiciels fournissent les fonctionnalités dont l'application globale a besoin et (2) la coordination entre ces éléments logiciels afin de réaliser l'application globale. Cette deuxième couche est représentée par un modèle de procédé et pilotée explicitement par un moteur de *workflow* ;
- **Couplage entre les éléments à composer** : Avec l'orchestration, le couplage entre les éléments logiciels est minimisé. En plus, comme BPM travaille également avec les éléments existants, ces éléments peuvent être construits indépendamment les uns des autres ;
- **Adaptation et évolution de l'application résultante** : Avec deux niveaux de programmation, l'adaptation et l'évolution de l'application globale peuvent demander de :

- **Modifier l'implémentation du procédé d'entreprise** : Ceci correspond à remplacer un élément logiciel par un autre. Ce travail sera simple à faire si le couplage entre les activités et les services qui les implémentent est minimisé. Avec les améliorations que nous avons discutées dans la section 5.4.2.2, le fait de remplacer un élément logiciel par un autre peut être fait aisément ;

- **Modifier le modèle de procédé** : Ce changement implique souvent de re-implémenter une partie ou tout le procédé d'entreprise.

En résumé, nous pouvons dire que l'évolution et l'adaptation des applications résultantes peuvent être faites avec BPM, mais ceci dépend de la flexibilité de chaque implémentation de BPM. De plus, lorsque l'on modifie le modèle de procédé, il est difficile de prédire les impacts sur l'implémentation de ce procédé d'entreprise ;

- **Réutilisation** : Comme BPM travaille avec les éléments logiciels existants, il est clair que ces éléments sont réutilisables dans plusieurs procédés d'entreprise différents.

  A propos de la réutilisation des modèles de procédé, ceci est possible à condition que ces modèles ne référencent pas explicitement les éléments logiciels qui les implémentent. Ceci dépend de chaque langage de procédé. Le BPEL4WS, par exemple, ne déclare les dépendances d'un procédé que vers des interfaces des services Web. Un tel procédé peut être utilisé pour orchestrer les services Web possédant les mêmes interfaces.

A propos des problématiques concernant la réutilisation des éléments logiciels que nous avons abordées dans la section 5.1, il reste encore des problèmes pour lesquels BPM ne propose pas de solution :

- Le partage des concepts entre les éléments logiciels et le besoin de garder la cohérence entre les éléments logiciels à propos des concepts partagés ;
- L'initiative et l'autonomie des éléments logiciels ;
- Une vue uniforme et de haut niveau d'abstraction pour les éléments logiciels.

Malgré tout, BPM est nettement plus intéressant que EAI. BPM peut être considéré comme l'état de l'art des mécanismes de composition.

6. Résumé et conclusions

Ce chapitre a fait un parcours des différents mécanismes de composition proposés par les approches actuelles pour la construction d'applications. Chaque approche se concentre sur des unités de composition différentes ; elle propose donc un opérateur de composition qui adapte à ses unités de composition. De cette manière, il n'est pas simple de faire une comparaison de ces approches.

Le tableau ci-dessous essaie de montrer des caractéristiques principales de ces mécanismes de composition.

	Composant	Conteneur	AOP	MDA	EAI	BPM
Sujet de composition	composants	services NF et composants	programme de référence et aspects	modèles	éléments existants	éléments existants
Opérateur de composition	par connexion	par coordination	par tissage/intégration	par intégration	par intégration	par orchestration
Type de composition	boite noire	boite noire	boite blanche	boite blanche	boite noire	boite noire
Contrôle explicite de la composition	non	oui	non	non	non	oui
Évolution de l'application résultante	oui		difficile	oui	difficile	oui
Adaptation de l'application résultante		oui	difficile	oui	difficile	oui

Tableau 3. Bilan récapitulatif des approches

Chacune des approches ci-dessus apporte une ou plusieurs idées intéressantes. Ainsi, il est important de tirer les points forts de ces approches afin de les réutiliser et/ou de s'en inspirer:

- De l'approche à composant, nous gardons l'idée de modéliser les éléments logiciels de diverses natures afin d'en avoir une vision uniforme. En plus, il est nécessaire d'isoler la partie fonctionnelle de la partie non-fonctionnelle des éléments logiciels afin de faciliter leur réutilisation ;

- De l'approche AOP, nous retiendrons un mécanisme de programmation extensible dans lequel un programme de référence peut être étendu en ajoutant des aspects pour certains usages ;

- De l'approche MDA, nous retiendrons un principe important : il faut réfléchir et travailler à un niveau d'abstraction plus élevé que le code. Il s'agit des modèles, qui permettent de définir les caractéristiques des applications de façon indépendante d'une implémentation particulière. Le fait de travailler avec des modèles métiers nous évitera d'être dépendant d'une technologie et d'une plate-forme particulière ;

- De l'approche EAI, nous pouvons retenir quelques techniques et patrons de construction des adaptateurs pour des applications existantes ;

- De l'approche BPM, nous retiendrons un mécanisme de composition par orchestration qui semble très intéressant. Toutefois, comme [AFGK02] l'a souligné, la plupart des cas d'évolution des applications globales impliquent des changements de la collaboration entre les participants. Il faut séparer la logique de l'interaction des participants et la traiter explicitement.

  Nous retiendrons aussi l'idée d'utiliser un *workflow* pour définir et d'exécuter un modèle de procédé représentant la logique de l'interaction des participants. Cependant, il faut nécessaire d'avoir des améliorations afin d'augmenter la flexibilité des WfMSs actuels.

Nous souhaitons pouvoir utiliser les points forts de chaque approche pour atteindre notre objectif. Cependant, nous pensons qu'il est important de disposer un cadre conceptuel dans lequel ces points forts peuvent être exploités.

Nous passons maintenant à la deuxième partie de ce document. Dans cette partie, nous tentons d'identifier une architecture logicielle qui s'applique dans un contexte de composition général et qui propose des solutions aux points ouverts des approches étudiées dans ce chapitre. De plus, nous essayons d'exploiter, de manière raisonnable, les points positifs des approches actuelles.

Chapitre III

Composition d'éléments logiciels par coordination

1. Introduction

Dans le chapitre précédent, nous avons étudié les différents mécanismes de composition qui sont utilisés dans les récentes approches pour la construction d'applications. Ceci nous permet à la fois de caractériser les éléments à composer et d'avoir un aperçu sur les différentes manières de les assembler :

- La programmation à base de composants compose les composants, appartenant au même modèle, en établissant des connexions entre eux ;
- Les conteneurs coordonnent les fonctionnalités d'un composant avec les services non-fonctionnels. Cependant, cette coordination est fixée et pré-calculée en considérant toutes les combinaisons possibles des services non-fonctionnels;
- L'approche BPM utilise un WfMS pour piloter et orchestrer explicitement un ensemble de participants, surtout des services Web ;
- L'approche EAI intègre les applications entreprises en s'appuyant sur des adaptateurs qui lient ces applications ;
- L'approche AOP intègre les aspects au sein d'un programme de référence en se basant sur un mécanisme de tissage ;
- L'approche MDA remonte le niveau d'abstraction en se concentrant sur les modèles ; la composition d'applications est dirigée par la composition de modèles.

Maintenant, nous nous concentrons sur le sujet central de cette thèse : la composition d'éléments logiciels, de diverses natures, basée sur la coordination et dirigée par la composition de modèles exécutables.

Ce sujet sera étudié tout au long de la deuxième partie de cette thèse. Cette deuxième partie, qui prend en compte les qualités et les spécificités des mécanismes de composition ci-dessus, est décomposée en trois chapitres.

Le chapitre 3 étudie la fédération comme étant une architecture logicielle ouverte et dynamique qui permet de composer, par la coordination, des éléments logiciels de diverses natures dans l'objectif de réaliser ensemble une application.

La chapitre 4 généralise la solution précédente en introduisant le concept de domaine et la composition de domaines. Du point de vue structurel, un domaine est un groupe d'éléments logiciels qui, par leur coordination, proposent une fonctionnalité complexe pour une préoccupation précise. Du point de vue conceptuel, un domaine est représenté par un modèle qui est une matérialisation de son espace conceptuel. La composition de domaines nous permet de réutiliser non seulement un élément logiciel mais aussi un groupe d'éléments avec une fonctionnalité de haut niveau.

Le chapitre 5 présente une façon de découpler la partie générique de la partie spécifique d'un domaine. Ceci nous emmène au concept de domaine générique qui peut être utilisé dans plusieurs contextes d'utilisation différents.

2. Le problème à résoudre

Ce chapitre a pour objectif d'étudier une architecture logicielle qui permette de construire des applications par la composition d'éléments logiciels de diverses natures. Nous nous intéresserons plus spécialement aux éléments logiciels existants afin de réutiliser au mieux les fonctionnalités que proposent ces éléments.

Les approches EAI et BPM, que nous avons étudiées dans le chapitre 2, s'intéressent également à la composition des éléments logiciels existants. Cependant, il reste plusieurs points pour lesquels ces approches ne proposent pas de solution adéquate tels que :

- Une vue uniforme et de haut niveau d'abstraction des éléments logiciels. En effet, les adaptateurs, utilisés par l'approche EAI, ne représentent qu'un moyen technique pour ajuster les éléments logiciels, mais ils ne proposent pas une vue de haut niveau d'abstraction de ces éléments ;
- Le partage et l'hétérogénéité des concepts partagés par plusieurs éléments logiciels. En réalité, l'hétérogénéité peut être liée à la syntaxe, la structure et la sémantique que chaque élément logiciel utilise pour décrire les concepts partagés ;
- L'autonomie et l'initiative des éléments à composer. En réalité, certains éléments logiciels sont conçus et construits pour être utilisés indépendamment, ils peuvent changer d'état en réponse aux changements de l'environnement (tels que les applications interactives). Or, puisqu'ils peuvent partager des concepts, le fait de changer l'état d'un concept partagé implique que les autres éléments doivent se synchroniser ;
- L'évolution et l'adaptation de l'application résultante.

Ce chapitre tente d'apporter une solution adéquate à ces problèmes. En sachant que ces problèmes sont complexes, nous allons les étudier séparément en considérant les points positifs des mécanismes de composition étudiés dans le chapitre 2. Le fait de mettre ensemble les solutions pour tous ces problèmes va nous amener à définir notre architecture logicielle. Nous appellerons cette architecture une ***fédération***.

Étant donné que les éléments à composer sont de diverses natures, nous proposons d'utiliser le terme neutre ***participant*** pour parler des éléments homogènes et « conscients »

de notre approche de composition. Un participant peut être « conscient » de deux manières : (1) il est écrit pour être utilisé dans une fédération, (2) il est un élément logiciel existant qui a été adapté pour être utilisé dans une fédération. Dans le dernier cas, nous utilisons le terme ***exécutant*** pour désigner un élément logiciel hétérogène qui doit être adapté pour être un participant.

Le reste de ce chapitre est structuré comme suit. La section 3 présente la coordination comme une manière de composer en préservant l'autonomie et l'initiative des participants. La section 4 étudie le problème de partage de concepts entre les participants et la gestion de la cohérence et l'hétérogénéité des concepts partagés. Le concept d'univers commun sera introduit comme étant un espace partagé avec lequel les participants se coordonnent. La section 5 discute une manière de contrôler explicitement la coordination des participants avec l'univers commun. La section 6 étudie l'adaptation des exécutants pour devenir des participants. La section 7 fait un résumé sur notre architecture de fédération. La section 8 résume et discute ce chapitre.

3. Composition par coordination

3.1. Objectifs

Les premiers objectifs de notre approche de composition sont de :

- Préserver l'autonomie et l'initiative des participants. Ceci va nous aider à travailler aisément avec les participants autonomes et interactifs ;
- Faciliter l'évolution de l'application globale résultante de la composition.

Pour atteindre ces objectifs, il est nécessaire d'avoir une approche de composition dans laquelle il n'y a pas de connexions directes entre les participants. Par conséquent, le couplage entre les participants sera minimisé ; les participants seront indépendants les uns des autres.

3.2. La solution proposée

La solution que nous proposons consiste à considérer la coordination comme moyen de composition. Pour cela, il y a un **coordinateur** dont la tâche est d'aider les participants à travailler ensemble afin de réaliser ensemble l'application globale.

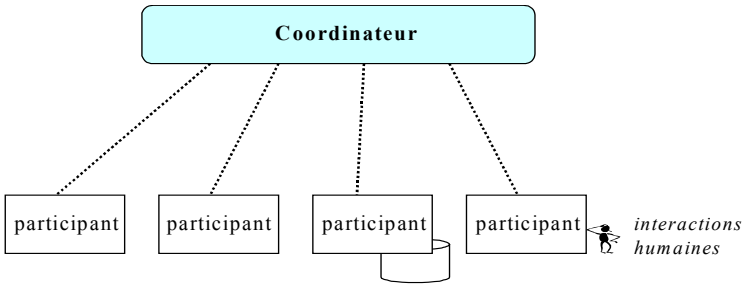

Figure 30. Coordination comme un moyen de composition

Le coordinateur connaît tous les participants. Il les aide à travailler ensemble en assurant, à tout moment, la cohérence des concepts partagés entre eux. Les participants n'ont pas besoin de se connaître entre eux. Cependant, ils doivent connaître le coordinateur et « collaborer » avec le coordinateur afin de l'aider à accomplir sa tâche.

Comment les participants collaborent-ils avec le coordinateur ?

Pour répondre à cette question, nous proposons de distinguer trois types de participant :

- Les participants **actifs** qui peuvent changer leur état interne par leur propre initiative. Les applications interactives sont un exemple typique de participants actifs ;

- Les participants **réactifs** qui, tout seul, ne changent pas leur état interne. Cependant, ils ont éventuellement des réactions additionnelles suite à une demande de changer leur état provenant du coordinateur ;

- Les participants **passifs** qui n'ont ni de la capacité de changer leur état interne, ni de réactions additionnelles suite à une demande du coordinateur. Autrement dit, ils réagissent toujours comme demandés par le coordinateur.

Par exemple, supposons que nous réalisons une application bancaire en s'appuyant sur des participants existants qui ne se connaissent pas. Parmi ces participants, nous pouvons trouver les applications destinées à la clientèle (e.g. une application pour les guichets) qui sont les participants actifs. Des monitorings (i.e. les outils qui tracent l'exécution de l'application bancaire) sont les participants passifs. En revanche, l'application qui traite des opérations métiers (e.g. un manager) est un participant réactif. La Figure 31 montre la coordination de ces participants à l'aide du coordinateur.

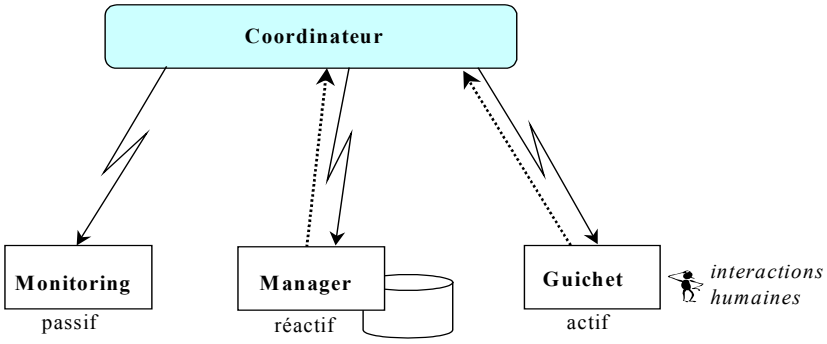

Figure 31. Participants actifs, réactifs et passifs dans l'application bancaire

Pour les participants actifs et réactifs, il est indispensable que leurs changements d'état interne, causés par leur propre initiative ou par leurs réactions, soient notifiés au coordinateur. Face à ces notifications, le coordinateur peut demander aux autres participants de réagir (i.e. de mettre à jour leur état) afin de garder la cohérence de l'ensemble.

De cette manière, les participants actifs ou réactifs sont des participants qui ont des influences sur les autres participants, mais au travers du coordinateur. En conséquence, il

est possible de déléguer la gestion du but, que l'ensemble de participants va réaliser, à ces participants. Dans ce cas, ils peuvent être considérés comme étant des **contrôleurs du but** de l'ensemble de participants.

La manière qu'ont les participants de notifier leurs changements au coordinateur, et la façon qu'a le coordinateur de demander aux participants de réagir, dépend l'implémentation du coordinateur. Dans le cas le plus simple, il peut s'agir d'événements comme dans la Figure 31.

Notre démarche ressemble à l'approche BPM (*Business Process Management*), dans l'idée d'extraire la coordination entre les participants afin de pouvoir la gérer explicitement. Cependant, ce qui est différent avec l'approche BPM[4], est que nous avons séparé clairement le contrôleur du but (i.e. celui qui impose l'objectif global à l'ensemble de participants) et le coordinateur (i.e. celui qui est chargé de garder la synchronisation entre des participants). Ceci nous apporte plusieurs avantages :

- Un contrôleur du but guide les participants à travers le coordinateur et par conséquent, il ne doit pas être adapté pour pouvoir travailler avec les participants comme dans le cas de BPM (cf. Figure 32) ;

  Dans le BPM, un WfMS (*Workflow Management System*) joue le rôle du contrôleur du but. Ce dernier doit donc connaître les participants afin de pouvoir les piloter. Ceci implique un fort couplage entre le contrôleur du but et les participants, en conséquence, les difficultés de faire évoluer et adapter l'application globale ;

- Les participants actifs peuvent être gérés aisément.

  L'approche BPM ne permet pas de considérer les participants bien qu'il y ait souvent des participants actifs dans un procédé d'entreprise. Ce sont les participants ayant des interactions avec les clients ;

- Il est simple et naturel de faire cohabiter plusieurs contrôleurs de but qui, eux aussi, se coordonnent avec l'aide du coordinateur. De cette manière, il n'est pas obligatoire qu'ils se connaissent et se basent sur un même standard afin de pouvoir travailler ensemble

  Dans l'approche BPM, la cohabitation des contrôleurs peut être faite, mais avec beaucoup de difficultés : le procédé global est divisé en sous-procédés dont chacun est exécuté par un WfMS. Ces WfMSs gèrent eux-même toutes les coordinations entre eux ; ils doivent ainsi se connaître et se baser sur un standard pour pouvoir se coordonner.

Graphiquement, ces différences peuvent être résumées dans la Figure 32.

[4] Le contrôle explicite dans le BPM est à la fois le contrôleur du but et le coordinateur.

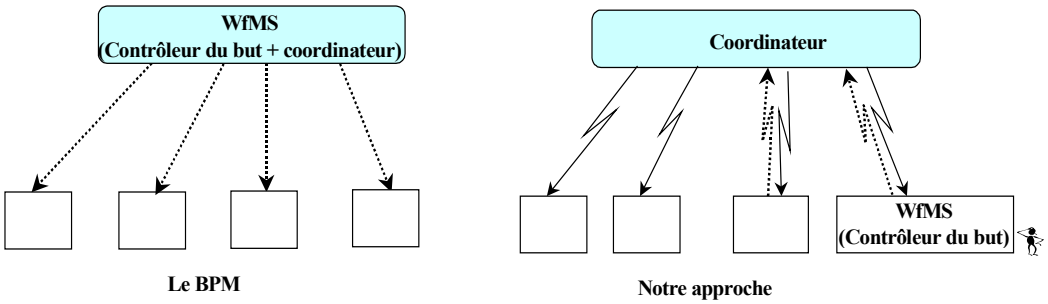

Figure 32. Le BPM et notre approche

Comment construit-on le coordinateur ? Quels sont ses éléments ? Nous répondons à ces questions dans la section suivante en considérant d'autres problèmes que nous avons cités au début de ce chapitre.

4. Matérialisation des concepts partagés

4.1. Objectifs

Le sujet que nous voulons discuter dans cette section est le partage des concepts entre les participants. Lié à ce sujet, plusieurs problèmes doivent être considérés. Il s'agit de la gestion de l'hétérogénéité (aux trois niveaux : syntaxique, structurel et sémantique), et de la cohérence entre les participants en ce qui concerne les concepts partagés.

La question de fond qu'il faut discuter est de savoir qui est chargé de connaître et de gérer la cohérence des concepts partagés entre les participants ? Il est certain que ceci ne peut pas être à la charge des participants.

4.2. Univers Commun

La solution que nous proposons consiste à : (1) matérialiser explicitement les concepts partagés entre les participants dans un format compréhensible par tous et (2) passer au coordinateur la responsabilité de maintenir la cohérence de ces concepts.

Pour ce faire, nous proposons de réserver un espace dans le coordinateur comme étant un espace partagé et commun, contenant une matérialisation explicite des concepts partagés entre les participants. Ceci veut dire que cet espace maintient l'état des concepts partagés entre les participants. Nous appelons cet espace l'univers commun (UC).

Concrètement, lors de définition de l'UC, on va trouver une classe CC pour représenter un concept partagé entre au moins deux participants. Toutefois, ces participants peuvent représenter et utiliser, à leur manière, ce concept partagé. Pour simplifier, on suppose qu'ils le représentent respectivement par deux classes C1 et C2. La classe CC doit contenir suffisamment d'informations pour permettre d'exprimer la manière de gérer la cohérence entre C1 et C2.

Lors de l'exécution, on peut trouver dans l'UC un objet `ObjC`, qui est une instance de la classe CC pour chaque couple d'instances `Obj1`, `Obj2`, qui sont respectivement des instances de C1 et C2. L'instance `ObjC` est utilisée pour synchroniser les instances `Obj1`, `Obj2`. Ceci est illustré graphiquement dans la Figure 33.

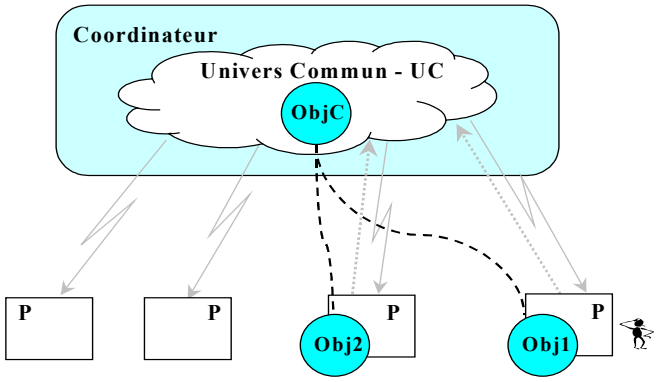

Figure 33. Synchronisation par l'univers commun

Désormais, la coordination entre les participants est traduite par la synchronisation de l'état de leurs instances avec l'instance du concept partagé dans l'UC (i.e. l'instance `ObjC`) et vice-versa. Ceci peut être résumé comme suit :

- Les participants actifs et réactifs notifient à l'UC leurs changements liés aux concepts partagés Cette notification a pour objectif de demander à l'UC de synchroniser son état avec celui des participants. Elle peut être faite par un appel de méthode de l'UC ou bien par un événement ;

- Les participants peuvent souscrire aux changements d'état des instances des concepts partagés de l'UC. Lorsqu'il y a un changement de ces instances, une notification sera envoyée à ces participants. En réponse à ces notifications, les participants peuvent se synchroniser avec l'UC.

Revenons à l'application bancaire présentée auparavant. L'univers commun de cette application contient les concepts partagés entre les participants tels que les concepts de compte, de titulaires de compte, etc. Lors de l'exécution de cette application, on trouvera les instances de ces concepts dans l'UC. L'état de ces instances représente le moyen fondamental pour coordonner les participants `'monitoring'`, `'manager'` et `'guichet'`.

La façon ci-dessus dont les participants se coordonnent autour de l'UC constitue le paradigme de coordination « **anarchique** » dans lequel il n'existe aucun contrôle, aucune règle ; les participants peuvent librement souscrire aux notifications émises par l'UC et réagir à ces notifications. Ils peuvent, en plus, changer l'état des instances de l'UC bien que ceci puisse provoquer des conséquences dans leur propre état et dans celui des autres. Par conséquent, le paradigme anarchique peut être utilisé dans des situations simples et à condition que les participants réagissent de manière « correcte » dans l'objectif de réaliser collectivement l'application globale.

5. Contrôle de la coordination

5.1. Objectifs

Au cours de nos expérimentations, nous avons trouvé plusieurs situations dans lesquelles le paradigme « anarchique » ne permet pas d'obtenir un résultat cohérent et correct [Her00, EL01, EVC01, Le02] :

- Les participants ayant des fonctionnalités similaires essaient de faire la même choses simultanément ou bien des choses contradictoires sur l'UC. Par conséquent, les laisser faire ce qu'ils veulent peut impliquer l'incohérence de l'UC. Ainsi, il est important de préciser ce que chaque participant pourrait et devrait faire sur l'UC;
- Un changement de l'UC peut impliquer la réaction de plusieurs participants à la fois. S'il existe une relation entre ces réactions, il est nécessaire d'ajouter les contraintes liées à la manière d'accomplir ces réactions. Des exemples de contraintes peuvent être des contraints temporelles, d'ordonnancement, transactionnelles, etc.. Par exemple, dans une application bancaire, on peut imaginer que l'opération `transfert()`est divisée en deux sous-opérations `créditer()` et `débiter()` dont chacune est réalisée par un participant `'serveur'`. Il est clair que des contraintes de type transactionnel doivent être considérées entre ces participants ;
- Un changement de l'UC peut impliquer la réaction d'un seul participant, mais cette réaction doit être faite de manière synchrone par rapport à l'exécution de l'UC. Il est clair que dans ce cas, on ne peut pas laisser ce participant réagir librement.

Pour tous ces problèmes, il est nécessaire d'avoir un moyen visant à contrôler explicitement la manière dont les participants se coordonnent autour de l'UC. Ceci nous aide à garantir la cohérence entre l'UC et l'ensemble de participants.

5.2. Autorité commune

Nous appelons l'**autorité commune** l'organe qui fait partie du coordinateur et qui a pour but de contrôler explicitement la synchronisation des participants avec l'univers commun.

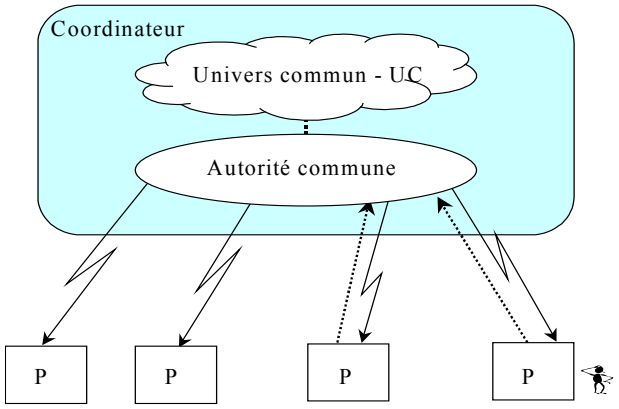

Figure 34. Univers commun contrôlé par l'autorité commune

Nous distinguons deux facettes de contrôle que l'autorité commune assure :

- Contrôle des accès à l'univers commun (cf. 5.3) : Ce contrôle détermine ce que chaque participant peut faire sur l'univers commun ;
- Contrôle de la coordination (cf. 5.4) : Ce contrôle définit la manière dont les participants réagissent lors d'un changement de l'univers commun.

Dans le cas où les participants sont fortement contrôlés, c'est l'autorité commune qui décide ce que chaque participant peut faire et ce que les participants doivent faire afin de réaliser l'application globale. Ce paradigme de coordination est appelé le **paradigme dictatorial**.

De cette manière, nous disposons d'une palette de paradigme de coordination qui s'étend depuis le paradigme anarchique jusqu'au paradigme dictatorial. Le paradigme mixte est une combinaison des paradigmes anarchiques et dictatoriaux. Avec ce paradigme, l'autorité commune ne sera sollicitée que dans les situations où il est indispensable de contrôler les participants afin d'obtenir un résultat cohérent. Dans les autres situations, on peut laisser les participants agir librement.

Le choix d'un paradigme de coordination dépend de chaque application globale à construire.

5.3. Contrôle des accès à l'univers commun

Pour contrôler des accès à l'UC, nous proposons d'ajouter le concept de **droit** pour désigner ce qu'un participant peut faire sur l'univers commun [Her00, Le02]. Nous définissons plusieurs types de droits :

- Le droit de créer et de détruire une instance de l'UC ;
- Le droit d'effectuer un changement dans l'UC en appelant une méthode de l'UC ;
- Le droit de recevoir une notification concernant un changement de l'UC ;
- Le droit de recevoir une notification concernant la création/destruction d'une instance de l'UC ;

Lors de la définition de chaque participant, nous lui associons les droits pour spécifier ce qu'il peut faire sur l'UC. En réalité, les droits font partie du contrat fonctionnel que les participants doivent respecter afin de pouvoir participer à l'application globale. Nous reviendrons sur ce sujet dans la partie 6.3.

Quant à l'autorité commune, elle vérifie tous les accès à l'UC afin de ne laisser faire que les participants qui en ont le droit.

5.4. Contrôle de la coordination

Le contrôle de la coordination consiste à définir explicitement la manière dont les participants réagissent lors d'un changement de l'UC. Ceci correspond à définir les

réactions des participants et les contraintes qui sont éventuellement liées à l'ensemble de ces réactions.

Concrètement, lors d'un changement de l'UC, l'autorité commune doit :

- Sélectionner le participant le plus adéquat. Ceci est particulièrement important lorsque plusieurs participants peuvent fournir des fonctionnalités similaires ;
- Imposer explicitement ce que chaque participant sélectionné doit faire en considérant des contraintes, telles que des contraintes d'ordonnancement, temporelles, de synchronisation par rapport à l'UC, etc., afin d'obtenir un résultat cohérent.

Pour ce faire, il est indispensable que l'autorité commune connaisse les réactions potentielles de chaque participant. La façon la plus simple est d'exprimer les réactions potentielles d'un participant sous forme des méthodes offertes par ce participant. Ces méthodes représentent la contribution du participant à la réalisation de l'application globale (celles-ci seront exprimées grâce au concept de rôle dans la partie 6.3).

Nous proposons d'exprimer explicitement la coordination des participants lors d'un changement de l'UC sous forme d'un **contrat de coordination.** Ceci correspond au fait qu'une partie de l'autorité commune sera exprimée par un ensemble de contrats de coordination. Lors de son exécution, un contrat va guider la coordination des participants en imposant explicitement ce que chaque participant doit faire et en respectant les contraintes de coordination, les conditions d'exécution, etc.

5.4.1. Contrat de coordination

En général, le terme contrat est utilisé pour désigner « *un accord de volonté entre deux ou plusieurs participants ayant pour effet de créer une ou plusieurs obligations* » [DIC].

Dans notre cas, nous définissons un contrat de coordination comme étant un accord entre plusieurs participants ayant pour effet de réagir, de manière cohérent, à un changement produit dans l'UC.

Un contrat de coordination a les caractéristiques suivantes :

- Un contrat est associé à un changement de l'UC (i.e. l'appel d'une méthode de l'UC), ce qui est considéré comme étant la condition de déclenchement du contrat ;
- Un contrat n'est pas binaire : plusieurs participants peuvent être sollicités pour participer au contrat ;
- Un contrat a un contexte d'exécution qui regroupe toutes les informations de l'UC qui sont liées au contrat et qui sont déterminées lors du déclenchement du contrat ;
- Un contrat est dynamique (i.e. il peut s'adapter selon le contexte d'exécution). L'ensemble de participants qui participent au contrat peut être déterminé en fonction de la disponibilité des participants. De plus, un contrat peut référencer et sélectionner les participants en fonction de leurs caractéristiques non-fonctionnelles.

- L'exécution d'un contrat peut échouer. L'échec d'un contrat de coordination peut être causé par le fait qu'un participant n'arrive pas à accomplir sa partie du contrat ou peut être causé par un problème matériel (e.g. réseau, etc.). L'échec de l'exécution d'un contrat ne signifie pas nécessairement une erreur, mais simplement que l'ensemble des participants ne peuvent pas se coordonner correctement. Ceci doit être notifié au coordinateur afin qu'il puisse prendre les décisions qui s'imposent.

En réalité, en cas d'échec d'un contrat, la réaction du coordinateur peut être de défaire les parties du contrat exécutées, mais aussi et surtout de défaire les actions dans l'univers commun qui ont déclenché le contrat. La réaction à l'échec d'un contrat est un thème de recherche en soi, non abordé dans le cadre de cette thèse.

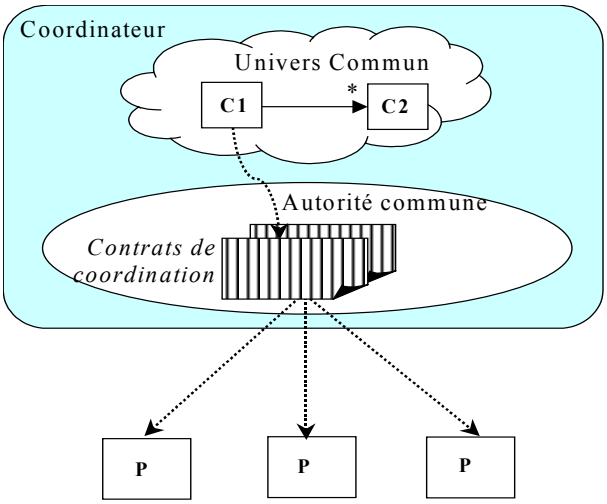

Figure 35. Contrats de coordination

Dans notre implémentation, l'exécution de l'UC doit être instrumentée (i.e. interceptée) dans l'objectif de pouvoir ajouter dynamiquement, lors de l'exécution, des contrats de coordination aux bonnes méthodes de l'UC. Ceci ressemble au mécanisme d'intégration dynamique de l'AOP. Nous reviendrons sur notre implémentation dans le chapitre 6.

5.4.2. Modèle de contrat de coordination

Nous avons construit un modèle destiné à définir des contrats de coordination en tenant compte de leurs caractéristiques citées auparavant. Dans ce modèle, un contrat est défini par :

- Un nom du contrat ;
- La classe de l'UC et la méthode de cette classe à laquelle le contrat est associé ;
- Un ordre d'exécution du contrat. Ceci explique l'exécution du contrat par rapport à celle de la méthode de l'UC à laquelle est attaché le contrat. En s'inspirant le principe d'AspectJ, nous considérons que les contrats peuvent être exécutés avant, après et à la place des méthodes de l'UC ;

- Une liste d'arguments destinés à paramétrer le contrat. Dans l'implémentation actuelle, un contrat reçoit : (1) les paramètres qui sont ceux de la méthode de l'UC à laquelle il est attaché, (2) un objet représentant l'instance de la classe de l'UC, (3) un objet représentant le contexte d'exécution du contrat. Nous reviendrons sur ceux-ci dans le chapitre 6 ;
- Une réponse qui représente le résultat de l'exécution du contrat ;
- Un ensemble de participants concernés par le contrat ;
- Un corps de contrat qui exprime la coordination des instances des participants au sein du contrat. Notons que ce sont des instances de participants qui participent réellement au contrat lors de son exécution.

Le corps du contrat est la partie centrale d'un contrat de coordination. Il comprend deux parties principales :

- La localisation des instances des participants. Celle-ci nous permet de trouver les instances des participants ayant les caractéristiques non-fonctionnelles souhaitées telles que la machine d'exécution, le mode de connexion, etc. Notons que l'application globale peut être totalement distribuée ; les instances des participants peuvent être exécutées sur plusieurs machines clientes différentes ;
- La coordination entre des instances des participants. Celle-ci est exprimée en terme de séquence d'opérations : les appels des méthodes offertes par les participants et les opérations assurées par le contrat lui-même. Ces dernières peuvent être considérées comme une préparation avant de passer le contrôle aux instances de participants.

Nous avons construit des éditeurs graphiques permettant de spécifier les caractéristiques des contrats de coordination en suivant le modèle ci-dessus. La plupart des caractéristiques sont définies à travers des formulaires offerts par nos éditeurs. En revanche, le corps de contrat peut être défini par un langage textuel simple, basé sur `java`. Dans le futur, il est envisageable d'utiliser un langage spécifique de coordination, tel que CLF (*Coordination Language Facility*) [APR00], pour définir le corps de contrat.

Revenons à notre exemple d'application bancaire, la méthode `addCompte()` de la classe `Personne` est chargée de faire évoluer des objets de l'UC. Elle est écrite comme suit :

```
addCompte ( String NoCompte, int type ){
  //réifier un objet de compte dans l'UC
  Compte newCompte = new Compte ( NoCompte, type) ;
  listComptes.add(newCompte) ;
  ….
}
```

Figure 36. Méthode `addCompte()` de la classe `Personne` de l'UC

La Figure 37 montre le corps du contrat `addCompte` attaché à la méthode `addCompte()` de la classe `Personne` ci-dessus. Ce contrat est de type `'AROUND'` (i.e. il va s'exécuter à la place de la méthode `addCompte()` de l'UC). L'information concernant le type du

contrat est définie dans le formulaire proposé par l'éditeur de contrats. Nous reviendrons sur cet éditeur dans l'annexe A.

```
contrat addCompte( String NoCompte, int type )of Personne
// déclarer les paramètres à passer au contrat
{
      // "instance" représente l'objet qui a déclenché le contrat,
      // contrat à exécuter seulement si le nombre de compte est inférieur à 5
      when instance.getCompteCount() <= 5

      //Localiser des instances de participants
      participants
      {
            managerComponent: managerRole at SERVER
            guichetComponent: guichetRole at ctxt.getAttribute("host");
      }
      body( JAVA )
      {
            //saisir l'information concernant le nouveau compte
            String[] infor = guichetComponent.newCompte();

            //Exécuter la méthode de l'UC, addCompte, récupérer son résultat
            if ( proceed ( infor[0], infor[1] )) {
            // synchroniser avec le participant managerComponent
                managerComponent.addCompte(infor[0], infor[1], instance.getName() );
            }
    }
}
```

Figure 37. Le contrat `addCompte`

Dans ce contrat, il y a deux participants : `managerRole` qui représente le `'manager'` gérant la base de données et `guichetRole` qui représente l'application `'guichet'` qui propose des formulaires pour saisir les informations concernant le nouveau compte (voir la Figure 31). Les instances de ces participants sont localisées respectivement sur la machine centrale (i.e. la machine serveur) et sur la machine distante, qui a provoqué l'exécution du contrat, dont le nom est trouvé dans le contexte d'exécution du contrat (i.e., *ctxt.getAttribute("host")*). Lorsque les informations sont saisies, ce contrat invoque explicitement la méthode *addCompte* de l'UC, puis il synchronise le `'manager'` si la méthode de l'UC est bien exécutée.

Puisque les contrats sont exprimés dans un langage déclaratif, il est simple de les écrire et modifier. Grâce aux contrats de coordination, il est aisé de :

- Sélectionner le participant le plus adéquat pour réaliser une fonctionnalité. Ceci peut être effectué en sélectionnant un participant convenable ou une instance d'un participant avec les caractéristiques non-fonctionnelles souhaitées ;
- Contrôler explicitement la coordination entre les participants afin d'obtenir un résultat correct et cohérent. Les contraintes liées à l'exécution des contrats tels que les contraintes d'ordonnancement, de synchronisation, etc. sont assez simples à exprimer avec nos contrats de coordination.

Nous venons de présenter notre approche de composition en faisant l'hypothèse que les participants sont homogènes et conscients de notre manière de composer. Nous étudions maintenant la manière de représenter et de « transformer » un exécutant quelconque en un

participant. Ceci nous permet de nous placer dans un contexte général où les éléments à composer, appelés des **exécutants**, sont de nature quelconque.

6. Un exécutant vu comme un participant

6.1. Le problème

Pour faire « participer » des exécutants de nature quelconque, il est indispensable de disposer de moyens permettant de :

- Exprimer la capacité fonctionnelle des exécutants en considérant : (1) les méthodes offertes à l'application globale, (2) la capacité de changer leur état interne de leur propre initiative ou leurs réactions et (3) les droits d'accès à l'univers commun. Pour cela, nous avons introduit le concept de *rôle* (cf. section 6.3) ;
- Exprimer les propriétés non-fonctionnelles des exécutants (cf. section 6.4) ;
- Adapter un exécutant pour qu'il puisse implémenter les rôles qu'il joue dans l'application globale, mais sans avoir à changer son code source. Pour cela, nous avons introduit le concept d'*adaptateur* (cf. section 6.5). Avec ces concepts, un participant est un triplet : (rôles, adaptateur, exécutant) ;
- Gérer le *cycle de vie* des instances des participants. Ceci implique la capacité de créer et de détruire une instance des exécutants et de leur adaptateur (cf. section 6.6).

Nous allons étudier ces divers aspects.

6.2. Le concept d'exécutant

Dans cette section, nous essayons de donner quelques critères pour caractériser des exécutants. Ceci nous aide également à classifier les participants selon leurs caractéristiques afin de mieux les gérer.

6.2.1. Application versus outil

En se basant sur la nécessité d'utiliser des données (e.g. un modèle, un fichier, etc.) pour initialiser l'état d'un exécutant, on distingue les exécutants de type **application** et les exécutants de type **outil**.

Pour nous, une application n'a pas d'état, ou un état qu'il n'est pas possible de modifier depuis l'extérieur. Autrement dit, une application a un comportement et un état interne spécifique.

Par contre, l'état des participants du type outil est géré de façon externe. Par conséquent, il est possible d'intervenir sur leur état.

L'état initial d'un outil est paramétré par un modèle (i.e. une donnée). A titre d'exemple, on peut citer un système de gestion de base de données tels que *Microsoft Access*. Il est clair que pour travailler avec *Microsoft Access*, on doit choisir un fichier *'.mdb'* afin de l'initialiser.

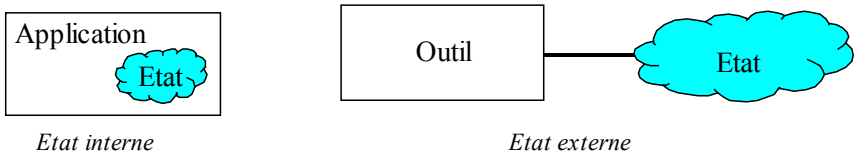

Figure 38. Application versus outil

Il est nécessaire d'initialiser l'état des exécutants du type outil. Les données utilisées peuvent être disponibles dans l'application globale, mais sous un format différent. Une conversion des données est souvent nécessaire. Ceci sera à la charge de l'adaptateur lié à l'exécutant (cf. section 6.5).

6.2.2. Local versus distant

La distribution des exécutants nous permet de distinguer les exécutants **locaux** et les exécutants **distants**.

Les exécutants locaux s'exécutent sur la machine centrale de notre environnement (i.e. la machine serveur). Ce type d'exécutant ne peut avoir qu'une seule instance.

En revanche, les exécutants distants peuvent s'exécuter sur la machine centrale et aussi sur des machines clientes distantes. De cette manière, un exécutant distant peut avoir plusieurs instances qui s'exécutent simultanément sur plusieurs machines clientes différentes.

Il est important de noter qu'avec ces deux types d'exécutants, on est capable de définir toutes les applications, même des applications complexes. Par exemple, une application de type client-serveur peut être déclarée comme étant deux exécutants :

- Un exécutant local, avec une seule instance, représentant l'application serveur ;
- Un exécutant distant, avec plusieurs instances sur des machines différentes, représentant l'application cliente.

Le fait de déclarer une application client-serveur comme ci-dessus a plusieurs avantages : le cycle de vie de l'application serveur et cliente, sera complètement géré par notre plate-forme. La communication entre le serveur et les clients peut être effectuée directement ou via notre système de communication.

Par conséquent, nous avons pris la distribution comme le critère principal pour classifier les exécutants. D'une part, comme la distribution est une propriété difficile à gérer, nous préférons faciliter la gestion de celle-ci. D'autre part, le fait qu'un exécutant soit un outil ou une application n'est pas très important pour notre plate-forme de support, mais surtout pour la réalisation de son adaptateur. Comme la réalisation des adaptateurs est spécifique

pour chaque exécutant, nous préférons ne pas considérer cette propriété dans notre modèle de participant.

6.3. Modèle fonctionnel – le concept de rôle

Dans notre modèle de participants, nous avons utilisé le concept de rôle pour représenter une abstraction de la capacité fonctionnelle des exécutants.

Du point de vue de l'application globale, un rôle permet de définir ce qu'on attend d'un exécutant jouant ce rôle. Autrement dit, un rôle représente le contrat fonctionnel qu'un exécutant jouant ce rôle doit respecter. Un exécutant peut jouer un ou plusieurs rôles. Pour chaque rôle joué, il doit respecter le contrat fonctionnel définit par ce rôle.

Le concept de rôle permet d'introduire une couche intermédiaire entre l'univers commun et l'ensemble des exécutants. Par conséquent, le couplage entre l'univers commun et les exécutants sera ainsi minimisé ; les contrats de coordination s'appuient sur les rôles, et non pas sur les exécutants. L'évolution de l'application globale sera plus aisée. Le fait de remplacer un exécutant par un autre peut être fait facilement, à condition qu'il puisse jouer le même rôle.

Le concept de rôle permet également de regrouper des exécutants « similaires » (i.e. des exécutants qui sont capables de répondre à un même contrat fonctionnel). En fonction du contexte d'exécution, on peut choisir l'exécutant le plus adéquat.

Par exemple, dans un prototype antérieur, nous avions le rôle '*gestion de version*' qui était assuré par deux outils : RCS (*Revision Control System*) [RCS] et CVS (*Concurrent Version System*) [CVS]. Lorsqu'un document à versionner contient un seul fichier, on utilise l'outil RCS. Par contre, il faut utiliser l'outil CVS pour des documents composés de plusieurs fichiers.

6.3.1. Modèle de rôle

Dans notre modèle, un rôle est un tuple <Nom, IF, N, D, E> avec :

- **Nom** : un nom symbolique du rôle ;
- **IF** : une interface qui regroupe les méthodes fournies par le rôle. Le couple (nom, interface) permet de rapprocher du concept de facette dans les modèles de composants ;
- **N** : le nombre de participants pouvant jouer ce rôle. Dans une application globale, un rôle peut être joué par un ou plusieurs exécutants à la fois. C'est pour cette raison que nous avons distinguée des rôles simples des rôles multiples (cf. section 6.3.2) ;
- **D** : le droit du rôle. Ceci comprend le droit d'appeler des méthodes de l'UC et d'écouter les événements émis par l'UC ;
- **E** : les événements émis par le rôle dus à son initiative. Ceci n'est important que pour les participants actifs ou réactifs.

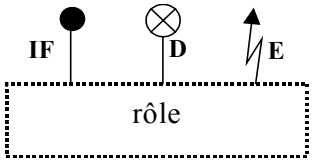

Figure 39. Un rôle

Il est important de noter que nous ne considérons pas les interfaces requises par des rôles comme dans les modèles de composant. Ceci, parce que nous sommes basés sur l'hypothèse que les rôles (et donc les exécutants) sont autonomes (i.e. ils peuvent s'exécuter indépendamment les uns des autres). Des connexions directes entre les participants sont permises, mais elles ne sont pas gérées par notre plate-forme bien que celle-ci propose des services pour la gestion de cycle de vie et la localisation des exécutants. Dans ce cas, il est à la charge des exécutants de gérer les connexions directes entre eux.

6.3.2. Type de rôle

Nous avons distingué deux types de rôles dans notre modèle : (1) rôle simple et (2) rôle multiple.

- Un rôle simple est un rôle qu'un seul exécutant peut jouer à un moment donné. Ainsi, il y a une projection directe entre le rôle et l'exécutant jouant ce rôle.

  Même si que plusieurs exécutants peuvent jouer ce rôle, un seul sera sélectionné pour une configuration d'exécution de l'application globale. La configuration d'exécution d'une application globale est définie lors de la phase de conception. Nous reviendrons sur ce sujet dans l'annexe A où les éditeurs de notre environnement de conception seront présentés.

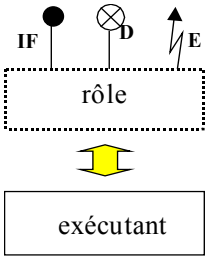

Figure 40. Un rôle simple

- Un rôle multiple est un rôle qui peut être joué, en même temps, par plusieurs exécutants. Par conséquent, il est nécessaire d'avoir une fonction permettant de sélectionner l'exécutant adéquat à utiliser. Cette fonction peut se baser sur le contexte d'exécution ou bien un ensemble de critères quelconques, spécifié lors de la phase de définition de rôle.

  Les rôles multiples sont importants lorsque l'on veut varier l'exécutant à utiliser, pour chaque poste client, selon la préférence ou bien la condition d'exécution tels que le *fire-wall*, la performance, la sécurité, etc. lié à ce client.

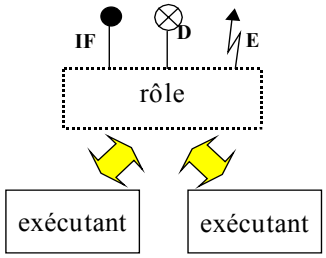

Figure 41. Un rôle multiple

Nous avons ajouté un rôle spécial : le rôle *anarchique*. L'objectif de ce rôle est de donner plus de liberté et de flexibilité aux exécutants jouant ce rôle : l'autorité commune ne cherche pas à savoir qui joue ce rôle et quelle est sa contribution à l'application globale.

De cette manière, le rôle anarchique permet de faciliter la participation des exécutants qui ne font qu'observer ou bien tracer l'exécution de l'application globale comme les outils de monitoring.

Dans notre modèle, le rôle anarchique n'est défini que par un ensemble de droit d'accès à l'univers commun. Le rôle anarchique est libre d'agir ou de réagir ; l'autorité commune ne contrôle pas les actions des exécutants jouant le rôle anarchique.

Par défaut, toutes les applications globales ont un rôle anarchique. Ce rôle peut être joué par zéro, un ou plusieurs exécutants. Ces exécutants peuvent être déclarés ou non dans la base de participants. S'ils sont déclarés, notre plate-forme de support leur propose les services comme pour tous les autres exécutants.

Il est intéressant de remarquer que lorsque l'on ne définit une application globale qu'avec le rôle anarchique ayant tous les droits d'accès à l'UC, on a le paradigme de coordination anarchique.

6.4. Modèle non-fonctionnel

Ce modèle défini les caractéristiques non-fonctionnelles d'un participant :

- Le type du participant. Il s'agit du type local ou distant ;
- Le moment où il faut « créer » une instance de ce participant. Dans notre modèle, nous proposons plusieurs possibilités concernant le moment de créer une instance du participant : au moment du démarrage de l'application globale, au moment où il y a « quelque chose » d'important (e.g. un événement qui se produit dans l'UC), au moment où le coordinateur a besoin d'un service du rôle correspondant, etc.
- La manière de créer une instance du participant. Pour des raisons de simplicité, nous considérons la création d'une instance d'un participant comme étant la création d'une instance de son adaptateur (cf. section 6.6.). De cette manière, il est nécessaire de spécifier la fabrique de l'adaptateur du participant.
- Le nombre d'instance du participant qui peuvent être « crée » pour une session d'exécution de l'application globale.

- S'il y a plusieurs instances d'un participant, ces instances doivent être exécutées sur des machines différentes ou sur une seule machine. Quelles sont ces machines ?

A partir des informations non-fonctionnelles ci-dessus, notre plate-forme de support est capable de gérer le cycle de vie des instances de participants. Notre plate-forme propose également des services pour localiser une instance d'un participant et pour communiquer avec cette instance même si elle est derrière un *fire-wall*. Nous détaillons l'implémentation de notre plate-forme de support dans le chapitre 6.

6.5. Concept d'adaptateur

Le concept d'**adaptateur** a été introduit afin de permettre à des exécutants existants de jouer leurs rôles dans une application globale sans avoir à changer leur code source.

Concrètement, l'adaptateur aide l'exécutant à communiquer avec le coordinateur dans les deux sens :

- Depuis le coordinateur vers l'exécutant (cf. section 6.5.1) : ceci veut dire que cette adaptation permet au coordinateur de communiquer avec l'exécutant ;
- Depuis l'exécutant vers le coordinateur (cf. section 6.5.2) : ceci veut dire que cette adaptation permet à l'exécutant de notifier son initiative au coordinateur.

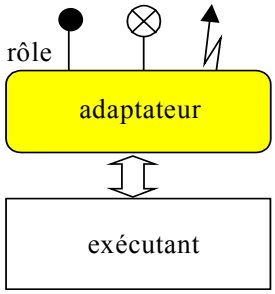

Figure 42. Adaptateur aide l'exécutant à implémenter ses rôles

6.5.1. Adaptateur : depuis le coordinateur vers l'exécutant

Dans l'objectif de supporter toutes les communications depuis le coordinateur vers l'exécutant, l'adaptateur doit :

- Implémenter l'interface des rôles joués par l'exécutant. Dans notre modèle, le rôle est abstrait ; il ne définit qu'une interface qui regroupe les fonctionnalités offertes. Cette interface doit être réellement implémentée par l'adaptateur en s'appuyant sur les fonctionnalités de l'exécutant ;

- Assurer toutes les adaptations techniques entre l'univers commun et l'exécutant. Ceci nous permet de surmonter l'hétérogénéité entre l'univers commun et l'exécutant en ce qui concerne : la représentation des concepts, la représentation des fonctionnalités, le protocole de communication, etc. ;

- Souscrire aux événements de l'UC et les écouter. Face à ces changements, la décision d'agir est à la charge de l'adaptateur.

La Figure 43 illustre le rôle de l'adaptateur pour assurer la communication depuis le coordinateur vers l'exécutant.

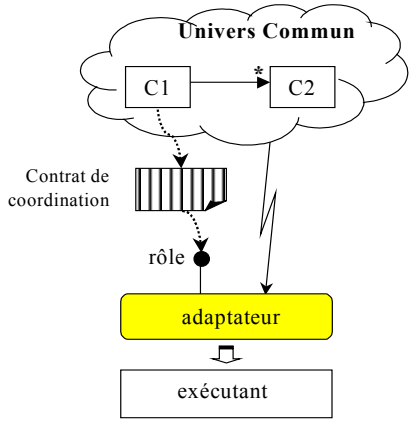

Figure 43. Depuis le coordinateur vers l'exécutant

6.5.2. Adaptateur : depuis l'exécutant vers le coordinateur

Pour les exécutants actifs ou réactifs (i.e. des exécutants ayant la capacité de changer leur état interne par leur initiative ou par leurs réactions), l'adaptateur doit assurer une tâche de plus : observer les changements de l'état interne de l'exécutant afin de les notifier au coordinateur.

Il s'agit d'une tâche importante. Elle permet à un exécutant actif ou réactif de propager ses changements aux autres exécutants. Pour ce faire, deux questions importantes doivent être posées : (1) A qui notifier ?, (2) Comment notifier ?

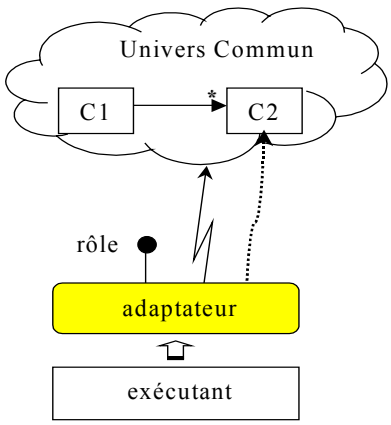

Figure 44. Depuis l'exécutant vers le coordinateur

Pour la première question, nous avons deux options : soit l'autorité commune (i.e. les contrats de coordination), soit l'univers commun, reçoit les notifications des participants. Parmi ces deux options, il est plus convenable de laisser l'UC recevoir les notifications car

c'est l'UC qui maintient l'état des concepts partagés entre les participants. En effet, même si l'on laisse les contrats de coordination recevoir les notifications, il est nécessaire de les synchroniser avec l'univers commun afin de garder la cohérence entre l'univers commun et les participants.

Pour la deuxième question qui est liée au moyen de notifier les changements, nous pouvons également envisager deux options :

- **Option 1** : L'adaptateur émet un événement. Dans ce cas, il est à la charge de l'UC d'écouter, d'analyser l'événement afin de savoir ce qu'il doit faire afin d'assurer la cohérence entre lui et l'ensemble des participants. Pour cela, il est nécessaire que l'événement contienne suffisamment d'informations pour que l'UC puisse analyser ce qui s'est passé chez ce participant et les conséquences pour l'ensemble des participants.

  Cette option est difficile à réaliser car elle demande à l'UC de connaître ce qui se passe chez les participants pour pouvoir se synchroniser avec eux.

- **Option 2** : L'adaptateur notifie l'UC en appelant une (des) méthode(s) de l'UC. Dans ce cas, il est à la charge de l'adaptateur de savoir les influences de son changement sur l'UC afin de déterminer la (les) méthode (s) de l'UC à appeler. Pour ce faire, il est nécessaire que chaque participant puisse accéder à l'UC et localiser les instances concernées.

  Cette option permet de rendre l'UC indépendant des participants. Il est à la charge des participants de savoir comment il faut synchroniser ses changements avec l'UC en déterminant la (les) méthode(s) de l'UC à appeler.

Dans notre implémentation, la deuxième option est sélectionnée. Cependant, un participant peut toujours émettre des événements pour notifier ses changements pour ceux qui s'y intéressent. Ceci peut être l'UC ou bien un participant quelconque.

6.5.3. Structure d'un adaptateur

Comme nous l'avons présenté dans la section 6.2, nous avons privilégié la distribution comme étant le critère principal pour classifier les exécutants. Dans cette section, nous discutons l'impact de la distribution sur la structure d'un adaptateur.

Dans le cas où l'exécutant est de type distant, nous proposons de décomposer classiquement l'adaptateur en deux parties : (1) le mandataire et (2) l'emballage.

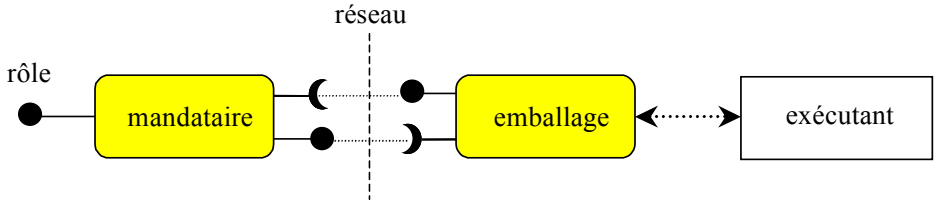

Figure 45. Adaptateur pour un exécutant distant

Le **mandataire**[5] est le représentant de l'exécutant auprès du coordinateur. Il s'exécute sur la machine serveur (i.e. la machine centrale). Sa tâche est d'assurer la communication entre le coordinateur et l'emballage de l'exécutant.

Concrètement, le mandataire implémente l'interface des rôles joués par l'exécutant en s'appuyant sur l'emballage. Lors de l'exécution, il transfère les appels de méthodes sur cette interface à l'emballage. Il écoute également les changements de l'UC afin de les envoyer à l'emballage.

Si l'exécutant est actif ou réactif, le mandataire est chargé de faire des appels de méthodes sur l'UC afin de notifier les changements d'état de l'exécutant. Il est important de noter que c'est l'emballage qui lui signale ces changements.

En revanche, l**'emballage**[6] est lié à l'exécutant. Il s'exécute sur la machine de l'exécutant. Sa tâche est d'assurer la communication entre le mandataire et l'exécutant. Concrètement, il :

- Gère le cycle de vie de l'exécutant (créer et détruire l'instance de l'exécutant, etc.) ;
- Observe l'exécutant afin de notifier au mandataire les changements (seulement pour les exécutants actifs et réactifs) ;
- Invoque les fonctionnalités de l'exécutant lorsque le mandataire le lui demande.

L'emballage dépend seulement de l'exécutant ; il peut être réutilisé dans plusieurs applications globales différentes. Par contre, le mandataire est spécifique à chaque composition. Ceci parce que le mandataire assure le lien avec le coordinateur qui varie en fonction de chaque composition pour construire une application globale spécifique.

Dans le cas où le participant est de type local, nous pouvons construire seulement le mandataire avec toutes les responsabilités de l'adaptateur comme le montre la Figure 46.

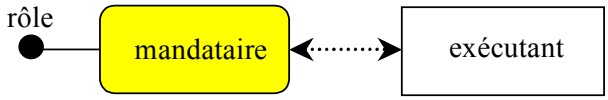

Figure 46. Adaptateur pour un exécutant local

Il est important de noter que la structure de l'adaptateur présentée ci-dessus correspond au cas général où l'exécutant est un élément logiciel existant. Dans le cas où un participant est construit spécifiquement pour être utilisé dans une fédération, les fonctionnalités du participant peuvent être réalisées directement dans l'emballage si le participant est de type distant, ou dans le mandataire si le participant est de type local. Dans le dernier cas, il n'y a ni emballage, ni exécutant (i.e. le mandataire est l'exécutant). Ceci nous permet d'éviter les surcoûts et les échanges inutiles, mais de respecter le même modèle pour tous les exécutants.

[5] Le terme en anglais est Proxy

[6] Le terme en anglais est Wrapper

Dans notre environnement de support, nous avons construit des éditeurs graphiques pour définir les participants en fournissant des formulaires destinés à saisir les caractéristiques fonctionnelles, non-fonctionnelles d'un participant et les fabriques de l'adaptateur. Nous allons revenir sur ces éditeurs dans l'annexe A.

6.5.4. Réalisation d'un adaptateur pour un exécutant existant

La réalisation d'un adaptateur d'un exécutant existant passe par l'implémentation du mandataire et de l'emballage.

L'implémentation d'un mandataire est assez simple par le fait que nous possédons l'ensemble des informations nécessaires telles que l'implémentation de l'univers commun, celle de l'emballage, etc.

Par contre, l'implémentation d'un emballage est plus complexe car elle est liée à l'exécutant dont nous n'avons pas forcément le code source. Dans ce cas, pour observer l'exécution, nous pouvons, par exemple : (1) souscrire à des événements émis par l'exécutant et (2) espionner l'exécutant.

La première solution est la plus simple, mais elle ne peut être utilisée qu'à condition que l'exécutant émette des événements pour notifier l'état de son exécution. En revanche, la deuxième solution demande de surveiller l'exécutant afin de raisonner sur ce qui se passe dans l'exécutant.

Actuellement, il y a plusieurs travaux qui s'intéressent à espionner l'exécution d'une application existante. Le travail de Balzer dans [Bal97, BG99] en est un exemple. Dans notre environnement, nous avons utilisé la machine à objet étendu – MOE [DES02] pour instrumenter l'exécution d'exécutants à la condition qu'ils soient écrits en `java`. Cette machine travaille avec le code binaire (i.e. *byte code*, en anglais) des exécutants.

Concernant la tâche d'exploiter des fonctionnalités offertes par un exécutant existant, elle sera plus simple si l'exécutant possède une API (*Application Programming Interface*) et à la condition que la sémantique de cet API soit compatible avec celle proposée par les rôles joués par l'exécutant. Le fait d'être compatible signifie qu'il est possible de traduire la sémantique des rôles en des fonctionnalités de l'API de l'exécutant. Par contre, pour des applications qui n'ont pas d'API, il faut les instrumenter afin d'intervenir sur les points nécessaires.

Nous avons donné quelques directions de travails concernant la réalisation l'emballage d'un exécutant existant. Néanmoins, nous n'avons pas souhaité nous attarder sur ce problème particulier car il est hors de la portée de cette thèse.

6.6. Création des instances de participants

La création d'une instance d'un participant n'est pas une tâche simple. Selon la nature des exécutants, la création d'une instance peut avoir des significations différentes.

Par exemple, pour une application, la création d'une instance correspond simplement à démarrer l'application sur la bonne machine, pour un service Web, la création d'une instance correspond à localiser le service Web.

Pour un outil, le fait de démarrer sur la bonne machine n'est pas suffisant ; il est nécessaire de passer une donnée (e.g. un fichier, un modèle, etc.) pour initialiser son état. Selon les caractéristiques des outils, le protocole d'initialisation peut être complexe. Ce peut être une commande simple telle que `'winword.exe C:\docs\scenario.doc'` pour démarrer `Microsoft WinWord` et passer `'C:\docs\scenario.doc'` comme fichier initial. Par contre, pour démarrer un client CVS, il faut passer le nom d'un compte et aussi le mot de passe pour ouvrir une section de travail [CVS].

De cette manière, nous proposons de déléguer la tâche de créer une instance d'un exécutant à son adaptateur. Ce dernier est, en effet, le seul qui connaît le protocole d'initialisation de l'exécutant. Par conséquent, pour notre plate-forme, la création d'une instance de participant correspond à créer une instance de son adaptateur. Il est ainsi nécessaire de définir la fabrique de l'adaptateur dans le modèle du participant.

Chaque instance d'un exécutant est liée avec une instance de son adaptateur. Ce dernier a pour objectif de gérer l'exécutant et de l'aider à travailler avec le coordinateur. Leur cycle de vie est identique. Pour notre environnement, la gestion du cycle de vie d'une instance de participant correspond à gérer celle de son adaptateur.

7. Fédération de participants

Nous utilisons le terme '*fédération'* pour désigner l'architecture logicielle qui résulte de la matérialisation du coordinateur. Elle constitue ainsi un univers commun et une autorité commune.

Une fédération de participants est une application globale construite à partir d'éléments de diverses natures en se basant sur le principe de coordination par le coordinateur. Elle est définie par un triplet <P, UC, AC> avec :

- **P** *:* Un ensemble de participants qui sont des membres de la fédération. Chaque participant est représenté par un triplet : rôles, exécutant et adaptateur ;

- **UC** *:* Un univers commun qui est utilisé pour synchroniser les participants. Il est une matérialisation des concepts partagés par les participants ;

- **AC** : Une autorité commune qui a pour objectif de contrôler explicitement la manière dont les membres de la fédération se coordonnent avec l'univers commun. L'autorité commune est matérialisée par un ensemble de droits d'accès à l'univers commun et un ensemble de contrats de coordination entre les participants. Il est important de noter que les droits d'accès à l'univers commun sont définis dans le modèle de participants. L'autorité commune ne fait que vérifier les accès à l'univers commun en considérant les droits définis dans le modèle des participants.

Il est important de noter que le fait d'étendre ou de restreindre le pouvoir de l'autorité commune permet de changer, de manière très flexible, le paradigme de coordination des participants avec l'univers commun. Lorsque l'autorité gère toutes les situations, nous obtenons le paradigme dictatorial. Si nous n'utilisons pas l'autorité commune (i.e. les droits et les contrats de coordination), nous avons le paradigme anarchique.

On peut exprimer simplement et graphiquement une fédération de participants comme dans la Figure 47. Le contrôle montant résulte de l'initiative des participants. Par contre, le contrôle descendant représente un guide du coordinateur visant à aider les participants à travailler ensemble afin de réaliser l'application globale.

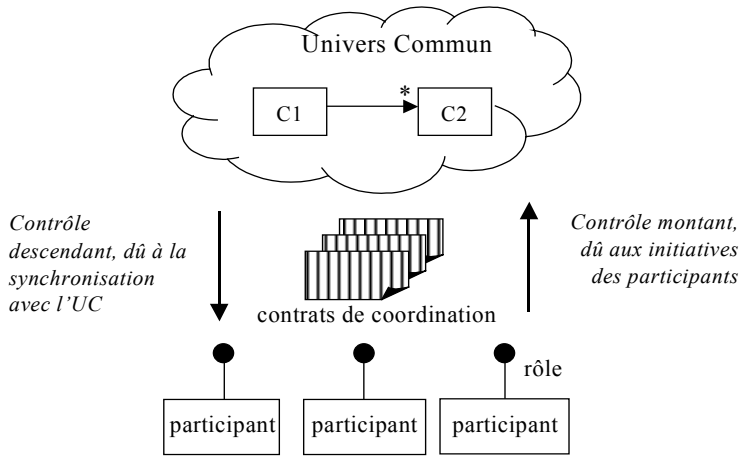

Figure 47. Une fédération de participants

8. Résumé et conclusions

Dans ce chapitre, nous avons étudié l'architecture de la fédération comme étant une solution pour résoudre les problèmes que nous avons cités au début de ce chapitre. Cette architecture est basée sur la coordination des participants avec l'univers commun. Ce dernier est conçu comme étant une matérialisation des concepts partagés entre participants. L'autorité commune peut être sollicitée afin de contrôler explicitement la coordination des participants avec l'univers commun.

Avec cette architecture, nous arrivons à construire les applications dans le cadre de la collaboration de cette thèse. Cependant, au moins deux problèmes ne sont pas résolus par notre proposition dans ce chapitre :

- L'évolution d'une fédération de participants. Pour ceci, il est nécessaire de diminuer les impacts dus au fait de remplacer, d'ajouter ou d'enlever un participant à une fédération ;
- La réutilisation d'une fédération de participants. Si c'est possible, elle nous permet de réutiliser non seulement un élément logiciel existant mais aussi un groupe d'éléments logiciels avec une fonctionnalité de haut niveau. Nous pensons que ceci va nous aider à construire des applications de grande taille.

Le chapitre suivant a pour objectif d'étudier les améliorations nécessaires pour mieux supporter l'évolution et la réutilisation d'une fédération de participants.

Chapitre IV

Domaines et composition de domaines

1. Introduction

Dans le chapitre précédent nous avons étudié une fédération comme étant une architecture logicielle, ouverte et dynamique, permettant de construire une application par la composition d'éléments logiciels de différentes natures.

Dans ce chapitre nous tentons de généraliser cette architecture. L'objectif principal est d'arriver à une approche de composition qui peut être utilisée pour construire les applications de petite taille et de grand taille (i.e. les applications entreprises d'intégrée évoquées dans [Dsouza01]) en facilitant l'évolution et l'adaptation de ces applications. Nous montrerons qu'avec le concept de domaine, nous sommes capables de grouper des éléments logiciels afin de mieux les réutiliser et les faire évoluer. Nous montrons aussi qu'avec la composition de domaines, il est possible de construire des applications de grande taille en composant des domaines qui peuvent être existants.

Le concept de domaine et la composition de domaines présentent un grand intérêt pour la réutilisation. Par la réutilisation de domaines, nous pouvons réutiliser non seulement un élément logiciel mais aussi un groupe d'éléments qui, par leur coordination, proposent une fonctionnalité de haut niveau.

Ce chapitre est organisé de la manière suivante. Dans la section 2, nous montrons les points faibles de l'architecture, présentés dans le chapitre 3. Dans la section 3, nous présentons la vision globale de notre solution en abordant rapidement le concept de domaine et la composition de domaines. Dans la section 4, nous étudions en détails la composition de domaines via la composition de modèles exécutables. La section 5 discute des manières de réutiliser et d'adapter un domaine. La section 6 fait le point sur la relation de notre approche avec les approches que nous avons étudiées dans le chapitre 2 et conclut ce chapitre.

2. Le problème à résoudre

L'architecture de fédération, que nous avons étudiée dans le chapitre 3, propose un mécanisme de composition intéressante. Tout d'abord, il nous permet de construire des applications en s'appuyant sur la coordination d'un ensemble d'éléments logiciels de diverses natures. En plus, les problèmes que nous avons cités au début du chapitre 3, tels

que le partage des concepts entre les éléments logiciels, l'initiative d'un élément logiciel, etc. sont réglés de manière assez simple et performante.

Néanmoins, au cours de nos expérimentations (cf. chapitre 6), nous nous sommes aperçus que cette architecture a quelques points faibles à propos de :

- *L'évolution d'une fédération de participants*. Dans une fédération, nous pouvons remplacer facilement un participant par un autre, à condition qu'ils répondent au même contrat fonctionnel (i.e. ils jouent les mêmes rôles). Néanmoins, il n'est pas aisé d'ajouter ou d'enlever un participant d'une fédération. Puisque l'univers commun est une matérialisation des concepts partagés entre tous les participants, le fait d'ajouter (ou d'enlever) un participant peut imposer d'ajouter (ou d'enlever) des concepts à l'univers commun et de reconsidérer tous les contrats de coordination déjà définis.

- *La réalisation d'une application de grande taille comme étant une fédération de participants*. Dans [Dsouza01], Dsouza a identifié les caractéristiques des applications de grande taille comme suit:

    - Elles couvrent plusieurs domaines d'activité (*subject area,* en anglais) dont chacun se concentre sur une activité métier différente. De cette manière, chaque domaine d'activité possède un ensemble de concepts liés à son activité métier ;
    - Les domaines d'activités peuvent partager des concepts ou être conceptuellement disjoints.

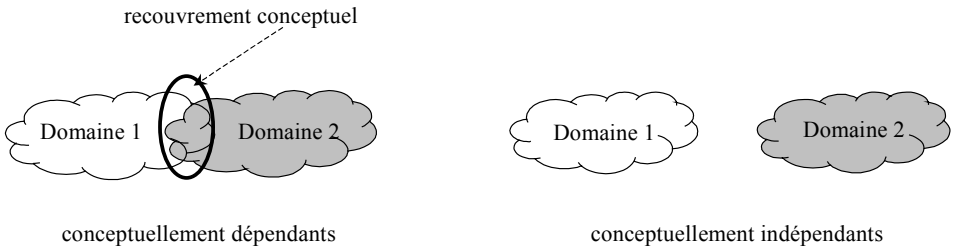

Figure 48. Interdépendances entre domaines

    - Un domaine d'activité peut être réalisé par un ou plusieurs éléments logiciels. Ces éléments implémentent les fonctionnalités du domaine en manipulant les concepts du domaine.

  Face à ces caractéristiques d'une application de grande taille, la définition de l'univers commun de la fédération correspondante s'est révélé difficile. En effet, il n'est ni simple, ni réaliste de matérialiser les concepts partagés entre tous les participants provenant de différents domaines d'activités si les domaines sont éloignés les uns des autres.

  En outre, pour ce type d'application, le point faible concernant l'évolution d'une fédération de participants devient très délicat. Le fait de reconstruire une fédération de grande taille coûte très cher en terme de temps et d'efforts.

Nous pensons que le problème de fond de l'architecture, proposé dans le chapitre 3, est de travailler à la fois avec tous les participants d'une fédération. D'une part, la réalisation de la fédération est difficile si les participants sont éloignés. D'autre part, le fait d'ajouter ou d'enlever un participant provoque les impacts profonds sur cette fédération.

Pour cela, il est nécessaire de grouper les participants qui sont liés et qui, en conséquence, partagent des concepts. Ainsi, le fait d'ajouter ou d'enlever un participant dans un groupe aura les impacts sur la réalisation de ce groupe et éventuellement sur les relations entre ce groupe avec d'autres groupes, mais, il n'y a pas d'impacts sur la réalisation de ces derniers.

Nous proposons d'utiliser le concept de domaine pour isoler un groupe de participants. Au sein d'un domaine, les participants peuvent se coordonner en utilisant l'architecture de fédération, présentée dans le chapitre 3. Un domaine sera vu par les fonctionnalités de haut niveau d'abstraction que ses participants fournissent en se coordonnant. En revanche, une application globale sera construite comme étant le résultat de la composition de domaines.

Nous pensons que le concept de domaine et la composition de domaines résolvent les problèmes de réutilisation et d'évolution :

- Par la réutilisation des domaines, nous pouvons réutiliser non seulement un élément logiciel mais aussi un groupe d'éléments logiciels qui fondent le domaine ;
- Par l'isolation des domaines, nous pouvons minimiser les impacts, causés par les changements dans un domaine.

Pour ce faire, dans ce chapitre nous allons :

- Etudier la manière de spécifier un domaine comme étant une unité de composition réutilisable ;
- Etudier la manière de composer les domaines qu'ils soient conceptuellement dépendants ou indépendants ;
- Etudier les possibilités de réutiliser et d'adapter un domaine.

Nous pensons qu'il s'agit d'un problème plus générique mais également plus complexe et difficile que le problème étudié dans le chapitre 3. En réalité, une application globale peut inclure un ou plusieurs domaines. Pour les applications globales ayant un seul domaine, il « suffit » de réaliser leur domaine unique. Par contre, les applications couvrant plusieurs domaines, seront considérées comme étant le résultat de la composition de domaines.

3. Vision globale de notre proposition

3.1. Concept de domaine

Nous définissons un *domaine* comme étant un groupe d'éléments logiciels, coordonnés afin de proposer une fonctionnalité clairement identifiée par un modèle de haut niveau d'abstraction.

A titre d'exemple, nous pouvons citer le domaine bancaire dont la fonctionnalité est la gestion des activités des comptes bancaires. Cette fonctionnalité est fournie par l'ensemble des éléments logiciels fondant ce domaine. Il s'agit d'un outil permettant d'ouvrir des comptes, d'une base de données de comptes, d'un outil servant à gérer les activités des comptes, d'un outil gérant les cours de la bourse, etc.

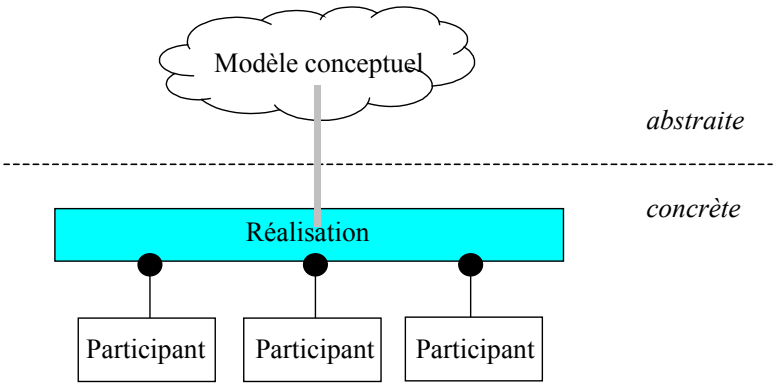

Figure 49. Les deux parties d'un domaine

Du point de vue conceptuel, un domaine contient deux parties principales :

- La partie abstraite, qui représente l'espace conceptuel du domaine. En contenant la sémantique abstraite du domaine, cette partie est constituée d'un ensemble de concepts, d'un ensemble de relations entre ces concepts, d'un ensemble de règles déterminant le comportement du domaine et d'un ensemble de règles déterminant la cohérence du domaine ;

- La partie concrète, qui représente la réalisation du domaine. En contenant la sémantique abstraite du domaine, cette partie est constituée d'un ensemble de participants qui contribuent à concrétiser la sémantique abstraite du domaine en utilisant les concepts venant de l'espace conceptuel du domaine.

La sémantique complète du domaine sera l'union de la sémantique abstraite complété par la sémantique concrète offerte par les participants.

La partie abstraite est fondamentale pour un domaine. D'une part, elle permet d'avoir une vision globale et de haut niveau d'abstraction du domaine. D'autre part, elle est indépendante de la réalisation (i.e. la partie concrète) du domaine. Il est clair que l'on peut sélectionner des participants différents pour réaliser un domaine à condition qu'ils soient capables de rendre concrète la sémantique abstraite du domaine.

Par conséquent, la partie abstraite d'un domaine peut être utilisée comme étant le critère principal pour sélectionner les participants de ce domaine. Un participant est considéré comme faisant partie d'un domaine s'il manipule les concepts de ce domaine en respectant les règles de fonctionnement et de cohérence de ce domaine.

Le fait de représenter la partie abstraite d'un domaine en UML nous donne le **modèle conceptuel** de ce domaine. Le modèle conceptuel d'un domaine contient la structure des

concepts mais aussi la sémantique abstraite du domaine. Cette sémantique est représentée par les méthodes des classes UML. Les règles de cohérence peuvent être représentées par des expressions de condition OCL.

Du point de vue de l'implémentation, un domaine est une fédération de participants. Les concepts partagés par les participants est un sous-ensemble des concepts provenant de l'espace conceptuel du domaine. Par conséquent, les classes de l'univers commun de la fédération correspondant au domaine, représentent un sous-ensemble des classes du modèle conceptuel du domaine.

3.2. Composition de domaines

3.2.1. Composition au niveau modèle

Pour nous, la composition de domaines ne pourrait pas être réalisée en connectant directement les participants provenant de ces domaines. En effet, avec des connexions directes entre participants, le fait de changer ou d'enlever un participant d'un domaine impliquerait de vérifier, dans tous les autres domaines, les liens vers ce participant. L'évolution d'un domaine deviendrait difficile, et nous ne pourrions pas considérer un domaine comme une unité de composition de haut niveau d'abstraction. C'est justement ce que nous cherchons à éviter.

Nous proposons de considérer en détail une connexion entre deux participants P1 et P2 appartenant à deux domaines. Supposons que cette connexion soit réalisée par un appel de méthode, de *m1_2()* de P1 vers *m2_1()* de P2 (cf. la Figure 50). En réalité, cette connexion est la conséquence du fait que l'application globale a établi une relation R entre deux concepts C1 et C2 manipulés respectivement par P1 et P2.

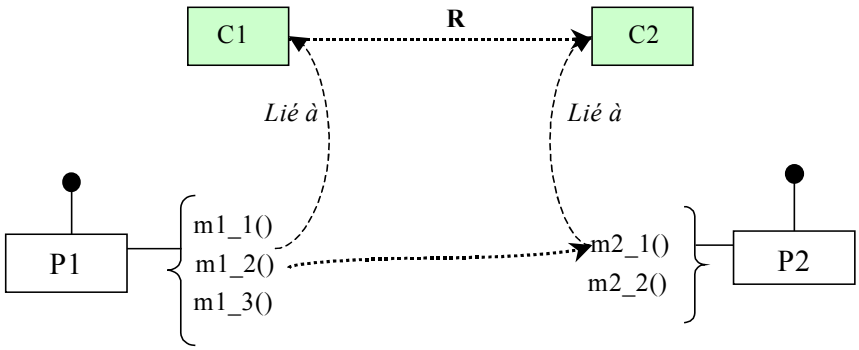

Figure 50. Connexions entre deux participants

Par exemple, imaginons que nous composons un domaine bancaire, travaillant sur les concepts de titulaire et de compte, etc. avec le domaine de sécurité sociale qui manipule les concepts d'assurée, de contrat d'assurances, etc. Une connexion peut être établie entre l'outil de gestion bancaire et l'outil de gestion des sécurités. Celle-ci est due au fait qu'il y a une relation entre un assuré et un titulaire de compte (e.g. ce sont les même personnes ou bien il y a des relations familiales entre eux).

Une relation entre concepts est exprimée à un niveau d'abstraction plus élevé que les relations entre participants ; elle nous permet d'étudier la source des besoins de connecter les participants.

Notre objectif est d'étudier un moyen permettant de remplacer les connexions directes entre des participants, par les connexions entre des concepts provenant des modèles conceptuels des domaines à composer. Ceci nous permet de :

- Réfléchir, travailler et valider la composition de domaines en se basant sur la composition de modèles conceptuels de ces domaines ;
- Faciliter l'évolution de l'application globale, car le modèle conceptuel d'un domaine est indépendant des participants et des plates-formes.

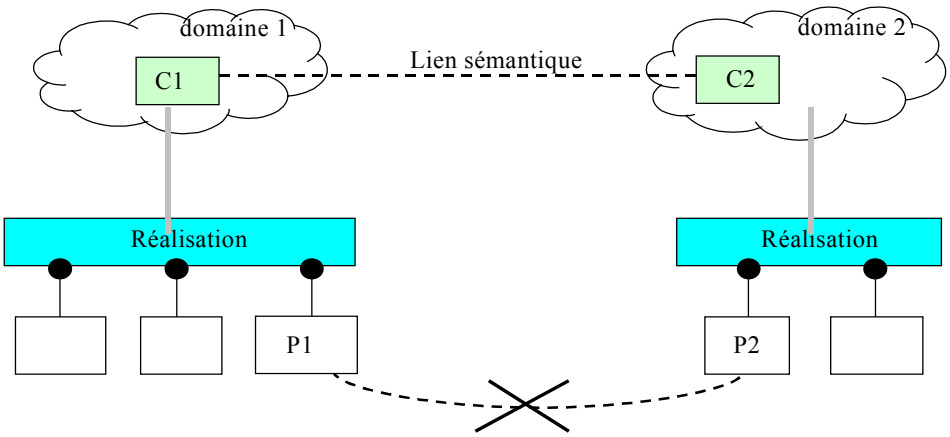

Figure 51. Lien entre domaines

Pour atteindre cet objectif, il est nécessaire d'étudier la manière permettant de :

- Rendre exécutable le modèle conceptuel d'un domaine. Ceci nous permet d'effectuer la composition de modèles au lieu de la composition de participants ;
- Synchroniser, à tout moment, l'état du modèle exécutable et l'état des participants. Ceci nous permet de transformer réellement la composition de modèles en la composition de participants.

3.2.2. Transformation du modèle conceptuel en l'univers commun

Avec l'architecture de fédération, présentée dans le chapitre 3, nous sommes capables de : (1) rendre exécutable le modèle conceptuel d'un domaine et (2) de synchroniser l'état du modèle exécutable et l'état des participants. Pour ce faire, il « suffit » de traduire le modèle conceptuel du domaine en l'univers commun de la fédération correspondante au domaine.

En effet, la coordination des participants avec l'univers commun fait en sorte que :

- L'exécution des participants rend le modèle conceptuel exécutable ;
- L'exécution du modèle implique l'exécution des participants du domaine ;

- La cohérence entre l'état du modèle conceptuel et l'état des participants, est assurée à tout moment.

Par conséquent, la composition de modèles conceptuels exécutable implique indirectement la composition de participants.

Dans le chapitre précédent, nous avons définit l'univers commun comme étant une abstraction des concepts partagés entre les participants du domaine. Nous pensons que ceci n'est pas en contradiction avec le fait de concevoir l'univers commun comme la matérialisation du modèle conceptuel du domaine car :

- Les concepts partagés entre les participants d'un domaine représentent, en général, un sous-ensemble du modèle de domaine. Ils sont nécessaires et suffisants pour coordonner les participants au sein d'un domaine unique ;

- Si l'on souhaite composer ce domaine avec d'autres domaines, il est indispensable de remonter, à l'univers commun, tous les concepts qui sont potentiellement nécessaires pour la composition de domaines, même s'ils ne servent pas à coordonner plusieurs participants au sein du domaine.

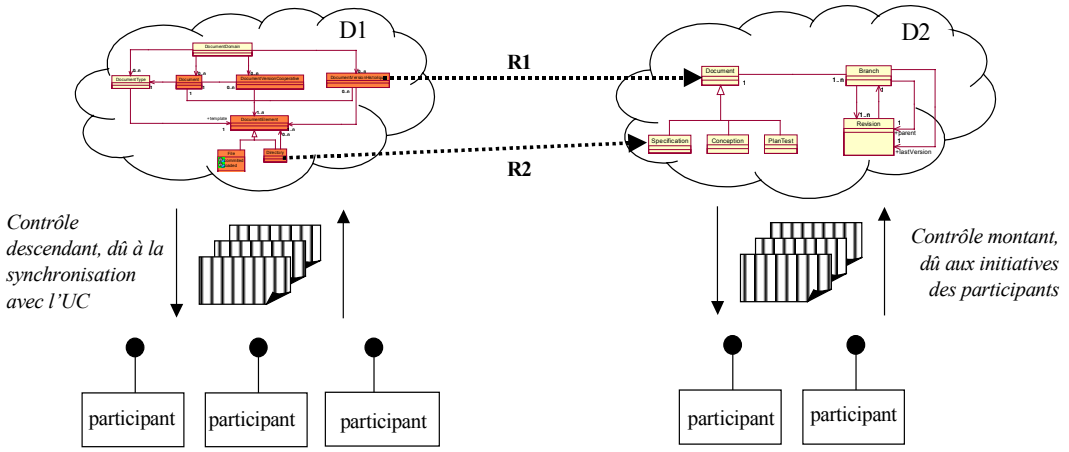

Figure 52. Composition de modèles exécutables pour composer de domaines

De cette manière, dans l'objectif de pouvoir composer les domaines en s'appuyant sur la composition de modèles exécutables, nous proposons de transformer le modèle UML d'un domaine en des classes (les classes `Java`, par exemple) de l'univers commun de la fédération correspondant à ce domaine. Pour simplifier, nous proposons d'appliquer une transformation simple et directe depuis les concepts du modèle conceptuel du domaine vers les classes de l'univers commun :

- Un concept sera transformé en une classe ;
- Une classe inclut les informations liées au contexte d'exécution. Ceci veut dire qu'en se basant sur l'information contenue dans une instance d'une classe, il est possible de raisonner sur son état d'exécution.

Dans la suite de ce document, nous ne faisons pas de différence entre un concept et la classe correspondante. Nous utilisons le terme de concept lorsque nous voulons rester à un

haut niveau d'abstraction. Nous utilisons le terme de classe lorsque nous parlons de l'implémentation.

Dans la partie 4, nous allons étudier la composition de modèles exécutables et la gestion de cette composition. Cependant, nous préférons présenter l'application de gestion documentaire qui nous permettra, tout au long de ce chapitre, d'illustrer notre proposition.

3.3. Application de gestion documentaire

Il s'agit de l'application développée dans le cadre du projet Centr'Actoll dans lequel cette thèse a été effectuée [Actoll].

L'objectif de cette application est de construire un environnement générique de gestion documentaire dans lequel plusieurs personnes travaillent afin de produire collectivement des documents. Du point de vue non-fonctionnel, Actoll souhaite que cette application soit simple à modifier et que le processus de production d'un document soit modélisé et contrôlé par un moteur de procédé.

Le cahier des charges de cette application peut être résumé comme suit :

- Un document n'est pas juste un fichier, il peut contenir un nombre élevé des fichiers qui sont structurés hiérarchiquement dans des répertoires. Un document peut être une *'spécification'*, un *'plan de test'* ou bien un document de *'conception'*. Ces trois types de documents sont distingués par les types de fichiers qu'ils peuvent contenir. Un document est versionné (i.e. il peut avoir plusieurs versions) ;

- La production d'un document est décomposée en deux phases dont chacune est composée de plusieurs étapes (i.e. tâches) pouvant être réalisées en parallèle ou en séquence.

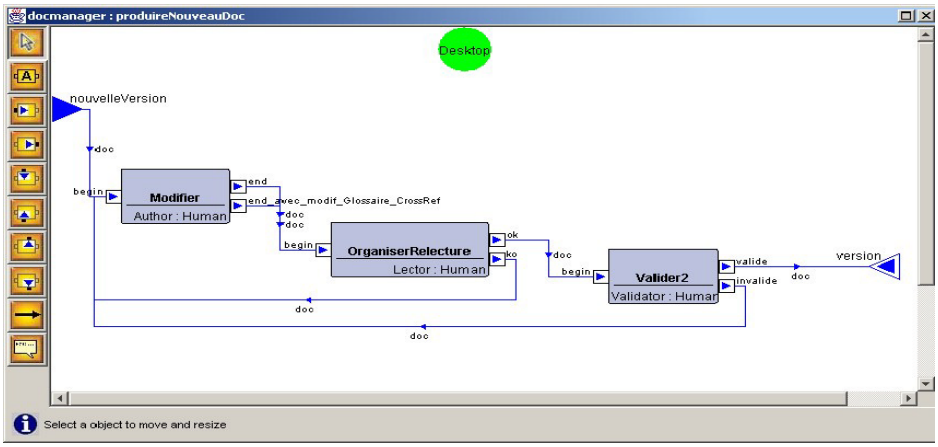

Figure 53. La deuxième phase du processus de gestion documentaire

La première phase a pour objectif de créer la version initiale du document. Elle est décomposée en trois étapes nommées *'créer'*, *'rédiger'* et *'valider'* servant respectivement à créer, rédiger et valider un document.

La deuxième phase a pour objectif de modifier, de relire et de valider le document crée dans la première phase. Elle est constituée de neuf activités structurées à deux niveaux dont les activités de premier niveau sont illustrées dans la Figure 53.

- On distingue quatre types de personnes en fonction de leur capacité et leurs responsabilités dans le processus de production d'un document : '*auteur'*, '*lecteur', 'organisateur'* et '*validateur'* ;
- Chaque personne a son espace de travail sur une machine, dans lequel sont stockées les versions des documents sur lesquels elle travaille.

De manière simple, cette application est composée de quatre domaines d'activité : (1) la gestion des ressources humaines dont la tâche est de gérer les membres d'un groupe, (2) la gestion des documents qui a pour objectif de définir et de créer les documents et leurs versions, (3) la gestion de procédé dont la tâche est de définir et de contrôler le processus de production d'un document et (4) la gestion des espaces de travail qui a pour objectif de gérer les espaces de travail des membres d'un groupe.

Le domaine de procédé joue le rôle d'un domaine de contrôle. Il guide les autres domaines afin de supporter la production coopérative d'un document. Dans le cadre de nos expérimentations, ce domaine fait partie de toutes les applications ayant besoin d'un guidage par un procédé.

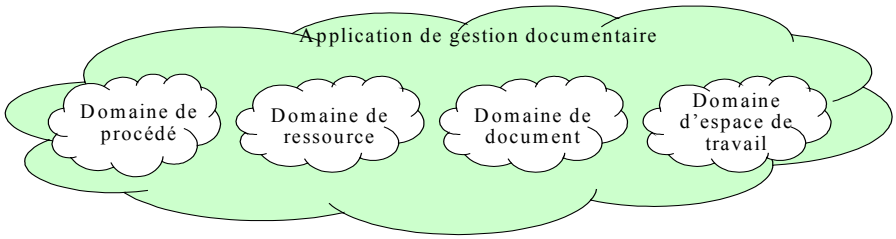

Figure 54. Vision globale de l'application de gestion documentaire

Pour réaliser cette application nous avons utilisé plusieurs éléments logiciels existants qui sont des outils issus du marché, des applications patrimoines trouvées au sein de notre équipe et également des codes spécifiques qui sont développés pour cette application. Des exemples de ces éléments sont : APEL qui est un outil de procédé développé au sein de notre équipe depuis les années 90s [EDA97, EDA98], CVS (*Concurrent Versions System)* qui est un outil générique de gestion des versions [CVS], des applications de gestion d'espaces de travail, etc.

Dans cette section, nous ne considérerons que les modèles conceptuels de ces domaines afin d'illustrer nos propositions à propos de la composition de domaines guidée par la composition de modèles exécutables.

3.3.1. Domaine de ressource

Le concept de personne est central dans modèle conceptuel de ce domaine. Quatre types de personnes sont spécialisés par quatre sous-classes de la class *Personne* (cf. Figure 55).

Ils ont pour objectif afin de spécifier explicitement les responsabilités et les droits des personnes de ces types.

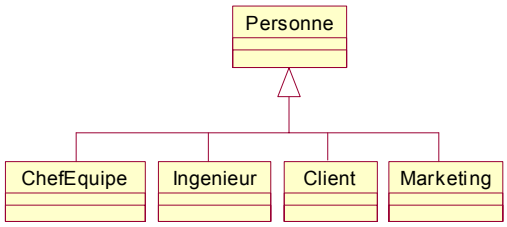

Figure 55. Modèle simplifié du domaine de ressource

3.3.2. Domaine de document

Le concept de document est central dans le modèle de ce domaine. Un document a plusieurs versions qui sont structurées dans les branches différentes. La classe *Document* est spécialisée par les sous-classes dans l'objectif de définir les différents types de document comme le montre la Figure 56.

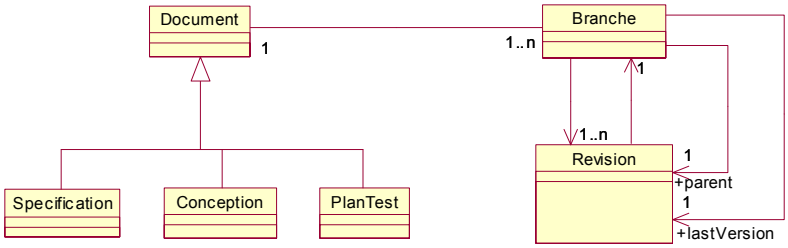

Figure 56. Modèle simplifié du domaine de document

3.3.3. Domaine de procédé

Le domaine de procédé est conçu comme étant une machine de flot de donnée. Son modèle conceptuel simplifié est illustré dans la Figure 57.

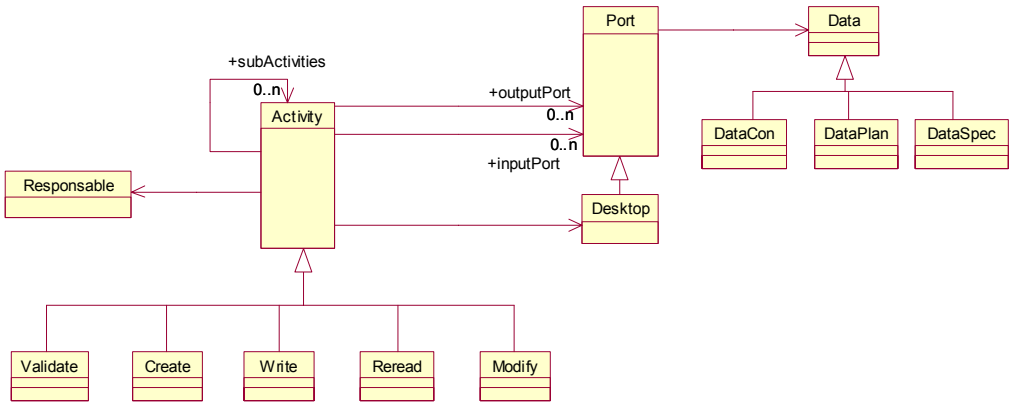

Figure 57. Modèle simplifié du domaine de procédé

Les activités principales du processus de production sont spécifiées par les sous-classes de la classe *Activity*. Les données consommées à l'entrée et produites à la fin d'une activité sont représentées par la classe *Data*. Il y a trois types de données, représentés par les sous-classes de *Data*, qui peuvent circuler dans le procédé de ce domaine.

3.3.4. Domaine d'espace de travail

Ce domaine travaille avec les espaces de travails qui sont une hiérarchie de répertoires et de fichiers. Deux types d'espace de travail sont définis, distingués par les stratégies de contrôle des fichiers et des répertoires contenus dans l'espace de travail de ce type.

Chaque personne peut avoir un espace de travail courrant et plusieurs espaces de travail historique. Dans l'espace de travail courrant, il y a les fichiers en cours de modification.

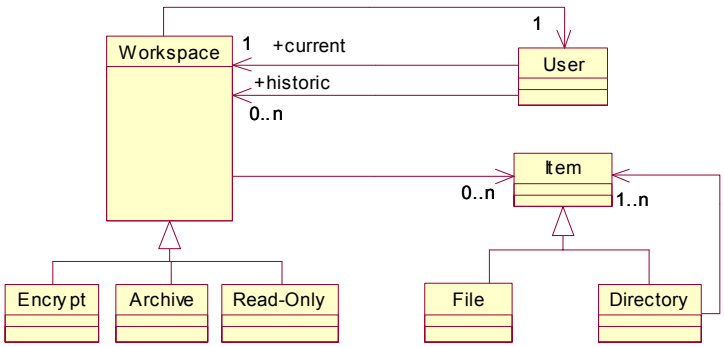

Figure 58. Modèle simplifié du domaine d'espace de travail

4. Composition de modèles exécutables

En ce qui concerne la composition de modèles exécutables, notre objectif n'est pas de fusionner ces modèles pour construire un modèle unique comme dans [HK03, KK03, AK02], mais de garder, à tout moment, la synchronisation entre ces modèles. La raison principale de ce choix est que nous voulons préserver l'indépendance et l'autonomie des domaines composés en gardant l'indépendance de leur modèle exécutable. Le modèle exécutable d'un domaine sert non seulement à la coordination avec d'autres domaines mais aussi à coordonner les participants au sein de ce domaine.

Dans cet objectif, il est nécessaire de définir des relations entre des modèles conceptuels et de gérer ces relations lors de l'exécution. Nous proposons de considérer des relations entre des modèles conceptuels comme des relations entre des concepts provenant de ces modèles. De cette manière, le principe de la composition de modèles conceptuels exécutables peut être résumé comme suit :

- Lors de la conception, il est nécessaire de définir les relations entre des concepts provenant des modèles conceptuels des domaines composés. La définition d'une relation doit contenir l'information spécifiant la manière de gérer cette relation ;

- Lors de l'exécution de l'application résultante, gérer une relation consiste à assurer, à tout moment, la synchronisation entre des instances des concepts concernés par cette relation.

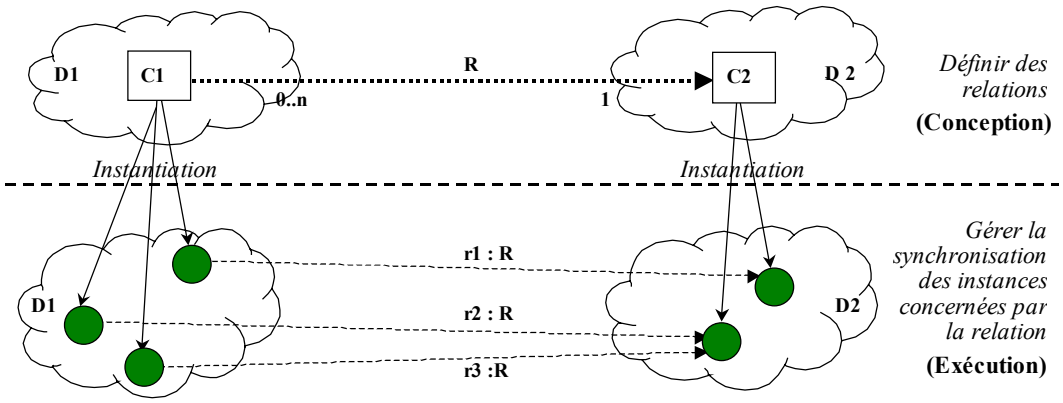

Figure 59. Principe de la composition de modèles exécutables

De cette manière, les domaines se coordonnent à travers les relations établies entre leurs modèles. Une relation n'est pas simplement un lien conceptuel entre deux concepts, mais un moyen de synchroniser les modèles exécutables des domaines. Ce qui implique, à son tour, la synchronisation des participants de ces domaines.

Tout d'abord, nous expliquons quelques termes de base qui sont liés à la définition et à la gestion des relations entre les concepts.

4.1. Relations

Nous définissons une relation comme étant un lien sémantique entre des concepts. Une relation peut être établie entre deux ou plusieurs concepts. Dans le premier cas, nous avons une **relation binaire**.

Dans la suite de ce document, nous ne nous intéressons qu'aux relations binaires. Une manière pour transformer une relation n-binaire en une relation binaire sera étudiée dans la section 4.4.

4.1.1. Orientation d'une relation

Considérons une relation R1 établie entre deux concepts C1 et C2 appartenant respectivement aux domaines D1 et D2, le domaine D1 est considéré comme un **domaine actif** par rapport à cette relation lorsqu'il est capable d'agir sur les instances du concept C1 sans y être invité par le domaine C2.

Un changement sur les instances du C1 peut impliquer de changer l'état les instances correspondantes de C2. Si le concept C2 n'est modifié que comme conséquence d'une action sur C1, le domaine D2 est considéré comme étant un **domaine passif** par rapport à cette relation.

Par conséquent, on peut considérer que le concept C1 comme étant le **concept actif** et le concept C2 comme le **concept passif** par rapport à la relation entre eux.

Nous proposons d'utiliser l'orientation d'une relation pour désigner l'initiative des concepts liés à cette relation. Une relation peut être orientée ou non. Si elle est orientée, il y a certainement un domaine actif et un domaine passif qui participent à cette relation. Sinon, les deux domaines sont actifs par rapport à cette relation.

Il est intéressant de noter qu'une relation non-orientée, entre C1 et C2, peut être considérée comme étant deux relations orientées, une depuis C1 vers C2 et l'autre depuis C2 vers C1. C'est pour cette raison que dans la suite de ce document, nous ne nous intéressons qu'aux relations binaires orientées.

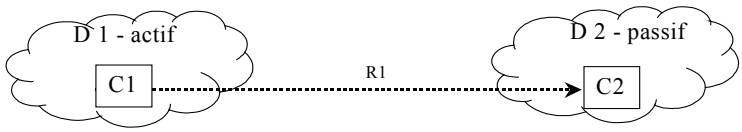

Figure 60. Domaine actif et passif par rapport à R1

Dans l'exemple illustré par la Figure 60, le domaine D1 est actif tandis que le domaine D2 est passif par rapport à la relation R1 entre C1 et C2. Par conséquent, C1 est considéré comme le concept actif tandis que C2 est le concept passif par rapport à R1.

4.1.2. Paires d'instances

Une relation peut avoir plusieurs **instances**. Une instance de la relation relie une paire d'instances dont chacune est l'instance d'un concept concerné par cette relation

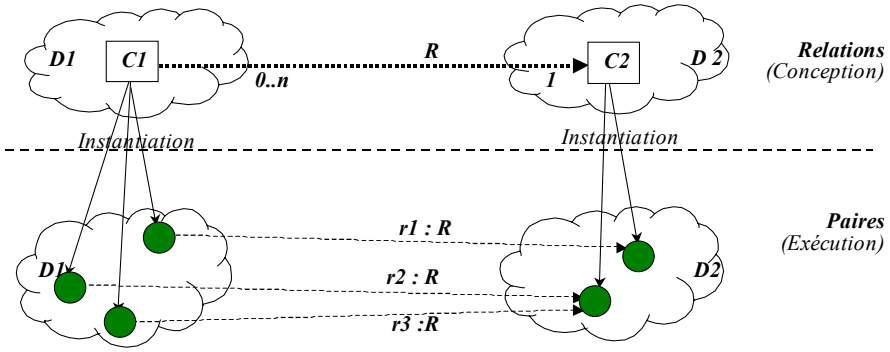

Figure 61. Paires d'instance d'une relation

Nous proposons d'utiliser la cardinalité d'une relation pour désigner le nombre de paires auxquelles une instance d'un concept peut participer. Considérons, par exemple, la relation de cardinalité N-1 entre C1 et C2 dans la Figure 61, une instance de C1 ne peut participer qu'à une paire, mais une instance de C2 peut participer à plusieurs paires.

La gestion d'une relation correspond au fait d'établir des paires d'instances et de garder la cohérence entre les instances d'une paire.

4.1.3. Contraintes sur une relation

Du point de vue du gestionnaire de relations, la relation elle-même permet de spécifier la manière d'organiser les instances qui sont liés. Ceci veut dire que l'on ne peut mettre ensemble que les éléments qui sont les instances des concepts concernés par une relation.

Pour être plus précis, on peut définir les **contraintes** additionnelles sur une relation pour définir la manière d'établir les paires d'instance pour cette relation. Si les contraintes sur une relation sont suffisamment précises, elles peuvent permettre de calculer automatiquement les paires. Ceci veut dire que lorsqu'une instance d'une paire est déterminée, l'autre instance peut être identifiée sans ambiguïté.

Pour exprimer une contrainte sur une relation, on peut utiliser les expressions de conditions. Par exemple, pour la relation R entre C1 et C2, on peut imaginer une contrainte simple spécifiant le fait que les instances d'une paire doivent porter un même nom comme suit :

```
C1.nom = C2.nom
```

Cette règle est simple mais assez performante. Dans l'application de gestion documentaire, nous appliquons souvent cette règle pour calculer automatiquement les paires d'instance de nos relations.

Les contraintes sur les relations sont très intéressantes lorsque l'on veut restreindre les candidats potentiels ou calculer automatiquement les paires d'instances. Notons que les contraintes ne sont pas obligatoires pour définir une relation.

4.1.4. Sémantique d'une relation

La sémantique d'une relation permet de spécifier la manière de gérer cette relation. Elle contient des informations permettant de : (1) établir les paires de cette relation et (2) synchroniser les instances d'une paire tout au long de leur cycle de vie.

Etablir les paires pour une relation consiste à définir :

- Le moment où il faut établir les paires. Il est très fréquent qu'une paire soit établie lorsqu'une instance du concept actif est crée ;
- La façon d'établir les paires d'instances. Les contraintes sur les relations (cf. 4.1.3) représentent un bon moyen pour définir la façon d'établir les paires d'instances. Si la relation n'a pas de contraintes ou les contraintes ne permettent pas de calculer les paires, il faut utiliser les moyens supplémentaires : par l'intervention humaine. Nous regardons ces moyens plus tard dans ce document (cf. 4.3.3).

En revanche, synchroniser les instances d'une paire consiste à définir les actions de C1 entraînant une synchronisation de C2, et en quoi consiste la synchronisation de C2 si la relation est orientée depuis C1 vers C2.

En se basant sur ces informations, le gestionnaire de relations est capable d'établir les paires d'instances et de les synchroniser tout au long de leur cycle de vie.

4.2. Types de relation

Dans cette section, nous essayons d'identifier quelques types de relations entre des concepts provenant des modèles conceptuels des domaines bien qu'ils soient conceptuellement dépendants (i.e. ils partagent des concepts) ou indépendants (i.e. ils ne partagent pas des concepts).

4.2.1. Relation identique

Lorsque deux domaines sont conceptuellement dépendants, ils partagent des concepts bien que chaque domaine ait sa propre manière de représenter et d'utiliser ces concepts.

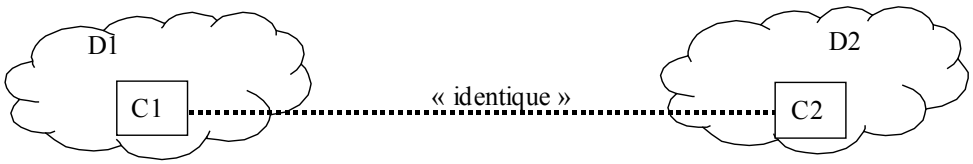

Figure 62. Relation identique

Dans la Figure 62, le domaine D1 modélise le concept partagé par son concept C1 et le domaine D2 le modélise par son concept C2. Néanmoins, C1 et C2 sont le même concept dans la composition de ces deux domaines. Nous proposons d'appeler la relation entre C1 et C2 comme étant une relation de type **'relation identique'**.

Une relation identique signifie qu'il faut garder la cohérence, à tout moment, entre les concepts liés par cette relation. Lorsqu'un domaine change son concept, il est nécessaire que l'autre domaine synchronise l'état de son concept afin de garder la cohérence entre ces concepts.

En théorie, une relation identique n'est pas orientée. Ceci, parce que les domaines concernées ont le même rôle par rapport à cette relation. Tous ces domaines peuvent décider d'agir sur leur concept lié à la relation. En réalité, une relation identique est souvent orientée. Ceci dépend de l'initiative des domaines concernés par la relation.

Revenons à l'application de gestion documentaire, les relations identiques ont été utilisées pour composer les domaines. Dans la Figure 63 les relations identiques ont été établies entre le domaine de procédé et le domaine de document.

Le domaine de procédé possède le concept de `Data` désignant une donnée qu'une activité consomme à l'entrée et produit à la sortie. En revanche, le domaine de document définit et gère le concept de `Document`. Or, un document ne représente qu'une donnée spécifique. Nous pouvons dire que deux domaines partagent le concept de donnée.

Ainsi, dans la composition de ces domaines, une relation identique, de cardinalité N-1, est établie entre `Data` et `Document`. Par conséquent, toutes les manipulations du domaine de procédé sur les données sont transférées vers les documents correspondants ; le domaine de procédé est donc capable de guider le processus de production des documents.

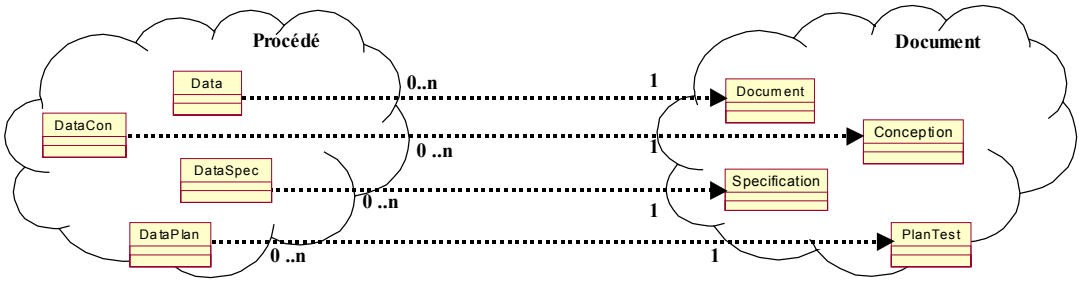

Figure 63. Les relations identiques entre le domaine de procédé et document

Les relations identiques sont également établies entre les types de données et les types de document afin de raffiner la relation entre `Data` et `Document`. Pour chacune de ces relations, nous pouvons définir la manière de la gérer différemment.

4.2.2. Association entre concepts

Lorsque des domaines à composer sont conceptuellement indépendants, leurs concepts peuvent être reliés dans la composition de domaines par une relation ayant une sémantique associée. Nous considérons une telle relation comme étant une association entre concepts.

L'objectif principal d'une association de concepts est de permettre à un domaine – le domaine actif – de propager ses actions vers l'autre domaine jouant le rôle du domaine passif par rapport à cette association. Pour cet objectif, une association entre deux concepts est toujours orientée, depuis le domaine actif vers le domaine passif. Par conséquent, les actions sur le concept actif vont impliquer des effets de bord sur le concept passif.

Il est intéressant de noter qu'une relation identique n'est qu'un cas particulier d'une association entre concepts. Une relation identique qui n'est pas orientée entre deux concepts C1 et C2, peut toujours être traduite en deux associations orientées : une de C1 vers C2 et une de C2 vers C1.

A titre d'exemple, on peut considérer la composition entre le domaine de procédé et le domaine d'espace de travail (*workspace*, en anglais). L'idée de cette composition est de pouvoir gérer l'espace de travail des utilisateurs qui participent au processus de production des documents.

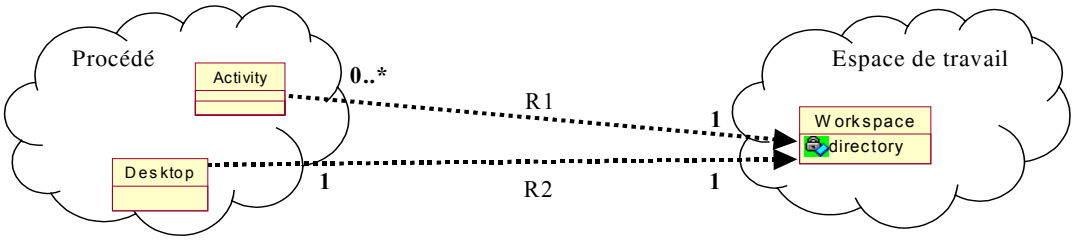

Figure 64. Les associations le domaine de procédé et d'espace de travail

Pour cet objectif, deux associations sont établies comme l'illustre la Figure 64 :

- R1 entre `Activity` et `Workspace` : Cette association a pour objectif de synchroniser l'espace de travail par rapport à l'activité ; Un espace de travail est crée (ou détruit) lorsque l'activité correspondante est crée (ou détruite) ;
- R2 entre `Destop` et `Workspace` : Cette association a pour objectif de synchroniser les produits dans un bureau[7] (i.e. *destop*, en anglais) avec ceux contenus dans un espace de travail. Chaque fois qu'un produit arrive (ou part) dans le `Destop`, il faut ajouter (ou retirer) un document[8] dans l'espace de travail correspondant.

Ces associations sont toutes orientées depuis le domaine de procédé vers le domaine d'espace de travail. Ceci, parce que le domaine de procédé est le seul actif parmi les quatre domaines de l'application de gestion documentaire. C'est ce domaine qui guide tous les autres domaines dans le processus de production des documents.

4.3. Gestion de relations

4.3.1. Extension de la sémantique par l'instrumentation du code

Pour gérer une relation, la question de fond est de savoir qui doit connaître et garantir la cohérence entre les domaines. Il est certain que ceci ne peut pas être la responsabilité des domaines composés. Ces domaines doivent rester inchangés ; en revanche, ils seront guidés dans l'objectif de garder la synchronisation entre leurs modèles.

Nous proposons d'étendre le concept actif afin de gérer les relations auxquelles il participe. Ceci veut dire que l'on ajoute une sémantique additionnelle dans les instances du concept actif (i.e. l'instance active) pour : (1) exprimer le lien vers l'instance du concept passif (i.e. l'instance passive) et (2) synchroniser l'instance passive par rapport à chaque action des instances actives.

Figure 65. Extension de la sémantique pour gérer la relation

Cette solution est assez simple. La sémantique d'une relation est représentée par l'extension du concept actif. Cependant, il reste plusieurs questions ouvertes : (1) Quel est le moyen pour étendre le concept actif, mais sans le changer, (2) Comment exprimer le

[7] Notons que le *Destop* est un port spécifique d'une activité. Ce port sert à stocker toutes les données qui sont utilisées par l'activité.

[8] Notons que l'on compose aussi avec le domaine de document. Ce dernier aide à déterminer ce qu'est le document à mettre (à retirer) dans l'espace de travail.

lien vers l'instance passive ? (3) Comment exprimer et gérer la synchronisation de l'instance du concept passif pour chaque action de l'instance du concept actif ?

Afin d'étendre le concept actif, mais sans avoir à le changer, la solution que nous proposons consiste à instrumenter le code de la classe représentant le concept actif, afin d'ajouter dynamiquement, lors de l'exécution, une sémantique additionnelle visant à gérer les relations auxquelles ce concept participe.

Dans la section 4.1.4, nous avons étudié ce qu'est la sémantique d'une relation. Par conséquent, considérons la relation R orientée depuis la classe C1 vers la classe C2, ce qu'il faut ajouter à la classe C1 afin de gérer la relation R constitue :

- Un moyen pour représenter les paires. Pour ceci, il « suffit » d'ajouter un attribut dans la classe C1 afin de référencer l'instance passive d'une même paire ;
- Le moyen pour établir les paires. Pour ceci, il est nécessaire d'ajouter, aux constructeurs de C1 et/ou des méthodes ayant pour objectif de créer une instance de C1, une sémantique `S` visant à localiser l'instance de C2 convenable et à établir un lien physique vers cette instance (i.e. remplir l'attribut ajouté dans C1 pour représenter l'instance passive d'une même paire) ;
- Le moyen pour synchroniser l'instance passive. Pour ceci, il est nécessaire d'ajouter, pour chaque méthode `m()` de C1 ayant besoin d'une réaction de l'instance de C2, une sémantique `S` qui a pour objectif de solliciter l'instance de C2 à synchroniser.

Afin de représenter la sémantique `S` ajoutée à une méthode de la classe active, nous proposons d'utiliser un autre type de contrat : les contrats de composition. De cette manière, chaque méthode `m()` de C1 ayant besoin de synchroniser des instances de C2, devient `m'()` dont la sémantique peut être considéré comme `m()` et l'ensemble de contrats attachés.

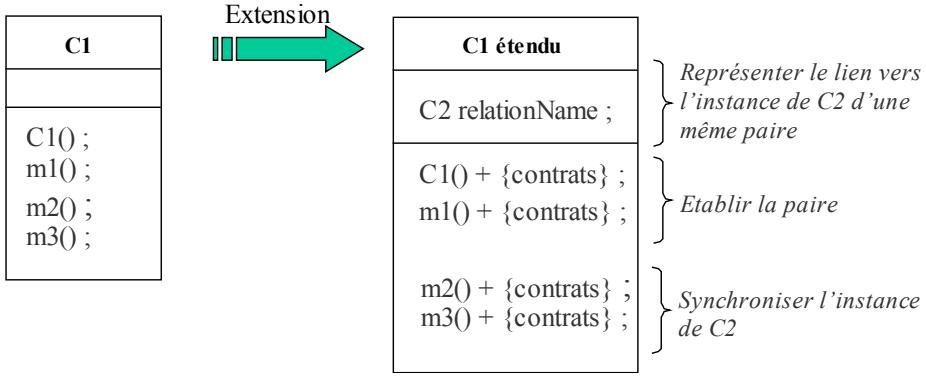

Figure 66. L'extension par l'instrumentation

Le fait de déclencher un contrat de composition sur l'instance active, revient à demander à l'instance passive d'une même paire, de se synchroniser comme défini par le contrat.

Dans notre implémentation, nous sommes basés sur la machine à objets étendus – MOE [DES02] dans l'objectif d'ajouter des sémantiques additionnelles à une classe. Cette

machine permet d'ajouter, de manière simple et dynamique, des attributs à une classe ainsi qu'à des instances. Elle permet aussi d'instrumenter l'exécution des méthodes afin d'ajouter dynamiquement des contrats de composition. Cette machine sera présentée dans le chapitre 6.

4.3.2. Contrats de composition

Un **contrat de composition** représente un accord entre deux concepts, concernés par une relation, ayant pour effet de garder la synchronisation entre eux. Un contrat de composition permet d'exprimer la manière de synchroniser l'instance du concept passif lorsqu'il y a un changement d'état de l'instance du concept actif d'une même paire.

Un contrat de composition a des caractéristiques qui sont assez similaires aux contrats de coordination présentés dans le chapitre précédent :

- Il est attaché à une méthode de classe active. L'appel de cette méthode est considéré comme la condition de déclenchement du contrat ;
- Il a des participants qui sont les instances provenant d'une même paire ;
- Il a un contexte d'exécution qui regroupe les informations concernant l'état d'exécution du domaine actif ;
- Il peut échouer ou réussir. L'échec d'un contrat de composition signifie vraiment une erreur. Due à cette erreur, les domaines concernés n'arrivent pas à se coordonner. A l'heure actuelle, nous n'avons pas encore étudié un mécanisme de récupération lorsqu'un contrat de composition échoue.

Les contrats de composition présentent quelques différences avec les contrats de coordination. Le tableau suivant montre ces différences.

Contrat de coordination	Contrat de composition
Contrat entre participants d'un même domaine.	Contrat entre deux concepts provenant des modèles conceptuels des domaines à composer.
Composition verticale, de l'abstrait au concret, au sein d'un domaine.	Composition, à un même niveau d'abstraction, des domaines.
On appèle ce type de contrat les **contrats verticaux**.	On appèle ce type de contrat les **contrats horizontaux**.

Tableau 4. Contrats de composition vs. contrats de coordination

Du point de vue pratique, on s'aperçoit que les contrats horizontaux et verticaux sont différents seulement dans la localisation de leurs participants. C'est pour cela que nous avons utilisé le même modèle pour représenter les contrats horizontaux et verticaux. Nous avons également utilisé le même langage pour exprimer le corps des contrats horizontaux et ceux des contrats verticaux.

A titre d'exemple, nous présentons le corps du contrat horizontal, nommé `data_Document` dans la Figure 67. Ce contrat a pour objectif d'établir les paires d'instances pour la relation entre `Data` et `Document` ; il est installé sur le constructeur de la classe `Data`.

Avant de progresser dans l'explication de ce contrat, il est important de noter que, dans notre modèle, tous les contrats installés sur les constructeurs sont déclenchés après que les constructeurs soient bien exécutés. Par conséquent, l'instance de la classe active est déjà créée. Appelons `ObjD` l'instance de `Data`, qui vient d'être crée avant de lancer ce contrat, ce dernier est expliqué comme suit :

- La ligne 1 déclare le contrat avec un nom, une liste de paramètres qui sont des paramètres de la méthode interceptée, et la classe interceptée (i.e. `Data`) ;
- La ligne 2 commence le corps du contrat, ce qui est exprimé par notre langage textuel basé sur Java ;
- La ligne 3 montre la localisation du *contrôleur de l'UC* du domaine de document en consultant le registre de nom de l'UC et en donnant le nom enregistré du *contrôleur*. Notons qu'afin de localiser des instances dans l'UC d'un domaine, nous nous basons sur une hypothèse forte : tous les domaines doivent fournir un **contrôleur de l'UC** dont l'objectif est de permettre de naviguer et de localiser des instances dans l'UC ;
- La ligne 4 demande au *contrôleur* de fournir une instance de `Document` ayant le même nom que l'instance `ObjD`. Notons que, sur cette relation, on est basé sur une contrainte que le nom de `Data` est le même que celui de `Document` ;
- Les lignes 5, 6 et 7 ajoutent dynamiquement un attribut dans `ObjD` pour représenter l'instance de `Document` à laquelle elle est liée. Notons que le mot *'instance'* est un mot clé dans notre langage de contrat, il est utilisé pour désigner l'objet intercepté (i.e. l'instance `ObjD`).

```
1:    contrat data_document ( String dataName ) of Data
      {
2:          body( JAVA )
            {
3:                DocumentDomain pd = ( DocumentDomain )commonUniverse.getRoot( "documentRoot" );
4:                Document doc = pd.getDocument (dataName);

5:                IEO ieo = ( IEO )instance;
6:                ieo.addAttr( "document" );
7:                ieo.setAttr( "document", doc );
            }
      }
```

Figure 67. Contrat horizontal `'data_document'`

Nous avons construit un éditeur dans notre environnement afin de supporter la définition des relations entre des concepts. La Figure 68 montre une image de cet éditeur ouvrant la composition entre trois domaines : procédé, ressource et document (cette information est montrée dans l'onglet *'Général'* de l'éditeur). La relation sélectionnée est ici entre `'Data'` et `'Document'`. Le contrat `'data_document'` (i.e. l'aspect dans l'image) est

attaché sur le constructeur de la classe `'Data'`. Ce contrat s'exécutee après l'exécution de ce constructeur.

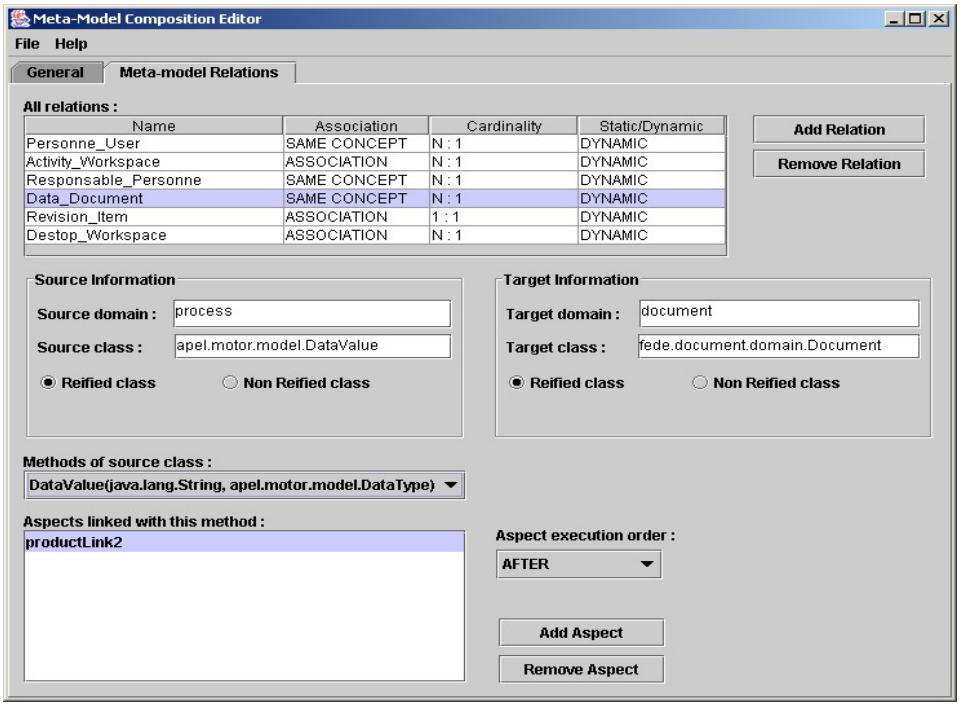

Figure 68. Définition des relations par l'éditeur

4.3.3. Participants réalisant des relations

Les contrats horizontaux représentent un bon moyen pour gérer les relations. Néanmoins, ils ne sons pas suffisants pour traiter les cas où la gestion de la relation nécessite une interaction d'une ou plusieurs personnes travaillant sur des postes différentes.

Dans ce cas, nous proposons de nous appuyer sur les participants distants et interactifs (i.e. un outil ou bien une application) pour implémenter la partie de relation liée à l'interaction humaine. Par conséquent, les contrats horizontaux gèrent des relations en s'appuyant sur ces participants.

En comparant avec des participants réalisant des domaines, ces participants ont des caractéristiques intéressantes :

- Ils connaissent les domaines composés et la sémantique des relations qu'ils gèrent. Ceci parce qu'ils sont crées afin d'implémenter la sémantique des relations ;
- Ils ne peuvent pas être autonomes ; ils doivent s'exécuter en collaborant avec des domaines composés ;
- Ils sont spécifiques à une composition de domaines particulière. En effet, puisqu'ils sont crées pour implémenter une relation, ils ne peuvent être réutilisés que dans les compositions où cette relation existe.

A titre d'exemple, on peut considérer la gestion de la relation entre deux concepts `User` et `Humain` venant respectivement du domaine de procédé et de ressource. Il s'agit d'une relation « identique » de cardinalité 1 : 1 qui a pour objectif d'assigner le responsable pour chaque activité dans le processus de production d'un document.

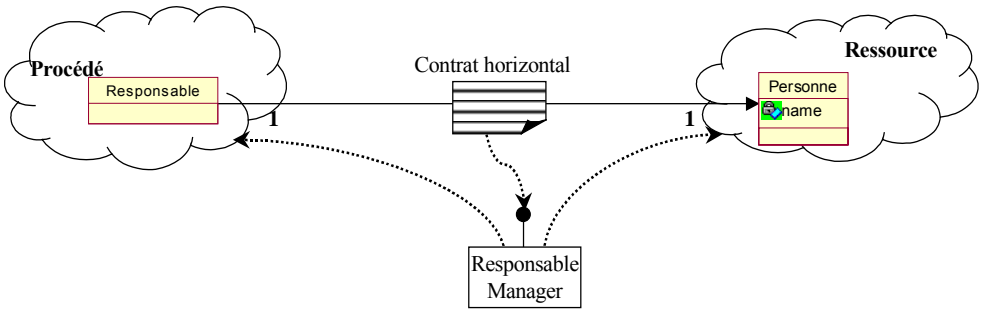

Figure 69. Participant réalisant une relation

Pour chaque activité, on souhaite que l'assignation du responsable soit faite par le responsable de l'activité englobante. Autrement dit, celle-ci doit être faite de manière interactive et distribuée selon le poste du responsable de l'activité englobante. Pour ce faire, nous avons construit un outil, nommé `'Responsable Manager'`. Cet outil est capable d'identifier les personnes disponibles dans le domaine de ressource et de proposer au responsable de l'activité englobante de choisir une personne convenable pour être le responsable d'une activité.

4.4. Domaine émergeant

Dans cette section, nous étudions les concepts émergeants comme étant une solution pour : (1) ajouter une signification conceptuelle aux relations et (2) transformer une relation n-binaire (i.e. relation entre plusieurs concepts) en plusieurs relations binaires (i.e. relation entre deux concepts).

En réalité, dans certaines situations, on souhaite ajouter une sémantique additionnelle à une relation en lui ajoutant des attributs, des politiques de gestion spécifiques, etc. D'une part, il n'est pas simple d'implémenter la sémantique additionnelle en utilisant les contrats et/ou les participants, d'autre part, le fait d'implémenter la sémantique dans les contrats horizontaux et/ou les participants peut disperser cette sémantique. Ceci provoque les difficultés pour maintenir et faire évoluer cette sémantique.

Nous pensons que, dans ce cas, il est préférable de représenter explicitement un nouveau concept qui émerge de la relation. Nous appelons les concepts de ce type comme des *concepts émergeants*. Le fait de faire évoluer la sémantique d'une relation correspond à faire évoluer le concept émergeant de cette relation. Les concepts émergeants peuvent être comparés aux classes d'associations du UML.

Avec un concept émergeant Cr, une relation R, entre deux concepts C1 et C2, se traduit en la relation R1, entre C1 et Cr, et la relation R2, entre Cr et C2. Indépendamment du fait

qu'un concept soit émargeant ou non, la définition et la gestion d'une relation entre ce concept et les autres concepts respectent le principe présenté dans ce chapitre.

Par exemple, considérons la relation entre `Responsable` et `Personne`, par l'ajout du concept `Assign`, nous pouvons représenter explicitement la stratégie d'assignation de ressource dans ce concept. Désormais, la relation entre `Responsable` et `Personne` se traduit par les relations entre `Responsable` et `Assign` et `Assign` et `Personne`.

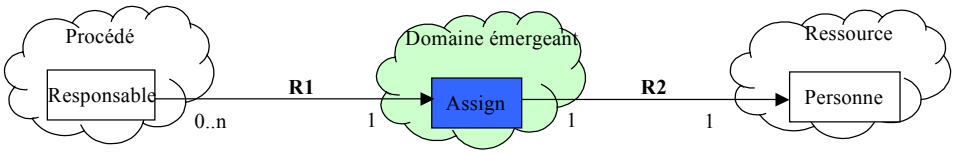

Figure 70. Concept émergeant

De la même manière, nous pouvons utiliser un concept émergeant pour transformer une relation n-aire en des relations binaires. Pour cela, nous introduisons un concept émergeant pour exprimer la sémantique de la relation entre les concepts concernés. Pour chacun de ces concepts, nous pouvons établir une relation avec le concept émergeant. Ainsi, ces concepts se synchronisent à travers le concept émergeant. Ce dernier sert d'intermédiaire entre les concepts concernés par la relation n-aire.

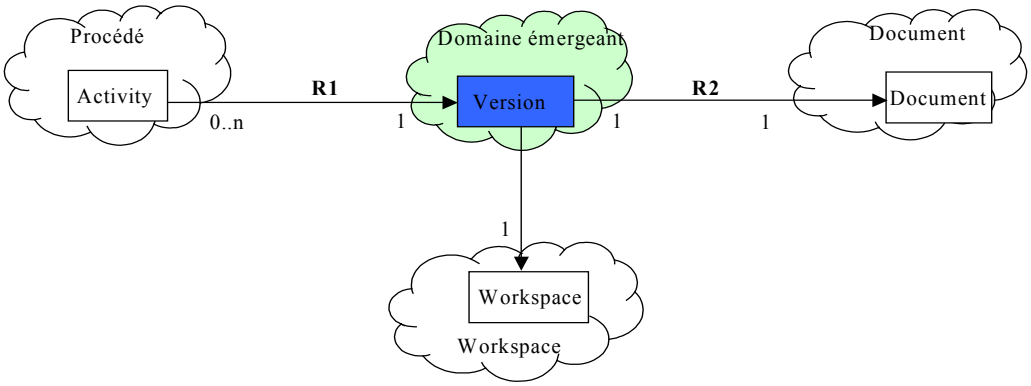

Figure 71. Transformation une relation n-aire en relations binaires

Par exemple, imaginons que nous voulions définir des stratégies de versionnement pour un document dans un espace de travail. Or, les stratégies de versionnement sont liées aux trois concepts `Activity`, `Document` et `Workspace` provenant respectivement du domaine de procédé, de document et d'espace de travail. Pour cela, nous avons utilisé le concept de `Version` pour exprimer les stratégies de versionnement en se basant sur ces trois concepts comme le montre la Figure 71.

Nous proposons d'introduire le concept de domaine émergeant pour maintenir l'ensemble des concepts émergeant. Un domaine émergeant est très spécifique d'une composition ; il est donc impossible de réutiliser le domaine émergeant en dehors de cette composition.

Comme tous les autres domaines, un domaine émergeant peut avoir ses propres participants qui ont pour objectif de rendre concrète la sémantique des concepts

émergeants. Il est évident que ces participants dépendent fortement de la logique de l'application à construire. Ce ne sont pas des participants réutilisables.

Les participants, qui réalisent une relation (cf. section 4.3.3), peuvent être aussi considérés comme étant les participants du domaine émergeants. Ces participants connaissent tous les domaines et ils communiquent éventuellement avec les domaines afin d'atteindre leurs objectifs.

4.5. Vers un langage de haut niveau de description de relations

Par l'instrumentation du code des UCs des domaines actifs, nous sommes capables de tisser les contrats horizontaux servant à gérer les relations entre des concepts provenant des domaines composés. Ceci nous permet de tisser indirectement ces domaines.

Néanmoins, il n'est pas très simple de définir des relations en utilisant directement des contrats horizontaux. Comme la sémantique d'une relation est dispersée dans plusieurs contrats horizontaux, nous n'avons pas une vision globale de la gestion de cette relation.

De cette manière, il est nécessaire de remonter le niveau d'abstraction en cachant le plus possible les détails concernant l'implémentation d'une relation. Pour cela, nous tentons de construire un langage de haut niveau de définition de relation. L'objectif de ce langage est de simplifier la spécification des relations en regroupant les informations liées à la structure (i.e. cardinalité et orientation) et la sémantique d'une relation (i.e. les contrats horizontaux et les participants) ;

A partir de la description d'une relation par ce langage, une grande partie des contrats horizontaux seront générés et attachés aux méthodes des classes de l'UC du domaine actif. Cette partie est liée à la localisation des *contrôleurs de l'UC* des domaines composés ; la localisation des instances d'une paire, l'établissement du lien physique de l'instance active vers l'instance passive, etc. Ceci permet de faciliter le travail des personnes qui définissent la sémantique des relations.

Pour l'instant, la grammaire de ce langage est la suivante :

<relation> ::= relation <name> **:** <cardinality> "{"
[<instances>] [<constraints>] [<connection>][<coordination>] "}"
<cardinality> ::= 1-1 | 1-N | N-1 | N-N
<instances> ::= instances "{"
source "**:**" <class> of <domainSource>
destination "**:**" <class> of <domainDestination> "}"

<class> désigne la classe qui participe à la relation ; <domaineSource> et <domaineDestination> désignent respectivement le domaine source et destination concernés par la relation.

Des contraintes peuvent être définies sur les relations afin de limiter les instances candidates lors de l'établissement des groupes d'instances concernées par la relation.

<constraints> ::= constraints "{" (<expressions>)* "}"

La partie connexion spécifie le moment et ce qu'il faut faire pour localiser et établir des liens entre les instances concernées par la relation.

<connection> :: = connection "{" (<contrat>)* "}"

La partie coordination désigne la synchronisation des instances d'une paire au long du cycle de vie de l'instance du concept actif. Nous avons distingué la partie coordination et la partie connexion dans l'objectif de faciliter la génération des contrats horizontaux. Pour les contrats qui assurent la connexion, il est nécessaire de générer le code qui créer des liens physiques entre instances en se basant sur la machine à objets étendus – MOE.

<coordination> :: = coordination "{" (<contrat>)* "}"

Les contrats sont définis en se basant sur le langage de contrat (cf. annexe A). Cependant, la déclaration de contrats est un peu différente. Ceci est dû au fait que la classe à laquelle on attache le contrat est, implicitement, la classe source.

< contrat > ::= <type> <name> "(" <parameters> ")" "{"
[<trigger>] [<participant>] <body> "}"
<type> ::= BEFORE | AFTER | AROUND

Dans le corps du contrat, de nouveaux mots clefs sont disponibles :

- source : pour désigner l'instance source liée par la relation ;
- destination : pour désigner l'instance destination liée par la relation ;
- controleurSource : pour désigner le *contrôleur* de l'UC du domaine source ;
- controleurDestination : pour désigner le *contrôleur* de l'UC du domaine destination.

A titre d'exemple, la Figure 72 nous montre la description de la relation entre `Data` et `Document`. Le contrat horizontal pour établir une paire d'instance, que nous avons montré dans la Figure 67, est également exprimé dans la partie connexion.

```
relation Data_Product : N – 1 {
        instances {
                source : apel.motor.model.Data of process ;
                destination : fede.document.domain.Document of document ;
        }
        constraints {
                source.name = destination.name ;
        }
        connexion {
                AFTER Data( z: java.lang.String ) {
                        body(JAVA) {
                                destination = controleurDestination.getDocument ( z );
                        }
                }
        }
}
```

Figure 72. Exemple d'une relation avec le langage de relation

Le langage de description de relation permet d'exprimer plus d'information concernant une relation que l'éditeur montré dans la Figure 68. En plus, il facilite la définition des contrats en générant une partie des contrats horizontaux. Il est donc nécessaire de s'orienter vers ce langage.

Au moment de mettre sous presse, nous sommes en train de construire le compilateur et l'éditeur pour travailler avec des relations définies par ce langage.

5. Réutilisation et adaptation d'un domaine

La réutilisation d'un domaine est très intéressante ; elle permet de réutiliser non seulement un élément logiciel mais un groupe d'éléments logiciels avec des fonctionnalités de haut niveau résultées par la coordination d'éléments logiciels du groupe.

Nous pensons qu'un domaine peut être réutilisé dans différentes applications globales, mais éventuellement avec des adaptations afin de bien s'adapter aux besoins fonctionnels et, en particulier, aux besoins non-fonctionnels, tels que la distribution, la sécurité, la transaction, etc. de ces applications.

Par exemple, lorsque l'application est distribuée et que le domaine ne supporte pas la distribution, il est nécessaire d'adapter le domaine en ajoutant un participant distant (cf. chapitre 3). De la même manière, la sécurité peut impliquer de changer un participant du domaine par un autre afin de pouvoir répondre aux besoins de l'application à propos de la sécurité d'information.

5.1. Adaptation d'un domaine

Nous avons distingué trois possibilités d'adapter un domaine :

- Adaptation de réalisation, qui consiste à changer la réalisation du domaine en respectant la logique du domaine (cf. section 5.1.1) ;
- Adaptation horizontale, qui consiste à adapter et/ou étendre la logique d'un domaine en modifiant légèrement son modèle conceptuel (cf. section 5.1.2) ;
- Adaptation variante, qui consiste à varier le comportement du domaine selon le contexte d'utilisation, mais sans changer sa logique et sa réalisation. Cette adaptation est très intéressante, mais difficile. Nous réservons le chapitre 5 pour l'étudier.

5.1.1. Adaptation de réalisation

Dans le sens vertical, un domaine peut être vu par une architecture à trois niveaux comme montre la Figure 73 :

- L'univers commun représente la sémantique abstraite du domaine ;
- Les exécutants représentent la réalisation concrète du domaine ;
- Les rôles représentent la réalisation d'abstraite du domaine dans laquelle les propriétés non-fonctionnelles ne sont pas prises en compte.

De cette manière, nous pouvons changer la réalisation à deux niveaux. Tout d'abord, nous pouvons changer seulement la partie réalisation concrète. A ce niveau, un exécutant peut être remplacé simplement par un autre à condition qu'il joue le même rôle. Ceci permet

d'utiliser des exécutants différents pour réaliser un domaine selon les préférences et les standards de chaque client. Par exemple, pour la gestion de versionnement des documents, une entreprise préfère l'outil CVS [CVS] tandis que l'autre veut RCS [RCS]. En fonction de l'usage du domaine pour ces deux entreprises, nous pouvons sélectionner CVS ou RCS pour gérer le versionnement des documents décrit par le rôle *'versionManager'*.

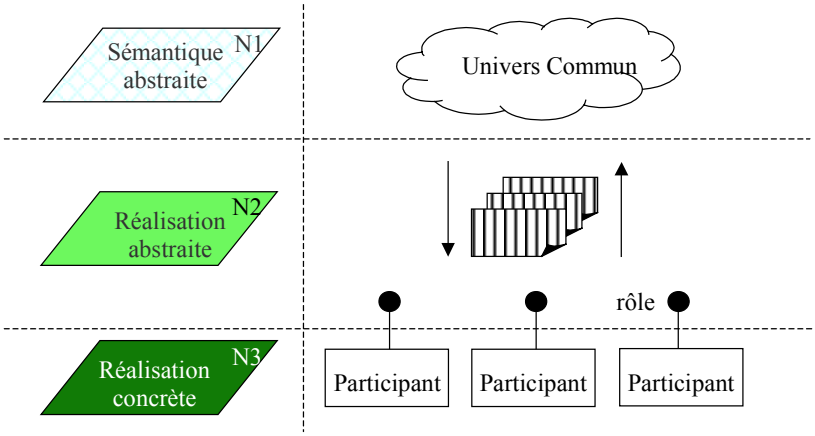

Figure 73. Architecture de trois niveaux d'un domaine

Nous pouvons également changer la réalisation abstraite et concrète d'un domaine. A ce niveau, nous ne gardons que la sémantique abstraite du domaine ; la manière de concrétiser cette sémantique est totalement substituée. Toutefois, cette adaptation est difficile ; elle doit être faite par un développeur des domaines.

5.1.2. Adaptation horizontale

Lorsque nous réutilisons un domaine dans une composition, il est possible que la logique de ce domaine ne corresponde pas exactement à ce que cette composition attend de ce domaine. Par conséquent, il est nécessaire d'adapter et/ou de modifier la logique du domaine. Ceci correspond au fait d'adapter et/ou de modifier le modèle du domaine.

Pour cela, il est nécessaire d'avoir des moyens permettant d'étendre le modèle conceptuel d'un domaine en y ajoutant de nouveaux concepts ou étendant la sémantique des concepts existants. Nous avons identifié, pour l'instant, deux options :

- Construire D1' comme étant une extension de D1 comme illustré dans la Figure 74. Cette option a besoin des relations spécifiques permettant d'ajouter ou d'étendre le modèle conceptuel d'un domaine en basant sur des relations.

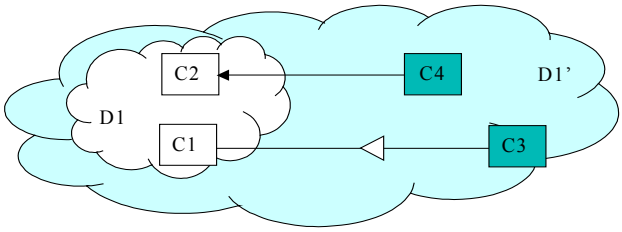

Figure 74. Extension un domaine

- Construire D1' comme étant la composition de D1 et D2 contenant tous les concepts à ajouter. La composition de D1 et D2 respecte le principe que nous avons étudié tout au long de ce chapitre.

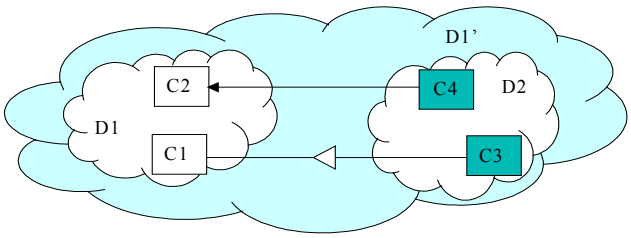

Figure 75. Extension un domaine

Nous pensons que l'adaptation conceptuelle d'un domaine a besoin des types de relations spécifiques et également de moyens permettant de structurer les domaines. Ceux-ci sont en cours d'élaboration lors de la rédaction de cette thèse.

Il est important de noter que l'adaptation horizontale d'un domaine implique souvent une adaptation verticale car il est nécessaire d'ajouter la concrétisation pour les concepts ajoutés ou d'enlever la concrétisation liée aux concepts retirés.

5.2. Gestion de configuration d'une composition de domaines

Il y a plusieurs manières d'utiliser et d'adapter un domaine. Par conséquent, un domaine peut avoir plusieurs modèles conceptuels et plusieurs réalisations. Ces réalisations permettent d'adapter le domaine aux usages spécifiques ou aux standards et préférences des clients.

Ainsi, lors de l'usage d'une composition de domaines chez un client, il est nécessaire d'avoir un moyen de spécifier explicitement la réalisation des domaines composés qui sont convenables pour ce client. Pour cela, nous proposons d'utiliser le concept de configuration pour spécifier explicitement les éléments provenant de chaque domaine.

La configuration d'une composition de domaines contient :

- Un modèle conceptuel pour chaque domaine et les relations entre les classes provenant des modèles des domaines ;
- Un ensemble d'éléments implémentant chaque domaine. Il s'agit des rôles, des exécutants, des contrats verticaux ;
- Un ensemble d'éléments pour concrétiser les relations entre les domaines. Il s'agit des contrats horizontaux, les participants réalisant les relations, etc.

Une composition de domaines peut avoir plusieurs configurations. Des configurations différentes peuvent avoir les comportements différents. En effet, le fait de choisir les participants détermine la manière de rendre concrète la sémantique du domaine. Le fait de

sélectionner les contrats verticaux différents décide des différentes manières de synchroniser l'état du modèle et l'état des participants.

Pour l'instant, nous définissons les règles assez simples pour déterminer si une configuration peut être considérée comme cohérente:

- Tous les rôles référencés par les contrats verticaux et horizontaux sont présents dans la configuration ;
- Tous les rôles utilisés doivent être implémentés ;
- Tous les rôles simples (cf. chapitre 3) ne doivent être implémentés que par un seul participant.

Nous avons construit un outil graphique pour faciliter la définition des configurations d'une fédération de domaines. Nous présentons cet outil dans l'annexe A.

6. Discussions et conclusions

6.1. Fédération de domaines

Tout au long de ce chapitre, nous avons étudié la composition de domaines à travers la composition de modèles exécutables de ces domaines. Celle-ci propose une manière de structurer et de réaliser des applications grande taille en facilitant l'évolution et l'adaptation de cette application.

D'une certaine manière, la composition de domaines se base sur l'architecture de fédération que nous avons étudiée dans le chapitre 3. Une composition de domaines contient les éléments d'une fédération comme suit :

- Un ensemble de participants qui sont des domaines composés. Chaque domaine est défini en terme d'un univers commun, d'un ensemble de participants et d'un ensemble de contrats verticaux. Ces derniers représentent les règles pour que les participants se coordonnent autour de l'univers commun dans l'objectif de réaliser le domaine ;
- Un univers commun – UC – qui contient l'UC des domaines composés, les concepts émergeants et les relations établies entre les concepts provenant des domaines composés ;
- Un ensemble de contrats horizontaux qui gèrent les relations entre concepts venant des domaines composés ;
- Et éventuellement, un ensemble de participants qui implémentent les relations entre concepts.

De cette manière, nous pouvons considérer une composition de domaines comme étant une fédération de domaines.

Grâce à la gestion de configuration (cf. section 5.2), nous pouvons construire différentes configurations d'exécution pour une même composition de domaines. Ceci nous permet de varier le comportement d'une fédération de domaines.

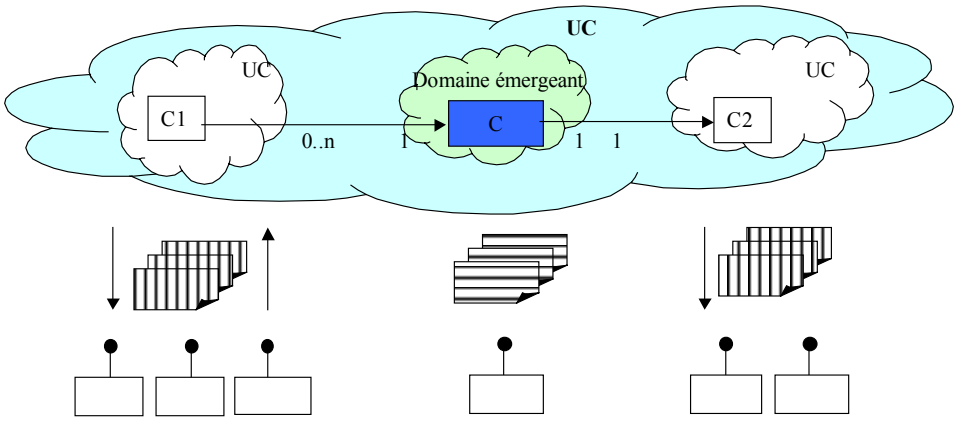

Figure 76. Une fédération de domaines

6.2. Fédération et les autres approches

Notre approche de composition a été conçue en séparant des préoccupations architecturales. Ceci permet de recevoir une palette d'architecture qui peut s'adapter aisément aux divers besoins et aux divers domaines d'activité. En fonction de la complexité du problème à résoudre, nous pouvons nous appuyer sur une fédération de participants ou bien sur une fédération de domaines.

Il est intéressant de noter que notre architecture de fédération peut supporter les mécanismes de composition que nous avons étudiés dans le chapitre 2. Nous présentons brièvement dans la suite la manière d'utiliser notre architecture.

6.2.1. Fédération et des systèmes à composants

Notre objectif n'est pas de concevoir un modèle de composant particulier, ni de construire un système à composant. Cependant, dans l'objectif de fournir une vue uniforme et homogène des exécutants de diverses natures, nous avons construit un modèle de participant qui est, sans doute, un modèle de composant.

Notre environnement propose également une machine virtuelle visant à supporter la gestion des composants. Cette machine propose des services pour gérer : (1) le cycle de vie des instances de composants, (2) la distribution des instances de composants, (3) la localisation des instances d'un composant avec des caractéristiques souhaitée, (4) la récupération d'erreur, etc.

Dans notre environnement, les composants ont deux manières de communiquer :

- Premièrement, ils ne se connaissent pas et ils communiquent indirectement, par coordination, grâce à la fédération. Ceci est « traditionnel » dans notre approche ;

- Deuxièmement, ils se connaissent et ils peuvent utiliser les services de notre environnement pour localiser un participant particulier et pour se connecter avec lui. Cependant, dans ce cas, la fédération ne contrôle pas les interconnexions directes entre des participants.

Dans le chapitre 6, nous allons étudier en détail la gestion de composants fournis par notre environnement. Pour conclure, nous pouvons dire qu'il est possible de réaliser des applications à base de composants distribuées dans notre environnement de support.

6.2.2. Fédération et l'AOP

Concernant le fait d'étendre la sémantique d'un programme de référence, notre architecture de fédération propose un cadre conceptuel à l'AOP dans lequel l'extension peut être faite selon deux dimensions : verticale et horizontale. La Figure 77 montre ces deux dimensions d'extension.

Le programme de référence sera considéré comme étant l'UC du domaine principal. D'une part, sa sémantique sera complétée, par l'extension verticale, en s'appuyant sur des participants de ce domaine. D'autre part, sa sémantique pourra être complétée, par l'extension horizontale, en le composant avec d'autres domaines.

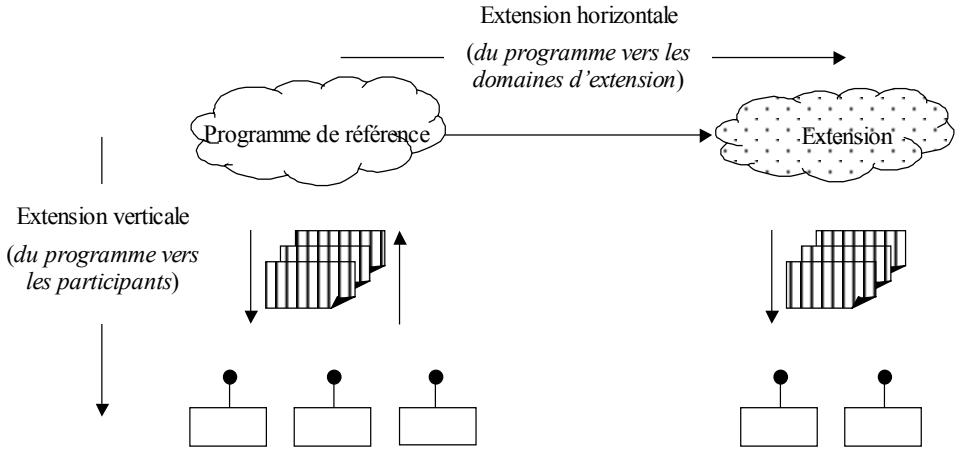

Figure 77. Extension verticale et horizontale

La dimension verticale représente une extension de l'abstrait vers le concret dans laquelle les participants sont utilisés pour concrétiser et/ou étendre la sémantique du programme de référence, mais sans ajouter de nouveaux concepts à l'espace conceptuel de ce programme. Cette dimension représente ce que les systèmes de POA font en général.

L'extension horizontale considère la composition d'un ou de plusieurs domaines avec le domaine principal. Ceci permet d'étendre et d'ajouter de nouveaux concepts et de nouvelles fonctionnalités au programme qui est le résultat de cette composition.

Du point de vue de la séparation, nous pouvons dire que notre architecture propose une manière de :

- Travailler avec des préoccupations conceptuelles dont chacune est réalisée par un domaine ;
- Tisser les préoccupations conceptuelles pour construire une application globale.

6.2.3. Fédération et le BPM

Notre approche de fédération partage avec BPM (*Business Process Management*), l'idée de considérer la coordination comme le moyen de composer les participants. Cependant, dans notre approche, nous avons séparé le coordinateur et le contrôleur du but dans l'objectif d'offrir plus de flexibilité et de dynamicité.

Avec notre approche, nous répondons à plusieurs questions que le BPM cherche à résoudre. Ceci peut être constaté sur les trois points suivant :

- L'interopérabilité des WfMSs monolithiques ;
- La construction des WfMS extensibles ;
- La construction d'un système BPM extensible et flexible.

6.2.3.1. Interopérabilité des WfMSs

L'interopérabilité des WfMSs monolithiques est facilement atteint en construisant une fédération de procédé dans laquelle ces WfMSs sont les participants. Le modèle conceptuel du domaine de procédé est utilisé comme étant l'UC de cette fédération ; des contrats verticaux vont coordonner des WfMSs dans l'objectif de rendre concret et de compléter la sémantique de ce modèle.

De cette manière, les WfMS monolithiques peuvent être coordonnés indirectement bien qu'ils ne se connaissent pas. Ceci permet d'éviter le couplage et la dépendance entre eux. En plus, il n'est pas obligatoire qu'ils soient basés sur un même standard pour représenter et pour exécuter ensemble des modèles de procédé.

6.2.3.2. Construction des WfMS extensibles, non-monolithiques

Actuellement, la plupart des WfMS sont monolithiques. Le fait qu'ils soient monolithiques réduit la flexibilité, l'ouverture et l'adaptation de ces systèmes.

Avec notre approche, il est assez simple de réaliser un WfMS comme le résultat de la composition de plusieurs éléments logiciels, existants ou pas. Le modèle conceptuel du domaine de procédé sera utilisé comme étant l'UC de cette fédération. Cet UC contient, en effet, une sémantique de base de ce WfMS. Plusieurs outils utilisés pour compléter cette sémantique peuvent être, entre autres :

- Les agendas représentant le moyen qui permet à un utilisateur de participer dans le procédé d'entreprise ;
- Le monitoring pour observer et tracer l'exécution des procédés d'entreprise ;
- Le moteur qui interprète et gère des instances de procédé d'entreprise qui sont en exécution ;

Les outils à utiliser peuvent être composés au fur et à mesure et en fonction des besoins fonctionnels de ce WfMS. La manière dont ces outils complètent la sémantique de l'UC sera faite grâce aux contrats verticaux.

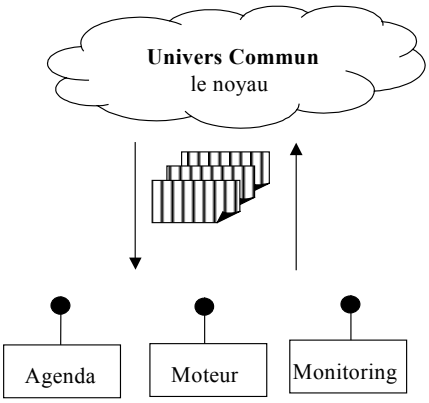

Figure 78. Un WfMSs extensible

6.2.3.3. Construction d'un système BPM extensible et flexible

Par la composition de domaines, on peut construire un système BPM très extensible et flexible. Un WfMS, ouvert et extensible, est construit comme étant un domaine. Les participants qui réalisent le procédé d'entreprise appartiennent au domaine d'application qui contient des concepts liés au métier de l'entreprise.

A travers le domaine d'application, on a une vision claire de ce que le procédé d'entreprise tente de réaliser. La composition du domaine de procédé et celui de l'application permet au domaine de procédé de contrôler et de guider le domaine d'application, mais indirectement. Le couplage entre le procédé et l'ensemble des participants qui réalisent le procédé d'entreprise est alors minimisé.

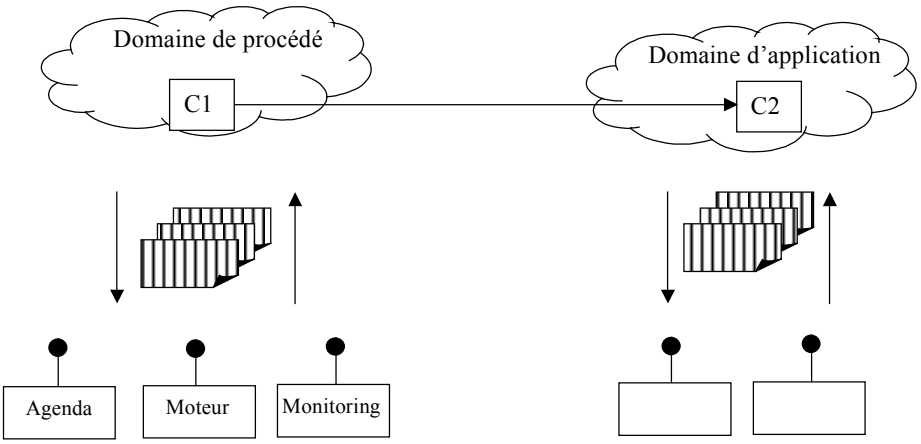

Figure 79. Système BPM avec des domaines séparés

Il est important de noter que l'on peut construire un ou plusieurs domaines d'application, qui sont liés au métier du procédé d'entreprise, selon leur activité métier.

6.2.4. Fédération et le MDA

Notre approche n'est pas née pour résoudre des problématiques de MDA. Au contraire, comme présenté dans le chapitre 1, nous avons pour objectif de composer les éléments logiciels de diverses natures afin de construire une nouvelle application. Pourtant, dans le processus de recherche d'une solution, nous sommes persuadés qu'il faut remonter à un niveau d'abstraction plus élevé pour composer des éléments logiciels. Il est clair qu'il s'agit de la direction du MDA.

Malgré le MDA et notre approche partagent l'idée de travailler au niveau d'abstraction plus haut que le code, la solution du MDA et notre solution sont différentes. Le MDA se base sur la transformation des modèles jusqu'à que l'on obtienne du code exécutable. Par contre, nous sommes basées sur l'usage de modèles exécutables, d'une part pour coordonner des participants et d'autre part, pour composer des domaines à un haut niveau d'abstraction. Nous utilisons une opération de *projection* plutôt qu'une opération de *transformation* de modèles.

Nous pensons que ce qui diffère le plus entre le MDA et notre approche, est que nous nous concentrons sur l'architecture des applications résultantes. Ceci n'est pas encore abordé dans l'approche MDA.

6.3. Résumés de notre approche

Pour résumer notre architecture, nous respectons les mêmes points que nous avons énoncé au début du chapitre 2 :

- **Eléments de composition** : Nous nous intéressons à des éléments logiciels de diverses natures, surtout des éléments existants ;

- **Manière de composer** : Nous sommes basés sur la composition, par coordination et dirigée par le modèle conceptuel, pour composer les participants au sein d'un domaine et la composition de modèles conceptuels exécutables pour composer des domaines ;

- **Couplage entre les éléments à composer** : Les participants sont indépendants. Les domaines eux aussi sont indépendants les uns des autres ;

- **Adaptation et évolution de l'application résultante** : En s'appuyant sur une composition au niveau modèle conceptuel, l'adaptation et l'évolution des domaines sont quasiment transparentes à la composition de domaines ;

- **Réutilisation** : Tout d'abord, nous pouvons réutiliser un élément logiciel de nature quelconque à condition qu'il soit possible de construire l'adaptateur pour cet élément. En plus, par la réutilisation d'un domaine, nous pouvons réutiliser non seulement un élément logiciel séparé, mais aussi un groupe d'éléments avec les fonctionnalités, de haut niveau d'abstraction, résultées par la coordination des éléments au sein de ce groupe.

6.4. Conclusions

Les problématiques liés à la composition d'éléments logiciels dans l'objectif de construire une nouvelle application ont été étudiées dans notre équipe depuis plusieurs années [DEA98, Ami99, Ver01, Vil03]. C'est grâce à beaucoup d'études et d'expérimentations réelles que nous avons identifiées ce qui peut être une bonne solution pour ces problématiques. Ceci nous a permet de proposer la solution présentée dans ce document.

Dans le chapitre suivant nous allons étudier une manière de construire des domaines génériques. De cette manière, un domaine propose non seulement une solution pour un problème concret dans un contexte particulier, mais aussi une solution pour un problème commun dans des contextes d'utilisation similaires.

Chapitre V

Construction des domaines génériques

1. Introduction

Dans le chapitre précédent, nous avons étudié le concept de domaine et la composition de domaines. Ceux-ci nous offrent une manière efficace pour la structuration et la construction des applications de grande taille ainsi que la réutilisation des domaines.

Ce chapitre va terminer la deuxième partie de cette thèse en considérant la troisième dimension d'adaptation d'un domaine. Il s'agit de la dimension variante qui a pour objectif d'adapter, de manière simple, un domaine pour qu'il puisse être utilisé dans différents contextes d'utilisation.

La section 2 commence ce chapitre en discutant nos attentes à propos d'un domaine générique. Ensuite, nous étudions rapidement la généricité proposé par les *frameworks* de type boîte blanche [Mat00] et les *templates* [Lan03], leurs avantages et leurs inconvénients. Enfin, nous présentons notre manière de construire des domaines génériques paramétrés qui est, à notre avis, la plus adaptée à nos objectifs. Dans la section 3, nous étudions la composition de domaines qui peuvent être spécifiques ou génériques. La section 4 montre l'environnement générique de gestion documentaire comme un exemple. Dans la section 5, nous mentionnons quelques discussions et conclusions.

2. Construction des domaines génériques

2.1. Domaines génériques

Dans le chapitre 4, nous avons étudié un domaine qui apporte une solution à un problème spécifique dans un contexte particulier. Nous appelons un tel domaine un *domaine spécifique*.

Par *domaine générique*, nous entendons un domaine qui peut résoudre un problème commun dans plusieurs contextes d'utilisation similaire. Un contexte d'utilisation est composé d'un ensemble de concepts et d'un ensemble de relations entre ces concepts ; nous considérons que deux contextes sont similaires lorsque leurs concepts sont de même nature.

Un domaine générique devient un domaine spécifique lorsqu'il est utilisé dans un contexte concret. Par conséquent, un domaine générique pourra être transformé en autant de domaines spécifiques qu'il y a de contextes d'utilisation différents.

A titre d'exemple, nous pouvons considérer le domaine de ressource ayant pour objectif de gérer des personnes. Les différents contextes d'utilisation de ce domaine sont caractérisés par les différents types de personnes à gérer. Le contexte d'une banque, par exemple, est composé des concepts de client, de guichetier, de technicien, etc. Le contexte d'un laboratoire informatique, à son tour, est constitué des concepts de responsable, de permanent, de secrétaire, de thésard, de stagiaire, etc.

Nous pensons qu'il doit être très simple à exprimer et/ou à ajouter un contexte d'utilisation, à domaine générique, même par un utilisateur final. Par conséquent, la construction d'un domaine générique a besoin de moyens permettant de :

- Caractériser ses contextes d'utilisation ainsi que son espace des contextes valides ;
- Exprimer facilement un contexte d'utilisation particulier dans cet espace ;
- Construire le domaine générique de sorte qu'il soit capable de travailler avec tous les contextes provenant de cet espace.

Il est intéressant de s'apercevoir qu'actuellement, il existe des approches qui tentent de construire des codes génériques. Nous allons regarder quelques approches représentatives dans la section suivante.

2.2. La programmation générique

La programmation générique, introduite par D. Musser et A. Stepanov, constitue un cadre détaillé et rigoureux pour l'écriture de code efficace et réutilisable dans une grande variété de contextes d'utilisation [MS88]. Les deux approches les plus connues sont :

- Les *frameworks* de type boîte blanche [Mat00] (cf. section 2.2.1) ;
- Les *templates* dans le langage C++ et UML [Lan03, DGD] (cf. section 2.2.2).

2.2.1. Les *frameworks* boîte blanche

"A framework is a set of classes that embodies an abstract design for solutions to a family of related problems" [JF88].

Un *framework* est constitué d'un ensemble de classes (pas nécessairement abstraites). Il n'est pas construit pour résoudre un problème concret, mais pour être étendu afin de traiter la famille des problèmes liée à un domaine. Pour cela, il est composé des éléments qui sont communs à domaine ainsi que les relations et les interactions entre ces éléments.

En se basant sur les techniques d'extensibilité des *frameworks*, on distingue différents types de *framework* [Mat00]. Parmi ces types, nous sommes intéressés par des *frameworks* de type boîte blanche qui se basent sur l'héritage des classes et/ou la redéfinition des méthodes du *framework* pour construire une application concrète.

Un domaine générique peut être réalisé comme étant un *framework* boîte blanche. La transformation de ce domaine générique en un domaine spécifique, dans un contexte particulier, demande de réaliser des classes qui constituent ce contexte et qui héritent et/ou spécialisent des classes du *framework*. La Figure 80 montre le domaine de gestion de ressource comme un *framework* boîte blanche et le domaine spécifique correspondant au contexte d'un laboratoire.

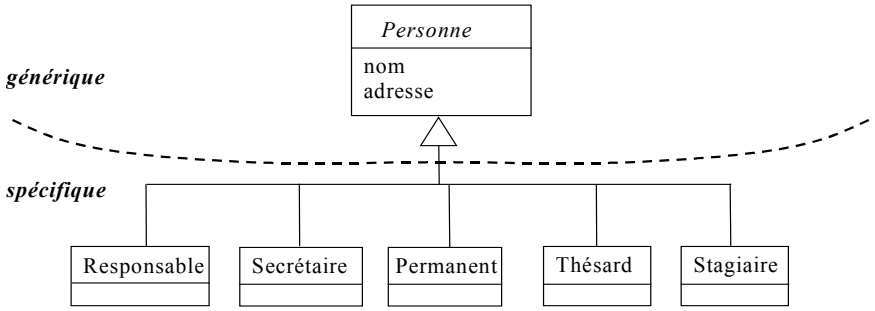

Figure 80. Domaine générique et domaine spécifique pour un laboratoire

De cette manière, l'héritage est le moyen de caractériser des contextes d'utilisation similaires pour un *framework*. Les concepts constituant ces contextes d'utilisation doivent être construits comme des sous-classes du *framework*. Ces concepts partagent ainsi la sémantique définie dans la super-classe.

L'héritage a un avantage : il est possible de spécifier et d'exécuter des codes très différents pour chaque concept en incluant sa sémantique spécialisée dans la sous-classe correspondante. Cependant, il est indispensable que les concepts avec leur sémantique, soient connus lors de la phase de codage.

Cependant, l'héritage représente également une limitation : si l'on veut utiliser le domaine générique dans un nouveau contexte, on est obligé de coder les classes constituant ce contexte, et puis de compiler et de déployer le domaine spécifique. Il est clair que ce travail doit être fait par un développeur qui connaît bien le domaine générique (i.e. le *framework*) ainsi que le domaine spécifique lié à ce nouveau contexte.

2.2.2. Les *templates*

Les *templates* sont proposés dans un certain nombre de langages de programmation dont le C++ [Lan03, DGD] et aussi dans UML [UML].

Il existe en pratique des situations où un même traitement peut être appliqué sur plusieurs types de données. Face à ces situations, l'idée de base des *templates* est d'écrire ce traitement comme étant une fonction générique pouvant prendre en paramètres le type de données sur lesquels ce traitement doit être appliqué.

En appliquant le principe des *templates*, les traitements communs à tous les types de personnes, du domaine de gestion des ressources, sont écrits dans le *template* `Personne`. En revanche, les traitements et les propriétés spécifiques de chaque type de personne sont

mis dans la classe correspondante. Le fait d'appliquer le *template* `Personne` aux classes représentant des types de personnes, nous permet d'obtenir le traitement générique de ce *template* sur ces types de personnes.

L'idée des *templates* est très intéressante par le fait qu'un traitement peut s'appliquer à un certain nombre de types qui sont connu lors de la phase de compilation. Néanmoins, puisqu'il n'est pas obligatoire que ces types soient de même nature, il n'existe pas de moyen de caractériser les types valides pour un *template*.

2.2.3. Discussions

Ces deux approches de la programmation générique sont intéressantes car elles permettent de construire des domaines génériques pouvant être utilisés dans des contextes d'utilisation différents. Cependant, par rapport à nos objectifs, ces deux approches ne conviennent pas car :

- D'une part, il faut travailler au niveau codage pour exprimer/modifier des contextes d'utilisation pour un domaine générique. Il est clair que ceci ne peut pas être fait par un utilisateur final ;
- D'autre part, il n'existe pas de moyen simple visant à caractériser des concepts constituant des contextes d'utilisation. L'héritage dans les *frameworks*, représente un moyen rudimentaire ; Il demande d'être fait par un développeur lors de l'implémentation des classes matérialisant des concepts.

De cette manière, ces deux approches ne peuvent pas être utilisées pour nos domaines génériques où les contextes sont très dynamiques et inconnus lors de la phase du codage. Pour ce faire, nous pensons qu'il est nécessaire de s'appuyer sur une couche située à un niveau d'abstraction plus élevée que les concepts constituant les contextes afin de pouvoir les caractériser et les construire facilement et dynamiquement. Nous étudions dans la suite notre manière de construire des domaines génériques.

2.3. Notre approche de programmation générique

2.3.1. Méta-modèle et contextes d'utilisation

En s'inspirant de la généricité des *templates*, notre approche consiste à rendre les contextes d'utilisation explicites, externes et nommés, pour un domaine générique. Ce dernier prend en paramètre un contexte d'utilisation particulier pour lequel la solution qu'il apporte peut être appliquée. Ceci nous permet d'obtenir un domaine spécifique pour le contexte d'utilisation donné en paramètre.

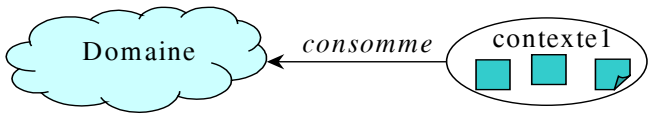

Figure 81. Domaine générique avec le contexte externe et nommé

Néanmoins, ce qui est différent avec les *templates* est que nous essayons de caractériser des contextes valides pour un domaine générique. Au lieu de se baser sur l'héritage comme les *frameworks*, nous proposons de remonter le niveau d'abstraction en nous appuyant sur le **méta-modèle** du domaine. Un contexte d'utilisation particulier sera instantié à partir du méta-modèle du domaine.

Dans [AK03], Atkinson et Kühne ont distingué deux types de méta-modèles : linguistique et ontologique. Un méta-modèle linguistique est défini comme étant le langage pour définir des modèles. En revanche, un méta-modèle ontologique s'intéresse à la sémantique d'un domaine ; il permet non seulement de distinguer un modèle d'un autre, mais aussi de définir les caractéristiques communes à tous les modèles issus de ce méta-modèle.

Avec cette définition, un méta-modèle ontologique n'est pas très loin d'un *framework* boîte blanche. Ils contiennent tous les deux des éléments communs à tous les modèles d'un domaine. Pourtant, les mécanismes d'extension pour obtenir un domaine spécifique, à partir des éléments communs, sont différents. L'un se base sur l'instantiation ontologique tandis que l'autre utilise l'héritage.

Nous prenons la définition des méta-modèles ontologiques pour désigner le méta-modèle de nos domaines. Pour nous, une grande partie de la construction d'un domaine générique consiste à travailler avec son méta-modèle. D'une part, ceci nous permet de rester sur les contraintes, les besoins, les caractéristiques, etc. qui sont communs à tous les contextes d'utilisation du domaine. D'autre part, le méta-modèle représente une contrainte importante pour déterminer les contextes valides pour le domaine.

Considérons le domaine de gestion des ressources. Son méta-modèle constitue le méta-concept `Personne`. Ce concept est matérialisé par la méta-classe `Personne` qui contient des informations permettant de caractériser et/ou distinguer différents types de personne. La Figure 82 montre deux instantiations ontologiques de ce méta-modèle pour construire les concepts constituant le contexte d'un laboratoire et d'une banque.

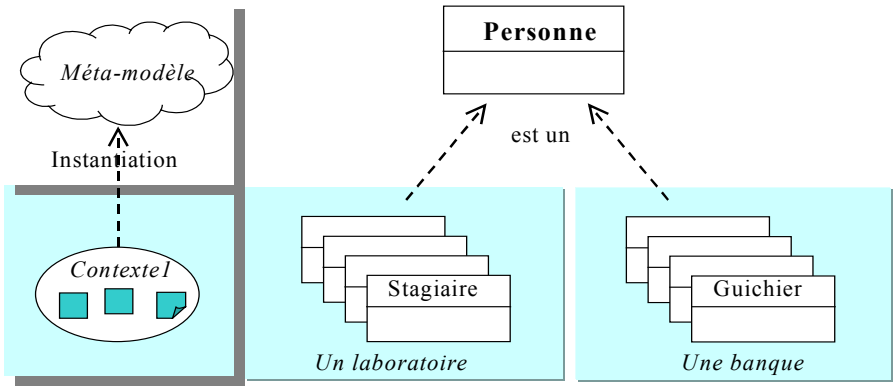

Figure 82. Deux instantiations ontologiques pour construire deux contextes d'utilisation

Dans l'objectif de faciliter la création des contextes d'utilisation par un utilisateur final, nous proposons de nous baser sur un éditeur graphique. Ce dernier se base sur le méta-modèle du domaine afin de produire des contextes d'utilisation pour ce domaine.

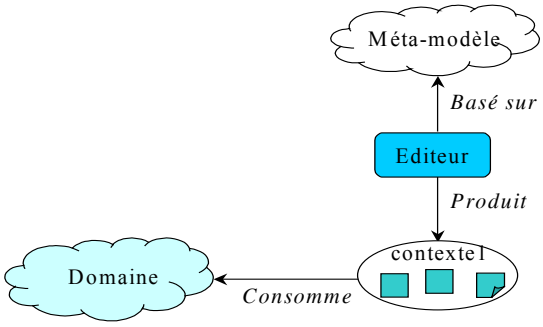

Figure 83. Editeur définit des contextes d'utilisation

L'éditeur peut aider également à vérifier des contextes d'utilisation valables pour ce domaine. Pour ce faire, il est certain que l'éditeur connaît bien la sémantique du domaine ainsi que son espace des contextes valide.

Nous avons identifié deux manières pour que l'éditeur produise un contexte d'utilisation pour un domaine :

- Générer des classes représentant les concepts constituant ce contexte d'utilisation (cf. section 2.3.2) ;
- Générer une description de ce contexte d'utilisation (cf. section 2.3.3) .

2.3.2. Génération des classes

L'éditeur peut générer des classes pour représenter des concepts constituant un contexte d'utilisation pour un domaine générique. Ainsi, un contexte d'utilisation est un groupe nommé comprenant des classes qui constituent ce contexte.

Le domaine générique prend en paramètre un contexte d'exécution en chargeant l'ensemble de classes correspondant à ce contexte. Ceci nous permet d'obtenir un domaine spécifique pour ce contexte.

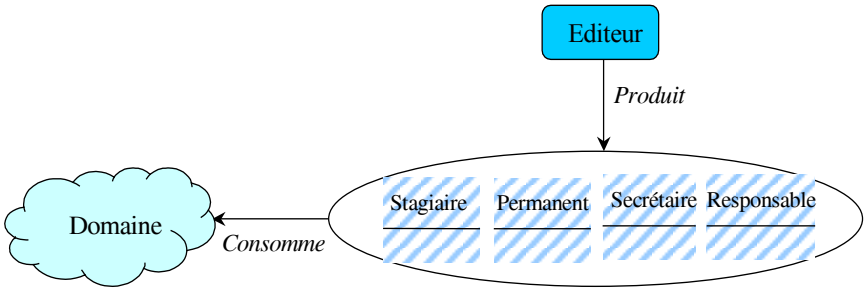

Figure 84. Génération des classes représentant le contexte

La Figure 84 montre le domaine de gestion de ressource avec des classes générées pour représenter les concepts trouvés dans un laboratoire. Bien que l'ensemble de ces classes ressemble beaucoup à celui de la Figure 80, ces deux ensembles de classes sont très différents. Par l'héritage dans des *frameworks* boîte blanche, chaque sous-classe peut contenir sa propre sémantique par rédéfinition des méthodes de la super-classe. Par contre,

la sémantique des classes générées est prédéfinie par l'éditeur (i.e. l'éditeur génère des méthodes et des attributs des classes en fonction des caractéristiques du contexte saisies par l'utilisateur). Selon la manière de générer, ces classes peuvent contenir des sémantiques différentes ou proches les uns des autres. Dans ce dernier cas, il y a une répétition de codes dans des classes générées.

A propos de l'implémentation d'un domaine générique, basé sur la génération des classes pour représenter des contextes d'utilisation, il est intéressant de remarquer que :

- Les concepts constituant un contexte d'utilisation font partie du modèle conceptuel du domaine spécifique lié avec ce contexte. Ces concepts sont matérialisés explicitement par des classes générées ;
- Chaque domaine spécifique doit être construit comme une fédération différente dont l'UC est la matérialisation de son modèle conceptuel ;

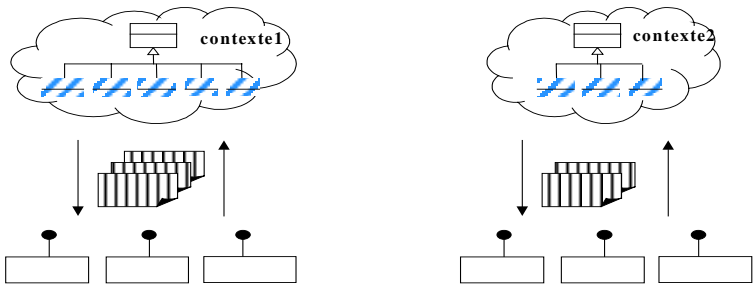

Figure 85. Différentes fédérations pour un domaine générique par génération des classes

- Le comportement des fédérations peut être personnalisé. Dans ces fédérations, les participants se coordonnent avec des UCs différents ; les contrats verticaux peuvent être installés sur les méthodes des classes générées qui sont différentes.

Néanmoins, il est très lourd de construire une nouvelle fédération chaque fois que l'on veut utiliser le domaine dans un contexte différent, ce qu'un utilisateur final ne peut pas faire. C'est pour cette raison que nous ne sommes pas favorables à la génération des classes pour représenter un contexte d'utilisation.

2.3.3. Interprétation des modèles de contexte d'utilisation

Au lieu de générer des classes pour représenter un contexte, un éditeur peut générer une description de ce contexte. Cette description peut être exprimée par un formalisme quelconque dont XML [XML] est un candidat adéquat.

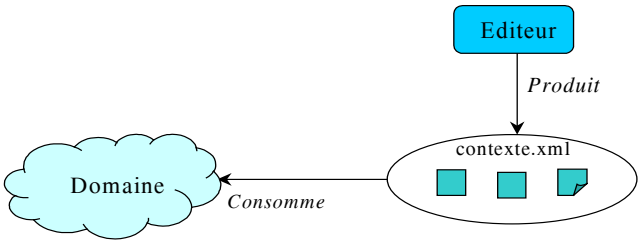

Figure 86. Editeur produit des descriptions de contexte d'utilisation

Comme son nom l'indique, la description d'un contexte, produit par l'éditeur, contient des informations destinées à exprimer ce contexte. Chaque concept du contexte est représenté par un élément dans cette description. Nous proposons d'appeler un modèle de contexte une telle description.

Les modèles ne contiennent qu'une description structurelle des contextes correspondants. Afin de travailler avec ces modèles, le domaine doit jouer le rôle d'un interpréteur. Ceci veut dire que le domaine interprète des modèles afin de faire varier son comportement selon les caractéristiques des contextes correspondants.

Par conséquent, le domaine doit être réalisé comme étant une solution paramétrée par ses contextes d'utilisation potentiels. Autrement dit, le domaine implémente le comportement nécessaire, mais n'a pas d'état initial. Le fait de charger un modèle permet d'initialiser le domaine. Ce dernier a désormais son comportement et son état initial. Il devient spécifique pour ce modèle.

Pour ce faire, il est indispensable que le domaine possède une manière, dynamique et flexible, pour représenter les concepts qu'il va potentiellement trouver dans des modèles.

Dans cet objectif, notre proposition consiste à :

- Ne pas matérialiser explicitement les concepts par des classes, mais à les réifier, à l'exécution, par des objets afin de pouvoirs les créer dynamiquement ;
- Ajouter un faux concept dont les instances représentent les concepts constituant des contextes d'utilisation.

En effet, l'idée de fond de notre proposition est de gérer des éléments d'un contexte (e.g. un stagiaire Toto) par deux objets ; un objet qui représente son type (i.e. stagiaire) et un objet qui représente l'élément lui-même (i.e. Toto). Par conséquent, il est à la charge du domaine d'initialiser ces objets pour représenter les éléments d'un contexte et leur type.

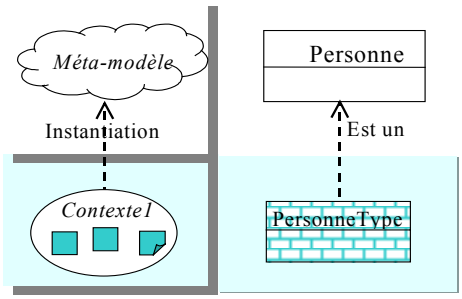

Figure 87. Personne et leur type

La Figure 87 montre la modélisation du domaine de gestion des ressources. Le faux-concept `PersonneType` est ajouté afin de représenter tous les concepts de personne potentiels. Une `PersonneType` est une instance ontologique de `Personne` comme distingués par Atkinson et Kühne.

Au niveau implémentation, le diagramme des classes utilisé par l'interpréteur de ce domaine est illustré dans Figure 88 ; chaque instance de `Personne` a un pointeur vers une

instance de `PersonneType` désignant son type. Ce diagramme est expliqué par le fait que l'interpréteur doit travailler au niveau méta-modèle et également au niveau modèle afin d'instantier les éléments d'un contexte ainsi que leur type.

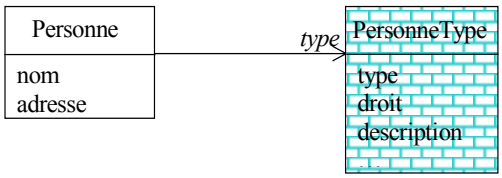

Figure 88. Diagramme des classes utilisé par l'interpréteur générique

Récemment, d'autres travaux tentent de séparer les éléments de leur type comme dans notre approche. Un exemple typique est le patron de type d'objets (*Type Object Pattern*) [JW]. Le fait de séparer les éléments de leur type apporte plusieurs avantages :

- Les concepts constituant des contextes d'utilisation peuvent être ajoutés, de manière dynamique et flexible, en créant des instances des faux-concepts du domaine ;
- Un modèle ne contient qu'une description structurelle d'un contexte. Ainsi, nous évitons la duplication de code comme dans le cas de la génération des classes ;
- Les éléments d'un contexte sont gérés indépendamment de leur type. Par conséquent, nous pouvons changer facilement le type d'un élément. Un élément peut aussi avoir plusieurs types.

A propos de l'implémentation d'un domaine comme étant un interpréteur à partir d'un ensemble de participants, il est intéressant de remarquer que :

- L'ensemble des participants joue le rôle d'interpréteur. Par conséquent, il est nécessaire qu'au moins un des participants soit capable d'interpréter les modèles. Autrement dit, il faut au moins un participant de type d'outil ;
- Il y a une fédération unique pour réaliser l'interpréteur. L'UC de cette fédération est la matérialisation du méta-modèle du domaine ainsi que des faux-concepts dont les instances représentent les concepts constituant des contextes de ce domaine. Le diagramme des classes dans la Figure 88, par exemple, représente l'UC de la fédération correspondant à l'interpréteur générique du domaine de ressources ;

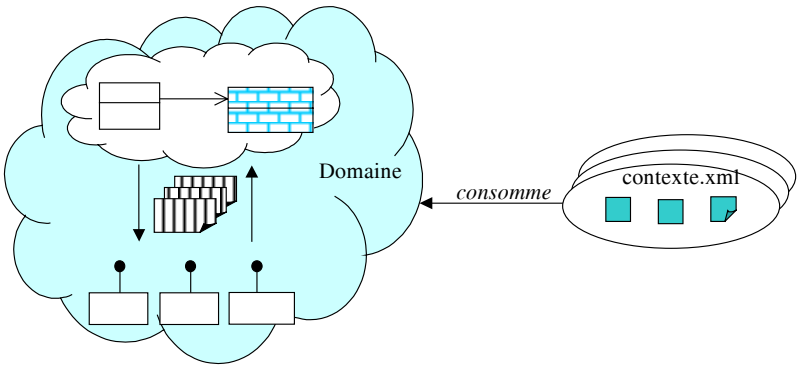

Figure 89. La fédération unique pour réaliser le domaine comme un interpréteur

- Bien que le modèle de contexte ne contienne qu'une description structurelle d'un contexte, il exprime une partie important du modèle conceptuel du domaine spécifique lié à ce contexte.

Avec ces observations, nous pouvons dire qu'il est plus simple et flexible de réaliser des domaines génériques par l'interprétation des modèles de contexte que par la génération des classes. Quel que soit le contexte d'utilisation, la réalisation du domaine, par une fédération des participants, est la même. Ceci permet réellement aux utilisateurs finaux de définir leurs contextes d'utilisation par des éditeurs graphiques.

3. Composition de domaines

3.1. Les problèmes

L'idée de construire des domaines génériques afin de pouvoir les utiliser dans différents contextes d'utilisation est très intéressante. Cependant, nous pensons que ceci n'est pas obligatoire pour tous les domaines. Par conséquent, lors de la composition de domaines pour construire une application globale, nous pouvons trouver des domaines de diverses natures :

- Des domaines spécifiques créés pour être utilisés dans des contextes particuliers ;
- Des domaines génériques qui se basent sur la génération des classes pour représenter des contextes d'utilisation ;
- Des domaines génériques qui se basent sur l'interprétation des modèles de contexte d'utilisation.

Face à cette situation, nous pensons qu'il est nécessaire de composer des domaines avec une même spécificité. La manière la plus simple est de se baser sur la composition de domaines spécifiques. Pour cela, nous « transformons » des domaines génériques en des domaines spécifiques, en déterminant leur contexte d'utilisation, avant d'effectuer la composition de domaines.

Imaginons que nous allons composer deux domaines :

- Le domaine de gestion des ressources humaines dans le contexte d'un laboratoire dans lequel les types de personne sont les permanents, les responsables, les stagiaires, les secrétaires, etc. ;
- Le domaine de gestion d'activités dans le contexte d'un projet de recherche où les types d'activités sont l'administration, la recherche le développement, etc.

Composer ces deux domaines consiste à confier l'activité d'*administration* à la ressource *responsable*, l'activité *recherche* et le *développement* à la ressource *permanent*, l'activité *développement* au *stagiaire*, etc. Ceci correspond au fait de lier les concepts provenant du modèle conceptuel de ces deux domaines spécifique. Or, selon la manière de réaliser ces domaines, ces concepts peuvent être :

- Matérialisés explicitement par des classes ;

Cette situation survient lorsque : (1) les domaines sont spécifiques par leur nature (i.e., ils sont crées pour être utilisé dans un contexte spécifique) (2) les domaines sont génériques et ils génèrent des classes pour représenter leurs contextes d'utilisation (cf. section 2.3.2) ;

- Décrits structurellement dans un modèle et réifiés, à l'exécution, par des objets.
  Cette situation survient lors que des domaines sont génériques et qu'ils se basent sur des modèles pour représenter leurs contextes d'utilisation (cf. section 2.3.3).

Face à cette situation, la question se pose de savoir comment nous composons ces domaines ?

3.2. Contrôleurs de modèles des domaines génériques

La composition de domaines, que nous avons proposé dans le chapitre 4, se base sur une hypothèse assez forte : il y a une transformation simple et directe entre un modèle conceptuel et son implémentation (i.e. un concept est représenté simplement par une classe, voir section 3.2.2 du chapitre 4) pour tous les domaines à composer. De cette manière, il n'y a pas de différence entre une composition de modèles conceptuels et l'implémentation de cette composition.

Or, cette hypothèse n'est pas assurée, dans la condition où les domaines à composer sont de diverses natures et peuvent utiliser de manières différentes pour implémenter leurs modèles conceptuels.

La solution que nous proposons consiste à composer des domaines en se basant sur leur modèle conceptuel, exprimé par une représentation commune et indépendamment de l'implémentation. Une transformation automatique sera favorisée afin de traduire la composition de modèles conceptuels en une composition d'implémentations de ces modèles. De cette manière, nous sommes libres d'implémenter les modèles conceptuels d'un domaine, spécifique ou générique. Pour un domaine générique, nous pouvons non seulement appliquer par nos deux manières de paramétrer un domaine générique, mais aussi réaliser un domaine générique comme un *framework* boîte blanche, un *template,* etc.

Dans cette direction, nous proposons d'ajouter, pour chaque domaine générique, un **contrôleur de modèles**. Les tâches de ce dernier peuvent être résumées comme suit :

- Etant donné un contexte d'utilisation, il reconstitue le modèle conceptuel du domaine spécifique lié à ce contexte en l'exprimant dans une représentation commune (e.g. un diagramme de classes UML) ;
- Etant donné le modèle conceptuel d'un domaine spécifique, il fournit les informations liées à l'implémentation réelle de ce modèle.

Avec l'aide des contrôleurs de modèles, nous possédons explicitement le modèle conceptuel, exprimé par une représentation commune et indépendamment de son implémentation, des domaines à composer. Nous sommes ainsi prêts à composer les modèles conceptuels des domaines en établissant des relations entre des concepts

provenant de ces modèles. Pour exprimer cette composition, nous nous sommes intéressés par un langage de relation de haut niveau d'abstraction permettant de définir la sémantique d'une relation entre deux concepts quelle que soit leur implémentation. Il est clair que le langage de relation que nous avons présenté dans le chapitre 4 devra être amélioré afin de prendre en compte la relation entre un concept et son implémentation.

Nous proposons d'ajouter également un **gestionnaire de composition**. Ce dernier a pour objectif de transformer une composition de modèles conceptuels (i.e. entre des concepts) en une implémentation de cette composition. Cette dernière consiste à établir des relations entre les représentations des concepts, qui peuvent être des classes ou des objets et à installer des contrats horizontaux sur les « bonnes » méthodes afin de pouvoir gérer la sémantique des relations.

Il est clair que, pour accomplir sa tâche, le gestionnaire de composition doit collaborer avec les contrôleurs de modèles pour savoir le lien entre un modèle conceptuel et son implémentation. Ce principe est illustré dans la Figure 90.

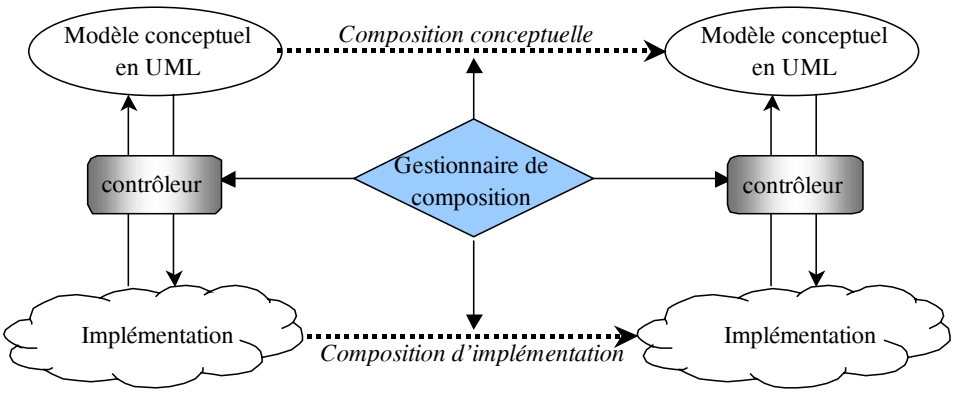

Figure 90. Principe de composition de domaines de différentes natures

A première vue, le principe de la transformation, réalisé par le gestionnaire de composition, ressemble à celui du MDA que nous avons abordé dans le chapitre 2. Cependant, si nous regardons plus en détail, notre transformation est beaucoup plus simple car :

- Le méta-modèle d'une composition conceptuelle et celui d'une composition d'implémentation sont prédéfinis. Ceux-ci sont définis respectivement par des types de relation et le modèle de contrats horizontaux que nous avons proposé dans le chapitre 4 ;
- La transformation effectuée par le gestionnaire de composition est simplifiée par les contrôleurs des modèles. Ces derniers sont chargés de la transformation bidirectionnelle entre un modèle conceptuel et son implémentation ;
- Il est possible de retenir des patrons concernant la manière d'implémenter un modèle conceptuel. Excepté la réalisation un domaine générique comme un interpréteur, un concept d'un contexte est toujours matérialisé par une classe, programmée ou générée.

Pour l'instant, nous avons identifié quelques possibilités pour transformer une composition conceptuelle en une composition d'implémentation comme suit :

- Lorsque les concepts à composer sont tous matérialisés explicitement par des classes, programmées ou générées, la transformation est directe. En effet, pour la gestion de contrats, il n'y pas de différence entre un concept, une classe générée et une classe programmée ;

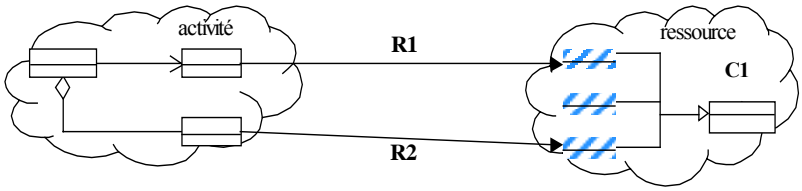

Figure 91. Des concepts sont matérialisés explicitement

- Lorsque certains concepts sont réifiés à l'exécution, la relation peut être établie sur de fausses-classes (i.e. la matérialisation des faux-concepts) en combinant avec des contraintes pour désigner une instance concrète (i.e. un concept). Ceci est illustré dans la Figure 92.

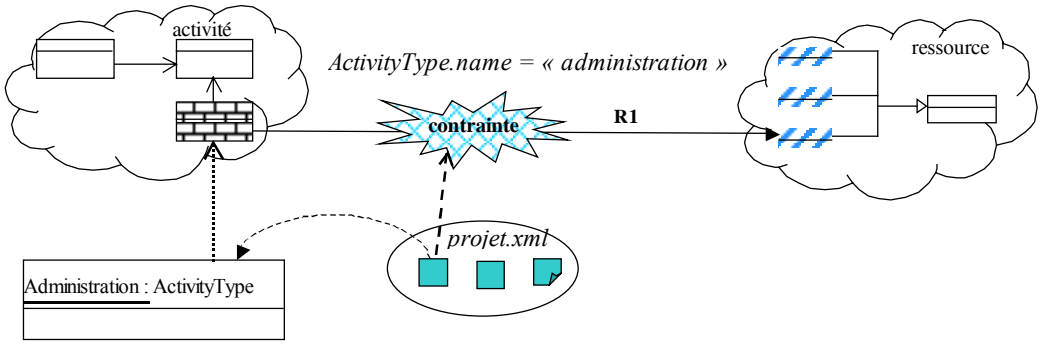

Figure 92. Relation sur des fausses-classes et contraintes

Nous sommes actuellement intéressés à appliquer les langages de transformation de modèles issus du MDA pour exprimer et effectuer automatiquement cette transformation de composition. Toutefois, il est important de posséder un langage permettant d'exprimer, de manière simple, nos méta-modèles de composition au niveau conceptuel et d'implémentation ainsi que des règles pour transformer une relation conceptuelle en une relation d'implémentation. Dans cette direction, nous partageons beaucoup de similitudes avec l'approche MDA.

4. Application de gestion documentaire générique

Nous avons appliqué le principe des interpréteurs de modèle pour la réalisation de l'environnement générique de gestion documentaire. Nous résumons, dans cette section, le diagramme de classe sur lequel se base l'interpréteur de ces domaines génériques.

4.1. Domaine de procédé générique

Le diagramme simplifié, sur lequel se base l'interpréteur de ce domaine est illustré dans la Figure 93. Les concepts plus foncés sont des « faux-concepts » ajoutés afin de faire varier les contextes d'utilisation potentiels du domaine. Avec ces concepts, nous pouvons réaliser différents modèles de procédé avec différents types d'activités qui supportant différents types de donnés.

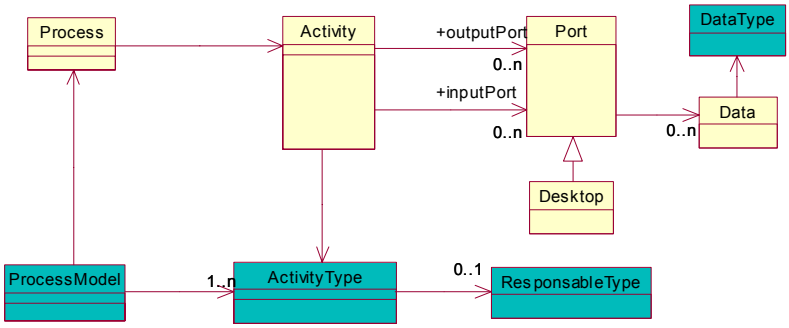

Figure 93. Diagramme des classes de l'interpréteur du domaine de procédé générique

4.2. Domaine de document générique

Le diagramme simplifié de l'interpréteur du domaine de document générique est illustré dans la Figure 94. Le concept `DocumentType` est un faux-concept permettant de caractériser différents types de documents que ce domaine peut traiter.

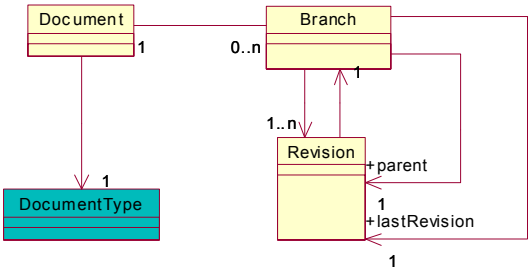

Figure 94. Diagramme des classes de l'interpréteur du domaine de document générique

4.3. Domaine de ressource générique

Le diagramme simplifié de l'interpréteur du domaine de ressource générique a été étudié tout au long de ce chapitre. Nous reprenons ce diagramme dans la Figure 95.

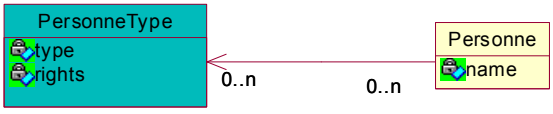

Figure 95. Diagramme des classes de l'interpréteur du domaine de ressource générique

4.4. Domaine d'espace de travail générique

Le diagramme simplifié de l'interpréteur du domaine d'espace de travail générique est illustré dans la Figure 96. Le concept `WorkspaceType` est un méta-concept permettant de caractériser différents types d'espace de travail que ce domaine peut traiter.

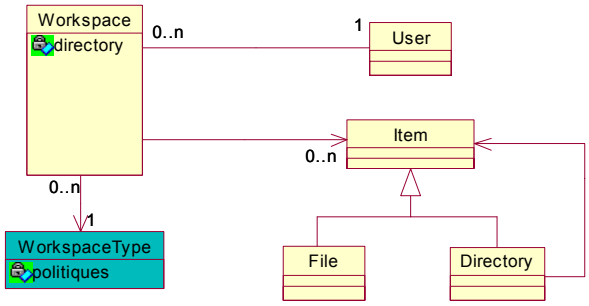

Figure 96. Diagramme des classes de l'interpréteur du domaine d'espace de travail générique

5. Discussions et conclusions

Dans ce chapitre nous avons étudié une manière de construire des domaines génériques pouvant être utilisés dans des contextes d'utilisation similaires. Nous avons également étudié la composition lorsque les domaines à composer sont génériques ou spécifiques.

Bien que notre manière de rendre des domaines génériques est aussi étudiée dans d'autres travaux [JW], nous pensons que celle-ci ne peut s'appliquer qu'à des cas où le concept de type est très important et varié. Dans ces cas, on a besoin d'ajouter et d'étendre, de façon très flexible et très simple, des types différents.

Cependant, comme les domaines sont paramétrés, ils ne sont pas capables de traiter un nombre infini de modèles. Par conséquent, les éditeurs de modèles de contexte, ne proposent que des contextes que les domaines savent traiter.

Malgré cette limitation, cette manière de paramétrer des domaines nous permet de construire non seulement une application, mais une famille d'application.

Dans le chapitre suivant, nous allons étudier l'implémentation concrète des propositions étudiées tout au long de cette deuxième partie ainsi que les diverses applications que nous avons développé pour montrer la viabilité de notre approche.

Chapitre VI

Expérimentations et validation

1. Introduction

Ce chapitre constitue la dernière partie de cette thèse. Dans ce chapitre, nous présentons les diverses expérimentations réalisées dans l'objectif de montrer la viabilité de notre approche de composition. Deux grands sujets seront discutés :

- La réalisation d'une plate-forme permettant la définition, l'exécution et l'administration des applications globales. Ces dernières peuvent être réalisées comme étant une fédération de participants ou une fédération de domaines ;

- La réalisation de plusieurs applications développées au cours des dernières années. Parmi ces applications, l'application de gestion documentaire dont le cahier des charges a été présenté dans le chapitre 4, sera étudiée plus en détail. Actuellement, cette application est validée au niveau industriel dans le cadre de la collaboration de cette thèse.

Ces expérimentations, réalisées en `Java`, constituent environ 300 000 lignes de code au total. Elles comportent plusieurs versions avec des architectures et des fonctionnalités différentes. Celles-ci marquent nos évolutions au niveau conceptuel et au niveau implémentation.

Ce chapitre est structuré de la manière suivante. La section 2 présente la vision globale de notre plate-forme de support. Les sections 3 et 4 montrent l'implémentation du modèle de participant, de l'UC et des contrats. Le support pour la composition de domaines sera présenté dans la partie 5. Dans la section 6, nous présentons les applications que nous avons réalisées et leurs différentes versions. Pour montrer la viabilité de notre approche, cette section compare les différentes versions de ces applications en ce qui concerne le volume de code, la stabilité, l'extensibilité, etc.

Nos expérimentations représentent un travail volumineux réalisé grâce aux efforts de plusieurs personnes pendant plusieurs années et ont fait l'objet de deux thèses [Ver00, Vill03]. Ainsi, je tâcherai, dans la section 7, de clarifier, bien que ce ne soit pas toujours facile dans un projet collectif, quelle fut ma participation personnelle, tant au niveau conceptuel que technique.

La section 8 discute et conclut ce chapitre.

2. Vision globale de la plate-forme de support

Notre approche propose non pas une architecture, mais une famille d'architectures pour construire des applications globales. Une application globale peut être simple en étant réalisée comme une fédération anarchique de participants ; elle peut être très complexe lorsqu'elle résulte de la composition de domaines.

Nous avons construit une plate-forme de support constituée d'outils, de langages de haut niveau, etc. afin de supporter nos proposions. Notre plate-forme est décomposée en deux parties : (1) le support pour la conception et la construction d'une application globale et (2) le support pour l'exécution et l'administration d'une application globale. Cette section ne présente que la vision globale de cette plate-forme.

2.1. Construction d'une fédération

Notre environnement pour la conception et la construction d'une fédération est constitué d'un éditeur graphique. Ce dernier propose plusieurs outils pour la vérification, la construction, la génération, l'édition pour les éléments d'une fédération.

Lié à l'édition des éléments d'une fédération, les outils les plus importants sont :

- L'éditeur de participant permettant de définir les rôles et les participants d'une fédération ;
- L'éditeur d'UCs permettant de définir des classes exécutées dans l'UC ;
- L'éditeur de contrats verticaux permettant de définir des contrats verticaux qui sont attachés aux méthodes des classes de l'UC d'une fédération ;
- L'éditeur de composition de domaines visant à définir la composition des domaines en spécifiant les relations entre concepts et les contrats horizontaux gérant ces relations ;
- L'éditeur de configuration destinant à définir différentes configurations d'une composition de domaines.

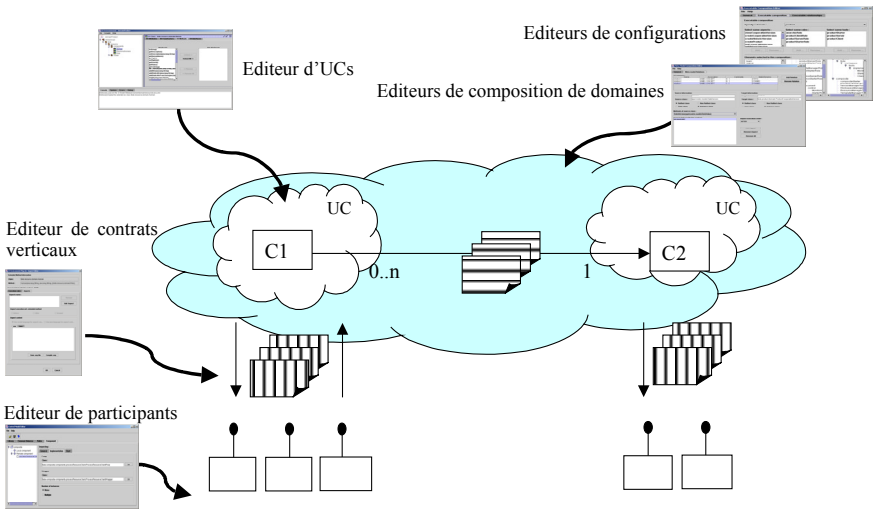

Figure 97. Editeurs pour définir les éléments d'une fédération

Tous ces éditeurs ont les propriétés communes suivantes :

- Ils utilisent XML (*eXtensbile Markup Language*) pour la représentation et le stockage des facettes d'une fédération. Les fichiers *'.jar'* sont utilisés pour les exécutables des fédérations ;
- Ils utilisent un schéma XML et *castor* [Castor] pour générer des classes visant à représenter et à stocker des facettes des fédérations ;
- Ils travaillent en se basant sur des conventions de l'architecture physique des fédérations.

Une fois qu'une fédération est définie en utilisant ces éditeurs, il est nécessaire de passer le résultat à la machine à objets étendus – MOE [DES02] (voir section 4.1) pour préparer les points d'extension (i.e. interceptés) des classes de l'UC. Cette fédération est ainsi prête pour être exécutée par le moteur de fédération. La Figure 98 résume ce processus.

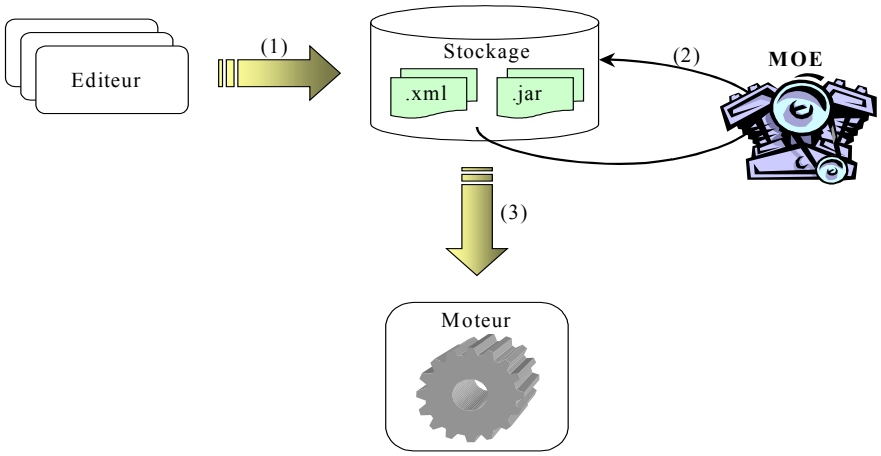

Figure 98. Le processus simplifié de construction et exécution d'une fédération

2.2. Architecture globale du moteur de fédération

Le moteur de fédération propose un environnement pour exécuter et administrer les fédérations. Sur le même principe que l'environnement de conception, le moteur sera structuré en une architecture à plusieurs couches afin de mieux supporter notre famille d'architecture. Il est actuellement constitué de deux couches : le moteur de base et le moteur d'extension.

Le moteur de base gère les participants, l'UC et les contrats, verticaux ou horizontaux. Il ignore délibérément le concept de domaine et de composition de domaines. En réalité, le moteur de base est la combinaison d'une machine à composants et d'une machine à l'AOP. Il peut être donc utilisé pour réaliser des applications à base de composants et / ou pour des applications à l'AOP.

Le moteur de base propose une interface graphique d'administration. Cette interface contient plusieurs onglets dont chacun propose des services permettant de surveiller, administrer et contrôler une facette de l'exécution de la fédération en cours. On trouve des

onglets pour la gestion de l'UC, la gestion des contrats (i.e. des aspects dans l'image), la gestion de la communication et du démarrage des participants, etc.

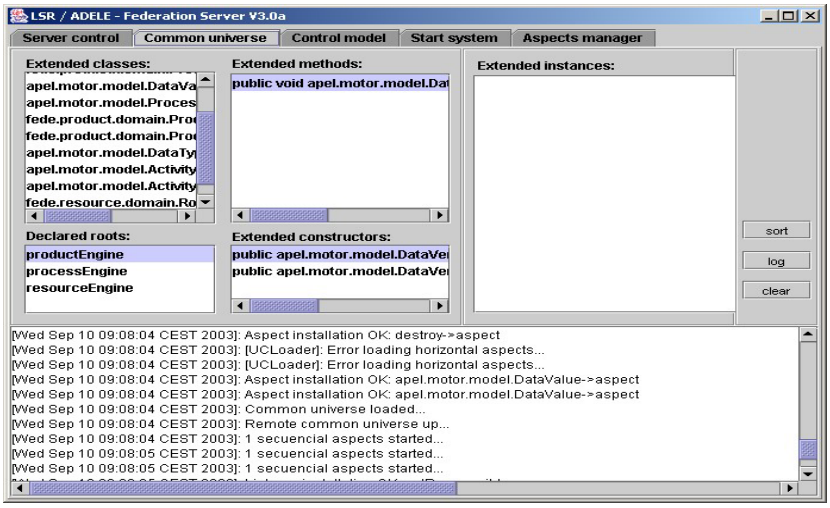

Figure 99. Fenêtre administrative du moteur de fédération

En revanche, le moteur d'extension connaît le concept de domaine et la composition de domaines. Il étend le moteur de base pour supporter les fédérations de domaines.

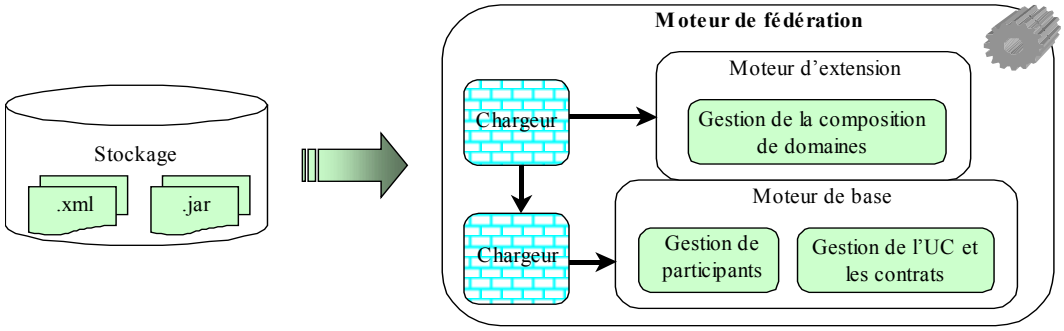

Figure 100. Architecture globale du moteur de fédération

A chaque couche du moteur, il y a un *chargeur* qui initialise le moteur correspondant, avec les informations de la fédération à exécuter. Grâce aux chargeurs, le moteur est indépendant de l'architecture physique et du langage utilisé par les éditeurs pour décrire les diverses facettes d'une fédération.

Dans les sections qui suivent, nous allons présenter les services du moteur pour la gestion de participants, la gestion de l'UC et des contrats et la composition de domaines.

3. La gestion de participants

La gestion de participants est basée sur notre machine virtuelle à composants. Cette machine est à la fois dynamique et distribuée. Le fait d'être dynamique permet à tout moment, d'ajouter/d'enlever des participants, de changer les propriétés non-fonctionnelles

des participants, de créer et de stopper des instances de participants. Le fait d'être distribué permet de centraliser la gestion de la distribution des participants au moteur. D'une part, les développeurs de participants ne doivent pas se préoccuper de la distribution ; la réutilisation des participants est plus aisée. D'autre part, l'adaptation de la gestion de la distribution aux nouvelles technologies de *middleware* sera commode à réaliser.

Concernant les propriétés non-fonctionnelles des participants, seul un ensemble de propriétés non fonctionnelles est prédéfini et géré par le moteur (la distribution, le cycle de vie des instances des participants, la récupération d'erreurs). Par contre, contrairement aux plates-formes à composant « traditionnelles », d'autres services non-fonctionnels ont pour objectif d'être ajoutés librement, en les considérant comme des domaines, par exemple (domaine de transaction, de persistance, de sécurité, etc.).

3.1. Le moteur de participants

Le *framework* du modèle de participants est constitué d'un éditeur de participants et d'un moteur de participant. Cet éditeur permet de définir les rôles et les participants en spécifiant leurs propriétés fonctionnelles et non-fonctionnelles.

Le moteur de participants se base sur les informations définies par l'éditeur pour contrôler l'exécution des participants. Du point de vue architecture, le moteur est composé d'un serveur, de plusieurs clients, appelés les **starters,** qui s'exécutent sur des machines clientes, et d'un système de communication. Ce dernier permet au serveur et aux starters de communiquer dans un environnement *Intranet* et *Internet* où il y a des *fire-walls*.

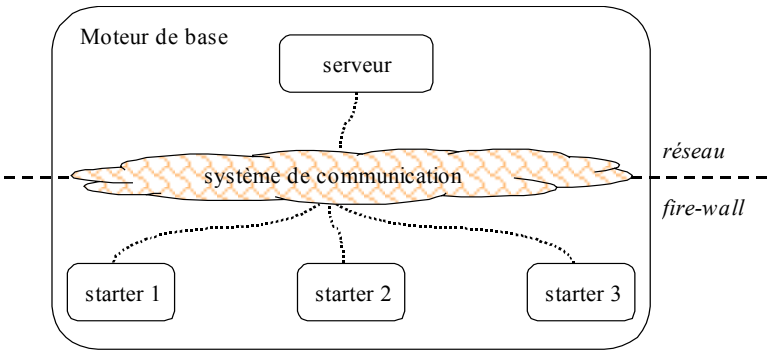

Figure 101. Architecture globale du moteur de base

3.1.1. Le serveur et les starters du moteur

Le serveur est le noyau du moteur. Il est chargé de :

- Interpréter les modèles de participants ;
- Gérer l'exécution des participants, locaux et distants, en gérant le cycle de vie de leur mandataire ;
- Localiser les instances de participants ayant les caractéristiques non-fonctionnelles souhaitées lorsqu'il y a une demande (voir section 3.2) ;

- Aider le mandataire et l'emballage des participants distants à communiquer lorsqu'il y a un *fire-wall* du côté client (voir section 3.1.2).

Les starters sont les représentants du moteur sur les machines clientes. Ils ont pour objectif d'aider le noyau du moteur à :

- Gérer l'exécution des participants distants en gérant le cycle de vie de leur emballage;
- Fournir un point d'entré pour communiquer avec les emballages et les participants qui sont derrières un *fire-wall* (voir section 3.1.2).

La Figure 102 montre le partage des responsabilités entre le serveur et le starter pour la gestion des instances des participants distants et locaux. L'exécution des participants locaux est complètement supportée par le serveur. En revanche, pour l'exécution des participants distants, le serveur et les starters doivent collaborer.

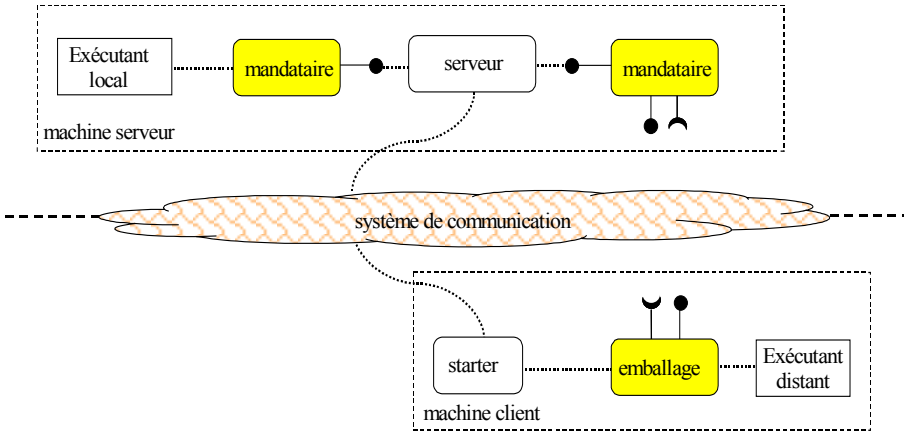

Figure 102. L'instance d'un participant local et distant

Dans l'objectif de faciliter l'adaptation dynamique des participants, le serveur du moteur émet un ensemble d'événements pour signaler l'état des participants et celui de la fédération. Concrètement, ces événements notifient le fait que : (1) le modèle d'un participant vient d'être ajouté/enlevé/modifié, (2) une instance de participant vient d'être créée ou détruite, (3) le moteur ou le starter vient d'être démarré sur une machine spécifique, etc. Un participant quelconque peut écouter ces événements afin de réagir ou bien adapter son exécution avec le reste de la fédération.

Le moteur de participants propose également un protocole de la récupération en cas d'arrêt du côté serveur ou starter. En cas d'arrêt du moteur, une fois relancé, le nouveau serveur est capable de récupérer son ancien état d'exécution en se basant sur l'état des starters. En revanche, en ce qui concernent l'arrêt des starters, nous souhaitons mieux contrôler leur récupération. Pour cet objectif, nous avons ajouté une nouvelle couche du côté starter : **le service**. Il s'agit d'une petite application dont la tâche est de gérer le cycle de vie du starter et d'assurer la connexion entre le serveur et le starter. De cette manière, il y a une séparation claire :

- Le service, qui gère le cycle de vie du starter ;

- Le starter, qui gère le cycle de vie des emballages.

Avec l'aide des services, l'exécution et la récupération des starters sont complètement contrôlées par le moteur et les services correspondants. Les détails du protocole de récupération seront présentés en annexe A.

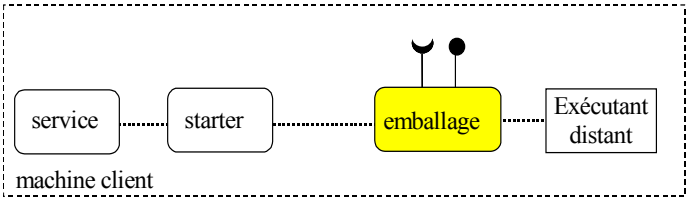

Figure 103. Service et starter

3.1.2. Système de communication

Dans le cas simple où la fédération s'exécute dans un réseau *Intranet*, une communication *RMI (Remote Method Invocation)* doit être suffisante. Grâce au registre *RMI* (*RMIregistry*) sur la machine serveur, un starter localise le serveur et lui signale sa présence. Lorsque le serveur reçoit ce signal, il garde le lien vers ce starter pour toutes les communications ultérieures vers ce starter.

De la même manière, chaque emballage localise son mandataire par le nom *RMI* sur le registre *RMI* (ce nom est attribué par le serveur lors de la création de l'instance du participant, voir section 3.2). Ensuite, il demande à son mandataire de garder le lien. Désormais, le mandataire et l'emballage peuvent communiquer par *RMI*.

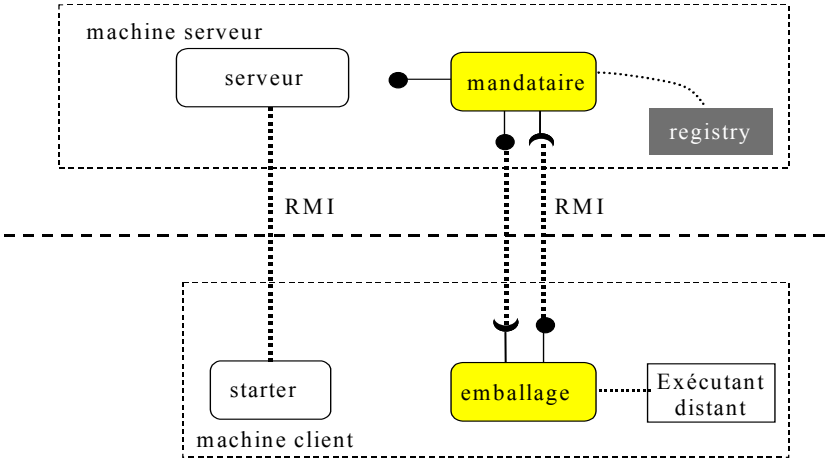

Figure 104. Communication par RMI

Dans le cas général, une application globale doit être capable de s'exécuter dans un réseau *Internet* où il y a potentiellement des *fire-walls*. Nous considérons, tout d'abord, la situation où il y a le *fire-wall* du côté client. Pour cette situation, nous avons construit un système de communication par messagerie basé sur des *Sockets*.

Le principe du système de communication par messagerie peut être résumé comme suit :

- La communication du starter vers le serveur et de l'emballage vers son mandataire peut être réalisé en RMI puisqu'il n'existe pas de *fire-wall* du côté serveur ;
- Toutes les communications depuis le serveur vers le starter sont effectuées par messagerie et à travers des *sockets* multiplexés. Un *socket* est réservé pour envoyer les appels de méthodes et l'autre pour recevoir les réponses. Les appels de méthodes et les réponses sont sérialisés sous forme des messages. Ces *sockets* sont créés par le starter et sont gardés par le serveur pour les communications ultérieures ;
- La communication entre un mandataire vers son emballage est effectuée à travers deux *sockets* liant le serveur et le starter ;

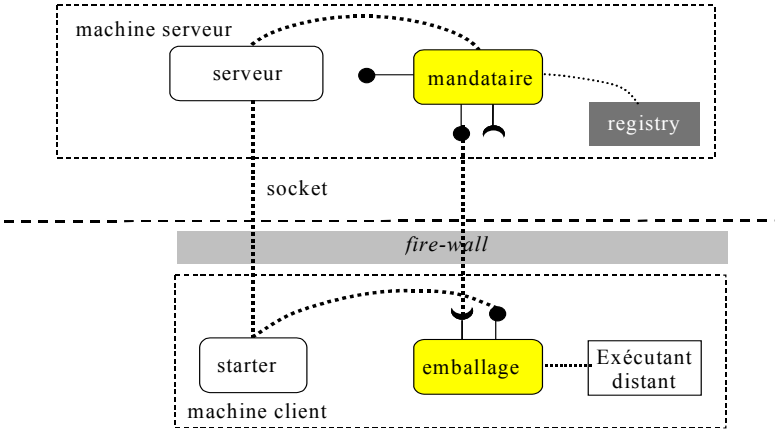

Figure 105. Communication par *Socket* avec le *fire-wall* du côté client

Dans le cas où le *fire-wall* est à la fois du côté serveur et du côté client, la communication par messagerie peut être faite en ouvrant des *Sockets* à travers un serveur *HTTP* du côté serveur et un du côté client. Cependant, dans le cadre de nos expérimentations, nous nous limitons à la situation où le *fire-wall* se trouve du côté client.

Le choix de mode de communication, par *RMI* ou par *Socket*, est déclaré dans le fichier de configuration du starter de chaque poste client. Le serveur du moteur peut communiquer avec différents starters en utilisant différents modes de communication.

Il est important de noter est que le protocole, RMI ou Socket, servant à mettre en correspondance le mandataire et l'emballage d'une instance d'un participant distant, est effectué par les classes de base du *framework* de l'adaptateur et au moment d'instantier cette instance. Le code de l'adaptateur des participants est très simple en héritant de ces classes. Le *framework* de l'adaptateur sera présenté en annexe A.

3.2. Création et localisation des instances de participants

Pour localiser une instance de participant, il est nécessaire de passer le nom d'un rôle que le participant joue et optionnellement, un ensemble des propriétés non-fonctionnelles que l'instance à localiser doit satisfaire. En outre, il est possible de spécifier explicitement la

manière de localiser cette instance. Il s'agit de trois opérations *find*, *create* et *new* parmi lesquelles le *'create'* est l'opération par défaut :

- Avec l'opération *'create'* le moteur cherche une instance qui correspond aux caractéristiques souhaitées ; s'il n'en trouve pas, le moteur crée une nouvelle instance avec ces caractéristiques ;
- Avec l'opération *'find'* le moteur s'arrête s'il ne trouve pas d'instance qui satisfasse les caractéristiques demandées ;
- Avec l'opération *'new'* le moteur crée toujours une nouvelle instance du participant avec les caractéristiques demandées.

Pour créer une instance, le moteur doit interpréter le modèle de participant pour savoir, tout d'abord, quel(s) participant(s) joue le rôle donné pour sélectionner le plus approprié. Ensuite, il est nécessaire de considérer le type de ce participant, local ou distant, et les fabriques nécessaires pour créer une instance de ce participant.

Il est important de rappeler que la création d'une instance de participant correspond à la création d'une instance de l'adaptateur de ce participant. Pour des participants locaux, la création d'une instance de l'adaptateur peut être faite seulement par le serveur. En revanche, celle d'un participant distant doit être faite grâce à la collaboration entre le serveur et le starter de la machine : le moteur crée l'instance du mandataire tandis que le starter crée celle de l'emballage. Les informations concernant le mandataire et l'emballage, tels que le nom *RMI* du mandataire enregistré dans le serveur de nom (*RMIregistry*), la machine cliente et serveur, etc., doivent être connues par le mandataire et l'emballage afin qu'ils puissent se contacter et communiquer ultérieurement.

Il est aussi important de noter que l'objet retourné à celui qui demande l'instance de participant est l'instance du mandataire du ce participant.

4. Univers Commun et contrats

Notre implémentation de l'univers commun et des contrats est constituée de trois éléments principaux :

- L'éditeur de l'UC permettant de spécifier les classes qui s'exécutent dans l'UC ;
- L'éditeur de contrats verticaux qui définit des contrats, s'il y en a, attachés sur les méthodes interceptées de classes de l'UC ;
- Un compilateur visant à traduire les contrats, exprimés dans notre langage de contrat textuel, en des classes `Java` ;
- Un moteur de l'UC qui gère l'exécution des classes de l'UC et des contrats attachés aux méthodes de ces classes.

Nous n'étudions, dans cette section, que le moteur de l'UC. Les éditeurs, le langage de contrat et le compilateur seront présenté dans l'annexe A.

4.1. Instrumentation de l'univers commun

Afin d'accomplir ses tâches, le moteur de l'UC doit instrumenter l'exécution des classes de l'UC afin de pouvoir intervenir dans les cas nécessaires. Nous avons considéré deux options pour instrumenter l'exécution des classes dans l'UC : (1) utiliser des *proxies* dynamique de `Java` et (2) utiliser la machine à objets étendus – MOE [DES02].

Avec les *proxies* dynamiques de `Java` [DP], il y a, pour chaque objet dans l'UC, une instance de *proxy* jouant le rôle d'un mandataire entre les clients et cet objet. Pour ce faire, au moment où le moteur reçoit une demande de création d'un objet dans l'UC, il fait l'appel à une fabrique afin de générer un *proxy* pour cet objet. L'objet retourné aux clients est le *proxy* qui implémente la même interface que l'objet de l'UC.

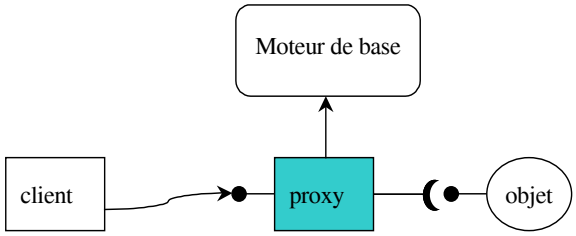

Figure 106. Le *proxy* dynamique fait le lien entre le client et l'objet de l'UC

Tous les appels de méthode vers les objets de l'UC sont passés d'abord à son *proxy*. Par conséquent, le *proxy* peut passer le contrôle au moteur afin qu'il puisse déclencher les contrats attachés à cette méthode ou bien émettre un événement s'il y a des observateurs qui s'intéressent à cet événement.

Dans nos anciennes implémentations (les versions *fedeV1* et *fedeV2*), nous avons utilisé les '*proxies'* dynamiques. Néanmoins, l'usage des *proxies* dynamique de `Java` présente quelques inconvénients qui sont, entre autres :

- Le nombre de *proxies* est celui des objets dans l'UC. Ceci affecte la performance du moteur ;
- Le code doit être écrit pour qu'il puisse être instrumenté. Ceci, parce que les *proxies* dynamiques travaillent avec les couples (interface, classe) pour l'instrumentation ;
- Il est difficile d'ajouter de nouveaux attributs à une classe. Ceci est très important pour la gestion des contrats horizontaux où des nouveaux attributs sont ajoutés dynamiquement pour représenter les paires d'instances concernées par une relation.

La MOE a été construite afin d'éliminer les inconvénients des *proxies* dynamiques de `Java` [DES02]. Il s'agit d'une machine permettant d'instrumenter afin de modifier le comportement d'une ou plusieurs classes `Java` d'une application sans déstabiliser le bon fonctionnement de cette dernière.

Concrètement, pour chaque classe de l'application à étendre, cette machine permet d':

- Ajouter des nouveaux attributs de manière statique (i.e. pré-compilation) ou dynamique (i.e. en exécution) ;

- Intercepter des méthodes, y compris des constructeurs, afin de pouvoir intervenir dans l'exécution de ces méthodes ;
- Ajouter de nouvelles interfaces.

Cette machine prend en entré le code binaire (*bytecode*, en anglais) des classes de l'application à étendre et une description des extensions qui est fournie par l'éditeur de l'UC (cf. annexe A). En modifiant directement le *bytecode* des classes, cette machine génère une nouvelle application avec les classes modifiées comportant les extensions souhaitées.

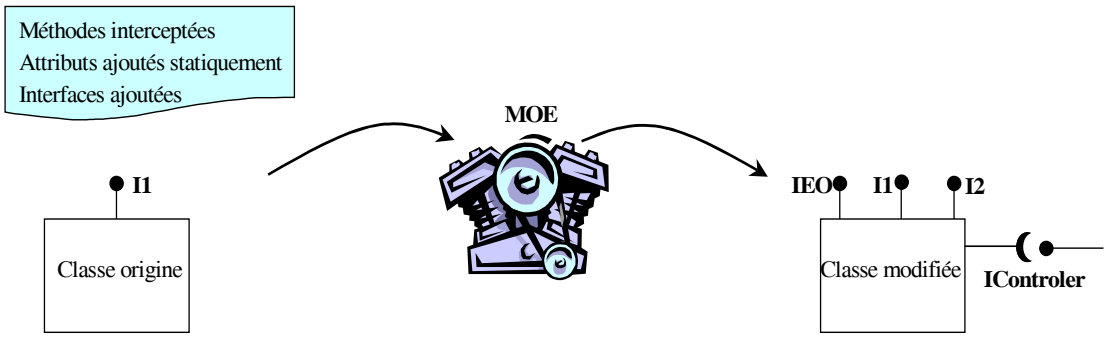

Figure 107. Principe de la machine à objets étendu - MOE

Chaque classe modifiée implémente l'interface `IEO`. Cette dernière contient des méthodes permettant de manipuler les attributs ajoutés statiquement et également d'ajouter dynamiquement d'autres attributs lors de l'exécution.

Lors de l'exécution de l'application étendue (i.e. l'UC pour les fédérations), les instances d'une classe étendue passent le contrôle à une classe qui implémente l'interface `IControler`, chaque fois qu'une méthode (ou un constructeur) interceptée est appelée. Ainsi, le moteur de fédération doit implémenter l'interface `IControler` afin de prendre le contrôle de l'exécution des classes de l'UC.

Dans l'implémentation actuelle (version *fedeV3*), nous avons utilisé la machine MOE non seulement pour instrumenter de l'UC des domaines, mais aussi pour ajouter de nouveaux attributs servant à notre gestion, en particulier, des relations entre concepts. En plus, cette machine est servie pour instrumenter les exécutants écrits en `Java`. De cette manière, on est capable de surveiller et de contrôler l'exécution de ces participants sans avoir besoin de modifier leur code source et sans avoir rencontré beaucoup de difficultés.

4.2. Accès aux objets de l'univers commun

4.2.1. Accès local

Les accès locaux sont réservés aux participants, distants ou locaux, qui sont déclarés dans la base de participants d'une fédération. Ces participants accèdent aux objets de l'UC et travaillent avec ces objets grâce à leur mandataire qui s'exécutent sur la même machine que le moteur.

Pour les accès locaux, le moteur propose le concept de **racine**. Une racine est un objet dans l'UC, désignée par un nom symbolique unique. L'ensemble de toutes les racines est géré par le registre de nom du moteur. Ce registre est différent du registre *RMI*.

La Figure 108 montre l'exemple d'une racine, nommé *'documentRoot'*, de l'UC du domaine de document. Lorsque cette instance est localisée, il est nécessaire de faire le *casting* afin de recevoir un bon objet. Il est possible de trouver tous les autres objets dans l'UC de ce domaine en naviguant à partir de cet objet.

```
.....
ICommunUnivers communUnivers = CommonUniverse.getCommonUniverse() ; //pattern singleton
DocumentDomain pd = (DocumentDomain) communUnivers.getRoot( "documentRoot" );

....
```

Figure 108. Exemple d'utilisation de la racine de l'UC

4.2.2. Accès distant

L'accès distant est réservé aux participants non-déclarés jouant le rôle *'anarchique'* d'une fédération (cf. chapitre 3).

Pour supporter l'accès distant, le moteur enregistre l'UC dans le registre *RMI* par un nom unique. Ainsi, les participants distants peuvent localiser l'UC à distance afin de travailler avec les objets de l'UC.

Néanmoins, il est important de noter que les participants non-déclarés ne travaillent pas directement avec les objets de l'UC, mais à travers leur identificateur unique attribué par le moteur. Grâce à ces identificateurs, les participants non-déclarés demandent au moteur d'exécuter les méthodes sur les objets correspondants de l'UC.

Pour éviter les problèmes de *fire-wall*, les participants non-déclarés peuvent se baser sur le système de communication par messagerie, proposé par le starter, pour accéder à l'UC.

4.3. Evénements de l'univers commun

Le moteur de l'UC distingue deux catégories d'événements de l'UC qui ont pour objectif de notifier le fait que : (1)

- Une instance est créée ou détruite dans l'UC ;
- Une méthode ou un constructeur intercepté est appelé.

Pour ces deux catégories, les événements ne sont réellement émis par le moteur que lorsqu'il existe des observateurs (e.g. des participants) qui s'intéressent à ces événements. Ceci nous permet d'augmenter la performance du moteur.

Pour recevoir des notifications de l'UC à distance, le moteur propose deux manières : par *RMI* et par *Socket*. Le principe ressemble à celui du système de communication (cf.

section 3.1.2). La communication par *Socket* doit être utilisée lorsque l'observateur est derrière un *fire-wall*.

4.4. La gestion des contrats

Notre moteur propose un mécanisme de gestion de contrats similaire à celui d'AspectJ [Kic et al., Elr et al.]. Comme AspectJ, nous distinguons les types de contrat en fonction du rapport entre l'exécution du contrat et de la méthode à laquelle le contrat est attaché. Nous avons ainsi trois types de contrat : Avant (*Before*), Après (*After*) et A la place (*Around*). Dans le dernier cas, l'exécution de la méthode de l'UC sera remplacée totalement par celle des contrats. Cependant, à tout moment, il est possible d'appeler la méthode interceptée en appelant *proceed* ().

Lié à une méthode interceptée de l'UC, il est possible d'avoir un ou plusieurs contrats de chaque type. L'ensemble de tous les contrats représente le protocole d'extension sémantique ou bien le contrat global de cette méthode.

Pour déterminer la manière d'exécuter un ensemble de contrats attachés à une même méthode interceptée, nous définissons les règles d'ordonnancement suivantes :

- *Exécution* qui définit la manière d'exécuter les contrats d'un même type. Pour cela, il y a deux possibilités : séquentielle ou parallèle ;
- *Priorité* qui détermine explicitement l'ordre d'exécution des contrats d'un même type lorsqu'ils sont s'exécutés en séquentiel. Avec la *priorité*, nous pouvons spécifier explicitement l'ordre d'exécution des contrats d'un même type ;
- *Acceptation* qui permet de déterminer la condition pour laquelle le protocole d'extension d'une méthode est réussi. Il y a trois possibilités : (1) toujours réussi, (2) au moins un contrat est réussi et (3) tous les contrats ont réussi.

Nous avons adopté un protocole simple destiné à gérer l'exécution du contrat global d'une méthode interceptée.

- Des contrats d'un même type peuvent être exécutés en respectant les règles d'*exécution* et de *priorité* ;
- Les contrats de type *'Before'* sont exécutés en premier. A la fin de ces contrats, il faut vérifier la règle d'*acceptation*. Si le contrat global échoue, il faut envoyer la notification d'échec et arrêter l'exécution des autres contrats ;
- Les contrats de type *'Around'* sont ensuite exécutés. De la même manière, il faut vérifier que le contrat global est réussi ou échoué pour décider de continuer l'exécution des contrats ou d'envoyer une notification d'échec ;
- Les contrats de type *'After'* vont être exécutés si les contrats des autres types sont bien exécutés. A la fin de l'exécution de ces contrats, la notification réussie sera envoyée s'ils sont bien exécutés. Sinon, il s'agit de la notification d'échec.

Ce protocole est illustré graphiquement dans la Figure 109.

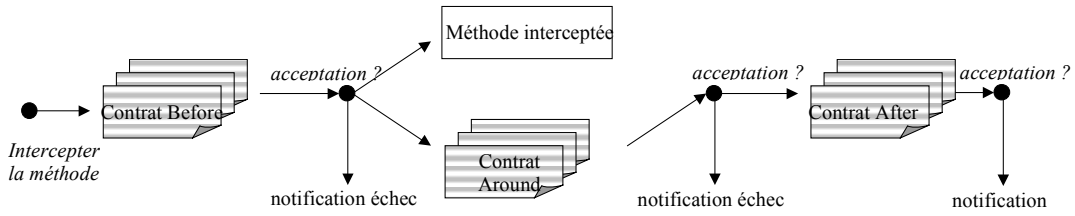

Figure 109. Protocole d'exécution des contrats attachés à une même méthode

4.5. Contexte d'exécution d'un contrat

Lors de l'exécution, chaque instance d'un contrat reçoit un objet qui représente son contexte d'exécution. Cet objet est créé au moment du déclenchement du contrat. Il contient les informations ci-dessus :

- Un objet, qui est l'instance de la classe de l'UC à laquelle cette instance de contrat est liée ;
- Une méthode qui représente la méthode interceptée de l'UC qui a déclenché le contrat ;
- Un ordre qui désigne l'ordre d'exécution du contrat par rapport à la méthode interceptée (i.e. *before, after, around*) ;
- Un booléen permettant de savoir si la méthode interceptée a déjà été exécutée ;
- Un objet qui contient le résultat de la méthode interceptée si elle est déjà exécutée. Sinon, il est nul ;
- Un ensemble d'attributs et leur valeur.

Tous les contrats attachés à une méthode interceptée partagent le même contexte d'exécution ; ils peuvent utiliser des opérations pour consulter ou modifier la valeur des attributs définis dans le contexte, ce qui leur permet de s'échanger des informations.

5. Composition de domaines

Concernant l'implémentation de la composition de domaines, notre plate-forme de support est constituée de plusieurs éléments :

- Editeur de composition de domaines et de configuration visant à définir les différentes facettes liées à la composition de domaines. Ces éditeurs seront présentés dans l'annexe A ;
- Le langage de relation et son compilateur visant à générer une partie des contrats horizontaux. Ceux-ci sont en cours d'élaboration afin de simplifier la gestion des relations. La grammaire de ce langage est déjà présenté dans le chapitre 4.
- Le moteur d'extension, qui connaît le concept de domaine et de composition de domaines.

En ce qui concerne le moteur d'extension, tout d'abord, il est important de noter qu'avec les services qu'il propose, le moteur de base est capable de supporter l'exécution des fédérations de domaines bien qu'il ne connaisse pas le concept de domaine et de composition de domaines.

Par conséquent, il sera à la charge du chargeur du moteur d'extension d'analyser les informations d'une fédération de domaines, produites par les éditeurs liés à la composition de domaines, et de passer les éléments nécessaires au moteur de base afin qu'il puisse exécuter cette fédération. En revanche, le moteur d'extension ne représente qu'un module simple au-dessus du moteur de base, chargé de fournir les informations concernant la composition de domaines. Ces informations sont, entre autres :

- La liste des modèles (de contexte) que chaque domaine peut charger et manipuler ;
- Les éléments fondants la fédération de domaines tels que les participants, les relations, les contrats horizontaux, etc.
- Les informations concernant la composition de l'application globale.

Pour l'instant, l'administration de l'exécution d'une application globale est totalement basée sur les services proposés par le moteur de base. Il s'agit d'un de nos objectifs dans le futur de mieux supporter l'exécution, l'administration et le déboguage d'une fédération de domaines.

6. Validation

Dans l'objectif de valider et d'évaluer notre approche, nous avons réalisé différentes applications en nous basant sur l'architecture de la fédération. Dans cette section, nous étudions les applications suivantes :

- Application APEL [EDA98, Est98, EAD99] et ses versions. Il s'agit d'un environnement de gestion de procédé qui est utilisé pour toutes les applications guidées par un procédé de notre équipe ;
- Application de gestion documentaire et ses diverses versions. Concernant cette application, nous présentons les besoins réels qui nous ont poussé à réaliser ces diverses versions et, petit à petit, à améliorer l'architecture de fédération afin d'obtenir l'architecture présentée dans ce document.

La comparaison, en terme de volume de code, d'extensibilité, etc. des diverses versions des applications ci-dessus nous aidera à valider et à montrer la viabilité de notre approche.

6.1. APEL

Le travail de notre équipe autour APEL a commencé au début des années 90 dans l'objectif de fournir un environnement de support complet pour la gestion des procédés logiciels [BEM91]. Quatre versions successives ont été construites d'abord pour donner naissance à un formalisme graphique de haut niveau visant à capturer et modéliser des

procédés ensuite pour l'exécution des procédés de haut niveau et enfin, pour le support à l'évolution dynamique des procédés logiciels. Ces quatre versions étaient des systèmes monolithiques, ce qui rendait leur évolution difficile.

6.1.1. APEL V5

Au début des années 2000, notre objectif fut de casser le monolithisme d'APEL V4 afin de mieux le faire évoluer. Pour cela, nous avons construit la version APEL V5 en combinant des composants à travers du système de messagerie JMS (*Java Message Service*).

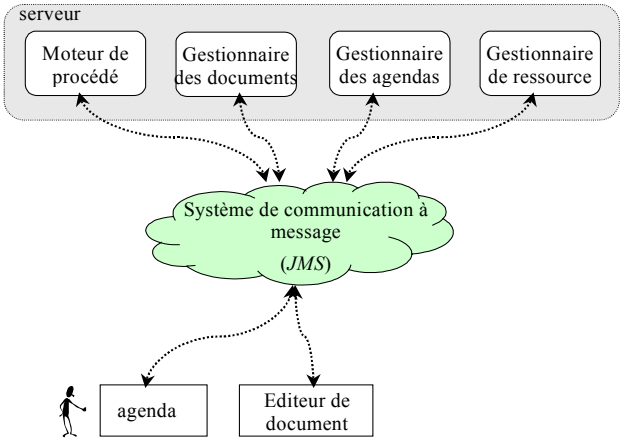

Figure 110. Architecture d'APEL V5

A cette époque, cette architecture représentait une architecture extensible et évolutive dans laquelle le découplage entre composants est faible. L'ajout ou bien le remplacement d'un composant par un autre peut être fait assez simplement.

Concernant la version V5 d'APEL, les chiffres sont assez intéressants :

- Il y a, au total, 968 classes avec environ 160K lignes de code ;
- Le fichier *'apelV5.jar'* fait 1,40 MB.

Lors de l'usage d'APEL V5 dans des entreprises, apparut le besoin de réutiliser des outils existants tels qu'un gestionnaire de ressources, un gestionnaire de produits, des éditeurs de documents, etc. Par conséquent, est apparue l'idée de faire interopérer et de composer les applications existantes pour construire une application globale. C'était l'idée de base des fédérations. En réalité, APEL V5 peut être considéré comme une fédération anarchique.

6.1.2. APEL V5.1

APEL V5.1 est une version intermédiaire entre APEL V5 et APEL V6. Elle a été développé lors de la création du *'docManagerV2'* (cf. section 6.2.2).

APEL V5.1 est construit comme la composition de deux domaines : (1) le domaine de procédé comportant la gestion de flot de contrôle et de flot de donnée et (2) le domaine

des ressources. Cependant, à cette époque, le concept de domaine était préliminaire ; seule la partie concrète avait été identifiée. C'est pour cette raison que APELV5.1 fut rapidement remplacé par APEL V6.

6.1.3. APEL V6

APEL V6 a été développé lors de la création de *'docManagerV3'* (cf. section 6.2.3). APEL V6 a été construit comme étant la composition de domaines de procédé et de ressource, mais l'espace conceptuel de ces domaines est matérialisé explicitement.

En terme de fonctionnalité, APEL V6 a quelques fonctionnalités de moins qu'APEL V5. Il s'agit des fonctionnalités qui sont très peu utilisées telle que la gestion des diagrammes d'état des activités, etc. Cependant, lorsque nous avons besoin de ces fonctionnalités, il sera simple de les brancher car notre architecture de fédération est très extensible.

En terme de volume, le domaine de procédé contient de 378 classes dont 16 seulement sont des classes représentant son espace conceptuel. Le reste représente les participants de ce domaine. Les fichiers *'.jar'* de ce domaine sont de 986KB pour les participants (e.g. l' agenda, l'éditeur, etc. dont une grande partie de code est liés à l'interface graphique utilisateur) et 54KB pour l'UC. Le nombre de contrats verticaux est de 2.

Le domaine de ressource contient 27 classes dont 2 seulement représentent les concepts de son espace conceptuel. Le fichier *'.jar'* de ce domaine est de 62KB. Le nombre de contrats verticaux est de 5.

Le domaine de document contient 21 classes dont 6 matérialisent les concepts de son espace conceptuel. Les fichiers *'.jar'* de ce domaine font 21 KB pour l'UC et 23 KB pour les adaptateurs et les participants manquants de ce domaine.

Le nombre de contrats horizontaux faisant le lien entre ces trois domaines est de 6 au total. L'ensemble de tous les contrats horizontaux et verticaux d'APEL V6 contient environ de 500 lignes de code.

6.2. Application de gestion documentaire

6.2.1. Version V1

Le cahier de charge initial de cette application a été présenté au début du chapitre 4. Il nous a été soumis en septembre 2001 par la société Actoll. La première implémentation, '*docManagerV1'*, fut livrée fin décembre 2001. Actuellement, cette version n'est plus en exploitation.

'*docManagerV1'* est construit en appliquant l'architecture présentée dans le chapitre 3. Ses caractéristiques sont les suivantes :

- APEL V5 est considéré comme un participant actif. Il impose le processus de production d'un document aux autres participants à travers l'UC ;

- L'UC matérialise explicitement les concepts partagés entre APEL V5 et CVS, le gestionnaire d'espace de travail (WSManager dans l'image), etc. ;
- L'UC contient 17 classes. La plupart des classes représentent des concepts d'APEL V5. Le reste matérialise les relations entre les concepts utilisés par d'APEL V5 et les autres participants ;
- Le nombre de classe pour réaliser les adaptateurs de participants ainsi que les participants manquants est de 38 ;
- Le fichier *'docManagerV1.jar'* (sans APEL V5) a une taille de 142KB;
- Il y six contrats verticaux qui font, au total, environ 500 lignes de code. Toutefois, ces contrats sont grands car ils contiennent toute la sémantique du scénario.

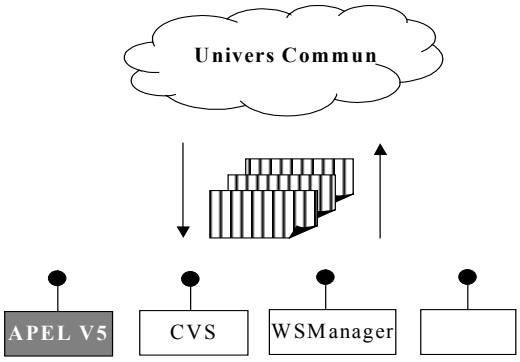

Figure 111. docManagerV1

Du point de vue fonctionnel, cette version répond bien à son cahier de charge. Néanmoins, il y a plusieurs points à considérer à propos de la réalisation de cette version :

- La réalisation de l'UC est difficile car les outils utilisés, étant très différents, n'avaient pas de concepts communs ;
- Le fait de changer un outil affecte profondément la réalisation de l'UC et l'ensemble des contrats verticaux. Un besoin d'encapsulation apparaît, afin qu'un changement n'affecte qu'une partie de la fédération ;
- La logique du scénario est spécifiée dans les contrats. Par conséquent, il faut être un expert de l'application afin de pouvoir faire varier son comportement. Ces petites variations peuvent être par exemple : (1) modifier le modèle (template) d'un document, (2) modifier la stratégie de versionnement d'un document, etc.. Apparaît ici le besoin d'avoir un certain degré de généricité pour faciliter les modifications légères du scénario par des utilisateurs « finaux » ;
- Certaines parties de la logique du scénario ne concernent qu'un sous-ensemble de participants et sont assez génériques. Par exemple, la logique de versionnement d'un document n'est liée qu'à des participants de gestion de documents. Apparaît ici le besoin d'avoir des modèles pour représenter et réutiliser la logique de scénario lié à un groupe de participants.

Les retours ci-dessus ont été provoqués l'apparition de la version '*docmanagerV2'* avec laquelle le concept de domaine et de modèles ont été introduit, mais de manière préliminaire.

6.2.2. Version V2

En juillet 2002, la version *'docManagerV2'* est mise en exploitation chez Actoll et chez les partenaires ; elle sera bientôt remplacée par la version *docmanagerV3*.

En terme d'architecture, cette version contient plusieurs améliorations :

- Le concept de domaine apparaît. Cependant, un domaine n'est défini que par la partie concrète (i.e. l'ensemble des participants qui réalisent les fonctionnalités du domaine). La partie conceptuelle d'un domaine n'a pas encore été identifiée ;
- '*docManagerV2'* est considéré comme étant la composition d'APEL V5.1 avec les participants du domaine de gestion d'espace de travail et ceux du domaine de document ;
- Seul l'espace conceptuel du domaine de procédé est matérialisé explicitement. Par conséquent, le domaine de procédé est central. Les autres domaines se synchronisent avec ce domaine en s'appuyant sur un ensemble de contrats « branchés » directement sur l'UC de ce domaine. De cette manière, les relations entre domaines sont traduites en termes de contrats « diagonaux ».
- Il y au total 20 aspects verticaux et diagonaux faisaient la composition de ces domaines ;
- Le fichier *'docManagerV2'* fait 170KB.

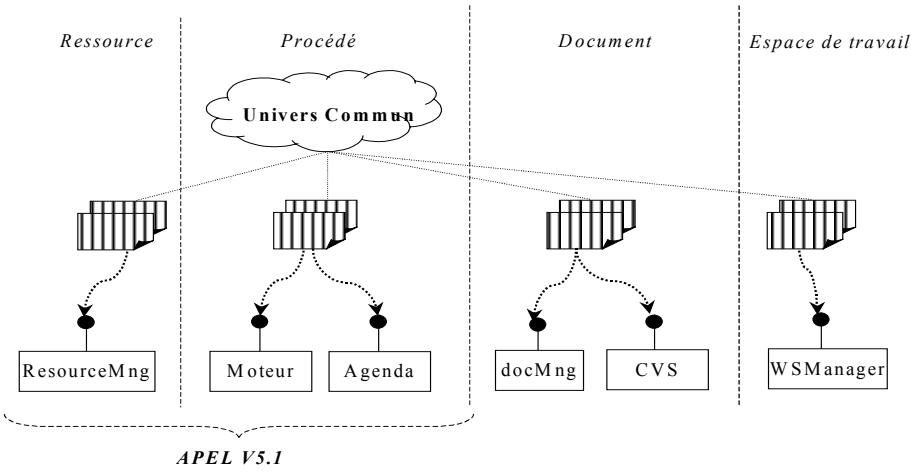

Figure 112. APEL V5.1 et docManagerV2

En terme de fonctionnalité, cette version offre des nouvelles fonctionnalités intéressantes:

- Il s'agit non seulement d'une application mais d'une famille d'application où les différents contextes d'utilisation sont modélisés par les modèles interprétés par des domaines ;

- Il est simple de modifier les modèles interprétés par des domaines. Ceci ne demande pas d'être un expert, mais un utilisateur « final » ;
- Il est possible de réutiliser des modèles de contextes d'utilisation des domaines.

Néanmoins, en terme de réalisation et d'évolution de l'application résultat, cette solution a plusieurs inconvénients qui sont, entre autres :

- Il n'y a pas d'une vision explicite de l'espace conceptuel de chaque domaine composé. Par conséquent, il faut traduire toutes les relations conceptuelles en terme des relations fonctionnelles avec l'aide des contrats « diagonaux » ;
- On ne peut pas composer des domaines entre eux, mais seulement avec le domaine central (i.e., le domaine de procédé dans notre cas).
- Les contrats « diagonaux » impliquent les difficultés pour l'évolution de l'application globale. Un changement de l'UC du domaine central implique de recalculer et de retraduire toutes les relations avec tous les autres domaines.

Ces inconvénients nous donnèrent l'idée de représenter les domaines par leur espace conceptuel, ce qui peut être utilisé pour représenter le contrat de composition du domaine. Ceci nous donne la naissance de la version *'docmanagerV3'*.

6.2.3. Version V3

Les exemples présentés tout au long du chapitre 4 et 5, font partie de la version *'docMangerV3'*. Cette version représente une amélioration de *'docMangerV2'* en matérialisant explicitement l'espace conceptuel de chaque domaine et en exprimant les relations entre domaines à travers les relations entre concepts. Elle utilise APEL V6 pour le guidage ; elle peut être considérée comme la composition de l'APEL V6 avec le domaine de document et le domaine d'espace de travail.

Pour donner une idée concrète *'docMangerV3'*, nous donnons quelques chiffres liés à son implémentation :

- Il y a quatre domaines avec environ une trentaine de classes qui représentent le méta-modèle de ces quatre domaines ainsi que les faux-concepts ajoutés ;
- Il y a environ une vingtaine de participants dans la réalisation de ces quatre domaines ;
- Il y a 6 relations entre concepts provenant des méta-modèles de ces domaines. Ces relations sont fixées quels que soient les modèles de contexte pris par ces domaines. La Figure 113 montre une synthèse de ces relations. En fonction des modèles pris, d'autres relations doivent être ajoutées pour spécifier des concepts de ces modèles ;
- Le nombre d'aspect horizontal pour gérer ces relations est de 20. Notons que pour chaque relation entre deux concepts, il faut avoir plusieurs contrats pour gérer la synchronisation entre instances concernées tout au long de leur cycle de vie.

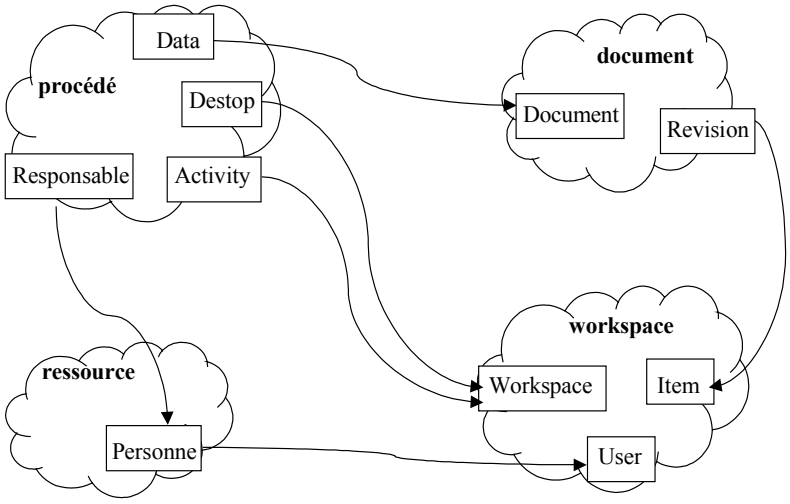

Figure 113. Résume des relations entre concepts provenant des méta-modèles

La réalisation de cette version nous a permis de tirer plusieurs enseignements importants :

- Pour chaque domaine, le méta-modèle est beaucoup plus petit et plus stable que la réalisation (i.e. des participants). Il est facile, sans affecter les autres domaines, de remplacer un participant par un autre à condition qu'il joue le même rôle ;
- La plupart des domaines peuvent s'exécuter en autonome. Ceci nous permet de bien tester les domaines indépendamment avant la composition de domaines pour être sur que les domaines sont réalisés correctement ;
- Il est possible de retenir des patrons concernant la réalisation d'un domaine. Ceci est lié à la manière de structurer son UC, d'attacher les contrats sur les méthodes de l'UC et la manière de répartir la sémantique entre l'UC, les contrats horizontaux, les contrats verticaux et les participants. Le travail actuel consiste à identifier l'ensemble des patrons et la méthodologie qui guidera la planification et la construction de domaines et la composition de domaines ;
- La gestion des propriétés non-fonctionnelles d'une composition de domaines n'est pas simple à faire. Ceci parce que les modèles conceptuels des domaines ne contient que la partie fonctionnelle des domaines.

6.3. Validation

Les versions de l'application de gestion documentaire ont beaucoup de différences en terme de fonctionnalité et en terme d'architecture ; leur comparaison est assez difficile.

Pour APEL, il est possible de comparer les versions APEL V5 et APEL V6 puisqu'il s'agit plus ou moins des mêmes fonctionnalités, mais réalisées par les architectures différentes. C'est pour cette raison que nous préférons comparer APEL V5 et APEL V6 sur deux facteurs importants : (1) volume de code et (2) extensibilité.

D'autres facteurs, tels que la stabilité et la performance, sont assez délicats à comparer car ceux-ci dépendent beaucoup la manière de programmer. Comme ces deux versions sont

écrites par des groupes de personnes différentes, nous préférons ne pas comparer ces facteurs.

6.3.1. Volume de code

APEL V6 est beaucoup petite qu'APEL V5 en terme de volume de code : APEL V6 contient 378 classes totalisant 57K lignes de code tandis que APEL V5 contient 968 classes avec environ 160K lignes de code.

Ceci s'explique, d'une part, par le fait qu'APEL V6 est exécuté sur le moteur de la fédération, qui propose beaucoup de services et de facilités de base et d'autre part, une partie de la sémantique d'APEL V6 est spécifiée par des contrats verticaux et horizontaux. Ces contrats contiennent environ de 500 lignes de code.

Néanmoins, APEL V6 est plus complexe et plus difficile à comprendre que APEL V5. Ceci, parce que la sémantique d'APEL est dispersée sur plusieurs éléments : les participants, les univers communs, les contrats verticaux et horizontaux. Il est nécessaire de comprendre non seulement ces éléments mais aussi la manière dont ils collaborèrent pour avoir une vision de la sémantique globale.

Nous pensons que cette complexité est raisonnable puisque nous plaçons dans un contexte de composition difficile. En plus, nous voulons faire une claire séparation abstrait-concret afin faciliter la composition à un haut niveau d'abstraction. En particulier, notre approche propose non seulement une application mais aussi une famille d'application. Le fait de disperser la sémantique nous permet d'une part, de personnaliser simplement le comportement de cette famille d'application et d'autre part, de réutiliser des domaines.

6.3.2. Extensibilité

Bien qu'APEL V5 soit extensible grâce à un système de communication par événement, il est clair APEL V6 est beaucoup plus extensible qu'APEL V5. L'extensibilité d'APEL V6 est prouvée par le fait que les participants sont bien isolés par le concept de domaine et que la composition de domaines est dirigée par composition de modèles conceptuels :

- Il est simple de changer un participant par un autre à condition qu'ils répondent au même contrat fonctionnel (i.e., ils implémentent un même rôle) ;
- L'ajout ou le retrait d'un participant d'un domaine est complètement transparent aux autres domaines à condition que le modèle conceptuel ne change pas ;
- L'ajout d'un domaine n'affecte pas l'implémentation des autres domaines.

7. Ma contribution

Les expérimentations dans ce chapitre représentent un travail volumineux. Celles-ci ont été réalisées par de plusieurs personnes. Ma contribution dans ces expérimentations est la suivante :

- J'ai réalisé la première version *fedeV1* du moteur de base avec : (1) le *framework* de l'adaptateur, (2) le serveur et le starter du moteur avec la communication *RMI* entre eux. *fedeV1* est basé sur les *proxies* dynamiques de `Java` pour instrumenter l'univers commun. Les services qu'il propose sont similaires avec ceux du *fedeV3* présentée dans ce chapitre ;

- J'ai réalisé les éditeurs de l'environnement de conception tout au long des trois versions du moteur *fedeV1*, *fedeV2* et *fedeV3* (excepté l'éditeur de l'univers commun V3 qui fait partie de la MOE). Ces trois versions diffèrent sensiblement en terme de modèle de fédération et de langage pour la représentation et le stockage des modèles. Au début nous avons utilisé *ObjectStore* comme la base de stockage ;

- J'ai réalise les deux premières versions de l'application gestion documentaire *'docmanagerV1'* et *'docmanagerV2'*. Dans ces versions, j'ai réalisé des participants de différents domaines, l'univers commun et tous les contrats pour coordonner des participants. La version *'docmanagerV3'* est en cours de réalisation en même temps que la rédaction de ce document.

8. Discussions et conclusions

Les expérimentations, que nous avons présentées dans ce chapitre, nous permettent de retenir des conclusions importantes concernant notre approche :

- La plate-forme de support est performante et stable. Elle permet, tout d'abord, de construire des applications comme un domaine ou une composition de domaines. Elle peut être utilisée pour supporter les applications à base de composants ou à l'AOP. Dû à sa dynamicité, elle est bien adaptée pour réaliser l'adaptation dynamique des applications à base de composants ;

- Les applications réalisées par notre approche, constituent un résultat prometteur, convaincant et intéressant. Il est intéressant de noter que ces applications ont été réalisées avec des objectifs différents par des personnes différentes. Nous avons constaté que même celles qui n'avaient pas une connaissance précise de l'implémentation pouvaient créer des fédérations. Ceci nous permet de dire que notre approche bien que complexe, peut être utilisée sans trop de difficultés ;

- Il reste encore des travaux à faire pour surmonter la complexité de notre approche lorsque l'application globale est construite par la composition de domaines. Ceux-ci sont surtout liés à la méthodologie, les supports pour déboguer et avoir une vision globale de l'architecture et surtout de la relation entre les éléments fondant l'application globale.

Chapitre VII

Conclusions et perspectives

1. Synthèse des travaux effectués

Cette thèse a pour objectif d'étudier une manière de structurer et de construire des applications par composition d'éléments logiciels, en particulier d'éléments existants. Dans cet objectif, les résultats que nous avons retenus peuvent être résumés comme suit :

- Etude de différents opérateurs de composition et les éléments de composition qui vont avec. Quatre opérateurs utilisés par les approches actuelles pour la construction des logiciels sont : connexion, intégration, coordination et orchestration ;

  L'intégration représente une manière de base niveau visant à tisser des éléments à composer afin d'obtenir une solution complète ; elle est utilisée par l'AOP et l'EAI ;

  L'approche des composants se base sur l'idée de remonter le niveau d'abstraction pour définir des composants en terme des responsabilités ainsi que des besoins en cachant leur structure interne. Par conséquent, les connexions sont suffisantes pour composer des composants ;

  Les conteneurs proposent une coordination prédéfinie pour composer des éléments indépendants ;

  L'orchestration est une variante de la coordination dans laquelle ce qui pilote les éléments logiciels impose également un objectif global. L'orchestration est proposée par l'approche BPM ;

- Etude de la coordination comme une solution pour la composition d'éléments qui sont des boîtes noires, hétérogènes et autonomes et qui partagent des concepts. Grâce à cette étude, nous avons proposé une fédération comme une architecture logicielle permettant de faire travailler ensemble des éléments logiciels afin d'atteindre un objectif global ;

- Proposition du concept de domaine et de la composition de domaine qui est dirigée par les modèles. Le concept de domaines nous permet d'isoler des groupes d'éléments qui sont liés afin de mieux les réutiliser. La composition de domaines propose une bonne manière de structurer et de réaliser des applications de grande taille en facilitant l'évolution et l'adaptation ;

- Construction d'une plate-forme de support pour la conception, l'exécution et l'administration des fédérations. D'une part, cette plate-forme est viable, stable,

performante et extensible. D'autre part, elle est assez générique afin de pouvoir d'être utilisé pour les systèmes à composants et/ou à l'AOP.

- Construction des différentes applications pour valider notre approche. Bien qu'il soit nécessaire de construire plusieurs applications dans différents domaines et de réaliser des tests plus avancés, ces expérimentations donnent un résultat significatif.

2. Leçons retenues

A travers la réalisation de cette thèse, au niveau conceptuel comme au niveau expérimental, nous retenons plusieurs leçons:

- La composition n'est pas un travail simple, surtout lorsque les éléments à composer sont des boîtes noires, hétérogènes et autonomes. La composition demande non seulement une manière de lier des éléments logiciels pour construire une application, mais aussi toutes les facilités visant à spécifier des éléments de composition et la logique de la collaboration entre ces éléments. La capacité de faire évoluer et adapter l'application résultante sont des critères essentiels pour distinguer différents modèles de composition ;

- A l'heure actuelle, il y a plusieurs modèles de la composition. Chaque modèle a ses caractéristiques et par conséquent a des avantages et/ou des inconvénients face à des contextes d'utilisation particuliers. Bien que beaucoup d'efforts en génie logiciel aient été consacrés à la composition, beaucoup reste à faire avant que le « programmeur » devienne réellement un « intégrateur » ;

- Bien que l'idée de fond de l'approche AOP soit intéressante, l'application de l'AOP rencontre plusieurs limitations et difficultés. Comme la plupart des implémentations de l'AOP se basent sur l'interception de méthode afin d'ajouter une sémantique additionnelle, nous pensons qu'il est important que la structuration et la réalisation des méthodes à intercepter soient faites en sachant quelles vont être interceptées et étendues. Ceci pour trois raisons principales. Premièrement, il est impossible d'intervenir au milieu d'une méthode interceptée. Deuxièmement, il est très délicat d'intervenir lorsque le programme de référence n'est pas dans un état stable. Troisièmement, il est difficile de passer le contexte d'exécution du programme de référence aux aspects, surtout dans le cas où des aspects devraient réagir en fonction du contexte du programme de référence ;

  Dans le cadre de nos expérimentations, à cause de ces limitations de l'AOP, d'une part nous avons du faire attention à la structuration du modèle conceptuel des domaines et d'autre part, nous tentons de remonter le niveau d'abstraction afin de cacher le plus possible les détails liés à des aspects. Pour ce faire, nous avons introduit plusieurs langages : langage de contrat, langage de relation et le langage d'activité. Deux premiers langages sont génériques tandis que le dernier est spécifique à la fédération de déploiement. Ce langage spécifique permet d'exprimer, pour chaque activité abstraite d'APEL, les actions de déploiement des participants de cette fédération [Les03] ;

- La composition par coordination est captivante par le fait que la logique de la collaboration entre des éléments logiciels est gérée séparément. D'une part, ceci facilite l'évolution et l'adaptation de l'application globale et d'autre part, nous pouvons nous baser sur des langages dédiés pour représenter explicitement la logique de collaboration entre éléments logiciels. La coordination est la plus appropriée dans le contexte où les éléments à composer sont hétérogènes, autonomes et où ils partagent des concepts ;
- Il est nécessaire de réfléchir et de travailler à un niveau d'abstraction plus haut que le code. Avec les modèles conceptuels, nous pouvons étudier, discuter et valider la composition au niveau conceptuel, sans nous préoccuper des détails liés à l'implémentation de ces modèles conceptuels. De cette manière, l'approche MDA promet d'importants progrès dans le domaine du génie logiciel.

3. Perspectives

Les perspectives de cette thèse peuvent être organisées selon deux axes : (1) les travaux à moyen terme et (2) les recherches à plus long terme.

En ce qui concerne les travaux à moyen terme, nous avons identifié :

- Clarification et taxonomie des relations plus précises. Ceci facilite la génération automatiquement des contrats gérant ces relations ;
- Amélioration du langage de relation afin de tenir compte de la relation entre un modèle conceptuel et son implémentation. Ceci va faciliter la composition des domaines génériques avec lesquels il n'existe pas une relation directe entre le modèle conceptuel et son implémentation ;
- Construction de formalismes graphiques afin d'exprimer une vision globale et complète des fédérations. Actuellement, nos éditeurs ne permettent pas d'avoir une vision complète d'une fédération;
- Construction d'un environnement de conception avancé dans lequel les développeurs de fédérations sont guidées, étape par étape, afin de faciliter leur travail. La validation et le test d'une fédération sont également très importants lors du développement des fédérations ;
- Construction d'un moteur multi-fédérations. Ceci est très demandé par nos partenaires industriels qui ont en permanence plusieurs fédérations en cours d'exécution. De ce fait, un moteur doit supporter l'exécution de plusieurs fédérations à la fois. Ceci n'est pas un travail simple car différentes fédérations peuvent avoir différentes configurations et/ou différentes versions d'un domaine ;
- Amélioration du débogage et de l'administration des fédérations. Actuellement, ces services sont trop limités.

A plus long terme, nous avons identifié plusieurs pistes de recherche :

- Gestion des propriétés non-fonctionnelles (NF) dans la composition de domaines. Bien que la gestion des propriétés NF ne soit pas un travail simple dans le cas général, elle est encore plus difficile pour la composition de domaines. Ceci parce que les propriétés NF d'un domaine ne sont pas visibles à l'extérieur. Une solution potentielle est de représenter et de gérer les propriétés NF comme étant des domaines NF (e.g. domaine de persistance, domaine de transaction, etc.). L'application globale sera le résultat de la composition des domaines fonctionnels avec des domaines NFs ;

- Gestion de l'évolution dynamique et de la récupération d'erreur des fédérations. Actuellement, nos protocoles de récupération d'erreur (cf. annexe A) sont préliminaires. Ils ne sont pas suffisants pour faire à des situations dans lesquelles il est nécessaire d'analyser et de diagnostiquer l'état de la fédération en cours afin de proposer une solution adéquate pour régler le problème ou pour faire évoluer cette fédération. Il est également nécessaire d'avoir des langages de haut niveau d'abstraction permettant de configurer et de paramétrer des stratégies d'évolution et de récupération des erreurs ;

- Etude des possibilités pour une transformation semi-automatique (i.e., avec la participation des concepteurs) d'un domaine spécifique en un (ou plusieurs) domaine générique. Cette transformation consiste à construire : (1) un méta-modèle pour le domaine générique, (2) un éditeur pour définir des contextes potentiels pour ce domaine, (3) un contrôleur de domaine et (4) un interpréteur du domaine générique.

4. Conclusions

Les travaux de notre équipe autour de la composition d'éléments logiciels existants ont été commencés depuis des années. Les propositions, présentées dans cette thèse, représentent un pas important vers cet objectif.

Bien que nous n'ayons pas exactement la même problématique que les approches de construction des logiciels par la composition, nous avons quand même essayé de bénéficier des points positifs de ces approches afin de concevoir notre architecture de fédération. De ce fait, une fédération est une architecture logicielle qui peut être utilisée dès à présent pour réaliser un système basé sur des technologies à composant, l'AOP, le BPM, et même le MDA.

Une fédération est essentiellement une architecture logicielle et une approche méthodologique permettant la structuration et la construction des applications par composition. Elle s'est révélée viable et a déjà prouvé ses points forts et ses performances. Avec deux axes de composition, verticale et horizontale, nous pouvons séparer nos préoccupations en deux temps : lors de la composition horizontale, nous nous concentrons sur les modèles conceptuels des domaines en ignorant délibérément leur réalisation; en revanche, lors d'une composition verticale, nous ne considérons que la liaison entre les éléments, abstraits et concrets, d'un domaine.

Chapitre VIII

Bibliographie

[Aal et al 99] W.M.P. van der Aalst, T. Basten, H.M.W. Verbeek, P.A.C. Verkoulen, M. Voorhoeve. “Adaptive workflow. On the interplay between flexibility and support”. Proceedings of the first International Conference on Enterprise Information Systems, Vol. 2, p.353-360. Setúbal, Portugal, March 1999.

[Actoll] Projet Centr'Actoll. Disponible à : http://www-adele.imag.fr/Les.groupes/centractoll/index.html

[AFGK02] L. Andrade, J.L. Fiadeiro, J. Gouveia, G. Koutsoukos. "Separating computation, coordination and configuration". Journal of Software Maintenance 353-369. 2002.

[AJ] Xerox. AspectJ. http://aspect.org.

[AK02a] D. H. Akehurst, S. Kent. “A relational approach to defining transformations in a meta-model”. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings, volume 2460 of LNCS, pages 243–258.

[AK02b] C. Atkinson, T. Kühne. "The Role of Meta-modeling in MDA". In Jean Bezivin and Robert France, editors, Workshop in Software Model Engineering, Germany, October 2002.

[AK03] C. Atkinson, T. Kuhne. “Model-Driven Development : A Metamodeling Foundation ”. IEEE Software, September/October 2003. page 36-41.

[Aks et al 94] M. Aksit, J. Bosch, W. van der Sterren, L. Bergmans. "Real-Time Specification Inheritance Anomalies and Real-Time Filters". In Proceedings of the 8th European Conference on Object-OrientedProgramming (ECOOP'94), Lecture Notes in Computer Science, Vol. 821, pages 386-407, Springer-Verlag, Bologne, Italie, Juillet 1994.

[Aks et al. 93] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa. "Abstracting Object Interactions Using Composition-Filters". In Object-Based Distributed Processing, sous la direction de R. Guerraoui, O. Nierstrasz and M. Riveill, Springer-Verlag, pp. 152-184, 1993.

[Ami99] M. Amiour. "Vers une Fédération de Composants Interopérables pour les Environnements Centrés Procédés Logiciels". Thèse de doctorat de l’Université Joseph Fourier, Grenoble, France, juin 1999.

[APR00] D. Arregui, F. Pacull, M. Riviere. “Heterogeneous Component Coordination:

the CLF Approach". Proceedings of the Fourth International Enterprise Distributed Object Computing Conference (EDOC'00), Japan, September 2000.

[Area] Le site de la société Area http://www.area-autoroutes.fr/

[Ark02] A.Arkin et al. "Web Service Choreography Interface 1.0". Disponible à: www.sun.com/software/xml/developers/wsci/wsci-spec-10.pdf, 2002.

[Ask] Le site de la société Ask http://www.ask.fr/

[BA99] B. Boehm, C. Abts. "COTS Integration: Plug and Play ?". IEEE Computer, pp. 135-138, January 1999.

[Bal97] R. Balzer, "Instrumenting, Monitoring, & Debugging Software Architectures", 1997. Disponible à http://citeseer.nj.nec.com/411425.html

[Bar et al. 01] M. Bartorello, H. Maguin, M. Blay-Fornarino, A-M. Dery, M. Reveill. "Adjonction de services au sein d'un serveur d'EJB". Journées composants 2001, Besançon, 25 et 26 octobre 2001.

[BB02a] J. Bézivin et X. Blanc. "MDA : vers un important changement de paradigme en génie logiciel". Développeur Référence. Juillet 2002. http://www.devreference.net/

[BB02b] J. Bézivin et X. Blanc. "Promesses et interrogations de l'approche MDA". Développeur Référence. Septembre 2002. http://www.devreference.net/

[BBB00] F. Bachman, L. Bass, C. Buhman, S. Comella -Dorda. "Volume II: Technical Concepts of Component-Based Software Engineering". Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon University, Mai 2000.

[BCK98] L. Bass, P. Clements, R. Kazman. "Software Architecture in Practice", Addison Wesley, 1998.

[BEM91] N. Belkhatir, J. Estublier, and W. L. Melo. "Software process modeling in Adele: The ISPW-7 example." In I. Thomas, editor, Proc. of the 7th Int'l Software Process Work-shop, San Francisco, CA, October 16–18 1991. IEEE Computer Society Press.

[Bez et al 03] J.Bézivin, N. Farcet, J-M. Jézéquel, B. Langlois, and D. Pollet. "Reflective model driven engineering". In G. Booch P. Stevens, J. Whittle, editor, Proceedings of UML 2003, San Francisco, volume 2863 of LNCS, pages 175--189. Springer, October 2003.

[Bez01] J. Bézivin. "From Object Composition to Model Transformation with the MDA". TOOLS'USA 2001, Santa Barbara, August 2001, Published in Volume TOOLS'39, IEEE publications, Los Alamitos, California, ISBN 0-7695-1251-8, pp. 346-354.

[BG99] R.M. Balzer, N.M. Goldman. "Mediating Connectors". ICDCS Workshop on Electronic Commerce and Web-Based Applications. May 31 - June 04, 1999. Austin, Texas.

[BP02] M. Belaunde & M. Peltier. "From EDOC components to CCM components: a precise mapping specification", Fundamental Approaches to Software Engineering (FASE), Grenoble, France, April 2002.

[BPEL4WS] "Business Process Execution Language for Web Services version 1.1". Disponible à http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

[BPML] "Business Process Modelling Language". Disponible à http://www.bpmi.org/

[Car97] D. Carney. "Assembling Large Systems from COTS Components. Opportunities, Caution and Complexities". SEI Monograph on Use of Commercial Software in Government Systems. SEI, Pittsburgh, USA, June 1997.

[Cas et al.00] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, M-C Shan. "Adaptive and Dynamic Service Composition in eFlow". CaiSE 2000, LNCS 1789, pp13-31, 2000.

[Castor] Le projet Castor. Disponible à : http://castor.exolab.org/.

[CCM02] OMG, "CORBA Components – Version 3.0". Juin 2002. Disponible à : http://www.omg.org/technology/documents/formal/components.htm

[CE93] M. C. Cooper, and L. M. Ellram. "Characteristics of Supply Chain Management and the Implications for Purchasing and Logistics Strategy". The International Journal of Logistics Management 1993.

[Cla02] S. Clarke. "Extending standard UML with model composition semantics". Science of Computer Programming, 44(1): 71-100, July 2002.

[COM] Microsoft. "COM Specification".

Disponible à: http://www.microsoft.com/com/resources/comdocs.asp

[Cre92] S. McCready. "There is more than one kind of Workflow Software". Computerwork, Novembre 2, 1992.

[CS02] G. Caplat, J.L. Sourrouille. "Model Mapping in MDA". Workshop in Software Model Engineering, 2002.

[CS02] J. Cardoso and A. Sheth. "Semantic e-Workflow Composition". Technical Report, LSDIS Lab, Computer Science, University of Georgia, July 2002.

[CSC02] "The emergencce of Business Process Management". Rapport de la CSC. Version 1.0, 2002.

[CVK99] J.N. Cosquer, P. Veríssimo, S. Krakowiak, and L. Decloedt. "Support for Distributed CSCW Applications". LNCS 1752, p. 295 ff, 1999

[CVS] CVS – Concurrent Version System. Disponible à http://www.cvshome .org/docs/manual/cvs.html

[DAML-OIL] "DMAL + OIL". Disponible à http://www.daml.org/2001/03/daml+oil-index

[DAMLS] "DARPA Agent Markup Language for Services". Disponible à http://www.ai.sri.com/daml/

[DEM01] F. Duclos, J. Estublier, P. Morat. "Environnement pour la Gestion d'Aspects Non-Fonctionnels dans un Modèle à Composants". ICSSEA, Paris, décembre 2001.

[DES02] F. Duclos, J. Estublier and R. Sanlaville. "Une Machine à Objets Extensibles pour la Séparation des Préoccupations". Journées systèmes à Composants Adaptables et extensibles. October 2002, Grenoble, France.

[DGD] A. Duret-Lutz, T. Géraud, and A. Demaille. "Design Patterns for Generic

Programming in C++". Disponible à : <http://www.lrde.epita.fr/dload/papers/coots01.html>

[DIC] "Le grand dictionnaire terminologique". Disponible à http://www.grand dictionnaire.com/btml/fra/r_motclef/index1024_1.asp

[DP] Dynamic Proxy Classes. Disponible à http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html.

[Dsouza01] D. D'Souza. "Model-Driven Architecture and Integration: Opportunities and Challenges" Version 1.1, Document available at www.kinetium.com, February 2001.

[Duc02] F. Duclos. "Environnement de Gestion de Services Non Fonctionnels dans les Applications à Composants". Thèse en informatique à l'Université Joseph Fourier (Grenoble), octobre 2002.

[Dui01] S. Duivestein. "Web Services and Workflow". Web Services Architect (www.webservicesarchitect.com), Sep 2001.

[EAD99] J.Estublier, M. Amiour, S. Dami. "Building a federation of Process Support System". WACC Work, Activity Coordination and Cooperation; Siplan, Sigmod, Sigsoft conference; San Francisco 22, 26 February 1999. USA.

[EAFLO01] T. Elrad, M. Aksits, G. Kiczales, K. Lieberherr, H. Ossher. "Discussing aspects of AOP". Communications of the ACM. Volume 44, Issue 10 - October 2001.

[EDA87] J. Estublier and S. Dami and M. Amiour. "APEL, an effective Process Support Environment". First International workshop on Multi Facets Software Process Engineering. September 22-23. Tunis, Pages 123, 137. Tunisia. 1997.

[EDA98] J. Estublier, S. Dami, and M. Amiour. "APEL: A Graphical yet Executable Formalism for Process Modeling". Automated Software Engineering, ASE journal. Vol. 5, Issue 1, 1998.

[EGV03] J. Estublier, S. Garcia and G. Vega. "Defining and Supporting Concurrent Engineering policies in SCM". CM-11 May 2003, Portland, Oregon, USA.

[EL01] J. Estublier and A-T Le. "Design and development of Software Federation". 14èmes Journées Internationales "Génie Logiciel & Ingénierie de Systèmes et leurs Applications" - ICSSEA 2001. Paris, France. 4-6 December 2001.

[Elr et al.] T. Elrad, R.E. Filman, A. Bader. "Aspect-oriented programming: Introduction". Communications of the ACM. Volume 44, Issue 10 - October 2001.

[ELV03] J. Estublier, A-T. Le, J. Villalobos. "Using Federations for Flexible SCM Systems". Proceedings of the 11th International Workshop on Software Configuration Management (SCM-11), ICSE'2003, LNCS Series, Springer-Verlag, USA, May 2003.

[Est et al 03] J. Estublier, J. Villalobos, A-T. Le, S. Sanlaville and G. Vega. "An Approach and Framework for Extensible Process Support System". 9th European Workshop on Software Process Technology (EWSPT 2003) September 2003, Helsinki, Finland.

[Est et. Al 98] J. Estublier, S. Dami, and M. Amiour. "APEL: A Graphical yet Executable Formalism for Process Modeling". Automated Software Engineering, ASE

journal. Vol. 5, Issue 1, 1998.

[Est98] J. Estublier. “Federations of Process Support Systems”. Workshop on Process Modeling (WPM), 2 Octobre, 1998, Zurich, Suisse.

[FHS92] R. Freund, B. Haberstroh, C. Stary. "Applying Graph Grammars for Task-Oriented User Interface Development". In W. Koczkodaj, editor, Proceedings IEEE Conference on Computing and Information ICCI'92, pages 389-392, 1992.

[Fin98] L. Finch. "So Much OO, So Little Reuse". Dr. Dobb's Journal, disponible à: http://www.ddj.com/articles/1998/9875/, mai 1998.

[GHJ95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "Design Patterns : Elements of Reusable Object- Oriented Software". Addison-Wesley, 1995.

[GHS95] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. “An overview of workflow Management: From process modeling to workflow automation infrastructure”. Distributed and Parallele Databases, 3(2): 119-153, 1995.

[GKAF01] J. Gouveia, G. Koutsoukos, LF. Andrade, JL. Fiadeiro. “Tool support for coordination-based software evolution”. Technology of Object-Oriented Languages and Systems—TOOLS 38, Pree W (ed.). IEEE Computer Society Press: Los Alamitos CA, 2001; 184–196.

[GMH99] J. Grundy, W. Mugridge, J. Hosking. “Constructing Component-based Software Engineering Environments: Issues and Experiences”. Journal of Information and Software Technology, Special Issue on Constructing Software Engineering Tools, 1999.

[Gro01] W, Grosso. "Java RMI", O'Reilly, 2001.

[HC01] G. Heineman, W. T. Council, “Component-based Software engineering”. Addison Wesley, 2001.

[HG03] J.Hu, P. Grefen. “Conceptual Framework and Architecture for Service Mediating Workflow Management”. Proceedings 4th International Symposium on Collaborative Technologies and Systems; Orlando, Florida, USA, 2003; pp. 23-29

[HK03] J. H. Hausmann, S. Kent. “Visualizing Model Mappings in UML”. SOFTVIS 2003: 169-178.

[HSW01] J.J. Halliday, S.K. Shrivastava and S.M. Wheater. “Flexible Workflow Management in the OPENflow system”. Fifth IEEE International Enterprise Distributed Object Computing Conference. September 2001.

[JAC] Java Aspect Components. Disponible à : http:// jac.aopsys.com

[JF88] R. E. Johnson, B. Foote, “Designing Reusable Classes”, Journal of Object-Oriented Programming, Vol. 1, No. 2, June 1988, pp. 23-35.

[JW] R. Johnson, B. Woolf. “Type Object Pattern”. Disponible à : http://www.ksc.com/article3.htm.

[Kic et al.] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold. “Getting started with ASPECTJ”. Communications of the ACM. Volume 44, Issue 10 - October 2001.

[Kiz92] G. Kiczales. "Towards a New Model of Abstraction in the Engineering of Software". In Proceedings of IMSA'92. Workshop on Refection and Meta-

Level Architecture, 1992. (pp 113, 115, 135, 152, 153)

[KK03] M. Katara, S. Katz. “Architectural view of Aspects”. Proceedings of the 2nd international conference on Aspect-oriented software development. ISBN:1-58113-660-9. 2003.

[KK03] M. Katara, S. Kats. “Architectural views of aspects”. ACM Press. ISBN: 1-58113-660-9. 2003.

[KLM97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwin. "Aspect-Oriented Programming". Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP’97). Lecture Notes in Computer Science vol.1241, Springer-Verlag, Juin 1997.

[KRB92] G. Kiczales, J. des Rivieres, D. G. Bobrow. "The Art of the Metaobject Protocol". The MIT Press, Cambridge, Massachusetts, 1991. ISBN 0-262-11158-6 (hc.), second printing, 1992.

[KYX03] J. Kienzle, Y. Yu, J. Xiong, “On Composition and Reuse of Aspects”. In Proceedings of the FOAL 2003 (Foundation of Aspects Oriented Languages). Boston, March 17, 2003.

[Lan03] A. Langer. “C++ Expression Templates. An Introduction to the Principles of Expression Templates”. C/C++ Users Journal, March 2003. Disponible à : http://www.langer.camelot.de/Articles/Cuj/ExpressionTemplates/ExpressionTemplates.htm

[LB03] V. Lestideau and N. Belkhatir. “Providing Highly automated and generic means for software Process deployment”. 9th European Workshop on Software Process Technology (EWSPT 2003) September 2003, Helsinki, Finland.

[LBC02] V. Lestideau, N. Belkhatir and P.Y. Cunin. “Towards automated software component configuration and deployment”. PDTSD'02 July 2002, Orlando, Florida, USA.

[Le02] A-T. Le. “Process Support for Tools Interoperability”. Doctorial Sposium at 17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK. IEEE Computer Society 2002, ISBN 0-7695-1736-6.

[Lemesle00] R. Lemesle. “Techniques de modélisation et de méta-modélisation”. Thèse, Université de Nantes, octobre 2000. http://www.sciences.univ-nantes.fr/info/lrsg/Pages_perso/RL/Richard_2001.htm

[Les02] V. Lestideau. "Un Environnement de Déploiement Automatique pour les Applications à Base de Composants". International Conference "Software & Systems Engineering and their Applications" ICSSEA'02, France, December 2002.

[Les03] V. Lestideau. "Modèles et environnement pour configurer et déployer des systèmes logiciels". Thèse de doctorat LLP-ESIA, Université de Savoie, Annecy, décembre 2003.

[LEV03] A-T. Le J. Estublier and J. Villalobos. “Multi-Level Composition for Software Federations”. ETAPS-2003, Workshop on Software Composition (SC'2003), ENTCS Vol. 82, No. 5, Elsevier, Warsaw, Poland, avril 2003.

[LH95] C. V. Lopes, W. L. Hursch, “Separation of Concerns”, College of Computer Science, Northeastern University, Boston, February 1995.

[Lin00] D.S. Linthicum, “EAI Application Integration Exposed”, Software Magazine, Feb/Mar 2000, http://www.softwaremag.com/ archive/2000feb/EAI.html.

[LOO01] K. Lieberherr, D. Orleans, J. Ovlinger. “Aspect-Oriented Programming with Adaptive Methods”. Communications of the ACM, Vol. 44, No. 10, pp. 39-4 1, October 2001

[LR97] F. Leymann and D. Roller. “Workflow-based applications”. IBM System Journal. Vol. 36, No.1, p102-123, 1997.

[LRS02] F. Leymann, D. Roller, and M. Schmidt. “Web services and business process management”. IBM Systems Journal, Volume 41-2, 2002.

[LSH03] J. Lee, K. Siau , S. Hong. "Enterprise Integration with ERP and EAI". Communications of the ACM, Volume 46. Issue 2, February 2003.

[Lsr] Le site du laboratoire LSR – Logiciel Système et Réseau – http://www-lsr.imag.fr/lsr.html

[Lutz00] J.C. Lutz. «EAI Architecture Patterns ». EAI journal. March 2000.

[Mann99] J.E. Mann. “Workflow and EAI”. EAI Journal. September/October 1999.

[Mat00] M.Mattsson. “Evolution and Composition of Object-Oriented frameworks”, University of Karlskrona/Ronneby. Sweden. 2000.

[MDA01] “Model Driven Architecture Specification (MDA)”. Document No ormsc/2001-07-01. July 9, 2001.

[MDA03] “MDA Guide Version 1.0”. Document Number omg/2003-05-01. May 2003.

[Mey00] B. Meyer. "What to Compose". Software Development Magazine, mars 2000. Disponible à : http://www.sdmagazine.com

[Mey96] B. Meyer. "The Reusability Challenge". IEEE Computer, Component and Object Technology, février 1996.

[Mey97] B. Meyer. "Object-Oriented Software Construction". Prentice-Hall, second edition, 1997.

[Mey99] B. Meyer. "The Significance of Components". Software Development Magazine, novembre 1999. Disponible à : http://www.sdmagazine. com

[MM01] R. Marvie, P. Merle, "CORBA Component Model: Discussion and Use with Open CCM". Submitted to a Special Issue of the Informatica - An International Journal of Computing and Informatics dedicated to “Component Based Software Development”. 2001.

[MOF00] “Meta Object Facility (MOF) Specification”. OMG Document No 2000-04-03. Mars 2000, http://www.omg.org/technology/ documents/formal/mof.htm.

[MS88] D. Musser, A.A. Stepanov. “Generic Programming”. ISSAC 1988: page 13-25.

[MSZ01] McIlraith, S., Son, T.C. and Zeng, H. “Semantic Web Services”, IEEE Intelligent Systems. Special Issue on the Semantic Web. March/April, 2001. Copyright IEEE, 2001

[NetEAI] NetEAI. Disponible à : http://www.mediapps.com/web/site.nsf/$$pagesweb/products8

[OH98] R. Orfali, D. Harkey. "Client/Server Programming with Java and CORBA, 2nd Edition", John Wiley & Sons, March 1998.

[OSGI] "OSGi Service Gateway Specification", Release 1.0, Mai 2000.

Disponible à : http://www.OSGI.org

[PBG01] M.Peltier, J. Bézivin and G. Guillaume. "MTRANS: A general framework, based on XSLT, for model transformations", Workshop on Transformations in UML (WTUML), Genova, Italy, April 2001.

[PM01] E. Pitt, K. McNiff. "Java RMI: The Remote Method Invocation Guide", Addison Wesley, 2001.

[Pol01] J.T. Pollock, “The big issue: Interoperability vs. Integration”, EAI journal, October 2001, P48-52.

[Poo02] J. Poole. “Model-Driven Architecture: Vision, Standards And Emerging Technologies”. ECOOP 2002.

[PSD03] R. Pawlak, L. Seinturier, L. Duchien, “Jac Milestone 2003”, Research Report LIFL 2003-4

[RCS] RCS – Revision Control System. Disponible à http://www.cs.purdue.edu/ homes/trinkle/RCS/

[Rom99] E. Roman. "Mastering Enterprise JavaBeans and the Java2 Platform, Enterprise Edition", Wiley Computer Publishing, 1999.

[RSW98] F. Ranno, S.K. Shrivastava, and S.M. Wheater, “A Language for Specifying the Composition of Reliable Distributed Applications”, 18th IEEE Intl. Conf. on Distributed Computing Systems, ICDCS’98, Amsterdam, May 1998, pp. 534-543.

[SA98] R. Schmidt and U. Assmann. “Extending Aspect-Oriented-Programming In Order To Flexibly Support Workflows”. In Proceedings of the Aspect-Oriented-Programming Workshop at ICSE’ 98.

[SH02] J. Sutherland, W. Heuvel. "Enterprise Application Integration and Complex Adaptive Systems". Communications of the ACM, Vol. 45, No. 10, October 2002.

[She98] A.P. Sheth, “Changing focus on Interoperability in Information Systems: From system, syntax, structure to Semantics”. In Interoperating Geographic Information Systems, M. F. Goodchild, M. J. Egenhofer, R. Fegeas, and C. A. Kottman (eds.), Kluwer. 1998.

[Soley01] R. Soley et al, “Model Driven Architecture”, 2001. Disponible à http://www.kinetium.com/catalysis-org/publications/papers/2001-mda-Overview-00-11-05.pdf

[SPEM01] “Software Process Engineering Management”, OMG Document ad/2001-06-05, juin 2001.

[Szy02] C. Szyperski. “Component Software - Beyond Object-Oriented Programming”. Second Edition Addison-Wesley / ACM Press, 2002 ISBN - 201-74572-0.

[Tibco] Tibco. Disponible à : http://www.tibco.com/solutions/products/ business_ integration/process_management.jsp?m=d3.

[UML01] "Unified Modeling Language (UML), version 1.4", OMG Document formal/2001-09-67, Septembre 2001, http://www.omg.org/cgi-bin/doc?formal/01-09-67.

[Var02] D. Varro. "Towards Formal Verification of Model Transformations". In Proc. of PhD Student Workshop of FMOODS 2002, Formal Methods for Open Object-Based Distributed Systems. Univ. of Twente, the Netherlands 2002.

[Ver00] H. Verjus. "Conception et construction de fédérations de progiciels". Thèse de doctorat LLP-ESIA, Université de Savoie, Annecy, septembre 2001.

[VGD98] M.R. Vigder, W. M. Gentleman, J.C. Dean. "COTS Software Integration: State of the art". Rapport de recherché de NRC- CRC. Disponible à : http://wwwsel.iit.nrc.ca/abstracts/NRC39198.abs

[Vig98] M. Vigder. "An Architecture for COTS Based". NRC 41603. National Research Council of Canada. 1998. Disponible à http://wwwsel.iit.nrc.ca/projects/cots/COTSpg.html

[Vil03] J. Villalobos. « Fédération de composants : une architecture logicielle pour la composition par coordination ». Thèse en Informatique à l'Université Joseph Fourier (Grenoble) – juillet 2003.

[VK02] G. Valetto and G. Kaiser. "Using Process Technology to Control and Coordinate Software Adaptation". Columbia Computer Science. Technical report CUCS-021-02. 2002.

[VM94] J.L.Villarroel, P.R. Muromedrano. "Using Petri net models at the coordination level for manufacturing systems control". Robotics and Computer-Integrated Manufacturing, Vol. 11, No. 1, pages 41-50. 1994.

[VVP02] D. Varro, G. Varro, and A. Pataricza. "Designing the automatic transformation of visual languages". Science of Computer Programming, vol. 44(2):pp. 205–227, 2002.

[WAS90] A. Wasserman. "Tool Integration in Software Engineering Environments". In Software Engineering Environments: Internal Workshop on Environments, Berlin 1990.

[WC02] S. Weerawarana, F. Curbera. "Business Processes: Understanding BPEL4WS". IBM TJ Watson Research Center, August 2002. Disponible à : http://www-106.ibm.com/developerworks /webservices/library/ws-bpelcol1

[WfMC96] WfMC. "Workflow Management Coalition Terminology & Glossary", Workflow Management Coalition, Document No WFMC-TC-1011, June 1996.

[WfMC98] WfMC. " Interface 1: Process Definition Interchange Process Model", Workflow Management Coalition, Document No WfMC TC-1016-P, November 12, 1998.

[WSDL] « Web Service Description Language 1.1 ». Disponible à http://www.w3.org/ TR/wsdl#_wsdl

[WSR99] S.M. Wheater, S.K. Shrivastava, and F.Ranno. "OPENflow: A CORBA Based Transactional Workflow System". Advances in Distributed Systems, LNCS 1752, pages 354–374, 1999.

[XMI00] "XML Metadata Interchange (XMI) Specification v1.1", OMG Document formal/2000-11-02, November 2000, http://www.omg.org/ cgi-

bin/doc?formal/2000-11-02.

[XML00] “Extensible Markup Language 1.0”, Second Edition, October 2000, http://www.w3.org/TR/REC-xml

[XPATH] XPATH. Disponible à http://www.w3.org/TR/xpath

[XSLT] XSLT. Disponible à http://www.w3.org/TR/xslt

[YP02] J. Yang and M. P. Papazoglou. “Web Component: A Substrate for Web Service Reuse and Composition”. CaiSE 2002, LNCS 2348, pp21-36, 2002.

[ZBNN01] L. Zeng, B. Benatallah, P. Nguyen, and A. Ngu. “Agflow: Agent-based cross enterprise workflow management system”. In Proceedings of the 27th Int. Conference on Very Large Databases, September 2001.

Annexe A

1. Univers Commun et contrats

1.1. Éditeur de l'univers commun et de contrats verticaux

L'éditeur de l'UC permet de définir l'UC comme étant une application à étendre. Il produit, à la fin, un fichier *'.xml'* contenant la description des fichiers *'.jar'*, *'.zip'* fondant l'UC et la liste des extensions souhaités pour les classes de l'UC. Pour chaque classe, quatre types d'extensions sont possibles : (1) ajouter statiquement des nouveaux attributs, (2) ajouter des nouvelles interfaces, (3) intercepter des constructeurs et (4) intercepter des méthodes.

La Figure 114 montre l'image de cet éditeur en ouvrant l'univers commun du domaine de procédé. La partie gauche montre les classes de cet univers commun. La méthode *destroy* de classe *'ProcessInstance'* a été sélectionnée comme devant être interceptée. Les autres extensions de cette classe, définies dans les autres onglets de la partie droite de l'éditeur, ne sont pas montrées.

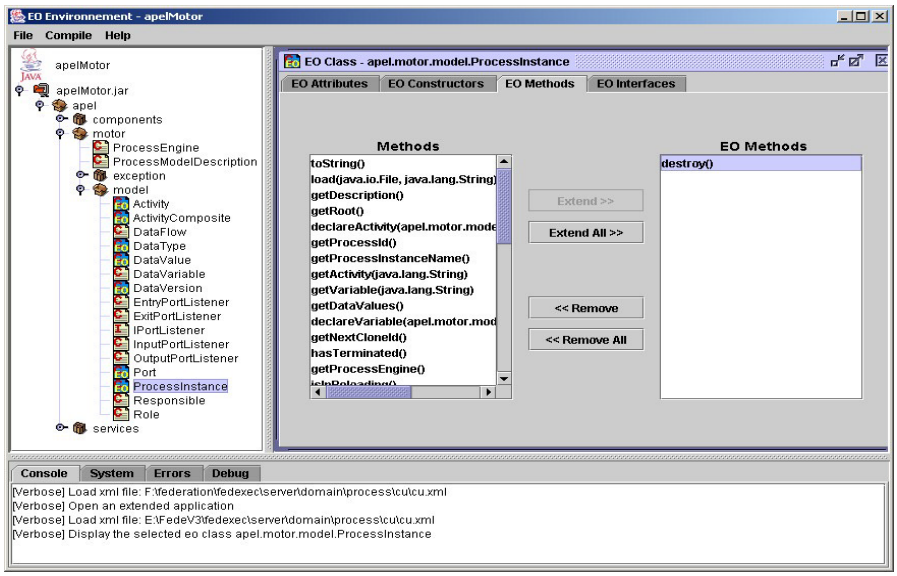

Figure 114. Editeur de l'univers commun

Il est important à noter que l'éditeur de l'UC et la MOE sont génériques et ne sont pas construits dans l'objectif d'être utilisé seulement pour la fédération [DES02]. C'est pour cette raison que l'éditeur de contrats verticaux est construit comme un *PlugIn* de cet

éditeur, bien que les contrats ne puissent être ancrés que sur les méthodes ou les constructeurs interceptés de l'univers commun.

Dans notre implémentation, nous avons utilisé le terme '*aspect'* afin de désigner des contrats. Ceci permet de rendre intuitive aux utilisateurs le concept de contrat en utilisant le vocabulaire de l'AOP.

Dans la Figure 115, nous sommes entrain de définir les aspects pour la méthode *destroy*() de la classe *ProcessInstance* de l'univers commun du domaine de procédé. Lié à cette méthode, il y a un seul aspect *destroyProcessInstance* qui est de type '*After'*. Le contenu de cet aspect, exprimé par notre langage d'aspect, est affiché dans la petite fenêtre en bas.

Les règles d'exécution le contrat global attaché à cette méthode, sont définis dans l'onglet *'Execution rules'* qui n'est pas montré dans cette image.

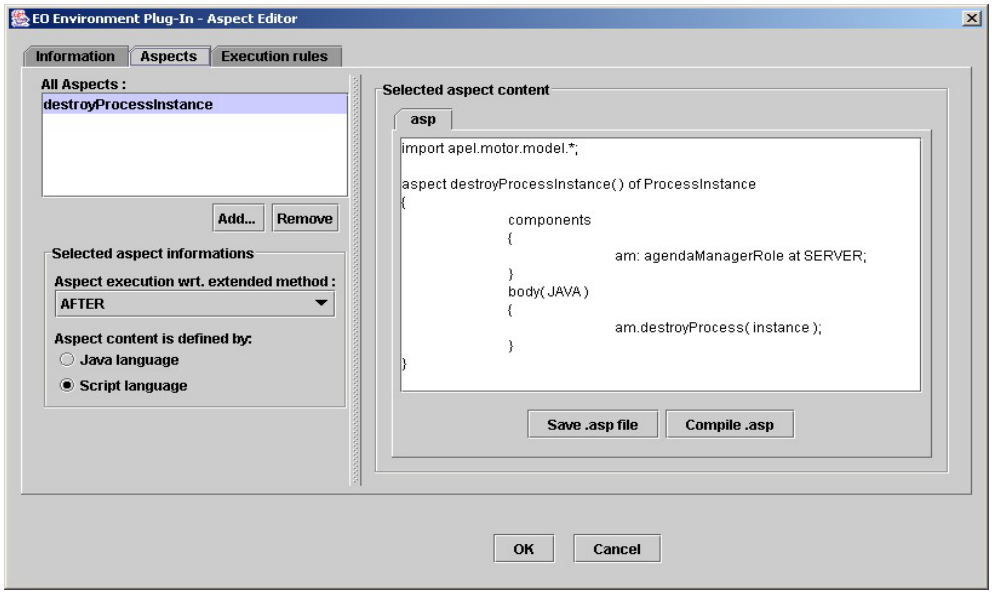

Figure 115. Editeur de contrats verticaux

Cet éditeur produit, pour chaque aspect, un fichier *'.xml'* décrivant les caractéristiques de l'aspect et un fichier *'.asp'* comportant le corps de l'aspect s'il est exprimé dans notre langage. Le fait de stocker séparément des aspects permet d'enlever/d'ajouter/remplacer, de manière simple, un aspect en fonction du contexte d'exécution de l'application comme nous avons abordé à la fin du chapitre 4.

1.2. Le langage et le compilateur de contrat

L'éditeur de contrats ci-dessus permet de définir la plupart des propriétés des contrats. Le langage de contrat que nous présentons dans cette section, complète la définition des contrats en spécifiant leur corps.

Tout d'abord, il est nécessaire de déclarer les informations générales concernant le contrat : la classe (i.e. représentée par *'type'*) sur laquelle il est installé et les conditions

dans lesquelles le contrat sera réellement déclenché (i.e. le trigger du contrat). Celles-ci sont exprimées comme suit :

<aspect> ::= aspect <name> "(" <parameters> ")" of <type> "{"
[<trigger>] [<participants>] <body> "}"

Le <type> désigne la classe de l'univers commun sur lequel l'aspect est installé.

Le trigger permet de spécifier les conditions pour exécuter réellement l'aspect. L'aspect est exécuté si l'expression de trigger est vraie.

<trigger> ::= when <expression>

<participants> exprime la localisation des instances de participants en se basant sur les services proposés par le moteur de participants :

<participants> ::= participants "{" <name> ":" <role> "at" <localization> [create | <new> | <find>] "}"

On peut spécifier la machine physique où le moteur localise l'instance. L'instance peut se trouver sur la machine serveur, une machine quelconque ou bien une machine particulière identifiée par un nom.

<localization> ::= SERVER | ANY | <machine>

Le moteur propose trois options pour localiser une instance : '*find'*, '*create'* et '*new'* parmi lesquelles le *'create'* est l'opération par défaut.

<create> ::= create [<attributes>]
<new> ::= new [<attributes>]
<find> ::= where <attributes>

Des attributs sont utilisés pour exprimer les caractéristiques non-fonctionnelles que l'instance à créer doit satisfaire. Certaines contraintes peuvent être liées aux caractéristiques physiques du participant, tels que le protocole de communication (*Socket* ou *RMI*), le *fire-wall,* etc. tandis que d'autres sont liées au modèle logique de participant, tels que le nombre d'instance, le protocole de démarrage, etc..

<attributes> ::= <attribute> ("," <attribute>)*
<attribute> ::= <name> "=" <value>

Par exemple, afin de chercher une instance d'un participant jouant le rôle `'fileTransfertRole'` sur une machine quelconque dont le mode de communication est *RMI*, on peut écrire comme suit :

```
participants
{
     ft : fileTransfertRole at ANY where communication=RMI
}
```

Figure 116. Exemple de localisation d'une instance

Le corps de contrat exprimant la coordination entre instances de participants commence par le mot clé `'body'`. Pour exprimer cette partie nous sommes basés, pour l'instant, sur `Java`. Cependant, nous avons ajouté quelques mots clés pour faciliter l'expression du corps de contrats :

- instance: pour désigner l'instance de l'UC lié à l'instance de l'aspect ;
- ctxt : pour désigner l'objet représentant le contexte d'exécution de l'aspect ;
- commonUniverse : pour désigner la racine de l'UC dont la classe fait partie.

Nous avons ajouté quelques instructions pour exprimer la manière d'exécuter l'aspect :

- fail: pour terminer explicitement l'exécution de l'aspect avec un échec ;
- end: pour terminer explicitement l'exécution de l'aspect avec succès ;
- proceed: pour appeler la méthode interceptée lorsque l'aspect est de type AROUND (i.e. il s'exécute à la place de la méthode interceptée).

Le compilateur d'aspects est chargé de générer des classes `Java` à partir des corps d'aspects exprimés dans notre langage. Ceci permet de rendre le moteur indépendant du langage d'aspect et en conséquence, de faciliter l'évolution le remplacement du langage actuel par un langage spécialisé dans la gestion de coordination dont CLF (*Coordination Language Facility*) est un exemple.

Le compilateur est intégré dans l'éditeur de contrats et dans le moteur. A chaque démarrage, le moteur vérifie s'il est nécessaire de recompiler des aspects en comparant la date de modification des aspects avec la date des classes générées.

2. Gestion de participants

2.1. Editeur de participants

Cet éditeur permet de définir les participants provenant d'un domaine ou bien d'une composition de domaine (e.g. les participants qui réalisent une relation horizontale, cf. section 4.3.4 du chapitre 4). La définition d'un participant est décomposée deux étapes :

- La définition des rôles, comme étant des participants abstraits ;
- La définition des participants concrets jouant un ou plusieurs rôles.

Cet éditeur se compose de quatre panneaux. Le panneau '*Library'* permet de spécifier les fichiers *'.jar'* contenant les classes qui implémentent les rôles, les participants, etc.

Le panneau '*Common Universe'* montre les méthodes interceptées de l'UCdu domaine ou de la composition de domaines (i.e. l'ensemble des UCs des domaines composés). Notons que ces méthodes sont également des événements potentiels de l'UC.

La Figure 117 montre le panneau de définition des rôles du domaine de procédé. La partie gauche montre la liste des rôles de ce domaine. La partie droite montre la définition du rôle sélectionné '*agendaManagerRole'*. Il s'agit d'un rôle qui propose une liste de services

regroupés dans une interface '*apel.components.agendaManager. AgendaManagerRole*' et qui doit être implémenté par un seul participant. L'onglet '*Right'*, qui permet de spécifier des droits du rôle, n'est pas montré.

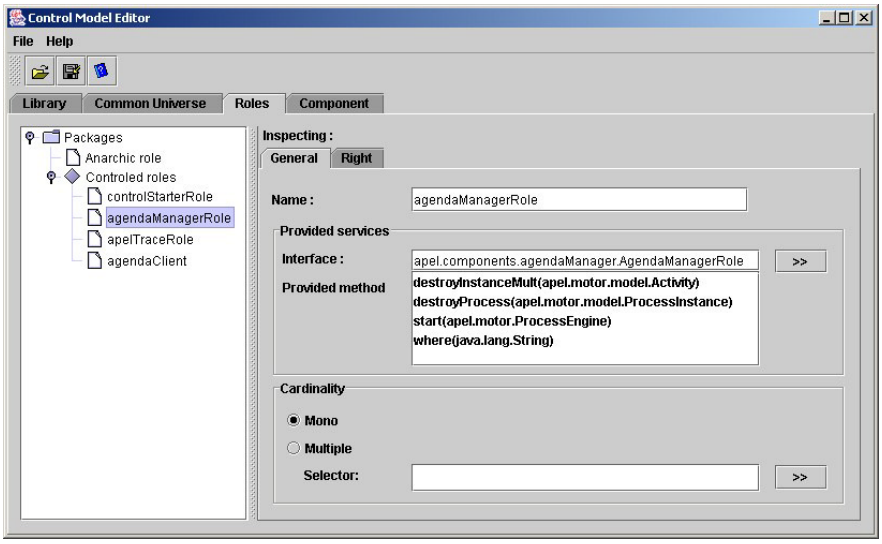

Figure 117. Définition de rôles

Le panneau de définition les participants concrets de ce domaine est illustré dans la Figure 118. La partie gauche montre la liste des participants en la séparant en deux catégories : local ou distant.

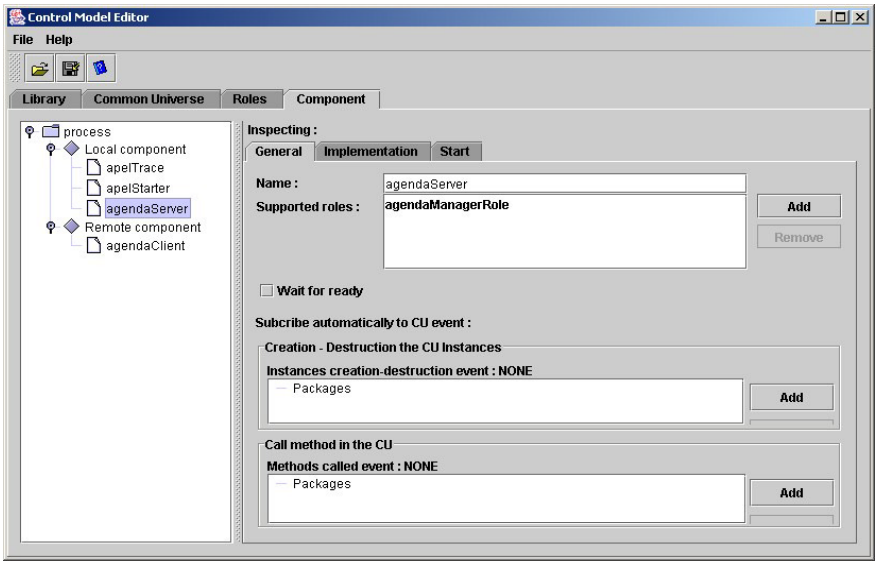

Figure 118. Définition de participants

La partie droite montre la description du participant '*agendaServer'*. Ce participant joue le rôle '*AgendaManagerRole*'. L'option '*Wait for ready'* permet de bloquer le moteur de fédération jusqu'à ce que le participant ait terminé son démarrage. Ceci est important lorsque le protocole de démarrage du participant est compliqué et surtout lorsqu'il est nécessaire que le moteur attende que le participant soit prêt avant de continuer son

exécution (et de lancer d'autres participants qui ont besoin du premier). L'ensemble des événements, émis par l'UC, auquel le participant souhaite souscrire automatiquement est spécifié en bas de cet onglet.

L'onglet '*Implementation'*, que nous n'avons pas montré dans cette figure, permet de définir l'implémentation de l'adaptateur du participant. Il est nécessaire de définir :

- la fabrique du mandataire, si le participant est de type local ;
- la fabrique du mandataire et de l'emballage, si le participant est de type distant.

Cet éditeur effectue quelques vérifications de cohérence simples :

- La fabrique du mandataire implémente efectivement les interfaces des rôles que le participant a déclaré jouer ;
- La mandataire et l'emballage doivent hériter des classes du *framework* de l'adaptateur (cf. section 2.2).

L'onglet '*Start',* n'est pas montré dans cette image, permet de spécifier la manière d'instancier un participant. Celle-ci comprend deux informations essentielles : où (i.e. la machine) et quand (i.e. le moment). Plusieurs options sont proposées dont la plus simple est de démarrer un participant lorsque la fédération à besoin d'une instance de ce participant sur une machine spécifique.

Cet éditeur produit un fichier *'.xml'* pour chaque rôle et pour chaque participant. Ceci permet de mieux ajouter/enlever/remplacer un rôle ou un participant par un autre.

2.2. Le *framework* de l'adaptateur

Le *framework* de l'adaptateur a été créé pour objectif de :

- Centraliser la gestion des propriétés non-fonctionnelles des participants, surtout la communication, la distribution, la récupération d'erreur, etc. ;
- Faciliter le codage de l'adaptateur des participants distants en héritant les classes de ce *framework*.

Ce *framework* contient un nombre de classes abstraites et d'interfaces qui implémentent, de façon générique :

- Les méthodes permettant d'initialiser le mandataire et l'emballage d'une instance d'un participant distant et à les mettre en correspondance ;
- Les méthodes permettant la communication par *Socket* entre le mandataire et l'emballage d'une instance d'un participant distant. Elles servent essentiellement à convertir un appel de méthode en un message et vice-versa ;
- Les méthodes permettant de gérer la synchronisation de l'exécution des participants avec le moteur. Ceci est important surtout pour la récupération d'erreur ;
- Les méthodes proposant des services de base pour tester ou bien arrêter des instances de participants (i.e. des opérations *ping*, *stop*, etc.).

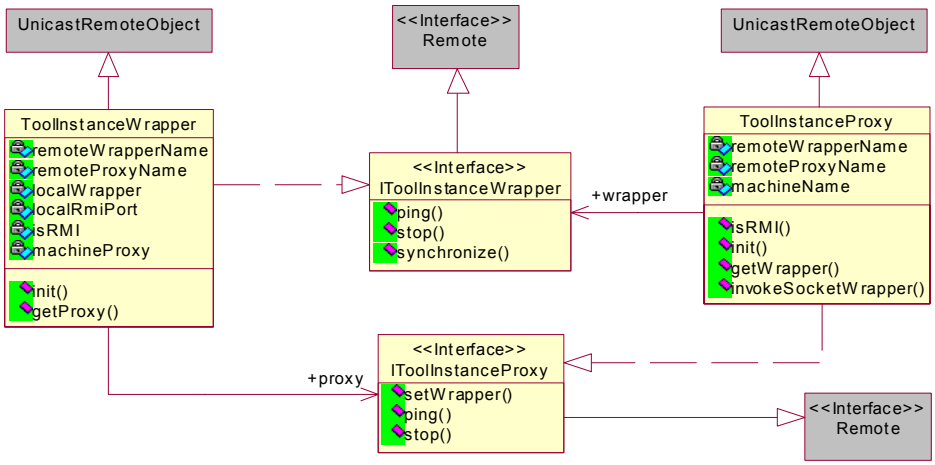

Figure 119. *Framework* de l'adaptateur

2.3. Protocole de récupération d'erreur

En parallèle avec le système de communication, nous avons construit un système de récupération d'erreur simple. Ce système permet de récupérer l'état d'exécution d'une fédération lorsqu'il y une rupture du côté le serveur ou bien du starter.

Lors de la récupération en cas d'arrêt du serveur, le protocole de récupération peut être résumé comme suit :

1. Le starter se connecte au registre *RMI* (*RMIregistry*) sur la machine du serveur afin d'essayer récupérer le *stub* représentant le serveur. Notons que nous sommes basés sur l'hypothèse qu'il n'y a pas de fire-wall du côté serveur ; la communication d'un starter vers le serveur est dans ce cas par *RMI* ;
2. Le starter teste le '*stub*' reçu afin de savoir s'il peut utiliser ce '*stub*' pour la connexion *RMI* avec le serveur ;
3. Le starter répète les étapes 1 et 2 jusqu'à ce qu'il arrive à se connecter au serveur ;
4. Le starter envoie au serveur la liste des emballages qu'il maintient pour demander au serveur de récréer les mandataires correspondants ;
5. Le starter demande aux emballages qu'il maintient de se synchroniser avec leur nouveau mandataire en consultant le registre *RMI* sur la machine serveur.

Le protocole de récupération lors de l'arrêt ou du blocage du starter peut être résumé comme suit :

1. Lorsque le service reçoit la demande du serveur, il teste si le starter est arrêté/bloqué ou s'il s'agit d'un problème de perte de connexion entre le serveur et le starter. Dans le dernier cas, le service demande au starter de se resynchroniser avec le serveur ;
2. Le service arrête le starter s'il est bloqué et relance une nouvelle session d'exécution du starter ;

3. Le nouveau starter doit, tout d'abord, rétablir sa connexion avec le serveur afin de récupérer l'état d'exécution actuel de la fédération ;
4. Le nouveau starter est chargé, ensuite, de récréer les emballages nécessaires en considérant les mandataires correspondants existants dans le serveur. Les emballages, à leur tour, sont chargés d'établir le lien avec leur mandataire respectifs.

Avec les deux protocoles de récupération ci-dessus, il est possible de récupérer l'état d'exécution d'une fédération, mais seulement en ce qui concerne l'existence des instances de participants. Notre moteur ne gère pas la persistance de l'univers commun, ni celle des participants. Ceci doit être implémenté par fédérations en fonction de leurs besoins.

3. Composition de domaines

3.1. Editeur de composition de domaines

Cet éditeur est l'éditeur le plus important parmi les éditeurs dédiés à la composition de domaines. Il permet de définir :

- L'ensemble des domaines composés ;
- Les relations entre classes venant de l'univers commun de ces domaines.

Le panneau, montré dans la Figure 120, est servi à définir les domaines composés est. Pour chaque domaine composé, il est nécessaire de spécifier la classe jouant le rôle du *'contrôleur de l'UC'* du domaine. Notons que le nom de la racine de chaque domaine (i.e. l'instance nommée du *contrôleur*) est calculé automatique par le moteur en combinant le nom du domaine avec *'Root'*.

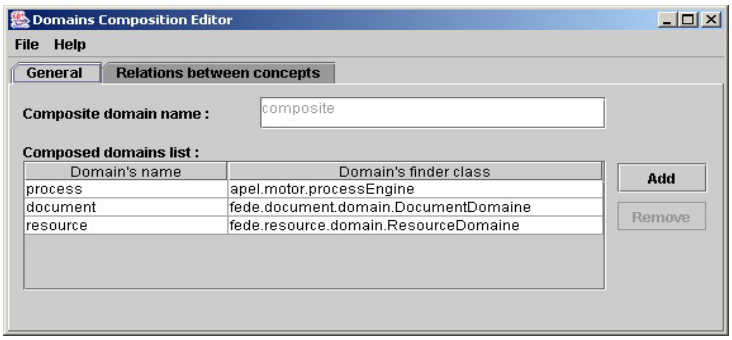

Figure 120. Domaines composés et leur *'finder'*

Le panneau, montré dans la Figure 121, permet de définir les relations entre classes provenant de l'UC des domaines composés. N'apparaissent, dans ce panneaux, que les méthodes étendues (cf. section 1.1) car on ne peut attacher de contrats que sur les méthodes interceptées des classes étendues.

En revanche, les classes du domaine destination sont des classes quelconques de l'univers commun du domaine destination.

Plusieurs relations ont été définies ; leurs caractéristiques principales, tels que le type, la cardinalité, etc. sont montrées en haut du panneau. Pour la relation sélectionnée (la relation `Data-Document` dans l'image), les classes concernées et les contrats pour gérer cette relation sont affichés en bas du panneau. Les contrats peuvent être exécutés avant, après ou à la place de la méthode étendue. Le contenu des contrats n'est pas montré dans cet éditeur.

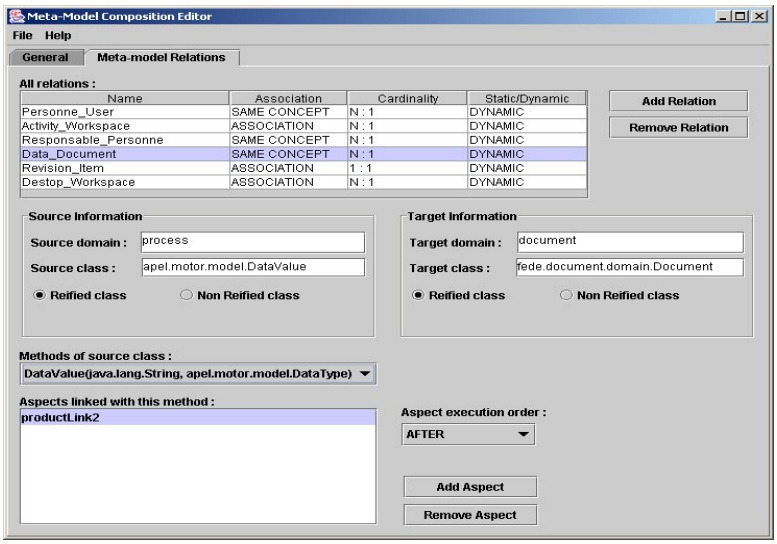

Figure 121. Définition des relations entre concepts

3.2. Editeur de configuration

Comme son nom l'indique, cet éditeur permet de définir les configurations d'exécution pour une fédération en spécifiant les contrats (i.e. les aspects dans l'image), les rôles, et les participants de chaque domaine composé.

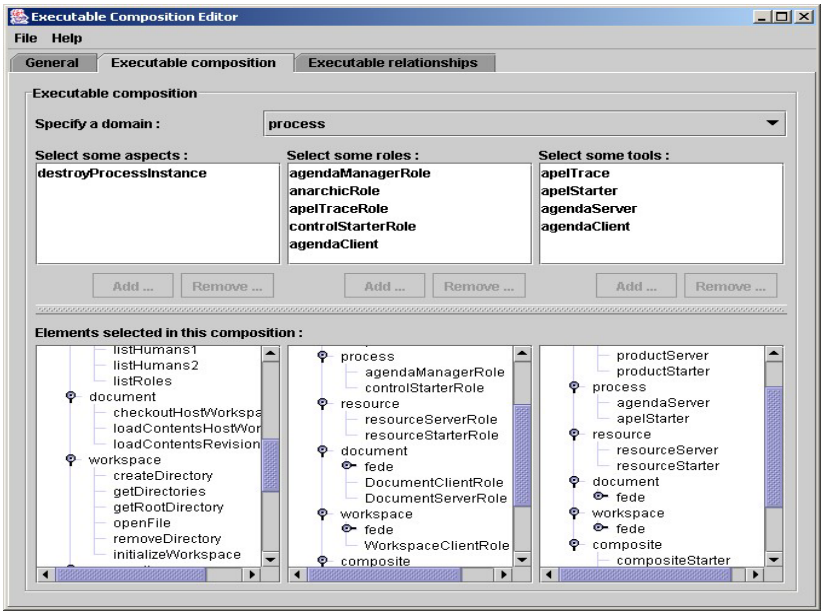

Figure 122. Définir une configuration

Dans la partie du haut, cet éditeur affiche, pour chaque domaine (dans l'image, le domaine de procédé), la liste de tous les contrats verticaux, les rôles, les participants, y compris les différentes versions s'il y en a, qui sont disponibles pour la sélection. Dans la partie du bas, l'éditeur montre la liste de tous les éléments sélectionnés de chaque domaine pour la configuration en cours de définition.

Cet éditeur réalise quelques vérifications de cohérence. Il analyse le contenu des contrats sélectionnés pour vérifier si tous les rôles nécessaires sont également sélectionnés. Pour chaque rôle sélectionné, il vérifie s'il y a un et seulement un participant jouant ce rôle s'il s'agit d'un rôle mono (i.e. le rôle qui peut être joué par un seul participant à un moment donné) et au moins un participant jouant ce rôle s'il agit d'un rôle multiple (i.e. le rôle qui peut être joué par plusieurs participants à un moment donné).

Dans l'onglet *'Excutable relationships'*, cet éditeur permet de définir l'ordre d'exécution des aspects, horizontaux et verticaux, attachées à une même méthode.

Cet éditeur produit à la sortie un fichier *'.xml'* de description de la configuration et un fichier *'.bat'*pour lancer l'application globale avec cette configuration. Dans ce fichier il calcule toutes les librairies nécessaires (i.e. le *classpath*). De cette manière, une fédération a autant de fichiers de lancement (i.e. ficher *'.bat'*) que de configurations.