FOCAS : un canevas extensible pour la construction d'applications orientées procédé

Gabriel Pedraza Ferreira

UNIVERSITE JOSEPH FOURIER - GRENOBLE I

THESE

Pour obtenir le grade de

DOCTEUR de l'Université JOSEPH FOURIER

Discipline : INFORMATIQUE

ECOLE DOCTORALE Mathématiques, Sciences et Technologies de l'Information, Informatique

Présentée et soutenue publiquement par

Gabriel Rodrigo PEDRAZA FERREIRA

Le 12 Novembre 2009

FOCAS : un canevas extensible pour la construction d'applications orientées procédé

Directeur de thèse :

Jacky ESTUBLIER

JURY

Président	Marie-Christine ROUSSET, Professeur à l'Université Joseph Fourier, Grenoble
Rapporteurs	Carlo MONTANGERO, Professeur à l'Université de Pise, Pise (Italie)
	Claude GODART, Professeur à l'Université Henri Poincaré, Nancy
Examinateurs	Claudia RONCANCIO, Professeur à l'INPG, Grenoble
	Hervé VERJUS, Maître de conférences à l'Université de Savoie, Chambéry
Encadrant	Jacky ESTUBLIER, Directeur de recherche au CNRS, Grenoble

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

Résumé

La récente introduction de l'approche à services a relancé la technologie des *workflow*. Cette technologie utilise le concept de modèle de procédé pour coordonner et automatiser la réalisation d'un ensemble de tâches. Ce patron de construction d'applications par assemblage de briques logicielles en utilisant un modèle de procédé fournit un mécanisme qui simplifie la spécification et l'évolution des *applications orientées procédé*.

Cette thèse s'intéresse à la conception, la spécification et l'exécution d'applications orientées procédé en général, et plus particulièrement à l'orchestration de services. Nous partons de la technologie *workflow* comme base de notre canevas FOCAS et nous proposons de suivre une approche d'ingénierie dirigée par les modèles (IDM) pour la spécification abstraite d'une orchestration. Dans FOCAS, la description abstraite de l'application est découplée des services (ou applications) supportant son exécution. Cette description permet d'abord, une indépendance vers la technologie utilisée pour l'implémentation de services, ainsi qu'une liaison dynamique à l'exécution aux services disponibles.

Nous proposons également des mécanismes permettant d'étendre notre canevas dans différents domaines ainsi que pour supporter des aspects non-fonctionnels. L'outillage supportant notre approche a été implémenté et validé par son utilisation dans les projets européens ITEA : S4ALL et SODA, et pour la réalisation d'une plateforme complète d'orchestration repartie et dynamique.

Mots clés : orchestration des services, applications orientées procédé, chorégraphie, *workflow*, composition de méta-modèles, CADSE, approche à services.

Summary

The recent introduction of service-oriented computing has improved the use of workflow technology. Workflow technology uses the concept of process model in order to coordinate and automate the execution of a set of tasks. This pattern of building applications assembling software pieces tailored by a process model provides a mechanism which simplifies specification and evolution of *process-based applications*.

This thesis deals with design, specification and execution of process-based applications in general, and particularly of service orchestrations. We use the workflow technology as basis of our framework FOCAS and we propose the utilization of the Model Driven Engineering (MDE) approach to create an abstract specification of a service orchestration. In FOCAS, this abstract description of an application is uncoupled from services (or applications) that support its execution. This abstract description allows both, independence of the technology used to implement services and binding with available services at runtime.

We also propose mechanisms to extend our framework in order to be used in different domains and to add support to non-functional aspects. Tools supporting the approach have been implemented. Our approach and its respective tools have been validated through their utilization in two ITEA European projects: S4ALL and SODA as well as by the implementation of a distributed and dynamic orchestration platform.

Keywords: services orchestration, process-oriented applications, choreography, workflow, meta-model composition, CADSE, service-oriented computing.

Remerciements

Je souhaite tout d’abord adresser mes remerciements à Messieurs Carlo Montangero et Claude Godart pour avoir voulu être les rapporteurs de cette thèse, ainsi que Mme. Marie-Christine Rousset, Mme. Claudia Roncancio et M. Hervé Verjus pour avoir accepté de faire partie des membres du jury de ma thèse.

Je tiens à remercier également mon directeur de thèse M. Jacky Estublier pour sa confiance, son soutien, ses conseils et son aide tout au long de ce travail. Je souhaite aussi remercier M. Pierre-Yves Cunin et M. Philipe Lalanda pour m’avoir accueilli dans l’équipe ADELE. Je remercie aussi, d'une manière générale, tous les membres de l'équipe ADELE, spécialement Germán pour sa collaboration et ses conseils et ainsi que Idrissa pour m’avoir supporté pendant trois ans en tant que collègue de bureau.

Mes sentiments les plus chaleureux sont pour ma famille et mes amis. En particulier, je remercie mes parents Hortencia y Rodrigo, qui m’ont toujours soutenu et encouragé dans mes décisions. Parmi mes amis, une pensée spéciale pour Nicole, Christiane, Amelita et Guillermo pour leur soutien et leur aide à la découverte de la culture française. D’autre part, mes amis latinos qui m’ont accompagné et partagé avec moi de très bons moments tout au long de ce chemin. Merci à, Carlos Jaime, Carlitos, Lupita, Luis, Walter, Marcia, Jennifer, Paola, Noé et tant d’autres personnes. Merci à vous tous pour votre amitié.

Je souhaite également remercier Dianita qui a accepté et même soutenu ma décision d’entreprendre une thèse, malgré les conséquences pour tous les deux. Enfin, je remercie Lulita qui est apparu comme un petit ange afin d’illuminer ma vie.

Table des Matières

Table des Figures

1. INTRODUCTION

1.1 CONTEXTE

Il est devenu banal de faire remarquer que la taille des logiciels est en croissance continue, que les besoins sont de plus en plus nombreux et complexes, et que les domaines d'application sont de plus en plus variés. La construction de logiciel est devenue une tâche d'une grande complexité, elle doit faire face à des défis comme la diversité des métiers et des technologies, ainsi que la nécessité d'évolution continue, pour adapter les applications aux besoins et aux plates-formes technologiques changeantes.

Les différentes approches du génie logiciel tentent de faire face à ces défis, elles s'attaquent au problème de la complexité, tout en essayant de garder la flexibilité permettant de faire évoluer les logiciels. Parmi ces approches, l'ingénierie dirigée par les modèles (IDM) considère l'intégralité du cycle de vie logiciel comme un processus de spécification, de raffinement et d'intégration de modèles. D'autres approches conçoivent le développement comme une activité d'assemblage de briques logicielles disponibles. Dans ce mouvement, nous trouvons des propositions intéressantes comme la technologie de *workflow* et l'approche à services.

La technologie de *workflow* a fait son apparition dans les années 90, son objectif était de fournir un outil (un système de *workflow* (WFMS)) capable de coordonner la réalisation d'un procédé, c'est-à-dire de coordonner la série d'interactions entre des personnes et des systèmes logiciels pour atteindre un but donné. Cette technologie rencontre diverses difficultés : couplage fort entre le système de *workflow* et les applications, difficultés d'intégration de technologies distribuées, hétérogènes et en évolution rapide, implémentations monolithiques, coûteuses, difficiles à configurer, adapter et étendre etc. Bien que l'idée soit intuitive, cette technologie n'a pas connue le succès escompté.

L'approche à services se concentre aussi sur le développement par assemblage de briques logicielles appelées services. L'approche à services fournit deux propriétés importantes : un faible couplage entre le client et le fournisseur d'un service, et la liaison retardée. Le couplage est réduit car la description d'un service est indépendante de son implémentation. La liaison est retardée car l'approche propose un cadre d'interaction permettant au client de découvrir et sélectionner à l'exécution le service à consommer, en recherchant dans un annuaire qui contient la description des services disponibles.

La composition de services est le mécanisme utilisé par l'approche à services pour construire des applications (ou de services plus complexes) en intégrant les fonctionnalités d'autres services. L'orchestration de services propose de composer les services en utilisant un modèle de procédé pour exprimer leurs interactions. Un composant logiciel, l'orchestrateur, est chargé de maintenir l'état de la composition et d'invoquer les services à composer dans l'ordre indiqué par le modèle. Les similitudes avec la technologie de *workflow* sont évidentes, même si le composant humain dans l'orchestration de services a disparu.

Un modèle de *workflow* est une abstraction qui exprime de façon externe et explicite la logique de coordination entre différentes unités. Le modèle peut être facilement modifié pour répondre à de nouveaux besoins ou pour améliorer le déroulement de l'application. En plus, la séparation entre l'application et les unités composées permet de faire évoluer ces unités sans avoir à se soucier de l'impact sur l'application.

Les travaux de cette thèse se placent à l'intersection entre la technologie des *workflow*, l'approche à services et l'ingénierie dirigée par les modèles, afin de fournir le support pour la création et l'exécution d'applications orientées procédé. De la technologie de *workflow* nous reprenons l'idée d'utiliser le modèle de procédé comme élément structurant. A l'approche à services nous empruntons la souplesse fournie par les propriétés de faible couplage et de la liaison retardée. Finalement, l'ingénierie dirigée par le modèles nous fournit un cadre pour la mise en place effective de notre approche, en utilisant les modèles comme artefacts de première ordre dans la construction des applications orientées procédé.

1.2 PROBLEMATIQUE

Nous avons étudié différents systèmes de gestion de *workflow* ainsi que diverses approches pour l'orchestration de services, et nous avons identifié plusieurs défauts de ces systèmes et/ou approches que nous résumons par les points suivants :

- Les formalismes utilisés pour représenter le contrôle de flux et de données entre les éléments coordonnés sont de bas niveau d'abstraction ;
- Certains de ces formalismes sont très spécialisés, ce qui restreint leur domaine d'application. D'autres, offrent un nombre élevé de concepts, ce qui rend difficile leur compréhension et manipulation ;
- Les procédés spécifiés dans la plupart de ces approches sont fortement couplés aux éléments qu'ils coordonnent. La modification de ces liens est une tâche difficile et risque d'affecter la cohérence de l'application ;
- Les systèmes sont construits pour coordonner des éléments appartenant à une technologie spécifique ; par exemple l'orchestration de services Web uniquement;
- Généralement, ces systèmes ne prennent pas en compte les préoccupations non-fonctionnelles. D'autres, ajoutent le support pour un nombre fixe de besoins non-fonctionnels.
- La capacité d'extension de ces systèmes est limitée. Certaines approches ne permettent aucune extension et laissent cette responsabilité aux développeurs. D'autres, ajoutent une extensibilité au niveau de formalismes, mais les outils généralement ne comprennent la sémantique de ces extensions.

Finalement, nous remarquons que même si différents types d'applications partagent la caractéristique d'être centrées sur le concept de procédé, la plupart des systèmes et/ou approches visent seulement un domaine d'application spécifique. Certains travaux ont constaté cette caractéristique et essayent de proposer des solutions plus génériques pouvant être utilisées dans divers domaines. Cependant, ces travaux sont souvent accompagnés de contraintes limitant leur adoption.

1.3 NOTRE PROPOSITION

Cette thèse propose une nouvelle approche pour la création d'applications orientées procédés. Nous partons de la conviction que notre système, doit être mis en œuvre autour d'un système basique et extensible. Cette thèse présente FOCAS, un canevas extensible pour la construction d'applications orientées procédé. Le canevas propose entre autres :

- Un noyau basé sur les concepts de l'orchestration de services. Dans ce noyau, les préoccupations basiques d'une application orientée procédé sont incluses, dont celles de contrôle, des données et des éléments à coordonner (services).
- Des mécanismes d'extension fonctionnel permettant aux canevas d'être utilisé dans des domaines différents que l'orchestration de services.
- Des mécanismes d'extension non-fonctionnelle permettant au canevas d'ajouter aux applications le support pour des aspects non-fonctionnels.
- Une architecture permettant de créer différents configurations du système, cette caractéristique permet sont utilisation dans différents environnements.
- Une approche IDM pour le développement des applications orientées procédé. Cette approche permet de structurer une application en différent points de vue autour de l'abstraction de procédé.

Le travail de cette thèse est intégralement implémenté. Il a été validé par l'utilisation du canevas FOCAS dans deux projets européens ITEA ainsi que par son utilisation lors de la création d'extensions dans de travaux de recherche suivis dans l'équipe Adèle.

1.4 PLAN DU DOCUMENT

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

La première partie, constituée de quatre chapitres, fait l'étude de l'état de l'art des systèmes de *workflow*, de l'approche à services et de l'orchestration de services. L'ingénierie dirigée par les modèles servant de base à notre approche est aussi présentée.

- Le chapitre 2 présente la technologie de *workflow*, les concepts et approches qui la supportent.
- Le chapitre 3 présente l'approche à services, ses concepts et le cadre d'interaction proposé par cette approche. Des plates-formes implantant cette approche seront aussi détaillées.
- Le chapitre 4 présente l'approche d'orchestration de services, ses caractéristiques et les différentes propositions au niveau industriel et académique existantes.
- Le chapitre 5 présente l'approche d'ingénierie dirigée par les modèles. La vision de l'approche est présentée, nous nous intéressons particulièrement aux techniques de méta-modélisation et de composition de modèles (et méta-modèles).

La seconde partie du document concerne la contribution de cette thèse. Elle est composée de deux chapitres :

- Le chapitre 6 fait la caractérisation des types d'application visés par notre canevas. Ensuite, elle présente l'approche suivie par le canevas FOCAS.
- Le chapitre 7 présente la mise en œuvre du canevas. Elle montre comment nous avons implémenté un environnement pour la spécification des applications orientées procédés, ainsi que un *runtime* assurant l'exécution des applications, et finalement, comment les mécanismes d'extension ont été implémentés dans ces deux composants.

La troisième et dernière partie concerne la validation, l'évaluation, et montre les résultats obtenus par la mise en œuvre de la proposition. Elle est divisée en trois chapitres :

- Le chapitre 8 décrit l'implémentation de l'approche d'orchestration repartie. Cette mise en œuvre est un résultat important des travaux de cette thèse. Cette extension sert également de validation et d'expérimentation des mécanismes d'extension du canevas.

- Le chapitre 9 décrit le contexte dans lequel les travaux de la thèse ont été effectués. Ce chapitre valide l'approche et les outils implémentés.

Finalement, le chapitre 10 conclut le document et propose les perspectives de travail.

2. LES *WORKFLOWS*

Les procédés ont été utilisés dans l'informatique de diverses manières. Une partie de la communauté utilise les procédés comme une définition soit ***informelle*** du processus de construction de logiciel (méthodes, conventions, savoir-faire en langage naturel), soit ***semi-formelle*** comme RUP (*Rational Unified Process*), soit encore ***formelle*** ce qui rend les modèles exécutables et permet l'automatisation partielle ou totale des procédés. Ceci est connu comme gestion des *workflows* ou gestion des procédés métiers. Dans le cadre de cette thèse nous nous plaçons dans ce dernier cas où les procédés sont utilisés avec l'idée de leur automatisation totale ou partielle.

Dans ce chapitre, nous décrivons dans un premier temps les concepts de base des *workflows* et des systèmes de gestion de *workflows*. Ensuite nous présentons des approches, langages et outils utilisés dans le monde académique et l'industrie avec le but de construire des systèmes de *workflows*. Puis nous identifions les caractéristiques des applications pilotées par les procédés et les différents besoins dans le cadre de leur construction. Finalement nous présentons une synthèse des principales idées de cette technologie.

2.1 LES CONCEPTS DE BASE

Le mot procédé est un terme utilisé informellement et il en existe plusieurs définitions. Dans [IEE90], un procédé est défini comme :

« A sequence of steps performed for a given purpose »

Dans cette définition, le procédé est décrit comme une séquence d'étapes pour arriver à un but spécifié, le mot séquence dénote une notion d'ordre entre les différentes étapes.

D'après [Ost87], la notion de procédé est définie comment :

« Our elementary notion of a process that it is a systematic approach to the creation of a product or the accomplishment of some task »

Cette définition met l'accent sur le fait qu'un procédé est une approche systématique, ceci implique qu'il existe une séparation entre l'exécution réelle du procédé et son modèle abstrait qui indique comment le procédé doit être exécuté. Elle précise également que le résultat de l'exécution d'un procédé peut être un produit.

Dans [MMWF93], une distinction est faite entre les procédés matériels (*material process*) et les procédés d'information (*information process*). Le but du premier est l'assemblage de composants pour construire un objet physique, pendant que le deuxième est basé sur des tâches automatisées (exécutées par des ordinateurs) et/ou partiellement automatisées (réalisées par des êtres humains en interagissant avec des ordinateurs) pour créer, traiter, gérer et fournir de l'information.

[GHS95] introduit le concept de procédé métier en disant qu'un procédé métier est une description orientée marché des activités d'une organisation qui peut être implémenté en tant que procédé matériel ou procédé d'information.

Une autre définition de procédé métier est faite par [Wes07] :

« A business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations »

Dans cette définition, le concept d'environnement organisationnel et technologique est ajouté à la réalisation du procédé, par conséquent un procédé métier est réalisé en utilisant des ressources, qui sont des êtres humains et/ou des machines. En plus, la définition dénote une approche orientée vers un objectif du métier qui est accompli par une organisation unique mais les procédés métiers de différentes organisations peuvent interagir entre eux.

2.2 SYSTEMES DE GESTION DES WORKFLOWS

La gestion des procédés métiers identifie les tâches qui permettent leur réalisation effective. Pour [Wes07], la gestion des procédés métiers est définie comme :

« Business process management includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes »

Dans cette définition, la gestion de procédés métiers se présente comme un processus d'ingénierie qui vise à la conception, l'administration, la configuration, la réalisation et l'analyse d'un procédé métier. Dans cette définition, nous trouvons l'idée de conception ou modélisation d'un procédé, donc la construction de modèles pour représenter des procédés métiers. Dans le chapitre dédié à l'IDM la tâche de modélisation (et celle de méta-modélisation) sera explorée plus en profondeur.

Dans [vdA04], l'auteur définit un cycle de vie pour la gestion des procédés métiers qui est constitué de quatre phases :

- conception des procédés : dans cette phase les modèles de procédés sont créés basés sur différentes perspectives (flux de contrôle, flux de données, organisationnelle, etc.).
- configuration du système : dans cette phase les différentes ressources (humaines et logicielles) sont mises en place et configurées pour être utilisées dans l'exécution du procédé.
- exécution du procédé : le procédé est réalisé en utilisant les modèles créés dans la première phase ainsi que la configuration réalisée dans la seconde phase.
- diagnostic : dans cette phase sont effectués des changements pour améliorer l'exécution, supporter de nouveaux procédés, et s'adapter aux changements de l'environnent. Un lien entre les phases d'exécution et de conception est effectué pour fermer le cycle.

Dans la littérature, nous trouvons des définitions qui se recouvrent pour décrire ce processus de gestion de procédés métiers, certains auteurs parlent de BPM (*Business Process Management*) et d'autres de WFM (*Workflow Management*).

Une différence est faite par [vdA04], qui considère le WFM comme un sous ensemble de BPM. Le WMF se concentre sur la partie basse du cycle de vie du BPM, plutôt sur la configuration du système et dans son exécution, ceci est une vision orientée technologie. D'autre part, le BPM est une vision plus orientée métier qui met l'accent sur le diagnostic, la flexibilité ainsi qu'une conception qui vise aux buts du métier, centrée sur les gens plutôt que sur l'infrastructure technologique.

Nous considérons que le WFM sert de base à la réalisation effective des BPM car des outils et des méthodes matures peuvent être trouvées dans cette approche.

Le système qui sert à effectuer la gestion des procédés est appelé système de gestion de *workflow* (ou système de gestion de procédés métiers). Un WFMS est un ensemble d'outils permettant la création du modèle de *workflow*, son exécution, son administration et son *monitoring* [WFM94].

Dans [Wes07], un système de gestion de procédés métiers est définit comme :

« A business process management system is a generic software system that is driven by explicit process representations to coordinate the enactment of business processes »

Un système de gestion de *workflow* utilise alors une représentation explicite du procédé (un modèle) pour coordonner la réalisation du procédé métier.

Dans la section suivante, nous présentons différents systèmes de gestion de *workflow* existants. D'une part seront présentés les langages utilisés afin de modéliser les procédés, ainsi que les différents systèmes créés pour les exécuter et pour réaliser le suivi de son exécution.

2.3 LANGAGES DE DEFINITION DES WORKFLOWS

Des nombreux systèmes de gestion de *workflows* ont été développés et utilisés dans les dernières années, la majorité d'entre eux sont des produits commerciaux (*COSA, VisualWorkflow, Forté Conductor, Lotus Domino Workflow, Mobile, MQSeries/Workflow, Staffware, Verve Workflow, I-Flow, InConcert, Changengine, SAP R/3Workflow, Eastman, FLOWer, etc*), autres, des projets *open source* (*JBoss JBPM, Bull Bonita, Active BPEL, Enhydra Shark-JaWE, etc*) et des prototypes du monde académique (*Meteor, ADEPTFlex, OpenFlow, YAWL, APEL, JOpera, etc*). La plupart de ces systèmes utilisent leurs propres formalismes (langages) pour exprimer les modèles de *workflow*.

Il existe aussi des organisations essayant de définir des standards au niveau des langages et des architectures de systèmes de *workflow*. Parmi elles, nous trouvons, la *WfMC* qui a établi un cadre de référence cherchant à assurer l'interopérabilité des systèmes de *workflow*. La *WfMC* a spécifié de cette façon son langage de modélisation de procédés *XPDL*. OASIS est chargé de la standardisation du langage d'orchestration de services web BPEL4WS, pour sa part l'OMG normalise BPMN qui est un formalisme graphique pour l'expression de procédés métiers, l'OMG est chargé aussi de définir la sémantique d'UML *activity diagrams*.

Dans cette section, nous allons présenter certains des formalismes utilisés pour l'expression des modèles de procédés. Nous allons nous concentrer dans les concepts proposés pour chacun des langages, leurs problèmes, leur évolution et leurs caractéristiques principales. Les formalismes étudiés ont été choisis soit par leur impact niveau industriel, soit parce qu'ils sont des standards industriels ou bien pour leur apport à la recherche.

L'évolution historique des différents langages de modélisation de procédés utilisés dans la technologie de *workflow* est présentée dans la Figure 1, ce diagramme est repris de [EAS08] et complété.

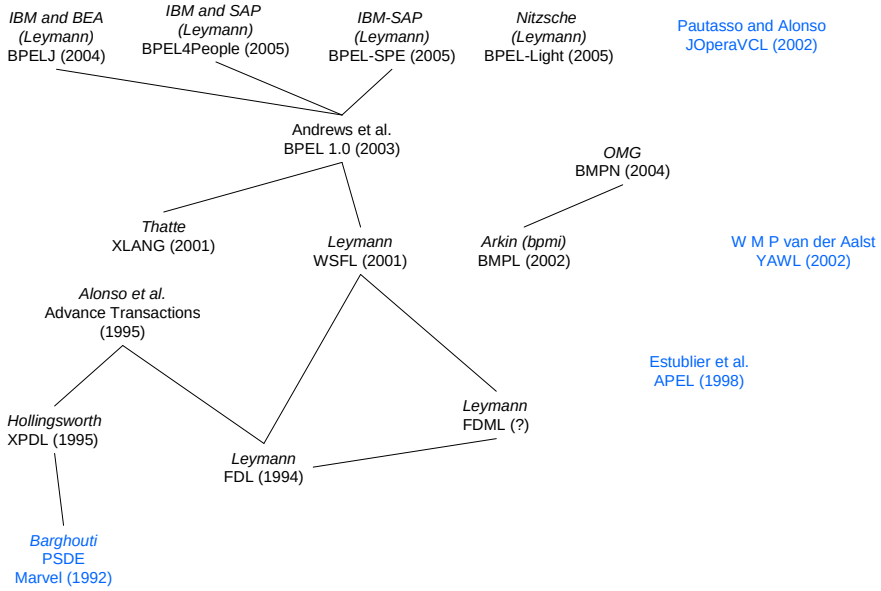

Figure 1. Evolution des langages de modélisation des *workflows.*

Les langages les plus anciens se trouvent dans la partie base de la figure et les plus récents dans la partie haute. Les traits représentent des influences entre les langages, selon [EAS08] ces influences sont de différents types comme : citation de papiers de recherche, déplacement des personnes d'une entreprise ou institut de recherche vers un autre, réutilisation des concepts, ou fusion d'anciennes propositions. En plus, les langages en noir sont ceux issus de la recherche industrielle, et en bleu ceux de la recherche académique.

C'est l'activité de recherche dans le domaine des procédés logiciels, qui a permis d'introduire des langages pour la description explicite des procédés, ces langages sont connus dans cette communauté comme PMLs (*Process Modeling Languages*). Plusieurs de ces langages ont été utilisés dans des contextes divers et non seulement dans les procédés logiciels, et ils ont fortement influencé la création de nouveaux langages. Ceci est le cas du langage PSDE (1992) du système *Marvel* [Bar92], ses principales notions ont été reprises et utilisées dans la définition du modèle de référence de *workflow* par la WfMC (acronyme en anglais de *Workflow Management Coalition*) et en conséquence pour son langage de définition de procédés XPDL (acronyme en anglais de *XML Process Definition Language*) [WFM02].

Nous pouvons classer les langages de définition des procédés en trois catégories. D'abord, les langages qui essaient de couvrir plusieurs domaines d'application, ce type de langages ont un haut niveau d'expression, par contre ce sont des langages complexes et d'utilisation difficile. Ensuite, les langages spécialisés pour la représentation des procédés dans un domaine métier ou une technologie spécifique, c'est le cas de langages comme *WSFL*, *XLANG* et *BPEL4WS* qui répondent aux besoins de l'orchestration de services web. Ces langages ont des difficultés lorsqu'ils essaient d'adresser des besoins hors de leur domaine d'origine. Finalement, certains langages ont été conçus dans le but de pouvoir exprimer des procédés dans un domaine quelconque sans faire référence à une technologie spécifique, c'est le cas de langages comme YAWL et APEL. L'idée de cette dernière catégorie de langages est l'extension du langage pour l'utiliser dans divers domaines, ce qui est une tâche non triviale.

2.3.1 XPDL

La *WfMC* est une organisation qui promeut et développe l'utilisation de la technologie de *workflow* grâce à l'établissement de normes pour établir une terminologie commune, améliorer l'interopérabilité et la connectivité entre les systèmes de gestion de *workflow*. La *WfMC* a spécifié un modèle de référence qui doit être implémenté pour les systèmes de gestion de *workflows*, ce modèle est composé de cinq interfaces bien définies. Ces interfaces établissent

des liens entre le moteur d'exécution et les autres composants du système, comme des éditeurs (Interface 1), des applications clients notamment la liste de tâches (Interface 2), les applications à invoquer par le moteur (Interface 3), d'autres moteurs de *workflow* (Interface 4) et des outils de monitoring (Interface 5). La Figure 2, prise du site de la *WfMC* schématise le modèle de référence.

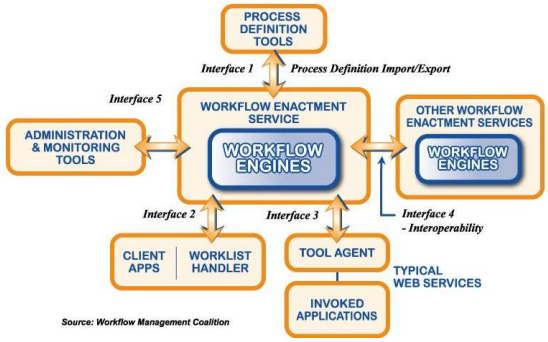

Figure 2. Modèle de référence d'un système de gestion de *workflow* défini par la *WfMC.*

La première interface est établie à l'aide d'un langage de définition de procédés, les outils de spécification produisent des modèles en utilisant ce langage et le moteur a la capacité de les exécuter. Le langage de définition de procédés spécifié par la *WfMC* est XPDL, il a été conçu avec l'objectif de servir de format d'échange entre les différents systèmes de gestion de *workflow*.

Les principaux concepts du langage XPDL sont : package, application, processus, activité, transition, participant, donnée (*datafield)* et type de donnée (*datatype)*. L'élément package sert comme conteneur des éléments qui peuvent être partagés par plusieurs définitions de *workflows*, par exemple la définition des participants ou des applications.

L'élément processus sert à définir un *workflow* ou une partie de *workflow*. Un *workflow* contient des activités et des transitions. La notion d'activité est le concept de base dans XPDL et les transitions sont utilisées pour lier des activités. Il existe trois types d'activités, les activités de routage (*Route*) qui sont utilisées pour exprimer les contrôles de flux complexes entre des activités qui ne peuvent pas être exprimés avec des transitions seulement. Ensuite, les activités de blocs (*BlockActivity*) qui servent à exécuter un ensemble d'activités. Finalement, les activités de type implémentation (*Implementation*) qui servent à exprimer des tâches du *workflow*, ces tâches peuvent être implémentées en utilisant des procédures manuelles (réalisées par des êtres humains), des outils (réalisées par une application) ou de type *workflow* (réalisées par un autre système de *workflow*).

L'élément participant décrit les entités qui vont réaliser les tâches des activités. Il existe six types différents de participants : ensemble de ressources (*resourceSet)*, ressource, rôle, unité organisationnelle, humain et système. Finalement, les éléments *datafield* et *datatype* sont utilisés pour exprimer les données utilisées à l'exécution du *workflow*.

XPDL inclut un mécanisme d'extension qui permet d'ajouter des attributs aux éléments qui font partie de la définition d'un *workflow*. Généralement ces attributs sont utilisés pour inclure, par exemple, des informations de positionnement graphique des éléments. Le mécanisme d'extension est assez permissif, et donc la sémantique associée aux extensions est seulement comprise par les outils qui ont défini l'extension.

Conclusion

Etant donné que XPDL a été créé dans le but de permettre l'échange de définitions de procédés entre différents systèmes de gestion de *workflow*, il contient un nombre élevé de concepts pour essayer de couvrir le plus grand nombre de systèmes de *workflow*. Cette caractéristique fait que le langage est riche mais difficile à utiliser dû à sa complexité. Le

mécanisme d'extension inclut dans le langage permet seulement de l'étendre au niveau syntactique, par conséquent la sémantique des extensions est uniquement comprise par les outils utilisés dans le contexte de l'extension.

2.3.2 JBPM

JBoss a développé le canevas JBPM dédié à l'exécution des applications pilotées par des procédés dans un environnement Java. Ce canevas fourni une machine virtuelle d'exécution de procédés, sur laquelle différents langages peuvent être exécutés. Dans la vision de JBPM, l'utilisation d'un langage spécifique pour la définition de procédés est plus pertinente pour un environnement ou type d'application spécifique que pour l'utilisation d'un langage générique. Cependant, ces langages de procédés partagent un ensemble de concepts communs, et pourtant une technique commune, celle dénommée par JBPM comme la programmation orientée graphes (POG).

Par rapport aux langages impératifs tels que Java, la POG introduit la notion centrale d' « état d'attente ». Un état d'attente est défini comme un pas dans l'exécution d'un flux qui attend un événement pour continuer son exécution. La POG est alors une technique permettant de définir est d'exécuter un graphe dans un langage orienté objet, dans le cas de JBPM le langage Java.

Pour la modélisation des graphes, deux concepts sont utilisés, celui de nœud et celui de transition. Les nœuds représentent les états d'attente et peuvent avoir un comportement associé, et les transitions sont utilisées pour lier les nœuds. A l'exécution un *token* indique quel nœud (ou ensemble de nœuds) est en exécution (état d'attente) à un moment donné.

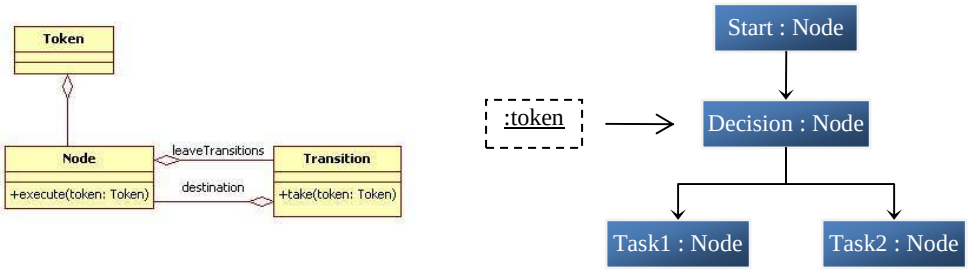

Figure 3. Classes implémentées par JBPM. Un graphe en exécution.

Dans la Figure 3, à gauche, sont présentées les classes implémentées par le canevas pour les concepts de nœud et transition. La classe *Node* contient la méthode *execute* qui sert à spécifier son comportement, la classe *Transition* contient la méthode *take* servant à faire passer le *token* à un nœud suivant du graphe. A droite, est montré un graphe en cours d'exécution, le *token* indique un nœud attendant un événement pour continuer l'exécution du graphe, dans ce cas le nœud *Décision.*

L'implémentation d'un langage de procédé dans le canevas est faite par l'extension de la classe *Node*, chaque nouvelle classe implémente la méthode *execute* pour définir le comportement spécifique du nouveau type de nœud créé dans le langage. Par défaut le produit JBPM inclut le langage JPDL pour la définition de workflows. Le langage JPDL spécialise le concept de nœud pour définir explicitement le concept de nœud de départ (*start*) et de fin (*end*), le concept de nœud de tâche (*task*) étant une unité de travail (à être effectué par des humains ou par de machines) et des nœuds de routage du contrôle (*Decision*, *Fork* et *Join).* La Figure 4 présente l'extension définie au canevas afin de produire le langage jPDL.

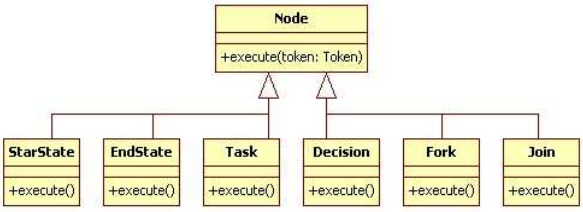

Figure 4. Extension du canevas JBPM pour spécifier le langage JDPL.

Le support fourni par JPDL pour le traitement des données est assez rudimentaire, à la conception des variables peuvent être définies, ces variables ont une portée globale, c'est-à-dire elles sont accédées pour tous les nœuds et elles ne sont pas typées. Le type d'une variable (des types simples Java) est décidé au moment de son affectation et peut être modifie par chaque nœud pendant l'exécution. Le canevas utilisé une structure des données de *tuples* (clé, value), donc, même de données sans une variable associée peuvent être ajoutées à cette structure.

Conclusion

Le canevas JBPM propose la technique de programmation orientée graphes comme base pour la construction de nouveaux langages de spécification de procédés et leur moteur d'exécution. JBPM fournit aussi un langage par défaut pour la définition de procédés, le langage jBPM. Malgré la simplicité offerte par le mécanisme d'extension du canevas, il vise seulement la création de nouveaux types de nœuds. Des autres aspects importants, comme la gestion de données, est la responsabilité du concepteur du nouveau langage, il doit alors spécifier la sémantique de définition et traitement de données et l'adapter au mécanisme simpliste fourni par JBPM.

De même, ni la perspective des ressources, ni celle de l'environnement opérationnel ne sont incluses dans la spécification du canevas. Nous considérons que l'idée d'extension du canevas pour la création de nouveaux langages de définition de procédés dédiés est pertinente, par contre les mécanismes offerts par JBPM sont d'assez bas niveau d'abstraction permettant principalement de réutiliser la bibliothèque (*Process Virtual Machine*) qui offre la notion de POG.

2.3.3 YAWL

Dans [vdAtHKB03], les auteurs proposent un ensemble de patrons ayant pour objectif d'évaluer le niveau d'expressivité des langages de définition de *workflow* depuis la perspective du flux de contrôle. Ce canevas d'évaluation a pour objetif d'identifier l'effort nécessaire pour exprimer dans un formalisme de définition de procédés chacun des patrons, donc la pertinence du langage au moment d'exprimer les patrons. Plusieurs formalismes ont été analysés et comparés en utilisant les patrons de *workflow*, le résultat de ces comparaisons peut être consulté dans [WPDH02] [vdAtHKB03].

A l'issu de cette étude, le formalisme YAWL (*Yet Another Workflow Language*) a été proposé [vdAH05]. YAWL est un langage de définition de procédés basé sur les réseaux de Petri. Les réseaux de Petri [WE98] sont des formalismes mathématiques utilisés dans la description de systèmes avec caractéristiques de concurrence et de partage de ressources. L'avantage de ce formalisme ce qu'il existe un nombre importante de techniques pour vérifier les propriétés structurelles des systèmes modélisés, comme par exemple, l'identification des interblocage (*deadlocks)*. Malgré son niveau d'expressivité élevé, un réseau de Petri n'est pas un formalisme intuitif et son niveau d'abstraction est assez bas (car il a été conçu pour les mathématiciens).

YAWL est un formalisme ayant des primitives d'un niveau d'abstraction plus élevé que les réseaux de Petri. Dans YAWL, un *workflow* est spécifié en utilisant un EWF-Net (réseau étendu de *workflow*) en tant qu'élément conteneur. Un EWF-Net est composé de tâches, de

conditions, et de connections. Les tâches peuvent être atomiques ou composites, une tâche composite est spécifiée par le biais d'un EWF-Net, donc le langage est structuré. YAWL possède des tâches spécialisées : tâches pour exprimer le contrôle de flux (*and-split, and-join, or-split, or-join, xor-split, xor-join*) ; des tâches pour exprimer des activités effectuées par des humains et des tâches qui interagissent avec des applications. Les conditions sont des points dans le *workflow* où des décisions affectant le flux de contrôle sont prises. Finalement, les connecteurs sont utilisés pour exprimer des liaisons entre une tâche et une condition ou vice-versa. Un connecteur peut être utilisé pour connecter deux tâches, ce cas se présente pour les taches ayant une condition implicite. Cette caractéristique dénote l'origine du langage (réseaux de Petri). Le diagramme de classes présenté dans la Figure 5 montre les principaux concepts de YAWL et leurs relations.

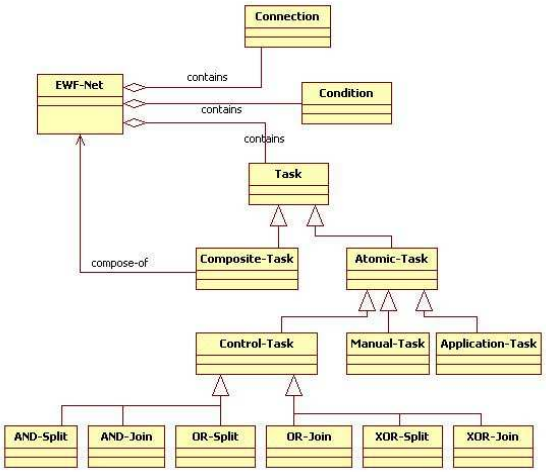

Figure 5. Concepts de la perspective de contrôle de flux du langage YAWL.

La perspective des données est aussi adressée dans YAWL ; des variables peuvent être déclarées et le système est fortement typé. Les types sont spécifiés dans la définition d'un *workflow* et le langage utilisé est XML-Schema. Les variables peuvent avoir une portée globale (pour l'instance du *workflow*) ou locale (pour une tâche spécifique).

Un modèle organisationnel ne peut pas être défini dans le langage, par contre il est possible de faire une spécification de ce modèle au moment de l'exécution dans l'implémentation fournie par défaut par YAWL. Ce modèle, peut être alors partagé par des définitions différentes de *workflows*. La méthode utilisée par le moteur pour affecter les ressources aux tâches est riche mais aussi complexe.

Conclusion

Le langage YWAL fournit un formalisme très riche avec un niveau d'expressivité haut, pouvant être utilisé dans différents types d'applications qui utilisent la notion de procédé. Par contre, les capacités d'extension du langage sont assez limitées car aucun mécanisme d'extension n'est fourni (au moins explicitement dans la documentation). Nous considérons qu'utiliser un formalisme unique pour couvrir plusieurs domaines d'application est une tâche compliqué, donc l'extensibilité est une propriété souhaitable pour tout langage ou canevas.

2.3.4 APEL

APEL est un langage issu de la recherche sur les procédés logiciels développé dans notre équipe de recherche. APEL a été conçu comme un langage minimal qui peut être étendu et utilisé dans divers contextes où l'expression d'un procédé est nécessaire. Les concepts du langage APEL sont présentés dans le diagramme de classes de la Figure 6 :

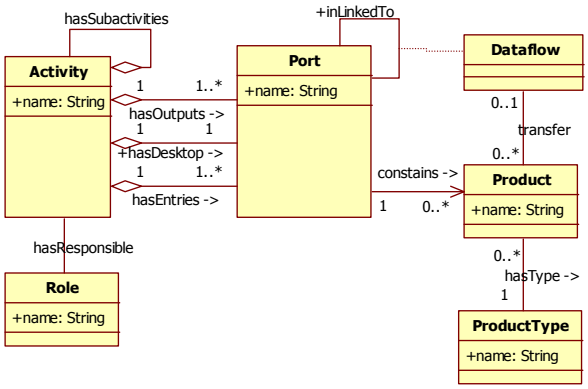

Figure 6. Concepts du langage APEL.

APEL est un langage basé sur le concept d'activité, les concepts basiques sont :

- Activité : une activité est une tâche qui peut être réalisée pendant l'exécution du procédé. Une activité peut être atomique ou composite, une activité composite peut contenir des sous activités imbriquées.
- Port : un port sert d'interface de communication pour une activité.
- Produit : un produit est une entité produite, transformée et/ou consommée par les activités.
- Type de produit : les produits sont typés dans APEL. Le type est seulement une chaîne de caractères, la structure interne du produit n'est pas décrite par son type dans le langage.
- Flux de données : exprime le flux de données et de façon implicite le flux de contrôle entre les activités.

Conclusion

Le langage APEL est un langage contenant un ensemble minimal de concepts de haut niveau d'abstraction, ce qui facilite son utilisation. Par conséquent, APEL peut être utilisé pour exprimer des situations où l'abstraction de procédé est nécessaire.

La structure des types de données n'est pas spécifiée dans APEL, cette responsabilité est déléguée à un autre langage, de même que la définition d'un modèle organisationnel et un modèle de l'environnement opérationnel. En plus, rien n'est dit par rapport à la réalisation des activités, il peut s'agir de tâches manuelles ou automatisées.

L'idée est de faire d'APEL un langage qui apporte un minimum de concepts et qui peut être étendu afin d'adresser diverses problématiques. Par contre, le langage lui-même ne fournit pas les mécanismes d'extension.

2.4 APPLICATIONS PILOTEES PAR LES PROCEDES

Pour [vdA04], les systèmes d'information sont organisés en couches comme le montre la Figure 7. La couche centrale ou noyau est composée par les systèmes d'exploitation qui se chargent de la gestion des ressources matérielles. Dans la deuxième couche nous trouvons des applications dites génériques, comme les feuilles de calculs ou les systèmes de gestion de bases de données, qui peuvent être utilisés dans différentes entreprises ou départements. Dans la troisième couche, les applications appartiennent à un domaine spécifique. Là, nous pouvons distinguer les applications orientées vers un type particulier de métier, utilisées dans une classe d'entreprise ou un département. Finalement, dans la quatrième et dernière couche, les applications faites sur mesure, qui sont créées pour une organisation en particulier. Chacune de

ces couches peut se servir des couches précédentes pour réaliser leurs fonctions, par exemple une application domaine spécifique (troisième couche) peut utiliser un système de gestion de bases de données qui se trouve dans la couche des applications génériques (deuxième couche).

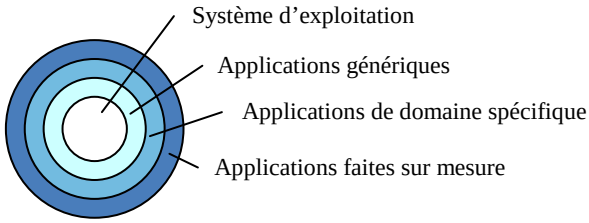

Figure 7. Organisation en couches des systèmes d'information [Aalst2004].

Actuellement trois tendances sont identifiées dans le développement des applications. Tout d'abord les couches essaient de s'élargir vers l'extérieur, par exemple, les systèmes d'exploitation ajoutent à chaque fois plus de fonctionnalité de même que les systèmes de bases de données. Deuxièmement, le changement d'une orientation basée sur les données vers une orientation basée sur les procédés. Dans les années 80 et 90, les systèmes ont été conçus et développés autour des modèles des données ; actuellement l'accent est mis sur le modèle de procédé. Dans [LR97], les auteurs analysent cette tendance comme étant d'abord l'élimination des applications de leur dépendance vers la manipulation de données, cette tâche est désormais déléguée aux DBMS. De la même façon, le contrôle des applications tends à être externalisé, ce qui réduit leur dépendance vis-à-vis du flux de contrôle et données.

En troisième lieu, comme les systèmes changent très rapidement surtout à cause des possibilités offertes par la technologie, la tendance est de créer des systèmes basés sur des éléments existants, donc une vision de construction des applications par assemblage. Cette vision favorisant la réutilisation, s'impose sur une vision de construction à partir de zéro.

Si nous considérons la deuxième tendance, (développement des applications en utilisant une externalisation du contrôle) et la troisième tendance (construction d'applications à partir d'éléments préexistants), nous pouvons alors introduire la notion d'application pilotée par un procédé. Nous considérons une application pilotée par un procédé, comme une application qui utilise un composant qui se charge explicitement du flux de contrôle et de données entre les différents éléments qui composent l'application. Les unités de composition sont coordonnées ou orchestrées en utilisant ce composant de contrôle. La nature des éléments coordonnés est variée, il peut s'agir d'applications logicielles, de composants, de services, de tâches réalisées par des humains etc.

Il existe un domaine d'application vaste pour les applications pilotées procédés, nous pouvons citer :

- Des applications pour la gestion de documents (une des premières utilisations de la technologie de workflow).
- Des applications pour supporter le travail collaboratif.
- La description des flux de navigation des différents pages et ressources d'une application web (*web flow*).
- L'orchestration des services.
- Des systèmes de gestion de ressources (ERP) comme SAP.
- Des applications faites pour une grille de calcul.
- Des applications pour supporter le processus de développement de logiciel

Nous remarquons qu'il existe une large utilisation des applications pilotées par les procédés mais, étant donnée la nature diverse des domaines adressés ainsi que les différents besoins dans chaque type d'application, nous considérons qu'essayer d'utiliser un formalisme

unique pour exprimer des solutions dans ces domaines n'est pas une solution envisageable. La caractéristique de la séparation du contrôle du reste de l'application présentée dans cette famille d'applications fourni un modèle de décomposition commun. A partir de ce modèle de base, nous voyons la possibilité d'ajouter les caractéristiques particulières du domaine d'application pour avoir une application répondant aux besoins spécifiques de ce domaine.

2.5 SYNTHESE

Au début, l'objectif d'utilisation de la technologie de *workflows* était l'automatisation des différents types de procédés métiers. Deux problèmes se présentaient, d'abord la technologie (au niveau logiciel et des réseaux) n'était pas au point, et plus important le composant non-déterministe, proactif et créatif du travail humain ne pouvait pas être ni modélisé, ni contrôlé, ni automatisé avec ce type d'outils [FKN94]. En conséquence, un objectif plus à la portée de la technologie de l'époque a été adressé, les *workflows* étaient alors utilisés pour la gestion des documents et le support du travail collaboratif, l'idée était l'automatisation des tâches administratives afin de diminuer le temps de leur réalisation, dans ce courant, des *workflows* plus simples et déterministes ont été utilisés.

Avec l'introduction des applications reparties, et plus récemment des technologies orientées services, certaines des notions des technologies des *workflows* ont été reprises, mais avec un objectif plus ciblé, celui d'orchestrer l'invocation des différentes entités logicielles pour construire des applications par assemblage de ces unités, cette nouvelle approche est connue comme l'orchestration des services.

Des centaines de systèmes de gestion de *workflows* commerciaux, issues du monde académique, des standards ou d'*open source* ont été proposé ces dernières années. Ils adhèrent essentiellement aux deux courants, le premier qui essaye de proposer des systèmes adressant des besoins spécifiques, soit dans un domaine métier comme par exemple la banque ou les assurances, ou dans un domaine technologique comme l'orchestration de services web. L'autre courant tente de construire des systèmes génériques avec l'objectif d'être utilisés dans un grand nombre de domaines différents.

Les systèmes spécifiques fournissent un cadre précis d'utilisation et une performance satisfaisante, par contre il est difficile de les utiliser dans d'autres domaines et de les étendre. L'adoption de ce type de système par des organisations exige un grand investissement car il faudrait un système différent pour chaque domaine adressé. Les systèmes dits génériques, en revanche, offrent des langages riches, avec un important nombre de concepts essayant de couvrir un large spectre d'applications, cependant ce sont des formalismes complexes et difficiles à utiliser.

Une troisième tendance est apparue ces dernières années visant à construire des systèmes de base qui peuvent être étendus pour supporter la technologie de *workflow* dans plusieurs domaines. Une de ces approches est le canevas JBPM proposant des mécanismes d'extension de très bas niveau de sa technologie de POG pour construire des langages de procédé spécialisés. APEL pour sa part propose un langage minimal, avec l'idée de l'étendre pour son utilisation dans d'autres domaines, mais laissant la responsabilité des extensions aux autres canevas.

Nous considérons que la technologie des *workflows* est prometteuse et envisageons de reprendre ses idées avec l'objectif de construire un système de support pour la construction des applications pilotées par les procédés. Cependant, nous considérons qu'il ne s'agit pas seulement de fournir des langages de définition de procédés riches (et complexes), mais plutôt d'offrir des mécanismes d'extension permettant l'utilisation du système dans divers domaines d'application ainsi que de fournir les outils nécessaires pour supporter cette technologie.

3. L'APPROCHE A SERVICES

Dans le génie logiciel, l'évolution des méthodes et technologies fait émerger de nouvelles approches pour le développement d'applications. Ainsi, ont fait leur apparition, l'approche à objets [Tay98], ensuite l'approche à composants [HC01] et de nos jours, l'approche à services. Toutes ces approches reposant sur les principes de base du génie logiciel : l'abstraction, la séparation de préoccupations et la modularité. Chaque nouvelle approche essaye de s'appuyer sur les anciens paradigmes tout en y ajoutant de nouveaux atouts.

L'idée de l'approche à services est de construire des applications rapidement par l'assemblage de services. Une mise en place effective de cette approche permet l'intégration de systèmes d'information. Généralement, ces systèmes sont hétérogènes et n'ont pas été conçus pour être intégrés dans un environnement commun. Donc, l'approche à services propose un modèle essayant de donner une réponse à cette problématique.

Le chapitre précédent a étudié la technologie des *workflows*, ses tendances et les difficultés pour sa mise en place effective. Nous avons mis en évidence que le développement technologique aux niveaux des réseaux et des systèmes distribués a stimulé la reprise de cette technologie. Grâce à l'introduction de l'approche à services, la technologie de *workflows* est relancée afin de mettre en place des applications construites par assemblage de services. Etant donné que la construction de ce type d'applications joue un rôle central dans cette thèse, nous allons étudier cette approche et ses propriétés.

Dans ce chapitre, nous allons détailler l'approche à services. Une première partie sera consacrée à définir les concepts de base de l'approche. Ensuite, nous expliquerons l'architecture spécifique à l'approche à services, puis, nous décrivons différentes plates-formes implémentant l'approche à services, leurs caractéristiques, leurs contextes d'utilisation et leurs différences seront exposées. Nous finirions par une synthèse qui résume l'approche à services.

3.1 LES CONCEPTS DE BASE

L'approche à services (en anglais SOC, acronyme de *Service Oriented Computing*) propose que la construction d'une application soit effectuée par l'assemblage de briques logicielles préexistantes, testées et validées ce qui fait que la productivité augmente avec la réutilisation de ces briques.

Ce paradigme n'est pas nouveau, déjà dans l'approche à composants l'idée était l'assemblage de blocs logiciels préfabriqués, appelés des composants. Un composant est défini comme une entité logicielle composable, réutilisable, décrivant explicitement ses capacités et ses dépendances. Dans cette approche un composant est l'unité de composition, de déploiement et de versionnement [SGM02]. Des modèles à composants, comme Fractal [BCS04], CCM [Gro01], EJB [Mic06b], définissent la structure des unités de composition (composants) et le mécanisme de composition utilisé pour les assembler.

Certaines de ces technologies, fournissent aussi un support pour des aspects non-fonctionnels tels que la distribution, la sécurité et la gestion de transactions. Ces modèles à

composants sont rigides, c'est-à-dire qu'une fois défini une application, généralement pendant la conception, il est difficile de modifier son architecture à l'exécution. Ce problème est dû principalement au fort couplage provenant de la définition explicite des liens entre les interfaces fournies et requises des composants constituant une application.

Dans l'approche à services une application est réalisée par composition des services logiciels qui sont mis à disposition par des fournisseurs divers [Pap03]. Un des principaux bénéfices de cette approche est le faible couplage entre les différentes entités qui constituent une application. Dans l'approche à services « pure », bien représenté par les services web, les liens exprimés entre les interfaces fournies et requises de l'approche à composants disparaissent. Ainsi un service est indépendant de l'état et du contexte d'exécution des autres services. C'est la responsabilité du fournisseur du service de résoudre ses dépendances et de gérer le cycle de vie du service. De plus, le client n'a pas à connaître les détails d'implémentation du service. Ce découplage augmente le degré d'indépendance et de dynamisme au niveau de l'architecture des applications du au fait qu'une implémentation peut être remplacée par une autre plus facilement, même au moment de l'exécution.

Dans l'approche à services, les entités basiques sont appelées services, mais la notion de service a été définie de diverses manières dans la littérature. Dans [PvdH07], un service est défini comme :

« "services", which are well defined, self-contained modules that provide standard business functionality and are independent of the state or context of other services »

Un service est alors une entité logicielle qui fournit une fonctionnalité métier. Comme propriétés de cette entité, l'auteur dit qu'elle est bien définie, auto contenue et qu'elle est indépendante de l'état des autres services.

Dans [FWK02], l'auteur liste les caractéristiques qu'un service doit respecter, ces caractéristiques sont :

- la modularité : un service est modulaire et réutilisable. Il est possible de construire des applications ou des services plus complexes à partir d'autres services,
- la disponibilité : un service est disponible est utilisable par des clients,
- la description : un service possède une description qui est lisible par des machines. La description sert à spécifier l'interface du service et sa localisation,
- l'indépendance d'implémentation : un service a une claire séparation entre sa description (ou interface) et son implémentation.
- la publication : la description du service est publiée dans un annuaire qui sera consultable par les clients du service.

Cette définition ajoute l'indépendance de la description d'un service par rapport à son implémentation, par le fait que les descriptions de services sont publiées dans un annuaire. L'annuaire peut alors être utilisé par des clients pour chercher et sélectionner les services à consommer.

Suite à ces définitions, nous pouvons donner une définition de service comme suit :

« Un service est une entité logicielle qui fourni une fonctionnalité et ayant une description qui sert à exprimer ses capacités fonctionnelles et non fonctionnelles. La description du service est utilisée par les clients pour rechercher, sélectionner et invoquer le service qui s'adapte le mieux à ses besoins. La description d'un service est indépendante de son implémentation. »

Nous avons ajouté aux définitions précédentes, que la description d'un service est composée de ses caractéristiques fonctionnelles et non-fonctionnelles, et que cette description est utilisée dans une phase de sélection réalisée par les clients avant de consommer un service.

3.2 L'ARCHITECTURE SOA

Pour la mise en place de l'approche à services un schéma définissant ses acteurs et un protocole d'interaction entre eux a été défini. Cette infrastructure est connue comme l'architecture orientée services SOA (*Service Oriented Architecture*). La Figure 8, ci-dessous présente le schéma de l'architecture SOA :

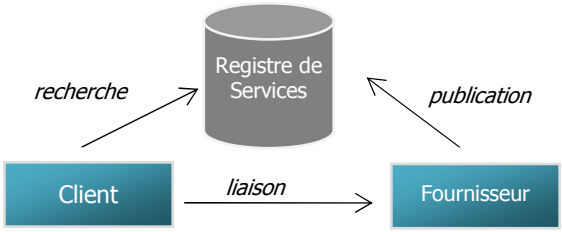

Figure 8. L'architecture orienté services.

Dans la SOA, un **fournisseur** d'un service représente une organisation (ou une personne) qui fournit une fonctionnalité sous la forme d'un service. Le fournisseur met à disposition des possibles consommateurs du service les informations nécessaires pour pouvoir utiliser le service. Ces informations sont regroupées dans la description du service, la description peut contenir des informations sur la fonctionnalité offerte, le comportement du service en cas de faille, les propriétés non-fonctionnelles du service et les protocoles de communication qui doivent être utilisés lors de sa consommation.

Pour la mise à disposition du service le fournisseur réalise la **publication** de la description du service auprès d'un **registre de services** (ou annuaire des services). Le registre de services joue un rôle d'intermédiaire entre les fournisseurs et les consommateurs des services.

Le **client** interroge le registre pour obtenir des descriptions de services disponibles pouvant satisfaire ses besoins. La **découverte** d'un service est réalisée grâce aux descriptions des services publiées. Le client sélectionne parmi les services découverts celui qui s'adapte le mieux à ses besoins et capacités. Une phase de négociation peut avoir lieu entre le client et le fournisseur du service afin de se mettre d'accord sur les conditions d'utilisation du service [AFM05]. Une fois le service choisi, le client utilise les informations disponibles dans la description pour effectuer une **liaison** avec le fournisseur et ensuite, pour consommer la fonctionnalité du service.

L'utilisation de ce cadre d'interaction permet à l'approche à services de bénéficier de deux propriétés importantes. D'abord, un découplage entre les clients et les fournisseurs des services, obtenu par l'intermédiation du registre de services. Le registre centralise l'information des services, donc les clients n'ont pas besoin de s'adresser directement aux fournisseurs avant de consommer le service. De plus, seule la connaissance de la description est partagée entre les différents acteurs, cette description n'a que l'information nécessaire pour rendre utilisable le service, ainsi les autres détails d'implémentation ne sont pas connus par le client, ce qui fait qu'il est indépendant de l'implémentation du service utilisé.

L'autre propriété importante est la capacité d'une liaison retardée, à différence des modèles à composants où l'architecture est établie au moment de la conception, dans l'approche à services, cette liaison peut être effectuée au moment de l'exécution. Un environnement à services est dynamique parce que des descriptions de services peuvent être publiées ou retirées du registre de services à tout moment, donc une liaison avec un client peut varier d'exécution en exécution, ou même, entre deux invocations différentes lors de la même exécution. Cette propriété de liaison retardée rend l'architecture du client reconfigurable à l'exécution.

3.2.1 Environnement d'exécution et d'intégration des services

Afin d'assurer la réalisation des différentes interactions définies dans l'architecture SOA, une plate-forme à services doit mettre en place un environnement d'exécution et d'intégration de services. Cet environnement doit être capable de gérer les interactions entre les acteurs définis dans l'architecture SOA.

Nous pouvons classer les éléments d'un tel environnement en deux catégories, illustrés dans la Figure 9 :

- les **mécanismes de base** qui permettent la publication, la découverte, la composition, la négociation, la contractualisation et l'invocation des différents services ;
- les **mécanismes additionnels** qui prennent en charge les aspects non-fonctionnels offerts par la plate-forme tels que la sécurité, la distribution, le dynamisme, la gestion de transactions etc.

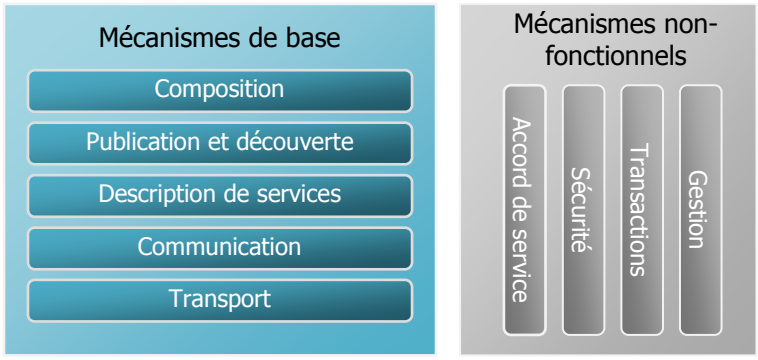

Figure 9. Mécanismes d'un environnement d'exécution et d'intégration de services.

Les mécanismes de transport et de communication constituent la base de l'environnement d'exécution et intégration de services, car ils permettent assurer la communication entre les acteurs. Le mécanisme de description de services permet de décrire les différents services présents dans l'environnement dans un langage spécifique. La description inclut les aspects fonctionnels et non-fonctionnels du service ainsi que la façon de le consommer. Le mécanisme de publication et de découverte, permet aux fournisseurs d'enregistrer les services qu'ils souhaitent rendre disponibles, et aux clients de rechercher des services qui répondent à ses besoins.

Les mécanismes additionnels assurent des aspects non-fonctionnels comme l'établissement d'un accord de service entre le fournisseur et le client, la sécurité, la gestion de transaction, la surveillance et le suivi des applications etc. Ces éléments sont additionnels et ajoutent de la valeur à l'environnement d'exécution.

Nous présenterons par la suite différentes plates-formes qui implémentent l'approche à services. Nous allons signaler les divers points de variation de ces plates-formes, tels que l'information incluse dans la description du service, le traitement du dynamisme de l'environnement, le moment de la liaison entre le client et le fournisseur. La mise en place des mécanismes de l'environnement d'exécution et l'intégration des services sera dévoilée dans la description de chaque plate-forme à services.

3.3 PLATES-FORMES A SERVICES

Diverses plates-formes à services ont été proposées, nous pouvons citer les services Web, la plate-forme OSGi, le modèle CORBA [Obj08] et JINI [Sunb]. Les services Web sont certainement la technologie la plus connue et la plus utilisé dans le monde industriel et académique pour la mise en place de d'architectures à services. L'explication de ce succès vient du fait que cette technologie utilise Internet et ses protocoles comme faisant partie de ses mécanismes de base.

La plate-forme OSGi, conçue à l'origine pour construire des systèmes qui s'exécutent sur des dispositifs ayant de fortes limitations en ressources matérielles (par exemple des téléphones portables), possède aujourd'hui un succès grandissant dans la création de systèmes plus « traditionnels », et devient de fait un standard pour la construction d'applications dynamiques.

Nous allons présenter ces deux plates-formes plus en profondeur car elles sont utilisées dans les travaux de cette thèse, en tant qu'objets d'étude, mais aussi comme un moyen pour mettre en œuvre notre approche.

3.3.1 Les services Web

Nous pouvons considérer la technologie de services Web comme l'évolution des technologies de middleware et de leurs mécanismes utilisés auparavant dans la création d'applications reparties. Les mécanismes comme l'invocation de procédures distante (RPC), et les langages de description d'interfaces (IDL) forment la base pour la mise en place actuelle de la technologie de services Web.

Un des objectifs de la technologie services Web est de fournir un cadre d'interopérabilité qui permet aux clients de services de les utiliser sans avoir à connaître les détails techniques de son implémentation, comme par exemple la plate-forme d'exécution ou le langage de programmation utilisé lors de son développement [ACKM03]. Pour atteindre cet objectif, cette technologie, au contraire des anciennes propositions, utilise le schéma d'interaction de la SOA et un ensemble de standards reposant sur le langage XML, standards définis par des organisations comme le W3C et OASIS.

Pour le W3C, un service Web est défini comme une entité logicielle dont les fonctionnalités sont rendues disponibles à travers un réseau (typiquement Internet). Un service Web décrit son interface en utilisant un langage qui est censé être traitée par des machines (WSDL). D'autres systèmes peuvent interagir avec le service web en échangeant des messages dans un protocole de communication établi (SOAP) [W3C04].

Un service web est perçu par un client comme une boîte noire qui fournit sa fonctionnalité par l'intermédiaire d'un ensemble de points d'accès ou ports. Du point de vue du fournisseur, les messages sont reçus par un composant de messagerie de la plate-forme d'exécution (un composant implémentant le protocole SOAP). Ce composant transforme les messages reçus (*decoding*) dans le format utilisé par la technologie d'implémentation, et ensuite ils sont dirigés vers le composant implémentant la fonctionnalité du service. L'implémentation du service peut elle-même invoquer d'autres services pour implémenter sa fonctionnalité ; ceci est connu comme la composition de services. Une fois la requête traitée, la réponse est envoyée par l'implémentation au composant de communication qui réalise la transformation inverse, du format de l'implémentation au format SOAP (*encoding*). Finalement, le client reçoit la réponse dans le format SOAP, sans avoir à connaître les détails de ces interactions. La Figure 10 présente la vision du client de service à gauche, tandis qu'à droite est montrée la vision du fournisseur.

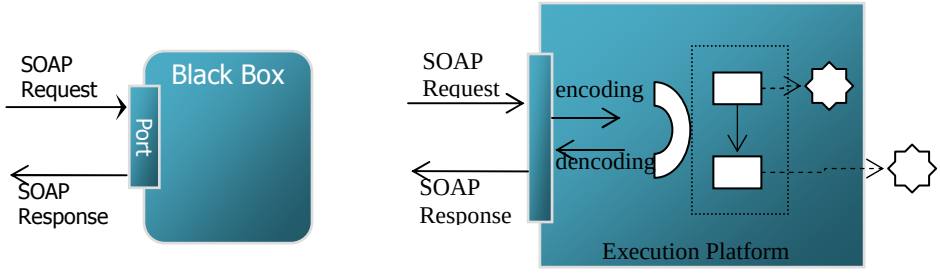

Figure 10. Implémentation d'un service web.

Les services web utilisent le plus souvent une communication synchrone, dans laquelle le client est bloqué jusqu'à la réception de la réponse de la part du service. Ce type de communication est approprié lorsque le temps de traitement du message est « bref » (de l'ordre de la seconde). Toutefois, une communication asynchrone est aussi possible ; dans ce cas le client envoie un message au service et continue son exécution sans être bloqué pour attendre la réponse. Ce type de communication est plus approprié s'il s'agit de messages dont le traitement peut mettre une période de temps plus importante (des heures ou de jours). Ce style de communication exige la connaissance de la part du service de la localisation de son client ou bien la mise en place d'une infrastructure de souscription/publication.

Un canevas de standards et de technologies basés sur le langage XML est utilisé pour spécifier l'environnement d'exécution des services web. Ces standards comprennent ; le langage WSDL (*Web Service Definition Language*) [W3C02b] pour la description de services, le protocole SOAP (*Simple Object Access Protocol*) pour la communication, et UDDI (*Universal Discovery and Integration*) [OAS04] comme service d'annuaire pour la publication et découverte des services. La figure ci-dessous montre comment sont organisés les standards qui forment une partie de la spécification de la plate-forme de services Web.

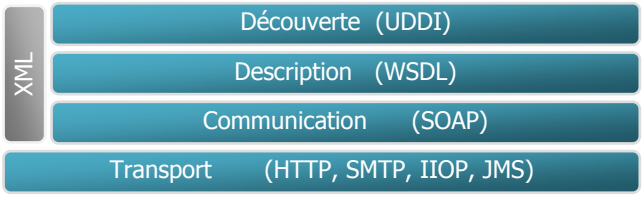

Figure 11. Environnement d'exécution de services web.

Ces standards fournissent les mécanismes de base pour la mise en place des environnements d'exécution et d'intégration des services Web. Nous allons par la suite les étudier plus en profondeur.

Le protocole de communication de services Web : SOAP

SOAP est un protocole RPC qui utilise une syntaxe base sur XML, son but est de permettre un échange standardisé de messages dans des systèmes distribués. A l'origine SOAP a été propose par Microsoft et IBM et aujourd'hui il fait partie des recommandations (standard) W3C.

Un message SOAP contient une enveloppe obligatoire, un entête facultatif, et un corps aussi obligatoire, ces composants sont présentés sur la forme d'un document XML. L'entête du message permet d'ajouter des extensions à un message, comme par exemple, des propriétés de sécurité du message ou des propriétés transactionnelles. SOAP a ignoré volontairement ces caractéristiques dans sa spécification parce qu'il a été conçu comme un protocole léger. Pour sa part, le corps du message contient l'information de l'opération à invoquer ainsi que les paramètres d'invocation.

SOAP peut utiliser divers protocoles de transport, le plus utilisé aujourd'hui est le protocole HTTP qui fourni l'avantage de pouvoir traverser de *proxies* et des pare-feu facilement, un problème qui existait dans les technologies de *middleware* antérieures. D'autres avantages sont l'indépendance vis-à-vis de la plateforme, du langage et de la technologie d'implémentation ainsi que son extensibilité et sa simplicité.

Par contre, SOAP présente une performance limitée, si nous comparons ce protocole avec ceux utilisés par d'autres technologies de *middleware* comme CORBA. Le problème de performance est associé principalement à l'utilisation de la syntaxe XML qui est verbeuse. Des alternatives ont été proposées afin de résoudre cette perte de performance comme par exemple MTOM (*Message Transmission Optimization Mechanism*) qui cherche de transmettre de contenu binaire plutôt que du texte.

Le langage WSDL

La séparation entre la description du service et son implémentation rend possible le découplage entre le service et ses clients puisque un client ne nécessite pas de connaître les détails d'implémentation du service.

Dans la technologie de services Web, le langage utilisé pour faire la description de services est nommé WSDL [W3C02b]. WSDL peut être considéré comme une évolution du langage IDL de CORBA [EAS08], mais WSDL, comme la plupart des standards des services web, utilise une syntaxe basée sur le langage XML. WSDL vise trois buts, d'abord, décrire la fonctionnalité du service (ensemble des opérations supportées), ensuite spécifier la façon de consommer le service (les protocoles de communication et méthodes de codage de données à utiliser) et finalement, indiquer la localisation du service (l'adresse réseau où le service est rendu disponible).

Une description WSDL est divisée en deux parties, la première contient la description abstraite du service, dont la liste des opérations et des messages utilisés par le service. La deuxième partie contient la description de l'accès au service ainsi que sa localisation et les protocoles utilisés lors de sa consommation.

WSDL s'articule autour de six éléments que nous présentons plus en détail par la suite :

- *Types* **:** contient la description des types de données utilisés par le service Web ;
- *Message :* est une définition abstraite des messages à échanger pour interagir avec le service Web ;
- *Operation :* décrit de façon abstraite une méthode proposée par le service web. Chaque opération utilise des messages d'entrée, de sortie et d'exception ;
- *PortType :* correspond à une interface abstraite du service Web. L'interface contient un ensemble d'opérations ;
- *Binding :* décrit un protocole concret pour une interface abstraite (*PortType*) ;
- *Port :* spécifie un point d'accès au service Web. Chaque port indique l'interface fournie (*PortType*), le protocole utilisé pour l'accéder (*Binding*) et une adresse réseau ;
- *Service :* correspond à un ensemble de ports.

Le langage WSDL exprime la fonctionnalité d'un service Web, mais il ne tient pas en compte de ses propriétés non-fonctionnelles. Des extensions pour exprimer ces propriétés ont été proposées, comme par exemple WS-Policy [W3C06]. En se basant sur la description, les clients recherchent les services en utilisant l'annuaire UDDI.

L'annuaire UDDI

Un des propriétés clés de l'approche à services est la possibilité de partager les implémentations de services existants. Dans le cas de services Web, les services sont rendus disponibles sur Internet ou sur l'intranet d'une entreprise par le biais d'un annuaire. Le standard UDDI spécifie comment le composant d'annuaire doit être mis en place pour la plate-forme de services Web.

L'objectif d'UDDI est d'être un standard pour le registre et recherche de services en se basant sur les descriptions de services (WSDL). Ce registre ainsi que tous les composants de la technologie de services Web doit être indépendant de la plate-forme d'exécution. L'information disponible dans l'annuaire doit permettre aux clients d'avoir la connaissance des entreprises fournissant les services, ainsi que des services disponibles pouvant répondre à leurs attentes.

Conceptuellement, un annuaire UDDI contient trois composants principaux:

- Les pages blanches : proposant des informations des entreprises, comme le nom, l'adresse, le contact, etc.

- Les pages jaunes : proposant une catégorisation des services selon une taxonomie industrielle standard.
- Les pages vertes : proposant d'information technique des services. Pour retrouver cette information les pages vertes référencent les fichiers de description WSDL des services enregistrés dans l'annuaire.

Le standard UDDI ne connaît pas une utilisation répandue aujourd'hui, des entreprises ont choisi de mettre en place ces propres mécanismes d'annuaire de services, généralement fonctionnant à l'intérieur d'elles mêmes, quelques unes de ces solutions implémentent juste une partie de la spécification UDDI. Nous considérons que l'utilisation d'un standard d'annuaire pour les services Web s'achèvera lorsque des technologies plus robustes et complètes seront mises en place.

Environnement d'exécution de services web et ESB

Il n'existe pas une implémentation de référence d'un environnement d'exécution de services web, et les services offerts par ces environnements peuvent varier significativement selon les implémentations. Nous pouvons imaginer un environnement très simple où seulement des mécanismes de communication sont assurés. Dans cette classe d'environnement, un compilateur se charge de générer des *stubs* (du client et du serveur) et utilise à l'exécution des librairies implémentant le protocole SOAP. De l'autre côte du spectre, nous trouverons des *middlewares* qui offrent des mécanismes sophistiqués afin d'assurer des capacités plus complexes, comme la sécurité, la médiation, le routage et persistance de messages, etc.

Une classe de *middleware* implémentant un environnement pour l'exécution, l'intégration, le déploiement et la gestion de services web sont les ESB (*Enterprise Service Bus*). Un ESB offre un bus de communication pour l'interaction de services et la possibilité de publier les applications et les sources de données d'une organisation comme des services Web et de les composer. Un ESB fourni comme services principaux:

- un bus de communication qui supporte différents types de communication et qui offre de propriétés de qualité de service sur la communication comme la sécurité, la médiation, la persistance, le routage et le support des transactions ;
- des mécanismes permettant d'exposer des applications et des sources de données hétérogènes comme des services avec une interface WSDL ;
- des mécanismes de sélection utilisant les descriptions de services (WSDL et politiques) pour découvrir et sélectionner les services.
- des systèmes d'administration et de monitoring des services déployés.

Conclusion

Une plate-forme implémentant l'environnement d'exécution de services Web permet la création d'applications en utilisant des briques logicielles qui peuvent être mises à disposition à travers Internet. Cette disponibilité sur Internet fait que les organisations peuvent se servir des fonctionnalités métiers qui seront rendues disponibles par d'autres organisations et ainsi parvenir à implémenter des échanges B2B (*Business to Business*). Les principes et protocoles de la plateforme de services Web ont été repris par des technologies comme UPnP [UPn08] (*Universal Plug and Play*) et DPWS [Mic06a] (*Device Profile for Web Services*) dont l'objectif est l'utilisation de l'architecture SOA dans les réseaux de dispositifs, spécifiquement dans la domotique.

Le succès des services Web peut être expliqué pour deux raisons, d'abord l'adoption de l'architecture SOA qui permet de bénéficier de ses propriétés. Ensuite, cette plate-forme a attaqué un de principaux problèmes des applications reparties, celui de l'interopérabilité. De nombreuses tentatives pour résoudre le problème de l'interopérabilité étaient proposées auparavant, telles que CORBA ou EDI, mais, grâce à l'adoption d'un nombre important de

standards comme SOAP, WSDL et UDDI les services Web sont parvenu à résoudre cette difficulté au niveau de l'interaction.

Néanmoins divers problèmes pour l'adoption de cette technologie existent encore. Tout d'abord, les mécanismes de base ne sont pas assez riches, par exemple, le langage de description ne permet pas de décrire toute l'information dont les clients peuvent avoir besoin au moment de sélectionner un service. Ensuite, le protocole de communication SOAP est assez verbeux ce qui dégrade la performance des systèmes qui l'utilisent. De plus la complexité technique associée à la réalisation d'une solution basée sur des services Web est assez élevée, due au grand nombre de standards et de leurs implémentations. Finalement, nous pouvons citer les problèmes associés à la composition des services Web, et à la difficulté de combiner de façon cohérente les différents standards. Ce point sera abordé dans le chapitre dédié à l'orchestration de services.

3.3.2 OSGi et iPOJO

OSGi est une plate-forme à services proposée par l'Alliance OSGi dans les années 2000 ; la version 4 est la plus récente et date de 2005. La spécification définit un environnement d'exécution de services et de déploiement modulaire d'applications. La description d'un nombre de services techniques est aussi incluse dans la spécification.

La spécification de OSGi avait au départ l'objectif de fournir un *middleware* pour les passerelles réseaux et résidentielles. Etant donné les limitations matérielles de ces passerelles en taille mémoire et capacité d'exécution, les composants logiciels créés pour ce type de dispositifs ont des contraintes fortes. En fait, de nouveaux composants logiciels doivent pouvoir être déployés et assemblés dynamiquement sur la passerelle, sans interrompre l'exécution des autres composants logiciels installés.

Aujourd'hui, la plate-forme OSGi devient de fait le standard dans la création d'applications avec des capacités de reconfiguration dynamique. Par conséquent, elle est utilisée dans de nouveaux domaines d'application, comme par exemple la téléphonie et l'automobile pour ce qui concerne l'informatique embarquée. En plus, elle est utilisée pour la construction de logiciels de taille importante et de nature diverse comme par exemple l'environnement de développement *Eclipse [GHM+05]* ou encore les serveurs d'applications JEE *OW2 JOnAS* [Des07] et *Sun Glassfish [Suna]*. Il existe plusieurs implémentations d'OSGi, nous pouvons citer par exemple Felix du consortium Apache, et Equinox développée par IBM.

Architecture OSGi

La plate-forme OSGi propose la construction d'applications en suivant un modèle architectural au-dessus de la plate-forme Java, comme présenté dans la Figure 12. Le premier objectif de cette plate-forme est de rendre modulaire les applications. Pour atteindre cet objectif, OSGi fournit un système de gestion de déploiement des applications. Ce système permet de déployer de façon indépendante les différents modules faisant partie d'une application. L'unité de déploiement et paquetage d'OSGi est le *bundle*.

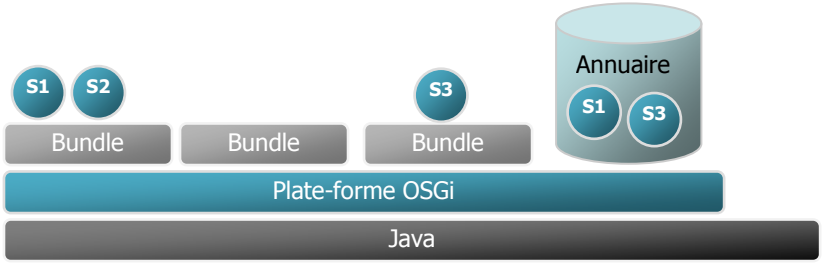

Figure 12. Architecture de la plateforme OSGi.

Un *bundle* est un fichier au format JAR contenant des classes Java pour implémenter la logique métier, des ressources (images, fichiers de configuration, etc.), et de méta-données contenues dans le fichier *manifeste*.

Les *bundles* sont déployés et administrés à l'exécution. Il est donc possible d'installer, de démarrer, d'arrêter, de mettre à jour ou de supprimer des bundles à l'exécution sans interrompre la plate-forme. Cependant, la plate-forme doit assurer que toutes les dépendances d'un bundle au niveau de code soient résolues au moment de le démarrer. Par conséquent, un bundle doit spécifier explicitement ses dépendances vers les autres bundles, cette expression est faite au niveau de packages Java, dans le fichier de manifeste.

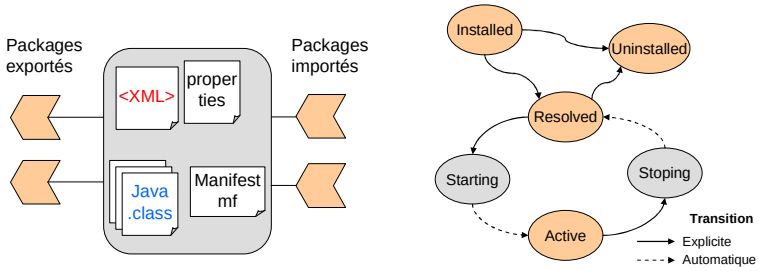

Figure 13. Bundle - Unité de déploiement OSGi.

Le cycle de vie d'un bundle dans la plate-forme est défini par six états : installé, désinstallé, résolu, en démarrage, activé, arrêté. Un bundle doit fournir une classe « Activateur » afin de gérer son cycle de vie. La Figure 13 présente le format d'un bundle et le diagramme d'états associé au cycle de vie d'un bundle.

OSGi, une plate-forme à services

Une plate-forme OSGi peut être utilisée comme un *middleware* de déploiement d'applications qui architecturalement sont divisés en bundles ; jusque là le concept de service n'apparaît pas dans ce modèle. Nous pouvons considérer que la plateforme à services est créée au-dessus de ces mécanismes de gestion de déploiement présentés auparavant.

La plate-forme OSGi est centralisée et basée sur le langage Java, cette caractéristique fait qu'elle n'a pas besoin de mécanismes de transport (exécution centralisée) et que le mécanisme de communication est basé sur les appels de méthodes Java. Quant à la description des services, elle est faite à l'aide d'interface Java et d'un ensemble de propriétés dont la sémantique est laissée libre aux développeurs. Ensuite, la plate-forme fourni un annuaire pour le registre et recherche des services. En plus du registre, la plate-forme ajoute des mécanismes permettant d'informer les applications de la disponibilité de nouveaux fournisseurs de services, ainsi que du retrait de services. Ce mécanisme de notification permet de réaliser la reconfiguration dynamique des applications, par exemple remplacer une instance de service par une nouvelle qui s'ajuste mieux aux besoins de l'application.

La plate-forme fournit différents mécanismes afin de mettre en place l'architecture SOA, par contre les interactions définies par l'approche doivent être gérées manuellement par le développeur. Cette tâche est complexe car elle demande une grande connaissance des mécanismes fournis par OSGi. La technologie iPOJO propose une gestion automatique des interactions.

IPOJO

IPOJO (acronyme de *injected Plain Old Java Objects*) est un modèle à composants orienté services. Un modèle à composants orienté services combine les avantages de l'approche à composants et de l'approche à services. Dans un tel modèle, une spécification de services est implantée par le biais d'un composant. Les dépendances d'un composant sont exprimées en termes des spécifications de services et elles sont résolues en utilisant le modèle d'interaction de l'approche à services. De cette façon, dans une application un composant peut être remplacé par n'importe quel autre composant pourvu qu'il respecte la même spécification de service. En utilisant une approche à composants orienté services, nous pouvons bénéficier d'une part, d'un modèle de développement simple et d'une description de la composition comme le préconise

l'approche à composants, et d'autre part d'un faible couplage et de la liaison retardée, caractéristiques apportées par l'approche à services.

La technologie iPOJO implante un modèle à composants orienté services pour la plate-forme OSGi, pourtant il reprend les mécanismes de base de cette plate-forme. iPOJO utilise la notion de conteneur, afin de dispenser au développeur de connaître les détails techniques des mécanismes de base offerts par la plate-forme OSGi. Donc, le conteneur est le responsable des interactions du composant avec la plate-forme. Le conteneur d'un composant iPOJO prend en charge des aspects techniques comme l'enregistrement de services dans l'annuaire, la découverte et sélection de services et la gestion des dépendances d'un composant.

Le conteneur est responsable non seulement des aspects techniques d'interaction avec la plate-forme OSGi, mais aussi d'assurer les propriétés non-fonctionnelles des composants iPOJO. Un conteneur iPOJO est composé de plusieurs *handlers* chacun d'entre eux prenant en compte un aspect non-fonctionnel du composant. Les *handlers* disponibles actuellement par défaut sont :

- *Service Requirement* qui gère la gestion des dépendances de services du composant,
- *Provided Service* qui gère la publication dans l'annuaire OSGi des services fournis par le composant,
- *Lifecycle Callback* qui gère le cycle de vie des instances du composant,
- *Configuration Handler* qui permet de réaliser la configuration d'un composant,
- *Event Handler* qui est chargé de la communication par événements.

Un important avantage d'iPOJO par rapport à d'autres modèles à composants utilisant la notion de conteneur, est son extensibilité. Il est possible dans iPOJO d'ajouter de nouveaux *handlers* au conteneur d'un composant pour couvrir d'autres aspects non-fonctionnels non pris en compte par les *handlers* fournis par défaut.

Conclusion

La plate-forme OSGi vise l'utilisation de l'architecture SOA dans un contexte centralisé et fournit un système de gestion de déploiement bien établi en dessus de la plate-forme Java. Un point fort de la plate-forme OSGi est la capacité de reconfiguration dynamique de l'architecture des applications. En fait, elle met à disposition un mécanisme de notification de changements d'état du registre (publication et retraite de services), permettant aux intéressés d'agir en conséquence et de modifier leur architecture par rapport à la disponibilité des services.

Malgré ses propriétés, des difficultés sont aussi rencontrées dans cette plateforme. Tout d'abord, il est difficile de l'utiliser dans un contexte reparti. Ensuite, il n'est pas simple d'exprimer des propriétés d'un service en utilisant une approche de couples clés et valeurs comme proposé par OSGi. En plus, les interactions avec la plate-forme sont la responsabilité du développeur des applications. Finalement, seulement des applications Java peuvent être utilisées dans cette plate-forme, donc elle n'est pas aussi ouverte et générique que les services Web.

La technologie iPOJO simplifie le développement de services pour la plate-forme OSGi, notamment en cachant la complexité associée aux interactions des applications avec la plate-forme. iPOJO fourni aussi un moyen d'étendre les conteneurs de composants afin d'ajouter le support pour d'autres aspects non-fonctionnels.

3.4 SYNTHESE

Nous avons présenté dans ce chapitre l'approche à services, qui définit un nouveau paradigme de programmation plaçant le service comme concept central. L'architecture SOA propose un cadre conceptuel pour l'utilisation de cette approche, elle spécifie les acteurs et leur modèle d'interaction pour la construction d'applications à services.

Nous avons mis en évidence la propriété la plus importante de l'approche à services, à savoir le faible couplage existant entre les entités composant une application. Cette propriété permet, l'évolution indépendante des différentes parties d'une application. Une autre propriété intéressante est la possibilité de construire des applications multifournisseur, dans ce type d'applications il n'existe pas une unité centrale d'administration.

Différentes implémentations de l'approche à services ont vu le jour, nous avons étudié dans ce chapitre les services Web et la plate-forme OSGi. Chacune de ces plates-formes est utilisée dans des contextes différents.

Les services Web proposent la mise à disposition des applications à travers d'un réseau. Ils sont utilisés aujourd'hui essentiellement pour l'intégration des applications patrimoniales, et les services ainsi exposés sont d'une grosse granularité.

OSGi est utilisé dans des contextes plus étroits. OSGi est utilisé dans la création d'applications avec des besoins de modularité importants, comme les systèmes à plugins ou les serveurs d'applications. En plus, des applications d'informatique embarquée, cherchant la mise en place d'applications dynamiques profitent des propriétés fournies par OSGi. Un tableau comparatif entre les deux plates-formes est présenté ci-dessous :

	Services Web	OSGi
Description	WSDL	Java + Propriétés
Communication	SOAP (RPC sur XML)	Appel de méthode Java
Contexte d'exécution	Reparti	Centralisé
Reconfiguration dynamique	Possible, mais ne fait pas partie de la spécification.	Supporté avec des mécanismes fournis par la plate-forme
Type de liaison	Statique Dynamique (Active)	Dynamique (Active et passive)
Unité de déploiement	Défini par la technologie d'implémentation.	Bundle.
Technologie d'implémentation	Indépendant de technologie	Java

Figure 14. Tableau comparatif des plates-formes de services Web et OSGi.

L'approche à service fourni un cadre de base qui sera utilisé non seulement pour réaliser des interactions entre les différents acteurs, mais aussi dans la construction de services plus complexes qui sont créés à partir de services basiques. Cette opération est connue comme la composition de services, particulièrement la technologie d'orchestration de services qui sera dévoilée dans le chapitre suivant est de notre intérêt, car elle est la base de notre approche.

4. ORCHESTRATION ET CHOREGRAPHIE DE SERVICES

L'approche à service fournit des mécanismes de base afin que des clients puissent découvrir des services rendus disponibles dans un registre, réaliser une liaison avec le fournisseur de services et finalement invoquer les opérations des services sélectionnés. Ces mécanismes ouvrent la possibilité de construire des applications en utilisant les fonctionnalités d'un ensemble de services rendus disponibles dans ce registre ; cette opération d'intégration est appelée la composition de services. Le résultat d'une composition de services peut être une application ou un autre service nommé service composite [ACKM03]. Cette propriété fait que la composition de services soit récursive ou hiérarchique, c'est-à-dire que des services atomiques ou composites peuvent être intégrés pour implémenter la logique d'autres services composites [KL03]. La composition de services peut être vue comme une façon de maîtriser la complexité, car des services complexes sont construits d'une façon incrémentale à partir de services de niveau d'abstraction plus bas et de granularité plus fine [ACKM03].

Dans ce chapitre, nous allons présenter les principaux concepts associés à la composition de services. D'abord, nous montrons comment le processus de composition est défini, ensuite deux approches de composition de services sont dévoilées, l'approche de composition structurelle et celle orientée procédé.

L'approche de composition orientée procédé nous intéresse particulièrement, donc elle sera présentée plus en détail. La composition orientée procédé reprend des idées déjà mises en pratique par les technologies de *workflow*, mais, en ajoutant les propriétés fournies par l'approche à services [vdA03]. La composition orientée procédé peut être découpée selon le point de vue des acteurs participant à la composition, cette distinction fait la différence entre les technologies d'orchestration et de chorégraphie de services. L'orchestration de services sera présentée dans la seconde section de ce chapitre tandis que la chorégraphie de services sera développée dans la troisième section.

Finalement, nous concluons le chapitre par une synthèse des différentes approches afin d'identifier leurs points forts et leurs limitations. La fin de cette synthèse propose des pistes de réflexions engagées pendant l'élaboration de cette thèse et qui seront développées dans la suite du document.

4.1 LA COMPOSITION DE SERVICES

La composition de services est un processus de raffinement permettant de passer d'une spécification abstraite de la composition vers une description exécutable. Les étapes pour la création d'une composition de services ont été définies par [YP04] comment suit :

- une phase de définition permettant de spécifier d'une manière abstraite la composition de services. Dans cette phase, il faut identifier en premier la fonctionnalité devant être fourni par la composition ainsi que, la fonctionnalité

devant être apportée par les différents participants. Finalement, les interactions entre les participants sont spécifiées.

- une phase de planification servant à déterminer comment et à quel moment les services seront exécutés. Dans cette phase, la conformité et compatibilité des services doivent être vérifiées.
- Une phase de construction fournissant une composition concrète et sans ambiguïtés prête à s'exécuter. Uniquement, les services potentiellement disponibles à l'exécution restent à déterminer.
- Une phase d'exécution implémentant les liaisons avec les services disponibles et ensuite exécute la composition.

Ce processus de composition de services est une activité de longue durée, chacune des phases définies auparavant peuvent être elles-mêmes divisées en sous-tâches. Souvent ces sous-tâches doivent être réalisées manuellement ou avec des mécanismes qui ne sont pas les plus appropriés. Par exemple, dans la phase de planification, lorsque les services à composer doivent être identifiés, il apparaît généralement des problèmes de compatibilité. Des mécanismes de médiation doivent être utilisés afin de résoudre différents types d'incompatibilités, ceci est une tâche difficile qui, dans la plupart des approches est effectuée de façon manuelle.

La complexité de la composition de services peut être associée d'une part à la complexité de la logique métier inhérent aux applications, d'autre par à la complexité de la mise en œuvre de l'approche à services. Donc, il est nécessaire de fournir aux développeurs des outils et des mécanismes d'abstraction pertinents, afin de se concentrer sur la logique métier de l'application plutôt que dans les détails techniques. Par conséquent, différentes approches de composition de services ont vu le jour, elles essayent d'attaquer de différente manière les problèmes associés au processus de composition.

Nous allons classifier ces approches par rapport à la façon de réaliser le contrôle de flux entre les services de la composition. Le contrôle d'une composition de services peut être extrinsèque ou intrinsèque aux services. Ces deux possibilités de gestion du contrôle définissent deux styles de compositions [FS05] : la composition structurelle et la composition orientée procédé qui sont présentées dans les deux sections suivantes.

4.1.1 La composition structurelle

Dans la composition structurelle, les composants qui fournissent les services sont clairement identifiés, chaque composant définit explicitement ses interfaces fournies comme celles requises. La composition est spécifiée comme l'assemblage des composants, elle exprime des liens entre couples d'interfaces fournie et requise par deux composants. Le formalisme utilisé pour indiquer cet assemblage dépend de l'approche suivie, il est connu comme langage de description d'architecture ADL (*Architecture Description Language*).

La logique de contrôle, exprimant comment et à quel moment les opérations des services composés doivent être invoquées, est implicite et repartie entre les différents composants. La Figure 15 présente un exemple de composition structurelle. Dans l'exemple, le control de flux indiquant comment sont consommés les services fournis par les composants C2 et C3, se trouve à l'intérieur de l'implémentation du composant C1 (C1.java).

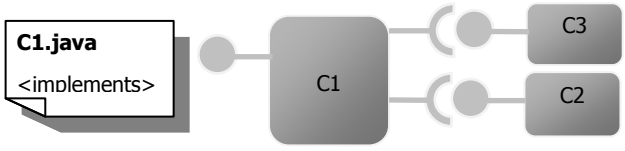

Figure 15. Composition structurelle.

Des approches suivant ce type de composition ont été proposés autant au niveau industriel que académique. Nous allons présenter l'approche industrielle SCA qui s'intéresse à

la composition de services Web et le modèle de composants iPOJO qui s'intéresse à la composition de services OSGi.

SCA

La technologie SCA permet d'utiliser un modèle à composants pour décrire la composition de services. L'objectif de SCA est, d'une part, de spécifier un ensemble de composants (unités de composition) formant une application, et d'autre part de décrire comment les services fournis par ces composants sont assemblés afin de construire l'application [Cha07].

Un composant SCA exprime explicitement les services qu'il fournit, les dépendances vers d'autres composants (références dans la terminologie SCA), et les propriétés permettant de configurer le composant lors de son instanciation. La description d'un composant est séparée de l'implémentation du composant qui est considérée dans SCA comme un aspect technique [BII+05]. La description offre aux assembleurs une vision uniforme des unités de composition.

En plus du modèle de composants, SCA fournit un modèle d'assemblage de composants en s'appuyant sur le langage de description d'architecture SCDL (*Service Component Definition Language*). SCDL introduit la notion de composite, un composite regroupe un ensemble de composants et spécifie les connexions entre eux. Un composite SCA peut lier des composants hétérogènes, c'est-à-dire créés avec différentes technologies d'implémentation. Les composites SCA, peuvent eux-mêmes être exposés comme des composants SCA ce qui permet un modèle de composition hiérarchique. Par conséquent, un composite SCA à la même vue externe qu'un composant, donc il spécifie un ensemble de services, de références et de propriétés.

Les services fournis par un composite SCA existent parmi les services fournis par ses composants. Un mécanisme connu sous le nom de promotion permet de faire une mise en relation entre les services fournis par le composite et ceux de ses composants internes. De la même façon, les références peuvent aussi utiliser ce mécanisme pour indiquer qu'une dépendance d'un composant interne sera résolue à l'extérieur du composite. Les propriétés d'un composite peuvent être aussi promues afin de pouvoir configurer les composants internes à partir des valeurs indiquées pour le composite.

Pour spécifier les liaisons entre les composants d'un composite, le concept de *wire* est introduit. Un *wire* connecte une référence d'un composant avec un service fourni par un autre. Une représentation graphique d'un composite SCA est schématisée dans la Figure 16.

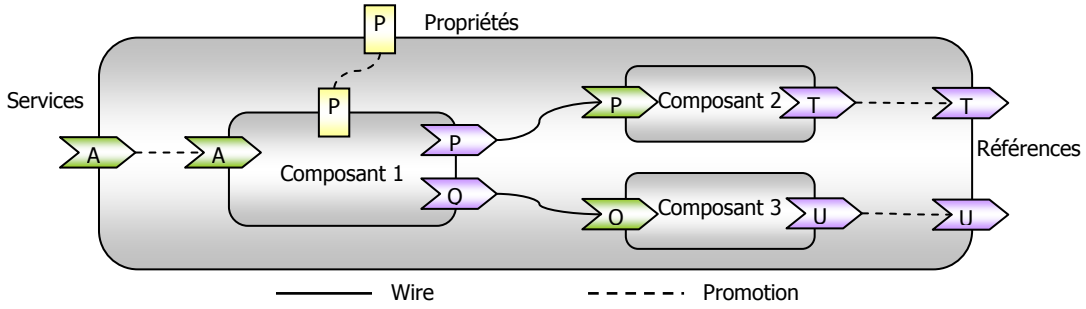

Figure 16. Représentation graphique d'un composite SCA.

Plusieurs implémentations fournissant une infrastructure d'exécution pour les concepts définis par SCA sont disponibles actuellement. Parmi, ces implémentations nous pouvons citer *Apache Tuscany [Apa08]*, *Newton [New08]*, et *Fabric3 [Fab08]*.

IPOJO

Nous avons présenté le modèle à composants orienté services dans le chapitre dédié à l'approche à services. Dans cette section, nous nous intéressons aux capacités de composition d'iPOJO.

Le schéma de composition dans iPOJO est hiérarchique comme dans SCA. Ainsi, un composite iPOJO est un composant, donc il peut offrir de services et il peut dépendre d'autres services. Par contre, à la différence de SCA, qui utilise seulement des composants comme unités de composition, dans iPOJO, il est possible de faire la composition de trois types d'éléments différents : des composants, des services abstraits et d'autres composites. Les spécifications fournies ou requises par un composite peuvent être déclarées à partir des éléments qu'il contient. Donc, des mécanismes d'import et d'export de spécifications de service sont disponibles.

Une notion de contexte est incluse dans le modèle de composition d'iPOJO. Donc, les services fournis par un composant ne sont visibles qu'à l'intérieur d'un contexte. Ainsi, les services fournis par les composants d'un composite, sont exploitables seulement par des autres composants se trouvant à l'intérieur du même composite. Le contexte agit comme une forme d'annuaire à l'intérieur du composite iPOJO. Ce mécanisme permet à iPOJO de fournir la propriété d'isolation de services.

Synthèse

SCA et iPOJO ont été créés avec des objectifs différents. IPOJO cherche à cacher la complexité de la plateforme OSGi pour des applications qui veulent utiliser au maximum les possibilités de dynamisme de cette plate-forme. SCA est une spécification qui n'est pas liée à une plate-forme spécifique, et qui essaye de résoudre les problèmes de l'interopérabilité de services. Les deux modèles partagent des caractéristiques communes comme la définition explicite de dépendances (références dans SCA et services abstraits dans iPOJO), ainsi que l'utilisation d'un langage de description d'architecture la composition. Le tableau ci-dessous compare les deux approches.

	SCA	iPOJO
Résolution de dépendances	Conception, déploiement ou exécution.	Exécution
Unités composés	Composants	Services, Composants, Composites
Architecture d'exécution	Distribuée	Centralisée
Dynamisme	N'est pas spécifie	Supporte la reconfiguration dynamique des applications
Interopérabilité	Implémentation dans diverses technologies.	Java (OSGi)

Figure 17. Tableau comparatif des approches de composition structurelle : SCA et iPOJO.

4.1.2 La composition orientée procédé

Dans cette approche, la composition est spécifiée en utilisant un modèle de procédé décrivant la logique de coordination et d'exécution des services utilisés par le composite. De cette façon, un service composite joue le rôle d'un coordonnateur de services. Un avantage de l'approche de composition orientée procédé est le fait qu'elle rendre explicite la logique de contrôle de la composition. En plus, cette logique est externe par rapport aux services composés car elle est décrite en dehors de ces services.

Le modèle de procédé est représenté généralement par un graphe, où chaque nœud correspond à l'invocation d'une opération d'un service, et les arcs servent à exprimer l'ordre d'enchaînement de ces invocations. Généralement, la spécification de la composition est faite dans un langage qui est interprété par un moteur d'exécution. La responsabilité du moteur est d'invoquer les services dans l'ordre spécifié, de faire le routage des données, de maintenir et gérer l'état des activités et du composite, ainsi que de gérer les situations d'exception.

Dans la composition orientée procédé, nous pouvons distinguer deux catégories de composition, l'orchestration et la chorégraphie de services :

- l'orchestration de services (schématisée dans la partie gauche de la Figure 18), représente la composition des services vue par un des participants de la composition.

Elle décrit du point du service composite, les interactions de celui-ci avec des autres services, ainsi que les opérations internes (ex. transformations de données) réalisées entre ces interactions.

- la chorégraphie de services (schématisée dans la partie droite de la Figure 18) décrit la collaboration entre un ensemble de services dont le but est d'atteindre un objectif commun. Dans ce cas, une vision globale de la composition est donné, et seulement sont visibles les interactions publiques de participants. En plus, il n'existe pas un agent central chargé de contrôler les activités de la collaboration, car chaque participant doit connaître sont rôle dans la collaboration.

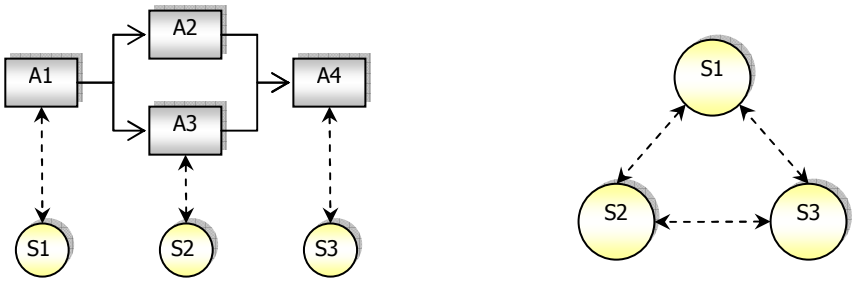

Figure 18. Orchestration et chorégraphie de services.

4.1.3 Synthèse

Nous avons utilisé le flux de contrôle afin de distinguer entre les deux principales tendances existant actuellement pour la composition de services. Cette caractérisation ne doit pas être considérée comme un critère de qualité ou de supériorité d'une tendance sur l'autre. En effet, le choix est lié au contexte de réalisation de la composition de services. La composition structurale est utilisée dans des domaines fermés où les développeurs ont le contrôle sur le cycle de vie des services composés. La composition orientée procédé est plus utilisée dans le contexte d'interactions entre différentes organisations et dans l'intégration d'applications.

En plus, la granularité des services peut être considérée comme un autre critère de sélection d'une approche. Les services à gros grain sont composés avec une vision plus centrée sur le métier, ce qui favorise une approche orientée procédé. En revanche, les fonctionnalités implémentées avec des services à grain fin correspondent à une vision plus programmatique, et une composition structurelle est plus pertinente. Finalement, l'utilisation combinée des approches est aussi possible, par exemple dans l'approche SCA, l'ADL décrit l'architecture de l'application, mais l'implémentation des composants peut elle-même utiliser une technologie orientée procédé comme WS-BPEL.

4.2 ORCHESTRATION DE SERVICES

Dans cette section, nous allons réaliser une caractérisation des modèles et canevas d'orchestration de services. Puis, nous détaillons certaines approches proposées par des industriels comme le standard WS-BPEL et d'autres issues du monde académique. Finalement, une synthèse permettra de faire un récapitulatif des approches, de leurs points forts et de leurs limitations.

4.2.1 Caractérisation des modèles d'orchestration de services

Afin de caractériser les modèles d'orchestration de services, dans [ACKM03] les auteurs ont proposé six dimensions, à savoir : la technologie des éléments composés, le formalisme de description de l'orchestration, le modèle de données et d'accès aux données, la sélection des services participants, le modèle de gestion de transactions utilisé et finalement le modèle de gestion des exceptions.

Nous allons reprendre ces dimensions, sauf celle du modèle de gestion transactionnel puisqu'elle est en dehors de la porte de cette thèse. En revanche, nous allons analyser autres deux dimensions, le support des aspects non-fonctionnels de la part des approches d'orchestration et finalement le critère d'extensibilité des modèles d'orchestration.

Technologie des éléments composés

Cette dimension détermine les hypothèses faites par l'approche d'orchestration par rapport au type des éléments qui seront composés. Plusieurs approches utilisent les services Web comme unités de composition. Par conséquent, la composition suppose que les éléments sont décrits avec le langage WSDL, qu'ils communiquent en utilisant le protocole SOAP, et que les données échangées sont représentées par des documents XML. Parmi ces approches nous trouvons WS-BPEL [OAS07b] et SELF-SERV [BSD03]. Pour sa part, eFlow [CIJ+00] utilise les e-Services (précurseur des services Web) comme unité de composition.

D'autres approches d'orchestration supportent un ensemble prédéfini de technologies capables de composer. Par exemple, le langage BPELJ [BGK+04] (extension du WS-BPEL) supporte la technologie de services web et du code Java (*Java Snippets*). Ces approches sont plus générales mais ont l'inconvénient d'être techniquement plus compliqué à cause de l'hétérogénéité des composants utilisés.

Une troisième alternative, consiste en fournir un support prédéfini pour un ensemble de technologies et offrir des mécanismes pour supporter l'ajout d'autres. Par exemple, JOpera [PHA06] supporte la composition de services Web, de code Java (*Java Snippets*) et d'un langage de *script*. L'avantage principal est la généralité et la capacité d'extension, mais l'hétérogénéité des composants et la complexité associée aux mécanismes d'extension rend problématique ce type d'approche.

Formalisme de description de l'orchestration

Cette dimension analyse le langage utilisé pour décrire l'ordre des invocations des services ainsi que les conditions nécessaires pour réaliser ou non l'invocation. Les approches d'orchestration de services généralement reprennent des formalismes issus de la technologie des *workflow*.

Un type de formalisme bien connu de la technologie de *workflow* est le diagramme d'activité. C'est un graphe dirigé où les nœuds correspondent aux activités (tâches) et les arcs représentent des liens exprimant des contraintes de flux de contrôle et/ou de flux de données entre les tâches. Les activités peuvent être spécialisées pour réaliser des tâches spécifiques comme par exemple l'invocation d'une opération d'un service Web. En plus, dans certains formalismes, il est possible d'exprimer des conditions sur les liens, pour indiquer par exemple, si un flot de contrôle est suivi ou non. Le langage d'orchestration WSFL [Ley01] et le canevas JOpera [PHA06] utilisent ce type de formalisme, le langage APEL [EDA98] peut être classe dans cette catégorie.

Bien que le diagramme d'activités soit le type de formalisme le plus utilisé par les approches d'orchestration, d'autres formalismes proposent des propriétés intéressantes et ils sont surtout utilisés par des approches académiques. Ainsi, les diagrammes d'états permettent de décrire un ensemble d'états et les transitions entre eux. Les transitions sont associées à des règles ECA (Evénement, Condition, Action), de cette façon lorsqu'un événement arrive, la condition est évaluée, ensuite l'action est réalisée et la composition change son état. Le canevas SELF-SERV [BSD03] et le système Mentor [WWWD96] utilisent formalismes de ce type.

Il existe une catégorie de formalismes plus formels, permettant l'analyse des propriétés structurelles des spécifications afin de détecter des problèmes comme par exemple les *deadlocks* ou les *livelocks*. Dans ce groupe nous trouvons les réseaux de Petri [WE98] et le *π-calculus* [MPW92]. Cependant, ces formalismes sont de bas niveau d'abstraction et peu intuitifs, plus adaptés à l'usage des mathématiciens. Néanmoins, certaines approches d'orchestration les

utilisant comme base de leurs propres formalismes, les réseaux de Petri ont inspiré Orchestration Nets [MPP02], YAWL [vdAH05] et [DLC+07], tandis que le *π-calculus* a inspiré le langage XLANG [Tha01].

Finalement, les hiérarchies d'activités permettant une vision structurée de la composition. Une spécification dans ce type de formalisme, dispose d'une activité de premier niveau qui se décline en un arbre des sous-activités. Chaque sous-activité peut être une feuille (activité simple) ou un autre arbre (activité composée). Les activités composées servent à définir de contraintes de contrôle flux appliquées à leurs sous-activités, pour indiquer par exemple, une exécution en séquence ou en parallèle. Les activités simples sont spécialisées pour la réalisation d'une tâche spécifique, par exemple l'invocation d'un service ou l'assignation d'une variable. WS-BPEL [OAS07b] utilise un formalisme basé sur une hiérarchie d'activités. Ce type de formalisme est moins intuitif et plus restrictif que le diagramme d'activités, puisqu'il a besoin d'utiliser des activités afin d'exprimer le flux de contrôle de la composition.

Modèle de données et d'accès aux données

Ce critère détermine la façon de manipuler les données pour communiquer avec les services, ainsi que les données associées à l'état de l'orchestration. Le modèle de typage de données utilisé par les approches d'orchestration sera aussi analysé.

L'approche d'orchestration utilise deux façons de manipuler les données afin d'interagir avec les services. La première approche est de considérer les données comme des boîtes noires (*black boxes*). Dans ce cas, l'orchestration n'a pas à connaître ni la structure ni le contenu des données qui sont échangées avec les services. Seulement des pointeurs sont passés d'une activité à l'autre. Par contre, les développeurs sont responsables de produire la logique de récupération de données, ainsi que la transformation des données en paramètres attendus par les services. L'avantage de cette approche est qu'elle évite des échanges de données complexes entre les activités, et dispense ainsi le moteur d'exécution de la manipulation de grands volumes de données.

Nous allons maintenant considérer comment les données servant à maintenir l'état de l'orchestration sont traitées. Deux approches sont utilisées, la première consiste à avoir un espace de mémoire dédié (*blackboard*) pour garder les valeurs des variables utilisées. La deuxième consiste à définir de façon explicite des flots de données entre les activités.

Dans l'approche de *blackboard*, chaque instance de l'orchestration possède son espace en mémoire où elle garde une copie des variables. Lors de l'invocation d'un service, les variables son accédées et passées comme paramètres de l'invocation, ensuite la réponse de l'opération affecte les valeurs des variables du *blackboard*. L'affectation des variables écrase leurs valeurs antérieures, donc le canevas est responsable de fournir une politique à appliquer dans le cas de modifications concurrentes de données. WS-BPEL et eFlow ont un système de traitement de données de type *blackboard*. La méthode *blackboard* a l'avantage d'être bien connue puisqu'elle est vastement utilisée par les langages de programmation conventionnels.

Pour sa part, l'utilisation de flots de données implique que l'orchestration doit indiquer explicitement les données qui seront transférées d'une activité à l'autre. Ainsi, les données d'entrée et de sortie d'une activité sont explicitement définies. APEL [EDA98] et JOpera [PHA06] usent la définition explicite de flots de données. JOpera sépare complètement le flux de données et de contrôle, tandis qu'APEL utilise l'abstraction de flot de données pour exprimer les deux. Une propriété du mécanisme des flots de données est qu'il permet d'avoir différentes versions de la même variable dans la composition, ce qui fournit une flexibilité pour sa définition, mais qui ajoute de la complexité à la composition.

Finalement, par rapport au modèle de typage de données, certains formalismes utilisent leur propre système de typage et des autres font recours à modèles répandus. Ainsi, JOpera [PHA06], eFlow [CIJ+00] et YAWL [vdAH05] utilisent un modèle propre. WS-BPEL et XPDL

utilisent la spécification XML-Schema, ce choix est naturel puisque la plupart des approches visent la composition de service Web.

Liaison de services

Ce critère vise à déterminer comment est réalisée la liaison entre l'orchestration est les services qu'elle compose. Une approche d'orchestration peut réaliser une liaison statique ou une liaison dynamique. Une liaison statique peut être réalisée au moment de la conception de l'orchestration ou au déploiement. A la conception, les services à utiliser sont exprimés dans la spécification de l'orchestration, tandis qu'au déploiement, généralement un fichier de configuration est utilisé pour les spécifier.

Une liaison dynamique ou à l'exécution peut être mise en place par le biais de différents mécanismes. En premier lieu, un mécanisme utilisant une variable pour stocker la référence du service à invoquer (*dynamic binding by reference*). Ce mécanisme n'indique pas comment la variable sera affectée. Une option consiste à déterminer les valeurs des références de services lors de la création d'une instance de l'orchestration. Par exemple, ces valeurs peuvent être affectées par un administrateur en utilisant une interface pour fournir pour ce but. Une autre option est d'utiliser le cadre d'interaction définie par l'approche à services, donc chercher dans un annuaire les références des services à utiliser. L'avantage de ce mécanisme est sa facilité d'utilisation et de mise en place. Par contre, le modèle de l'orchestration est pollué avec la définition des variables gardant les références des services, ainsi que pour les activités chargées d'affecter ces variables. Ce mécanisme est propose par la spécification de WS-BPEL [OAS07b].

Un autre mécanisme consiste à utiliser une requête exprimée dans un langage déterminé et basée sur les propriétés du service (prix, temps de réponse, etc.). Cette requête est associée aux références des services. Lors de l'exécution de l'activité d'invocation la requête est analysée par le *middleware* supportant l'exécution de l'orchestration afin de trouver les services pertinents (*dynamic binding by lookup*). Si lors de la requête, plusieurs services sont retournés par le *middleware,* une phase de sélection doit être accomplie afin d'en choisir un. SELF-SERV [BSD03], WSFL [Ley01] et eFlow [CIJ+00] utilisent ce mécanisme de liaison. SCENE [CDNM06], propose une extension de WS-BPEL incluant un langage de requêtes et un *runtime* afin de supporter ce type de liaison à l'exécution. En plus, ce canevas supporte la reconfiguration dynamique de la composition pour gérer des situations comme la disparition d'un service ou la détérioration de ses propriétés.

Il existe des travaux essayant de pourvoir des mécanismes de liaison et sélection de services pour les applications orientées services. Dans [EDSV09], la sélection de composants fournissant la fonctionnalité requise est une préoccupation gérée tout au long du cycle de vie d'une application à services, dès la conception jusqu'à l'exécution.

Gestion des exceptions

Ce critère détermine comment l'orchestration gère les situations d'exception lors de son exécution. Ces situations peuvent être conséquence des problèmes des fournisseurs des services, par exemple un serveur qui tombe en panne ou bien par une réponse indiquant une faille d'un service invoqué. Des actions associées à la logique de l'orchestration peuvent aussi produire des situations d'exception, par exemple l'annulation d'une commande dans une orchestration supportant un processus de vente de produits.

Afin de gérer ces situations d'exception, nous identifions trois moyens différents. D'abord, pour les approches qui n'ajoutent pas de constructions dans le formalisme de spécification pour gérer des exceptions. Une méthode basée sur le contrôle (*flow-based*) peut être utilisée, elle consiste à évaluer l'état de l'orchestration après l'invocation d'une opération et à ajouter des actions afin de gérer les situations d'exception. Dans APEL [EDA98], ce type de gestion est mis en place.

Une deuxième technique consiste à ajouter des constructions dans le langage afin de capturer, gérer et propager des exceptions (*try-catch-throw*). La logique de gestion des exceptions est ajoutée à une activité (ou à un groupe d'activités) et une condition booléenne est associée à un ensemble des variables, généralement le résultat de l'invocation de l'activité. Si la condition est évaluée à vrai, la logique de gestion est exécutée. Ensuite, l'orchestration peut soit continuer avec l'exécution de l'activité suivante, soit réessayer d'exécuter l'activité qui a échoué, soit terminer l'exécution du procédé. Cette technique a été adoptée par le langage WS-BPEL. Une technique *try-catch-throw* permet de séparer la logique des exceptions de la logique normale de l'orchestration, cette structuration facilite la conception et la maintenance de la spécification.

La troisième technique, consiste à gérer les exceptions en utilisant des règles ECA. Dans ce cas, des événements sont associés aux situations d'exception et la condition sert à vérifier si la situation doit être traitée, finalement l'action définit comment l'exception sera traitée. Les règles sont définies dans un langage autre que le langage de définition de l'orchestration. Cette technique permet aussi de séparer la logique normale de la composition de la logique de gestion des exceptions, par contre il n'existe de structuration de la spécification comme dans la technique *try-catch-thow*. Il existe aussi un problème associé au fait que le développeur doit maîtriser deux langages différents. Finalement, le passage à l'échelle est limité car une grande quantité de règles rend difficile de comprendre et traiter la spécification. Une variante intéressante est présentée par le système de *workflow* YAWL [AtHvdAE07]. En plus d'utiliser les règles pour détecter les situations d'exception, ce système sélectionne dynamiquement la logique (l'action) qui va gérer l'exception.

Orchestration de services et les aspects non-fonctionnels

Les modèles de composition fournissent des abstractions afin de faciliter la composition de services tout en cachant un certain nombre de détails techniques de bas niveau comme par exemple la communication entre la composition et les composants fournissant les services composés. Cependant, une composition de services comme toute autre application doit faire face à plus d'une préoccupation, notamment des aspects non-fonctionnels doivent être considérés comme par exemple, la sécurité, la gestion de transactions, etc.

L'alternative proposée à ce type de problématique dans les systèmes de *workflows* développés dans les années 90 était fournir toutes ces propriétés dans un seul système monolithique. Cette solution produisait des formalismes de définition de procédés complexes avec un grand nombre de concepts, et par conséquent difficiles à manipuler. En plus, ça donnait des implémentations de *middlewares* lourdes, difficiles à mettre en place, configurer, exploiter et maintenir. Ces *middlewares* sont très coûteux tant au niveau des licences, que du développement et de la maintenance. Un défi auquel se confrontaient les fournisseurs de ce type de solutions, était de déterminer les préoccupations qui seront adressées par leur produit. Chaque fournisseur proposait une solution généralement adaptée à un métier spécifique.

L'approche à services a permis de bien définir les interfaces entre les applications (les services) et le système contrôlant la composition (le *workflow*). Cependant, cette approche ne spécifie pas comment les aspects non-fonctionnels de services seront traités. Par exemple, dans le cas des services Web, prolifèrent un nombre important de standards cherchant à définir comment les services supportent les propriétés non-fonctionnelles. Parmi eux, nous pouvons citer *WS-Security* [OAS06] qui spécifie comment sécuriser le protocole SOAP et *WS-AtomicTransaction* [OAS07a] qui spécifie comment le *2PC* doit être utilisé par un groupe de services web participants dans une transaction. Pour sa part, les technologies de composition se focalisent dans la définition de la logique de composition laissant de côté le support des aspects non-fonctionnels.

Il existe deux façons alors de s'attaquer aux problèmes des aspects non-fonctionnels dans une composition de services. La première approche, consiste à inclure dans le formalisme de spécification de la composition des concepts servant à exprimer les aspects non-fonctionnels.

L'autre alternative est l'utilisation d'un middleware supportant l'aspect non-fonctionnel. Dans le premier groupe, nous trouvons des canevas comme WebTransact [PBM03] qui ajoute les concepts de traitement de transactions dans le langage de composition. Dans le second cas, nous pouvons citer par exemple des *middlewares* implémentant la spécification WS-*AtomicTransaction*. Par contre, ces *middlewares* exposent leurs services non-fonctionnels en tant que services Web, et la logique de la composition est modifiée pour ajouter l'invocation de ces services. Par conséquent, la logique de composition est mélangée avec la logique de traitement des aspects non-fonctionnels.

Une autre solution, consiste à utiliser des mécanismes similaires à ceux proposés par l'AOP [KLM+97]. Les différentes préoccupations sont alors spécifiées séparément de la logique de composition et ensuite, en utilisant une définition de *pointcuts*, ces aspects sont tissés pour produire une nouvelle composition. Dans ce groupe, nous trouvons des approches comme AOBPEL [CM04b] et [CF04]. Bien que résolvant les problèmes de modularité et de séparation des préoccupations, cette solution est soumise aux mêmes contraintes que la première, c'est-à-dire, soit le formalisme de composition fournit les concepts pour exprimer les aspects non-fonctionnels ou soit un *middleware* fournit le support en utilisant la même technologie à services que la composition.

Nous considérons qu'une troisième alternative consiste à permettre l'utilisation de différents formalismes pour exprimer les différentes préoccupations non-fonctionnelles. Une intégration des différentes préoccupations doit être réalisée afin de produire la spécification finale de la composition à exécuter. En conséquence, les machines d'exécution supportant chaque préoccupation doivent aussi être composées afin de créer une machine supportant la composition de services avec les aspects non-fonctionnels ciblés. Cette vision, permet d'abord de séparer les préoccupations et de modulariser la composition ; mais à la différence d'une approche du style AOP, il n'est pas obligatoire d'utiliser le même formalisme de composition pour exprimer tous les aspects non-fonctionnels. Une autre propriété importante de cette approche est son niveau de réutilisation. Etant donné que les préoccupations sont exprimées séparément, la spécification de la composition peut être réutilisée.

Extensibilité des modèles d'orchestration

Le support de différents types d'extension permet à un canevas ou modèle d'orchestration d'attaquer des préoccupations pour lesquelles il n'a pas été conçu au départ. Un nombre important de ces préoccupations est d'ordre non-fonctionnel, mais il existe aussi des préoccupations fonctionnelles pour lesquelles un canevas de composition peut être étendu. Par exemple, une propriété importante de systèmes de *workflows* est qu'ils peuvent utiliser différent types de ressources afin d'accomplir les tâches d'un procédé, notamment des humains. Les modèles d'orchestration font l'hypothèse simplificatrice que les tâches sont toujours exécutées par des services. Afin de supporter ce type d'assignation de ressources, des mécanismes d'extension doivent être mis en place dans le modèle de composition.

Les modèles d'orchestration étudiés offrent des mécanismes d'extension au niveau du langage de spécification. Ces extensions permettent l'introduction de nouveaux types d'activités spécialisées pour exécuter des tâches spécifiques. Dans ce type d'extension, nous trouvons BPEL4People [KKL+05] qui ajoute un type d'activité réalisée par des humains dans WS-BPEL. L'inconvénient avec ce type d'extensions est que le moteur d'exécution de la composition doit est capable de comprendre les nouveaux types d'activités, ce que implique la modification du *runtime*. Dans [CF04], une approche orientée aspects pour étendre un moteur d'exécution de WS-BPEL est proposée. D'autres extensions non invasives permettent d'ajouter des fonctionnalités sans modifier le langage de définition de la composition. Par exemple, SCENE [CDNM06] supporte la liaison à l'exécution et la reconfiguration dynamique d'une composition de services Web, sans ajouter de concepts dans le langage de composition.

Nous considérons qu'une alternative d'extension consiste à permettre l'utilisation de différents formalismes pour exprimer les différentes préoccupations (fonctionnelles et non-

fonctionnelles), et ensuite fournir des mécanismes permettant l'intégration des différentes préoccupations afin de produire la spécification finale de l'orchestration à exécuter. Conséquemment, les machines d'exécution supportant chaque préoccupation doivent aussi être composées afin de créer une machine supportant l'orchestration de services avec les nouvelles préoccupations ciblés.

4.2.2 WS-BPEL

WS-BPEL [OAS07b] est un standard visant la composition de services Web en utilisant un formalisme de définition de procédés issu du monde industriel. WS-BPEL est le résultat de la fusion de deux langages d'orchestration précurseurs, WSFL [Ley01] d'IBM et XLANG [Tha01] de Microsoft. Maintenant, le consortium OASIS est chargé de standardiser le modèle de composition WS-BPEL.

Afin de mieux comprendre le modèle WS-BPEL, dans cette section nous allons présenter d'abord un exemple de composition en utilisant WS-BPEL sans rentrer dans les détails et ensuite les différentes caractéristiques du formalisme seront décrites en utilisant l'exemple afin de les illustrer. La Figure 19 illustre l'exemple de procédé BPEL et décrivant un processus de traitement d'une commande. Le contrôle de la composition est réalisé comme suit : le client commence l'interaction en utilisant l'activité *Receive Order*, ensuite deux chemins peuvent être suivis en parallèle. Dans le premier chemin *Sequence1*, l'activité *Send Invoice* est chargée d'envoyer la facture et puis l'activité *Receive Payment* est responsable de recevoir le paiement. Dans le deuxième chemin *Sequence2*, l'activité *Ship Products*, envoi les produits au client. Une fois ces activités exécutes, les deux chemins se synchronisent et l'activité *Archive Order* est chargé d'archiver le dossier, finalement l'activité *Reply Order* confirme au client le succès de la transaction.

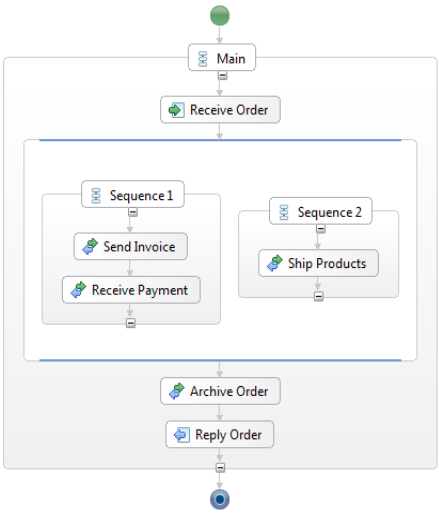

Figure 19. Exemple de procédé WS-BPEL.

Formalisme de description de l'orchestration

WS-BPEL dispose d'un formalisme héritant des caractéristiques de ses langages précurseurs, donc une vision très structurée proche d'un langage de programmation apportée par XLANG, mais aussi la possibilité de définir de flots de contrôle directement entre des activités comme celui défini par WSFL. Dans la pratique la vision structurée est la plus utilisée.

Le concept central du formalisme WS-BPEL est l'activité. Les activités sont typées et leur type détermine la sémantique d'exécution de l'activité. Les activités peuvent être basiques ou structurées. Les types d'activités basiques permettent l'interaction avec les services, comme *Invoke* (invocation d'une opération d'un service Web.), *Reply* (réponse d'une invocation de la part d'un client), et *Receive* (réception d'un message envoyé par un client). Des autres types

d'activités basiques sont *Assign* (assignation de données aux variables) et *Wait* (activité d'attente pour une période ou jusqu'une date). Les types d'activités structurées permettent l'expression du flot de contrôle du procédé. Dans ce groupe nous trouvons, *Sequence* (une exécution en séquence), *Flow* (exécution parallèle) et *While, ForEach, RepeatUntil* (équivalents aux boucles *while, for, repeat* dans un langage de programmation).

Nous pouvons situer le langage de composition du modèle WS-BPEL comme étant une hiérarchie d'activités. Cette caractéristique peut être constatée par notre exemple, ainsi l'activité structurée *Main* de type *Sequence* contient quatre sous-activités : *Receive Order*, *Flow1*, *Archive Order* et *Reply Order*. L'activité *Flow1* est une activité structurée de type *Flow*, elle possède deux sous-activités de type *Sequence*, donc *Sequence1* et *Sequence2*. L'activité *Sequence1* contient les sous-activités *Send Invoice* et *Receive Payment*, de sa part l'activité *Sequence2* contient la sous-activité *Ship Products*.

Liaison de services Web

La liaison entre le procédé WS-BPEL et les services Web qui vont offrir la fonctionnalité requise peut être effectuée statiquement (à la conception, au déploiement) ou bien à l'exécution du procédé. Si la liaison est effectuée à la conception, l'adresse (*endpoint*) du service concret est exprimée dans la définition du procédé. S'il s'agit d'une liaison au déploiement, un fichier de description de déploiement fournit cette information, par contre le format de ce fichier ne fait pas partie de la spécification WS-BPEL.

Lorsqu'il s'agit d'une liaison à l'exécution, le mécanisme d'assignation de références (*dynamic binding by reference*) existe dans le langage. L'utilisation de ce mécanisme entraîne la modification du modèle car il est nécessaire de définir des variables dans le modèle pour garder les références, ainsi que d'utiliser des activités (invocations) pour récupérer les références des services.

Modèle de données et d'accès aux données

Le langage WS-BPEL utilise une approche de *blackboard* afin de manipuler l'information de l'orchestration, le concept de variable permet comme dans les langages de programmation de définir des conteneurs pour garder l'information manipulée par l'orchestration. Les variables de WS-BPEL sont typées, mais le formalisme ne fournit pas un moyen de définir leurs types de données, en revanche il s'appuie sur le système de typage de *XML-Schema.*

Les variables ont une portée déterminée, les variables définies au premier niveau d'une spécification d'un procédé sont considérées comme des variables globales, les autres sont des variables locales. Les règles lexicales définissant la portée des variables sont les mêmes que celles des langages de programmation.

Il existe deux façons de modifier la valeur d'une variable dans WS-BPEL, tout d'abord les variables sont modifiées par l'invocation des opérations des services Web. Ensuite, la valeur d'une variable peut être modifiée en utilisant une activité de type *Assign*, ainsi une expression dans un langage de requêtes XML comme *XPath* est utilisée pour indiquer la modification.

Gestion des exceptions

WS-BPEL suit une approche *try-catch-throw* afin de gérer les exceptions. Chaque portée dans le langage peut définir des gestionnaires d'exceptions (*fault handlers*). Un gestionnaire détermine comment l'exception sera traitée, il indique l'exception à gérer ainsi que l'activité à exécuter en cas d'exception. WS-BPEL permet aussi de signaler une exception en utilisant une activité de type *throw*. Le mécanisme de gestion d'exception de WS-BPEL est semblable à celui employé par des langages de programmation comme Java.

Extensibilité

Etant donné que la spécification de WS-BPEL spécifie seulement le formalisme d'orchestration de services Web, les mécanismes d'extension proposés sont définis seulement au niveau du langage. Nous allons dans cette section, d'abord présenter le mécanisme d'extension proposé par le langage, et ensuite d'autres mécanismes proposés par certaines des implémentations de WS-BPEL.

Le mécanisme d'extension proposé par WS-BPEL est la création de nouveaux types d'activités. Une activité d'un nouveau type doit fournir les attributs obligatoires pour toutes les activités WS-BPEL, et en plus elle doit ajouter les attributs nécessaires pour l'exécution de l'activité (attributs XML). Si le moteur d'exécution a connaissance du type d'activité il pourra l'exécuter, autrement le moteur ignore l'activité (activité type *Empty)*.

Ce type extension offre uniquement un moyen de décrire syntaxiquement un nouveau type d'activité, par contre la sémantique d'exécution est la responsabilité de l'implémentation. Par conséquent, la sémantique d'exécution d'un nouveau type d'activité peut varier entre deux implémentations, ou bien elle n'est simplement pas supportée pour certaines implémentations. Ainsi, le principal inconvénient de ce type d'extension est le besoin de modifier l'infrastructure d'exécution afin d'inclure la sémantique des nouveaux types d'activités.

Quant aux extensions au niveau des implémentations, nous trouvons deux types d'extension : celles qui essayent de combiner le langage WS-BPEL avec d'autres langages, et celles qui ajoutent des propriétés dans l'infrastructure d'exécution afin de supporter des fonctionnalités non fournies dans le formalisme. Dans le premier groupe, sont classifiées des approches comme BPELJ [BGK+04] qui combine WS-BPEL avec le langage Java afin de supporter l'exécution du code Java à l'intérieur d'un procédé WS-BPEL. Pour sa part, [CM04a] propose un langage de règles combiné avec WS-BPEL afin d'avoir des orchestrations de services plus adaptables. Dans les deux cas, les moteurs d'exécution doivent être modifiés manuellement pour inclure la sémantique de l'extension.

Dans le second type d'extension, au niveau de l'implémentation, nous trouvons SCENE [CDNM06] qui permet d'inclure des caractéristiques comme la liaison des services à l'exécution d'une façon transparente pour l'orchestration. En plus, il est possible de changer les services choisis s'ils ne fournissent pas la qualité de service attendue. Ce type d'extension est techniquement réalisé avec le patron de conception *proxy* ; de cette façon la modification du moteur d'exécution n'est pas nécessaire, et les composants supportant la nouvelle fonctionnalité sont plus facilement intégrés.

Le langage WS-BPEL n'indique pas comment traiter les aspects non-fonctionnels de la composition, ainsi le support de *WS-Transaction* [OAS07a] et *WS-Security* [OAS06] est traité de manière différente pour chaque implémentation, ou non traité. Bien qu'une logique de séparation de préoccupations soit envisage par WS-BPEL (l'orchestration est spécifie indépendamment de ces aspects comme la sécurité), il est nécessaire d'inclure des mécanismes d'extension permettant ajouter ces aspects à l'orchestration et de laisser cette responsabilité comme une tâche de bas niveau réalisée par les développeurs.

4.2.3 JOpera

Malgré le niveau d'acceptation du modèle de composition des services WS-BPEL, plusieurs problèmes ont été identifies. Au niveau recherche deux directions ont été adoptées : d'abord ceux qui proposent des extensions (voire de modifications) de WS-BPEL, et ceux proposant de nouvelles approches pour l'orchestration de services. JOpera est un projet de l'université de Lugano visant à fournir un modèle de composition de services et des outils de spécification, d'exécution et monitoring d'orchestration. Nous allons catégoriser ce canevas par rapport aux critères définis auparavant.

Technologie des éléments composés

JOpera est un système ouvert par rapport aux technologies de services supportées. Par défaut, il fournit un support pour la composition de services Web, de méthodes Java et d'applications Unix. Cependant, JOpera offre des mécanismes d'extension afin de supporter la composition de nouveaux types de services [PA04].

Formalisme de description de l'orchestration

JOpera utilise le langage JVCL [PA05] (acronyme en anglais de *JOpera Visual Composition Language*) comme formalisme de description de l'orchestration de services. JVCL est un langage visuel du type diagramme d'activités (graphe orienté). Dans JVCL, un procédé est formé par un ensemble de tâches, de flots de contrôle et de flots de données entre les tâches. Une tâche (*Task*) peut être, soit une activité (*Activity*) qui représente l'invocation d'un service, soit un sous-procédé (*Subprocess*) exprimant l'invocation d'un autre procédé. Donc, le langage est organisé hiérarchiquement.

Les tâches ne sont pas typées dans JVCL. Une importante caractéristique du langage est qu'il permet de prédéfinir des tâches avec une sémantique associée ; ces tâches prédéfinies peuvent être utilisées dans la définition d'un nouveau procédé et adaptées à leur contexte d'utilisation. Cette propriété permet de créer une bibliothèque de tâches augmentant la réutilisation.

Modèle de données et d'accès aux données

JOpera ne fait pas différence entre les données pour maintenir l'état de l'orchestration et celles utilisées pour interagir avec les services. Les données sont traites avec un modèle de flux de données (*dataflow*). Donc, chaque tâche doit spécifier explicitement ses variables en entrée et ses variables en sortie. En plus, un flot de données est créé pour chaque pair de variables, ainsi une variable de sortie d'une tâche est liée à une variable d'entrée d'autre tâche par un flot de données.

Finalement, le système de typage de données est propre à JOpera, donc les types des données sont exprimés à l'intérieur d'un modèle d'orchestration.

Liaison de services

La liaison de services en JOpera est réalisée à la conception ou à l'exécution du procédé. Un registre de services est embarqué dans l'outil de spécification de l'orchestration, permettant de trouver les services que matchent syntaxiquement à la conception.

JOpera utilise un mécanisme de liaison dynamique de services, que nous considérons comme une combinaison du mécanisme de liaison dynamique par référence (*dynamic binding by reference)* et du mécanisme de recherche basée sur des expressions (*dynamic binding by lookup).* Donc, il est possible de donner des expressions pour indiquer les caractéristiques du service requis, mais une variable pour garder la référence du service ainsi qu'une tâche pour invoquer le registre sont nécessaires dans la spécification du procédé. En fait, il existe une activité prédéfinie dans une bibliothèque fournie par JOpera qui est utilisée pour récupérer des services du registre. Le résultat de l'utilisation du mécanisme, est qu'une le modèle de procédé est pollué par la logique de liaison de services.

Gestion des exceptions

Dans JOpera, le modèle de services exprime pour chaque service les paramètres d'entrée, les paramètres de sortie et un mécanisme de détection de failles. Ainsi, lorsqu'une tâche détecte une faille comme résultat de l'invocation d'un service, une autre tâche peut être associée pour gérer la situation d'exception (logique *try-catch* par tâche).

D'autre part, le langage ne propose pas de constructions pour gérer les exceptions associées à une portée déterminé (un groupe de tâches), ni pour signaler l'occurrence d'une situation d'exception dans la logique d'exécution du procédé.

Extensibilité

L'extensibilité dans JOpera est définie à deux niveaux. D'abord il est possible d'étendre le formalisme de composition en ajoutant des tâches prédéfinies. Ces tâches indiquent non seulement leur structure, mais aussi leur sémantique, ce qui évite que le moteur d'exécution soit modifié pour comprendre la sémantique. En fait, les activités prédéfinies peuvent être considères comme des pseudos types. Ainsi, l'outil de spécification connaît l'existence du type d'activité, mais pour le moteur d'exécution, ces types n'existent pas, donc il exécute des tâches normales.

Le second type d'extension abordée par JOpera est la capacité d'ajouter le support pour de nouvelles technologies à services. Ainsi, au niveau de la composition, les tâches sont définies avec un ensemble de paramètres d'entre et de sortie. Le mécanisme permet d'indiquer comment ces paramètres seront utilisés par le service d'une technologie spécifique.

Malgré les deux mécanismes d'extension proposés par JOpera, le canevas ne fournit pas (au moins explicitement) de mécanismes d'extension afin de supporter des propriétés non fonctionnelles comme la sécurité, la gestion de transactions, etc.

4.2.4 Synthèse

Dans l'orchestration de services, deux technologies convergent ; la technologie de *workflow* et l'approche à services. La technologie de *workflow* apportant des formalismes d'expression de procédés ainsi que des outils pour supporter la spécification, l'analyse, la validation et l'exécution de modèles de procédés. Pour sa part, l'approche à services (notamment les services Web) apporte les mécanismes pour réussir un faible couplage entre les services (applications) et leurs clients.

De ce fait, plusieurs modèles, langages, standards et canevas pour l'orchestration de services ont été proposés. Nous avons dans cette section, fait une catégorisation de différentes approches en utilisant un ensemble de critères pour les comparer. Ainsi, au fur et à mesure que les critères ont été dévoilés, les différents canevas (ou langages) d'orchestration ont été placés selon leurs propriétés vis-à-vis de ces critères.

Nous avons alors insiste sur le langage WS-BPEL [OAS07b], car il devient le standard de fait dans cette technologie, même s'il s'intéresse qu'à l'orchestration de services Web. Pareillement, nous avons analysé le canevas JOpera qui offre des caractéristiques intéressantes, et qui vient du monde académique. Dû au manque d'espace, nous n'avons pas détaillé l'approche SELF-SERV [SBDM02], mais nous considérons qu'elle offre une autre vision sur l'orchestration de services. Un tableau comparatif des trois approches est présenté dans la Figure 20.

De cette analyse, nous concluons que même si l'approche à services a résolu le problème du couplage et d l'interopérabilité (au niveau de protocoles) du système avec les éléments composés, les approches actuelles d'orchestration de services possèdent encore de problèmes hérités de la technologie de *workflow* comme :

- des formalismes complexes, de bas niveaux d'abstraction proches de langages de programmation ;
- support pour un seule type de technologie à services, généralement de services Web ;
- un support pauvre ou inexistant pour des aspects non-fonctionnels des applications.

Dans le cadre de cette thèse, nous allons proposer des solutions aux différents problèmes identifiés de l'approche d'orchestration de services.

	WS-BPEL	JOpera	SELF-SERV
Technologie des éléments composés	Services Web	Ensemble prédéfini + possibilité d'ajouter	Services Web
Formalisme de description de l'orchestration	Hiérarchie d'activités	Diagramme d'activités (Graphe dirigée)	Diagramme d'états et transitions
Liaison de services	Statique et dynamique (*dynamic binding by reference*)	Statique et dynamique (combinaison *dynamic binding by reference* et *dynamic binding by lookup*)	Dynamique (*dynamic binding by lookup*)
Modèle de données et d'accès aux données	*Blackboard* – Système de typage externe (XML-Schema)	*Dataflow* – Système de typage interne	*Blackboard* – Système de typage externe
Gestion des exceptions	Approche *try-catch-throw*	Approche *flow-based*	Non spécifiée
Support des aspects non-fonctionnels	Transactions partiellement supportées avec le concept de *Compensation Handler*	Aucun aspect non-fonctionnel traité.	Exécution répartie. L'aspect n'est pas explicitement présenté au développeur
Extensibilité	Nouveaux types d'activités définies au niveau syntaxiquement	Bibliothèque d'activités prédéfinies. Ajoute de technologies à services	Aucun mécanisme d'extension défini

Figure 20. Tableau comparatif des approches d'orchestration de services.

4.3 CHOREGRAPHIE DE SERVICES

La chorégraphie de services, contrairement à l'orchestration, ne spécifie pas la composition de services d'un point de vue central. Elle essaye plutôt de définir comment un ensemble de partenaires vont faire collaborer leurs procédés afin d'arriver à un but commun. Les procédés de chaque partenaire sont exposés en tant que services. La collaboration entre les différents partenaires est réalisée par l'envoi et la réception de messages. Une hypothèse forte de la chorégraphie de services, est que les participants se sont mis d'accord sur le protocole de collaboration avant de commencer les interactions. Une fois ce protocole établi, chaque participant se comporte de façon autonome tout en respectant le protocole.

Il existe des propositions essayant de définir des modèles de collaboration (chorégraphies) entre participants dans des domaines spécifiques. Nous pouvons citer par exemple RosettaNet [Ros09], qui spécifie des collaborations dans le domaine de chaînes d'approvisionnement (*supply chain*), et SWIFTNet [SWI09] pour le domaine des services financiers. Cependant, ces propositions ne sont pas assez flexibles pour définir des modèles de collaboration dans d'autres domaines d'application.

En conséquence, différentes approches ont été mises en place avec comme objectif de modéliser ces collaborations entre les procédés des différents partenaires offrant des services. C'est surtout dans la technologie des services Web que s'est développé cet ensemble d'approches. Ceci n'est pas surprenant si nous considérons que c'est la technologie la plus utilisée dans l'implantation des interactions du type B2B, où l'établissement de modèles de collaboration est prioritaire afin d'assurer la comptabilité des interactions entre les procédés des organisations.

4.3.1 Processus de définition de chorégraphies

Le processus de définition d'une chorégraphie de services est schématisé dans la Figure 21. Ce processus est divisé en deux phases : la conception de la chorégraphie et l'implémentation. Dans la phase de conception, des analystes du métier sont chargés de déterminer les participants de la chorégraphie (partenaires), la portée du domaine adressé, les buts partiels à satisfaire (*millestone*), et en général de déterminer les interactions. Le résultat de cette phase est la définition des messages et du protocole de collaboration. Cette définition est décrite par des architectes du système et exprimée dans des langages pour les formaliser. Les langages utilisés pour décrire ces collaborations dans le domaine technique des services Web sont WSCI [W3C02a], WSCDL [W3C05].

Le langage WSCI a été créé en ayant pour objectif la spécification de l'interface dynamique d'un service Web. Une description statique comme celle offerte par un langage comme WSDL permet la spécification d'un ensemble d'opérations et les types de message compris par les opérations. L'interface dynamique exprime les dépendances logiques et temporelles entre les messages échangés avec le service Web. Elle indique aux utilisateurs le flux correct des messages à implémenter lors de la consommation du service Web. Ce type de langage peut être utilisé comme une spécification au moment d'implémenter un client qui souhaite consommer un service, mais, il n'exprime pas la collaboration entre un ensemble de partenaires.

En revanche, le langage WSCDL permet de décrire la collaboration entre un ensemble de services. Le langage décrit depuis un point de vue global le comportement observable, commun et complémentaire des participants d'une interaction, où l'échange ordonné des messages emmène les participants à un but commun.

La phase de développement requiert qu'en plus de la définition de la collaboration, le comportement visible de chacun des participants soit spécifié. Le langage WS-BPEL est utilisé alors pour décrire ce comportement visible depuis l'extérieur (ce qui ne fait pas partie de l'implémentation privée du procédé). Ce type de procédé est connu dans WS-BPEL comme un procédé abstrait. Ensuite, à partir du procédé abstrait de chacun des participants, les développeurs implémentent le procédé pour chacun des partenaires ; ce procédé est alors conforme à la spécification de la collaboration.

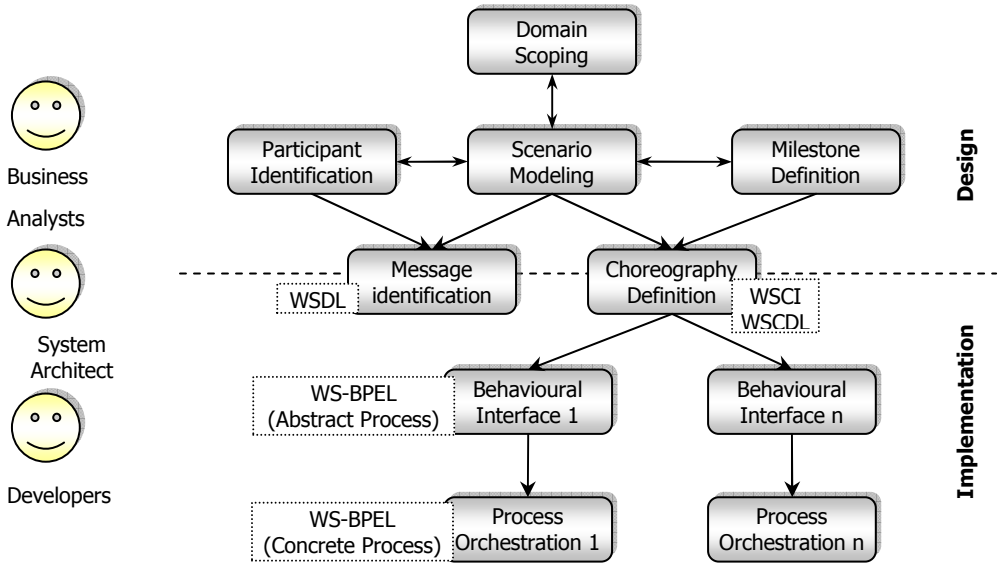

Figure 21. Processus de définition d'une chorégraphie.

4.3.2 Défis et opportunités de la chorégraphie de services

Bien que la chorégraphie de services (ou de procédés) permette d'assurer la compatibilité au niveau des types de messages et des interactions entre différents participants, cette technologie fait des hypothèses qui limitent son usage, à savoir :

- l'approche est totalement *top-down* ce qui implique modifier les procédés qui sont déjà mis en place par les participants ;
- les modèles de chorégraphie ne sont pas exécutables, ils fournissent uniquement une spécification du comportement visible de chaque participant ;
- chaque participant dans une chorégraphie appartient à différentes organisations, ce qui implique que seulement la partie visible des procédés peut être exposée par les participants.

Cependant, nous considérons que cette technologie fournit un ensemble de possibilités parmi lesquelles :

- l'amélioration de la performance des systèmes d'orchestration de services. Nous pouvons utiliser cette technologie pour décrire la collaboration d'un ensemble de procédés et ensuite déterminer un modèle de déploiement afin de placer les procédés dans les nœuds les plus proches des services composés afin de réduire les communications non nécessaires ;
- la création d'architectures d'exécution reparties pour les systèmes d'orchestration avec une vision pair-a-pair. La chorégraphie peut être utilisée pour décrire la collaboration entre les pairs.

Pour pouvoir profiter de ces nouvelles opportunités de la chorégraphie, il est nécessaire de disposer de formalismes de haut niveau d'abstraction et indépendants de technologie. Il faut aussi pouvoir assurer les aspects non-fonctionnels des compositions de la même façon que pour la technologie d'orchestration ; il faut donc des mécanismes supportant la mise en place de ces aspects.

4.4 SYNTHESE

De l'analyse réalise sur les approches d'orchestration de services, nous avons identifié un certain nombre de problèmes qui ont été hérités de la technologie de *workflow*. Bien que ces problèmes persistent, nous considérons que l'orchestration de services peut être utilisée comme base d'une technologie de construction d'applications basée sur le concept de procédé. Cependant, le problème principal de l'orchestration de services est le manque de mécanismes d'extension des canevas, ce qui empêche leur utilisation dans différents domaines d'application. Ces mécanismes peuvent être utilisés afin d'adapter les canevas à différents contextes, ainsi que pour inclure le support des aspects non-fonctionnels manquants.

Dans cette thèse nous nous intéressons à la construction d'un canevas pour la construction et l'exécution d'applications orientées procédés. Ce canevas prend l'orchestration de services comme point de départ, néanmoins les problèmes énoncés auparavant sont attaqués de diverses manières. Nous nous sommes spécialement intéressé dans l'ajout des mécanismes d'extension afin d'adapter le canevas à différents contextes d'utilisation ainsi que l'inclusion des aspects non-fonctionnels. Nous allons dévoiler ces mécanismes dans les chapitres dédiés à notre proposition et son implémentation.

Nous avons aussi identifié certaines propriétés de l'approche de chorégraphie de services, notamment la possibilité d'une exécution décentralisée. Nous voudrons récupérer cette architecture d'exécution, afin d'améliorer la performance dans l'exécution d'une orchestration de services. Dans le chapitre dédié à l'exécution repartie de l'orchestration nous présentons notre modèle d'exécution repartie. Cependant, il est important de clarifier, que l'établissement

d’un protocole de coordination des interactions entre services n’est pas le but de notre étude, donc la chorégraphie de services a été étudie avec le regard de l’exécution repartie.

5. L'INGENIERIE DIRIGEE PAR LES MODELES

Parmi les approches du génie logiciel, l'ingénierie dirigée par les modèles (IDM) propose un cadre afin de résoudre le problème de la complexité croissante du logiciel. Le principal objectif de l'IDM est de réduire la distance existante entre le domaine du problème adressé et le domaine technologique utilisé pour l'implémentation de la solution. L'IDM propose alors l'utilisation de techniques supportant la transformation systématique de spécifications décrites avec des termes proches du problème en des implémentations logicielles. Le problème est abordé en utilisant divers modèles de différents niveaux d'abstraction (niveaux de détail) ainsi que différents points de vue pour décrire un système complexe, ensuite ces modèles sont traités avec des outils qui réalisent la transformation et l'analyse de ces modèles.

L'idée de modélisation est présente depuis longtemps en génie logiciel par contre, jusqu'à nos jours, les modèles étaient des éléments servant à la compréhension des systèmes logiciels construits et à la communication entre les différents acteurs participant à leur construction. Ces modèles étaient utilisés dans les premières phases de construction des applications, c'est-à-dire la spécification de besoins et la conception. La phase de développement des applications était réalisée comme une tâche de transformation de ces modèles dans le code de l'application. Cette tâche permettant de passer du domaine du problème au domaine de la solution était jusqu'à présent assuré par l'expérience des ingénieurs dans le domaine d'application et dans la technologie d'implémentation, cette expérience est généralement exprimée en terme de patrons de développement.

De son côté, l'IDM préconise l'utilisation des modèles comme artéfacts de premier ordre dans la construction des applications. Généralement, les modèles appartiennent au domaine spécifique de l'application, ce qui permet la construction d'environnements de développement spécialisés essayant de capturer les différents patrons issus de l'expérience des ingénieurs dans le domaine précis de l'application. Une vision plus ambitieuse consiste à utiliser non seulement les modèles dans la phase de construction de l'application, mais aussi, de les utiliser dans la phase d'exécution du logiciel, celle-ci est la vision de l'IDM à laquelle nous allons adhérer dans cette thèse.

Dans [FR07], les auteurs ont identifié trois grands défis de l'IDM. D'abord l'utilisation et la définition des formalismes (langages) permettant de décrire les modèles formant une application. Ensuite, comment la séparation des préoccupations est adressée et comment les différents modèles spécifiant chacune des préoccupations sont composés d'une façon cohérente afin de construire la solution. Finalement, le traitement automatique de ces modèles qui consiste en leur transformation, leur versionnement et leur manipulation en général.

Dans ce chapitre, nous allons présenter d'abord les concepts de base de l'approche IDM, ensuite, une description des trois principaux défis de l'IDM et finalement, les travaux réalisés visant à faire face aux deux premiers défis dont les notions de langage et de méta-modélisation ainsi que la composition des modèles.

5.1 LES CONCEPTS DE BASE

5.1.1 Le concept de modèle

Il n'existe pas de consensus sur le concept de modèle, nous allons présenter quelques définitions que nous avons trouvées dans la littérature. Ces définitions sont indépendantes de la nature des modèles, donc elles ne sont pas limitées au cadre de l'IDM.

Pour [BG01], un modèle est défini comme :

« A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system. »

La définition met l'accent sur la capacité simplificatrice d'un modèle, l'intérêt est de réduire la complexité associée à la manipulation du système. En effet, seulement un ensemble de caractéristiques jugées utiles pour l'usage du modèle sont prises en compte. Donc, le modèle peut être utilisé à la place du système lorsqu'il s'agit d'étudier ses caractéristiques.

La définition adoptée par France et Rumpe dans [FR07] dit :

« A model is an abstraction of some aspect of a system. The system described by a model may or may not exist at the time the model is created. Models are created to serve particular purposes…»

Cette définition indique que, non seulement le modèle est plus simple que le système modélisé, mais en plus, le modèle peut faire référence juste à une partie du système. Elle est en accord avec le fait qu'un modèle est créé avec un objectif spécifique. Une propriété importante d'un modèle est qu'il peut modéliser un système inexistant. Ainsi, le modèle peut être utilisé comme une spécification dans la construction du système ; d'ailleurs cette utilisation des modèles est la plus courante en génie logiciel.

En dépit des différences, ces définitions adhèrent aux trois critères proposés par Ludewing [Lud03] pour caractériser un modèle dans le contexte du génie logiciel :

- critère de représentation : il existe un objet d'étude qui est représenté par le modèle. Dans la définition ci-dessus, cet objet est nommé système.
- critère de réduction : les propriétés de l'objet d'étude ne sont pas toutes représentées dans le modèle, cependant, le modèle doit refléter certaines propriétés de l'objet d'étude.
- critère de pragmatisme : le modèle peut remplacer l'objet d'étude pour un usage donné.

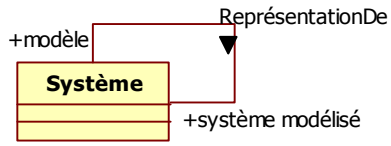

Figure 22. Modèle et Représentation.

Nous pouvons considérer qu'un modèle peut, à un moment donné, jouer le rôle de l'objet original. Donc, il est possible d'avoir un autre modèle représentant cet objet. Par exemple, un document de spécification d'un logiciel est un modèle du code à écrire, et le code lui-même, est un modèle des actions exécutées par un ordinateur. Donc, les notions de système original et de modèle sont relatives et jouent des rôles complémentaires basés sur la relation

« représentation de » qui lie le modèle au système modélisé, ces notions sont synthétisées dans le diagramme de classes extrait de [Fav04b], présentée dans la Figure 22.

Etant donné que la nature des modèles n'est pas précisée, une caractérisation des modèles sera développée par la suite.

5.1.2 La classification des modèles

Afin de comprendre le rôle joué par les modèles dans le cadre de l'IDM, il est nécessaire de clarifier les caractéristiques des modèles utilisés dans le cycle de vie du logiciel. Etant donné qu'un modèle est lui-même un système, la classification faite par la suite est valable pour les modèles ainsi que pour les systèmes.

Nature des modèles : physique, abstrait ou numérique

Favre [Fav04b] fait la distinction entre système physique, système abstrait et système numérique. Les modèles, auxquels s'intéresse l'IDM sont les modèles numériques pouvant être traités et exploités automatiquement.

Le fait d'automatiser le traitement des modèles impose de les exprimer formellement en utilisant des formalismes bien définis. D'ailleurs, certains auteurs ajoutent dans la définition même de modèle cette caractéristique. Dans [KWB03], les auteurs ont défini la notion de modèle comme :

« A model is a description of (part of) a system written in a well defined language. »

Nous reviendrons dans la section suivante sur la notion de langage de modélisation et la démarche employée pour sa définition, donc la notion de méta-modélisation.

Evolution des modèles : dynamique, statique

Bézivin [Béz05] fait la distinction entre système statique et dynamique, en fonction de leur capacité de changer dans le temps. Selon cette classification, il remarque l'usage répandu en informatique de modèles statiques pour représenter des systèmes dynamiques. Le modèle en soi ne change pas[1], mais il modélise l'évolution du système dans le temps.

Nous pouvons considérer alors la notion d'exécution ; un système dynamique peut être représenté par l'exécution d'un programme où l'exécution représente le comportement réel du système modélisé. Le code source du programme est une représentation statique de la dynamique d'exécution [Veg05].

Nature de la relation de représentation : descriptive ou spécification

Dans [Lud03], une précision est faite sur la nature de la relation de représentation existant entre le modèle et le système. Donc, elle distingue entre un modèle descriptif et un modèle normatif : le premier décrit le système modélisé, tandis que le second spécifie les caractéristiques du système modélisé.

Généralement, un modèle descriptif est créé à partir d'un système existant, et un nouveau système est créé à partir d'une spécification, donc d'un modèle normatif. Cependant, il est possible de réaliser des modèles descriptifs de systèmes inexistants, par exemple, en climatologie des modèles descriptifs de l'état du temps du lendemain sont créés, donc, le modèle décrit un système qui n'existe pas (encore).

Mais, cette distinction entre modèles descriptif et normatif est une propriété intrinsèque à la relation de représentation et non au modèle lui-même. Par exemple, si l'on veut copier un système existant, un modèle descriptif peut décrire le système original et à la fois servir de spécification pour la construction du système réplique.

[1] Au moins dans la même échelle de temps que le système

En plus, dans [Sei03], cette distinction est utilisée pour définir les notions de correction et de validité. Un modèle descriptif est dit correct si les réponses obtenues par le modèle sont effectivement constatées dans le système. Un système modélisé est dit valide, si les propriétés définies par la spécification sont satisfaites par le système.

L'utilisation du modèle : contemplatif ou productif

L'objectif des modèles contemplatifs est de servir de moyen de communication et de compréhension entre les différents acteurs humains participant à la construction d'un système logiciel. Les modèles contemplatifs étaient utilisés dans les méthodes de modélisation antérieures à l'IDM comme Booch [Boo93], OMT [Rum91] et Merise [RM89].

L'IDM préconise l'utilisation des modèles comme éléments de premier ordre dans la tâche de construction des applications. Ces modèles de différents niveaux d'abstraction sont créés et transformés afin de produire l'implémentation d'un système logiciel. Une vision plus ambitieuse est décrite par France dans [FR07], les modèles ne sont pas seulement utilisés pour la construction de l'application mais aussi, à l'exécution, pour supporter leur adaptation dynamique. Ces types de modèles utilisés tout au long du cycle de vie du logiciel sont considérés comme des modèles productifs.

5.2 DEFIS DE L'INGENIERIE DIRIGEE PAR LES MODELES

5.2.1 Au niveau des langages de modélisation

Des langages de modélisation sont utilisés afin d'exprimer des modèles servant à représenter un système. L'utilisation de ces langages pose un défi à l'IDM, car un soutien doit être fourni pour créer et utiliser des abstractions du niveau du problème dans ces langages ainsi que pour permettre l'analyse rigoureuse des modèles créés en utilisant ces abstractions. Ce défi peut alors être découpé en deux, le premier concernant le niveau d'abstraction des langages et le deuxième, le niveau de formalisation nécessaire, ils seront détaillés par la suite.

Le niveau d'abstraction

Dans ce premier point, la difficulté consiste à définir un langage qui permette à la fois de définir des abstractions du niveau du problème et d'exprimer la solution. Un des objectifs de l'utilisation des modèles est de réduire la complexité associée à la construction des applications informatiques. Pour atteindre ce but, le principal mécanisme utilisé par l'IDM consiste à élever le niveau d'abstraction des modèles utilisés dans la construction des applications. Par exemple, il existe des approches essayant de classifier les modèles informatiques selon le degré d'abstraction vis-à-vis des détails de la plateforme technologique utilisée lors de son implémentation [OMG03b].

Formalisation du langage de modélisation

Le second point vise à identifier quels sont les éléments d'un langage qui doivent être formalisés pour permettre des traitements automatiques. En plus, il est nécessaire d'identifier quels sont les mécanismes à utiliser pour aboutir à la formalisation de ces langages.

Par rapport aux mécanismes utilisés pour la formalisation de ces modèles, nous pouvons considérer les différents types de modèles utilisés en informatique ainsi que l'espace technique dans lequel ces modèles ont été créés. La notion d'espace technique est définie dans [KBA02] comme suit :

« a working context with a set of associated concepts, body of knowledge, tools, required skills, and various other possibilities. »

Dans [FEBF06], quatre espaces techniques sont cités, chacun correspond à un domaine de recherche particulier dans l'informatique, et chacun utilise son propre mécanisme de

formalisation de leurs modèles, mais tous partagent une architecture commune qui sera présentée dans la section dédiée à la méta-modélisation. Ainsi, nous pouvons trouver l'espace de modèles (*Modelware*) défini par des familles des méta-modèles (MOF, Ecore), l'espace des grammaire (*Grammarware*) défini par des langages de définition de grammaires (BNF, EBNF), l'espace des documents (*Docware*) défini par des schémas de documents (XML Schema, DTD), et l'espace des bases de données (*DBware*) défini par des formalismes de spécification de schémas comme les algèbres relationnelles.

En outre, il est important de distinguer la notion de précision d'un modèle de celle de formalisation. Lee [Lee00] définit la précision d'un modèle comme une mesure du degré de granularité de son abstraction. La précision peut être réduite en éliminant des détails non essentiels ou en faisant recours à des descriptions qualitatives et non quantitatives. Il faut remarquer que le degré de précision d'un modèle est indépendant de sa formalité et de sa justesse. Un modèle informel et vague peut très bien être une description juste d'un système pour une tâche donnée ; et un modèle formel peut être entièrement faux.

La section 5.3 montre les mécanismes de formalisation utilisés dans l'espace technique des modèles qui est celui qui nous intéresse dans le cadre de cette thèse, sans oublier que les autres espaces techniques (domaines de recherche en informatique) ont beaucoup apporté dans ce développement.

5.2.2 L'IDM et la séparation de préoccupations

En plus de l'abstraction, un autre mécanisme utilisé par l'IDM avec le but d'attaquer la complexité des applications est la séparation des préoccupations. Des approches comme la programmation par aspects [KLM+97] essaient de réaliser une séparation et puis une synthèse des différentes préoccupations logicielles qui composent une application. Néanmoins, ces approches sont basées sur le code de l'application, nous pouvons citer dans ce courant, AspectJ [KHH+01] et HyperJ [OT01] qui sont des extensions du langage de programmation Java ajoutant le concept d'aspect.

Dans l'IDM, certaines approches comme le MDA proposent de séparer les préoccupations métiers des préoccupations technologiques (voir Section 5.3.3). D'un point de vue plus général, chaque aspect ou point de vue particulier d'une application est pris en compte par un modèle particulier [FEB06]. Le fait que la notion de modèle soit liée à la notion d'abstraction fait que la nature exacte de la relation de représentation est relative à un point de vue et à un objectif donné; ce qui correspond aux critères énoncés auparavant de réduction et de pragmatisme. Il est donc possible de produire, et de traiter, à la fois plusieurs modèles d'un même système, en fonction des diverses perspectives et des différents acteurs impliqués. En conséquence, la séparation des préoccupations est une caractéristique intrinsèque de l'ingénierie dirigée par les modèles.

Dans le cas des modèles descriptifs, ces différents aspects sont liés dans le système modélisé, et la combinaison des différentes facettes est toujours envisageable [Bez05]. Dans le cas des modèles de spécification, il faut assurer la cohérence de tous ces modèles, et la construction du système modélisé revient à effectuer la composition et le tissage de tous les aspects [JGMB05].

Les défis consistent alors à résoudre les problèmes associés à la modélisation des systèmes par différents acteurs en utilisant de multiples points de vue qui, potentiellement, se chevauchent. En plus, les modèles utilisés dans les différents points de vue peuvent utiliser différents langages de modélisation. Finalement, la composition et le tissage de ces modèles afin de produire l'application correspondent à un problème courant. Dans la section 5.4, la séparation des préoccupations sera traitée plus en détail, ainsi que les différentes approches essayant de réaliser la composition et tissage des divers modèles représentant les divers points de vue.

5.2.3 Manipulation et traitement des modèles

Le troisième grand défi de l'IDM, après l'abstraction et la séparation des préoccupations, est de gérer les artefacts (la représentation des modèles et méta-modèles). Nous pouvons lister des problèmes :

- la définition, l'analyse et l'utilisation des transformations de modèles,
- la maintenance de traces de ces transformations,
- la maintenance de la cohérence entre les différents points de vue,
- la composition et fusion des modèles,
- le suivi des versions des artefacts comme modèles et méta-modèles,
- l'utilisation des modèles à l'exécution des applications.

Diverses plates-formes de gestion de modèles ont été proposées pour essayer de résoudre certains des problèmes listés ci-dessus, parmi celles-ci nous pouvons citer AMMA [FJ05], RONDO [MRB03], Epsilon [KPP06a] et MOMENT [BCR05].

Ce dernier défi ne sera pas traité plus en détail parce qu'il n'est pas abordé dans notre travail. Par contre, les sections suivantes correspondent aux efforts réalisés autour des deux premiers défis décrits.

5.3 LA NOTION DE LANGAGE ET DE META-MODELE

Dans [Kle08], un nouveau domaine d'étude de l'informatique est dévoilé, celui de l'ingénierie des langages logiciels. L'expérience dans divers domaines de recherche en informatique et, leurs espaces techniques associés a beaucoup apporté dans ce nouveau domaine de définition de langages informatiques. Par exemple, le domaine des langages de programmation (espace technique de *Grammarware*) a développé des mécanismes pour la spécification de la grammaire des langages (syntaxe abstraite, syntaxe concrète et sémantique), des méta-formalismes comme BNF et EBNF sont utilisés dans cette démarche. Le domaine des compilateurs, pour sa part, apporte l'expérience dans la transformation des modèles (programmes) écrits dans un langage particulier vers des modèles écrits dans un autre langage, généralement d'un niveau d'abstraction plus bas.

Comme nous l'avons introduit, l'IDM s'intéresse au traitement automatisé de modèles informatiques avec l'objectif de les utiliser comme des modèles productifs tout au long du cycle de vie d'un système logiciel. Afin de pouvoir automatiser le traitement de ces modèles, ils doivent être exprimé formellement, c'est-à-dire, écrits dans un langage clairement et formellement défini. Nous introduisons par la suite la notion de langage de modélisation, et nous nous intéressons particulièrement à la démarche employée lors de la définition de ces langages, dont celle de la méta-modélisation.

5.3.1 Langages de modélisation

Le concept de langage a toujours été présent en informatique, les langages sont utilisés d'une façon ou d'une autre pour la spécification, la modélisation et la création de logiciel. Dans ce courant, nous pouvons trouver des langages de programmation, des langages dédiés à un domaine spécifique DSL (*Domain Specific Language*), des langages de modélisation DSML (*Domain Specific Modeling Language*).

Partant du domaine de recherche des langages de programmation, un langage est défini par un ensemble de règles servant à indiquer la structure du langage (sa syntaxe), et son sens (sa sémantique). Cet ensemble de règles est organisé en trois éléments basiques : la syntaxe abstraite, la syntaxe concrète et la sémantique du langage.

La syntaxe abstraite d'un langage définit les concepts qui composent le langage et les relations entre ces concepts, ce qui permet de définir la structure d'un modèle (un programme) écrit dans le langage. Un ensemble de règles syntaxiques sont aussi définies et peuvent être utilisées pour valider un modèle et ainsi assurer qu'il est syntaxiquement correct.

La syntaxe concrète correspond à la notation utilisée pour la représentation des modèles. Il existe deux types de syntaxes concrètes : textuelles et graphiques. La syntaxe textuelle permet d'écrire un modèle sous forme de texte ; en revanche, la syntaxe graphique sous forme de diagramme.

Finalement, le sens des modèles écrits dans le langage est définit par la sémantique du langage. Il existe plusieurs façons de décrire la sémantique d'un langage [CSW08] :

- la sémantique de traduction : les concepts du langage sont traduits dans un autre langage ayant une sémantique précise ;
- la sémantique opérationnelle : décrit la manière d'exécuter un modèle (programme) ;
- la sémantique extensionnelle : la sémantique est définie comme l'extension d'un autre langage, elle est utilisée lorsqu'un langage est créé par héritage des concepts d'un langage de base.
- la sémantique dénotationnelle : les concepts du langage sont associés à des objets mathématiques tels que des nombres, des tuples, ou bien des fonctions. Par exemple, le concept de nombres naturels (0 à l'infini) est associé au concept de type *Integer* dans le langage Java. Le concept du langage « dénote » l'objet mathématique, et l'objet est « dénoté » par le concept.

Dans la théorie des langages, un langage est défini comme un ensemble (dans le sens mathématique du terme) de phrases, et la notion de grammaire correspond alors à un modèle de ce langage, donc un modèle de l'ensemble des phrases. De façon analogue, dans [BBB+05], un langage de modélisation est défini comme un ensemble de modèles (un ensemble de systèmes jouent le rôle de modèles), et un modèle de ce langage correspond alors à la notion de méta-modèle. Par la suite, nous présenterons le concept de méta-modèle.

5.3.2 Le concept de méta-modèle

Etant donné qu'un langage de modélisation est lui-même un système, il peut devenir le sujet de modélisation. La notion de méta-modèle est introduite alors dans [Fav05].

« *Un méta-modèle est défini comme le modèle d'un langage de modélisation* ».

Puisqu'un système peut être représenté par différents modèles, un langage de modélisation peut avoir alors plusieurs méta-modèles associés. Il est possible, par exemple, d'avoir un méta-modèle pour chacune des facettes d'un langage de modélisation, à savoir, la syntaxe abstraite, la syntaxe concrète et la sémantique. Il est possible aussi qu'un méta-modèle représente en même temps plus d'une facette.

L'introduction de la notion de méta-modèle implique de définir la relation qui existe entre un modèle et son méta-modèle. Cette relation peut être dérivée par l'intermédiaire de la relation entre le modèle et le langage de modélisation (relation dénotation) et entre le langage de modélisation et le modèle qui le représente (relation de représentation). La relation dérivée est connue comme étant la relation de conformité d'un modèle vis-à-vis de son méta-modèle.

Dans [BBB+05], différentes manières de vérifier la relation de conformité d'un modèle vis-à-vis de son méta-modèle ont été identifiées :

- Vérification de conformité empirique. Dans ce cas, il n'existe pas un méta-modèle explicite. Par exemple, des modèles d'une application peuvent être produits et ils sont dits conformes s'ils correspondent à un programme en fonctionnement.

- Vérification de conformité littérale. Dans ce cas, il existe un méta-modèle qui n'est ni formel ni exécutable. Une comparaison entre un document (décrivant le méta-modèle) et le modèle (le code) produit est faite à la main. Par exemple, une comparaison entre une spécification UML et le code produit.
- Vérification de conformité théorique. Dans ce cas il existe un méta-modèle formel mais non exécutable. La vérification est faite aussi à la main, mais la différence est que le méta-modèle est décrit par une théorie. Par exemple, si l'on considère que les règles OCL sont une théorie, la vérification de ces règles vérifie que le modèle est conforme au méta-modèle.
- Vérification de conformité outillée. Dans ce cas, le méta-modèle n'est pas formalisé, mais il existe un outil permettant de valider le modèle. Par exemple, un analyseur syntaxique peut valider un programme source même si le langage n'est pas formellement décrit. Il existe la possibilité de réaliser la validation *a priori*, par exemple, un éditeur qui produit des modèles conformes à son méta-modèle.
- Vérification de conformité par spécifications exécutables. Dans ce cas, il existe une définition formelle du méta-modèle qui est utilisée pour produire automatiquement les outils de vérification. Par exemple, dans le domaine de langages, une grammaire peut être utilisée pour produire l'analyseur syntaxique.

5.3.3 Approches de méta-modélisation

Nous considérons que la principale utilité d'avoir une représentation explicite et formelle des méta-modèles est la capacité de produire de façon fiable, voire automatique, l'ingénierie nécessaire pour le traitement automatique des modèles. Cette représentation permet de créer les outils servant à l'édition des modèles, leur vérification automatique, leur combinaison et leur transformation. Ainsi, les modèles peuvent être utilisés comme artefacts de premier ordre dans la construction des applications.

Dans cette section, nous allons présenter certaines des approches de méta modélisation, choisies pour leur utilisation répandue.

MDA, MOF et UML

L'OMG (acronyme en anglais de *Object Management Group*) est un consortium ayant eu au départ comme objectif la définition des standards pour la construction des applications en utilisant la technologie des objets reparties, et qui s'intéresse aujourd'hui à la définition des standards pour la mise en œuvre de la technologie IDM. L'OMG propose le canevas MDA (acronyme en anglais de *Model Driven Architecture*) qui reprend l'idée de la séparation entre la spécification d'un système, et les détails de comment le système utilise la plate-forme dédiée à son exécution.

La démarche proposée par MDA consiste à (1) spécifier le système indépendamment de la plate-forme utilisée pour son exécution, (2) spécifier les différentes plates-formes d'exécution, puis (3) choisir une des plates-formes et (4) finalement transformer la spécification du système en une spécification pour une plate-forme choisie. Les objectifs sont d'augmenter la portabilité, l'interopérabilité et la réutilisation.

MDA préconise l'utilisation de trois perspectives dans la construction d'une application :

- Indépendante de l'informatique (CIM) ; ce point de vue s'intéresse aux besoins de l'application et de son environnement. Les détails comme la structure et le traitement fait par le système sont inconnus.
- Indépendante de la plate-forme (PIM) ; ce point de vue décrit la logique d'exécution du système sans indiquer les détails associés à une plate-forme d'exécution spécifique. Le degré d'indépendance d'un modèle vis-à-vis de sa plate-forme

d'implémentation est considéré comme une propriété du modèle [OMG03b]. Les modèles PIM sont exprimés dans un langage de modélisation à usage général ou bien dans un langage dédié à un domaine d'application.

- Dépendante de la plate-forme (PSM) ; dans ce point de vue se combinent les détails exprimés dans la spécification PIM avec les détails de la plate-forme qui sera utilisée lors de l'exécution du système.

Si l'idée de séparation de la logique métier d'une application de sa plateforme d'exécution parait intuitive, la notion de plate-forme technologique est assez vague et dépendante du contexte, donc une distinction claire entre un modèle PIM et PSM est difficile à réaliser. Cependant, il est important de signaler l'idée de construction d'un logiciel comme une tâche de raffinement entre plusieurs modèles d'un système à différents niveaux d'abstraction.

MDA est basé sur trois technologies basiques : le langage de modélisation objet UML, le standard pour la transformation des modèles QVT (acronyme en anglais de *Query/View/Transformation*), et le formalisme pour la spécification de méta-modèles MOF (acronyme en anglais de *Meta-Object Facility*), qui est la base de la technologie de méta-modélisation de MDA.

MDA propose une architecture de méta-modélisation basée sur quatre couches : la couche de base M0 représente les objets de la réalité ou dans notre terminologie le système modélisé. Ensuite, dans la couche M1, nous retrouvons les modèles du système modélisé, la couche M2 contient le méta-modèle utilisé dans la définition des modèles de la couche M1. Finalement la couche M3 contient un méta-méta-modèle qui est utilisé pour décrire tous les méta-modèles utilisés dans M2, ce méta-méta-modèle est auto-descriptif (méta-circulaire).

La relation de conformité établie entre un modèle et son méta-modèle peut être utilisée aussi entre un méta-modèle et son méta-méta-modèle, ce qui permet l'utilisation d'un langage unique pour la méta-modélisation, ce qui fournit un cadre unifié pour la création de méta-modèles, dans le cas du MDA ce langage est MOF. Le méta-modèle du langage UML lui-même a été défini en utilisant MOF, ainsi que les autres méta-modèles proposés par MDA comme CWM [OMG03a] et SPEM [OMG08]. Dans la figure ci-dessous est présentée l'architecture de méta-modélisation proposée par MDA.

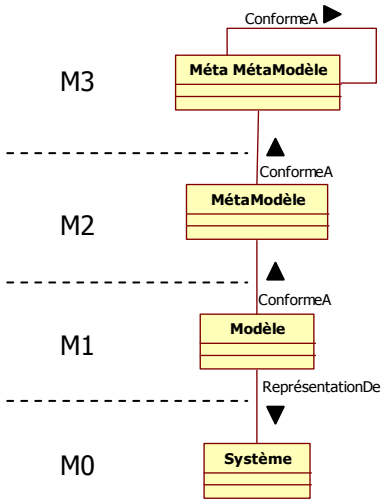

Figure 23. Architecture de méta-modélisation propose par l'approche MDA.

L'Eclipse Modeling Framework (EMF)

EMF est un canevas de traitement des méta-modèles. Il est composé d'un méta-langage orienté objet *Ecore* pour la spécification de méta-modèles, d'une API permettant le traitement

de ces méta-modèles et de leurs modèles, et des outils de génération de code dans le langage Java [BSM+03]. EMF utilise une architecture de méta-modélisation similaire à celle de MDA, en effet, *Ecore* est considéré comme un sous-ensemble du formalisme MOF proposé par l'OMG. Le canevas EMF connaît une vaste utilisation, principalement du au fait de la disponibilité des outils (contrairement au MOF qui est juste une spécification), et du fait qu'il a été bâti au-dessus de l'environnement Eclipse.

L'intérêt d'EMF n'est pas de fournir un langage généraliste de modélisation orienté objet comme UML, mais de fournir tout l'outillage nécessaire pour la définition de méta-modèles et leur exploitation dans différents types d'applications. En fait, deux principaux usages sont visés par le canevas EMF : d'abord son utilisation dans les applications qui font du traitement intensif de données, ensuite dans la définition de nouveaux langages dédiés et la création des outils associés.

EMF fournit des outils de génération de code afin de représenter en Java les différents concepts définis dans le méta-modèle, ce code est généré à partir de la définition explicite d'un méta-modèle en *Ecore*. Une API de traitement de modèles est aussi fournie par EMF, cette API permet la création, la navigation, la validation, la modification et la sauvegarde des modèles conformés aux méta-modèles. Cette API et le code généré pour représenter les concepts du méta-modèle sont indépendants de la plateforme Eclipse ; ils peuvent donc être utilisés dans n'importe quelle application Java. Il est possible aussi de produire des artefacts intégrés dans l'environnement Eclipse, par exemple des éditeurs et des validateurs qui sont des outils précieux par exemple, pour définir de nouveaux langages dédiés.

Autres approches

Il existe différentes façons de définir la sémantique d'un méta-modèle. D'abord, il est possible de définir une transformation spécifiant comment les concepts d'un méta-modèle seront transformés en concepts d'un autre méta-modèle ayant une sémantique connue. Une autre option consiste à construire une machine d'exécution (dans un langage de programmation avec une sémantique définie) pour le méta-modèle. Cette machine est chargée d'interpréter ou d'exécuter les modèles conformes au méta-modèle. Finalement, il est possible aussi d'inclure dans la définition explicite du méta-modèle la sémantique opérationnelle, une machine d'exécution est chargée de comprendre cette spécification. Par contre, la machine est générique et non spécifique à un méta-modèle donné.

Parmi les approches de méta-modélisation ajoutant explicitement la sémantique opérationnelle dans le méta-modèle, nous pouvons citer Kermeta [MFJ05] et Xactium [CSW08].

Kermeta propose un canevas de méta-modélisation qui étend le méta-langage EMOF (*Essential MOF*) afin d'ajouter la sémantique opérationnelle aux méta-modèles. Des extensions consistent à inclure dans le langage : des structures de contrôle (des conditionnels, des boucles), la définition de variables locales, des expressions d'invocation, des expressions d'affectation de variables, des expressions de gestion des exceptions entre autres. Kermeta permet étendre un méta-modèle existant en ajoutant la sémantique opérationnelle sans modifier la partie structurelle du méta-modèle. Cette propriété fait que le méta-modèle original continue à être conforme au langage EMOF.

Des approches de méta-modélisation comme Kermeta et Xactium sont utiles pour la création de langages dédiés exécutables, car la définition explicite de la sémantique opérationnelle permet de simuler l'exécution et tester le langage au moment de la conception. Par contre, elles doivent inclure des machines d'exécution au niveau méta-modèle capables d'interpréter la définition de la sémantique, ce qui fait que les approches rentrent dans les problèmes de définition de la sémantique d'exécution déjà abordés dans les langages orienté objet. Par exemple, dans le cas de la redéfinition des méthodes, il faut déterminer le code qui doit être exécuté lorsqu'une invocation d'une méthode est effectuée.

Une autre préoccupation consiste à déterminer si la machine d'exécution est assez performante pour être utilisée comme interprète du langage. Dans le cas où la performance d'exécution est souhaitée, il sera nécessaire de construire une machine plus performante dédiée à l'exécution des modèles.

5.4 COMPOSITION DE MODELES

La séparation des préoccupations dans l'IDM doit faire face à deux défis, d'abord comment choisir les divers points de vue utilisés pour la décomposition de l'application, et ensuite, une fois l'application spécifiée en utilisant les différents point de vues, déterminer comment composer les modèles qui ont servi à spécifier l'application depuis ces points de vue.

Certaines approches préconisent l'utilisation d'un nombre fixe de points de vue. Pour sa part, le langage UML offre aussi un bon support pour la modélisation de systèmes depuis un nombre de points de vue fixes. Dans UML, des concepts qui apparaissent dans un des point vue sont dépendants des concepts d'autres points de vue. Par exemple, les participants d'un diagramme d'interaction UML doivent avoir leurs types (classes) définies dans un modèle structurel (diagramme de classes) [OMG07].

Malgré le support offert par ces langages et canevas comme UML et MDA, il existe le besoin d'un meilleur support pour gérer la complexité du développement d'une application adressant différentes préoccupations qui dépendent les unes des autres [FR07]. Une tendance actuelle consiste à utiliser des points de vues dédiées. Un point de vue dédié contient des éléments permettant de représenter les concepts d'une préoccupation donnée à différents niveaux d'abstraction ainsi que des indications pour créer des modèles en utilisant ces éléments. Généralement pour faciliter cette tâche, pour un point de vue est défini un langage dédié, qui est utilisé afin d'exprimer les concepts et la façon de les utiliser.

Bien qu'il soit important de déterminer comment une application doit être découpée en différentes préoccupations, les travaux dans ce domaine sont peu nombreux, généralement ce découpage est une décision des concepteurs des applications. Par contre, il existe un nombre important de travaux essayant de résoudre le problème de la composition de ces divers points de vue. Particulièrement, ils visent depuis plusieurs perspectives, la composition des modèles utilisés pour représenter chacune des préoccupations.

Afin de clarifier cette opération de composition, nous allons d'abord présenter le concept de composition de modèles et ensuite faire une caractérisation des mécanismes de composition, finalement quelques approches traitant la composition de modèles seront dévoilées.

5.4.1 Définition de la composition de modèles

Une composition de modèles est définie comme :

« Model composition is an operation that combines two or more models into a single one. » [FBV06]

« Model composition in its simplest form refers to the mechanism of combining two models into a new one. »[HHKR+07]

Dans ces définitions, la composition est décrite comme un mécanisme pour combiner deux (ou plus) modèles dans un nouveau modèle. Ces définitions sont très abstraites et ne font pas d'hypothèse sur les modèles en entrée, les modèles en sortie ni l'opération de composition. Une caractérisation de la composition de modèles sera réalisée par la suite afin de mieux identifier les approches de composition.

5.4.2 Caractérisation de la composition des modèles

Nous allons réaliser une caractérisation de mécanismes de composition de modèles basée sur deux groupes de critères : d'abord, selon les caractéristiques des modèles source et résultat et ensuite, selon les caractéristiques de l'opération de composition utilisée.

Caractéristiques des modèles en entrée et en sortie

La quantité de modèles en entrée

L'opération de composition peut recevoir en entrée deux ou plusieurs modèles. La plupart des approches prennent deux modèles en entrée pour la composition. S'il est nécessaire de composer plus de deux modèles, la composition se fait de façon itérative, d'abord en composant deux modèles, ensuite le modèle composite avec le suivant, etc. Il faut remarquer que si l'opération de composition est associative et commutative, la composition successive de plusieurs modèles produit sémantiquement le même modèle composite. D'autre part, il existe des approches de composition permettant la composition de plusieurs modèles à la fois, c'est le cas d'EMF, où plusieurs modèles peuvent être importés et composés [BSM+03].

Le rôle des modèles composés

Il existe des approches utilisant un modèle de base et des modèles des aspects au moment de la composition. Ces approches, dites de composition asymétrique [RGF+06] utilisent le même style de composition que les langages orientés aspect comme AspectJ. D'autres approches ne font pas de différence entre les modèles à composer ; ce sont des approches symétriques [Ber03].

Le type de modèles composés

Certaines approches composent des modèles structuraux, comme par exemple, les schémas de bases de données [Ber03], ou les diagrammes de classes UML [FRGG04] [BC04]. D'autres, s'intéressent aux modèles comportementaux comme les réseaux de Pétri ou des diagrammes d'état UML [NSC+07], etc.

Les méta-modèles des modèles composés

Si nous considérons une composition binaire prenant en entrée les modèles *m1* et *m2* conformes aux méta-modèles *M1* et *M2* respectivement, et produisant comme résultat de la composition un modèle *m3* conforme au méta-modèle *M3*. Deux cas de figure peuvent se présenter, à savoir :

- Cas1 : $M1 = M2 = M3$: les modèles en entrée et en sortie de la composition sont conformes au même méta-modèle.
- Cas 2 : $M1 \neq M2 \neq M3$: les modèles en entrée et en sortie sont tous conformés à des méta-modèles différents.

Le premier cas correspond aux approches dites endogènes, c'est le cas le plus répandu, des approches comme GME [KML+04] correspondent à ce cas. Des mises en correspondance sont nécessaires seulement au niveau modèle car au niveau méta-modèle on compose uniquement des éléments du même type. Une hypothèse faite pour ce type d'approches est certaines paires d'élément (e1, e2), appartenant au modèle *m1* et au modèle *m2* correspondent à différentes vues d'un même élément. La correspondance des éléments est déterminée par leur signature qui spécifie l'unicité d'un élément, dans ce cas, *e1*, *e2* ont la même signature. Une opération (fusion, surcharge, remplacement) est appliquée afin d'avoir dans le nouveau modèle *m3* un élément composé *e3*. Un cas *dégénéré* apparaît lorsque le méta-modèle utilisé est fixe, c'est le cas de la composition de modèles UML (principalement des modèles structuraux comme les diagrammes de classes). Des approches comme Theme/UML [BC04] [Cla02] et l'approche présentée par [FFR+07] réalisent ce type de composition en faisant des extensions

dans le méta-modèle d'UML pour inclure la sémantique de composition. L'avantage principal de la composition endogène est que la sémantique des méta-modèles est bien connue, et que les relations sont faites uniquement au niveau modèle.

L'autre cas correspondant à une composition exogène, est plus général, mais aussi plus complexe et ambitieux. Dans ce cas des mises en correspondance sont établies au niveau des méta-modèles sources pour signaler l'existence d'une relation entre les types des éléments à composer. Certaines approches utilisent des relations définies explicitement comme dans AMW [FBV06], autres approches utilisent des règles pour établir les correspondances comme dans EML [KPP06b]. La définition des relations n'exprime pas la sémantique de la composition. Par exemple, si un type *T1* appartenant au méta-modèle *M1* a une relation avec un type *T2* du méta-modèle *M2*, il est nécessaire de définir des règles indiquant comment les instances de ces types seront traitées afin d'obtenir une instance d'un type *T3* du méta-modèle *M3*. La composition exogène applique les opérateurs de composition au niveau de types, donc tous les éléments de *m1* ayant pour type *T1* seront composés avec tous les éléments de *m2* ayant pour type *T2*.

Caractéristiques de l'opération de composition

Le type de composition

Nous avons identifié essentiellement deux types de composition : par application des opérateurs ou bien par établissement de relations.

Dans la composition par application d'opérateurs, des opérateurs tels que la fusion, le remplacement, l'union, le tissage, etc. sont définis dans une première phase de la composition. Ensuite, la composition est réalisée en appliquant les opérateurs de composition sur les modèles source. Le résultat de la composition est le modèle produit par l'application des opérateurs. Ni les modèles sources ni les opérateurs définis ne font parti du modèle composite.

Dans la composition par l'établissement de relations, des relations telles que l'association, l'agrégation et l'héritage sont créées entre les éléments des modèles qui sont composés. Le résultat de la composition contient les modèles source plus les relations établies entre les modèles, ces relations font partie de la spécification de la composition ainsi que du modèle composite final.

Le formalisme utilisé dans la spécification de la composition

Des formalismes sont utilisés afin de spécifier les opérations d'une composition. Ces formalismes varient selon les caractéristiques de l'approche de composition utilisée. Nous pouvons citer les types de formalismes suivants: des langages de tissage [FBV06], des méta-modèles de règles de composition [KPP06b] et de profils UML [Cla02].

Nonobstant la diversité des formalismes utilisés dans la composition de modèles, il est possible de les comparer selon deux propriétés, d'abord les abstractions de composition fournies par le formalisme, et ensuite, leur capacité d'extension.

Les formalismes utilisant des opérateurs fournissent des abstractions comme la correspondance (*matching*), le tissage (*weaving*), la fusion (*merge*), le remplacement (*replace*), le surcharge (*override*), etc. Certaines approches proposent des opérateurs prédéfinis, par exemple AMW propose l'opérateur de correspondance comme prédéfini, pour sa part, EMF fourni la correspondance, la fusion et la transformation. Il faut noter que même si différentes approches fournissent les mêmes types d'opérateurs, la sémantique et l'implémentation de ces opérateurs peuvent varier entre les approches.

Malgré les abstractions fournies par les formalismes de composition, qui permettent en général la mise en œuvre d'un grand nombre de scénarios de composition, il existe des cas où la composition ne peut pas être exprimée en utilisant seulement ces abstractions. La propriété d'extensibilité des formalismes devient alors importante, afin de fournir aux utilisateurs la capacité de définir leur propre sémantique de composition de modèles. L'extensibilité d'un

formalisme peut s'effectuer de deux façons : en créant de nouvelles abstractions dans le formalisme, ou en redéfinissant la sémantique des abstractions prédéfinies dans le formalisme. Par exemple, AMW [FBV06] permet l'inclusion de nouvelles abstractions, alors qu'EML [KPP06b] admet de modifier la sémantique des abstractions du formalisme.

Stratégie de composition

Trois grands domaines de recherche s'intéressent à la composition de modèles, la modélisation orientée aspect (AOM), les plates-formes génériques pour le traitement de modèles et les approches de méta-modélisation. Chacune de ces approches utilisent leur propre stratégie de composition, que nous allons détailler par la suite.

La modélisation orientée aspect (AOM) est un paradigme qui s'inspire des travaux techniques de programmation orienté aspect. Les approches suivant ce paradigme, décrivent les différentes préoccupations en utilisant des langages de modélisation généralistes.

Il existe une première catégorie des approches AOM qui fournit des techniques pour étendre des langages de modélisation généralistes comme UML avec des concepts de l'AOP comme des points de jonction (*join point*) et des greffons (*advice*). Les modèles ne sont pas vraiment composés, ils sont transformés en programmes implémentés à l'AOP. C'est-à-dire, que lors d'une phase de génération, le code des aspects est produit. Le couplage entre les abstractions de ces approches et les concepts de l'AOP est fort.

Une seconde catégorie d'AOM, cherche également à séparer les préoccupations mais, depuis des niveaux d'abstraction plus hauts. Dans ces approches, à chaque aspect correspond une vue exprimant (généralement en UML) comment la préoccupation est traitée à la conception [FRGG04] [Cla02] [BC04]. Les approches AOM utilisent une vue de base et des aspects, les éléments appartenant aux modèles des différentes vues peuvent avoir une information partielle d'un concept. Par exemple, une classe peut voir certains attributs dans le modèle de base et d'autres sur une vue représentant un aspect. L'opérateur de composition utilisé par l'AOM est le tissage de modèles qui produit un modèle ayant une perspective globale. Ce modèle global peut être utilisé pour générer une implémentation de l'application, mais cette implémentation n'utilise pas les techniques d'AOP.

Les approches basées sur les plates-formes de gestion de modèles utilisent les opérateurs génériques fournis par ces plates-formes afin d'exprimer la sémantique de composition de modèles. Parmi les opérateurs mis à disposition par ces plates-formes, nous pouvons trouver : des opérateurs pour l'identification des éléments des modèles à composer (*match*, *relate* et *compare)*, des opérateurs pour l'intégration des modèles (*merge*, *compose* et *weaving)*, ou des opérateurs pour lier des modèles sans modifier leur structure (*sewing).*

Parmi les plates-formes de gestion de modèles possédant quelques uns de ces opérateurs nous pouvons trouver AMMA [FJ05], RONDO [MRB03], Epsilon [KPP06a], MOMENT [BCR05] et GME [KML+04].

Finalement, certaines approches de méta-modélisation offrent des capacités de composition de méta-modèles. Il est important de clarifier que les mêmes mécanismes appliqués dans la composition de modèles, sont potentiellement applicables aux méta-modèles, étant donné que nous considérons qu'un méta-modèle est lui-même un modèle d'un langage de modélisation [Fav04a].

Par exemple, la technologie de méta-modélisation EMF permet d'importer des méta-modèles existants dans la spécification d'un nouveau méta-modèle. Des relations peuvent être alors établies entre les éléments des méta-modèles importés et les éléments du nouveau méta-modèle. Il est possible de réaliser ce type de composition car EMF utilise une approche endogène, donc le méta-méta-modèle est le même pour tous les méta-modèles (le langage Ecore). La sémantique des relations établies entre les éléments est aussi définie en utilisant Ecore. Il est pertinent de clarifier, si les modèles conformes aux méta-modèles composés seront eux aussi composés. Dans le cas d'EMF, il est possible de réutiliser les modèles existants dans

la définition d'un nouveau modèle, en fait, la réutilisation est même obtenue au niveau des outils.

5.4.3 Approches de composition de modèles

Plusieurs approches de composition de modèles ont été proposées. De ces approches nous avons choisi celles qui sont les plus utilisées et qui appartiennent aux différents domaines de recherche présentés auparavant.

AMW

AMW est un module de la plate-forme de gestion de modèles générique AMMA. L'objectif de ce module est la création de relations (de liens) entre les éléments des modèles ou des méta-modèles. Les modèles composés par AMW sont des modèles structuraux, spécifiquement orientés objet, définis en utilisant Ecore.

Le formalisme de tissage utilisé par AMW permet de définir des liens structuraux (éléments de composition) entre les modèles. Les définitions de ces liens sont sauvegardées dans un modèle de tissage. Les liens de base définis dans le formalisme n'ont pas de sémantique associée, c'est par extension du méta-modèle de tissage que de nouveaux concepts de tissage sont exprimés. Ce mécanisme d'extension permet d'avoir un méta-modèle de tissage dédié à la composition de modèles. Le formalisme de tissage peut être étendu pour définir des opérations différentes à la composition telles que la comparaison et même la transformation des modèles.

Le modèle de tissage spécifie l'opération de composition, mais le modèle lui-même n'est pas exécutable. Afin d'obtenir le modèle composite, il faut traduire la spécification sous la forme d'un programme dans un langage exécutable, et c'est l'exécution de ce programme qui va produire le modèle composite. Etant donné qu'AMW fait partie de la plateforme AMMA, par défaut, les spécifications décrites dans le formalisme d'AMW sont traduites vers des spécifications de transformation dans le langage ATL. Ensuite, l'application de la transformation aux modèles sources, produit le modèle composite.

EMF

Afin de définir des nouveaux méta-modèles, EMF permet d'utiliser des méta-modèles existants. L'intégration est faite par une importation de méta-modèles existants dans la définition du nouveau méta-modèle. Ainsi, les concepts du nouveau méta-modèle peuvent définir des références (relations) vers les éléments des méta-modèles importés. Nous pouvons considérer ce mécanisme comme une composition par relations. Le (méta) modèle composite contient le nouveau méta-modèle, les méta-modèles importés et les références intra-modèles qui ont été définies. Dans la sémantique d'EMF, une référence inter-modèle n'est pas différente d'une référence intra-modèle.

Dans EMF une grande quantité de code peut être généré à partir de la définition d'un méta-modèle. Un de principaux usages de cette technique est la génération des éditeurs permettant de créer des modèles conformes aux méta-modèles définis. Lorsqu'un éditeur est généré à partir d'un méta-modèle composite, les modèles conformes aux méta-modèles composés qui ont été créés auparavant peuvent être importés dans le nouvel éditeur, cette propriété permet de réutiliser les modèles créés ainsi que les éditeurs eux-mêmes.

EML

EML (*Epsilon Modeling Language*) est un langage à base de règles (*rule-based language*) pour la composition de modèles. Il est créé comme une extension du langage EOL (*Epsilon Object Language*) qui est un langage générique pour la gestion de modèles. EOL est basé lui-même sur le langage de définition de contraintes OCL. D'autres langages ont été définis au-dessus d'EOL parmi eux: ECL pour la comparaison de modèles, ETL pour la

transformation de modèles et EGL pour la transformation de modèle textuels. Tous ces langages ont été créés en utilisant la plateforme Epsilon.

EML spécifie trois types de règles afin de réaliser la composition de modèles ; la comparaison, la fusion et la transformation. Une règle de comparaison sert à déterminer si deux éléments de deux modèles sont équivalents. Cette règle a une partie comparaison (*compare*) et une partie de vérification de conformité (*conform*). A l'exécution, la partie comparaison est appliquée afin de déterminer si deux éléments se correspondent ; la partie vérification de conformité est appliquée pour déterminer si les éléments sont conformes.

Les règles de fusion sont utilisées pour définir la sémantique de fusion des éléments ; cette spécification est définie par la personne spécifiant la règle. Une règle de fusion définit trois paramètres. Les deux premiers correspondent aux modèles sources, et le troisième à l'élément du modèle composite qui est le résultat de la fusion des deux premiers. Le corps de la règle définit les actions à suivre afin de fusionner les éléments. Les règles de fusion sont appliquées uniquement sur les paires d'éléments qui sont correspondants et conformes.

Le troisième type de règles (les transformations) indique comment un élément d'un modèle source sera transformé en un élément du modèle composite. Une règle de transformation définit deux paramètres, le premier pour l'élément source et le deuxième pour l'élément destination. Les règles de transformation sont appliquées sur les éléments qui ne sont pas conformes par les règles de comparaison. De la même façon que pour les règles de fusion, la sémantique des règles de transformation est spécifiée par l'utilisateur dans le corps de la règle.

Le processus de composition consiste alors d'abord en une phase de comparaison suivie d'une phase de fusion. Dans la phase de comparaison, le moteur d'exécution va déterminer si deux éléments des modèles à composer sont correspondants et conformes. Une fois les règles de comparaison appliquées, les éléments des modèles sources sont classés en quatre catégories : 1) correspondants et conformes, 2) correspondants mais non conformes, 3) ni correspondants ni conformes, 4) les éléments pour lesquels il n'y a pas de règles de comparaison à appliquer. Dans la phase de fusion, pour les éléments classés comme correspondants et conformes, les règles de fusion sont exécutées ; pour les éléments ni correspondant ni conformes, les règles de transformation sont appliquées. Pour des éléments correspondants mais non conformes, le processus de composition sera arrêté, car EML considère qu'il s'agit d'éléments en conflit. Finalement, pour des éléments pour lesquels aucune règle de comparaison n'a été appliquée, un avertissement est fait à l'utilisateur pour indiquer qu'il est possible que la spécification de composition soit incomplète.

L'approche EML supporte l'utilisation de divers méta-modèles même s'ils ont été définis en différentes technologies de méta-modélisation comme MOF et Ecore. Explicitement EML vise seulement la composition de modèles et non celle de méta-modèles. Un avantage d'EML est l'existence d'outils supportant la fusion de modèles, et la disponibilité de ces outils dans la plate-forme Eclipse.

5.4.4 Synthèse

Dans cette section, nous avons caractérisé la composition de modèles et nous avons présenté les propriétés de trois approches essayant de réaliser de différente manière cette opération. Par contre, la composition de méta-modèles n'a pas été explicitement dévoilée. Le fait qu'un méta-modèle est lui-même un modèle (d'un langage de modélisation), permet utiliser les mécanismes de composition de modèles pour réaliser la composition de méta-modèles. Il est important de remarquer que lorsqu'il s'agit de composition de modèles exogène, où les modèles à composer ne partagent pas le même méta-modèle, des mises en correspondances doivent être définies entre les concepts des méta-modèles. Mais, cette définition de relations ne peut pas être entendue comme la composition de méta-modèles puisqu'il ne produit pas un nouveau méta-modèle en partant de méta-modèles originaux.

Il existe un autre type de composition fournissant des mécanismes permettant la composition à deux niveaux, méta-modèle et modèle. Dans ce type de composition un nouveau méta-modèle *M3* est le résultat d'une composition des méta-modèles *M1* et *M2*. A savoir, *M3* = (*M1* Γ *M2*)[2]. De la même façon, les modèles m1 et m2 (respectivement conformes à M1 et M2) sont composés pour produire un modèle *m3* conformé à *M3*. Une propriété souhaitée dans ce type de composition est la capacité de garder les modèles originaux sans modification. Cette propriété améliore la réutilisation, surtout si l'on considère des modèles exécutables, où la réutilisation n'est pas seulement importante au niveau de modèles (et méta-modèles) mais aussi des machines d'exécution. Un tableau résumant les approches étudiées et leurs principales caractéristiques est présenté dans la Figure 24.

	AMW	EMF	EML
Type de composition	Par opérateurs	Par relations	Par opérateurs (fusion)
Technologie de méta-modélisation	Ecore	Ecore	MOF, Ecore
Domaine de recherche	Gestion de modèles	Méta-modélisation	Gestion de modèles
Formalisme	AMW (Transformé vers ATL)	N/A	EML
Mécanisme de composition	Tissage (Transformation)	Etablissement de références	Fusion

Figure 24. Tableau comparatif des technologies de composition de services.

5.5 SYNTHESE

Nous avons présenté dans ce chapitre l'approche IDM qui vise à utiliser les modèles d'artefacts de premier ordre dans la construction des applications. L'IDM à notre regard fournit deux propriétés essentielles pour la construction des applications. Tout d'abord, la méta-modélisation préconise l'utilisation de langages de haut niveau d'abstraction, pouvant être exploitables automatiquement. Ensuite, l'expression d'une application est faite par le biais de plusieurs modèles, où chacun représente différents points de vue de l'application, une séparation des préoccupations est ainsi appliquée. La technique de composition de modèles (et méta-modèles) est alors utilisée pour l'intégration de ces points de vue.

Dans le cadre de cette thèse, l'ingénierie dirigée par les modèles sera utilisée comme paradigme pour la construction des applications orientées procédé. Nous voulons profiter des deux propriétés fournies par l'approche, tout en l'utilisant dans un contexte concret et spécifique, celui des applications orientées procédé. En plus, afin de faire un usage réussi de l'approche, nous considérons qu'un outillage adéquat doit supporter les tâches d'expression, de transformation, de raffinement, de composition et d'exécutions des modèles.

[2] Ici Γ est l'opérateur de composition

6. NOTRE PROPOSITION

Dans le cadre de cette thèse nous nous intéressons à la construction d'un canevas supportant la création des applications orientées procédé. Ce système sera utilisé pour la création de différents types d'application comme la gestion de documents, le suivi automatisé de l'exécution de procédés métiers (*workflow*), la spécification de systèmes de médiation, etc. Cependant, nous nous sommes focalisés sur l'approche d'orchestration de services, et nous avons décidé d'utiliser cette approche comme la base pour la construction du canevas.

Bien qu'il existe des standards au niveau des modèles d'orchestration de services, ces standards et leurs implémentations respectives ont des inconvénients comme nous l'avons décrit dans le chapitre 4. Par conséquent, notre canevas appelé FOCAS (acronyme en anglais de *Framework for Orchestration, Composition and Aggregation of Services*) fait face aux défis présentés dans le chapitre de l'état de l'art sur l'orchestration de services. Mais, en plus FOCAS fournit des mécanismes d'extension qui permettent de créer des applications orientées procédé adaptées aux besoins et contextes d'utilisation spécifiques. Le canevas et ses mécanismes d'extension suivent une approche IDM et ont été développé avec trois principes basiques du génie logiciel en tête : la séparation de préoccupations, la remonté du niveau d'abstraction et la réutilisation.

Dans ce chapitre, nous allons présenter notre proposition ainsi que l'approche suivie pour sa mise en application. Tout d'abord nous allons clarifier le concept d'application orientée procédé et nous allons mettre en évidence les problèmes liés à cette technologie. Ensuite, nous allons montrer que l'expression de ce type d'application comporte trois points de vue : le contrôle, les données et les services. Puis, nous allons montrer comment nous avons bâti le noyau de notre canevas en intégrant ces trois points de vue. La quatrième section du chapitre, est dédiée à la définition du mécanisme d'extension fonctionnelle du canevas. La section suivante, est dédiée à la présentation du mécanisme d'extension non-fonctionnelle. Finalement, nous concluons par une synthèse de notre proposition.

6.1 NOTRE PROPOSITION : VISION GENERALE

Dans cette section, nous allons tout d'abord caractériser les applications orientées procédé, les propriétés de ce type d'application seront présentées. Ensuite, nous allons mettre en évidence la problématique associée à ce type d'application. Puis, l'objectif de cette thèse sera présenté. Finalement, nous présentons de façon générale notre proposition pour atteindre l'objectif ciblé.

6.1.1 Concept d'application orientée procédé

Une application orientée procédé est une application qui coordonne (orchestre) la consommation d'un ensemble de fonctions. Une fonction est responsable de faire un traitement de l'information. Cette coordination est faite par le biais d'un modèle de contrôle qui exprime l'ordre de consommation des fonctions (le flot de contrôle) ainsi que le routage de l'information à traiter par les fonctions (flot de données). L'application est responsable de maintenir l'état de

la coordination, décider l'ordre de consommation des fonctions et fournir l'information à traiter par les fonctions.

Ce patron est retrouvé dans différents types d'applications. Par exemple, dans des systèmes de routage de documents (une des premières utilisations de la technologie de *workflow*), l'application coordonne l'envoi des documents peu structurés (information) entre différents membres d'une organisation (ressources réalisant les fonctions). Un autre exemple est l'orchestration de services : une orchestration invoque différents services (ressources fournissant les fonctions) dans un ordre spécifique en passant les bons paramètres (information) aux services. Ces deux types d'applications utilisent un modèle qui indique explicitement l'ordres de consommation des services et le routage de l'information, nous l'appelons le modèle de procédé.

La vision des applications orientées procédés est une vision de construction d'applications par l'assemblage des fonctionnalités existantes. En fait, c'est la vision aussi de l'approche à composants, mais la caractéristique des applications orientées procédé est l'externalisation du contrôle. De la même façon que l'introduction de bases de données a permis une indépendance entre les données et les applications (fonctions), l'utilisation d'un modèle explicite et externe du contrôle vise à fournir l'indépendance du contrôle des applications [LR97]. Cette indépendance permet de créer des applications flexibles qui peuvent s'adapter rapidement au changement de besoins.

Un modèle de procédé est exprimé en utilisant un graphe orienté, généralement dans un langage graphique. Ce type de formalisme permet d'exprimer des patrons de contrôle complexes [vdAtHKB03], comme par exemple le branchement multiple (*And-Split*) et la synchronisation (*And-Join*). L'expression de ces patrons de contrôle, constitue une différence entre les formalismes utilisés par les applications orientée procédé et les langages de programmation conventionnels. En fait, dans un langage de programmation conventionnel, il est difficile d'exprimer ces patrons (notamment le parallélisme), en plus cette spécification est mélangée avec les fonctionnalités à consommer.

Quelles sont les propriétés de ce type d'applications ?

- Le contrôle s'exprime de façon explicite et il est externe aux fonctions (ou tâches) qui sont coordonnées. Le contrôle est spécifié en utilisant un langage dédié, généralement ce langage possède une syntaxe concrète graphique associée.
- Des patrons de contrôle complexes peuvent être exprimés dans une application orientée procédé. Ces patrons sont difficiles à exprimer dans un langage de programmation conventionnel. Généralement, un interpréteur exécute les modèles de procédés, et ainsi prend en charge la mise en place de ces patrons.

  La nature des tâches d'une application orientée procédé est variée. Par exemple, l'orchestration utilise la notion de service pour représenter les tâches, dans les *workflows* les tâches sont réalisées par des êtres humains, même s'ils utilisent de systèmes logiciels pour les réaliser.
- Une application orientée procédé coordonné des ressources de diverse granularité. Par exemple, dans les applications EAI les ressources sont de grosses applications. Un langage de programmation utilise uniquement le processeur de l'ordinateur comme ressource, et la granularité des instructions est petite.
- Une application orientée procédé utilise la notion d'état d'attente, et elle délègue la réalisation d'une tâche à un tiers. L'application attend que la ressource (service, application, etc.) chargée d'effectuer la tâche signale la finalisation de son travail.

  Le temps d'exécution de ces applications peut être des heures ou des jours voire même des mois. La reprise en cas de panne est alors une propriété souhaitable voire

indispensable pour certains types d'applications orientées procédé, car la possibilité d'exécuter à nouveau l'application est très coûteuse.

6.1.2 Problématique et objectifs

L'étude de l'état de l'art que nous avons présenté dans la première partie de ce document, nous a permis de faire un certain nombre de constatations qui sont à la base de nos travaux :

- *il existe de nombreuses ressources pouvant réaliser les fonctions d'une application orientée procédé*. Parmi ces ressources nous pouvons trouver des applications, des composants, des services, des humains, etc. En général, les applications orientées procédés se focalisent sur un seul type de ressource. Par exemple, l'orchestration se focalise sur les services.
- *il existe de nombreuses technologies de services.* Même, dans l'orchestration de services les approches étudiées ciblent une unique technologie de services, la plupart de ces approches orchestrent des services Web.
- *les formalismes utilisés sont de bas niveau d'abstraction.* L'expression des applications orientées procédés se fait dans des formalismes spécialisés pour la technologie ciblée. Ils incluent de nombreux concepts ce qui rend difficile leur utilisation.
- *les applications orientées procédé doivent comporter des aspects non-fonctionnels.* L'orchestration de services par exemple, permet d'intégrer des services fournis par des tiers dans un environnement ouvert et distribué, des failles de sécurité peuvent se présenter dans un tel environnement. D'autres aspects non-fonctionnels comme la gestion de transactions et la distribution doivent être supportés.
- *les applications orientées procédés doivent être extensibles.* Dans l'orchestration par exemple, des fonctions réalisées par des humains ne sont pas supportées. Une application orientée procédé doit offrir des mécanismes d'extension fonctionnelle afin de viser de nouveaux domaines d'utilisation.
- *les applications orientées procédés doivent être flexibles.* Les besoins des applications orientées procédé peuvent varier d'une application à une autre. Des systèmes flexibles pouvant s'exécuter dans environnements restreints ainsi que dans environnements robustes, doivent être fournis.

Notre objectif est de simplifier la réalisation d'applications orientées procédé. Nous pensons que la spécification de ce type d'applications doit être simple, compréhensible et maintenable. En plus, nous voulons offrir un système extensible pouvant s'adapter aux différents domaines et contextes d'exécution. Pour atteindre cet objectif, nous proposons le canevas FOCAS pour la spécification, l'exécution et le monitoring d'applications orientées procédé. Ce canevas suit trois principes du génie logiciel : la séparation des préoccupations, l'augmentation du niveau d'abstraction et la réutilisation. Une approche basée sur les modèles a été préconisée dans notre canevas pour sa mise en place, afin de supporter les trois principes ciblés.

6.1.3 Notre démarche

Nous avons divisé notre démarche en trois étapes : tout d'abord, la construction du système noyau pour les applications orientées procédé, ensuite la définition des mécanismes d'extension fonctionnelle et finalement la définition des mécanismes d'extension non-fonctionnelle pour le canevas.

Ainsi, la première partie de la démarche consiste à identifier la fonctionnalité de notre noyau pour les applications orientées procédé. Nous avons choisi l'orchestration de services comme point de départ, car elle contient les trois éléments principaux de toute application orientée procédé, à savoir : le contrôle, les données et les services (les fonctions à coordonner ou

orchestrer). Nous considérons qu'une orchestration de services est la composition de ces trois points de vue, où chaque point de vue représente un aspect de l'orchestration. Les concepts appartenant à chaque point de vue ont été identifies et formalisés par le biais d'un méta-modèle. Pour chaque méta-modèle (langage de modélisation) un interpréteur offrant la sémantique du langage a été implémenté. Nous tenons à spécifier les trois points de vue d'une orchestration de façon séparée, mais cette séparation, impose de composer les trois points de vue pour constituer une vision globale de l'orchestration. Nous avons utilisé la technique de composition de domaines pour réaliser cet assemblage, par la suite nous allons utiliser le terme domaine pour nous référer à un point de vue. La section 6.2 présente les domaines de base pour l'orchestration. La section **Erreur ! Source du renvoi introuvable.** présente la composition de ces domaines pour construire le domaine de l'orchestration.

La deuxième partie de notre proposition consiste à définir les mécanismes nécessaires permettant l'usage des applications orientées procédé dans un domaine différent de l'orchestration de services. Ainsi, les caractéristiques d'une extension fonctionnelle ont été identifiées et le mécanisme a été mis en place. Nous proposons à nouveau, d'utiliser l'approche de méta-modélisation afin d'expliciter les concepts du nouveau domaine à intégrer. Ensuite, en utilisant le domaine de contrôle comme pivot, nous allons définir les relations entre les concepts du nouveau domaine avec ceux de l'orchestration. Finalement, la composition de domaines permet d'intégrer les concepts du nouveau domaine avec ceux de l'orchestration. La section 6.4 présente le mécanisme d'extension fonctionnelle.

Finalement, la troisième étape de notre démarche consiste à identifier comment une orchestration de services doit supporter des aspects non-fonctionnels. Ainsi, un mécanisme a été ajouté au canevas afin de supporter des aspects non-fonctionnels. Ce mécanisme utilise aussi la méta-modélisation pour formaliser les concepts de l'aspect non-fonctionnel. Ensuite, l'orchestration est *annotée* avec les propriétés à fournir. Une approche générative est utilisée pour assurer ces propriétés à l'exécution. La section 6.5 présente le mécanisme d'extension non-fonctionnelle proposé par FOCAS.

Toute notre démarche suit une approche de méta-modélisation pour formaliser les concepts de différents domaines ainsi que des aspects non-fonctionnels. Le but de cet usage est de *remonter le niveau d'abstraction* des langages utilisés lors de la spécification d'un aspect de l'application. Ensuite, le fait de séparer les points de vue (ou domaines) favorise la *séparation des préoccupations*, ainsi différents acteurs peuvent intervenir dans la construction d'une application orientée procédé. Finalement, *la réutilisation* des divers artefacts est envisagée, tout d'abord la réutilisation des modèles de chaque domaine, mais aussi des langages (et des outils de spécification) et des interpréteurs de ces langages.

Dans la suite de ce chapitre, nous allons présenter l'approche suivie pour la mise en œuvre de notre proposition. Tout d'abord, nous allons montrer comment la formalisation a été réalisée pour chacun des domaines de base : le contrôle, les données et les services. Ensuite, nous allons monter comment ces domaines sont intégrés afin d'avoir un domaine composite, celui de l'orchestration de services. Puis, les mécanismes d'extension fonctionnelle seront dévoilés et finalement, ceux mis en place pour l'ajout du support des aspects non-fonctionnels de l'orchestration.

6.2 LES DOMAINES DE BASE POUR L'ORCHESTRATION

Nous sommes intéresses dans la construction d'applications orientées procédé et nous avons choisi comme point de départ de notre démarche l'orchestration de services. Cette technologie vise la construction d'applications pour automatiser la réalisation d'un processus de traitement d'information (*Information Process*) [MMWF93]. Ce type de processus sont composés d'un ensemble de tâches automatisés (réalisées pour des applications) pour créer, traiter, gérer et fournir de l'information.

Dans un processus de traitement de l'information trois dimensions sont identifiées :

- une dimension indiquant *comment* le processus sera réalisé ;
- une dimension indiquant l'*information* que le processus traitera ;
- et une dimension indiquant quelles *applications* seront responsables de traiter l'information.

Ces trois dimensions correspondent aux trois point de vues ou domaines existants dans toute orchestration de services ; le contrôle, les données et les services. Le domaine de contrôle exprime l'ordre total ou partiel des tâches à réaliser (flot de contrôle) ainsi que le routage de l'information entre les tâches (flot de données). Ensuite, le domaine de données qui est responsable de la représentation et gestion des entités (l'information) à traiter. Finalement, le domaine de services qui fournit les ressources (applications) nécessaires pour la réalisation des tâches. Cependant, contrairement aux approches actuelles d'orchestration de services, nous avons décidé de maintenir ces trois points de vue indépendants. Les motifs qui nous ont ammené à cette décision sont :

- Préconiser la séparation de préoccupations. Les trois domaines impliqués peuvent être gérés et exprimés de façon indépendante. Par exemple, la logique de contrôle de l'orchestration est indépendante de la structure de donnés échangées par les activités. Notre intention est que chacun des points de vue puisse être exprimé par des acteurs indépendants.
- Améliorer le taux de réutilisation. L'expression indépendante des points de vue permet réutiliser les modèles. Par exemple, pour accroitre la réutilisation d'un modèle de contrôle exprime la logique d'exécution d'un processus particulier (traitement d'une demande d'achat). Si les points de vue sont séparés, ce modèle de contrôle pourra être adapté à différentes applications ou structures de données.
- Flexibilité et adaptation. Notre objectif est de pouvoir adapter le canevas à différents domaines d'utilisation. La séparation entre points de vue favorise la flexibilité car un point de vue peut être étendu ou même remplacé pour une utilisation particulière. Par exemple, si nous voulons un système où les tâches sont exécutées par des humains et non par des applications, le point de vue services peut être remplacé par celui de la structure organisationnelle de l'entreprise.

Le fait de séparer les points de vue de l'orchestration (contrôle, données et services) impose le besoin de les réconcilier afin de donner une vision globale de l'application. Nous allons utiliser le concept de domaine tel qu'il est défini par [Veg05] pour séparer et réconcilier les points de vue de l'orchestration. Ainsi, l'approche de méta-modélisation sera utilisée pour décrire et formaliser le méta-modèle (ensemble de concepts) appartenant à chaque domaine (point de vue). En plus, afin de donner une sémantique d'exécution à chaque méta-modèle, un interpréteur de leurs modèles est implémenté. La technique de composition de domaines, permettra alors de composer les domaines, c'est-à-dire de composer les méta-modèles (et leurs modèles) et les interpréteurs pour donner la sémantique d'exécution de l'orchestration.

Dans cette section nous allons présenter chacun des domaines de base de l'orchestration, son méta-modèle et interpréteur. Dans la section suivante nous allons présenter comment nous avons composé les domaines pour construire le domaine de l'orchestration.

6.2.1 Le domaine de contrôle : le langage APEL

Le domaine de contrôle est considéré comme domaine central dans notre approche. En effet, le modèle de contrôle est *impératif*, c'est-à-dire qu'il exprime l'ordre dans lequel les différentes activités doivent être réalisées. Donc, la dynamique d'exécution du modèle de contrôle est *proactive*, l'interpréteur est chargé d'exécuter le script indiqué par le modèle.

Le méta-modèle de contrôle

Nous avons choisi le langage APEL comme méta-modèle du domaine de contrôle. APEL a comme caractéristique principale d'être basé sur un ensemble minimal de concepts. Ces concepts se concentrent dans l'expression du flot de contrôle et des données entre un ensemble d'actions.

Le concept central d'APEL est le concept d'activité (*Activity*). Une activité exprime une action à réaliser sur un ensemble d'entités sans indiquer la nature de l'action ni celle des entités traitées. Une activité peut être composite si elle contient des sous-activités, dans le cas contraire est une activité est dite atomique. Les entités manipulées par les activités sont appelé produits (*Product*) ; un produit a une type associé (*ProductType*) qui est uniquement un identificateur (chaîne de caractères), c'est-à-dire que la structure d'un produit n'est pas indiqué dans le formalisme.

Chaque activité a un ensemble de ports (*Port*) correspondant à l'interface de communication d'une activité. Chaque port spécifie un ensemble de produits attendus. Une activité a trois classes de ports, à savoir, d'entrée, de sortie et le *desktop*. Les ports d'entrée d'une activité servent à décrire des besoins des activités en termes de produits afin de réaliser son action. Les ports de sortie indiquent les produits que l'activité fournit comme résultat de la réalisation de son action. Le port *desktop* est un port servant à la manipulation des produits pendant la réalisation de l'action de l'activité. Il existe deux types de ports, les ports synchrones qui influent sur le cycle de vie d'une activité, et les ports asynchrones que ne l'affectent pas.

Un flot de données (*Dataflow*) est utilisé pour exprimer une liaison entre deux ports. Un flot de données définit quels produits seront transférés d'un port à un autre. Un flot de donnés relie deux ports de deux activités et exprime un flux de données, c'est-à-dire que l'activité destination attend des entités produites par l'activité source pour réaliser son action. S'il existe un flot de données entre deux activités sans indiquer de données à transférer, un flot de contrôle est exprimé, donc l'activité destination doit attendre la fin de l'exécution de l'activité source pour initier sa propre exécution.

Finalement, pour indiquer la ressource chargée de réaliser l'action d'une activité, APEL utilise le concept de rôle (*Role*). Un rôle est l'abstraction d'un ensemble de ressources capables d'exécuter l'action de l'activité.

Nous ne voulons pas faire une description exhaustive du langage APEL. Notre but est de présenter les principaux concepts et leurs relations afin de comprendre comment sera composé le domaine contrôle avec les autres. Dans notre démarche, nous considérons le contrôle comme étant l'élément *central* autour duquel vont s'attacher les différentes extensions. Les concepts correspondant au formalisme APEL (partie statique du méta-modèle) et leurs relations sont schématisés par les classes en jaune du diagramme présenté dans la Figure 25.

Le langage APEL a une syntaxe concrète graphique. Toute activité possède une double représentation dépendant de la perspective d'édition. De l'extérieur, une activité est représentée par un rectangle avec son nom en haut et le nom du rôle de la ressource associé en bas. Les ports sont de petits carrés avec une flèche indiquant le sens du flux de produits, les ports à gauche d'une activité correspondent aux ports d'entrées synchrones, les ports à droite aux ports de sortie synchrones. Un flot de données est un trait qui lie deux ports, le nom de chaque produit envoyé est marqué sur le flot de données. Depuis l'intérieur, les ports sont représentés par des triangles, le *desktop* par un cercle, et les sous-activités sont visibles. La syntaxe concrète du langage APEL est schématisée dans l'exemple de la Figure 26.

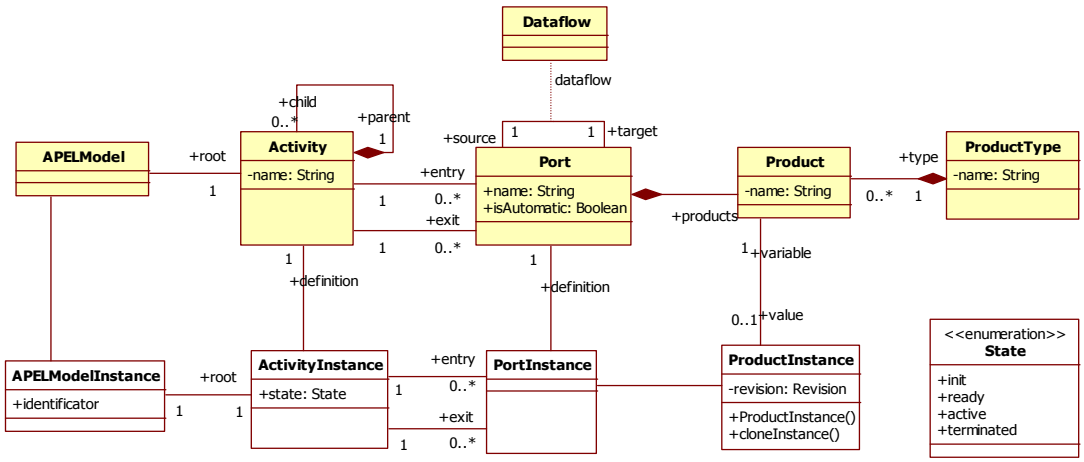

Figure 25. Méta-modèle du domaine de contrôle (APEL).

Exemple d'un modèle d'APEL

L'exemple d'un système d'alarme dans une usine de production est présenté dans la Figure 26. L'objectif du système d'alarme est de surveiller l'état de l'usine en utilisant un ensemble de capteurs et de prendre des actions correctives si l'état ne correspond pas aux niveaux normaux.

Le modèle APEL de l'exemple contient l'activité *Acquisition* qui est chargée de récolter des mesures sur l'environnement, par exemple la température en un point déterminé. Ensuite, ces données sont envoyées (produit *val*) de façon asynchrone vers l'activité *Analyse* qui est chargée de les agréger pour calculer la moyenne des mesures. La moyenne, une fois calculée, est envoyée vers les activités *Storage* et *Processing*. Ces deux activités peuvent être exécutées en parallèle ; l'activité Storage est responsable du stockage des données correspondant aux moyennes, et l'activité *Processing* est chargée de déclencher des actions sur l'environnement dans le cas où les mesures ne sont pas dans une plage de valeur acceptable.

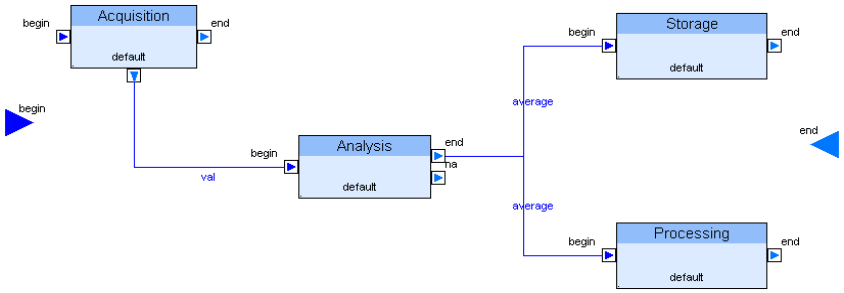

Figure 26. Modèle APEL d'un système d'alarme.

L'interpréteur d'APEL

La dynamique d'exécution d'une instance d'un modèle APEL est associée aux états des activités et de leurs ports. Les classes correspondant à la partie dynamique du méta-modèle et qui font partie de l'implémentation de l'interpréteur sont présentées en blanc dans le diagramme de classes de la Figure 25.

La dynamique d'exécution peut être expliquée par rapport au cycle de vie des activités, donc par rapport à l'ensemble des états possibles. Lorsqu'une instance d'un modèle APEL (*APELModelInstance*) est créée, toutes les instances des activités du modèle (*ActivityInstance*) passent à l'état *init*. Quand un port (*PortInstance*) d'entrée synchrone d'une activité est plein, c'est-à-dire qu'il contient au moins une instance de chaque produit (*ProductInstance*) attendu,

l'activité change son état à *ready*. L'activité reste en l'état *ready* jusqu'à ce qu'une ressource (*Ressource*) du rôle indiqué est assignée pour la réalisation de son action, à ce moment elle passe à l'état *active*.

Une activité reste dans l'état *active* pendant toute la durée de l'exécution de l'action associée à cette activité. L'exécution de l'action d'une activité est la responsabilité de la ressource qui lui a été assigné. Une fois finie l'exécution de l'action, la ressource doit placer les produits résultat dans les ports de sortie de l'activité. Lorsqu'un port de sortie synchrone est plein, il peut être déclenché, c'est-à-dire que les produits qu'il contient sont envoyés aux ports destination de ses flots de données. Le déclanchement d'un port de sortie synchrone passe l'activité à l'état *terminated*. Le diagramme d'état d'une activité APEL est présenté dans la Figure 27.

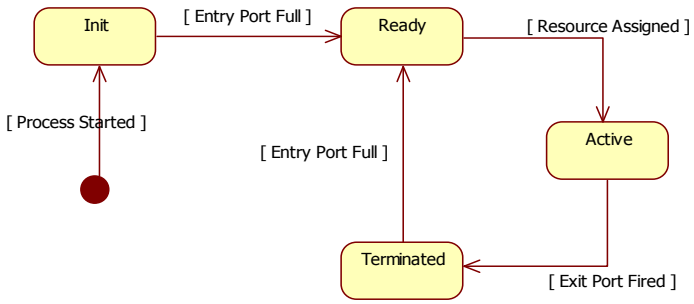

Figure 27. Diagramme d'état d'une activité APEL.

L'état global de l'instance d'un modèle est donné par l'état de toutes les instances des activités et par les produits qui sont dans ses ports à cet instant. La dynamique d'évolution de l'état global est dictée par les interactions entre les activités. Ces interactions sont la conséquence de la communication entre leurs ports, c'est-à-dire que dès qu'un port est déclenché, il envoie de façon atomique ses produits vers les ports destination de leurs flots de données. La transmission fait que des ports d'entrées des activités destination deviennent pleins et changent d'état (*ready* puis *active*).

6.2.2 Le domaine des données

Le domaine des données est chargé de la gestion (création, élimination, modification et consultation) des données utilisées par les instances des orchestrations. Au contraire du modèle de contrôle, le modèle de données est *descriptif*, il sert à exprimer la structure des données utilisées par une orchestration. La dynamique d'exécution de l'interpréteur est *réactive*, donc il exécute des actions sur les données, par exemple la création d'une instance de donnée.

Méta-modèle du domaine

La partie statique du méta-modèle du domaine correspond à un langage de typage permettant de définir des types de données à manipuler. Les concepts de ce langage sont exprimés par les classes en jaune du diagramme de classes de la Figure 28. Un type de données (*DataType*) peut être un type simple (*String, Integer, Float, Boolean*) ou un type complexe. Un type complexe spécifie une liste d'attributs (*Attribute*), les attributs eux-mêmes ont un type simple ou complexe. L'attribut définit aussi un ensemble de propriétés qui déterminent son comportement à l'exécution. Par exemple, si un attribut est définit comme clé (*isKey*) l'attribut sera utilisé pour identifier de façon unique une donnée ; si un attribut est définit en lecture seule (*read-only*), une fois affecté sa valeur il ne peut plus être changé. D'autres propriétés déterminent son comportement par rapport au versionnement. Par exemple, si un attribut est définit comme partagé (*shared*) il aura la même valeur dans toutes les versions de la donnée.

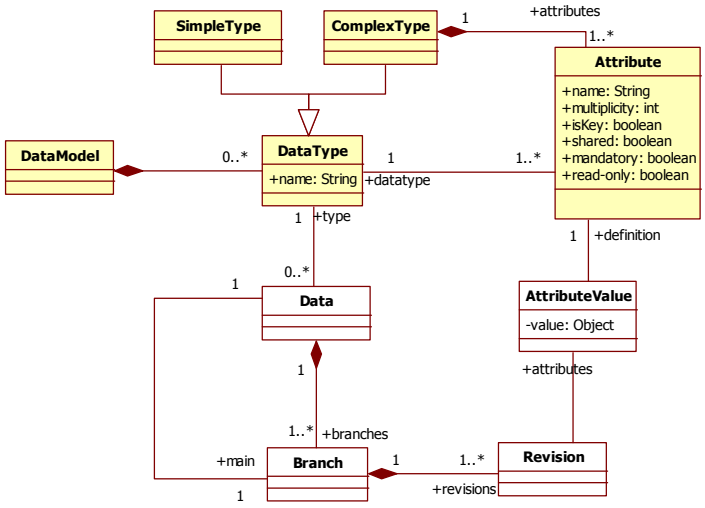

Figure 28. Méta-modèle du domaine de données.

Exemple d'un modèle de données

Un exemple de modèle de données possible pour le système d'alarme est présenté dans la Figure 29. Quatre types de données sont définis, *Temperature*, *Average*, *Action* et *TemperatureList*. Une température a deux attributs, la valeur (double) et l'unité (string) de la mesure (Fahrenheit ou Celsius). A droite de la figure la spécification des propriétés pour l'attribut *unit* du type *Temperature* est exposé.

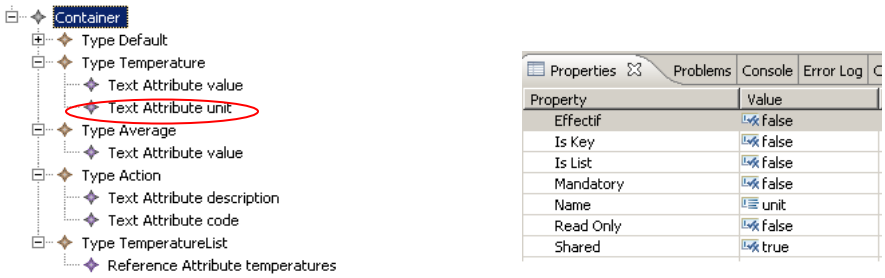

Figure 29. Modèle de données pour le système d'alarme.

L'interpréteur de données

L'interpréteur de données permet la création, élimination, modification et consultation d'un ensemble de données, ainsi que leur versionnement. Une donnée (*Data*) est créée en spécifiant les valeurs pour ses attributs. A la création de la donnée, une branche (*Branch*) principale (*main*) est définie et la première révision (*Revision*) est créée. Le comportement de versionnement de l'interpréteur permet la création de nouvelles branches et de nouvelles révisions. L'interpréteur fournit une API pour la modification des valeurs (*AttributeValue*) des attributs pour une donnée déterminée, ainsi que l'élimination et consultation des données.

La dynamique d'exécution des données est simple du à sa nature réactive, c'est-à-dire que l'interpréteur répond aux événements causés par ses utilisateurs, mais le changement des objets n'entraîne pas de modification de l'état d'autres objets présents dans l'interpréteur. Les classes en blanc du diagramme de la Figure 28 forment la partie dynamique du méta-modèle et font partie de l'implémentation de l'interpréteur de données.

6.2.3 Le domaine des services

Le domaine des services fournit les mécanismes de base de l'approche à services. Il est chargé de la spécification des services, leur sélection et découverte et d'assurer la communication avec les fournisseurs. Un modèle de services est *descriptif*, il spécifie des entités fournissant une fonctionnalité. Ce modèle peut servir à décrire des fonctionnalités requises (*top-down*) où bien de fonctionnalités fournies par les applications existantes (*bottom-up*). L'interpréteur du domaine possède une dynamique d'exécution *réactive*.

Méta-modèle des services

Le méta-modèle des services sert à décrire la fonctionnalité des services. Un service a une description (*Description*) qui contient une interface et un ensemble de propriétés (*Property*). L'interface définit la liste d'opérations (*Operation*) supportées par le service. Nous ne voulons pas spécifier un nouveau langage de description de services car il en existe plusieurs comme par exemple Corba-IDL, WSDL ou les interfaces Java. Nous avons choisi de reprendre les interfaces Java comme formalisme de spécification de services et de les compléter avec les concepts manquants. Étant donné que les interpréteurs des domaines sont eux-mêmes implémentés en Java, l'utilisation de ce formalisme facilite la manipulation des modèles de services à l'exécution. La partie statique du méta-modèle de services est représentée avec les classes en jaune du diagramme de classes de la Figure 30.

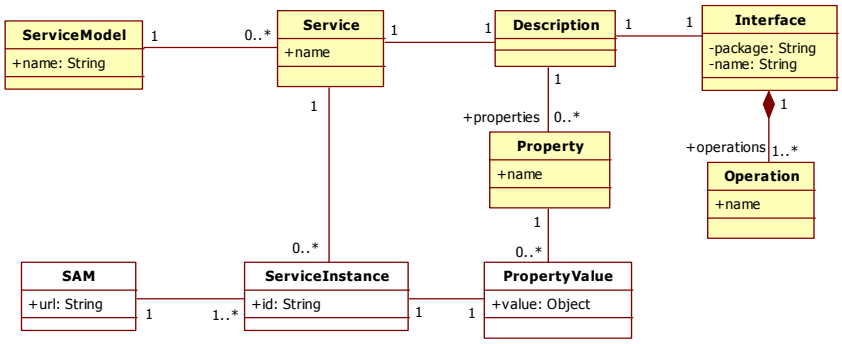

Figure 30. Méta-modèle de domaine de services.

Exemple d'un modèle de services

Un ensemble de services utilisables dans notre exemple de l'application d'alarme consiste en : un service capable de calculer la moyenne des températures (*Analysis*), un service de stockage pour garder l'historique des moyennes calculées (*Persistance*) et un service ayant la capacité d'agir sur les équipements de l'usine de production afin de régler les problèmes éventuels (*Controler*). Les interfaces Java, très simplifiées, servant à décrire ces services sont présentées dans la Figure 31.

```
public interface Analysis {
   public Average calculateAverage(TemperatureList tempertures);
}
public interface Persistance {
   public void save(Average average);
}
public interface Controler {
   public void doAction(Action action);
}
```

Figure 31. Modèle de services pour le système d'alarme.

L'interpréteur de services : SAM

L'interpréteur de services **SAM** (de son acronyme en anglais *Service Abstract Machine*) n'a pas comme objectif de fournir l'implémentation des services utilisés dans l'orchestration, mais d'offrir une machine capable d'assurer les mécanismes de base de l'approche à services. La SAM fournit des mécanismes pour la description, la découverte et la communication des services.

Une autre fonction de la machine abstraite de services est de cacher l'hétérogénéité des technologies utilisées pour l'implémentation de ces services. En fait, la machine SAM réifie les concepts de plates-formes à services sous-jacentes comme OSGi et les services Web. Ensuite, elle présente une vue homogène des services à ses clients pour enlever le souci de traiter avec la technologie concrète d'implémentation des services. Une description plus détaillée de cette machine sera présentée dans le chapitre suivant, dédié à la mise en œuvre du canevas FOCAS.

Les classes en blanc, *SAM*, *ServiceInstance* et *PropertyValue* sont utilisées par l'interpréteur SAM pour représenter l'état de l'exécution du domaine de services.

6.3 COMPOSITION DES DOMAINES POUR L'ORCHESTRATION

Nous allons présenter comment les trois domaines définis précédemment sont composés pour construire le domaine de l'orchestration de services. La composition de domaines est effectuée à trois niveaux, à savoir :

- Composition de méta-modèles. L'identification des relations entre les concepts de chaque méta-modèle doit être établie. Nous nous appuyons sur la définition explicite des méta-modèles afin d'établir les relations entre ces concepts.
- Composition de modèles. Les modèles créés dans chaque domaine doivent être intégrés afin de former des modèles d'orchestration « complets ». Cette composition doit être conforme à la spécification des relations entre les concepts des méta-modèles.
- Composition des interpréteurs. Les interpréteurs utilisés dans chaque domaine peuvent être récupérés et composés afin de construire l'interpréteur de l'orchestration de services.

Pour réaliser la composition de nos points de vue et construire notre système noyau d'orchestration nous avons suivi l'approche de composition de domaines exécutables proposée par [Veg05] et [Ngu08]. Cependant, cette approche générique ne profite pas de la connaissance des méta-modèles à composer et de leurs interpréteurs. Nous allons présenter l'approche de composition de domaines exécutables, et ensuite nous montrons comment elle a été utilisée et adaptée à nos besoins dans la composition de nos domaines.

6.3.1 La composition des domaines exécutables

Etant donné que les points de vue choisis pour l'orchestration ont été formalisés par le biais d'un méta-modèle et qu'un interpréteur existe pour chacun, pour créer notre système noyau d'orchestration nous avons repris les travaux de composition de domaines exécutables proposés par [Veg05] et [Ngu08]. Dans cette sous-section, nous allons introduire ces travaux afin de donner un contexte avant de présenter notre démarche de composition.

Le concept de domaine exécutable

Dans [Veg05], le concept de domaine est défini comme un ensemble de systèmes qui partagent un nombre de problèmes communs, et vraisemblablement une solution commune. C'est la vision de domaine utilisée dans les approches de lignes de produits [Har02]. Dans l'approche proposée par [Veg05], un domaine est formalisé en utilisant une approche de méta-modélisation. Une analyse du domaine donne comme résultat un méta-modèle (ou langage de

modélisation) contenant les concepts les plus pertinents du domaine par rapport à un objectif de modélisation.

Une fois le langage de modélisation spécifié, la notion d'exécution des modèles du domaine doit être définie. D'après l'approche IDM, un modèle peut être rendu exécutable par génération ou par interprétation. Nous considérons qu'une technique d'interprétation est plus adéquate pour fournir la sémantique d'exécution des modèles du domaine, car elle ne doit pas prendre en compte la dynamique d'évolution des plateformes d'exécution. C'est ainsi qu'un interpréteur (ou machine virtuelle d'exécution) est utilisé dans chaque domaine pour fournir la sémantique d'exécution du domaine comme proposé par [Thi98].

L'interpréteur du domaine fournit la sémantique d'exécution d'un modèle. Cette notion d'exécution spécifie les états possibles du système et sa dynamique d'évolution, c'est-à-dire l'exécution abstraite des modèles (vision utilisateur). Cependant, il est important de produire une vision de l'exécution basée sur la plate-forme d'exécution (perspective d'implémentation) connue comme l'exécution concrète du domaine. Cette exécution, doit se préoccuper des détails appartenant à l'espace de la solution, donc à l'infrastructure technologique supportant l'exécution du domaine. Ainsi, l'exécution concrète doit intégrer des composants et des applications existantes pour supporter l'exécution du domaine.

Architecture d'un domaine exécutable

L'architecture d'un domaine exécutable établit le lien nécessaire entre les deux visions d'exécution, l'exécution abstraite est concrète. Elle est héritée de la proposition de fédération de composants faite par [Vil03] et [Anh04] au sein de notre équipe de recherche pour faire face au problème d'intégration de composants hétérogènes. L'architecture est divisée en trois couches :

- ***La couche des outils*** qui contient les composants à intégrer. Différents types de composants peuvent être utilisés tels que des COTS, des systèmes patrimoniaux, des librairies, etc.
- ***La couche conceptuelle*** qui est composée de l'interpréteur du domaine et les modèles à exécuter. L'interpréteur pilote l'exécution des composants dans la couche des outils. Un contrat de coordination définit le comportement qui doit être fourni par les outils (rôles) et exprime comment les outils se comportent par rapport à l'évolution de l'exécution des modèles.
- ***La couche de médiation*** est chargée de résoudre les problèmes d'hétérogénéité des outils composés. Des adaptateurs sont inclus afin que les outils implémentent les rôles requis pour participer dans la composition.

L'architecture d'un domaine exécutable est schématisée dans la Figure 32.

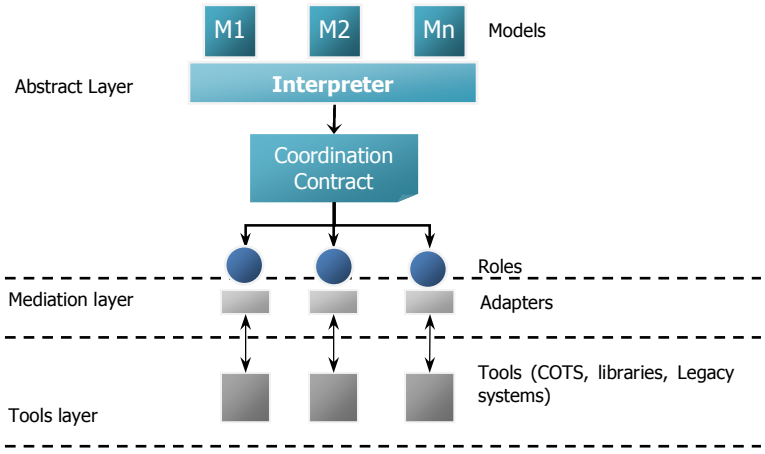

Figure 32. Architecture de la fédération des composants.

Le domaine comme unité de composition

L'approche de composition de domaines exécutables [Veg05] établit le domaine exécutable en tant qu'unité de composition. Des domaines exécutables plus vastes et complexes sont créés en utilisant un mécanisme de composition. Cette composition est faite au niveau des interpréteurs et des modèles (*la couche conceptuelle*) et n'est pas au niveau des outils supportant l'exécution concrète de domaine (*la couche des outils*), ce qui favorise l'évolution technologique du domaine. La composition de domaines exécutables est schématisée dans la Figure 33.

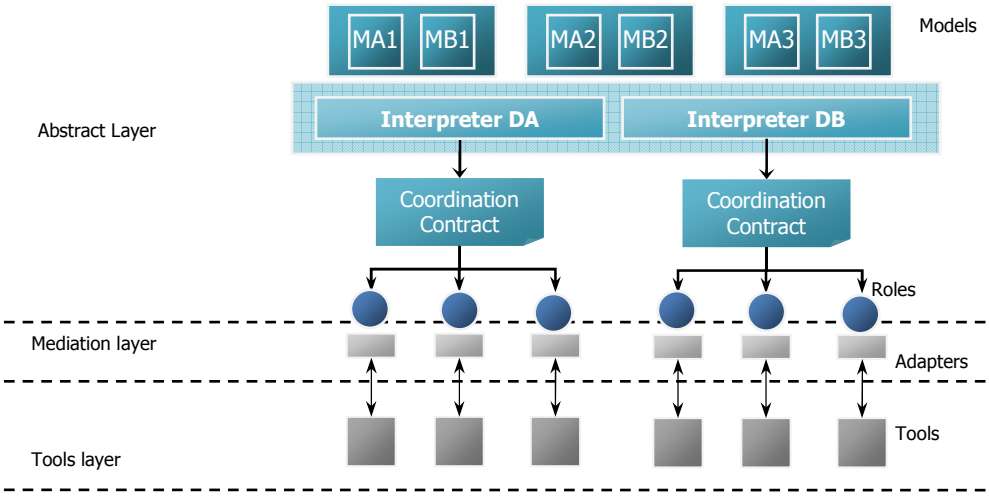

Figure 33. Le domaine comme unité de composition.

La composition au niveau conceptuel favorise la réutilisation des composants formant partie de l'espace de la solution, mais en plus, la réutilisation des autres artefacts du domaine peut être effectuée. Notamment, la réutilisation des langages de chaque domaine, des modèles exprimés dans ces langages, des outils de spécification de modèles et des interpréteurs.

Cependant, pour mettre en place une composition au niveau conceptuel, il faut la décliner sur trois mécanismes ; la composition de langages (méta-modèles), la composition de modèles et la composition des interpréteurs. Dans [Veg05] ces mécanismes sont définis avec une vision très pragmatique et *ad hoc ;* dans [Ngu08] ils ont été repris et définis explicitement. Par la suite nous allons présenter les trois axes comportant le mécanisme de composition de domaines exécutables.

Composition de méta-modèles

Dans la section précédente nous avons décrit la formalisation d'un domaine par le biais d'un méta-modèle et la mise en place d'une architecture structurée autour de l'interpréteur du domaine. Nous présentons maintenant comment ces unités de structuration sont composées afin de produire des domaines plus complexes à partir des domaines existants. L'objectif est de rester dans une vision de séparation de préoccupations pour améliorer la modularité et obtenir des taux de réutilisation plus importants.

D'abord, nous allons présenter comment les méta-modèles sont composés pour construire le méta-modèle composite du domaine. Chaque méta-modèle de base permet aux développeurs de décrire sous forme de modèle une vue particulaire du système à construire. La composition des méta-modèles décrit les relations entre ces différents points de vue. Une des propriétés de cette démarche est que les environnements de spécification de modèles de chaque domaine sont réutilisés, donc les développeurs utilisent ces outils pour spécifier le modèle correspondant à la vue du domaine.

L'approche suivie afin d'implémenter la composition de méta-modèles est l'établissement de relations (méta-liens) entre les concepts des méta-modèles composés. Ces relations sont binaires, donc un concept d'un méta-modèle est mis en relation avec seulement un concept de l'autre. Dans la Figure 34 un exemple de composition est présenté, une relation est établie entre le concept *Component* du méta-modèle *Development* et le concept *Unit* du méta-modèle *Deployment*.

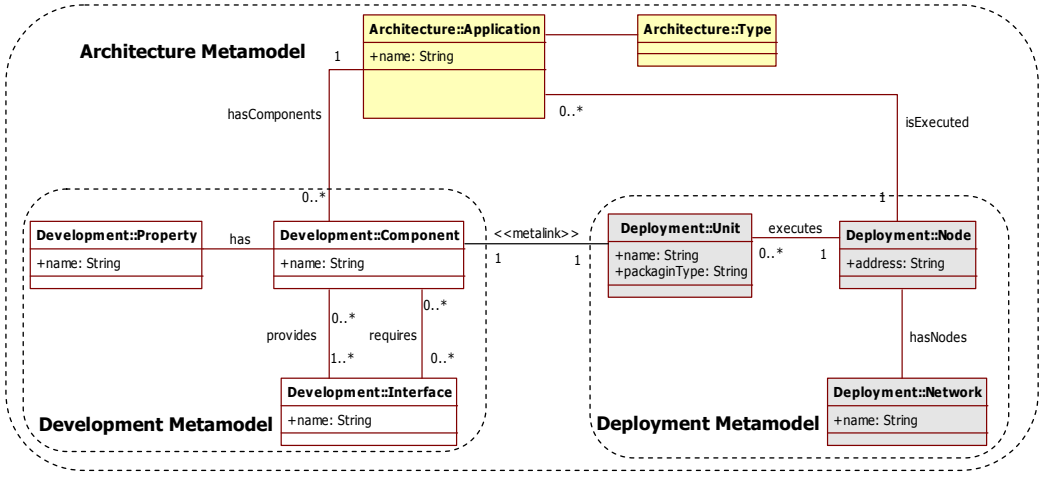

Figure 34. Composition de méta-modèles de domaines.

Il est possible que les concepts des méta-modèles composés ne soient pas suffisants pour exprimer tous les concepts du nouveau domaine. Les nouveaux concepts apparaissent dans le méta-modèle composite sont appelés concepts émergeants. Dans l'exemple les concepts *Application* et *Type* sont des concepts émergeants du méta-modèle composé *Architecture*. La sémantique des méta-liens n'est pas spécifiée, elle est matérialisée au moment de composer les interpréteurs des domaines ; cette composition sera présentée par la suite.

Composition de modèles

Un méta-modèle composite est un nouveau langage de modélisation permettant de spécifier le système depuis la perspective du domaine composite. Dans la démarche de composition de domaines exécutables, pour spécifier un modèle en utilisant le nouveau langage de modélisation, le concepteur doit s'intéresser seulement aux préoccupations associées à la composition, donc celles qui ne sont déjà considérées par les sous-domaines.

Par conséquent, un modèle conforme au nouveau méta-modèle est formé par un modèle importé pour chacun de sous-domaines, plus le modèle composite faisant référence aux modèles des sous-domaines et qui établie des liens explicites entre les modèles. Ces liens doivent être conformes aux méta-liens définis au niveau de la composition des méta-modèles. Des règles de cohérence peuvent être définies au niveau de la composition des méta-modèles permettant de valider qu'un modèle composite est bien formé. Cependant, ces règles ne peuvent pas être imposées à l'intérieur des modèles des sous-domaines.

La Figure 35 présente le modèle *Architecture-X* conforme au méta-modèle composite *Architecture* de l'exemple précédent. Les modèles *Development-Y* et *Deployment-Z* ont été importés pour constituer le nouveau modèle composite. Ensuite, un lien entre l'élément *SQL-Query* de type *Component* et l'élément *Query.jar* de type *Unit* a été spécifié pour lier des éléments des deux modèles. Ce lien doit être conforme au méta-lien défini dans la composition de méta-modèles. Pareillement, un lien a été spécifié entre l'élément *log4j* et l'élément *log4j.jar*. Puis, l'élément *db* du type émergent *Application* a été défini ainsi que ses relations avec les éléments de modèles de *Development-Y* et *Deployment-Z*. Finalement, le modèle composite *Architecture-X* est conformé par les modèles importés (*Development-Y* et *Deployment-Z)*, les

éléments conformes aux types émergents, les liens (*query*, *log*) conformes aux méta-liens et les relations entre l'élément *db* et les éléments de modèles importés.

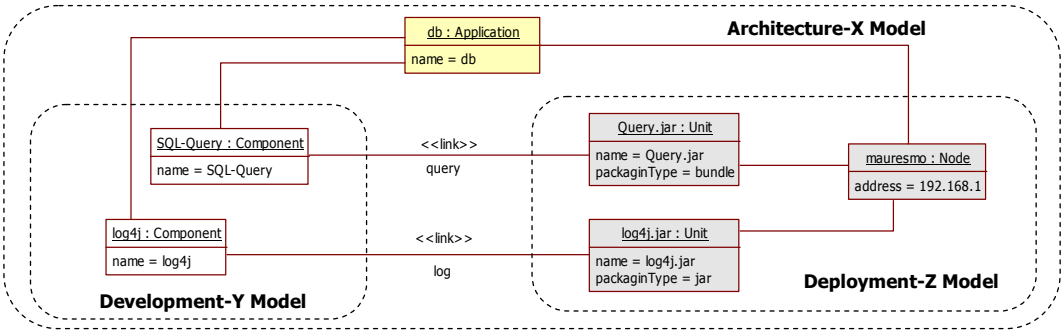

Figure 35. Composition de modèles de domaines.

Pour spécifier la composition de méta-modèles et de modèles nous avons repris le langage défini par [Ngu08] et l'outillage associé. L'outillage permet de spécifier les méta-liens entre les méta-modèles et ensuite, de générer un outil de composition de modèles respectant les contraintes exprimées avec les méta-liens.

Composition des interpréteurs

La composition des méta-modèles indique les relations existantes entre les concepts des domaines à composer. De la même façon que nous avons défini la notion d'exécution pour les domaines atomiques, nous devons définir la notion d'exécution pour les domaines composites. L'interpréteur du domaine composite est créé en réutilisant les interpréteurs de chaque sous-domaine sans les modifier. La composition consiste alors en une synchronisation des d'exécutions des interpréteurs des sous-domaines.

La composition des interpréteurs doit prendre en compte les méta-liens faits au moment de la composition des méta-modèles. Généralement, un méta-lien est traduit dans l'interpréteur par l'inclusion d'un champ dans une des classes faisant référence à une instance de l'autre classe. Par exemple, dans le cas du méta-modèle *Architecture*, un attribut de type *Component* (*Development*) peut être ajoute à la classe Unit (*Deployment*). Ainsi, cette traduction est potentiellement déduite de la spécification des méta-liens.

Cependant, dans un interpréteur, en plus des classes représentant la partie statique du domaine (les concepts du méta-modèle), il existe des classes servant à fournir sa dynamique d'exécution [BB01]. Les classes faisant partie de la partie dynamique du méta-modèle ne sont pas explicitées dans le méta-modèle structurel du domaine, par conséquent c'est la responsabilité du développeur de la composition de synchroniser leur dynamique d'exécution. Revenant sur notre exemple, lorsqu'une instance de *Unit* est éliminée dans l'interpréteur du domaine *Deployment*, une action doit être effectuée dans l'interpréteur du domaine *Development* (peut-être éliminer l'instance associée de *Component*).

Nous utilisons des techniques de programmation orientée aspect pour implémenter la composition des interpréteurs. Notamment, le langage *AspectJ* permet d'ajouter des membres dans des classes Java existantes et de modifier le code des méthodes, soit en ajoutent du comportement ou en le remplaçant. Techniquement, la composition des interpréteurs consiste à ajouter les classes qui correspondent aux concepts émergeants du méta-modèle composite. Ensuite, en utilisant la POA, le développeur ajoute des membres aux classes existantes et on modifie certaines méthodes pour synchroniser les états des deux machines.

Nous avons utilisé le cadre théorique de la composition de domaines exécutables pour fusionner les trois points de vue faisant partie de l'orchestration de services. Par contre, nous avons adapté les mécanismes de composition de domaines à nos outils et besoins particuliers. D'ailleurs, nous avons aussi implémenté nos propres mécanismes de composition des interpréteurs, car nous voulons profiter de la connaissance que nous avons sur les méta-modèles

et leurs interpréteurs. Cependant nous respectons les contraintes de la composition des domaines exécutables. Nous allons par la suite présenter comme la composition des nos domaines a été réalisée.

6.3.2 Composition du domaine de contrôle et de données

Le domaine de contrôle permet de spécifier une suite d'actions à exécuter sur un ensemble d'entité sans décrire la nature des entités. Le domaine de données permet d'exprimer la structure des entités sans dire comment elles seront manipulées. Chaque domaine permet de décrire le système depuis un point de vue autonome.

De l'analyse des relations existantes entre les deux domaines, nous pouvons faire certains choix pour réaliser la composition. Le concept de type de produit (*ProductType*) du domaine de contrôle sera **concrétisé** en utilisant le concept de type complexe de données (*ComplexType*) du domaine de données. Nous considérons que ces deux concepts expriment deux facettes différentes : d'une part le contrôle où le type de produit sert à différencier les produits et d'autre part les données où le type exprime la structure des données à manipuler.

De façon analogue, les concepts utilisés pour représenter l'état de l'exécution ont aussi des relations. Le concept de *ProductInstance* servant à déterminer une valeur pour chaque instance de *Product* dans le domaine du contrôle sera concrétise par le concept de *Revision* dans le domaine des données. Cette relation, n'est pas indépendante de celle définie auparavant ; à une instance d'un produit (*ProductInstance*) typé avec un type abstrait (*ProductType*) doit correspondre une instance d'une donnée (*Revision*) typé avec un type concret (*ComplexType*). La différence entre les deux types de relation est que la première est une relation entre les langages de modélisation tandis que la deuxième est une relation visible par les interpréteurs. Par la suite nous présentons la composition des méta-modèles et des interpréteurs.

Composition de méta-modèles

Au niveau des méta-modèles la composition consiste à définir un méta-lien entre le concept *ProductType* dans le domaine du contrôle et le concept *ComplexType* dans le domaine des données. La classe *ProductType* est un type de produit, mais il sert seulement à typer les produits sans donner de détails sur la structure du type. Cette classe sera concrétisée par la classe *ComplexType* qui va fournir la structure du type. Le méta-lien entre les deux classes est exprimé comme une association décorée avec le stéréotype *metalink* présentée dans la Figure 36.

La composition de ces méta-modèles permet d'avoir un nouveau langage, mais dans l'esprit de notre approche, nous allons réutiliser les modèles produits dans les sous-domaines contrôle et données afin de créer les modèles dans le nouveau langage.

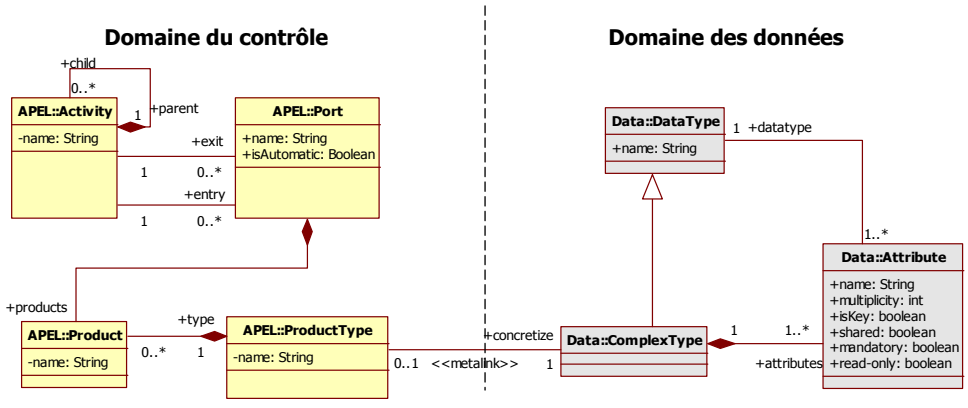

Figure 36. Composition de méta-modèles de contrôle et de données.

Cette composition est un bon exemple de l'utilisation du concept de domaine comme point de vue permettant ainsi de réaliser notre vision de séparation des préoccupations. En fait, le concepteur du modèle de contrôle se concentre sur la définition des activités et leur enchaînement sans s'inquiéter de la structure des entités manipulées, tandis que le concepteur du modèle de données spécifie la structure de données sans s'occuper de comment ces données seront traitées. Finalement, un intégrateur s'occupe de définir les liens entre les différents points de vue.

Composition de modèles

L'opération de composition de modèles consiste à importer un modèle existant pour chaque sous-domaine (contrôle et données), puis l'utilisateur doit effectuer la définition des liens entre les instances des concepts *ProductType* et *ComplexType* des modèles respectifs.

Nous allons illustrer la composition des modèles entre les domaines de contrôle et données en utilisant l'exemple du système d'alarme de l'usine de production présenté dans la section 6.2.1. Dans le modèle APEL du système d'alarme ont été définis les types de produits (*ProductType*) *Measure*, *MeasureList*, *Average*, et *Command*, d'autre part, dans le modèle de données ont été définies les types complexes de données (*ComplexType*) *Temperature*, *TemperatureList*, *Average* et *Action*. Une relation est spécifiée pour indiquer qu'un type de produit *Measure* sera concrétisé par un type complexe de donnée *Temperature*. De la même façon, les couples *MeasureList-TemperatureList*, *Average-Average* et *Command-Action* ont été spécifiés. Un modèle conforme au nouveau méta-modèle composite est formé alors par les deux modèles originaux plus la définition de liens présentés ici, les modèles originaux ne sont pas modifiés.

Dans la Figure 37, la composition des modèles est schématisée. La partie haute correspond à la définition de méta-liens entre les domaines, en bas, nous présentons les liens conformes à ces méta-liens dans le cas de notre exemple.

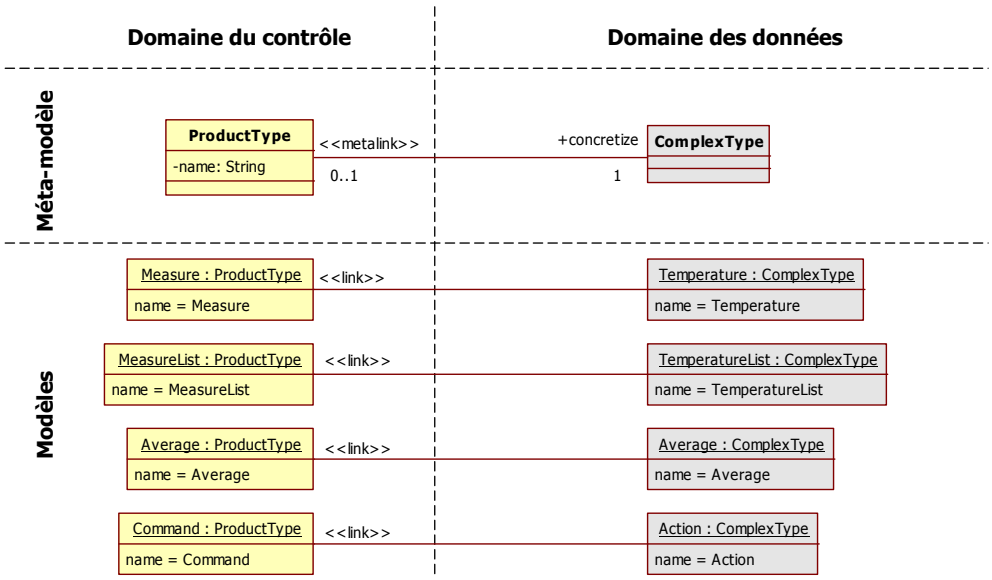

Figure 37. Composition des modèles du domaine de contrôle et de données.

Composition des interpréteurs

L'interpréteur d'exécution du domaine composite est construit en réutilisant le moteur d'exécution d'APEL (interpréteur de contrôle) et l'interpréteur de données sans les modifier. La composition est réalisée en deux parties : d'abord la composition des classes représentant les

concepts du langage de modélisation, ensuite la composition des classes servant à la représentation la dynamique d'exécution de l'interpréteur.

Au niveau des concepts appartenant aux langages de modélisation, le méta-lien exprimé entre *ProductType* et *ComplexType* dans la composition de méta-modèles va se matérialiser dans l'interpréteur par l'inclusion de l'attribut *dataType* de type *ComplexType* dans la classe *ProductType*. Cette composition peut être déduite à partir des définitions des méta-liens.

Pour implémenter la dynamique d'exécution du nouvel interpréteur, il est nécessaire de déterminer quels concepts seront synchronisés entre les deux interpréteurs de base. Nous avons pris la décision d'associer une révision (*Revision*) de l'interpréteur de données à chaque instance de produit (*ProductInstance*) du moteur d'APEL. La synchronisation de ces concepts requiert de :

- Inclure un attribut de type *Revision* dans la classe *ProductInstance* pour garder une référence vers la révision associée.
- Créer une nouvelle instance de la classe *Revision* chaque fois qu'une nouvelle instance de la classe *ProductInstance* est doit être créé.
- Créer une nouvelle instance de la classe *Revision* chaque fois qu'un clone d'un *ProductInstance* doit être créé, et l'initialiser avec les mêmes valeurs.

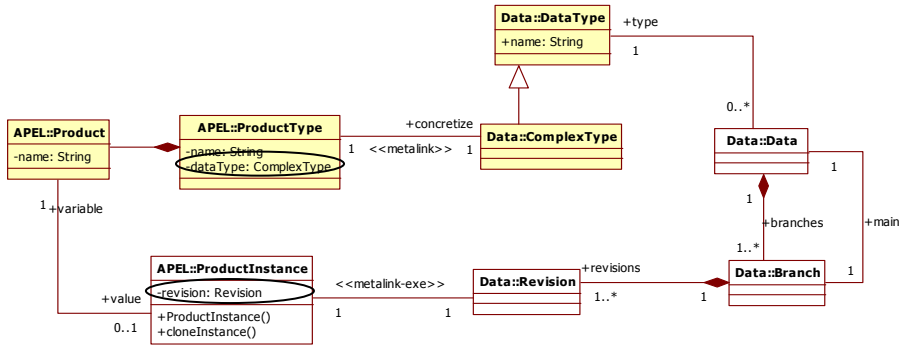

Figure 38. Méta-modèle composite de contrôle et données.

La matérialisation de cette synchronisation est faite par

- l'inclusion de l'attribut *revision* de type *Revision* dans la classe *ProductInstance*,
- la modification du code du constructeur de la classe pour ajouter la création d'une instance de *Revision*,
- l'assignation de cette instance à l'attribut *revision*,
- la modification de la méthode *cloneInstance()* pour ajouter la création d'une nouvelle instance de *Revision*,
- la copie des attributs de la révision originale dans la nouvelle révision.

La Figure 38 présente le méta-modèle composite de la machine d'exécution. Une association avec le stéréotype « metalink-exe » a été définie entre les classes appartenant à la partie dynamique du méta-modèle qui doivent être synchronisées.

```
package configuration;

import apel.*;
import data.*;

public aspect Associations {

   // Add the attribute dataType du type ComplexType into ProductType
   ComplexType ProductType.dataType;

   // Add the attribute revision du type Revision into ProductInstance
   Revision ProductInstance.revision;

   // Pointcut of cloneInstance method
   pointcut callCloneInstance(ProductInstance pi):
      call(ProductInstance ProductInstance.cloneInstance())
      && target(pi)
      && args();

  // Code executed when cloneInstance is invoked
  after(ProductInstance pi) returning: callCloneInstance(pi) {
      Revision rev = new Revision();
      rev.copyAttributes(pi.getRevision());
      ProductInstance productInstance = new ProductInstance();
      productInstance.setRevision(rev);
  }

}
```

Figure 39. Implémentation en AspectJ de la machine composite de contrôle et donnée.

Au contraire de la composition des concepts appartenant au langage de modélisation, la composition comportementale de l'interpréteur ne peut pas être déduite de la définition des méta-modèles, et doit être explicitement codée. Nous présentons dans la ci-dessus l'aspect développé en *AspectJ* pour implémenter les associations (méta-liens et méta-liens d'exécution) du méta-modèle composite. Le comportement associé à la méthode *cloneInstance()* est aussi inclus dans cet aspect.

6.3.3 Composition de domaines de contrôle et de services

Le modèle de contrôle n'indique pas comment les actions des activités seront réalisées, donc cette composition spécifie comment les services vont réaliser ces actions, nous appelons cette relation une **matérialisation** (*Grounding*) du modèle de contrôle. De l'analyse des interactions possibles certains choix ont été faits :

- l'action d'une activité peut être associée à une opération fournie par un service ;
- les activités ne réalisent pas toutes leurs actions en utilisant une opération d'un service. Les activités composites par exemple, définissent leurs actions en fonction de leurs sous-activités ;
- d'autres moyens peuvent être utilisés pour indiquer comment une activité réalise son action. Par exemple, un comportement peut être associé, ce qui ne correspond pas forcement à une opération d'un service ;

Basé sur ces choix, nous avons implémenté la composition comme suit.

Composition de méta-modèles

Le concept d'activité du méta-modèle de contrôle est associé au concept d'opération du méta-modèle de services, mais seules les activités atomiques seront associées aux opérations de services. En plus certaines activités atomiques peuvent effectuer son action autrement, donc la relation n'est pas obligatoire.

Les produits qui sont dans le port *desktop* ont une relation avec les paramètres de l'opération invoquée ; chaque produit doit correspondre à un des paramètres d'entrée ou de sortie de l'opération. La définition de méta-liens est présentée dans la Figure 40.

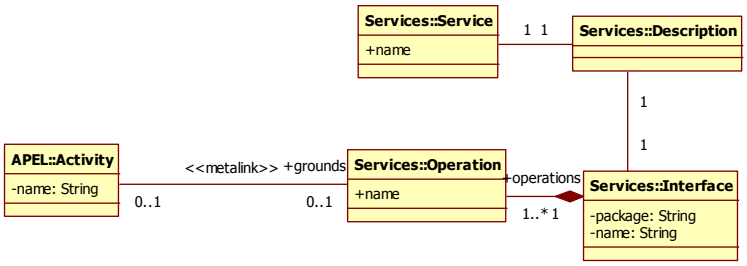

Figure 40. Composition de méta-modèles de contrôle et services.

Composition de modèles

Dans l'exemple du système d'alarme de l'usine de production, l'activité *Analysis* est associée avec l'opération *calculateAverage* du service *Analysis*, l'activité *Storage* avec l'opération *save* du service *Persistance* et finalement l'activité *Processing* avec l'opération *doAction* du service *Controler*. Les paramètres de ces opérations sont également associés avec les produits définis dans le port *desktop* de chaque activité. Une mise en correspondance automatique peut être réalisée à partir des types de données utilisés par les paramètres et ceux des produits, sinon le développeur doit explicitement indiquer les relations entre les paramètres et les produits de l'activité. Dans la Figure 41, la composition des modèles est schématisée : la partie haute correspond à la définition de méta-liens entre les domaines, en bas, nous présentons les liens conformes à ces méta-liens dans le cas de notre exemple.

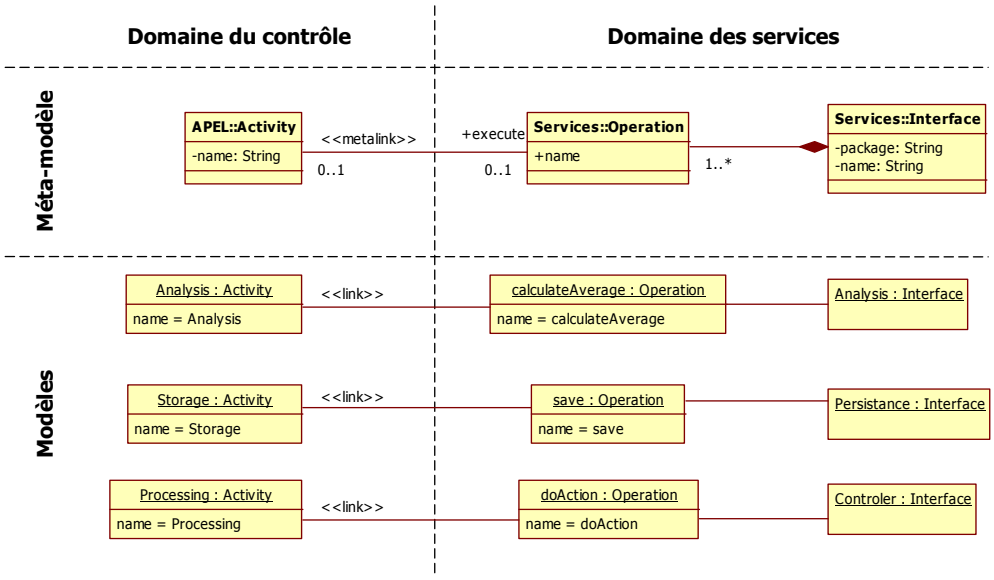

Figure 41. Composition des modèles des domaines du contrôle et des services.

Composition des interpréteurs

Etant donné que APEL a été conçu avec l'idée que la nature des actions effectuées par les activités ne doit pas être connue par le formalisme ni par le moteur d'exécution, nous avons ajouté un mécanisme simple afin de pouvoir spécifier l'action à réaliser par une activité. Ce mécanisme est basé sur le patron de conception observateur : une classe implémentant l'interface *ActivityObserver* peut souscrire pour écouter les changements d'état de l'instance

d'une activité. Lorsque l'activité passe à l'état actif, la méthode *doActive* de cette interface est invoquée, permettant à la classe de fournir l'implémentation de l'action de l'activité.

Nous allons profiter de la connaissance de l'implémentation de l'interpréteur d'APEL pour changer la façon d'implémenter la composition des interpréteurs. Plutôt qu'utiliser l'AOP pour composer les machines nous allons générer une classe implémentant l'observateur pour chaque paire activité-opération. L'observateur va être chargé d'écouter le moment de l'activation d'une activité (passage à l'état *active*). Une fois détectée cet événement, l'observateur pourra exécuter un ensemble d'actions afin d'invoquer l'opération du service associé. Les actions à exécuter consistent à réaliser une médiation de données si elle est nécessaire, c'est-à-dire, transformer les types de données utilisés par le *workflow* en des types utilisés par les services. Ensuite, une instance du service requis doit être trouvée ; l'observateur demande à la SAM qui retourne une instance de service implémentant l'interface du service requis. Puis l'observateur invoque l'opération en utilisant l'instance du service. Finalement si les données retournées par l'opération ont besoin d'une médiation, elle est réalisée et les données résultat sont mises dans le port de sortie de l'activité.

Une propriété de la machine SAM est qu'elle a la capacité de cacher l'hétérogénéité des implémentations des services. L'observateur n'a pas connaissance de la technologie utilisée par l'instance du service retournée. Du point de vue de l'observateur, tous les services sont des classes Java qui implémentant les interfaces utilisées pour décrire le service requis. La SAM est chargée alors d'implémenter le protocole utilisé pour communiquer avec la vraie instance du service, et de fournir un *proxy* à l'observateur. Actuellement les technologies OSGi, UPnP, DPWS, RFID, SNMP et services Web sont supportés par la SAM.

Avec ce mécanisme de composition, l'interpréteur d'APEL est découplé de la machine SAM, et c'est par l'intermédiaire des observateurs générés qu'ils sont composés. Autre propriété importante est que les dépendances vers les services utilisés par l'orchestration ne sont connues que par les observateurs, ce qui rend les interpréteurs indépendants des services à l'exécution. De plus, comme nous allons le présenter dans le chapitre d'implémentation les observateurs sont générés par notre environnement de spécification.

6.3.4 Synthèse

Nous avons présenté comment, en utilisant la technologie de composition de domaines exécutables, nous avons développé un système d'orchestration de services en se basant sur trois domaines basiques, le contrôle, les données et les services. Nous avons profité de la puissance des mécanismes de composition de domaines, et nous l'avons appliqué dans un domaine limité et bien connu : celui des applications orientées procédé (notamment l'orchestration).

La démarche utilisée préconise l'utilisation de trois mécanismes de base du génie logiciel : la séparation de préoccupations, le remonté du niveau d'abstraction et la réutilisation. La séparation de préoccupations est réalisée en définissant chaque préoccupation fonctionnelle comme un domaine tel que le contrôle, la gestion de données et de services, considérés comme divers points de vue au moment de spécifier une application orientée procédé.

La remontée de niveau d'abstraction est réalisée par l'utilisation d'une approche de méta-modélisation. Dans chaque domaine un méta-modèle (ou langage de modélisation) est défini. Autour de ce méta-modèle est structuré le concept de domaine exécutable ; un interpréteur fourni la sémantique d'exécution pour les modèles du point de vue. L'interpréteur est aussi utilisé pour piloter l'exécution concrète du domaine, en intégrant les différents outils appartenant au domaine.

La notion de composition de domaines exécutables permet d'intégrer les divers points de vue utilisés pour la spécification d'une application, et en plus, fourni le cadre pour avoir des taux de réutilisation importants. La réutilisation concerne d'abord les langages de modélisation (APEL, typage de données, spécification de services). La technique de composition ne modifie pas les méta-modèles originaux, ainsi les modèles existants peuvent aussi être réutilisés et de la

même façon les environnements utilisés pour spécifier ces modèles. Finalement, afin de spécifier la notion d'exécution pour le domaine composite, les interpréteurs de chaque sous-domaine sont réutilisés. Cette caractéristique permet de maintenir la sémantique d'exécution de chaque sous-domaine et d'ajouter la sémantique due à la composition. D'autre part cela permet de réaliser une composition conceptuelle du domaine, puisque la composition des interpréteurs entraîne la composition des outils qui supportent l'exécution concrète de chaque sous-domaine.

6.4 EXTENSIONS FONCTIONNELLES SUR L'ORCHESTRATION

La technique de composition de domaines exécutables a été utilisée pour construire un système de base pour l'orchestration de services avec des propriétés comme par exemple :

- l'indépendance de la technologie des services composés,
- l'utilisation des formalismes de haut niveau d'abstraction pour décrire l'orchestration,
- la séparation de préoccupations, car trois points de vues sont utilisés pour spécifier une application par orchestration,
- des taux importants de réutilisation, au niveau des langages, des modèles, des interpréteurs et des outils supportant la spécification et l'exécution concrète des domaines.

Cependant, d'autres besoins peuvent apparaître lorsque le système est utilisé dans des contextes plus spécifiques. Nous pouvons imaginer un système d'orchestration où des tâches doivent être réalisées par des humains, où bien un système de *workflow* traditionnel pour lequel, en plus des tâches réalisées par des humains, la gestion de documents est requise.

Notre approche utilise la technique de composition de domaines non seulement pour définir le système d'orchestration noyau, mais aussi pour réaliser des extensions lorsqu'il s'agit d'intégrer des domaines bien définis et structurés autour d'un méta-modèle à FOCAS. Nous identifions ces types d'extensions comme des extensions fonctionnelles, car un méta-modèle du domaine peut être défini en isolation, c'est-à-dire que des modèles conformes au méta-modèle peuvent être spécifies et exécutés sans avoir à être composé avec d'autres domaines.

Avec cette technique nous avons développé des spécialisations du système d'orchestration pour aborder les besoins des deux cas présentés auparavant. Un système d'orchestration avec le support tâches réalisées par des humains a été défini en incluant le domaine de ressources dans l'orchestration de base. Le méta-modèle composite ajoute du comportement dans la machine virtuelle pour effectuer la sélection de la ressource adéquate lorsqu'il s'agit d'une activité réalisée par un humain.

De la même façon, un système de *workflow* a été développé, ce système étend l'orchestration de base avec les domaines de ressources pour le support de tâches réalisées par des humains et le domaine de documents pour la gestion des documents numériques. Des comportements ont été ajoutés dans l'interpréteur afin de supporter l'envoi automatique des documents, l'exécution concrète du domaine fournissant les composants de communication assurant cette fonctionnalité.

Dans le diagramme de la Figure 42 les extensions faites sur notre système d'orchestration sont présentées. La structure du diagramme affiche les méta-modèles de chaque domaine qui sont utilisés comme élément autour duquel se structure la composition de domaines. Chaque extension est constituée de la composition du méta-modèle d'orchestration avec d'autres domaines. Plusieurs niveaux peuvent être identifiés dans la figure. En haut, nous plaçons des domaines assez génériques qui peuvent être utilisé dans différents types d'applications ; ils implémentent des fonctionnalités qui ne recouvrent pas entre elles, par conséquent ce sont des domaines indépendants et réutilisables. Ensuite, au milieu, nous plaçons notre système d'orchestration de base, qui fourni les fonctionnalités pour construire des applications basées sur la notion d'orchestration de services. Ce système est autonome et

extensible. En bas, des systèmes de support de procédés plus spécialisés, qui intègrent au système de base des nouvelles fonctionnalités fournis par d'autres domaines génériques.

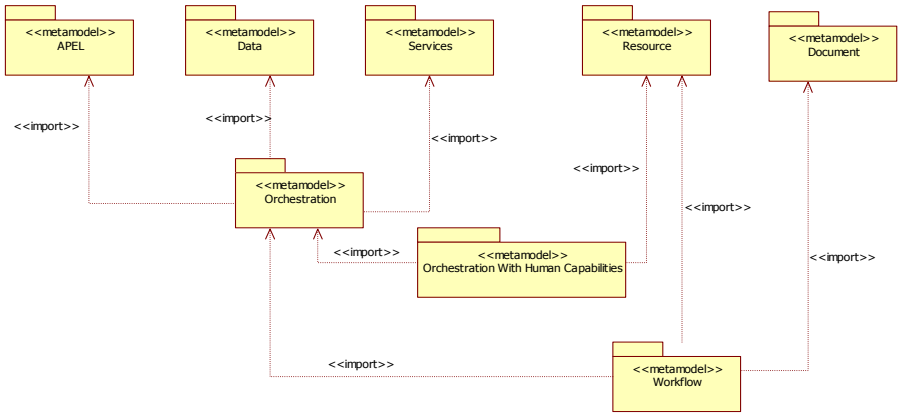

Figure 42. Applications orientées procédé spécialisées.

6.4.1 Synthèse

Le mécanisme de composition de domaines exécutables est utilisé pour réaliser notre canevas mais aussi pour l'étendre pour ajouter des capacités fonctionnelles. Comme pour les points de vue de base, l'ajout d'un nouveau point de vue se fait avec une composition avec le contrôle. Cette propriété facilite l'utilisation du mécanisme puisque il est plus simple faire une composition avec un domaine atomique (contrôle) qu'avec un domaine composite (orchestration).

Une autre propriété importante de notre mécanisme d'extension est qu'il s'applique aux formalismes mais aussi le *runtime* supportant l'exécution des applications, et à l'infrastructure existante. Les approches actuelles fournissent quelques mécanismes d'extension, mais généralement seulement au niveau de formalismes ce qui implique la nécessité de ré-implémenter les *runtimes* pour supporter l'exécution des applications qui utilisent ces extensions.

Afin de valider le mécanisme d'extension fonctionnelle, nous avons étendu notre système d'orchestration de base pour produire un système d'orchestration permettant que certaines tâches soient réalisées par des humains. Une autre extension a été effectuée pour construire un système de *workflow* traditionnel supportant, en plus, l'inclusion d'un domaine de gestion de documents.

Cependant, ce mécanisme s'applique mal aux propriétés non-fonctionnelles. Nous pensons que la difficulté vient du fait que ces propriétés traversent les domaines qui sont composés, et donc qu'elles ne peuvent pas être exprimées comme un autre domaine autonome et indépendant de la composition. Par la suite nous allons présenter les mécanismes d'extension ajoutés à notre canevas afin de supporter ce type de besoins.

6.5 EXTENSIONS NON-FONCTIONNELLES : LES ANNOTATIONS

Un défaut des canevas d'orchestration de services actuels est le manque de support des propriétés non-fonctionnelles. Cette carence est compréhensible si nous considérons que l'approche à services est récente ; les technologies de *middleware* se sont concentrées sur le support des mécanismes de base. En plus, certains canevas (et modèles) préconisent une séparation de préoccupations, laissant le support des aspects non-fonctionnels de l'orchestration à d'autres canevas ou à des développeurs.

Nous nous alignons sur ce dernier courant, en considérant que la séparation des préoccupations permet de gérer la complexité des applications logicielles. Par contre, notre

intérêt est de fournir des mécanismes d'extension dans FOCAS permettant aux développeurs d'inclure systématiquement le support des aspects non-fonctionnels dans les applications.

Nous avons d'abord essayé d'ajouter ce support des aspects non-fonctionnels en utilisant le mécanisme de composition de domaines exécutables. Malheureusement, les aspects non-fonctionnels possèdent certaines caractéristiques qui ne permettent pas d'appliquer cette technique. Parmi ces caractéristiques, nous pouvons mentionner :

- le fait que ces aspects peuvent traverser plusieurs domaines. Si nous prenons, l'aspect non-fonctionnel de la sécurité comme exemple, une propriété à assurer est la confidentialité des messages échangés entre l'orchestration et les services invoqués. Ainsi, le domaine de données est directement concerné, puisque ce sont les données qui devront être chiffrés pour assurer cette propriété, mais le domaine de services doit fournir des services capables de comprendre ces messages chiffrés, et le domaine de contrôle pour sa part, est chargé de connaître à quel moment chiffrer les messages.
- Il est difficile de définir un langage de modélisation pour des aspects non-fonctionnels en isolation. Si nous reprenons l'exemple de la sécurité, des concepts comme l'authentification, l'autorisation, la confidentialité font partie des concepts du méta-modèle de sécurité, mais ils n'ont pas un sens en eux-mêmes, ils doivent être projetés sur d'autres concepts. Par exemple, parler de la confidentialité en soi n'a guère de sens, par contre parler de la confidentialité d'une donnée bien particulière a un sens. L'approche des domaines exécutables exige la création d'un méta-modèle et d'une machine d'exécution associée, ce qui est très difficile de faire avec un méta-modèle ne contenant que des concepts abstraits sans comportement.
- Les aspects non-fonctionnels sont généralement dépendants de la plate-forme d'exécution qui va les supporter. Donc l'exécution concrète ne peut pas être facilement définie en s'abstrayant des outils utilisés par la plate-forme d'exécution.

Dans cette section, nous allons présenter un mécanisme d'extension de notre *canevas* en prenant en compte les aspects non-fonctionnels. Nous l'avons appelé ***extension par annotations***. Nous allons présenter comment ce mécanisme est appliqué au niveau des langages (méta-modèle), des modèles et de l'exécution.

6.5.1 Vision générale

La technique de composition de domaines exécutables ne peut être appliquée que pour des domaines pour lesquels il est possible de définir un langage de modélisation, de construire un interpréteur du langage et de spécifier des modèles indépendant d'autres domaines. Il existe des domaines pour lesquels il est envisageable de définir un ensemble de concepts du domaine, mais pour lesquels il n'est pas possible de construire un interpréteur ni de définir un modèle en isolation. Nous appelons de tels domaines des ***domaines abstraits***. Nous pouvons alors associer chaque aspect non-fonctionnel avec cette notion de domaine abstrait.

Le mécanisme d'extension pour supporter l'ajout d'aspects non-fonctionnels consiste à réaliser la composition d'un domaine abstrait représentant l'aspect non-fonctionnel avec le domaine concret de l'orchestration. Nous n'avons pas exploré la définition de cette composition dans un cadre général. L'idée est d'utiliser ce mécanisme pour ajouter des capacités non-fonctionnelles au système d'orchestration ou à des systèmes de support de procédés créés comme que des extensions du système d'orchestration.

L'ajout d'un aspect non-fonctionnel doit être défini aux trois niveaux utilisés pour la composition des domaines exécutables, donc au niveau du langage de modélisation, des modèles et de l'interpréteur d'orchestration, comme présentée dans la Figure 43. L'idée est qu'au niveau des langages de modélisation, les relations entre les concepts de l'aspect non-fonctionnel et ceux du méta-modèle d'orchestration soient exprimées explicitement par des méta-liens.

Au niveau des modèles, une technique d'annotation est utilisée. Etant donné que des modèles de l'aspect non-fonctionnel ne peuvent pas être exprimés en isolation, les instances de ces concepts vont *annoter* des modèles d'orchestration pour indiquer à quelles instances des concepts de l'orchestration seront appliquées les propriétés non-fonctionnelles. Finalement, une technique de génération est utilisée afin de produire le code qui doit s'exécuter lorsque le modèle annoté s'exécute pour assurer les propriétés définies pour ce modèle.

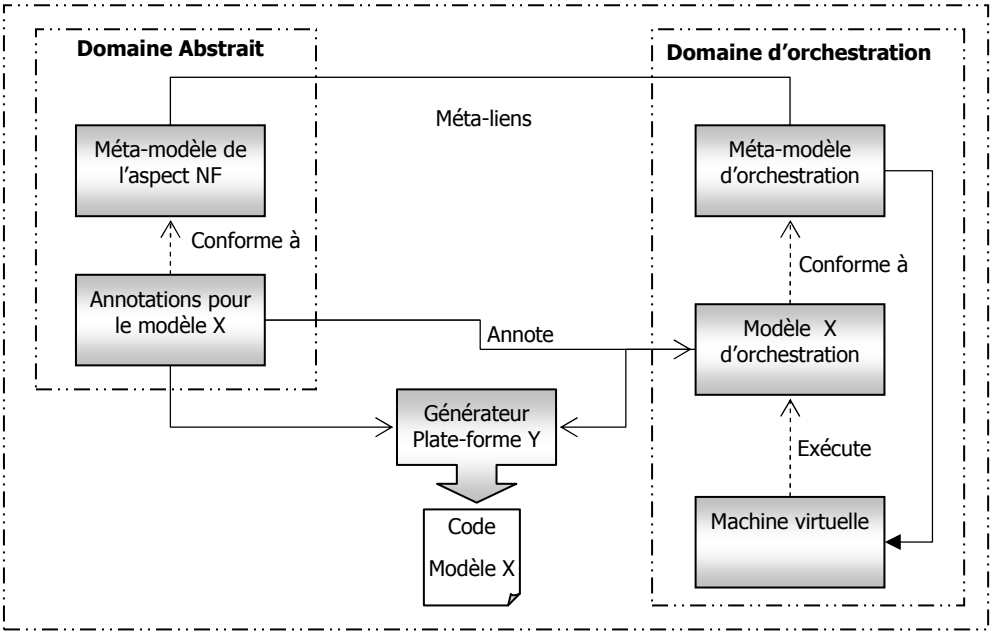

Figure 43. Vision générale de la composition par annotations.

6.5.2 Scénario d'ajout d'un aspect non-fonctionnel

Pour illustrer comment sont réalisées les extensions non-fonctionnelles dans FOCAS, nous allons développer un scénario d'extension qui ajoute l'aspect sécurité à une orchestration de services. L'aspect sécurité est lui-même un champ d'étude vaste et complexe ; nous ne voulons pas rentrer dans les détails mais utiliser ce scénario intuitivement afin d'illustrer comment la composition est réalisée.

Dans ce scénario trois propriétés doivent être assurées lorsqu'une orchestration de services est exécutée à savoir : l'authentification, la confidentialité et l'intégrité. L'authentification assure que les services sont fournis seulement à ceux qui sont autorisés. La confidentialité assure que les messages sont accessibles seulement pour ceux qui ont droit de les lire, tandis que l'intégrité indique que les messages ne subissent pas de modifications lors de leur transmission.

Nous allons reprendre l'exemple du système d'alarme de l'usine de production, pour lui ajouter des propriétés de sécurité. Pour cette orchestration, lorsque le service permettant de calculer la moyenne de température est invoqué, les messages échangés avec ce service doivent rester confidentiels. Puis, à l'invocation du service de stockage de données, on doit assurer l'intégrité de données. Finalement, le service qui réalise des actions sur les équipements de l'usine de production doit être utilisé seulement par les applications authentifiées par le service.

6.5.3 Composition de méta-modèles

Le premier pas pour la création d'un domaine abstrait consiste à définir les concepts qui font partie de l'aspect non-fonctionnel. Lorsque les concepts du domaine abstrait sont matérialisés dans un méta-modèle, des méta-liens sont définis entre les concepts du domaine

abstrait et ceux du domaine d'orchestration. Ces méta-liens sont exprimés de la même façon que dans la composition de domaines exécutables. Le méta-lien exprime une relation entre les concepts des deux domaines, mais la sémantique de la relation est fournie par le générateur de code. D'autre part le méta-lien exprime une contrainte syntaxique, elle indique quels concepts du domaine abstrait peuvent annoter quels concepts du domaine de l'orchestration.

Pour notre scénario d'extension avec la sécurité, les concepts d'authentification, de confidentialité et d'intégrité ont été définis. Ensuite, un méta-lien est créé entre le concept d'authentification du méta-modèle de sécurité et le concept d'activité du méta-modèle de contrôle. Il indique que lorsqu'une activité devient active et qu'il existe une opération d'un service associé à l'activité, l'orchestration doit s'authentifier auprès du service avant d'invoquer ses opérations. Un autre méta-lien est défini entre le concept de confidentialité et le concept de produit. Cette relation contient une sémantique indiquant que lorsqu'un service est invoqué, les données marquées par une annotation de ce type doivent avoir la propriété d'intégrité. Finalement, de façon équivalente, une relation est spécifiée entre le concept de confidentialité côté sécurité et celui de produit côté contrôle. La définition de ces méta-liens est présentée dans la Figure 44, les classes en gris correspondent au domaine abstrait de la sécurité alors que celles en jaune correspondent au domaine de contrôle.

Le domaine de contrôle possède la particularité d'être le domaine autour duquel se structurent les autres méta-modèles (données et services) au moment de créer le domaine composite de l'orchestration. Cette particularité peut s'expliquer car le méta-modèle de contrôle contient des concepts servant à spécifier les méta-liens avec d'autres méta-modèles. En plus, si nous prenons l'exécution d'une orchestration, la partie contrôle exprime la logique d'évolution de l'exécution, c'est-à-dire que le moteur d'APEL apporte une sémantique proactive à l'exécution pendant que les interpréteurs de données et services ont une sémantique réactive, ce qui fait qu'à l'exécution le moteur APEL soit aussi placée au centre de la composition.

Nous tirons profit de cette caractéristique car elle nous permet d'utiliser le méta-modèle de contrôle à la place du méta-modèle composite d'orchestration au moment de définir les méta-liens. Ce choix s'explique puisque la définition d'un modèle d'un domaine abstrait comme la sécurité par exemple, doit se faire sur les concepts d'un modèle concret. Donc afin de supporter les annotations, des extensions sur les environnements de spécification sont nécessaires. Il est moins complexe de faire ces extensions seulement sur l'environnement de spécification des modèles de contrôle, et ensuite appliquer les annotations sur les concepts d'autres domaines. Par exemple, si dans un modèle de contrôle, un produit (*APEL::Product*) est annoté avec une annotation d'intégrité, au moment de l'exécution ce sera l'instance de la donnée (*Data::Revision*) du domaine de données qui devra respecter cette propriété.

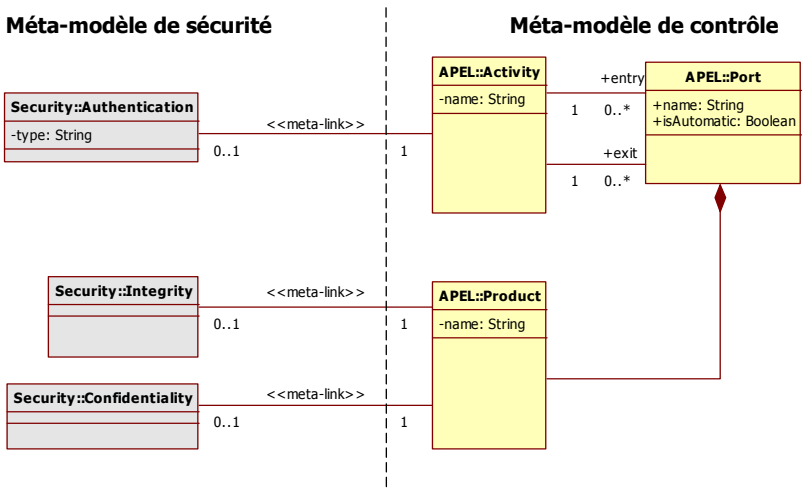

Figure 44. Composition de l'aspect sécurité avec le domaine d'orchestration.

6.5.4 Annotations sur les modèles

Au niveau des modèles, des instances de concepts du domaine abstrait vont annoter des instances des concepts du modèle de contrôle. Les annotations doivent être conformes aux méta-liens définis dans la phase de composition de méta-modèles.

Dans notre scénario, une instance du concept *Integrity* annote le produit *average* qui sera utilisé par l'activité *Storage* lors de l'invocation de l'opération *Save* du service *Persistence*. Une instance du concept *Confidentiality* annote le produit *val* qui sera utilisé par l'activité *Analysis* pour l'invocation de l'opération *Average* du service *Analysis*. Finalement, l'activité *Processing* est annotée avec une instance du concept *Authentication*. Cette annotation indique que lorsque l'activité invoque l'opération *DoAction* du service *Controler*, l'orchestration doit utiliser des certificats pour s'authentifier auprès du service avant de le consommer. Dans la Figure 45 les annotations faites sur le modèle APEL du système d'alarme sont présentées.

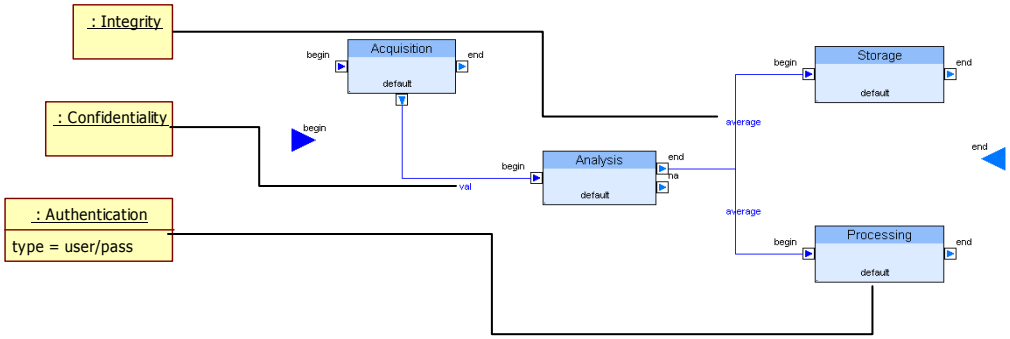

Figure 45. Annotations de sécurité sur le système d'alarme de l'usine de production.

Les annotations ne sont pas mélangées avec le modèle de contrôle, un modèle faisant référence au modèle APEL stocke les propriétés. De cette façon, différentes propriétés peuvent être appliqués sur le même modèle ce qui augmente la possibilité de réutilisation du modèle. Par contre, les annotations, elles, sont couplées au modèle de contrôle quelles annotent.

6.5.5 Générateur de code

La plate-forme supportant l'exécution des aspects non-fonctionnels est composée de la machine d'exécution d'orchestration et d'une plate-forme spécifique pour l'aspect non-fonctionnel. La responsabilité du générateur de code consiste alors à produire le code qui sera exécuté par la plate-forme spécifique de l'aspect non-fonctionnel, et de fournir aussi un code de coordination entre la machine d'exécution d'orchestration et le code généré.

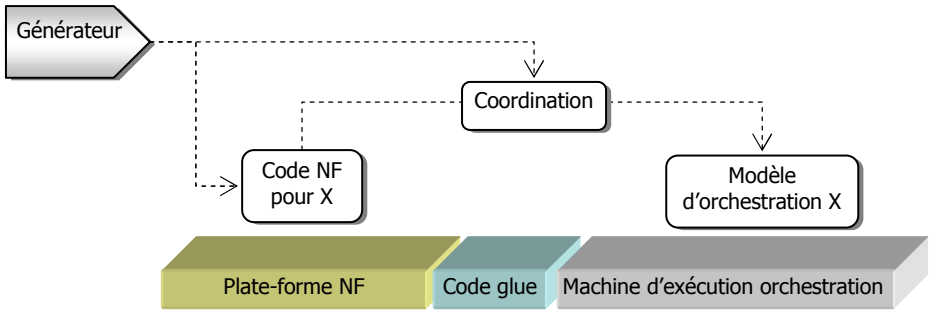

Figure 46. Générateur du code de support de l'aspect non-fonctionnel.

Dans notre scénario, la technologie de services utilisée est DPWS qui spécifie le support de l'approche SOA pour des dispositifs en utilisant principalement des standards de services Web. Le standard WS-Security spécifie des extensions sur le protocole SOAP pour ajouter des entêtes dans les messages permettant d'inclure des certificats pour l'authentification et des

informations concernant le cryptage de données. Nous avons choisi la plate-forme Axis qui fournit des composants supportant la communication SOAP et le standard WS-Security.

La plate-forme *Axis* a été composée avec notre machine virtuelle d'orchestration. L'unique opération à réaliser est d'inclure physiquement les composants pour qu'ils puissent être utilisés par le code généré. Le code généré consiste en des *stubs* Java pour l'invocation des services utilisés par l'orchestration, et des fichiers de configuration pour ajouter des informations de sécurité requises par les invocations. Le code de coordination consiste en des observateurs qui vont invoquer les *stubs* Axis lorsque les activités associées aux services s'activent.

Le code généré pour assurer les propriétés est complètement dépendant de la plate-forme choisie pour les supporter ; le générateur de code inclut implicitement la connaissance de l'implémentation de la plate-forme. Nous n'avons pas cherché à expliciter et extérioriser cette connaissance à cause de la nature très variée des différents aspects non-fonctionnels.

Notre approche préconise l'augmentation du niveau d'abstraction, ce qui permet d'exprimer les propriétés de l'aspect non-fonctionnel par un utilisateur qui n'est pas forcement un expert de la technologie utilisée pour implémenter le mécanisme qui va assurer la propriété. En plus, l'expression abstraite des propriétés de l'aspect non-fonctionnel offre la possibilité d'utiliser différentes plates-formes de support pour assurer les propriétés non-fonctionnelles.

6.5.6 Synthèse

Nous avons présenté, dans cette section, comment les aspects non-fonctionnels peuvent être intégrés dans notre *canevas* d'orchestration. Etant donné les particularités de ces aspects, il est difficile d'essayer de les exprimer comme des domaines exécutables ; par contre, une technique de composition en utilisant la notion de domaine abstrait est utilisée.

Le mécanisme de composition des domaines abstraits utilise une technique de génération de code pour inclure la notion d'exécution des aspects non-fonctionnels dans la machine virtuelle d'orchestration. A nouveau, selon de la nature de l'aspect non-fonctionnel, nous utilisons des techniques comme l'AOP pour tisser le code supportant l'aspect non fonctionnel avec celui de la machine d'exécution, ou bien la technique de l'implémentation d'un observateur afin de changer le comportement des actions associées aux activités.

Des travaux en cours dans notre équipe utilisent les mécanismes d'extension de FOCAS pour ajouter des propriétés comme la sécurité et le traçage des événements. Nous avons développé l'aspect non-fonctionnel de la distribution dans notre *canevas* avec l'objectif de supporter l'exécution repartie d'une orchestration. Un chapitre de cette thèse est dédié à l'illustration de cette extension que nous considérons comme un apport important à l'état de l'art dans le domaine de l'orchestration de services.

6.6 SYNTHESE

Dans ce chapitre, nous avons présenté FOCAS : un canevas pour la construction d'applications orientées procédé, notamment des orchestrations de services. Ce système possède plusieurs caractéristiques permettant de résoudre certaines des difficultés présentées dans le chapitre sur l'orchestration de services. Parmi, les apports les plus importants de FOCAS nous pouvons citer :

- l'indépendance de la technologie de services utilisés ;
- l'utilisation de divers points de vue pour la spécification d'une orchestration et ;
- la capacité d'extension du *canevas* afin de supporter des besoins spécifiques.

Nous avons choisi d'utiliser une démarche dirigée par les modèles pour construire notre système et pour permettre son extensibilité. Trois grands principes guident notre démarche : la séparation de préoccupations, l'augmentation du niveau d'abstraction et la réutilisation.

La séparation des préoccupations est effectuée en découpant la spécification d'une orchestration selon trois différents points de vue : le contrôle, les données et les services. Ce découpage permet aux concepteurs de l'application de se concentrer sur leur domaine d'expertise, puis à un intégrateur de produire une vision composite de l'orchestration.

L'augmentation du niveau d'abstraction offre la possibilité d'exprimer une orchestration en s'abstrayant des technologies des services utilisées pour son implémentation. L'utilisation de la technique de méta-modélisation est utilisée pour décrire les concepts pertinents de chaque point de vue, et un interpréteur donne la sémantique d'exécution des modèles.

La réutilisation est faite sur les divers artefacts de notre approche. D'abord les langages de modélisation et les modèles créés en utilisant ces méta-modèles. Ensuite, les interpréteurs supportant l'exécution abstraite de chaque point de vue et finalement, les outils, autant les outils spécification comme les outils supportant l'exécution concrète de chaque point de vue.

Cependant, nous considérons que l'apport le plus important de FOCAS est sa capacité d'extension. L'extensibilité nous permet de fournir une vision de famille d'applications orientées procédés, où une application spécifique peut être construite en faisant des extensions à notre système d'orchestration noyau. La technique de composition de domaines exécutables est utilisée comme mécanisme d'extension fonctionnelle, et la composition d'un domaine abstrait avec un domaine exécutable pour l'ajout des préoccupations non fonctionnelles. Les extensions sont effectuées au niveau de formalismes et des interpréteurs ce qui permet de réutiliser les outils supportant l'exécution concrète. Ceci est une propriété importante, si nous considérons que les rares mécanismes offerts par les formalismes actuels exigent la modification de *middleware* de support de l'exécution.

Finalement, nous avons constaté que la mise en place de notre approche exige la réalisation de tâches d'ingénierie assez lourdes, pour lesquelles est nécessaire de fournir des outils pouvant aider les développeurs dans leur réalisation. L'utilisation explicite de méta-modèles dans notre démarche nous offre la possibilité d'exploiter de façon systématique les modèles conformes à ces langages. Un des objectifs de cette thèse est aussi de fournir un ensemble d'outils servant à assister l'ingénierie nécessaire, pour deux types d'acteurs différents. D'une part ceux qui sont intéressés dans l'extension de *canevas* pour ajouter des fonctionnalités, et d'autre part les utilisateurs de FOCAS pour construire des applications orientées procédés. Dans le chapitre suivant, la mise en œuvre de ces outils est présentée.

7. MISE EN ŒUVRE DU CANEVAS FOCAS

Comme dans la plupart des technologies de *middleware*, notre canevas FOCAS est composé des trois éléments basiques : un environnement de spécification, un *runtime* et des outils de monitoring. Le chapitre sur l'approche a présenté essentiellement la partie conceptuelle de notre démarche ; ce chapitre montre la mise en œuvre du *runtime* et de l'environnement de spécification de FOCAS.

Pour décrire la mise en œuvre du canevas FOCAS, nous avons divisé ce chapitre en quatre sections. La première section, explique le *runtime* du canevas selon deux perspectives : celle de l'utilisateur qui développe des applications orientées procédé et celle de l'expert qui étend le canevas. La deuxième section, décrit l'environnement de spécification tel qu'il est perçu par le développeur d'applications. La troisième section montre l'implémentation de l'environnement de spécification et comment sont réalisées les extensions. La dernière section est chargée de montrer les mécanismes d'extension de l'environnement.

7.1 *RUNTIME* DU CANEVAS FOCAS

Dans cette section, nous présentons le *runtime* dédié à l'exécution des orchestrations de services. D'abord, nous montrons l'architecture mise en place pour les orchestrations de services selon la perspective du développeur. Ensuite, nous nous focalisons sur l'architecture du *runtime* de notre canevas en montrant comment est supportée l'exécution des orchestrations.

7.1.1 Architecture d'une orchestration

L'architecture d'une orchestration est divisée en trois couches. La couche la plus haute est connue comme l'orchestration abstraite. Dans cette couche, une orchestration de services est exprimée comme la composition des trois modèles : contrôle, données et services. La description abstraite exprime la logique de l'orchestration sans indiquer comment elle sera implémentée. Contrairement à la plupart des modèles actuels d'orchestration, aucune hypothèse n'est faite sur les services concrets qui vont fournir la fonctionnalité de l'orchestration : la dépendance envers la technologie utilisée pour l'implémentation des services à orchestrer est éliminée. La plupart des approches d'orchestration existantes supportent une seule technologie : les services Web. Certains travaux essayent d'enlever la dépendance envers cette technologie, par exemple BPEL-Light [NvLKL07] pour le langage WS-BPEL et JOpera [PHA06].

La couche la plus basse de l'architecture est celle des services concrets. Elle assure la disponibilité des services qui vont fournir la fonctionnalité requise pour l'orchestration. Les services concrets sont fournis par des tiers et sont créés indépendamment de l'orchestration. Dans cette couche les services peuvent être implémentés en utilisant différentes technologies (OSGi, UPnP, DPWS, services Web, etc).

Etant donné que les services concrets sont mis à disposition par des tiers, ils ne sont pas forcement compatibles entre eux et avec l'orchestration. Le découplage entre l'expression des besoins (services abstraits dans le modèle de service) et la réalisation de la fonctionnalité (services concrets) permet non seulement de faire abstraction de la technologie

d'implémentation des services, mais offre aussi la possibilité de résoudre certains des problèmes de comptabilité. Nous avons classés les problèmes de compatibilité en trois types :

- lexical : les services abstraits et concrets expriment les mêmes concepts en utilisant différents termes. Par exemple, si les noms des types de données utilisés pour les paramètres d'un service concret diffèrent de ceux utilisés par les services abstraits.
- syntaxique : la description du service abstrait n'est pas compatible avec celle du service concret, même s'ils fournissent la même fonctionnalité. Par exemple, si une opération d'un service concret fourni la fonctionnalité requise, mais les paramètres ne sont pas dans le même ordre que dans la définition du service concret.
- sémantique : il n'existe pas de service concret qui fournisse la fonctionnalité requise, mais cette fonctionnalité peut être définie en termes d'un ensemble de services concrets. Par exemple, si une opération d'un service peut être fournie par l'invocation d'une suite d'opérations d'un ensemble de services concrets

Afin de pouvoir résoudre ces différents types d'incompatibilités, nous avons inclus une couche d'adaptation qui se trouve entre la couche d'orchestration abstraite et celle des services concrets. Notre intérêt n'est pas de définir un schéma de médiation [WG97] mais d'offrir la possibilité d'inclure ces mécanismes dans l'architecture de notre canevas. Dans la section dédiée à l'environnement de spécification, nous allons présenter comment, pour des cas précis, une implémentation simple des médiateurs peut être générée.

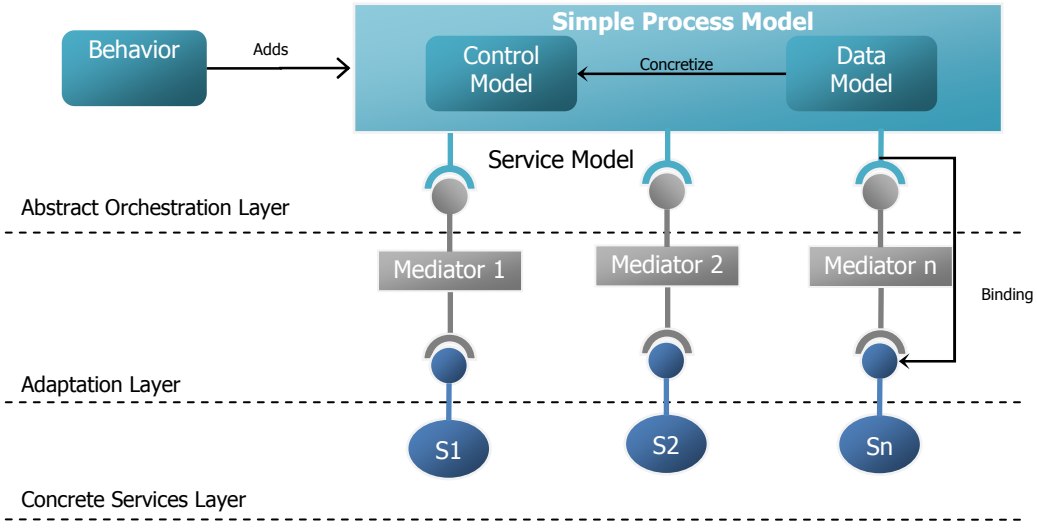

Figure 47. Architecture des orchestrations.

La Figure 47 schématise l'architecture des orchestrations de services dans notre approche. Le développeur de l'orchestration doit spécifier la logique métier dans l'**orchestration abstraite** située dans la partie haute de la figure. L'orchestration abstraite est composée du **modèle simple de procédé** et du modèle de services. Nous appelons modèle simple de procédé le résultat de la composition des modèles de contrôle et de données, la relation existant entre ces deux modèles est appelée **concrétisation** (*Concretization*). La relation existant entre le modèle de contrôle et celui de services est appelée **matérialisation** (*Grounding*), car elle permet de donner un sens à l'exécution des activités. La création de l'orchestration abstraite consiste à spécifier chacun des modèles et à définir les relations entre eux.

Une propriété intéressante de cette architecture est la capacité d'exécuter le modèle simple de procédé. Grâce à cette propriété, il est possible de simuler l'exécution d'une orchestration et de la déboguer sans avoir besoin d'une implémentation des services concrets. Par conséquent, pour la construction d'une orchestration, une approche *top-down* peut être

utilisée, à savoir un procédé exprime ce qu'il va faire sans indiquer comment il va le faire, et ensuite les moyens pour le mettre en œuvre sont indiqués.

Au moment de concrétiser l'exécution d'une orchestration, c'est-à-dire de faire une mise en correspondance entre les services abstraits et les services concrets, la relation de **liaison** (*Binding*) apparaît dans l'architecture. Cette liaison indique quel service concret fourni la fonctionnalité spécifiée par le biais d'un service abstrait. Le type de liaison entre un service abstrait et un service concret dépend de la nature des services utilisés et du moment de la réalisation de la liaison.

Si pendant la liaison l'interface fournie par un service concret ne correspond pas au service abstrait, un médiateur peut être inclus dans l'architecture pour résoudre l'incompatibilité. Le médiateur fournit une interface compatible avec l'interface requise pour l'orchestration abstraite et consomme l'interface fourni par le service concret.

Ajout de comportement à un modèle de procédé simple

En plus de la capacité d'exécuter un modèle simple de procédé, il est possible d'ajouter des comportements associés aux activités de ce modèle. Cette propriété permet de donner une sémantique d'exécution aux activités, ce qui évite que tous les calculs nécessaires dans une application orientée procédé doivent être réalisés par des services.

Le mécanisme d'ajout de comportement dans un modèle simple de procédé est basé sur une interface permettant d'écouter les changements d'états des instances d'activités et de réagir à ces événements. Afin d'utiliser cette interface le développeur doit fournir une classe qui implémente l'interface Java *ActivityObserver* du canevas FOCAS et d'ajouter le code exprimant le comportement de l'activité dans la méthode *doActive*. Ensuite, cette classe doit signaler à l'interpréteur qu'elle souhaite recevoir les événements de l'activité dont elle étend le comportement. Notre environnement de spécification génère cette classe et le code qui l'enregistrement auprès de l'interpréteur. Le développeur doit « seulement » écrire le code du comportement.

Nous utilisons l'ajout de comportement de deux façons dans notre canevas :

- l'ajout direct de comportement pour une activité d'un modèle simple de procédé. L'API ***ProcessManager*** de haut niveau pour la manipulation des instances de procédé simple est aussi fournie par le canevas afin d'offrir aux utilisateurs un moyen simple d'exprimer le comportement des activités.
- La définition de patrons d'activités. Un patron est un type d'activité prédéfinie avec un patron de code servant à générer une classe qui implémente son comportement, la classe doit implémenter l'interface Java *ActivityObserver*. Le mécanisme de patrons d'activités permet de spécifier une bibliothèque de types d'activités réutilisables.

Dans la Figure 47, le composant **comportement** (*Behavior*) contient l'ensemble des classes implémentées pour ajouter des comportements aux activités de l'orchestration abstraite. Le composant comportement est placé dans la couche abstraite puisqu'il n'utilise que les concepts du modèle simple de procédé.

Dans la Figure 48, un exemple d'ajout de comportement est présenté. Le code de la classe *ShowMessageObserver* ajoute du comportement pour une activité nommée *ShowMessage*. Dans la méthode *doActive* est spécifié que lorsqu'une instance de cette activité arrive à l'état activé, une fenêtre s'affiche pour demander à l'utilisateur si l'exécution du procédé doit continuer. En cas d'une réponse positive l'activité doit finir l'exécution en utilisant son port *end*. La classe *ShowMessageObserver* s'appuie sur l'API *ProcessManager* pour réaliser ses fonctions.

```
public class ShowMessageObserver implements ActivityObserver {

   public void doActive(Activity activity) {
      // A message dialog is showed
      boolean resume = MessageDialog.openQuestion(null, "Message", "Resume
                        Execution?");
      // If user clicks OK the activity terminates
      if (resume)
         ProcessManager.terminateActivity(activity, "end");
   }

}
```

Figure 48. Code de comportement pour une activité d'un modèle de procédé simple.

7.1.2 Architecture du *runtime* du canevas FOCAS

Dans cette section, nous allons présenter l'architecture du *runtime* d'exécution du canevas FOCAS. Nous avons montré dans le chapitre précédant que notre interpréteur d'orchestration est construit par composition des interpréteurs des trois domaines contrôle, données et service. Nous allons concrétiser cette idée en décrivant l'architecture de notre *runtime*.

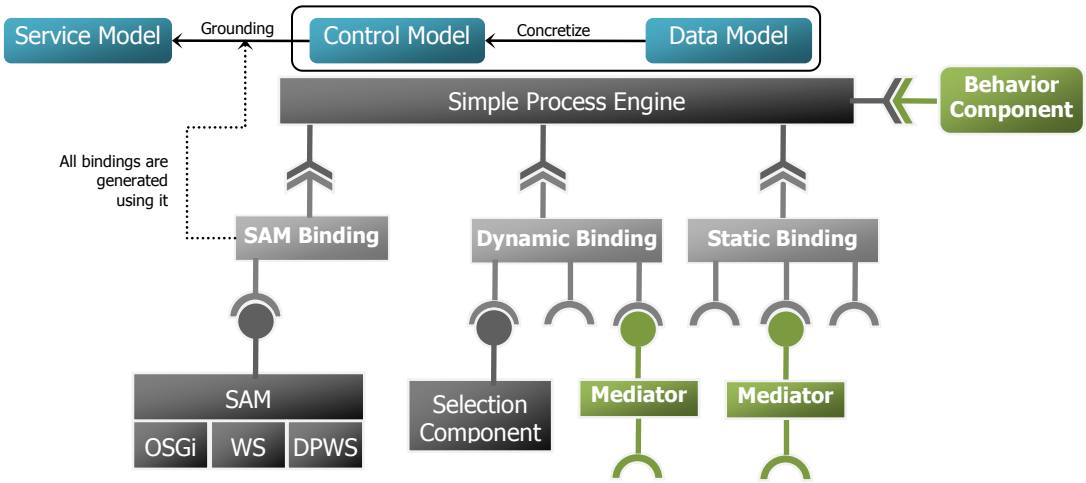

Figure 49. Architecture du runtime du canevas FOCAS.

Dans la Figure 49, l'architecture du *runtime* du canevas FOCAS est schématisée. Nous pouvons distinguer quatre types de composants :

- les composants fournis par le développeur des orchestrations. Spécifiquement le composant d'orchestration abstraite qui est spécifié par des modèles de chacun des trois points de vue et leurs relations.
- les composants partiellement générés par l'environnement de spécification pour une orchestration particulière. Par exemple, les classes qui implémentent le comportement pour une orchestration sont générées par l'environnement et complétées par le développeur. Dans cette catégorie nous trouvons, le composant de comportement et les médiateurs.
- les composants complètement générés par l'environnement de spécification pour une orchestration particulière. Le composant de liaison (*Binding*) est généré à partir des modèles spécifiés par le développeur. La génération de ce composant dépend de choix techniques comme l'utilisation d'une liaison statique (déterminée à la conception) ou d'une liaison dynamique (déterminée à l'exécution).

- les composants génériques de notre canevas qui supportant l'exécution de toutes les applications orientées procédé. Dans cette catégorie nous trouvons le **moteur simple d'exécution de procédés** et la machine SAM.

Dans la section dédiée à l'architecture des orchestrations, les deux premiers types de composants ont été expliqués, maintenant nous allons nous concentrer sur les composants fournis par le canevas et les composants complètement générés qui ne sont pas de la responsabilité des développeurs d'orchestrations.

Le moteur simple d'exécution de procédés

Le composant central de l'architecture est le moteur simple d'exécution de procédés. Ce moteur est formé par la composition des interpréteurs de contrôle et de données. En plus de la composition des interpréteurs, nous allons composer les outils servant à donner la notion d'exécution concrète de chaque domaine. Cette dernière composition est réalisée de façon conceptuelle, c'est-à-dire que la composition des interpréteurs entraîne la composition des composants supportant l'exécution concrète de chaque domaine.

Pour l'exécution concrète du domaine de données, le domaine a besoin d'un composant capable de faire persister les données traitées. Différents composants peuvent fournir cette fonctionnalité selon les caractéristiques d'utilisation du domaine. La configuration du domaine de données, que nous avons utilisée, implémente la persistance en utilisant une base de données relationnelle.

Dans le cas du domaine de contrôle, pour son exécution concrète, deux rôles (selon la terminologie de composition des domaines exécutables) ont été définis. Le premier rôle spécifie un composant capable d'assurer la persistance des instances des modèles de contrôle en train de s'exécuter. L'autre rôle définit un composant permettant l'interaction des utilisateurs avec les instances des modèles de contrôle.

La persistance des instances permet au moteur d'exécution la reprise en cas de panne, une propriété essentielle pour les applications orientées procédé. Nous avons développé un composant qui fournit cette fonctionnalité. Le composant d'interaction permet la création d'instances, leurs interactions et le suivi de leurs changements d'état. Deux implémentations de ce rôle existent actuellement : le composant agenda qui est une application avec une GUI qui présente graphiquement les instances de procédé ; l'autre est une version Web de l'agenda.

Suivant l'approche de composition des domaines exécutables, chaque domaine définit un contrat de coordination indiquant comment doivent se comporter les composants qui implémentant les rôles définis. Ensuite, la technique de la programmation orientée aspect (AOP), est utilisée pour synchroniser l'état d'exécution de l'interpréteur avec la fonctionnalité fournie par les rôles définis.

Dans la Figure 50 est présentée l'architecture du moteur d'exécution de procédés simple. A gauche est présenté le domaine de données avec son contrat de coordination, le rôle défini et le composant qui l'implémente. A droite, le domaine de contrôle est montré, avec son contrat, les deux rôles et les deux composants. La composition est faite au niveau des interpréteurs, un code de synchronisation assure cette composition. Nous utilisons à nouveau la technique d'AOP, pour tisser le code de synchronisation avec celui des interpréteurs. Etant donné que nos interpréteurs sont implémentés en Java le langage *AspectJ* a été choisi comme technologie d'AOP.

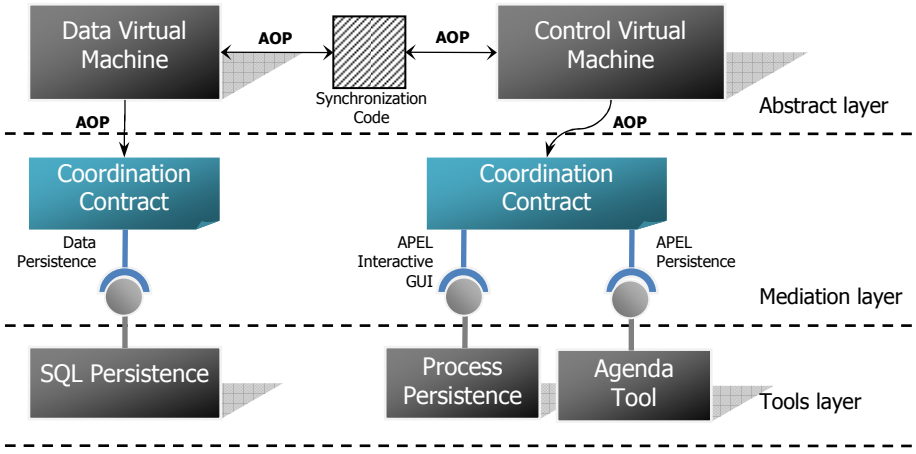

Figure 50. Architecture du moteur d'exécution de procédés simple.

La machine SAM

La machine abstraite à services SAM (de l'acronyme en anglais *Service Abstract Machine*) a deux responsabilités dans notre architecture. La première consiste à fournir des mécanismes de sélection de services, permettant, au composant de liaison de trouver les instances de services requis. Dans la machine SAM, une instance de service est recherchée en utilisant comme critères de sélection l'interface fonctionnelle (interface Java) et des contraintes exprimées en utilisant les propriétés du service [EDSV09].

L'autre responsabilité de la machine SAM est de cacher la technologie avec laquelle a été développée les services concrets utilisés pour l'orchestration. Cette fonctionnalité offre aux clients de la SAM la possibilité de récupérer une instance d'une classe Java comme instance d'un service requis. Etant donné que nous spécifions les services abstraits en utilisant des interfaces Java, la classe Java qui représente le service concret implémente l'interface Java avec laquelle le service a été décrit. De cette façon, la machine d'orchestration invoque la méthode pertinente sur une instance d'une classe Java, sans se soucier des protocoles de communication ce qui est responsabilité de la machine SAM.

La machine SAM est extensible, donc de nouvelles technologies à services peuvent être ajoutées, actuellement elle supporte les technologies OSGi, DPWS, UPnP et des services Web.

Le composant de liaison

Le canevas FOCAS supporte trois types de liaison des services :

- liaison statique. Si les instances des services concrets sont déterminées avant le déploiement, et tous les services concrets sont implémentés en utilisant la même technologie, une liaison statique peut-être utilisée.
- liaison dynamique homogène. Si les interfaces des services concrets sont connues avant le déploiement, mais les instances sont sélectionnées à l'exécution, et en plus tous les services sont implémentés en utilisant la même technologie, une liaison dynamique est utilisée.
- liaison dynamique hétérogène (SAM). Si les services concrets ne sont pas connus avant le déploiement et les services concrets sont implémentés en utilisant diverses technologies, une sélection et une liaison dynamique SAM sont utilisées.

Le composant de liaison est spécifique à chaque application ; il est complètement généré par l'environnement de spécification des orchestrations. Il dépend du choix du type de liaison. La responsabilité de ce composant est de trouver des instances des services requis par l'orchestration et d'invoquer les méthodes pertinentes.

Pour brancher le composant de liaison avec le moteur simple de procédés, nous utilisons la même interface d'événements utilisée par le composant de comportement. Ainsi, pour chaque activité avec une matérialisation (association entre une activité et une opération d'un service) une classe Java implémentant l'interface *ActivityObserver* est générée. Cette classe est chargée de trouver l'instance de service qui matérialise l'activité et d'invoquer la méthode correspondante. Par conséquent, une classe est générée pour chaque matérialisation du modèle d'orchestration. L'ensemble des classes générées correspond au composant de liaison de l'orchestration.

```
public class ProcessingObserver implements ActivityObserver {

   Machine machine; // Instance of SAM Machine
   Controler service = null; // Abstract service grounded for activity processing

   public void doActive(Activity activity) {
      ...
      // Instance of service to be found
      ServiceInstance myInstance = null;
      // Discovering component of SAM Machine
      ServiceInstanceBroker siBroker = null;

      // If an instance of service has not been found
      if (service==null) {
         // Initialisation of the discovery component
         siBroker = machine.getServiceInstanceBroker();
         // Retrieve of instances of services implementing the Controler interface
         Set instances =
siBroker.getServiceInstances("fr.lig.adele.demosec.Controler", null);
         // Selection of one instance
         myInstance = selectInstance(instances, null);
         if (myInstance!=null) {
            service = (Controler) myInstance.getServiceObject();
         }
      }
      // Invocation of operation
      if (service!=null)
         service.doAction(action);
   }
}
```

Figure 51. Liaison SAM pour l'activité *Processing* de l'application d'alarme.

Une classe implémentant une liaison dynamique SAM, demande au composant de sélection de la machine SAM de trouver une instance implémentant le service abstrait. Si l'instance est trouvée, elle est retournée et l'opération du service est invoquée. Par exemple, dans la Figure 51, la classe générée pour la liaison de l'activité *Processing* de notre système d'alarme est présentée. Cette classe récupère une instance du composant de découverte de services de la machine SAM (*ServiceInstanceBroker*), ensuite le composant de découverte retrouve toutes les instances de services implémentant l'interface *Controler* qui est le service abstrait associé à l'activité *Processing*, puis une sélection d'une instance du service parmi celles trouvées est effectuée. Finalement, l'opération *doAction* du service *Controler* est invoquée. Dans cet exemple le seul critère de découverte utilisé a été l'interface fonctionnelle du service.

Bien que l'utilisation d'une liaison dynamique SAM fournisse une abstraction des technologies utilisées dans l'implémentation des services, il est toujours possible d'implémenter un composant de liaison avec une connaissance précise de la technologie à services utilisée afin d'obtenir des performances plus importantes. C'est le cas des deux autres types de liaison supportés par notre architecture : la liaison dynamique homogène et la liaison statique.

La liaison dynamique homogène requiert la connaissance des interfaces des services concrets avant le déploiement d'une orchestration. Par contre, les instances des services concrets peuvent être trouvées à l'exécution. Pour sa part, la liaison statique, en plus de la connaissance des interfaces de services concrets, requiert la connaissance de la localisation des instances à utiliser. Dans notre canevas, nous avons inclus des composants de liaison dynamique et statique

pour la technologie des services Web. Des *proxies* pour invoquer les services sont automatiquement générés par notre environnement de spécification.

Dans la Figure 52, un exemple de liaison dynamique homogène est présenté. Le code d'une classe appartenant à un composant de liaison dynamique de services Web pour l'activité *Analyis* de notre système d'alarme est listé. Nous pouvons noter que dans ce cas l'interface du *proxy* de service concret (*AnalysisPortType*) est utilisée au lieu de l'interface du service abstrait (*Analysis*). Le composant *ServiceLocator* est chargé de découvrir et sélectionner une instance du service à utiliser, finalement l'invocation du service est fait par le biais du *proxy*.

```
public class AnalysisObserver implements ActivityObserver {

   // Interface du Proxy corresponding to Analysis Abstract Service
   AnalysisPortType service;

   // Discovery Component for Web Services
   AnalysisServiceLocator serviceLocator;

   public AnalysisObserver() {
      // Initialisation of the discovery component
      serviceLocator = new AnalysisServiceLocator();
   }

   public void doActive(Activity activity) {
      ...
      if (service==null) {
         // A instance of service is retrieved
         try {
            service = serviceLocator.getAnalysisPort();
         } catch (ServiceException e) { ... }
      }
      // Invocation of operation using the proxy
      if (service!=null)
         service.average(temperatureList);
   }
}
```

Figure 52. Liaison dynamique pour l'activité *Analysis* de l'application d'alarme.

Dans notre architecture, il existe deux façons d'ajouter le support d'une nouvelle technologie à services, la première consiste à étendre la machine SAM afin de supporter la nouvelle technologie. Il est aussi possible d'étendre notre environnement de spécification pour générer des composants liaison (dynamique homogène ou statique) pour la nouvelle technologie à utiliser. Dans les sections suivantes, nous allons présenter notre environnement de spécification d'orchestration et ses capacités d'extension.

7.2 CADSE-FOCAS : UN ENVIRONNEMENT POUR L'ORCHESTRATION

Nous avons constaté que la mise en place de notre approche est un travail complexe et difficile à réaliser lorsque l'on essaye de l'accomplir de façon manuelle, sans les outils adéquats. C'est pourquoi, en plus du *runtime*, notre canevas FOCAS a comme objectif de fournir un environnement de spécification permettant la construction des applications orientées procédé. Cet environnement permet d'utiliser plus facilement notre approche et nous sert aussi comme environnement d'expérimentation.

Nous avons défini un ensemble de caractéristiques que notre environnement doit respecter afin de s'aligner avec les caractéristiques de notre approche, elles sont :

- l'environnement doit être conçu comme un CADSE (acronyme en anglais pour *Computer Aided Domain Specific Environment*) centré sur le domaine de l'orchestration. Par conséquent, les développeurs utilisent des concepts du domaine de l'orchestration plutôt que de manipuler les concepts d'un environnement de développement générique tels que projet, package, fichier, etc.

- l'environnement doit fournir un niveau d'abstraction permettant aux développeurs de s'abstraire des détails techniques associés aux modèles de programmation et aux technologies utilisés par notre approche.
- l'environnement doit fournir un support pour la séparation des préoccupations. Différentes vues doivent être mises à la disposition des différents acteurs qui utilisent l'environnement pour qu'ils puissent accomplir leurs tâches dans la définition d'une application.
- l'environnement doit être extensible, afin d'évoluer en accord avec les extensions du *runtime* du canevas. Par conséquent, lorsque l'on ajoute une extension fonctionnelle ou un aspect non-fonctionnel au *runtime*, l'environnement doit refléter aussi cette nouvelle fonctionnalité.
- l'environnement doit être bâti sur Eclipse, car cette plate-forme fournit des mécanismes d'intégration et d'extension adéquats pour la mise en place de notre démarche. En plus, la technologie de méta-modélisation EMF que nous utilisons est fournie par la plate-forme Eclipse.

L'environnement CADSE-FOCAS doit permettre aux développeurs d'orchestrations de spécifier les artefacts qui font partie d'une orchestration. Ensuite, le CADSE-FOCAS doit appuyer le développeur dans les tâches d'ajout de comportement aux modèles d'orchestration et de génération partielle de médiateurs. Finalement, de façon transparente l'environnement doit générer les composants de liaison et les *proxies* s'ils sont nécessaires.

Nous allons d'abord présenter comment la vision basée sur les concepts d'orchestration est obtenue en utilisant des items logiques pour interagir avec l'environnement. Ensuite, nous présentons la démarche faite par un développeur pour construire une orchestration, ainsi nous dévoilons les outils mis à disposition par CADSE-FOCAS pour supporter cette spécification.

7.2.1 Les items et vues du CADSE-FOCAS

Nous voulons que le développeur des orchestrations ait une vision des applications en terme des concepts utilisés par notre approche plutôt que de la vision physique fournie par un environnement de développement standard. Pour offrir cette vision, le développeur va interagir avec le CADSE-FOCAS en utilisant un ensemble d'items logiques plutôt qu'en manipulant directement les concepts natifs de l'environnement de base tels que projet, répertoire et fichier.

Dans cet esprit, nous avons défini un type d'item pour chaque point de vue (type de modèle) utilisé par l'approche, donc les modèles de contrôle, de données et de services. En plus, nous définissons un type d'item composite « modèle d'orchestration » qui est composé des trois autres types d'items. Chaque item est associé à un projet Eclipse d'un type spécifique qui va contenir les artefacts utilisés pour spécifier le point de vue. En plus, afin de faciliter le traitement automatique des modèles, une structure est définie pour le projet ainsi que des conventions de noms. Le tableau de la Figure 53 présente les types d'items logiques, avec les types de projets Eclipse associés, la convention de noms utilisée et la structure des projets.

En plus des items logiques, des vues sont proposées afin que les différents acteurs (développeurs) puissent interagir avec l'environnement CADSE-FOCAS. Dans ces vues, les développeurs réalisent les opérations de création, d'importation, de modification et d'élimination des items logiques. Deux vues logiques ont été créées dans le CADSE-FOCAS, la vue *SimpleProcess* permet la manipulation de modèles simples de procédé. Dans cette vue, seuls des items logiques de type modèle de contrôle et de modèle de données sont manipulés. La vue *Orchestration* permet la manipulation de modèles d'orchestration, donc tous les types d'items logiques peuvent être manipulés dans cette vue.

Type d'item logique	Type de projet	Préfixe (convention de noms)	Structure
Modèle de contrôle	Projet Simple	Model.Control	Model.Control.Control1 model Control1.apel
Modèle de données	Projet Java	Model.Data	Model.Data.Data1 src JRE System Library [jre1.6.0_05] model Data1.product
Modèle de services	Projet Java	Model.Service	Model.Service.Service1 src actioner Controller.java JRE System Library [jre1.6.0_05] Item Dependencies
Modèle d'orchestration	Projet AspectJ	Model.Orchestration	Model.Orchestration.Orch1 src JRE System Library [jre1.6.0_05] AspectJ Runtime Library components Model.Control.Control1 Model.Data.Data1 Model.Service.Service1

Figure 53. Types d'items logiques : types de projets, convention de noms et structure.

Dans la Figure 54, les deux vues logiques du CADSE-FOCAS sont présentées à droite. A gauche de la figure, la vue *Package Explorer* (fournie par Eclipse) montre les projets associés aux items logiques dans l'environnement Eclipse. Par exemple, pour l'item de type modèle de contrôle *Control1* un projet *Model.Control.Control1* se trouve dans l'espace de travail d'Eclipse. Ce projet contient un sous-répertoire *model* et à l'intérieur un fichier *Control1.apel* qui correspond au modèle de contrôle.

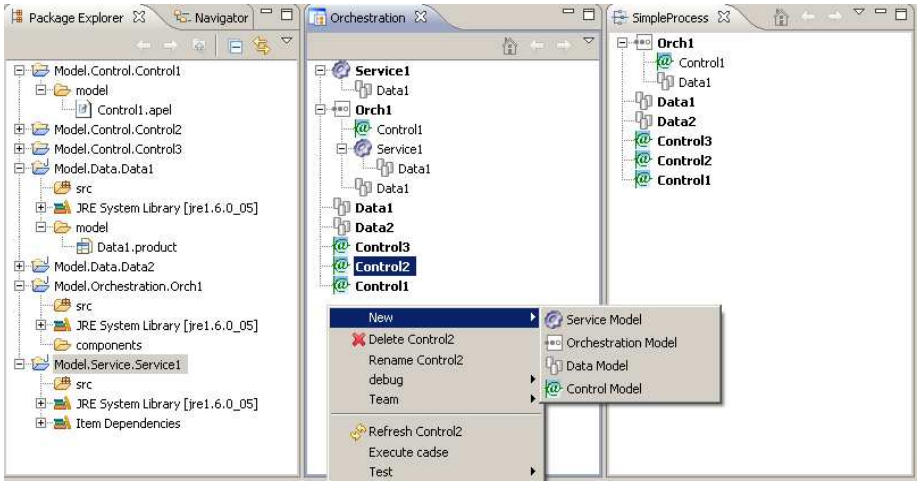

Figure 54. Items logiques et les projets Eclipse associés.

Lorsqu'un développeur souhaite construire une orchestration, il doit créer un item logique pour chaque type de modèle de base et un item de type modèle d'orchestration. Une fenêtre assistante (*wizard*) aide le développeur dans la création des items. Pour les modèles de base la seule information demandée par le *wizard* est le nom de l'item à créer. Pour l'item composite du modèle d'orchestration le *wizard* permet de sélectionner les items à composer parmi les items présents dans l'espace de travail. Par exemple, dans la Figure 54, l'item *Orch1* de type modèle d'orchestration, contient les items *Control1* de type modèle de contrôle, *Data1* de type modèle de données et l'item *Service1* de type modèle de services. Les vues spécifiques de notre environnement offrent aussi la possibilité de voir les relations entre les items logiques, ce qui n'est pas possible avec une vue physique comme celle de *Package Explorer*.

7.2.2 Spécification d'une orchestration

La création des items logiques entraîne la création des projets associés pour la spécification d'une orchestration. Cette définition des items permet d'avoir une vision de gros grain des modèles et de leurs relations dans notre approche. Ensuite, le développeur doit spécifier les modèles et établir les liens entre leurs concepts. Le produit résultat de cette tâche correspond au composant d'orchestration abstraite présenté dans l'architecture d'une orchestration. Nous allons maintenant présenter les outils supportant ces tâches, d'abord les éditeurs des différents modèles et ensuite les outils de composition de ces modèles.

Spécification des modèles

Comme nous l'avons présenté dans le chapitre dédié à l'approche, les concepts de chaque domaine sont explicitement décrits en utilisant un méta-modèle. Nous avons choisi la technologie EMF comme canevas de méta-modélisation dans notre démarche, par conséquent, le formalisme Ecore est utilisé pour spécifier nos méta-modèles. La représentation explicite des méta-modèles permet l'exploitation automatisée des modèles conformes à ces méta-modèles. En plus, EMF offre la possibilité de génération d'éditeurs pour nos langages, ceux-ci sont complètement intégrés dans la plate-forme Eclipse.

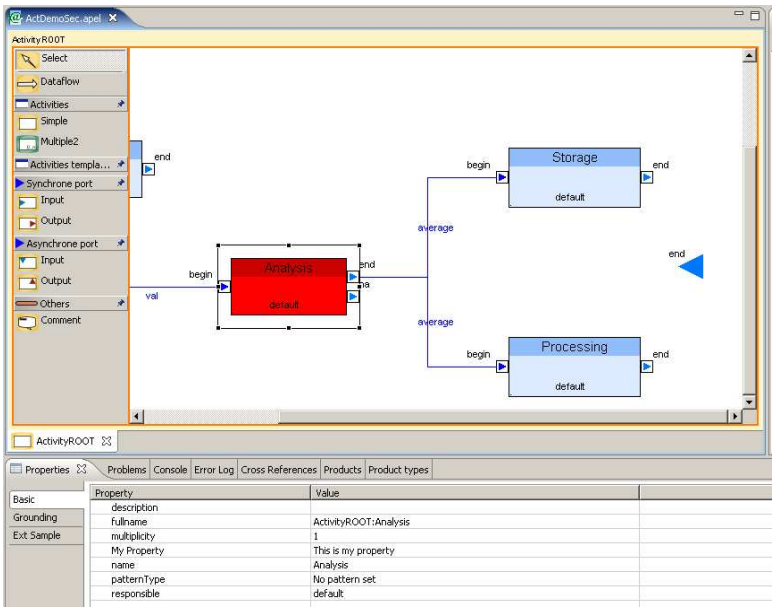

Figure 55. L'éditeur de contrôle (APEL) dans CADSE-FOCAS.

Dans le cas du langage APEL, nous avons des besoins spécifiques car une syntaxe concrète graphique est prédéfinie pour le langage. En plus, l'éditeur doit être intuitif, et facilement extensible étant donné que le modèle de contrôle est central dans notre démarche. Par conséquent, l'éditeur d'APEL n'a pas été généré avec le canevas EMF, cependant les classes générées par EMF pour le traitement des modèles ont été gardées. L'éditeur d'APEL a été construit comme un éditeur Eclipse, il est présenté dans la Figure 55.

Pour le domaine de données, nous avons généré un éditeur en utilisant EMF à partir de la spécification en Ecore du méta-modèle. Etant donné que ce méta-modèle est simple, nous n'avons pas besoin d'un éditeur plus sophistiqué. L'éditeur de données généré est présenté dans Figure 56.

Finalement, pour le modèle de services, nous n'avons pas défini explicitement en Ecore son méta-modèle, car celui des interfaces Java a été utilisé. En plus, pour spécifier les modèles de services, nous avons repris l'éditeur fourni par Eclipse.

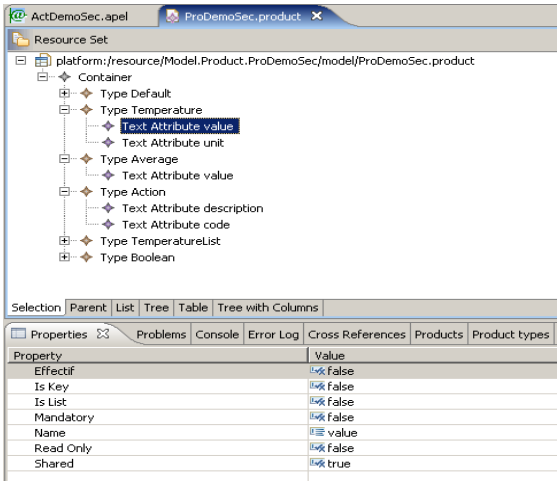

Figure 56. L'éditeur de données dans le CADSE-FOCAS.

Etablissement de relations entre les modèles

Après avoir défini chacun des modèles faisant partie de l'orchestration, l'opération de mise en relation entre eux doit être réalisée. Afin de supporter cette opération, le CADSE-FOCAS construit un projet composite d'orchestration et fournit les outils adéquats pour la définition des liens.

La tâche de construction du projet composite d'orchestration est basée sur la définition des items logiques. En fait, dans le projet associé à un item logique de type modèle d'orchestration, sont copiés les fichiers correspondant aux modèles de chaque domaine. Etant donné qu'une structure a été fixée pour ce type de projet, les fichiers sont placés dans les répertoires adéquats pour qu'ils soient consultés par les outils de composition de modèles. La tâche de construction est déclenchée automatiquement chaque fois qu'un modèle faisant partie du modèle composite est modifié.

La Figure 57 présente la structure d'un projet associé à un item de type orchestration. Dans l'exemple, pour l'item *Orch1* le projet *Model.Orchestration.Orch1* contient un sous-répertoire *components.* Dans le répertoire *components* sont copiés les répertoires des projets correspondant à chaque item logique composé. Dans le cas de l'exemple, sont copiés les répertoires *Model.Control.Control1*, *Model.Data.Data1* et *Model.Service.Service1* qui correspondent aux items *Control1*, *Data1* et *Service1* respectivement.

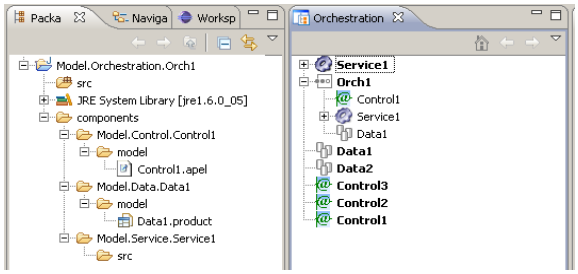

Figure 57. Projet correspondant à un item de type modèle d'orchestration.

Après la construction du projet composite d'orchestration, l'utilisateur peut exprimer les liens pour effectuer la composition des modèles. Pour la définition des liens entre les modèles, différents outils ont été construits et intégrés dans le CADSE-FOCAS.

Pour la composition des modèles de contrôle et des données l'outil *Codèle* a été intégré dans notre environnement. L'outil *Codèle* a été conçu pour réaliser la composition générique de modèles ; il doit être configuré en utilisant la définition d'un ensemble de méta-liens entre deux méta-modèles. Dans notre cas, la définition des méta-liens entre le méta-modèle de contrôle et

celui de données est faite une seule fois dans notre démarche, et ensuite elle est utilisée pour configurer *Codele* pour son utilisation dans CADSE-FOCAS. *Codèle* charge les modèles à composer en utilisant la définition d'items logiques, la convention de noms et les structures définies pour les projets. Par exemple, pour charger le modèle de contrôle correspondant à l'item logique *Orch1*, CADSE-FOCAS récupère, à partir de la définition des items, l'item correspondant au contrôle, donc *Control1*. Ensuite, par convention de noms et l'utilisation de la connaissance de la structure des projets, *Codèle* cherche, dans le répertoire *components* un sous-répertoire *Control.Model.Control1* et à l'intérieur le fichier *Control1.apel* de celui-ci qui correspond au modèle de contrôle, comme cela est montré dans la Figure 57.

La Figure 58 présente une capture d'écran de l'outil *Codele*. A gauche, des éléments de type *ProducType* du modèle de contrôle sont affichées et, à droite, des éléments de type *ComplexType* de modèle de données. L'utilisateur doit sélectionner les couples et appuyer sur le bouton *Map* pour créer un lien entre les deux instances de chaque modèle. L'exemple présenté correspond à la définition des liens faite pour notre système d'alarme.

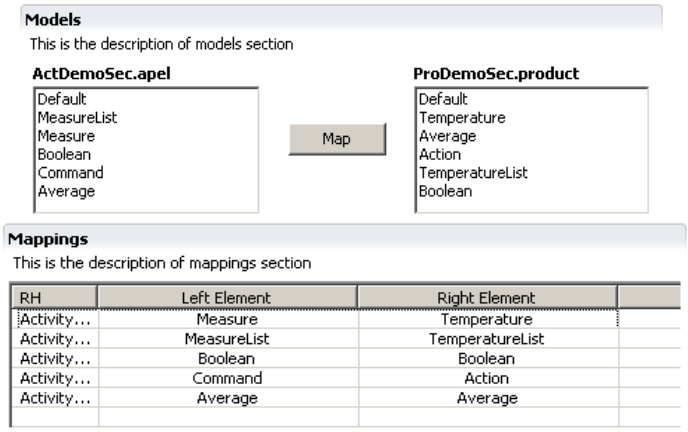

Figure 58. Spécification des liens entre le modèle de contrôle et données avec Codele.

D'autre part, la composition entre les modèles de contrôle et services est réalisée avec l'outil *Grounding* (matérialisation) de notre environnement. Bien que l'outil *Codèle* permette de définir des relations binaires entre deux modèles, nous ne l'avons pas utilisé puisque le méta-modèle de services n'a pas été décrit en utilisant Ecore ce qui est une exigence technique de *Codèle*. En plus, nous voulons que cette composition soit plus intuitive et soit effectuée directement sur l'éditeur de contrôle ; ainsi les utilisateurs peuvent manipuler directement les activités au moment de les associer aux opérations de services.

Par conséquent, l'outil de *Grounding* a été construit comme une extension de l'éditeur d'APEL. Dans l'outil, l'activité à matérialiser est sélectionnée et l'utilisateur choisit l'option *Activity Service Grounding* du menu affiché. Ensuite, une fenêtre avec une liste de services disponibles et leurs opérations est affichée. Si une opération compatible est sélectionnée l'opération de composition (matérialisation*)* finit. La compatibilité d'une opération est déterminée si les types de données du port *desktop* de l'activité correspondent aux types utilisés par les paramètres de l'opération. Dans le cas contraire, il est nécessaire de faire une association explicite des produits présents dans le port *desktop* avec les paramètres de l'opération choisie.

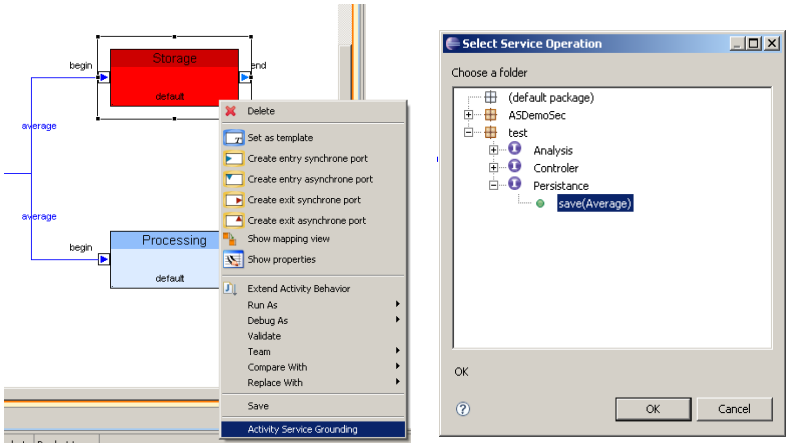

Figure 59. Composition des modèles de contrôle et service avec l'outil *Grounding*.

Dans la Figure 59, l'outil *Grounding* est présenté pour notre exemple, l'activité *Storage* est sélectionnée dans l'éditeur d'APEL, ensuite dans la fenêtre qui liste les services disponibles, l'opération *save* du service *Persistance* est choisie.

7.2.3 Génération de code : Comportement, Médiateurs, et Liaison

En plus des outils de spécification et de composition de modèles, le CADSE-FOCAS fournit un ensemble de générateurs de code servant à divers propos, comme la génération partielle du composant de comportement et des médiateurs, ou la génération totale du composant de liaison et le code de support des aspects non-fonctionnels. Les générateurs de code peuvent être utilisés de deux façons, au niveau du modèle d'orchestration ou d'une activité du modèle.

Les classes faisant partie du composant de comportement sont générées partiellement. S'il s'agit de l'ajout de comportement direct pour une activité, un menu permettant cette opération est présenté dans l'éditeur d'APEL lorsqu'une activité est sélectionnée. L'opération génère la classe implémentant l'interface (*ActivityObserver*) et le développeur inclut le code du comportement dans la méthode *doActive* de cette classe. D'autre part, s'il s'agit d'un type d'activité, le code est totalement généré, le développeur peut le générer pour chaque activité ou pour tout le modèle d'orchestration.

Les médiateurs sont générés pour les activités qui ont une relation de matérialisation (*grounding*) avec une opération d'un service. Lors de la génération, ils sont complétés par le développeur. Finalement, les classes de composant de liaison sont générées pour chaque relation de matérialisation (*grounding*) et de liaison (*binding*) ; elles peuvent être générées par activité ou pour tout le modèle.

Les générateurs de code de notre environnement utilisent la technologie JET (*Java Emitter Templates*) et ils sont modifiables, c'est-à-dire qu'ils peuvent être remplacés pour générer des artefacts selon les besoins des applications, par exemple le générateur du code de liaison est changé selon le type de liaison à utiliser par l'orchestration.

7.3 LE META-ENVIRONNEMENT FOCAS

Un environnement est utilisé dans notre canevas pour construire l'environnement CADSE-FOCAS. Cet environnement, que nous allons appeler méta-environnement FOCAS est utilisé par les développeurs du canevas pour créer des extensions au CADSE-FOCAS. La Figure 60 présente la relation entre le méta-environnement FOCAS et le CADSE-FOCAS, les rôles joués par les types de développeurs.

Nous pouvons considérer que le méta-environnement est composé de l'outil CADSEg, d'outils de méta-modélisation, d'outils de composition de méta-modèles et par le générateur de

code. Dans cette section, nous allons nous concentrer sur la mise en œuvre du CADSE-FOCAS en utilisant le méta-environnement FOCAS. Dans la section suivante, nous allons présenter comment les extensions du CADSE-FOCAS sont faites.

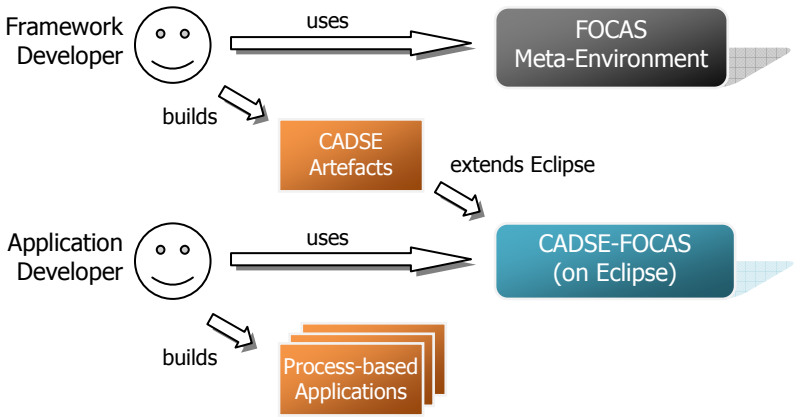

Figure 60. Méta-environnement FOCAS et sa relation avec le CADSE-FOCAS.

7.3.1 L'outil CADSEg

Pour définir nos types d'items logiques et leurs associations avec l'environnement Eclipse nous avons utilisé l'outil CADSEg (*CADSE Generator)*. Cet outil suit une approche IDM. En utilisant un ensemble de modèles spécifiant un CADSE, CADSEg génère les extensions nécessaires pour personnaliser un environnement de développement classique comme Eclipse.

Modèle des types items

Le premier modèle à spécifier dans CADSEg est le modèle de types d'items. Dans ce modèle, un type d'item est décrit avec un ensemble de propriétés et ses relations avec des autres items types. Dans le diagramme de classes de la Figure 61, nous présentons le modèle de type d'items utilisé dans le CADSE-FOCAS. Un item de type modèle d'orchestration doit contenir au moins un item de type modèle de contrôle et un item modèle de données. Par contre, l'item modèle de services est optionnel. Le modèle de services pour sa part utilise un modèle de données pour le typage des paramètres des opérations, ce qui implique une contrainte car on ne peut utiliser qu'un modèle de services compatible avec le modèle de données utilisé pour l'item composite. Finalement, les modèles définis dans CADSE-FOCAS sont réutilisables, c'est pour cela qu'un modèle de contrôle peut être utilisé dans plusieurs modèles d'orchestration, la même caractéristique est applicable au modèle de données et de services.

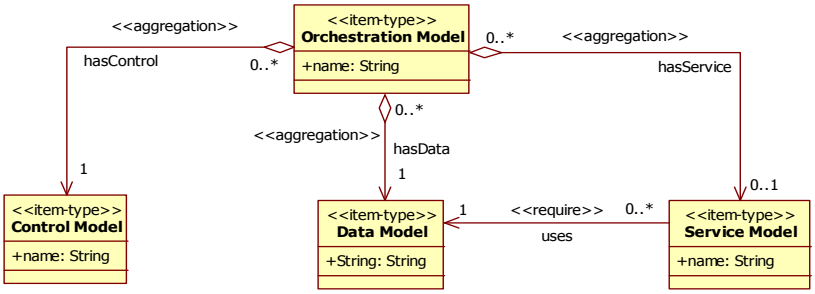

Figure 61. Modèle des concepts de l'environnement CADSE-FOCAS.

En fait, les relations du modèle des concepts de CADSEg déterminent le cycle de vie des items. Par exemple, la relation de type « *aggregation* » entre le modèle d'orchestration et

celui de contrôle indique une sémantique composite-composant, donc lorsqu'un item de type modèle d'orchestration est éliminé, le modèle de contrôle continue à exister. CADSEg offre des relations de type « *part* » qui expriment aussi une sémantique composite-composant, mais avec un cycle de vie couplé : lorsque le composite est éliminé le composant est également éliminé. Une relation de type « *require* » exprime un besoin entre deux items, c'est le cas de la relation entre l'item de modèle de services et l'item de modèle de données. Le diagramme de classes présenté contient des stéréotypes dans les associations pour indiquer le type des relations utilisées dans notre modèle des types d'items.

Correspondance entre le modèle de l'environnement et Eclipse

Le CADSEg offre un *runtime* qui s'exécute sur Eclipse ; il est capable d'interpréter le modèle des types d'items qui a été défini auparavant, donc des items logiques peuvent être créés, modifiés et éliminés. Cependant, pour faire une vraie exploitation de ces items, une correspondance doit être définie avec les concepts de l'environnement Eclipse. Dans CADSEg, il est possible d'associer des items logiques à divers types d'artefacts d'Eclipse comme des projets (Simple, Java ou AspectJ) ou des fichiers (texte, xml, jar, classes Java). Chaque type d'item logique est associé à un projet dans Eclipse. Le type de projet associé est choisi en fonction des besoins des items logiques en terme d'outils. L'item de type modèle de contrôle a été associé à un projet Eclipse simple, étant donné que le modèle de contrôle correspond à un fichier de type APEL (un fichier XML), nous avons uniquement besoin d'assurer la persistance de ce modèle. Pour sa part, l'item modèle de services a été associé à un projet Eclipse Java. Ce choix s'explique puisque nous utilisons le langage Java pour exprimer ce modèle, donc nous profitons des outils offerts par ce type de projet : le compilateur servant à valider le modèle et l'éditeur Java pour sa description.

Pour l'item de type modèle de données, un projet Java a été associé. Ce choix s'explique puisque, en plus du modèle de données défini dans ce projet, l'environnement CADSE-FOCAS génère une représentation en Java des types afin de faciliter le codage du composant du comportement. Finalement, pour l'item composite modèle d'orchestration un projet *AspectJ* a été associé, car ce type d'item doit supporter la définition du composant de comportement (faite en Java). Lorsque des aspects non-fonctionnels sont ajoutés, la technologie d'*AspectJ* est utilisée pour tisser le code produit pour le support de ces aspects avec l'interpréteur de l'orchestration.

Après la définition des correspondances, l'on décrit les interactions des utilisateurs avec l'environnement en utilisant les items logiques dans CADSEg. Les interactions sont matérialisées par des *wizards* qui sont générés par CADSEg pour permettre la création et la modification des items. Un *wizard* demande des informations à l'utilisateur au moment de la création de l'item servant à affecter ou calculer la valeur de ses attributs ou de ses relations avec d'autres items. Dans notre CADSE-FOCAS, les *wizards* de l'item modèle de contrôle et du modèle de données sont simples, l'unique information qu'ils demandent est la valeur pour l'attribut *name* de chaque modèle. Le *wizard* généré pour l'item modèle de services, en plus de demander l'attribut nom, permet de choisir un item de type modèle de données pour résoudre sa relation « *uses* ». Le *wizard* de l'item modèle d'orchestration pour sa part demande les trois items correspondant aux sous-modèles. Chaque *wizard* fait la création de l'item et du projet associé, en plus le code généré a été étendu pour créer la structure des répertoires pour chaque projet.

Finalement, il est possible de définir les vues logiques dans CADSEg, cette spécification consiste à indiquer les vues à générer et quels types d'items logiques peuvent être manipulés dans ce type de vues. Du code étendant Eclipse pour ajouter les vues est aussi généré par CADSEg à partir de la définition des vues. Dans la Figure 62, la spécification de modèles dans l'outil CADSEg est présentée, à gauche le modèle de types d'items (*data-model*). Pour chaque type d'item sont spécifiés ses attributs et les *wizards* (*creation dialog* et *modification dialog*). A droite en haut, les correspondances (*mapping*) entre les différents types de projets

Eclipse sont exprimées. A droite en bas, les vues (*view-model*) et les items qu'elles manipulent sont spécifiés.

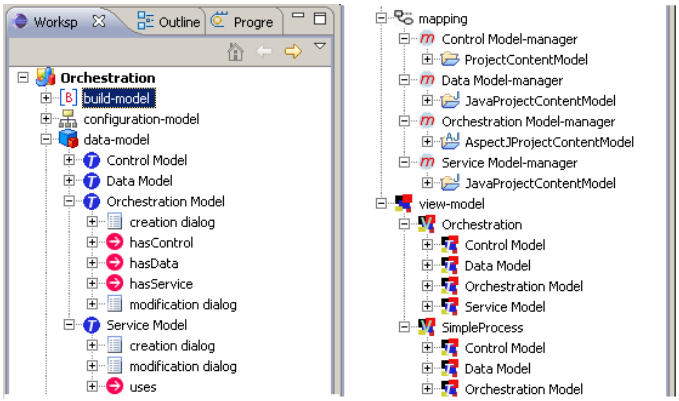

Figure 62. Spécification du CADSE-FOCAS dans l'outil CADSEg.

7.4 EXTENSIONS DU CADSE-FOCAS

Dans cette section, nous allons montrer comment notre environnement de spécification CADSE-FOCAS peut être étendu conformément aux extensions du *runtime* présentées dans le chapitre dédié à notre approche. Nous allons d'abord présenter les extensions fonctionnelles de l'environnement, servant à ajouter le support pour un nouveau domaine exécutable, et ensuite les extensions supportant l'ajout des aspects non-fonctionnels pour l'orchestration.

7.4.1 Extensions fonctionnelles

Une extension fonctionnelle dans FOCAS consiste en la spécification d'un nouveau domaine (point de vue), c'est-à-dire la définition de son méta-modèle et son interpréteur et ensuite la mise en relation de ce point de vue avec les autres domaines. L'inclusion d'un point de vue se traduit par l'obtention d'un interpréteur supportant l'exécution des applications orientées procédé supportant la nouvelle préoccupation. Une extension de ce type consiste en :

- l'inclusion dans CADSEg d'un nouveau type d'item à traiter ;
- la spécification du méta-modèle du nouveau domaine et la création d'un éditeur qui sera intégré dans l'environnement ;
- la définition d'un outil de composition de modèles servant à composer des modèles du nouveau domaine avec le modèle de contrôle. *Codèle* étant un outil générique pour la composition de modèles peut être utilisé ;

Pour illustrer le mécanisme d'extension fonctionnelle du CADSE-FOCAS, nous allons utiliser l'exemple de l'orchestration avec le support des tâches réalisées par des humains.

Extension par CADSEg

Afin d'ajouter cette capacité à l'environnement nous avons créé un type d'item modèle de ressources (*ResourceModel*), et nous avons défini un nouveau type d'item composite pour le nouveau type d'application orientée procédé (*HumanOrchestrationModel*).

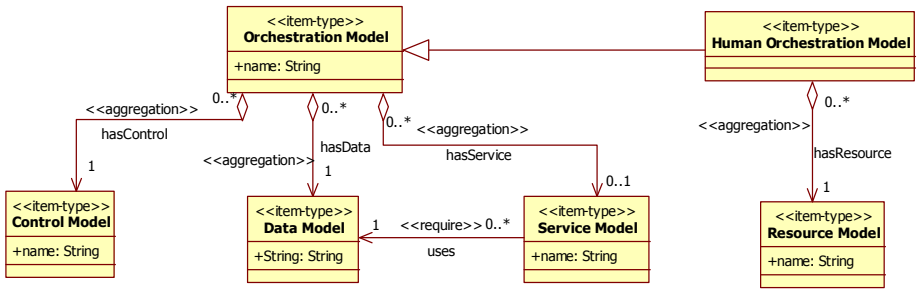

Figure 63. Modèle des concepts de l'environnement CADSE-FOCAS étendu.

Dans la Figure 63, le modèle correspondant aux types d'items CADSE-FOCAS étendu est présenté. Le type d'item de type modèle de ressource (*ResourceModel)* est spécifié de façon traditionnelle, tandis que le type d'item modèle d'orchestration humaine (*HumanOrchestrationModel*) est défini comme une extension de type d'item modèle d'orchestration.

L'extension est faite au niveau conceptuel, ensuite les correspondances avec les concepts d'Eclipse doivent être définies pour les nouveaux types d'items. Dans le cas de modèle de ressources un projet simple est utilisé, dans le cas de modèle d'orchestration humaine un projet *AspectJ* est choisi. La structure du projet correspondant à un item de type modèle d'orchestration humain est le même que pour un modèle d'orchestration, de cette façon les outils déjà mis en place n'ont pas besoin de modification. Par contre, la construction de ce nouveau type d'item doit inclure la copie du modèle de ressource dans le répertoire « *components* » de projet composite. En plus, une nouvelle vue est ajoutée au CADSE-FOCAS, qui permet de manipuler les nouveaux types d'items.

La Figure 64 présente la vue « *HumanOrchestration* » du CADSE-FOCAS étendu dans laquelle les items des nouveaux types peuvent être manipulés. L'item *Resource1* de type modèle de ressource et l'item *HumanOrch1* de type modèle d'orchestration humaine ont été définis. A gauche dans la vue *Package Explorer* les projets associés sont présentés. De la même façon que pour les types items de base, une convention de noms est utilisée.

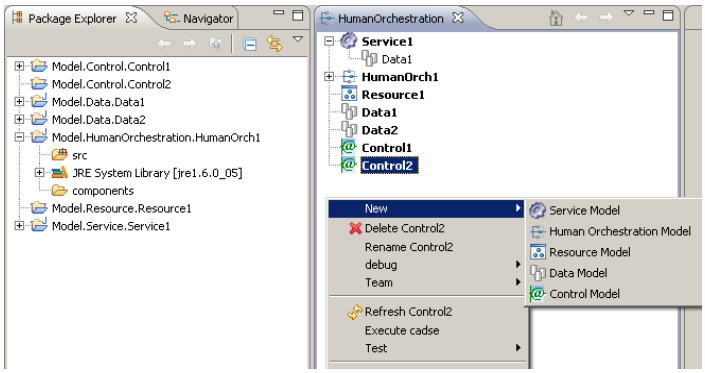

Figure 64. Extension du CADSE-FOCAS pour l'inclusion du modèle de ressource.

Nouveau méta-modèle, éditeur et outil de composition

Le méta-modèle du nouveau point de vue est ensuite spécifié en utilisant Ecore. Un éditeur peut alors être généré en utilisant la technologie EMF comme pour les autres domaines. Nous avons spécifié un méta-modèle simple pour le domaine de ressources et généré un éditeur qui a été intégré dans le CADSE-FOCAS.

Ensuite, les méta-liens entre les méta-modèles de contrôle et de ressources doivent être définis. Nous avons utilisé l'outil *Codele* (niveau méta) pour réaliser cette définition, ensuite la

définition des méta-liens est utilisée dans le CADSE-FOCAS pour configurer Codele (au niveau modèle).

La technique d'extension fonctionnelle de l'environnement CADSE-FOCAS a été utilisée pour construire des environnements spécialisés pour inclure l'aspect humain dans l'exécution d'une orchestration, et pour faire l'environnement du système de *workflow*.

7.4.2 Extensions pour le support des aspects non-fonctionnels

Pour l'ajout du support des aspects non-fonctionnels dans le CADSE-FOCAS nous avons utilisé uniquement l'outil CADSEg. Il est nécessaire d'implémenter deux composants lorsque l'on souhaite étendre l'environnement CADSE-FOCAS pour supporter un aspect non-fonctionnel, d'abord un composant permettant la spécification des propriétés de l'application et ensuite un composant chargé de générer le code pour la supporter. Nous allons reprendre l'exemple de l'ajout de l'aspect sécurité pour illustrer comment l'environnement CADSE-FOCAS est étendu.

Spécification des propriétés non-fonctionnelles

Dans notre approche, des annotations indiquent les propriétés non-fonctionnelles de l'application, cependant elles n'expriment rien par rapport au moyen qui sera utilisé pour les implanter. Par exemple, le fait d'annoter une activité avec la propriété *authentification* indique qu'avant d'invoquer l'opération du service associé, l'orchestration doit s'authentifier devant le service. Nonobstant le mécanisme à utiliser pour cette authentification n'est pas exprimé avec cette annotation.

Cependant, en plus des annotations indiquant les propriétés non-fonctionnelles pour une orchestration et selon les mécanismes choisis pour assurer ces propriétés, des informations de niveau technique doivent être fournies pour implémenter cet aspect. En plus, l'utilisateur responsable d'indiquer les propriétés d'un aspect non-fonctionnel pour un modèle d'orchestration n'est pas forcément un expert dans la technologie supportant l'aspect. Alors, une nouvelle séparation des préoccupations est nécessaire à ce niveau, permettant à chacun des acteurs de fournir l'information nécessaire pour le support de l'aspect. Par conséquent, le composant de spécification de propriétés non-fonctionnelles doit fournir la possibilité de spécifier les propriétés non-fonctionnelles pour les orchestrations, mais aussi les informations de niveau technique qui seront utilisées pour l'implémentation de l'aspect non-fonctionnel.

Dans le CADSE-FOCAS, un composant de spécification des propriétés non-fonctionnelles est créé comme une extension de l'éditeur d'APEL, ce qui permet graphiquement d'inclure les annotations sur le modèle de contrôle. Nous avons laissé un point d'extension dans l'éditeur d'APEL dans lequel des pages de propriétés peuvent être ajoutées, de cette façon lorsqu'une instance d'un concept est sélectionnée dans l'éditeur, des propriétés qui ne faisaient pas partie originalement du concept d'APEL peuvent ainsi être exprimées. De ce fait, un composant de spécification est une extension qui profite de ce point d'extension ajoutant des pages de propriétés à l'éditeur. Ainsi, une page est créée pour l'expression des propriétés définies dans le méta-modèle de l'aspect non-fonctionnel et une page est créée pour l'inclusion des informations techniques servant à l'implémentation de l'aspect.

Dans notre scénario d'inclusion de la sécurité dans l'orchestration, l'outil de spécification de propriétés de sécurité est divisé en deux pages de propriétés ajoutées à l'éditeur d'APEL. La première page permet d'annoter les concepts d'APEL avec les concepts de méta-modèle de sécurité comme l'authentification, l'intégrité et la confidentialité. Dans la Figure 65, cette page de propriétés (*General Security*) est présentée, lorsqu'une une instance d'une activité est sélectionnée. Des cases à crocher sont disponibles pour annoter l'activité afin d'indiquer si elle va supporter cette propriété. Par exemple, dans la figure, l'activité *Analysis* est sélectionnée

et elle est annotée avec la propriété d'authentification.

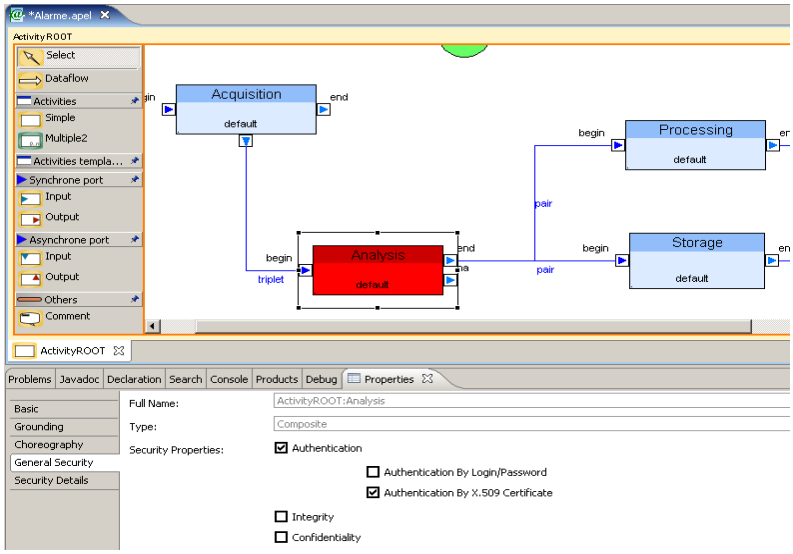

Figure 65. Spécification des annotations de sécurité sur le modèle de contrôle.

Dans la Figure 66, nous présentons la page de propriétés pour les détails techniques de l'aspect sécurité. Par exemple dans le cas de l'authentification, des informations comme l'utilisateur, mot de passe et localisation du fichier de certificat sont demandées. Ces informations seront utilisées pour l'implémentation des invocations des services.

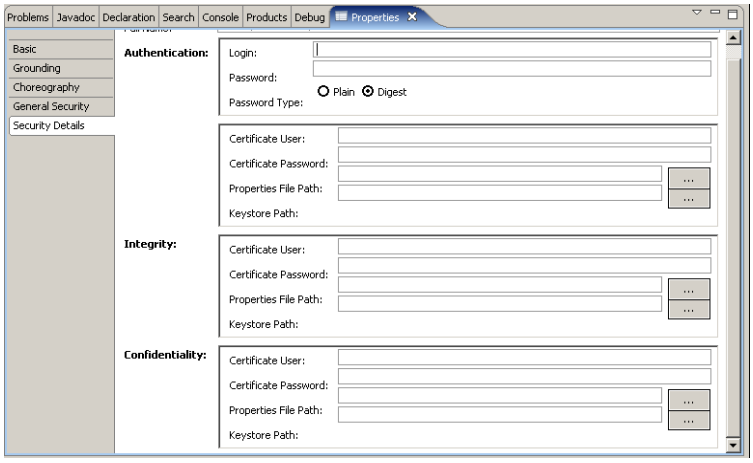

Figure 66. Page de propriétés des annotations concrètes pour la sécurité.

Génération du code de support des aspects non-fonctionnels

Le deuxième composant à implémenter pour étendre notre environnement pour le support des aspects non-fonctionnels est un générateur de code. Le CADSE-FOCAS fournit un point d'extension pour ce propos. Le point d'extension fournit des fonctionnalités de base comme la gestion des répertoires de code source, la navigation de la structure de projets (en se basant sur l'API de CADSEg) et l'accès aux méta-modèles de base (contrôle, données et services). Avant de présenter comment le générateur de code a été implémenté pour le scénario de l'extension de sécurité, il est nécessaire de rendre explicites certaines caractéristiques du type d'applications auxquelles l'extension sera appliquée :

- l'extension sera appliquée à des orchestrations qui utilisent comme technologie d'implémentation de services les services Web. La recommandation choisie afin de supporter la communication sécurisée est *WS-Security*.

- une liaison dynamique sera utilisée dans ces orchestrations. Les interfaces de services concrets seront connues avant le déploiement, ce qui permet de générer des *proxies* pour l'invocation des services Web. Par contre, les instances seront choisies à l'exécution.
- la plate-forme qui supportera l'implémentation de protocole SOAP afin de communiquer avec les services Web, est *Apache Axis*. *Axis* supporte la modification des messages SOAP pour inclure les aspects de sécurité. Axis utilise le composant *WSS4J* pour fournir cette capacité.

L'implémentation du générateur consiste à spécialiser la classe *Generator* de l'environnement. Le CADSE-FOCAS au moment de générer le code pour un modèle d'orchestration observe si des annotations ont été définies pour une activité du modèle, si c'est le cas la classe implémentant le générateur est invoquée. La technologie JET (*Java Emitter Templates*) est utilisée pour créer les *templates* du code à générer.

Pour l'extension de sécurité la classe *GenerateSecurityFiles* spécialise la classe *Generator* du canevas. Elle est chargée de générer une classe que sera invoquée par *Axis* au moment de l'invocation d'une opération pour récupérer le mot de passe qui sera utilisé par l'activité annotée avec la propriété d'authentification. Elle génère aussi la classe de liaison chargée d'invoquer l'opération. En fait, la logique d'invocation est identique à celle d'un service non sécurisé, mais l'initialisation du *proxy* pour invoquer le service est modifiée.

```
public class AnalysisObserver implements ActivityObserver {

   // Interface du Proxy corresponding to Analysis Abstract Service
   AnalysisPortType service;
   // Discovery Component for Web Services
   AnalysisServiceLocator serviceLocator;

   public AnalysisObserver() {
      // The serviceLocator configuration add security properties
      EngineConfiguration config = new FileProvider("Analisys_client-deploy.wsdd");
      serviceLocator = new AnalysisServiceLocator(config);
   }

   public void doActive(Activity activity) {
      ...
      if (service==null) {
         // A instance of service is retrieved
         try {
            service = serviceLocator.getAnalysisPort();
         } catch (ServiceException e) {
            ...
         }
      }
      // Invocation of operation using the proxy
      if (service!=null)
         service.average(temperatureList);
   }
}
```

Figure 67. Observateur d'invocation du service avec les propriétés de sécurité.

7.5 SYNTHESE

Dans ce chapitre, nous avons présenté la mise en œuvre de notre canevas FOCAS. Le canevas est formé d'un *runtime* supportant l'exécution des applications et d'un environnement de spécification CADSE-FOCAS permettant la spécification des applications en utilisant une démarche IDM.

Dans un premier temps, nous avons montré l'architecture des applications créées en utilisant notre approche. Nous proposons une architecture, qui découpe l'expression des besoins en terme de services, de l'implémentation concrète de services. Ce découpage permet d'exprimer une orchestration en termes abstraits et ainsi de séparer l'application de la plate-

forme technologique utilisée. Une autre caractéristique est la capacité d'ajouter une couche de médiation entre les services abstraits et les services concrets. Nous avons introduit aussi, le concept de modèle de procédé simple, et comment le comportement est ajouté à ce modèle.

Ensuite, nous avons présenté l'architecture du *runtime*. Nous avons montré comment en utilisant la technique de composition de domaines exécutables, le moteur d'exécution de procédés simple a été construit. Ce moteur est la base de l'architecture de notre *runtime*, il offre la possibilité d'exécuter un modèle de procédé simple, mais aussi de brancher le comportement associé aux modèles de procédé et de connecter les différents types de liaison aux modèles d'orchestration pour donner une notion d'exécution concrète aux applications.

En plus du *runtime* du canevas nous considérons que, pour une mise en place effective de notre approche, il est nécessaire d'avoir des outils de spécification permettant aux développeurs de profiter de la puissance de l'approche sans se soucier de tous les détails techniques associés. Par conséquent, nous avons développé un environnement spécifique pour les applications orientées procédé : CADSE-FOCAS. L'environnement fournit un ensemble de caractéristiques comme l'utilisation des éditeurs et des outils de composition de modèles dédiés, aux diverses vues pour la spécification des applications, des outils de génération de code, etc.

Nous considérons que notre approche et sa mise en place contribuent à l'état de l'art de l'orchestration de services et en général à l'implémentation des applications orientées procédés. Les apports de notre démarche sont :

- l'utilisation de divers points de vue dans la spécification d'une orchestration. Chaque point de vue est d'un niveau d'abstraction haut et exprimé dans un langage dédié.
- la séparation entre la spécification abstraite et les services concrets qui implémentent la fonctionnalité. Cette séparation élimine la dépendance traditionnelle envers une technologie à services spécifique, dans la plupart des cas, des services Web.
- l'utilisation d'une couche de médiation permettant d'adapter les services concrets existants aux besoins d'une orchestration. Ainsi, la logique d'adaptation peut être exprimée sans modifier la logique de l'application.
- la définition d'un modèle de procédé simple permettant la spécification de la logique d'une application en terme d'un ensemble d'activités et des données traitées. Ce modèle permet de spécifier l'objectif d'un procédé sans indiquer comment il sera réalisé. Le modèle est exécutable, ce qui permet la validation d'un modèle d'application avant de mettre en place tout le support pour son implantation.
- le modèle de procédé simple est extensible avec l'ajout de comportement. Ce comportement permet de donner une sémantique d'exécution à une activité. Le comportement permet de doter le modèle de procédé d'une capacité de calcul, évitant d'utiliser des services pour tous les calculs réalisés par une orchestration. Le comportement permet aussi, de créer des types d'activités qui peuvent être réutilisés dans de nouveaux modèles de procédés.
- le support de différents types de liaisons de services. Une liaison SAM à l'exécution (*dynamic binding by lookup*) indépendante de la technologie à services utilisée, mais aussi une liaison dynamique plus performante et une liaison statique. Il est possible aussi de réaliser des extensions pour le support de nouvelles technologies à services, ce qui rend les applications moins dépendantes des plates-formes.
- l'implémentation d'un *runtime* extensible supportant les différents besoins des applications orientées procédé lorsqu'elles sont utilisées dans différents contextes. La réutilisation est la propriété principale de notre *runtime*, car l'inclusion d'une nouvelle caractéristique n'entraîne pas la création d'un nouveau *runtime*, sinon l'extension du *runtime* de base pour supporter la caractéristique.

- l'implémentation du CADSE-FOCAS qui fournit un environnement intégré pour la spécification des applications orientées procédés. L'environnement est aussi extensible, donc l'inclusion d'un nouveau domaine dans le canevas, en plus d'être réalisé au niveau *runtime* est aussi fait au niveau de l'environnement. A nouveau, la réutilisation est la principale motivation, mais aussi la flexibilité permettant utiliser notre technologie dans un spectre assez large d'applications orientées procédés, tout en prenant comme base l'orchestration de services.

Nous allons dans le prochain chapitre montrer comment, en utilisant notre canevas et les outils présentés dans ce chapitre, une approche pour la création des applications en utilisant une exécution de l'orchestration repartie a été mise en place. Cette démarche valide d'abord notre approche, mais en même temps elle permet de vérifier les capacités des outils créés pour la supporter.

8. L'ORCHESTRATION REPARTIE

Dans ce chapitre, nous nous intéressons à l'utilisation d'une architecture repartie pour l'exécution des orchestrations de services. Les approches actuelles d'orchestration de services font l'hypothèse qu'un moteur d'exécution centralisé est responsable de charger un modèle de l'orchestration et ensuite, en utilisant ce modèle, les services sont invoqués avec les bons paramètres et aux bons moments. La technologie généralement utilisée masque les détails techniques associés aux protocoles de communication nécessaires pour interagir avec les services composés. En plus les applications utilisent des unités de granularité assez grosses (les services Web) et une spécification unique (le modèle d'orchestration) qui exprime l'application complète. Techniquement, le fait que le modèle soit interprété par une seule machine facilite l'implémentation, l'administration et le monitoring de l'orchestration. Par contre, la machine où l'orchestration s'exécute deviens le cœur du système dans lequel arrivent et sortent toutes les communications. Donc cette machine devient potentiellement un goulot d'étranglement et limite le passage à l'échelle de l'application.

D'autre part, la technologie de chorégraphie de services exprime une collaboration entre services (des services Web) comme un ensemble de messages échangés par ces services. Dans cette vision, les services communiquent directement entre eux sans besoin d'un moteur d'exécution centralisé. Ainsi, l'approche de chorégraphie possède des propriétés de passage à l'échelle beaucoup plus importantes que l'approche d'orchestration ; par contre l'expression d'une chorégraphie est complexe et difficile à comprendre. Du point de vue de l'implémentation, la chorégraphie impose des contraintes assez fortes puisque le protocole d'interaction doit être établi préalablement entre les pairs, ce qui rend difficile d'ajouter des services existants dans une chorégraphie. Une solution possible à ce problème est l'inclusion de code pour assurer le routage des messages suivant le protocole établi, mais il n'est pas toujours possible de déployer ce code dans les machines qui exécutent les services.

Par conséquent, nous avons mis en place une architecture d'exécution qui est une combinaison de ces deux approches. D'abord une orchestration est exprimée en utilisant un modèle global mais qui utilise un schéma d'exécution reparti comme dans la chorégraphie. Notre but est d'augmenter la performance des applications et d'assurer le passage à l'échelle par l'utilisation d'une architecture distribuée, tout en en évitant les contraintes imposées par l'approche de chorégraphie.

Dans la première section de ce chapitre, nous allons présenter une vision générale de l'exécution repartie d'une orchestration. Ensuite, nous montrerons l'architecture mise en place pour supporter ce type d'exécution. Puis, nous décrivons comment, en utilisant les mécanismes d'extension du canevas FOCAS, nous avons implanté cette architecture. Finalement, une synthèse de l'approche sera faite.

8.1 L'EXECUTION REPARTIE DE L'ORCHESTRATION

8.1.1 Vision générale

Notre proposition cherche à utiliser les avantages de l'orchestration et de la chorégraphie. De la première approche nous reprenons l'expression d'un modèle global pour spécifier la logique de l'orchestration, de la deuxième nous prenons le modèle d'exécution reparti. Il est important de clarifier que notre objectif n'est pas d'assurer un protocole de collaboration entre les services comme dans l'approche de chorégraphie pure. Nous nous sommes plutôt intéressés à utiliser une architecture repartie pour l'exécution de l'orchestration qui permettrait d'améliorer les performances et de permettre le passage à l'échelle des orchestrations.

L'idée est que, une fois le modèle d'orchestration spécifié, un ensemble d'annotations expriment la distribution logique de l'orchestration pour son exécution. Ces annotations vont indiquer quelles activités seront exécutées sur quel nœud du réseau, ensuite un algorithme utilise la spécification du modèle global et les annotations pour calculer les fragments du modèle qui seront exécutés par chaque nœud. De plus, un ensemble de tables de routage sont calculées pour qu'elles soient utilisées par les mécanismes de communication de l'architecture.

Nous profitons de certaines caractéristiques du modèle de contrôle afin de réussir la distribution d'une orchestration. D'abord, le concept d'activité est utilisé comme contexte d'exécution de tout modèle Appel, et une activité peut être composite. Par conséquent une activité à n'importe quel niveau d'un modèle de contrôle peut être considérée comme un modèle complet. Etant donné que notre modèle d'orchestration utilise une logique de flots de données, c'est-à-dire qu'une transmission explicite des données entre les activités du modèle est nécessaire, nous n'avons pas à gérer des contextes de données partagés entre des activités reparties. Par conséquent, toute activité d'un modèle APEL peut être considérée comme un modèle complet et valide, pouvant être exécuté par l'interpréteur.

Afin d'illustrer notre approche nous allons introduire un exemple d'orchestration qui sera exécuté de façon repartie. L'application consiste en un service de localisation d'un hôtel et un restaurant et le calcul de l'itinéraire pour aller de l'hôtel au restaurant. L'application commence par la réception des données de localisation et préférences de l'utilisateur, ensuite deux flux parallèles sont exécutés. Le premier flux invoque un service permettant de localiser un hôtel dans la zone (activité *Hotels*), ensuite un service d'annuaire fournit l'adresse de l'hôtel en utilisant comme paramètre son nom (activité *AddressH*). Le deuxième flux utilise un service de localisation de restaurants (activité *Restaurants*) pour trouver le plus prêt de la localisation de l'utilisateur avec les préférences indiquées, ensuite à nouveau le service d'annuaire est utilisé, cette fois-ci pour récupérer l'adresse du restaurant (activité *AddressR*). Une fois l'exécution des deux flux terminé, un service de cartes (activité *Maps*) qui utilise comme paramètres deux adresses est invoqué pour calculer l'itinéraire pour aller de l'hôtel au restaurant. Finalement, un service qui envoie un mail à l'utilisateur avec l'itinéraire est appelé (activité *SendMail*).

Une distribution possible de notre modèle de contrôle est exprimée par les annotations faites sur l'exemple. Ces annotations indiquent que les activités *Hotels* et *AddressH* seront exécutées dans le nœud *N1* du réseau, les activités *Restaurants* et *AddressR* dans le nœud N2 et finalement les activités *Maps* et *SendMail* dans le nœud N3. Le modèle de contrôle correspondant à l'exemple de calcul d'itinéraire ainsi comme sa distribution sont présentés dans la Figure 68.

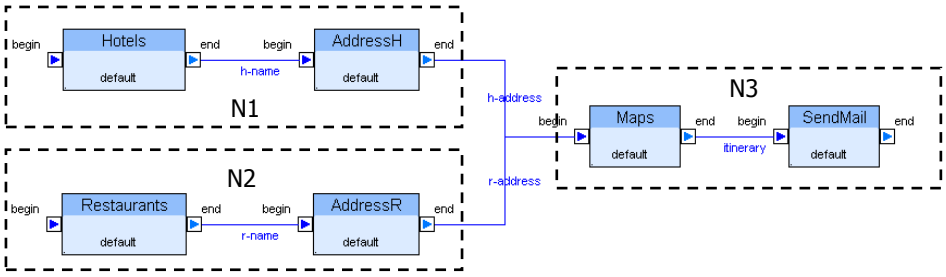

Figure 68. Modèle de contrôle pour l'exemple d'orchestration repartie.

8.1.2 Raisons pour la distribution

Il existe deux raisons principales pour lesquelles une exécution repartie des orchestrations est souhaitable : l'amélioration de la performance des orchestrations et le passage à l'échelle de l'architecture. Nous allons, maintenant présenter comment une exécution repartie de l'orchestration aide à atteindre ces objectifs.

Elimination du goulot d'étranglement

Les orchestrations de services sont par nature des applications distribuées, en fait les services peuvent être exécutés par des machines différentes de la machine qui exécute l'orchestration. Par contre, l'orchestration elle-même est exécutée en centralisé, ce qui implique que toutes les communications arrivent et sortent du nœud du réseau qui l'exécute ; ce nœud deviens alors un goulot d'étranglement.

La charge de la machine d'exécution est lourde car elle doit traiter tous les messages entrants et sortants, en plus de maintenir l'état d'exécution et la persistance de chaque instance d'orchestration, par conséquent le passage à l'échelle est compromis dans cette architecture. En outre, la performance des orchestrations se dégrade dû au fait d'être exécutées par une seule machine.

Réduction des coûts de communication

Une exécution repartie qui exécute sur la même machine un fragment d'orchestration et les services appelés par ce fragment diminue le trafic du réseau et le temps de la communication. Nous considérons que le placement optimal est celui qui minimise le coût de la communication, soit en termes du trafic du réseau soit en terme de temps de réponse des services.

Prenons dans notre exemple la séquence d'activités *Hotels-AddressH-Maps-SendMail*, pour illustrer comment le coût de communication est diminué par une distribution déterminée. Considérons que le temps d'exécution et les protocoles d'invocation pour les services associés aux activités de l'orchestration depuis les nœuds N1 et N2, sont spécifiées dans la Figure 69.

Si l'application est exécutée de façon centralisée dans le nœud N1, le temps de communication de l'application t_{cc} est égal à $(t_{h1} + t_{d1} + t_{m1} + t_{s1})$. D'autre part, le temps de communication lorsque l'application est repartie t_{cr} est égal à $(t_{h1} + t_{d1} + t_{m2} + t_{s2} + t_{12})$, en supposant une distribution comme la présentée dans la Figure 68 (*Hotels* et *Addresh* dans le nœud N1 et *Maps* et *SendMail* dans le nœud N2). Le temps gagné dû à l'exécution repartie de cette orchestration t_g est égal à $(t_{m1} + t_{s1}) - (t_{m2} + t_{s2} + t_{12})$. Dans le cas de notre exemple, le gain en performance est considérable étant donné que le temps d'invocation en utilisant le protocole SOAP (t_{m1} et t_{s1}) est de plusieurs ordres de grandeur supérieur à un protocole d'invocation local de méthode (t_{m2} et t_{s2}).

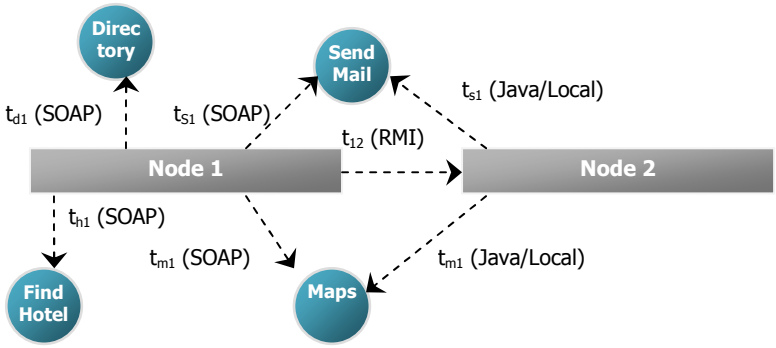

Figure 69. Temps et protocoles d'invocation pour l'application d'itinéraire.

D'auprès [JKH+04], l'invocation d'une opération d'un service utilisant comme paramètre un type de donnée simple (*int*, *long*, *float*, *double*, *boolean*) et le protocole SOAP/HTTP est en moyenne de 2,224 ms, dans le cas du protocole RMI le temps moyen d'invocation est de 0,251 ms. Cette étude montre que le protocole SOAP est approximativement 8,8 fois plus lent que le protocole RMI pour le scénario testé, et par conséquence beaucoup plus lent encore lorsqu'il est comparé avec une invocation locale. Si nous prenons le temps moyens d'invocation exposés par [JKH+04], et en faisant l'hypothèse d'un temps d'invocation locale semblable à celui du protocole RMI, le temps de communication pour notre exemple exécuté en centralisé t_{ec} est égal à 8,896 ms alors que le temps de communication pour l'exécution repartie t_{er} est égal à 5,201, ce qui donne un temps gagné t_g de 3.695 ms, c'est-à-dire une réduction du 41,54% du temps de communication.

En plus du gain de temps de communication, la configuration utilisée en notre exemple implique une réduction du trafic du réseau car les services *MapsService* et *SendMail* sont invoqués localement depuis le nœud *N2*.

Amélioration du parallélisme

Une orchestration qui contient des branches pouvant être exécutées en parallèle peut profiter d'un vrai parallélisme. Pour atteindre cet objectif, les branches deviennent des morceaux d'orchestration exécutés chacun dans un nœud du réseau.

Dans l'exemple de l'application d'itinéraire de la Figure 68, deux branches peuvent être exécutées en parallèle, la branche formée par les activités *Hotels* et *AddressH* et celle formée par les activités *Restaurants* et *AddressR*. Bien qu'une implémentation *multithread* du moteur simple de procédé puisse être utilisée, une vraie exécution parallèle est réussie seulement si nous plaçons chaque branche du contrôle dans un nœud d'exécution différent. En utilisant une implémentation *monothread* du moteur simple de procédé, l'exécution des deux branches est réalisée comme si c'était une séquence d'activités, donc le temps d'exécution est approximativement le double que si elles étaient reparties dans deux nœuds différents comme est le cas de la distribution proposée.

8.1.3 Les défis

Pour l'implémentation de notre approche d'exécution repartie de l'orchestration, plusieurs défis se posent, parmi lesquels:

- La sémantique d'exécution d'un modèle d'orchestration global doit être respectée, c'est-à-dire que le résultat de l'exécution repartie du modèle est équivalent à son exécution centralisée (au temps d'exécution près !).
- L'expression de la distribution est indépendante du modèle d'orchestration à repartir. Nous voulons que ce soit un acteur humain qui détermine comment le modèle d'orchestration sera reparti. Certaines approches comme [BSD03] [BDS05]

[CCMN04] essayent de calculer de façon automatisée la distribution du modèle d'orchestration.

- Chaque nœud du réseau n'a à connaître que le morceau du modèle qu'il est responsable d'exécuter. Dans [MM05], une exécution repartie est présentée, mais le modèle global est connu par chaque nœud participant.
- Les artefacts du *runtime* FOCAS ne doivent pas être modifiés. Le moteur centralisé d'exécution de procédés et la machine SAM sera utilisé tel quel dans l'architecture repartie. Le comportement de ces composants doit être similaire à celui d'une orchestration centralisé.
- L'architecture supportant l'exécution dans tous les nœuds doit être la même. Le déploiement de l'infrastructure pour supporter l'exécution d'une orchestration ne dépend pas de l'application à exécuter.
- Un même nœud doit supporter l'exécution potentiellement simultanée de plusieurs orchestrations et l'exécution de plusieurs instances de la même orchestration même si elle est repartie de manière différente.

8.1.4 Les travaux sur des exécutions reparties pour l'orchestration

Il existe des travaux proposant des modèles d'exécution repartie d'une orchestration (ou de *workflows*). Nous allons par la suite citer certains de ces travaux et les comparer à notre approche.

Dans [JCSS01], une implémentation repartie d'un WFMS est présentée, l'exécution des instances de procédé est déléguée aux différents nœuds du réseau. L'objectif principal de cette approche est le passage à l'échelle, des algorithmes de répartition de charge son utilisés. Dans notre approche, nous ne cherchons pas à évaluer la charge de chaque nœud, mais cela reste une possibilité intéressante pour décider une reconfiguration dynamique de l'exécution d'une orchestration.

Dans [PHA07] il est proposé un moteur qui divise l'exécution d'une orchestration en deux tâches principales : l'exécution du contrôle et l'invocation de services. Des composants capables de réaliser une de deux tâches sont repartis dans un réseau, et des mécanismes autonomiques sont utilisés afin de choisir la configuration optimale pour la distribution de ces composants. Dans ce cas, c'est le moteur d'exécution qui est reparti est par conséquent l'exécution de l'orchestration ; mais l'orchestration n'est pas explicitement repartie. Nous utilisons plutôt une architecture de pairs, dans laquelle chaque nœud participant à l'exécution possède les mêmes types des composants.

Dans [CCMN04] il est proposé un modèle d'exécution décentralisé pour une orchestration exprimée en BPEL4WS. Cette approche essaye de calculer les morceaux d'orchestration qui doivent être exécutés dans chaque nœud afin de maximiser la performance de l'exécution. Les algorithmes utilisés se basent sur l'analyse des programmes (analyse de flux de contrôle et de données) pour calculer la distribution optimale. Dans notre approche, c'est l'utilisateur qui décide la distribution de l'orchestration, cependant l'utilisateur peut décider d'utiliser des méthodes automatisées pour calculer la distribution.

8.2 ARCHITECTURE DE L'ORCHESTRATION REPARTIE

Nous allons maintenant introduire l'architecture utilisée par notre approche pour supporter l'exécution repartie de l'orchestration. L'architecture de l'infrastructure mise en place peut être entendue comme un système de pairs, où chaque nœud participant à l'exécution du modèle global fonctionne comme un pair. Le nœud est chargé d'exécuter son fragment du modèle d'orchestration global, mais aussi d'assurer la communication avec les nœuds ayant besoin d'interagir avec lui. Dans la Figure 70 est schématisée l'architecture de l'infrastructure

pour l'exécution d'une orchestration repartie, l'application de calcul d'itinéraire repartie est présentée avec l'infrastructure.

Un nœud d'exécution d'orchestration contient un ensemble de composant génériques, c'est-à-dire qui ne dépendent pas de l'orchestration à exécuter. Ces composants correspondent à l'infrastructure fournit par FOCAS, donc le moteur simple d'exécution de procédés et la machine à services SAM. Ces composants n'ont pas été modifiés et leur comportement est identique au cas centralisé. De nouveaux composants sont ajoutés pour assurer la communication entre les nœuds, à savoir les composants ODC (Output Data Component) et IDC (Input Data Component).

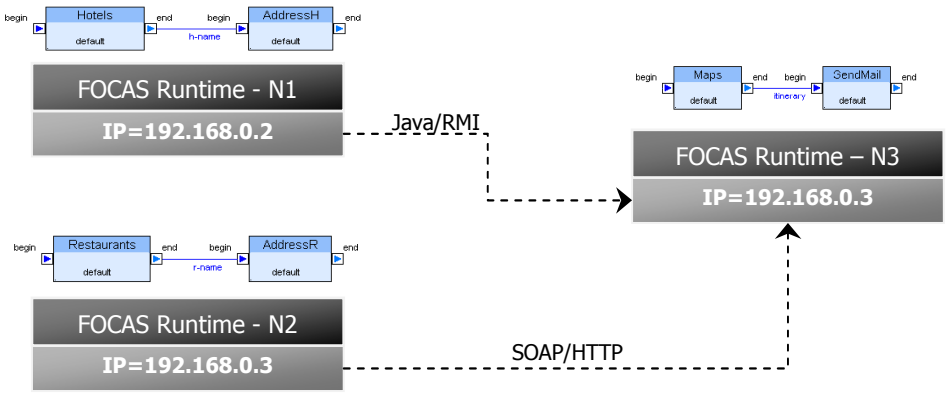

Figure 70. Architecture de l'orchestration repartie pour l'exemple de calcul d'itinéraire.

Sans aide spécifique, le développeur devrait fournit pour chaque nœud, l'orchestration valide, spécifie de façon classique (composition des trois modèles de base) devant être exécutée sur ce nœud. Ensuite, le développeur devrait définir une table de routage indiquant comment les messages sortant du fragment d'orchestration qui s'exécute dans un nœud, doivent être dirigées vers les nœuds destinataires. Finalement, le composant de liaison doit être aussi spécifié pour chaque nœud, comme dans une orchestration classique. La Figure 71 schématise l'architecture d'un nœud participant à l'exécution repartie d'une orchestration. Nous allons maintenant préciser les nouveaux composants ajoutés à l'architecture pour supporter la communication entre les nœuds, donc IDC et ODC.

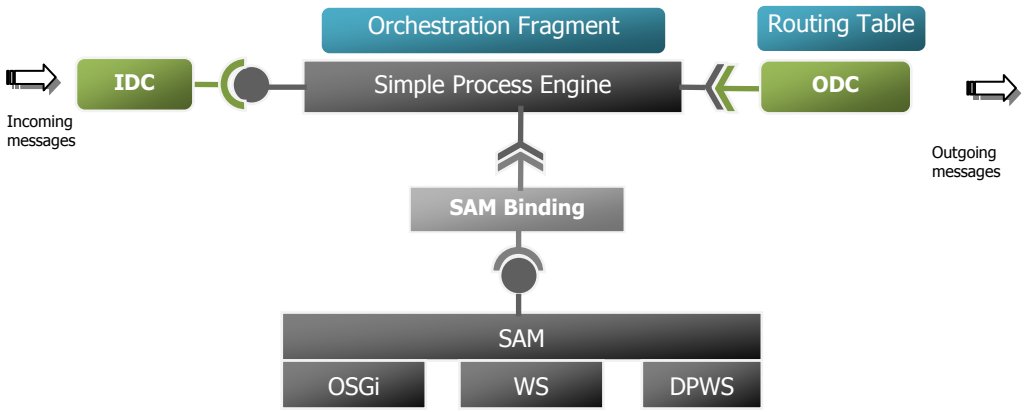

Figure 71. Architecture d'un nœud de l'exécution repartie de l'orchestration.

Le composant ODC

Le composant ODC (*Output Data Component*) est chargé d'envoyer les données depuis un nœud vers les autres nœuds de l'architecture répartie. Ce composant est configuré en utilisant une table de routage qui indique où les messages doivent être envoyés. Cette table est exprimée

en termes des nœuds logiques. Par exemple, la table de routage qui sera déployée dans le nœud N1 pour l'application de calcul d'itinéraire est présentée dans la Figure 72. Cette table indique que les données arrivant au port *end* de l'activité *P1/AddressH* doivent être transmises au port *begin* de l'activité *P3/Maps*. *P1* et *P3* correspondent aux procédés s'exécutant dans le nœud N1 et N3 respectivement.

	Nœud N2	
Activité	P1/AddressH	P3/Maps
Port	end	begin

Figure 72. Table de routage déployé dans le nœud N1.

Une autre table indiquant l'information concrète, fait aussi partie de la configuration du composant ODC. Cette table indique l'adresse réseau correspondant aux nœuds logiques et les protocoles de communication à utiliser pour communiquer avec eux. La table de configuration physique pour le nœud N1 est présentée dans la Figure 73

Nœud			N1		
Nœud	*N1*	**IP**	*192.168.0.2*	**Protocole**	*Java/Local*
Nœud	*N2*	**IP**	*192.168.0.3*	**Protocole**	*SOAP/HTTP*
Nœud	*N3*	**IP**	*192.168.1.5*	**Protocole**	*Java/RMI*

Figure 73. Table des adresses et des protocoles pour le nœud N1.

Le composant ODC communique avec le moteur simple d'exécution de procédés en utilisant une interface de notification d'événements. Le mécanisme d'intégration est similaire à celui employé par le composant de liaison, par contre les événements écoutés sont l'arrivée des données dans les ports et non les changements d'états des activités. Lorsqu'une instance de données arrive dans un port, le composant ODC la retire du port, construit le message à envoyer et utilise l'information de la table de routage pour diriger le message.

Il est important de remarquer que la connaissance disponible pour chaque nœud de l'architecture est limitée au fragment d'orchestration à exécuter et la table de routage, ce qui minimise la quantité de données à transférer d'un nœud à un autre. En plus, comme l'orchestration globale est ignorée, une reconfiguration dynamique de la topologie de l'orchestration est possible. Cette reconfiguration est faite en modifiant les tables de configuration (routage) du composant ODS. Deux possibilités sont possibles, soit par le changement d'un nœud logique par un autre dans la table de routage, soit par le changement d'une adresse dans la table des adresses de réseau.

Le composant IDC

Le composant IDC (*Input Data Component*) est chargé de recevoir les messages provenant des autres nœuds, ensuite de créer des instances de données et finalement de placer ces instances de données dans les instances de ports correspondants. Les messages dans notre architecture sont auto-contenus, c'est-à-dire que le message envoyé par un ODC indique le modèle d'orchestration et son instance spécifique à laquelle envoyer le message. Le composant IDC utilise l'API du moteur simple d'exécution de procédés pour placer les instances de données dans les ports.

8.3 EXTENSIONS DE FOCAS POUR L'ORCHESTRATION REPARTIE

Afin de supporter la mise en place de l'exécution repartie de l'orchestration, nous avons étendu notre canevas pour supporter la propriété non-fonctionnelle de la distribution. L'extension à lieu tant au *runtime* comme au niveau de l'environnement CADSE-FOCAS. La modification du *runtime* consiste en l'ajout des composants à l'architecture (IDC et ODC), que nous avons présenté dans la section précédente.

Au niveau de l'environnement CADSE-FOCAS, l'extension est réalisée en utilisant les mécanismes destinés à cet effet pour les aspects non-fonctionnels. Donc, d'une part nous avons développé un composant de spécification et d'autre part nous avons implémenté un générateur

de code pour produire les fragments d'orchestration à exécuter dans chaque nœud. Nous allons dans cette section, présenter les extensions réalisées à l'environnement CADSE-FOCAS pour supporter l'aspect non-fonctionnel de la distribution.

8.3.1 Spécification de la distribution

En suivant notre approche, la distribution est exprimée à différents niveaux d'abstraction, au moment de la spécification en termes abstraits, et au moment de déploiement en termes concrets. Ainsi, nous avons spécifié un méta-modèle représentant une topologie de réseau, le méta-modèle sert à spécifier des graphes où les nœuds correspondent aux machines du réseau et les arcs à des interconnexions. Des informations concernant les caractéristiques des machinés physiques ne sont pas exprimées par ce méta-modèle.

Ensuite, des annotations seront utilisées pour associer une activité (atomique ou composite) d'un modèle de contrôle à un nœud du réseau. La sémantique de l'annotation indique que l'activité sera exécutée dans le nœud signalé. Étant donné qu'un modèle de contrôle à une structure d'arbre, si une activité n'est pas annotée, elle sera exécutée dans le contexte de son activité englobant.

Un autre méta-modèle contentant des concepts plus concrets, comme par exemple l'adresse réseau des machines, ainsi que les protocoles de communication, est utilisé au moment du déploiement des orchestrations. Un modèle contenant cette information concrète est alors utilisé pour réaliser une mise en relation entre un nœud logique et la machine physique qui doit exécuter le fragment de l'orchestration.

Comme pour les autres extensions supportant des aspects non-fonctionnels, pour exprimer les annotations de la distribution, nous avons étendu l'éditeur d'APEL en incluant une nouvelle page de propriétés. Dans la Figure 74, cette extension est présentée pour notre exemple de calcul d'itinéraire, l'activité *Hotels* (à gauche dans la figure) est annotée pour indiquer qu'elle sera exécutée dans le nœud N1, ainsi que l'activité *Restaurants* (à droite dans la figure) pour être exécutée dans le nœud N2.

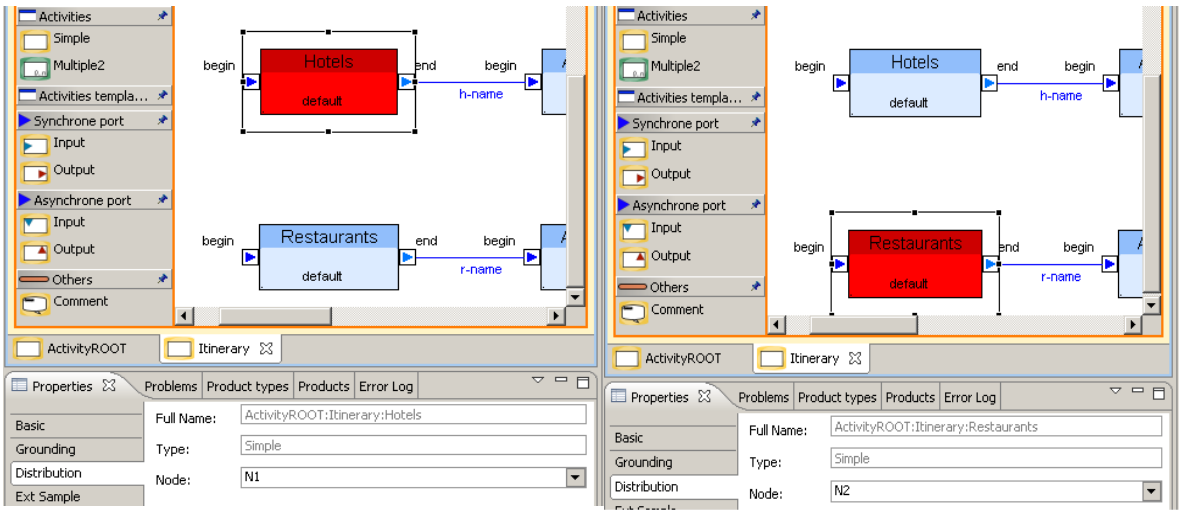

Figure 74. Page de propriétés exprimant la distribution d'une orchestration.

Pareillement à toutes les annotations, cette spécification est indépendante du modèle d'orchestration, ce qui permet d'annoter autrement le même modèle afin de spécifier une distribution différente pour la même orchestration.

8.3.2 Génération du code

La tâche de génération de code pour la distribution des orchestrations est différente des tâches réalisées pour les autres types d'extensions comme la sécurité ou le traçage

d'événements. En fait, nous n'allons pas générer du code source qui devra être intégré avec le moteur simple d'exécution des procédés. Plutôt, nous allons générer les fragments (sous modèles) d'orchestration qui seront exécutés dans chaque nœud du réseau, ainsi que les tables de routage utilisées pour configurer les composants de communication reliant ces sous modèles.

Le CADSE-FOCAS a été étendu en utilisant le point d'extension offert par l'environnement pour la création de générateurs de code, ce que permet de profiter des APIs de CADSE (items logiques), pour accéder facilement aux modèles et aux répertoires des projets Eclipse impliqués dans la spécification d'une orchestration.

Un algorithme de division du modèle global d'orchestration a été développé en utilisant la technologie EMF. L'algorithme est appliqué d'abord au modèle de contrôle et ensuite les autres modèles (données et services) sont coupés pour correspondre aux fragments de contrôle générés. Nous allons, par la suite présenter l'algorithme de division utilisé par le générateur.

Algorithme de division du modèle global

L'algorithme de division du modèle global est appliqué sur chaque activité composite du modèle de contrôle d'une orchestration. L'algorithme s'applique à un modèle global en deux étapes, à savoir l'étape de détermination du contexte d'exécution et l'étape d'analyse des flots de donnés (*dataflows)*.

Pour illustrer l'algorithme de division, nous allons utiliser un exemple simple qui comprend une activité composite *Comp* contenant trois sous-activités *SubAct1*, *SubAct2* et *SubAct3* comme schématisé dans la Figure 75. Cet exemple sera modifié au fur et à mesure que les différents cas seront exposés.

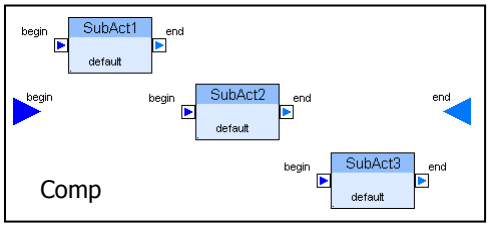

Figure 75. Exemple de base pour l'algorithme de division de l'orchestration repartie.

Etape 1 : Détermination du contexte d'exécution

Dans cette étape, l'algorithme détermine si une sous-activité est exécutée dans un nœud différent de son activité englobante. Étant donné que chaque sous-activité nécessite un contexte d'exécution (son activité englobante), pour chaque ensemble d'activités qui vont s'exécuter dans un nœud différent, l'algorithme produit une activité de « contexte ». L'activité de contexte est un clone de l'activité composite originale, elle sera nommée de la même façon plus le suffixe *Context*, plus le nom du nœud, en utilisant la notation suivante : *NomActivité*_Context_*NomdeNœud*. Pour les activités s'exécutant dans le même nœud que l'activité composite, l'activité de contexte a le même nom que l'activité originale. Ensuite, dans chaque activité de contexte sont placées les sous-activités qui correspondent avec son nœud. Les *dataflows* entre des activités allant dans la même activité contexte sont aussi répliqués.

Pour illustrer cette étape, l'exemple de base a été modifié en ajoutant un *dataflow* entre la sous-activité *SubAct2* et la sous-activité *SubAct3*. L'activité *Comp* de l'exemple est annotée pour être exécutée dans le nœud N1, tandis que les sous-activités *SubAct2* et *SubAct3* sont annotées pour être exécutés dans le nœud N2. Étant donné que la sous-activité *SubAct1* n'est pas annotée, elle sera exécutée dans le contexte de l'activité composite *Comp*.

L'algorithme, pour le nœud N1, produit l'activité de contexte *Comp* et à l'intérieur, il place la sous-activité *SubAct1*. Pour le nœud N2, l'activité contexte *Comp_Context_N2* est produite, à l'intérieur sont placés les sous-activités *SubAct2* et *SubAct3* et le *dataflow* qui les

relie. Dans la Figure 76 sont présentés l'activité composite originale et les activités de contexte produites pour le nœud N1 et N2.

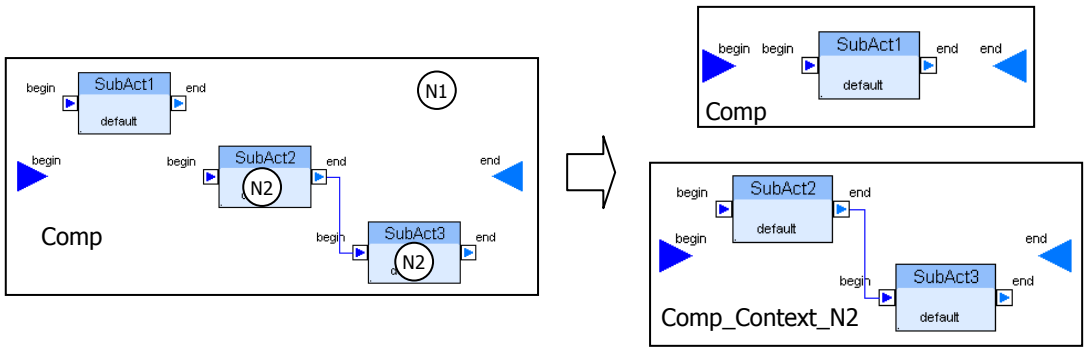

Figure 76. Exemple pour l'étape de détermination du contexte d'exécution.

Etape 2 : Analyse des dataflows

Une fois déterminé le contexte d'exécution pour les sous-activités, une analyse des *dataflows* est réalisé. L'objectif est de découvrir quels *dataflows* seront « repartis » dû au fait qu'ils relient des activités qui s'exécutent dans des nœuds différents. Le composant ODC de l'architecture est chargé d'envoyer les messages vers les nœuds distants, donc ce composant effectue la fonction d'un *dataflow* distant. Par conséquent, l'algorithme de division, en plus des fragments de modèles d'orchestration, produit des tables de routage pour configurer le composant ODC dans chacun des nœuds du réseau.

Le mécanisme que le composant ODC utilise, consiste en capturer l'arrivée des données dirigées à une activité distante, et ensuite à envoyer ces données au nœud où l'instance de l'activité s'exécute. Pour réaliser la capture, une activité *proxy* est placé dans le nœud d'origine, le composant ODC capture l'arrive des données pour l'activité *proxy*, et ensuite il les envoie vers le nœud destination. Le composant IDC de ce nœud récupère les données et les place dans l'instance correspondant à la « vraie » activité.

Dans l'étape d'analyse des *dataflows*, il existe trois cas possibles selon l'activité origine et destination du *dataflow*. Nous allons maintenant présenter le traitement fait par l'algorithme pour chacun de ces cas.

Cas 1 : Un dataflow d'une activité composite vers une sous-activité

Ce cas se présente lorsqu'une activité composite à un *dataflow* vers une sous-activité qui s'exécute dans un nœud différent. Dans ce cas, une sous-activité « *proxy* » est créée à l'intérieur de l'activité composite, cette activité est un clone de la sous-activité originale, mais elle ne possède pas de *dataflows* sortants. La sous-activité *proxy* est nommée de la même façon que la sous-activité originale plus le suffixe *Proxy*. Finalement, un *dataflow* reliant l'activité contexte composite et l'activité *proxy* est créé.

Pour illustrer ce cas, considérons l'exemple de base mais en ajoutant un *dataflow* de l'activité composite Comp vers la sous-activité SubAct1. Les annotations indiquent que l'activité Comp et la sous-activité SubAct2 seront exécutées dans le nœud N1, pendant que les sous-activités SubAct1 et SubAct3 seront exécutées dans le nœud N2. L'exemple est présenté à gauche de la Figure 77. Les activités contextes produites sont présentées à droite de la Figure 77, la sous-activité *proxy* SubAct1_Proxy est incluse dans l'activité de contexte *Comp* pour représenter la « vraie » activité qui s'exécute *SubAct1* dans le nœud N2.

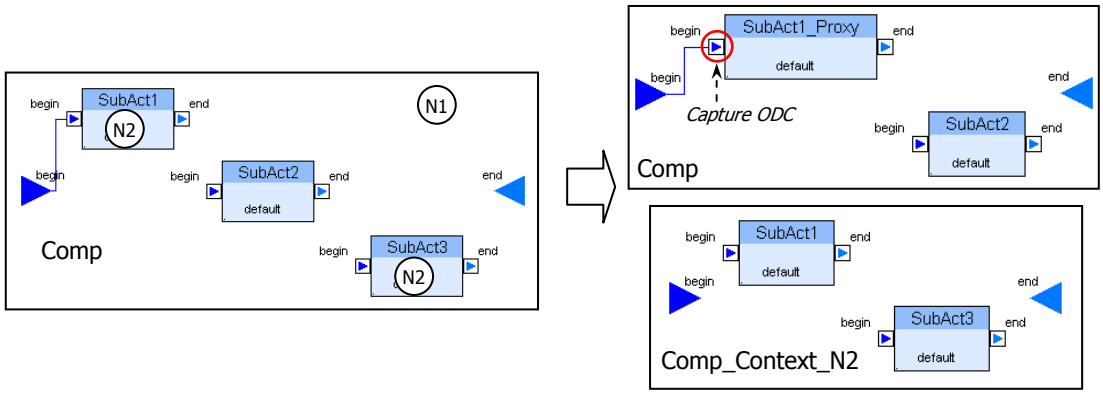

Figure 77. Résultat de l'algorithme : *dataflow* d'activité composite vers une sous-activité.

L'événement associé à l'arrivée de données dans le port destination d'un *dataflow* d'une sous-activité *proxy* doit être écouté pour le composant ODC afin de capturer les données et les envoyer vers la « vraie » sous-activité. Pour notre exemple, le port « intercepté » par le composant ODC est le port *begin* de la sous-activité *SubAct1_Proxy* dans le nœud N1, ces données sont envoyées vers le port *begin* de la sous-activité *SubAct1* du nœud N2. Le composant ODC est configuré avec la table de routage pour réaliser cette fonction, la table de routage générée par l'algorithme pour le nœud N1 est présentée dans la Figure 78.

	Nœud N2	
Activité	Comp/SubAct1_Proxy	Comp_Context_N2/ SubAct1
Port	begin	begin

Figure 78. Table de routage pour le nœud N1.

Cas 2 : Un dataflow d'une sous-activité vers une l activité composite

Ce cas se présente lorsqu'une sous-activité possède un *dataflow* vers son activité composite qui s'exécute dans un nœud différent. Du fait que l'activité composite et la sous-activité s'exécutent sur différents nœuds, une activité de contexte est produite par l'étape 1 de l'algorithme comme contexte d'exécution de la sous-activité. Ainsi, il n'est pas nécessaire de créer une activité *proxy* car l'activité de contexte peut être utilisée à sa place. Finalement, un *dataflow* reliant la sous-activité et l'activité de contexte est créé.

Pour illustrer ce cas, considérons l'exemple de base mais en ajoutant un *dataflow* de la sous-activité *SubAct3* vers l'activité composite *Comp*. Les annotations indiquent que l'activité *Comp* et la sous-activité *SubAct2* seront exécutées dans le nœud N1, pendant que les sous-activités *SubAct1* et *SubAct3* seront exécutées dans le nœud N2. L'exemple est présenté à gauche de la Figure 79 et les activités contextes produites sont présentées à droite.

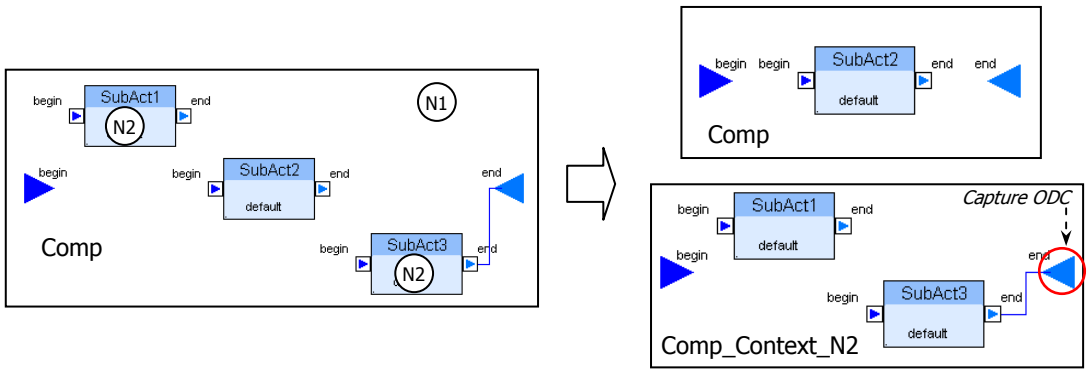

Figure 79. Résultat de l'algorithme : *dataflow* d'une sous-activité vers l'activité composite.

Pour ce cas, rien n'est ajouté aux modèles de contrôle dans la deuxième étape de l'algorithme, cependant étant donné que l'activité de contexte est un clone de l'original, le

composant ODC capture l'arrivé des données à ses ports et les envoi vers la « vraie » activité. La table de routage permet de spécifier la capture de données dans les ports de l'activité de contexte. Pour notre exemple, le port *end* de l'activité de contexte Comp_Context_N2 est intercepté et ses données envoyées ver le port *end* de la « vraie » activité *Comp* qui s'exécute dans le nœud N1. La Figure 80 montre la table de routage pour le nœud N2 pour le cas et l'exemple analysés.

	Nœud N2	
Activité	Comp_Context_N2	Comp
Port	end	end

Figure 80. Table de routage pour le nœud N2.

Cas 3 : Un dataflow entre activités du même niveau

Ce cas se présente lorsque deux sous-activités exécutées dans des nœuds différents et qui ont un *dataflow* qui les relie. Dans ce cas, l'algorithme suit la même stratégie que pour le cas 1, c'est-à-dire la création d'une sous-activité *proxy* pour représenter la « vraie » sous-activité s'exécutant dans l'autre nœud. La sous-activité *proxy* correspond à la sous-activité destination du *dataflow* et elle est incluse dans l'activité contexte de la sous-activité origine du *dataflow*. Pareillement, la sous-activité *proxy* est nommée de la même façon que la sous-activité originale plus le suffixe *Proxy*. Finalement, un *dataflow* reliant la sous-activité et la sous-activité *proxy* est créé.

Pour illustrer ce cas, considérons l'exemple de base mais en ajoutant un *dataflow* de la sous-activité *SubAct1* à la sous-activité S2, et un autre de la sous-activité *SubAct2* à la sous-activité *SubAct3*. Les annotations indiquent que l'activité *Comp* et la sous-activité *SubAct1* seront exécutées dans le nœud N1, la sous-activité *SubAct2* dans N2 et la sous-activité *SubAct2* dans N3 comme est présenté à gauche de la Figure 81.

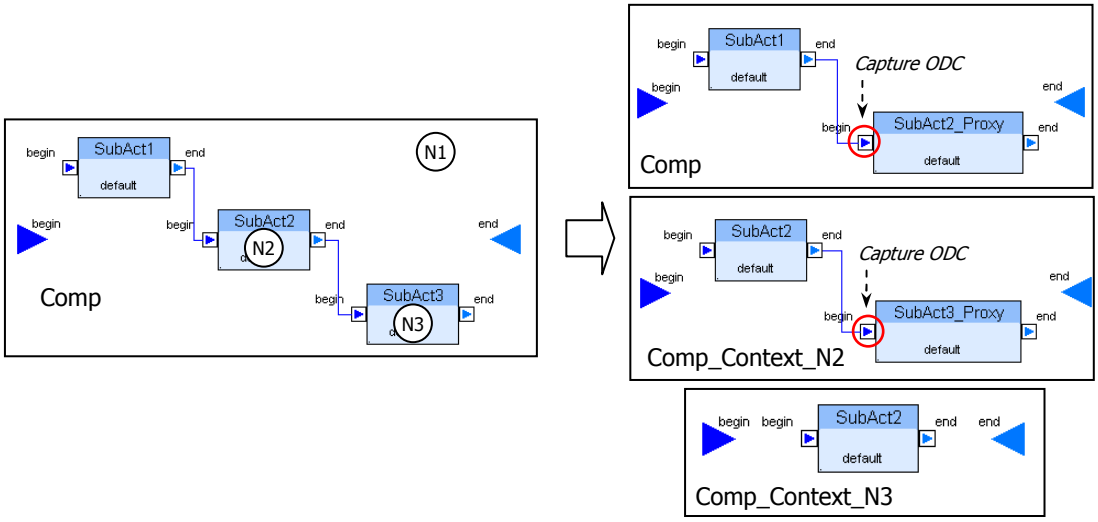

Figure 81. Résultat de l'algorithme : *dataflow* entre activités du même niveau.

La première étape de l'algorithme produit les trois activités de contexte, *Comp*, *Comp_Context_N2* et *Comp_Context_N3*. Dans l'activité de contexte *Comp* est incluse la sous-activité *proxy SubAct2_Proxy* et dans l'activité de contexte *Comp_Context_N2* est incluse la sous-activité *proxy SubAct3_Proxy* comme est présenté à droite dans la Figure 81.

Les tables de routage générées pour les nœuds N1 et N2 sont présentées dans la Figure 82.

	Nœud N2	
Activité	Comp_Context_N2	Comp
Port	end	end

	Nœud N3	
Activité	Comp_Context_N2	Comp
Port	end	end

Figure 82. Tables de routage pour les nœuds N1 et N2.

Division du modèle de l'application d'itinéraire

Nous allons maintenant montrer le résultat de l'application de l'algorithme au notre exemple d'orchestration pour le calcul d'itinéraire. L'étape de détermination du contexte d'exécution de l'algorithme donné trois activités de contexte, comme suivent :

- L'activité *ActivityROOT* est l'activité contexte pour le nœud N1. A l'intérieur sont situées les activités *Hotels* et *AddressH* et le *dataflow* qui les relie.
- L'activité *ActivityROOT_Context_N2* est l'activité contexte pour le nœud N2. A l'intérieur sont placés les activités *Restaurants* et *AddressR* et le *dataflow* qui les relie.
- L'activité *ActivityROOT_Context_N3* est l'activité contexte pour le nœud N3. A l'intérieur sont placés les activités *Maps* et *SendMail* et le *dataflow* qui les relie.

Ensuite, l'étape d'analyse des dataflows reparties ajoute des activités *proxies* comme suit :

- Dans l'activité *ActivityROOT* (nœud N1) est incluse une sous-activité *proxy Maps_Proxy* pour représenter l'activité *Maps* (nœud N3).
- Dans l'activité *ActivityROOT* (nœud N1) est incluse une sous-activité proxy *Maps_Proxy* pour représenter l'activité *Maps* (nœud N3).

Dans la Figure 83, sont présentés les trois modèles de contrôle générés par notre environnement, en utilisant les annotations proposées pour notre exemple d'application de calcul d'itinéraire.

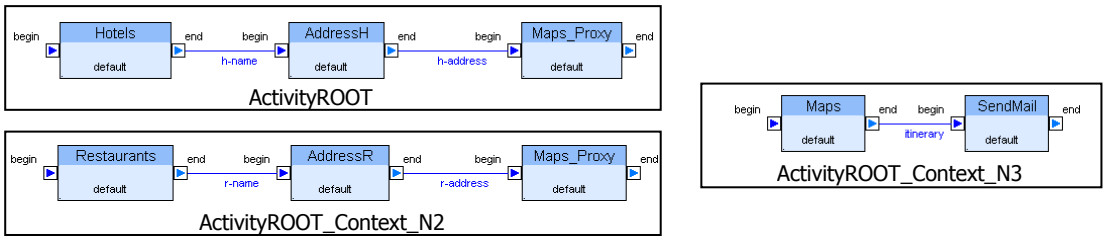

Figure 83. Modèles générés pour l'application d'itinéraire.

Les tables de routage générées pour chacun des nœuds sont présentées dans la Figure 84.

	Nœud N1	
Activité	ActivityROOT/Maps_Proxy	ActivityROOT_Context_N3/Maps
Port	begin	begin

	Nœud N2	
Activité	ActivityROOT_Context_N2/Maps_Proxy	ActivityROOT_Context_N3/Maps
Port	begin	begin

Figure 84. Tables de routage générées pour l'application d'itinéraire.

8.3.3 Déploiement des orchestrations reparties

Le fait d'utiliser une architecture repartie pour l'exécution d'une orchestration impose de nouveaux défis en termes de déploiement des orchestrations. Bien qu'une technique de déploiement ah-doc puisse être utilisé lorsqu'il s'agit d'une exécution centralisée, cette technique n'est plus envisageable lorsque nous devons déployer l'infrastructure FOCAS et les fragments d'orchestrations dans plusieurs machines d'un réseau.

Afin de couvrir ce besoin, nous avons utilisé une extension de la machine SAM permettant de déployer des composants OSGi et des ressources dans différentes machines d'un réseau. Cette extension, que nous allons nommer SAM-Deployer connaît toutes les machines SAM qui sont déployés dans le réseau. Le SAM-Deployer utilise des primitives proposées par la SAM afin de déployer des artefacts dans chaque machine du réseau. En plus, lorsqu'une nouvelle machine SAM arrive, un protocole de découverte permet de souscrire la machine au le SAM-Deployer.

Le SAM-Deployer exécute un plan de déploiement qui à une liste de machines SAM, et pour chacune de ces machines, une liste de ressources et de composants à déployer. Les composants sont installés sur la même plate-forme OSGi sur laquelle s'exécute la machine SAM (la machine SAM est un composant OSGi). Les ressources sont copiées dans les répertoires indiqués pour le plan de déploiement dans les machines distantes.

Les composants correspondant à l'infrastructure d'exécution FOCAS sont génériques, c'est-à-dire que les mêmes composants sont déployés dans chaque nœud du réseau. Ces composants génériques sont le moteur simple de procédés, ODC et IDC. Le moteur simple de procédés a été adapté pour être exécuté sur une plate-forme OSGi ; les composants ODC et IDC ont été développés directement pour cette plate-forme.

Le fragment d'orchestration à déployer dans chaque nœud correspond au modèle d'orchestration, formé par les trois modèles de base (contrôle, données et services) et ses relations. En plus, il faut déployer une table de routage pour configurer le composant ODC dans chaque nœud. Les modèles de contrôle et données et leurs relations sont déployés en tant que ressources, ainsi comme la table de routage, car ils sont des fichiers XML. Le modèle de services est empaqueté en tant qu'un composant OSGi, ce composant contient les fichiers binaires résultats de la compilation des interfaces Java utilisées pour spécifier le modèle de services. Finalement, le composant de liaison est adapté pour être exécuté dans la plate-forme OSGi. Toutes ces tâches d'empaquetage et adaptation de composants est réalisé de façon systématique pour le CADSE-FOCAS, finalement il génère aussi le script du plan de déploiement.

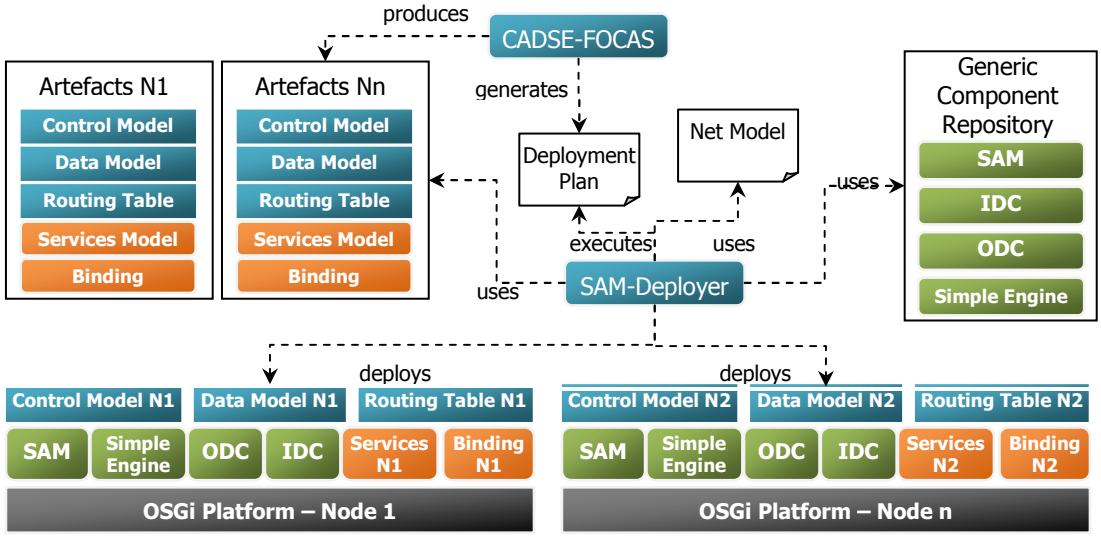

Figure 85. Génération et exécution d'un plan de déploiement.

Dans la Figure 85 est présenté comment le déploiement est réalisé, pour une orchestration déterminée. L'environnement CADSE-FOCAS produit un ensemble d'artefacts pour chaque machine du réseau et un plan de déploiement. Pour l'outil SAM-Deployer, les artefacts produits sont de deux types, des ressources (modèle de contrôle et de données et la table de routage) et des composants OSGi (modèle de services et composant de liaison). En plus, un dépôt contenant les composants génériques (SAM, IDC, ODC et le moteur simple de procédés) est aussi accédé pour l'outil SAM-Deployer. SAM-Deployer lit l'information physique du réseau et envoie les artefacts à déployer dans les nœuds pertinents.

8.4 SYNTHESE

Dans ce chapitre nous avons présenté la mise en place d'une architecture pour l'exécution distribuée d'une orchestration de services. Afin d'implémenter la distribution pour ce type d'application, nous avons repris la facilité d'expression fourni par l'approche d'orchestration de services, ainsi que l'architecture d'exécution proposé par l'approche de chorégraphie de services. L'objectif de notre approche est d'améliorer la performance de l'exécution des orchestrations de services et d'assurer le passage à l'échelle.

Nous considérons que la capacité d'exécution repartie de l'orchestration est un apport de cette thèse à l'état de l'art de la technologie d'orchestration de services. Bien qu'il existe des approches visant l'exécution distribuée des orchestrations comme nous l'avons présenté, nous considérons que les avantages de notre approche sont:

- la réutilisation du *runtime* utilisé pour l'exécution centralisée de l'orchestration. En fait, tous le composants du *runtime* ont été repris sans aucune modification et seulement un paire de composants (ODC et IDC) on été ajoutés à notre architecture.
- nous n'utilisons pas de nouveaux formalismes lors de l'expression de ce type d'applications. Plutôt, nous avons profité des certaines caractéristiques du langage APEL et de la technique des annotations pour exprimer la distribution d'une orchestration.
- l'aspect de la distribution suit aussi une approche de remontée du niveau d'abstraction et de séparation de préoccupations. Ainsi, au moment de la spécification, le modèle d'orchestration est annoté de façon abstraite avec des identificateurs des nœuds logiques. Au moment du déploiement, des machines physiques sont mappés aux nœuds logiques et des protocoles de communications sont établis.
- Nous n'essayons pas de déterminer la meilleure distribution pour une orchestration, la méthode de détermination de la distribution est laissée au choix de l'utilisateur.

L'exécution repartie de l'orchestration est en soi-même un résultat important de nos travaux, cependant nous considérons que la mise en place de cette technologie est aussi une façon de valider les propriétés d'extensibilité de notre canevas FOCAS.

Tout d'abord, nous avons étendu le *runtime* du canevas pour inclure les composants de communication IDC et ODC afin d'assurer la liaison entre les nœuds participant à l'exécution repartie d'une orchestration. L'implémentation de ces composants est réalisée d'une façon non-intrusive, c'est-à-dire qu'elle n'implique pas de modification des autres composants de l'architecture.

Ensuite, une extension au niveau de l'environnement CADSE-FOCAS est effectuée. Comme toute extension non-fonctionnelle réalisée dans le CADSE-FOCAS, nous avons spécifié le méta-modèle de la distribution, ensuite nous avons créé un composant pour spécifier la distribution d'une orchestration, ainsi le modèle de contrôle est annoté avec cette information. Puis, le point d'extension pour ajouter de générateurs de code a été utilisé pour implémenter le générateur chargé d'appliquer l'algorithme de division du modèle global pour produire les fragments (sous-modèles) à être exécuté dans chaque nœud de l'architecture.

Finalement, un autre apport de notre approche est le mécanisme de déploiement utilisé. La machine SAM nous permet de déployer l'infrastructure ainsi comme les modèles à exécuter dans chacune de machines pour participer à l'exécution d'une orchestration.

9. VALIDATION ET EVALUATION

Les chapitres précédents ont permis de détailler l'approche et la mise en œuvre d'un canevas extensible pour la construction d'applications orientées procédés. Puis, en utilisant cette technologie, nous avons mis en place une approche pour l'exécution repartie d'une orchestration de services.

Dans ce chapitre, nous allons détailler le contexte dans lequel le canevas FOCAS a été mis en place, celui des projets européens ITEA, S4ALL et SODA. Dans la première partie de ce chapitre, nous allons présenter quel a été le rôle de notre canevas dans ces projets, les besoins de chacun des projets et notre apport pour procurer une solution à ces besoins. Nous considérons que la solution fournie par notre canevas dans ces contextes d'utilisation est un moyen de valider nos travaux autour des applications orientées procédé. Ensuite, nous estimons pertinent de faire une évaluation des performances obtenues lors de l'utilisation de l'exécution répartie de l'orchestration. Ainsi, la deuxième partie de ce chapitre est dédié à la réalisation d'une expérience pour évaluer l'intérêt d'utiliser l'approche d'exécution répartie pour les orchestrations.

9.1 LE PROJET S4ALL

Le projet S4ALL a comme objectif de mettre en place une vision d'un environnement où des services peuvent facilement être créés, partagés et consommés. Pour mette en place cette vision des nouvelles technologies pour la création, la personnalisation, le déploiement et la prestation de services doivent être introduites.

Ainsi, les technologies à services doivent faciliter l'exploitation des services, trois opérations sont ainsi visées :

- la création facile de services. Dans la vision de S4ALL les services doivent être faciles à implémenter, même par de non spécialistes. Ces services peuvent potentiellement utiliser des infrastructures technologiques robustes, et des dispositifs mobiles. Les services doivent être créés en utilisant des outils graphiques permettant facilement l'intégration de fonctionnalités;
- la mise à disposition facile de services. Une infrastructure doit permettre de mettre facilement à disposition d'une communauté les services créés. La création des services doit être découplée de cette infrastructure afin de faciliter, ou même automatiser la tâche de déploiement ;
- la consommation facile des services. Les services doivent être catégorisés dans une taxonomie avec d'information sémantique pouvant améliorer leur recherche. L'idée est de retrouver le service le plus adapté aux besoins de consommateur.

Nous avons participé dans la partie qui cible la création facile de services. Notre objectif était la mise à disposition d'un environnement de spécification et d'exécution d'orchestrations de services. Cet environnement devait disposer de niveaux d'abstraction

supérieurs aux canevas existants, l'idée était de permettre la composition de services par des acteurs qui ne sont pas des experts de la technologie des services.

Les besoins du projet S4ALL pour cet environnement étaient :

- utiliser des formalismes de haut niveau d'abstraction, évitant l'inclusion de détails technologiques lors de la spécification d'une composition de services ;
- l'environnement devait être intuitif pour permettre l'expression et la composition de différentes préoccupations à prendre en compte ;
- l'approche devait prendre en compte diverses technologies de services, et faire abstraction de ces technologies au moment de spécifier l'orchestration ;
- la possibilité d'ajouter dans l'architecture des mécanismes d'adaptation afin de consommer différents services qui n'ont pas été conçus pour être utilisés dans une composition spécifique.

Pour faire face à ces besoins, nous avons d'abord préconisé l'utilisation d'une approche IDM pour mettre en place des formalismes d'haut niveau d'abstraction. Chaque formalisme exprimait une préoccupation différente de la composition. Afin de résoudre les problèmes d'hétérogénéité de technologies et d'alignement de services nous avons introduit une architecture en trois couches qui utilise le concept de service abstrait. Finalement, nous avons agrégé dans un environnement tous les outils (éditeurs, composeurs, générateurs de code) permettant spécifier et rendre exécutable une application de services.

Bien que notre participation dans le projet S4ALL, a permis valider les idées autour de la composition de domaines et de l'architecture en trois couches, nous avons observé que notre environnement de spécification ne fournissait pas le support adéquat. En fait, plutôt qu'intégrer les différents outils, ils étaient agrégés dans l'environnement, mais un mécanisme permettant une intégration aisée des outils n'existait pas à l'époque. Par conséquent, des travaux ont été repris afin de produire notre environnement CADSE-FOCAS.

9.2 LE PROJET SODA

Le travail de cette thèse a en partie été réalisé dans le cadre du projet européen ITEA SODA (acronyme de *Service Oriented Device and Delivery Architecture*). L'objectif du projet est de créer un écosystème à services offerts par divers dispositifs. Ces dispositifs doivent interagir en utilisant une infrastructure qui fournit des mécanismes de communication de haut niveau. Cet écosystème comportera un ensemble d'outils permettant la création d'applications qui utilisent les services offerts par les dispositifs, ainsi que de services pourvus par des applications métiers. La vision du projet est présentée dans la figure ci-dessous, à gauche sont schématisées les tâches du cycle de vie logiciel supportées par un ensemble d'outils. A droite, nous trouvons l'environnement d'exécution des applications. Des applications métier et de contrôle utilisent les services fournis par les divers dispositifs. Ces dispositifs agissent sur un environnement physique.

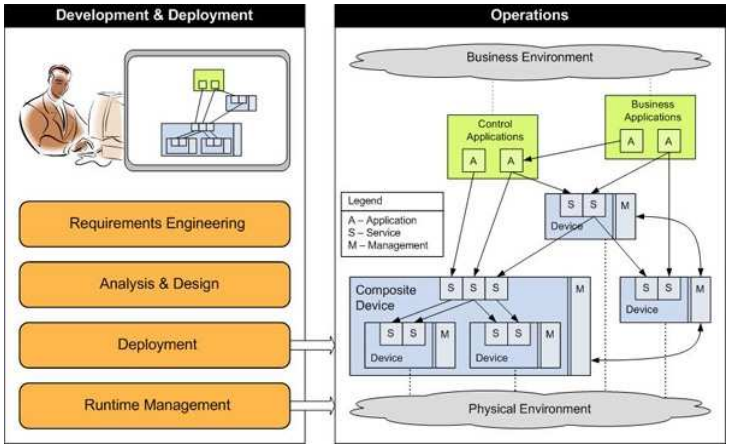

Figure 86. Ecosystème de services SODA.

Le projet a été réalisé en partenariat avec plusieurs organismes de recherche publiques et privés, tels que l'équipe Adèle du Laboratoire Informatique de Grenoble, et des industriels comme Schneider Electric, PSA et Thales. Plusieurs domaines d'application sont visés dans le projet SODA, parmi eux : l'automation industrielle, l'automobile, la domotique, la télémesure, etc.

Pour atteindre son but, SODA a divisé les efforts en quatre type de mécanismes présentés par la suite :

- mécanismes de base permettant la présentation en fonction des dispositifs en tant que services. Donc, un canevas est nécessaire pour la description de services et dispositifs. En plus, des mécanismes de communication et contrôle ainsi que de découverte des dispositifs doivent être fournis.
- mécanismes de composition de services pour construire applications qui utilisent les services fournis par les dispositifs et services offerts par des applications métiers. Donc, il s'agit des langages de spécification de composition de haut niveau d'abstraction et d'outils pour la spécification, le déploiement et l'exécution des applications.
- mécanismes pour assurer la qualité de services. Des propriétés comme la messagerie fiable, la propagation d'événements, la communication sécurisée et le monitoring et gestion de services doivent être ajoutés aux mécanismes de base.
- mécanismes pour améliorer le cycle de vie des applications. Deux stratégies sont envisagées. Tout d'abord, la combinaison d'une approche *top-down*, permettant de décomposer une application en divers services, avec une approche *bottom-up* pour réutiliser les services disponibles et offerts par l'environnement. Ensuite, la mise en place d'une approche pour séparer la vue logique des applications de l'architecture physique qui les supporte.

L'équipe Adèle du laboratoire LIG, que nous avons représenté dans le cadre de ce projet, a été chargée de fournir les langages et les outils pour supporter la spécification et l'exécution d'applications basés sur les services exposés par des dispositifs ainsi que les services métiers et techniques disponibles. Le canevas FOCAS a été utilisé pour remplir cet objectif.

Dans le cadre de notre participation, un cas d'utilisation proposé par le partenaire Thales a été implémenté et a servi comme moyen d'expérimentation de nos mécanismes d'extension. Par la suite, nous allons présenter le cas d'utilisation, sa mise en œuvre et ensuite des expériences que nous avons réalisées en profitant cette application.

9.2.1 Cas d'utilisation : système de surveillance

Le cas d'utilisation concerne la mise en place d'une application de surveillance d'une usine de production. L'application utilise un ensemble de capteurs pour récupérer des données de l'environnement physique de l'usine, ensuite les valeurs récupérées sont analysées. Le résultat de l'analyse est stocké dans une base de données afin de garder un historique. En fonction du résultat de l'analyse, des actions pour modifier le comportement des équipements de la chaîne de production peuvent être exécutes. De même, un mail ou un SMS est envoyé vers un responsable de la chaîne de production.

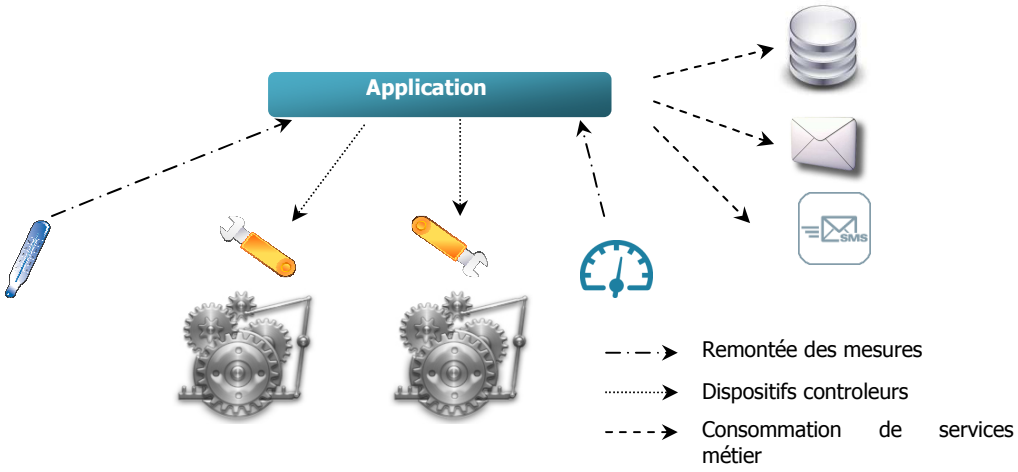

Figure 87. Environnement d'exécution de l'application de surveillance d'usine.

Dans la Figure 87, est présenté l'environnement d'exécution de l'application de surveillance de l'usine de production. Dans cet environnement existent trois types de services : tout d'abord, les services qui font la remonté de mesures, ils sont fournis par les divers capteurs. Ensuite, les services chargés du contrôle des équipements de l'usine, fournis par des dispositifs contrôleurs. Finalement, les services métiers (ou techniques) de l'entreprise pour supporter le stockage et le service de messagerie.

Les besoins de l'application

L'application à construire comporte des besoins spécifiques :

- gestion de l'hétérogénéité. Les dispositifs utilisés pour l'application sont très hétérogènes. Ils peuvent implémenter les services en utilisant différentes technologies (langages de description, représentation de données). En plus, ils peuvent utiliser plusieurs protocoles de communication comme UPnP et DPWS ;
- gestion de l'évolution. L'application doit supporter l'évolution de différents services. Par exemple, un capteur peut être remplacé par autre et l'application doit continuer à fonctionner.
- gestion de la sécurité. Etant donné l'environnement d'exécution de l'application, la sécurité est un aspect crucial, afin d'éviter la lecture ou modification des informations sensibles ainsi que la manipulation malintentionné des machines de l'usine. Donc, il s'agit de garantir l'authentification et le contrôle d'accès aux services, ainsi que de garantir la confidentialité et l'intégrité des communications.

Pour faire face à ces défis nous avons spécifié l'application en utilisant une technologie d'orchestration de services, la spécification est présentée par la suite.

Spécification de l'orchestration

L'application de surveillance d'usine a été spécifiée comme est schématisé dans la Figure 88.

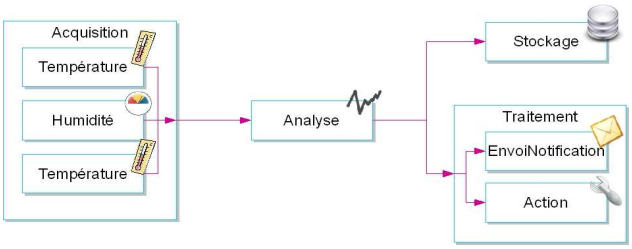

Figure 88. Spécification de l'application comme une orchestration.

Les détails de la spécification sont les suivants :

- la première partie est l'acquisition de données qui sont de températures et de taux d'humidité pour une machine ;
- ensuite, les données récupérées sont agrégées par lots de dix valeurs, puis une analyse est effectuée sur les lots de données. Cet analyse consiste en calculer la moyenne de températures et de taux d'humidité ;
- finalement, les données analysées sont stockées et traitées. Le stockage permet de garder l'historique des valeurs. Le traitement consiste en évaluer les valeurs de moyenne pour déterminer si elles dans les intervalles attendus. Si nécessaire, l'orchestration envoie un message au responsable du site (mail ou SMS), et envoi un commande sur la machine (arrêt, ralentissement…).

9.2.2 Mise en œuvre de l'application de surveillance dans FOCAS

Nous allons par la suite, montrer comment l'application de surveillance d'usine a été implémentée dans notre canevas FOCAS. L'application a été divisée en deux parties, la partie abstraite conforme par les modèles de base et leurs relations, et la partie concrète qui fait référence à la technologie d'implémentation. Nous allons présenter dans cette section, les trois modèles de base (et leurs relations), et ensuite la mise en place de la partie concrète de l'application.

Modèle de contrôle

Le modèle de contrôle pour l'application de surveillance est présenté dans la Figure 89. L'activité *Acquisition* est chargée de récolter les données des capteurs. Elle est décomposée en quatre sous-activités, deux qui se chargent des capteurs de température (*Temperature1* et *Temperature2*) et autre du capteur d'humidité (*Humidity*). La quatrième activité *Aggregator* est chargée d'attendre une mesure prise par chacun des capteurs, et ensuite envoyé un triplet avec les trois mesures par le port asynchrone *out*.

Puis, l'activité *Analysis* est chargée d'agréger dix triplets de mesures et ensuite, de calculer une moyenne des vingt mesures de température et une moyenne des dix mesures d'humidité. Le résultat de ce calcul est envoyé à l'activité *Storage* qui est chargée de stocker ces moyennes dans une base de données.

De façon parallèle, l'activité *Processing* se charge de prendre des actions selon les valeurs observées pour les mesures. L'activité *Processing*, est divisée en trois sous-activités, *ProcAnalysis* détermine si les mesures prises se trouvent dans leurs valeurs normales, si c'est le cas l'activité finalise par son port *none*. Par contre, si les mesures se trouvant dans un premier intervalle considéré anormal, l'activité *Processing* finalise par son port *mail* ayant comme conséquence l'exécution de l'activité *SendNotification* qui envoie un message au responsable de

la chaîne de production. D'autre part, si les mesures sont dans un deuxième intervalle anormal, en plus d'envoyer le message au responsable, l'activité exécute des commandes pour modifier le comportement des machines en utilisant l'activité *Action.*

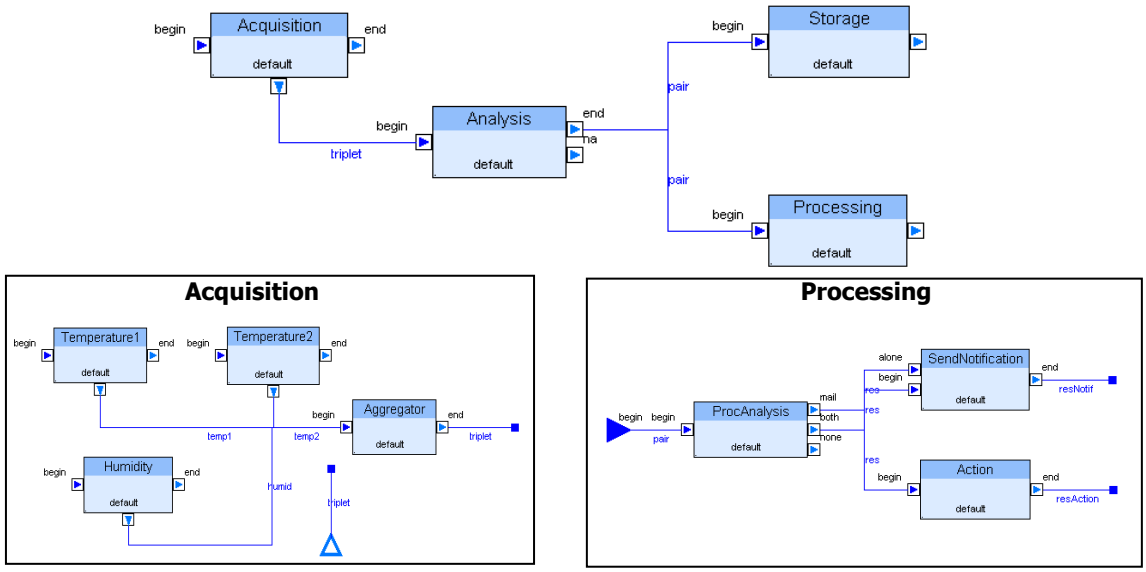

Figure 89. Modèle de contrôle pour l'application de surveillance.

Modèle de services

Le modèle de services de l'application de surveillance est présenté dans la Figure 90. Le service *GetData* est invoqué pour les activités de collection de données chaque n secondes. Il contient deux opérations, une pour récupérer une mesure de température et l'autre pour récolter une mesure d'humidité. Ensuite, le service *DoAverage* est invoqué par l'activité *Analysis* pour calculer la moyenne de mesures. De sa part, le service de *Storage* est chargé de la persistance de mesures, il est invoqué par l'activité *Storage*. Le service *SendNotification* est chargé d'envoyer le message de notification au responsable, il est invoqué par l'activité *SendNotification.* Finalement, le service *SendCommand* est invoqué pour l'activité *Action* pour agir sur les machines.

```
public interface GetData {
   public DoubleA getTemperature();
   public DoubleA getHumidity();
}

public interface DoAverage {
   public Average doAverage(LotMeasures triplets);
}

public interface Store {
   public BooleanA storeData(Average average);
}

public interface SendNotification {
   public BooleanA sendMail (MessageData mail);
   public BooleanA sendSMS (MessageData sms);
}

public interface SendCommand {
   public BooleanA runAction (IntegerA id);
}
```

Figure 90. Modèle de services de l'application de surveillance.

Modèle de données

Le modèle de données utilisé par l'application de surveillance est résumé dans le tableau ci-dessous. Pour faciliter la composition du modèle de contrôle avec celui de données, nous avons crée un type de produit dans le modèle de contrôle correspondant avec chaque type du modèle de données, en plus les types ont les mêmes noms dans les deux modèles.

Type	Description
DoubleA	Type complexe pour représenter un double. Ce type est utilisé pour garder les valeurs de mesures de température et humidité.
Measure	Type complexe conforme par un triplet de *DoubleA*. Il sert a agréger deux mesures de températures et une de humidité.
LotMeasures	Type contenant une collection de *Measure*. Il est utilisé par le service *Average* pour calculer la moyenne de 10 mesures.
MessageData	Type qui encapsule l'information pour envoyer un message (adresse, sujet, texte du message).
BooleanA	Type complexe pour représenter un booléen. Il est utilise par le service *sendCommand* pour indiquer que l'action sur la machine a été effectuée.

Figure 91. Modèle de données de l'application de surveillance.

Partie concrète de l'application

Les modèles présentés auparavant et leurs relations font partie de la définition abstraite de l'application. Mais, pour rendre cette application fonctionnelle, il est nécessaire de faire une liaison avec les services fournis par les dispositifs ainsi que les services métiers de l'entreprise. Bien que la première partie de l'application soit faite avec une approche *top-down*, une approche *bottom-up* est utilisée afin de récupérer les services déjà existants, et ainsi implémenter la partie concrète de l'application.

Nous avons ainsi intégré un générateur de code dans CADSE-FOCAS permettant à partir de la description DPWS d'un dispositif, de générer le code nécessaire pour invoquer leurs services, c'est-à-dire de générer les *proxies* d'invocation. Même si la technologie de services est connue (DPWS), les instances des services peuvent être déterminées à l'exécution, nous avons alors développé deux cas différents, le premier avec une liaison statique, comportant dans le code de liaison l'adresse de réseau de services par défaut. La deuxième implémentation utilise le mécanisme de liaison dynamique pour retrouver les instances de services à l'exécution.

En plus, nous avons généré une série de médiateurs pour permettre d'adapter les données utilisés dans notre modèle d'orchestration aux données utilisées par les services offerts par les dispositifs. Cette génération des médiateurs est partielle, c'est-à-dire les classes de médiation sont générées, mais la logique de médiation est à remplir par le développeur.

Une fois spécifié tous les modèles et leurs relations, et générés les *proxies* des services à invoquer, le composant de liaison peut être généré afin de lier les parties abstraite et concrète de l'application. Le composant de liaison est responsable pour instancier les *proxies* et invoquer les services qui sont associées aux activités en utilisant les médiateurs.

9.2.3 Mise en œuvre de l'aspect de la sécurité

La section précédente a montré comment nous avons développé l'application de surveillance d'usine partant du cas d'utilisation. Cependant, ce cas possède une partie dédiée à la sécurité de l'application. Nous allons par la suite présenter comment l'aspect de la sécurité a été ajouté à l'application de surveillance d'usine. Les besoins de sécurité exposés par le cas d'utilisation sont :

- Les données récoltées par les capteurs doivent être confidentielles.
- Les services *DoAverage* et *SendCommand* possèdent un mécanisme de control d'accès, donc une authentification préalable à l'exécution est nécessaire.

- Les données échangées avec les services *SendNotification* et *Storage* doivent maintenir leur intégrité.

En fait, différents mécanismes sont fournis par les services pour assurer les propriétés de sécurité. Le service *DoAverage* supporte l'authentification en utilisant une technique de certificat, tandis que le service *SendCommand* le fait en utilisant une paire utilisateur/mot de passe. Pour assurer la confidentialité les services de collection de mesures échangent leurs données en les chiffrant. Finalement, un mécanisme de signature digitale est utilisé par les services métier *SendNotification* et *Storage* afin de prouver que les messages n'ont pas été modifiés pendant la communication.

Dans un premier essai pour ajouter les aspects de sécurité dans l'application, nous l'avons ajouté manuellement. Ainsi, le composant de liaison a été modifié pour mettre en place les propriétés de sécurité. Donc, le code d'invocation des services *DoAverage* et *SendCommand* à été modifié pour inclure la logique d'authentification avant l'invocation. D'autre part, le code d'invocation du service *GetData* a été modifié pour déchiffrer les données provenant de capteurs. Finalement, le code d'invocation des services *SendNotification* et *Storage* a été modifié pour ajouter la signature avant l'invocation et pour vérifier la signature de services après l'invocation.

Pour implémenter la consommation sécurisée des services, nous avons utilisé la plate-forme *Apache Axis* pour la communication avec des services DPWS et services Web en utilisant le protocole SOAP. Cette plate-forme fournit le composant WSS4J qui est chargé d'implémenter la spécification *WS-Security*. Cette spécification propose des extensions du protocole SOAP pour procurer une communication sécurisée entre les services et leurs clients.

L'implémentation manuelle des aspects de sécurité dans l'application n'a pas été appropié, car les changements des besoins de sécurité ou des mécanismes fournis par les services, entraînent des modifications dans le code de bas niveau écrit pour supporter les propriétés. Par conséquent, nous avons étendu le canevas FOCAS pour supporter les aspects de la sécurité dans les orchestrations de services, l'extension sera présentée par la suite.

Extension du canevas FOCAS pour ajouter l'aspect de la sécurité

L'extension faite pour supporter la sécurité suit l'approche d'extension non-fonctionnelle du canevas FOCAS présentée dans les chapitres de proposition et mise en œuvre. Ainsi, d'abord nous avons défini le méta-modèle de sécurité, contenant les concepts pertinents. Ensuite, les relations entre ce méta-modèle et celui de l'orchestration ont été établies. Puis, nous avons étendu l'environnement de spécification CADSE-FOCAS pour permettre la spécification des propriétés sur les orchestrations de services. Finalement, un *template* est ajouté au générateur de code, pour étendre le comportement du composant de liaison afin d'inclure le support des propriétés de sécurité.

L'environnement de spécification CADSE-FOCAS a été étendu pour ajouter le composant de spécification des propriétés de sécurité. Le point d'extension fourni par le canevas a été utilisé, donc nous avons ajouté des onglets dans l'éditeur de contrôle permettent d'exprimer les propriétés de sécurité d'une application. En plus, différents niveaux d'abstraction sont utilisés, d'abord un onglet permet de spécifier les propriétés à respecter pour les activités de l'orchestration, et un autre onglet permet de donner les détails techniques des mécanismes utilisés pour assurer la propriété.

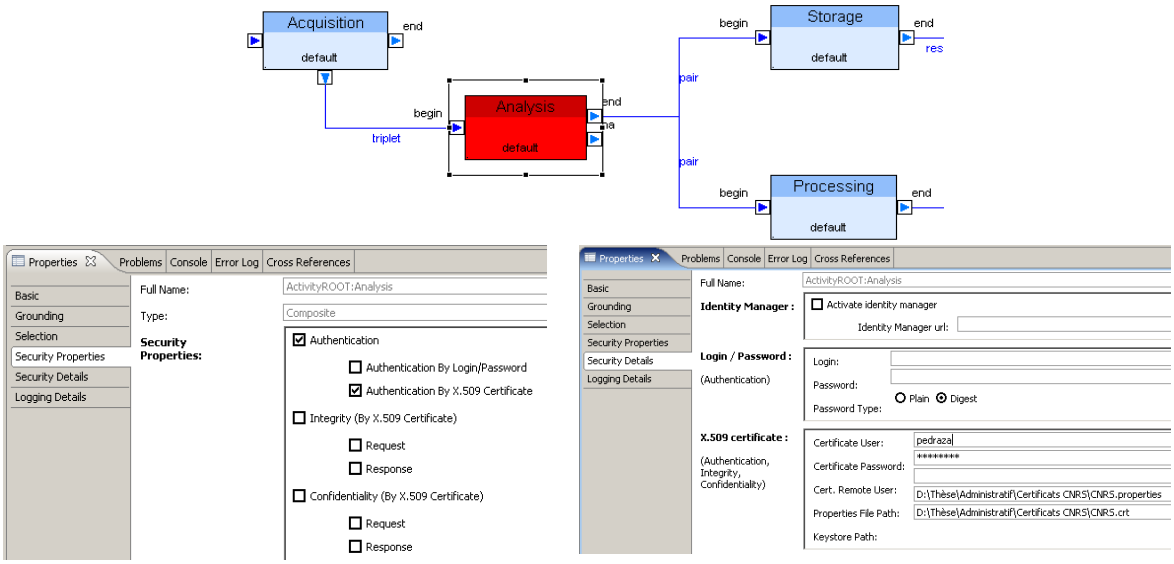

Figure 92. Extensions du CADSE-FOCAS pour l'expression des aspects de sécurité.

Par exemple, l'activité *Analysis* est annotée pour indiquer qu'une authentification doit être utilisée lors de l'invocation de son service associé *DoAverage*, cette information décrit la propriété de l'authentification sans rien dire par rapport au mécanisme à utiliser. Ensuite, des détails comme le nom de l'utilisateur, son mot de passe ainsi que l'emplacement du fichier de certificat sont ajoutées dans les détails techniques. La Figure 92 présente pour l'activité *Analysis*, à gauche l'onglet de propriétés de sécurité, et à droite l'onglet de détails techniques du mécanisme utilisé.

L'autre partie de l'extension consiste en fournir un générateur responsable de produire le code qui assure la propriété de la sécurité. En effet, un générateur du code supportant l'aspect non-fonctionnel a une dépendance vers la plate-forme utilisée pour assurer l'aspect. Ainsi, notre générateur de code de la sécurité a une dépendance vers la plate-forme *Axis* et son composant de sécurité *WSS4J*.

Le *template* utilisé pour générer le code responsable d'assurer les aspects de sécurité est une extension faite sur un *template* utilisé pour générer le composant de liaison. Notamment, l'extension faite consiste à générer un fichier de configuration (WSDD *Web Service Deployment Descriptor*) afin d'indiquer comment le *proxy* (généré par la partie *bottom-up*) doit gérer les propriétés de sécurité. En plus, la classe de liaison chargée d'écouter l'événement associé au changement d'état de l'activité et d'invoquer le service associé, doit être aussi modifié pour indiquer l'utilisation de ce fichier de configuration à la création de l'instance du *proxy*. L'information technique permettant de générer le fichier de configuration est retrouvée dans la spécification de détails techniques des mécanismes de sécurité.

Dans la Figure 93 est présente l'architecture du code généré pour supporter l'aspect de la sécurité dans le canevas FOCAS. A gauche est schématisé l'architecture simple de la première version de l'application, la classe de liaison est responsable d'invoquer le service associé à une activité. Elle écoute l'événement produit par le moteur indiquant que l'activité devient activé, ensuite, elle crée une instance du *proxy Axis* et fait l'invocation du service DPWS en utilisent le *proxy*. A droite dans la figure, est montré l'architecture lorsque les propriétés de sécurité doivent être assurées. Dans ce cas, la classe de liaison utilise l'archive de configuration pour créer l'instance du *proxy Axis*. Ce fichier indique les opérations à réaliser lors de l'invocation du service, ces opérations peuvent être : inclure une entête d'authentification pour le message SOAP, signer le message ou le chiffrer. Ainsi, lorsqu'une

invocation doit être réalisée, le *proxy* lit sa configuration et s'il a besoin, demande au composant WSS4J de réaliser les opérations de modification du message SOAP requises.

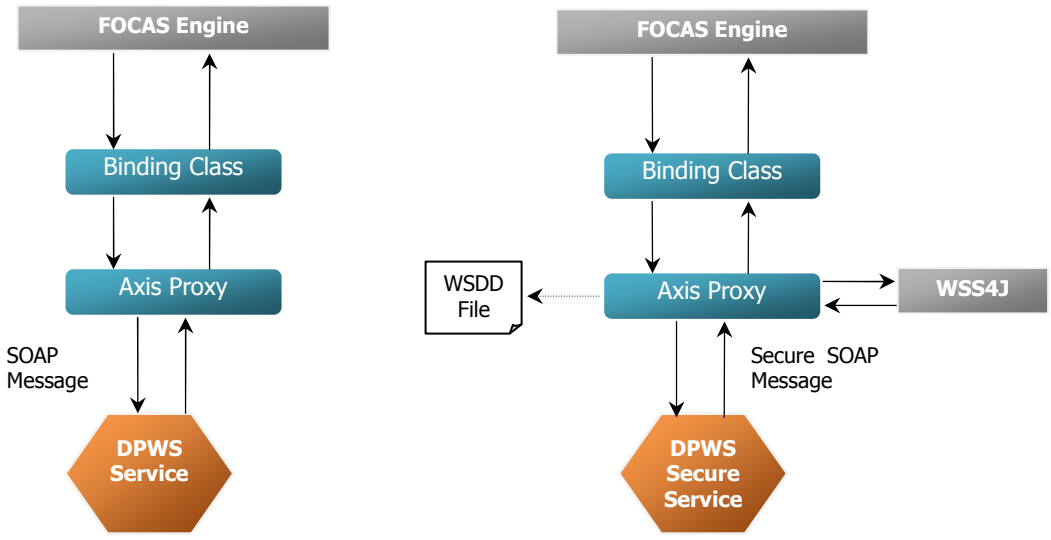

Figure 93. Architecture du code généré pour assurer la propriété de la sécurité.

La mise en place de l'extension de sécurité a permit de valider nos idées autour des extensions non-fonctionnelles du canevas FOCAS. Pour notre cas d'utilisation elle a fourni un mécanisme de haut niveau pour résoudre le problème de l'ajout de propriétés de sécurité à l'application. Il permet l'évolution des besoins (autres ou différentes propriétés de sécurité) ainsi que des évolutions technologiques (utiliser autre plate-forme pour supporter l'aspect).

9.2.4 Métriques de l'application de surveillance d'usine

Du développement de l'application pour le cas d'utilisation proposé nous avons pris un certain nombre de métriques afin d'évaluer la capacité du canevas aider les développeurs. Donc, nous avons mesuré combien de lignes de code sont générées par le canevas dans le cas d'une l'orchestration simple et ensuite lorsque nous lui ajoutons les propriétés de sécurité. Nous avons aussi réalisé une expérience pour déterminer l'impact de l'utilisation du canevas pour supporter l'aspect de la sécurité. Nous allons par la suite présenter ces métriques.

Génération du code

La première métrique sert à mesurer combien lignes de code sont générés par l'environnement CADSE-FOCAS pour implémenter l'orchestration et combien sont écrites par les développeurs. Nous avons classé le code (généré ou écrit) dans quatre groupes, présentés dans le tableau ci-dessous :

Type de code	Description
Logique de l'orchestration	Correspond au code ajoute pour les développeurs pour compléter la logique de l'orchestration. Par exemple, le code correspondant à l'agrégation de mesures dans l'activité *Analysis*, ou le code pour évaluer les actions à prendre dans l'activité *ProcAnalysis*.
Implémentation de données	Correspond au code généré par le canevas pour cacher le format dans lequel le données sont traites par le moteur. Cette génération fournit des interfaces Java pour manipuler ces données.
Composant de liaison	Correspond au code implémentant la logique d'invocation du service. Il est charge d'instancier le proxy et d'invoquer ses opérations.
Médiation de données	Correspond aux classes générées pour implémenter un médiateur des données par défaut. Par contre, le code de médiation doit être rempli par le développeur.

Code de *proxies*	Correspond au code généré par la partie *bottom-up* du canevas pour créer les *proxies* de services existants.

Figure 94. Types de code généré par le canevas pour l'orchestration.

Les mesures sont prises dans quatre scénarios : le deux premiers correspondent à l'implémentation de l'orchestration sans les aspects de sécurité. Dans le premier scénario, une liaison statique est utilisée, tandis que pour le deuxième, il s'agit d'une liaison dynamique. Les autres deux scénarios correspondent à l'orchestration avec les aspects de sécurité, avec la même variation dans la liaison.

Les tableaux ci-dessous présentent le nombre de lignes de code pour chacun des scénarios.

	Orchestration Statique				Orchestration Dynamique		
	Total	Générées	Ecrites		Total	Générées	Ecrites
Logique de l'orchestration	255	125	130		255	125	130
Composant de liaison	541	537	4		611	607	4
Implémentation de données	347	347	0		347	347	0
Médiation de données	154	136	18		154	136	18
Total	**1297**	**1145**	**152**		**1367**	**1215**	**152**

Figure 95. Code généré pour l'orchestration sans les propriétés de sécurité.

	Orchestration Statique				Orchestration Dynamique		
	Total	Générées	Ecrites		Total	Générées	Ecrites
Logique de l'orchestration	255	125	130		255	125	130
Composant de liaison	864	860	4		935	931	4
Implémentation de données	347	347	0		347	347	0
Médiation de données	154	136	18		154	136	18
Total	**1620**	**1468**	**152**		**1691**	**1539**	**152**

Figure 96. Code généré pour l'orchestration avec les propriétés de sécurité.

Du nombre des lignes générées nous pouvons faire certaines réflexions. D'abord, notre approche permet d'inclure une partie de la logique de l'application en tant que code Java, nous avons appelé cette logique modèle de comportement (voir section 7.1.1 du chapitre de mise en œuvre). L'idée d'est éviter de créer des services pour des calculs simples (c'est le cas d'approches comme WS-BPEL) ainsi que de compléter la logique métier. Pour notre cas d'utilisation, 255 lignes de code sont utilisées pour exprimer cette logique, le canevas en a généré 125 et le développeur en a ajouté 130. En fait, le canevas génère les classes correspondant au comportement d'une activité, et le développeur doit compléter les méthodes pour exprimer la logique. En plus, le code généré pour l'implémentation des données vise à faciliter la tâche du développeur lorsqu'il écrit le code de la logique d'application, car des classes Java simples (*Beans*) cachent le traitement des données de la part du moteur d'exécution. Ce code est complètement généré par le canevas et correspond à 347 lignes.

Ensuite, le code de médiation permettant de faire un alignement de types de données abstraits avec les types utilisés par les services. Ce code contient 156 lignes, desquels 18 ont été écrites par le développeur. En fait, normalement le code de médiation comporte une partie générée inferieure à celle écrite, mais dans notre cas d'utilisation l'approche *bottom-up* utilisée fait que les services abstraits ont un pourcentage élevé de compatibilité.

Finalement, le code du composant de liaison correspond à 541 lignes du total du code de l'application. Donc, il fait approximativement le 42% du code de l'application. Ce code est responsable de l'instanciation des *proxies*, l'invocation de médiators avant et après de consommer le service et de l'invocation du service en utilisant le *proxy*. En plus, lorsqu'une liaison retardée est utilisé le nombre de lignes augmente à 611, car le composant de liaison code doit en plus, invoquer le composant de sélection de services.

D'autre par, de l'analyse de lignes de code ajoutées pour supporter les propriétés de sécurité, nous remarquons qu'elles correspondent à 323 lignes, dans le deux cas liaison statique et dynamique. Ces lignes sont ajoutées seulement dans le composant de liaison, les autres types de code restent constants.

Nous ne pouvons pas évaluer le nombre de lignes générées comme une mesure de la qualité du canevas. Cependant, intuitivement nous pouvons dire que le canevas aide de façon significative le travail du développeur. Par exemple, dans la première version de l'application comportant les propriétés de sécurité, nous avons développé le code assurant ces propriétés à la main. C'était une tâche complexe et de bas niveau, avec l'extension faite, le canevas génère tout le code et aucune modification de ce code n'est nécessaire.

Expérience d'ajout de la sécurité

Afin de valider le niveau d'opérabilité de l'extension de sécurité nous avons réalise une expérience. Pour la réalisation de cette expérience, trois scenarii de sécurité ont été définis, chaque scenario comporte des propriétés de sécurité différentes définies pour l'application. Le point de départ de chaque scenario est l'application de surveillance d'usine complètement fonctionnelle, mais sans aucune propriété de sécurité établie.

Un ensemble de développeurs seront chargés de définir les propriétés de sécurité pour l'application. Ces développeurs ne connaissent pas le canevas FOCAS au moment de définir les propriétés de sécurité, et ils ne sont pas non plus des experts dans les technologies de sécurité. Les temps de spécification pour chaque scenario seront mesurés. Puis, les mêmes utilisateurs vont à nouveau réaliser l'expérience, cette fois-ci avec un niveau d'expertise supérieur puisqu'ils l'ont fait déjà une fois.

Les scénarios sont résumés dans le tableau de la Figure 97. Pour chaque paire activité-service une propriété de sécurité (ou aucune) a été définie. Le développeur doit spécifier cette propriété en utilisant l'extension du canevas présentée auparavant (voir section 9.2.3). Nous avons donnée les détails techniques associés à chaque propriété de sécurité dans le document du protocole à suivre pour réaliser l'expérience, une introduction au canevas FOCAS est aussi ajoutée dans ce document afin de préparer ce développeur à son utilisation.

		Scenarii		
Activité	**Service**	**Scenario 1**	**Scenario 2**	**Scenario 3**
Acquisition				
Temperature1	*GetData*	Confidentialité	Confidentialité	Aucune
Temperature2	*GetData*	Aucune	Aucune	Aucune
Humidity	*GetData*	Confidentialité	Confidentialité	Aucune
Analysis	*DoAverage*	Authentification	Authentification	Authentification
Storage	*Store*	Intégrité	Aucune	Intégrité
Processing				
SendNotification	*SendNotification*	Intégrité	Intégrité	Aucune
Action	*SendCommand*	Authentification	Confidentialité	Authentification

Figure 97. Scenarii de sécurité à spécifier.

Les temps utilisés par les développeurs pendant la première session de l'expérience sont présentés dans le tableau de la Figure 98.

	Développeurs								
Scénario	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**D8**	**Moyenne**
1	07:20	17:58	14:39	17:10	11:24	14:45	13:30	09:59	**13:20**
2	05:07	09:42	08:39	09:25	07:37	09:01	07:34	05:32	**07:49**
3	01:55	04:08	04:08	03:00	02:28	04:39	04:38	02:06	**03:22**

Figure 98. Temps résultat de l'expérience niveau débutant.

Les temps utilisés par les développeurs pendant la deuxième session de l'expérience sont présentés dans le tableau de la Figure 99

	Développeurs								
Scénario	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**D8**	**Moyenne**
1	03:40	06:20	05:45	05:14	05:40	05:53	05:43	04:34	**05:21**
2	04:05	06:19	04:53	04:48	07:02	07:13	05:58	04:10	**05:33**
3	01:27	02:51	02:28	02:25	01:53	02:27	02:31	01:18	**02:10**

Figure 99. Temps résultat de l'expérience niveau expert.

D'après les résultats nous pouvons observer que les temps mis pour la deuxième session ont diminué significativement par rapport à la première. Ainsi le temps moyen pour réaliser le premier scénario était de 13 minutes 20 seconds pour la première session, et de 5 minutes 21 secondes dans la deuxième, une diminution d'approximativement 60% du temps employé est obtenue. Pour le scénario 2, la réduction du temps moyen a été de 29% approximativement, tandis que pour le scénario 3 elle a été de 35%.

Afin d'avoir un point de référence permettant d'évaluer les temps utilisé pour mettre en place les scénarii dans FOCAS avec l'extension de sécurité, nous avons mesuré le temps mis par deux experts dans la technologie de sécurité pour implémenter les scénarios manuellement. La démarche a consisté à prendre le code de l'application et à le modifier pour ajouter le support des propriétés de sécurité. Les temps moyens pour les trois scenarios ont été de 23 minutes, 24 minutes et 15 minutes respectivement. Si nous comparons ces temps, avec ceux obtenues par les développeurs pendant la première session, la diminution a été de 44%, 71% et 80% pour les trois scénarios. Ensuite, pour la deuxième session cette diminution a été de 79%, 79% et 86% respectivement. Nous pouvons conclure, que notre extension fourni un support qui réduit significativement le temps de développement des orchestrations sécurisés.

9.2.5 Synthèse

L'utilisation de notre canevas dans des projets européens, nous a permit d'une part de valider notre proposition, mais aussi nous a apporté des idées par rapport aux lignes de recherche à suivre. Etant donné que de vrais besoins sont identifiés, au niveau recherche nous visons à résoudre ce type de problèmes, plutôt que d'essayer de les identifier. La participation conjointe de partenaires industriels et académiques dans ce type de projets fournit un moyen de réaliser une recherche apportant des solutions aux problèmes présents dans le milieu industriel. Pour sa part, l'utilisation des prototypes de recherche dans des contextes industriels permet de valider les approches proposées par les académiques.

Notre canevas a été utilisé dans les deux projets ITEA. Du projet S4ALL nous avons récupérer des besoins qui nous ont guidé dans l'amélioration de notre environnement de spécification CADSE-FOCAS. Du projet SODA, nous avons profité du cas d'utilisation pour mettre en place une orchestration avec des besoins particuliers. Ainsi, pour valider nos mécanismes d'extension, l'aspect de sécurité a été ajouté à notre canevas. Des expériences ont montré la pertinence de la réalisation de l'extension.

9.3 VALIDATION POUR L'ORCHESTRATION REPARTIE

Nous avons proposé une approche pour l'exécution répartie d'une orchestration de services, ce modèle d'exécution a été mis en place en utilisant la technologie d'extension du canevas FOCAS. Nous considérons que la réalisation de cette extension constitue en soi une validation des capacités d'extension du canevas. Néanmoins, nous présentons dans cette section une validation des motivations d'utiliser une architecture répartie pour l'exécution des orchestrations de services.

Nous avons cité trois raisons servant de motivation à l'exécution distribuée. La première consiste en l'élimination du goulot d'étranglement lorsqu'une seule machine est chargée d'exécuter l'orchestration. La deuxième est la diminution du coût de communication en

termes de trafic du réseau et de temps de communication des services distants. La troisième est la possibilité d'obtenir du vrai parallélisme lorsque des chemins concurrents d'une orchestration sont exécutés sur des machines différentes. Dans cette section, nous allons présenter deux expériences, la première permettant de valider l'hypothèse de diminution des couts de communication, autant en termes du trafic de réseau comme de temps associés à l'exécution des services distants. La deuxième expérience, permettra de mesurer le gain de performance associé à un vrai parallélisme de l'exécution des chemins concurrents d'une orchestration.

9.3.1 Expérience réduction de cout de communication

Pour la réalisation de cette expérience nous avons utilisé un modèle d'orchestration simple, il comporte neuf activités s'exécutant en séquence, chacune des activités est responsable d'invoquer un service. Le modèle de contrôle est montré dans la Figure 100.

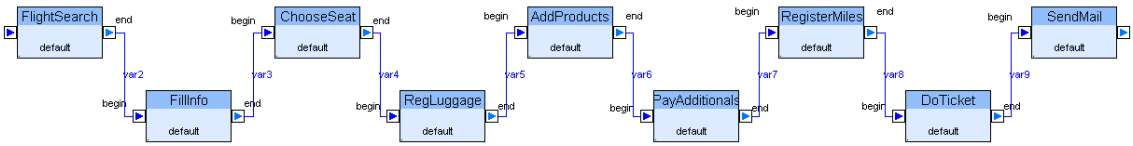

Figure 100. Modèle de contrôle pour l'expérience de réduction de cout de communication.

Afin de faciliter la mise en place de l'expérience un seul type de données a été utilisé dans l'orchestration, ainsi le type *TypeA* est un type complexe ayant huit attributs de type String. Chacune des activités reçoit une variable de *TypeA* en entrée et produit une autre variable du même type en sortie, par exemple, l'activité *FillInfo* reçoit la variable *var2* et produit la variable *var3*, les deux variables sont de type *TypeA*.

Un unique service est utilisé dans l'orchestration, le service *DummyService* qui contient une unique opération, *dummyOperation*. Cette opération a pour responsabilité d'ajouter un fragment de texte à chacun des attributs de la variable de *TypeA* reçue comme paramètre, et ensuite d'afficher un message sur la console pour indiquer que l'opération a été invoquée. Nous avons développé deux implémentations différentes du service, une première implémentation en tant que service Web et une autre pour la plateforme *OSGi*.

Pour la réalisation de l'expérience nous avons configuré deux nœuds, le premier nœud *Node1*, est responsable de l'exécution du premier fragment de l'orchestration, l'autre nœud *Node2*, est responsable de l'exécution du deuxième fragment de l'orchestration ainsi que du service invoqué. Le fragment d'orchestration qui se trouve dans le nœud *Node2* invoque le service de façon locale (invocation de l'implémentation *OSGi* du service), tandis que le fragment du nœud *Node1* invoque le service de façon distante en utilisant le protocole SOAP (implémentation Web du service). La configuration utilisée pour l'expérience est schématisée dans la Figure 101.

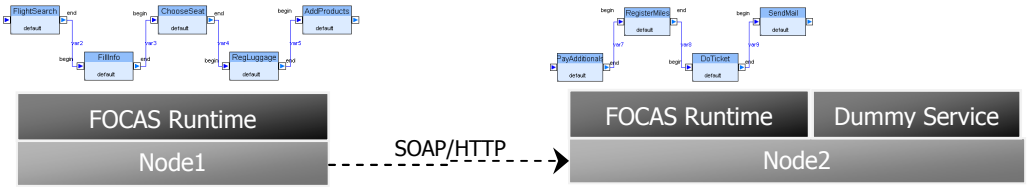

Figure 101. Nœuds pour l'expérience de réduction de cout de communication.

L'expérience consiste alors à exécuter le fragment de l'orchestration dans chaque nœud et à mesurer le temps total de l'exécution. Cinq cas ont été définis, dans chaque cas nous changeons les fragments d'orchestration à exécuter dans les nœuds, l'objectif est d'observer la variation du temps d'exécution dû à la distribution de l'orchestration. Le cas *Case0* exécute

toute l'orchestration dans le nœud *Node1*, le cas *Case1* déplace les deux dernières activités de l'orchestration (*DoTicket*, *SendMail*) vers le nœud Node2. La totalité des cas sont décrits dans le tableau ci-dessous.

Cas	Fragment du nœud Node1	Fragment du nœud Node2
Case0	Toute l'orchestration	
Case1	Contient les 7 premières activités.	Contient les 2 dernières activités.
Case2	Contient les 6 premières activités.	Contient les 3 dernières activités.
Case3	Contient les 5 premières activités.	Contient les 4 dernières activités.
Case4	Contient les 4 premières activités.	Contient les 5 dernières activités.

Figure 102. Cas à exécuter dans l'expérience de réduction de cout de communication.

Protocole de test

Le protocole suivi pour les cinq différents cas de notre expérience est le même, l'expérience est réalisée en deux phases. La phase de préparation où, d'abord, nous allons démarrer dans chaque nœud le *runtime* de FOCAS et dans le nœud *Node2* le serveur chargé d'exécuter le service. Ensuite, dans chaque nœud est créé et démarré l'instance du fragment d'orchestration à exécuter dans ce nœud. Puis, le service est invoqué une fois dans chaque nœud afin d'éviter un possible ralentissement du au chargement et création des instances de classes responsables d'invoquer le service (*proxies*), surtout lorsqu'il est invoqué de façon distante. Une fois la préparation finie, la phase d'exécution consiste en l'envoi d'une donnée en entrée de la première activité de l'orchestration (*FlightSearch*), cet événement provoque alors le déclenchement et exécution de l'orchestration.

Chaque activité invoque le service, soit en local si elle s'exécute sur le nœud *Node2*, soit en distant si elle s'exécute dans le nœud *Node1*. Le temps d'exécution total de l'orchestration correspond au temps qui passe entre le démarrage de la première activité (*FlightSearch)* et la finalisation de la dernière (*SendMail*). Ce temps se calcule comme le temps d'exécution de chaque fragment de l'orchestration plus le temps de communication entre les deux nœuds. L'expérience est répétée 50 fois pour chaque cas, ensuite nous allons éliminer les valeurs aberrantes en utilisant la méthode test Q.

L'expérience a été effectuée sur deux ordinateurs, le premier *Paraguana* disposant d'un processeur à 1,73 GHz de marque Intel (modèle Pentium), et de 2 Gb de mémoire, fonctionnant sur le système d'exploitation Windows XP Professionnel. Le deuxième ordinateur, *Mauresmo* dispose d'un processeur à 1.80 GHz de marque Intel et de 1 Gb de mémoire, *Mauresmo* fonctionne sur le système Windows XP Professionnel. Les deux machines ont Java version 6, *Felix* (plate-forme OSGi) version 1.8.0 et *Tomcat* (serveur web) version 6.0.23.

Afin de constater de possibles différences dues à la configuration choisie, nous avons réalisée l'expérience deux fois, une en utilisant *Paraguana* comme nœud logique *Node1* et *Mauresmo* comme *Node2* et l'autre en utilisant la configuration inverse.

Résultats et analyses

Les résultats de l'expérience sont visibles dans les tableaux ci-dessous, le temps d'exécution est mesuré en millisecondes.

	Paraguana	Mauresmo		
	Node1	Node2	Total	%
Case 0	349,20	0,00	349,20	0%
Case 1	262,27	50,07	318,55	8,78%
Case 2	216,53	66,93	286,87	17,85%
Case 3	161,93	91,47	256,46	26,56%
Case 4	130,20	120,33	256,03	26,68%

	Mauresmo	Paraguana		
	Node1	Node2	Total	%
Case 0	334,53	0,00	334,53	0%
Case 1	245,34	46,32	295,17	11,76%
Case 2	210,54	64,20	281,53	15,84%
Case 3	158,92	91,71	255,10	23,74%
Case 4	124,84	118,32	247,64	25,97%

Figure 103. Tableaux de résultat de l'expérience de réduction de cout de communication.

Le tableau situé à gauche présente les résultats lorsque la machine *Paraguana* à été utilisée comme nœud *Node1* et M*auresmo* comme nœud *Node2*. Le tableau à droite présente les résultats pour la configuration inverse.

Dans chaque tableau, les lignes correspondent à chacun des cas testés, les colonnes *Node1* et *Node2* présentent le temps moyen d'exécution du fragment d'orchestration dans chaque nœud. La colonne *Total* présente le temps moyen d'exécution pour l'orchestration complète, ce qui correspond à l'addition de temps de chaque nœud plus le temps associé à la communication entre les nœuds. La dernière colonne du tableau, présente le pourcentage moyen de temps d'exécution qui est gagné grâce à la distribution de l'orchestration, ce pourcentage est calculé par rapport au cout de l'exécution complètement centralisée dans le nœud *Node1*.

De ces résultats nous pouvons remarquer des détails intéressants. Tout d'abord, pour la configuration *Paraguana-Mauresmo*, nous observons un gain de performance de 8,78% dans le cas *Case1* par rapport à l'exécution centralisé, ensuite pour le cas *Case2* le gain est de 17,85%, lorsque l'on regarde le cas *Case3* le gain est que de 26,56%, mais pour le dernier cas ce gain est de 26,68%. Ces résultats montrent que le fait de distribuer l'orchestration donne une amélioration significative de la performance, par contre cette amélioration reste pratiquement constante pour le dernier cas. Nous expliquons ce phénomène par le fait que l'exécution du service *DummyService* doit être assuré pour la machine *Node2*, ce qui signifie l'exécution additionnel d'un serveur *Tomcat.*

En utilisant la seconde configuration, nous notons un comportement similaire, car le pourcentage d'amélioration augmente dans la même proportion, par contre les temps d'exécution sont plus petits car dans cette configuration, la machine *Mauresmo* utilisée comme Nœud *Node2* est un peu plus performante.

Finalement, nous remarquons que l'utilisation d'une architecture répartie pour l'exécution d'une orchestration séquentielle apporte un gain en performance si des invocations vers des services distants peuvent être évitées, par contre, lorsqu'il existe un déplacement vers un nœud chargé et peu puissant le gain n'est pas significatif.

9.3.2 Expérience exécution parallèle

Pour la réalisation de cette expérience nous avons utilisé un modèle d'orchestration qui comporte une activité au départ (*GenerateData*), ensuite trois branches de cinq activités chacune (*A1-A5, B1-B5, C1-C5*) et finalement une activité de synchronisation *Final*. Le modèle de contrôle est montré dans la Figure 104.

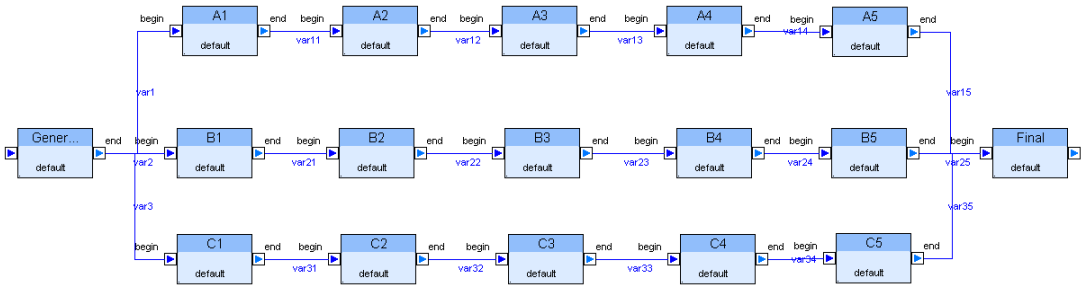

Figure 104. Modèle de contrôle expérience d'exécution parallèle.

Dans cette expérience, nous avons utilisé le même type de données (*TypeA*) et le même service (*DummyService*) de l'expérience précédente. Chacune des activités du modèle d'orchestration doit alors invoquer l'opération du service. Seulement l'implémentation service Web du service sera utilisée dans cette expérience.

L'expérience consiste alors à exécuter un fragment de l'orchestration, c'est-à-dire une branche dans chaque nœud, et de mesurer le temps total d'exécution de l'orchestration. Ainsi,

trois cas on été définis, le premier cas *Case0* exécute toute l'orchestration dans le nœud Node1. Ensuite, le deuxième cas *Case1*, exécute la branche (A1-A5) dans le nœud *Node2* et le reste de l'orchestration dans le nœud *Node1*. Finalement, le troisième cas *Case2* exécute la branche (*A1-A5*) dans le nœud *Node2*, la branche (*B1-B5*) dans le nœud *Node3* et le reste de l'orchestration dans le nœud *Node1*.

Protocole de test

Le protocole suivi pour les trois différents cas de notre expérience est semblable à celui de l'expérience précédente. Donc, une phase de préparation pour instancier les fragments d'orchestration et les *proxies* d'invocation. Une fois la préparation finie, la phase d'exécution consiste en l'envoi d'une donnée en entrée de la première activité de l'orchestration (*GenerateData*), cet événement provoque alors le déclenchement et exécution de l'orchestration.

Le temps d'exécution total de l'orchestration correspond au temps qui passe entre le démarrage de la première activité (*GenerateData)* et la finalisation de la dernière (*Final*). Etant donné que ces activités vont s'exécuter toujours sur le nœud *Node1*, le temps dans ce nœud correspondra au temps total d'exécution de l'orchestration. L'expérience est répétée 50 fois pour chaque cas.

A la configuration de l'expérience précédente (ordinateurs *Paraguana* et *Mauresmo*) nous avons ajouté un troisième ordinateur *Guajira*. *Guajira* comporte un processeur Intel à 3,00 GHz et une mémoire de 1 Gb. *Guajira* utilise le système d'exploitation Windows XP Professionnel.

Résultats et analyses

Les résultats de l'expérience sont visibles dans le tableau ci-dessous, le temps d'exécution est mesuré en millisecondes.

	Node1	Node2	Node3		
	Mauresmo	Paraguana	Guajira	Total	%
Case0	550,81	0	0	550,81	0,00%
Case1	420,73	193,88	0	420,73	23,62%
Case2	265,76	187,14	171,23	265,76	51,75%

Figure 105. Tableau de résultat de l'expérience d'exécution parallèle.

Dans le tableau, les lignes correspondent à chacun des cas testés, les colonnes *Node1, Node2* et *Node3* présentent le temps moyen d'exécution du fragment d'orchestration dans chaque nœud. La colonne *Total* présente le temps moyen d'exécution pour l'orchestration complète, ce qui correspond au temps d'exécution du nœud Node1 car dans ce nœud sont exécutés les activités *GenerateData* et *Final*. La dernière colonne du tableau, présente le pourcentage moyen de temps d'exécution qui est gagné grâce à la distribution de l'orchestration, ce pourcentage est calculé par rapport au coût de l'exécution complètement centralisée dans le nœud *Node1* du cas *Case0*.

En effet, nous observons un gain considérable de la performance lorsque les différentes branches d'exécution ont été distribuées. Tout d'abord, lorsque la branche *A1-A5* a été déléguée au nœud *Node2* le gain a été du 23,62%, ensuite lorsque l'on délègue la branche *B1-B5* au nœud *Node3* le gain est de 51,75%. Nous pouvons alors conclure, qu'une vraie exécution parallèle de l'orchestration apporte une amélioration significative par rapport à son exécution complètement centralisé.

9.3.3 Synthèse

Les deux expériences réalisées ont permis de montrer que l'exécution répartie améliore les performances d'exécution de l'orchestration. L'expérience de réduction de coût de

communication, montre que l'amélioration est significative même lorsque l'on utilise un service simple (*DummyService*). Dans ce cas, le coût d'exécution du workflow est plus important que le coût de communication avec le service, mais si l'on considère des services échangeant des volumes de données plus élevés, l'amélioration de la performance est considérablement plus importante.

D'autre part, la distribution pour l'exécution parallèle d'orchestrations permet d'augmenter la performance de l'orchestration indépendante de l'emplacement des services. Ce type de distribution peut être utilisé dans des applications qui utilisent des calculs parallèles intensifs (par exemple dans les grilles de calcul). En plus, si nous considérons une configuration où, en plus de distribuer les chemins parallèles d'exécution, nous plaçons ces fragments d'orchestration plus proches des services à invoquer, cette configuration cumule les deux avantages étudiés (réduction du coût de communication et parallélisme) ce qui donne un gain de performance plus important.

Ainsi, nous considérons que les hypothèses introduites pour la distribution ont été prouvés par la réalisation de ces expériences, tout en restant prudents, car la détermination d'une adéquate distribution ne fait pas partie de notre étude.

10. CONCLUSION

10.1 SYNTHESE

Dans cette thèse, nous nous sommes principalement intéressés à la création d'applications orientées procédés et aux différents problèmes afférents. L'étude de ce domaine, effectuée dans les chapitres 2 et 4, nous a permis de montrer les similitudes entre la technologie de *workflow* et l'approche d'orchestration de services. Ensuite, nous avons identifié les *applications orientées procédé* comme étant un type d'applications utilisant l'abstraction de procédé comme élément autour duquel se structure sa construction.

Nous avons ensuite montré les exigences de ce type d'applications et quelles sont les contributions faites dans cette thèse pour faire face à ces exigences.

10.1.1 Exigences des applications orientées procédés

Dans la première partie de cette thèse nous avons montré que les approches actuelles pour la spécification et l'exécution d'applications orientées procédés étaient insuffisantes. En fait, il existe une grande quantité de systèmes, canevas, langages et standards proposant diverses solutions dans différents domaines d'application. Nous avons classé ces systèmes en trois grands groupes. D'abord les propositions essayant de fournir le support dans un domaine étroit et bien connu, comme par exemple l'orchestration de services Web. D'autres approches, proposent des systèmes avec un grand nombre de concepts et de fonctionnalités essayant d'aborder plusieurs domaines d'application. Finalement, un nouveau courant récemment apparu essaye de fournir des systèmes minimaux avec des capacités d'extension et d'adaptation.

Les systèmes spécialisés fournissent généralement des solutions performantes, mais avec l'inconvénient d'une porté réduite de réutilisation. Inversement, les solutions riches en concepts et fonctionnalités ont des formalismes complexes et de mauvaises performances. En plus, il existe toujours le risque que les abstractions proposées ne procurent les concepts requis pour un domaine spécifique. Finalement, les systèmes offrant un noyau minimal et des capacités d'extension sont une alternative encourageante, à condition que le noyau proposé et les mécanismes d'extension soient pertinents.

Nous avons alors identifié un ensemble de besoins qui doivent être couverts par un canevas afin de supporter la mise en place effective des applications orientés procédés, ils seront présentés par la suite.

- Formalismes de description : les formalismes de description d'une application orientée procédé doivent être d'un haut niveau d'abstraction. Chaque formalisme doit se concentrer sur un aspect de l'application et non essayer d'exprimer tous les aspects dans un seul formalisme.
- Différents types d'éléments peuvent être coordonnés : La nature des éléments à coordonner peut varier selon le domaine dans lequel l'application est utilisée. En fait, l'abstraction de service est intéressante car elle uniformise la nature des éléments à

coordonner et, même dans ce cas, diverses technologies peuvent être utilisées lors de l'implémentation des services.

- Couplage faible entre le procédé et les éléments qu'il coordonne. Afin de permettre l'évolution, les applications ne doivent pas être fortement couplées aux implémentations des éléments qu'ils cordonnent. La modification de ces éléments ne devrait pas impacter l'application.
- Support pour une liaison dynamique à l'exécution : cette propriété favorise l'évolution de l'application et permet de continuer à exécuter l'application même si les unités de composition sous-jacentes sont en train d'être modifiés. Cette propriété permet aux applications de s'adapter à l'environnement d'exécution et/ou à leur contexte d'exécution.
- Extensibilité et flexibilité. La solution proposée doit pouvoir s'adapter à différents domaines d'application. Elle doit tenir compte des caractéristiques du type d'application et de l'environnement d'exécution afin d'offrir un *runtime* spécialisé. Des mécanismes permettant d'ajouter le support pour différents aspects fonctionnels et non-fonctionnels doivent être fournis par le canevas.

En plus de remplir les exigences énumérées ci-dessus, nous considérons qu'une solution, pour être adoptée, doit être accompagnée d'un ensemble d'outils supportant les travaux des différents acteurs participant à la mise en place des applications.

D'abord, un environnement d'exécution doit être capable de s'adapter aux besoins d'un type d'application particulier. Par exemple, si une orchestration doit être exécutée dans un équipement avec des capacités mémoire réduites, certaines fonctionnalités comme la reprise sur pannes peuvent être éliminées. Par contre, s'il s'agit d'un procédé supportant une activité centrale au sein d'une entreprise, le *runtime* doit comporter des propriétés de reprise sur pannes, d'interopérabilité, et de monitoring.

Tout autant que l'environnement d'exécution, un environnement supportant la spécification de ce type d'applications est nécessaire. Cet environnement de spécification doit aider les différents acteurs impliqués dans la construction d'une application orientée procédé. Il doit offrir aux développeurs des abstractions permettant de réaliser leur travail sans se soucier des « détails » technologiques. De même que le *runtime*, l'environnement de spécification doit être extensible. Les extensions pour l'environnement visent le support des différentes extensions faites sur le noyau du canevas. Ainsi, si une nouvelle préoccupation fonctionnelle ou non-fonctionnelle est ajoutée au type d'application visée, son traitement doit être supporté à la fois par l'environnement de spécification et d'exécution.

10.1.2 Nos contributions

Afin de remplir les exigences définies précédemment, cette thèse propose un canevas pour la construction d'applications orientées procédés appelé FOCAS. Le canevas fourni des composants pour la spécification, l'exécution et le monitoring des applications orientées procédé.

Dans la construction de notre canevas, nous avons utilisé une approche centrée sur les modèles. Des mécanismes utilisés par l'IDM comme la remonté du niveau d'abstraction, la séparation des préoccupations et la réutilisation sont repris par notre canevas. Nous proposons que la spécification d'une application orientée procédé soit faite par la spécification de chacune de ses préoccupations dans des modèles de haut niveau d'abstraction. Ensuite, ces différents modèles seront composés afin de donner une vision globale de l'application.

Chaque modèle est formalisé dans un langage de modélisation exécutable, c'est-à-dire qu'un interpréteur est capable de donner une sémantique d'exécution au modèle doit être disponible. Comme résultat de l'analyse des besoins des applications orientées procédé, notre canevas propose un noyau contenant trois points de vue centraux, à savoir : le contrôle, les

données et les services. Ces trois domaines techniques correspondent à la vision de l'orchestration de services, qui est un type d'applications ayant comme objectif la composition de services par le biais d'un modèle de procédé.

Bien que nous ayons pris l'orchestration comme point de départ de notre canevas, certaines hypothèses faites par les technologies implémentant cette approche ont été relâchées. La plus importante est l'indépendance de notre canevas vis-à-vis de la technologie d'implémentation de services. Des mécanismes de liaison retardée de services ont été implémentés dans la machine sous-jacente supportant l'exécution du domaine de services. Donc, nous pouvons dire que le noyau de notre canevas offre une solution aux cinq premières exigences présentées dans la section précédente.

Cependant, nous considérons que la capacité d'extensibilité et de spécialisation du canevas FOCAS est sa caractéristique la plus importante. Pour cela, des mécanismes permettant d'ajouter des extensions fonctionnelles et non-fonctionnelles ont été proposés. Les extensions fonctionnelles permettent utiliser le canevas dans des autres contextes ayant besoin d'ajouter de concepts au noyau. Par exemple, pour construire un système de *workflow* supportant les tâches réalisés par des humains et la manipulation de documents. Certaines approches étudiées proposent des extensions au niveau des formalismes, mais dans ce cas, le *runtime* doit être implémenté à nouveau pour supporter les nouveaux concepts. Notre canevas, au contraire, permet les extensions tant au niveau de langages que du *runtime*, évitant ainsi de modifier ou de redévelopper le *runtime* existant.

De façon similaire nous avons développé un mécanisme d'extension permettant d'ajouter des propriétés comme la sécurité, le support de transactions et la distribution. Contrairement aux extensions fonctionnelles, ce mécanisme est basé sur une approche générative de code et non sur l'approche de composition de domaines. L'idée reste la même, réutiliser la plate-forme d'exécution et faire des extensions pour supporter les nouveaux aspects.

Bien que le canevas de base propose une solution possédant les propriétés souhaitées et que les mécanismes d'extension fournissent un moyen puissant pour spécialiser le canevas, plusieurs problèmes persistent. En effet, ces mécanismes sont puissants, mais leur utilisation est une tâche lourde et complexe, sujette à erreur. Par conséquent il est impératif de fournir un environnement pouvant prendre en charge la spécification des applications orientées procédés et ayant aussi des propriétés d'extensibilité pour être adaptées en même temps que le *runtime*, pour prendre en charge de nouveaux aspects fonctionnels et non-fonctionnels.

Le CADSE-FOCAS est un environnement de spécification d'applications orientées procédés, il est basé sur la notion de CADSE (*Computer Aided Domain Specific Environment*). Il préconise la construction d'applications en utilisant les concepts centraux des procédés plutôt que les concepts d'un environnement de développement générique. Dans le cas de notre canevas, ces concepts sont ceux de modèle de contrôle, des données et services. La composition des modèles est complètement assistée ainsi que la génération de code pour supporter les aspects non-fonctionnels de l'application. Des mécanismes d'extension permettant l'évolution de l'environnement sont aussi ajoutés.

Finalement, la dernière contribution de cette thèse est la proposition d'une architecture pour l'exécution répartie d'une orchestration. Les avantages d'une telle architecture sont l'amélioration des performances de l'exécution ainsi que la fiabilité du système. Notre *runtime* a été enrichi pour supporter l'exécution distribuée, mais nous n'avons pas modifié les composants de base. Cette propriété fait que la mise en place de l'architecture est simple et que les mécanismes utilisés pour l'exécution centralisée peuvent être repris dans l'architecture repartie.

10.2 PERSPECTIVES

Le canevas proposé dans cette thèse permet la création et l'exécution d'applications orientées procédé. Nous considérons que ce canevas constitue une base de travail qui ouvre un

nombre important de perspectives. Nous avons classé les perspectives ouvertes par cette thèse en quatre catégories.

10.2.1 Les applications orientées procédé flexibles

Autres domaines d'application

Bien que notre mécanisme d'extension fonctionnelle soit puissant, nous considérons qu'il reste des alternatives d'amélioration pour l'utilisation du canevas dans d'autres domaines d'application. Dans notre approche, le méta-modèle composite est formé par l'union de tous les concepts des méta-modèles composés alors que dans certain contextes l'utilisation de tous les concepts est redondante. Une première alternative consiste à explorer la possibilité de modifier le mécanisme de composition de domaines exécutables afin de pouvoir masquer certains concepts qui ne sont pas pertinents dans la composition, ou bien de fusionner des concepts qui se recouvrent.

Un autre scénario imaginable, est celui où notre canevas peut être utilisé pour fournir uniquement le support pour l'exécution des applications. C'est le cas lorsque l'on vise des domaines qui possèdent des formalismes matures et d'utilisation vaste à l'intérieur de leurs communautés. Il faudrait alors, produire les modèles de base utilisés par FOCAS à partir de la spécification dans l'autre formalisme. Une correspondance sémantique est à assurer, car il est possible que les modèles de base ne fournissent pas tous les concepts exprimés dans le formalisme original.

Des applications orientées procédé dynamiques

Une propriété souhaitée dans tout type d'application est la capacité d'évoluer afin de s'adapter à de nouveaux requis, pour améliorer sa qualité, pour s'adapter à un nouvel environnement d'exécution ou à un nouveau contexte d'utilisation. Certaines applications ont besoin de s'adapter pendant leur exécution ; on les appelle des applications dynamiques. Une perspective de nos travaux consiste à mettre en place des applications orientées procédés dynamiques.

Notre approche utilise le formalisme APEL, dans lequel une activité est une boîte noire, des autres activités ne connaissent pas son implémentation, uniquement les données qu'elle attend et que elle fournit. Il est alors possible d'imaginer un cadre d'interaction similaire à celui de l'approche à services, où les implémentations d'une activité peuvent être recherchées et sélectionnées à l'exécution. Cependant, des mécanismes pour la description, recherche et sélection doivent être mise en place pour assurer l'interaction.

La reconfiguration dynamique d'une application orientée procédé peut être réalisée alors à deux niveaux différents ; soit par le changement d'une instance de service soit par le changement d'une instance d'activité. Dans les deux cas, des mécanismes pour assurer cette reconfiguration dynamique doivent être définis, ces mécanismes servent à identifier le besoin d'une reconfiguration (changement de contexte, des services disponibles, etc.), à assurer la cohérence de l'application lors de la reconfiguration et à appliquer la reconfiguration.

10.2.2 Extensions de l'exécution repartie de l'orchestration

La mise en place de l'architecture repartie pour l'exécution fournit des propriétés intéressantes, cependant nous considérons qu'un ensemble de travaux doivent s'achever avant une utilisation effective de cette technique.

Découpage automatique

Nous n'avons pas cherché à définir les critères pouvant être utilisés lors du découpage d'un modèle d'orchestration pour son exécution repartie. Intuitivement, nous avons utilisé comme critère de découpage l'emplacement des fragments d'orchestration de façon à réduire les

coûts de communication. Néanmoins, l'utilisation de ce critère fait l'hypothèse d'une connaissance de l'emplacement des services préalable au déploiement de l'orchestration répartie, ce qui n'est pas toujours le cas, surtout lorsqu'une liaison retardée est utilisée.

Une perspective intéressante est l'identification des critères possibles de découpage d'une orchestration, ainsi que l'automatisation de ces critères. Par exemple, un découpage hiérarchique peut être envisagé, ainsi une machine qui est chargée de l'exécution des activités de premier niveau délègue les activités du niveau suivant à d'autres machines. Ce type de découpage peut être utilisé pour répartir la charge d'une application, et fournir au client de l'orchestration la sensation d'interagir avec une unique machine. En plus, il existe des algorithmes qui font des analyses du contrôle de flux de programme (*slicing*), afin de pouvoir déterminer la meilleure distribution d'un programme. Ce type d'algorithmes basés sur les critères de découpages définis, peut être ajouté à l'environnement de spécification de CADSE-FOCAS.

Vision globale de l'orchestration répartie

Une propriété intéressante de notre approche est la capacité de récupérer l'état des modèles qui sont en train de s'exécuter. Cette caractéristique est due à l'utilisation des interpréteurs qui réifient les concepts des modèles à l'exécution. Cependant, la mise en place de l'architecture distribuée, permet d'avoir seulement une vue partielle de l'état de l'exécution d'une orchestration dans chacun des nœuds. Une perspective de notre travail consiste à construire une vision globale de l'état d'exécution d'une orchestration répartie.

A cet effet, un site peut centraliser la représentation de l'état de l'application globale. Les événements permettant de constituer l'état global de l'application doivent être dirigés vers le site central. Des mécanismes de communication similaires à ceux utilisés pour la communication des instances reparties peuvent être utilisés pour remonter cette information.

10.2.3 Les applications orientées procédé et l'informatique autonomique

Nous avons proposé une approche pour la mise en place d'une exécution repartie d'une orchestration de services. L'introduction d'une architecture répartie amène des problèmes récurrents lorsque l'on construit des applications distribuées. On peut faire des hypothèses simplificatrices comme par exemple que le réseau est fiable, que la latence est nulle, ou que la bande passante est infinie. Cependant, lorsque l'on désire des applications robustes ces problèmes ne peuvent pas être ignorés.

Ainsi, dans l'approche d'exécution repartie des orchestrations, certaines décisions sont prises afin de fournir de mécanismes de base ayant comme objectif de faire face aux défis des applications distribuées. Par exemple, nos tables de routage ont été divisées en un niveau logique et un niveau physique, c'est un mécanisme qui permet de changer l'adresse du nœud physique pour utiliser un nouveau nœud du réseau, si le premier n'est plus disponible.

Cependant, des mécanismes complémentant ceux de base doivent être ajoutées à l'architecture afin d'avoir une architecture robuste d'exécution repartie. Par exemple, un mécanisme permettant d'identifier l'indisponibilité d'un nœud de réseau, ainsi qu'un mécanisme prenant en charge les actions à réaliser lors de la découverte du problème, par exemple en déployant le fragment d'orchestration dans une nouvelle machine et ensuite en remplacent l'adresse physique du nœud logique dans les tables de routage. Nous pouvons encore imaginer des mécanismes de répartition de charge permettant de réorganiser l'architecture d'exécution si un nœud est très chargé à un moment donnée.

En effet, les mécanismes permettant à une application de se récupérer de possibles défaillances, ou même d'optimiser son exécution pour obtenir des performances supérieures sont l'objet d'étude d'un courant émergeant en informatique connue sous le nom d'informatique autonomique (*Autonomic Computing*) [HM08]. L'objectif de l'informatique autonomique est de construire des systèmes autogérés gouvernés par des politiques de haut niveau spécifiées par les

humains [KC03]. Les systèmes autogérés exhibent quatre caractéristiques ; l'auto-configuration, l'auto-réparation, l'auto-optimisation et l'auto-protection.

L'informatique autonomique fait converger plusieurs domaines de recherche, parmi eux nous pouvons citer : la supervision, l'expression de politiques et les plates-formes autonomiques. Les travaux dans la supervision cherchent à fournir des mécanismes capables d'observer un système en exécution. Ensuite, la recherche en expression de politiques fournit des mécanismes pouvant exprimer les objectifs d'un système, ainsi que les actions à prendre pour atteindre ces objectifs. Finalement, les plates-formes autonomiques cherchent à utiliser les techniques du génie logiciel pour la création des applications ayant de propriétés autonomiques [Bou08].

Nous considérons que notre plate-forme d'exécution répartie d'orchestrations peut être un point de départ pour la construction d'un système autonomique dans le domaine de l'orchestration de services. En plus, nous imaginons que les propriétés de notre plate-forme d'exécution répartie, comme la modification possible de l'architecture (en modifiant les tables de routage) et le remplacement possible de fragments d'applications à l'exécution, sont des propriétés de base dans la construction d'une plate-forme autonomique générique et non seulement pour les orchestrations. Cependant, pour atteindre un tel objectif, de nombreux problèmes restent à résoudre, comme la détection et modélisation du contexte d'exécution, l'expression des objectifs, l'exécution des politiques, etc.

En fait, nous pouvons supposer que les applications orientées procédés exhibent des caractéristiques, qui peuvent être utilisées pour l'expression de politiques des applications autonomiques. Par exemple, l'expression des actions à prendre par un composant autonomique peut être faite en utilisant la même classe de formalismes que ceux utilisés par l'approche d'applications orientées procédé. Cependant, il reste à résoudre des problèmes comme la combinaison de l'expression des actions avec l'expression des objectifs, ainsi que l'architecture d'exécution des actions.

10.2.4 CADSEs pour le support du processus logiciel

Nous avons constaté que l'approche de composition de modèles (et méta-modèles) utilisée par notre canevas afin de fournir des mécanismes d'extension est riche, mais sa mise en œuvre est très complexe. Donc, les développeurs sont confrontés à plusieurs défis comme : l'utilisation de divers formalismes, la définition de relations entre les modèles, la génération du code, le changement d'espaces technologiques (représentation dans un langage vers une représentation XML), la manipulation d'un grand nombre de fichiers, le versionnement des artefacts, etc. En définitive, pour réaliser toutes ces tâches, les ingénieurs ont besoin des divers outils que les aident.

La leçon retenue est que l'utilisation d'un environnement supportant l'ensemble des tâches et intégrant les différents outils est cruciale. Un environnement de ce type, connu comme CASE (*Computer-Aided Software Environment*) a deux avantages principaux, d'abord réduire l'effort nécessaire pour la construction d'un produit (programme, modèle, etc.), ensuite améliorer la compréhension qu'ont les ingénieurs du produit et du processus de création. Néanmoins, un tel environnement doit avoir des propriétés comme :

- être spécialisé pour un domaine d'application, nous appelons ces environnements CADSE (*Computer-Aided Software Specific Environment*);
- utiliser des concepts de haut niveau d'abstraction appartenant à l'espace de la solution de la technologie utilisée ;
- intégrer les différents outils qui supportent les différents tâches à réaliser dans l'environnement ;
- être extensible afin de supporter des nouveaux domaines d'application.

Bien que notre environnement CADSE-FOCAS fournisse ces caractéristiques, il cible uniquement la phase du développement de l'application. D'autres étapes du processus du génie logiciel ne sont pas supportés par notre CADSE, notamment l'étape de déploiement qui constitue un manque important au niveau des outils dans notre canevas.

Ainsi, une nouvelle perspective s'ouvre, partant du constat du bénéfice de l'utilisation d'un CADSE, mais aussi du manque du support des autres activités du processus logiciel dans notre canevas. Cette perspective consiste à explorer la possibilité de fournir un support pour tout le cycle de vie du logiciel en utilisent l'enchainement d'un ensemble de CADSEs. Dans cet idées, chaque CADSE est responsable du support des activités d'une étape (ou d'une sous-étape) du processus de construction de l'application. Les CADSEs produisent des artefacts qui sont compréhensibles par les CADSEs qui le suivent dans l'enchainement. Il est ainsi possible d'utiliser l'environnement le mieux adapté à chacune des étapes du développement, tout en respectant un processus global de construction de logiciel.

Cependant, des questions restent encore ouvertes lorsque l'on envisage la construction de cet enchainement de CADSEs, que nous allons associer à la définition d'environnement intégré de support du processus logiciel, IPSE (*Integrated Process Support Environment*). Les problèmes ouverts sont :

- la définition des mécanismes d'intégration des différents CADSEs ;
- la gestion de configuration des artefacts produits par les différents CADSEs ;
- la maintenance des traces permettant de suivre l'évolution des artefacts d'une étape à la suivante ;
- l'assurance du suivi d'un processus logiciel que l'environnement supporte, etc.

Nous considérons que les technologies utilisées dans les travaux de cette thèse, comme CADSEg permettant construire des CADSEs extensibles, ainsi que l'IDM plaçant les modèles comme les artefacts de premier ordre dans la construction d'une application, sont des pistes pour envisager la construction des IPSEs comme un enchainement des CADSEs.

Dans ce travail, nous avons identifié un type d'application fournissant des propriétés intéressantes, les applications orientées procédé. Nous avons aussi proposé des mécanismes et des outils supportant la mise en place de ce type d'applications. Ces mécanismes et outils offrent de propriétés intéressantes en termes d'extension, de réutilisation, et de facilité de mise en œuvre. Nous considérons avoir fait une contribution dans cette voie, pourtant de nombreuses applications et améliorations restent à explorer.

11. BIBLIOGRAPHIE

[ACKM03] G. Alonso, F. Casati, H. Kuno, and H. Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer Verlag, 2003.

[AFM05] M. Aiello, G. Frankova, and D. Malfatti. What's in an agreement? an analysis and an extension of ws-agreement. In *Service-Oriented Computing - ICSOC 2005*, LNCS, pages 424–436, 2005.

[Anh04] Tuyet Le Anh. *Fédération : une architecture logicielle pour la construction d'applications dirigée par les modèles*. PhD thesis, University Joseph Fourier Grenoble, 2004.

[Apa08] Apache Software Foundation. The Apache Tuscany Project, 2008. http://tuscany.apache.org/.

[AtHvdAE07] Michael Adams, Arthur H.M. ter Hofstede, Wil M.P. van der Aalst, and David Edmond. Dynamic, extensible and context-aware exception handling for workflow. In *Proceedings of the 15th International Conference on Cooperative Information Systems (CoopIS 2007)*, 2007.

[Bar92]Naser S. Barghouti. Supporting cooperation in the marvel process-centered sde. In *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 21–31, New York, NY, USA, 1992. ACM.

[BB01] Erwan Breton and Jean Bézivin. Towards an understanding of model executability. In *FOIS '01: Proceedings of the international conference on Formal Ontology in Information Systems*, pages 70–80, New York, NY, USA, 2001. ACM.

[BBB+05] J. Bézivin, M. Blay, M. Bouzeghoub, J. Estublier, and J-M. Favre. Rapport de synthèse: Action specifique cmrs sur l'ingénierie dirigée par les modèles. Technical report, Centre National de Recherche Scientifique CNRS, Disponible sur http://www-adele.imag.fr/mda/as, 2005.

[BC04] E. Baniassad and S. Clarke. Theme: an approach for aspect-oriented analysis and design. *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 158–167, 2004.

[BCR05] Artur Boronat, José Á. Carsí, and Isidro Ramos. Automatic support for traceability in a generic model management framework. In *Model Driven Architecture – Foundations and Applications*, volume 3748/2005 of *Lecture Notes in Computer Science*, pages 316–330. Springer Berlin / Heidelberg, October 2005.

[BCS04] E. Bruneton, T. Coupaye, and J.B. Stefani. The fractal component model. Specification available at http://fractal.objectweb.org/specification/index.html, February 2004. Draf Version.

[BDS05] B. Benatallah, M. Dumas, and Q. Z. Sheng. Facilitating the rapid development and scalable orchestration of composite web services. *Distributed and Parallel Databases*, 17(1):5–37, 2005.

[Ber03] Philip A. Bernstein. Applying model management to classical meta data problems. In *Proceedings of the 2003 CIDR Conference*, 2003.

[BG01] J. Bézivin and O. Gerbé. Towards a precise definition of the omg/mda framework. *Automated Software Engineering, International Conference on*, 0:273–282, 2001.

[BGK+04] M. Blow, Y. Goland, M. Kloppmann, F. Leymann, G. Pfau, D. Roller, and M. Rowley. BPELJ: BPEL for Java. A Joint White Paper by BEA and IBM, March 2004.

[BII+05] BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, and Sybase. Service component architecture. building systems using a service oriented architecture. Available at http://www.iona.com/devcenter/sca/SCA_White_Paper1_09.pdf, November 2005.

[Boo93] G. Booch. *Object-Oriented Analysis and Design with Applications (2nd Edition)*. Object-Oriented Software Engineering. Addison-Wesley Professional, 1993.

[Bou08] J. Bourcier. *Auto-Home : une plate-forme pour la gestion autonomique d'applications pervasives*. PhD thesis, University Joseph Fourier Grenoble, 2008.

[BSD03] B. Benatallah, Q. Z. Sheng, and M. Dumas. The self-serv environment for web services composition. *Internet Computing, IEEE*, 7(1):40–48, 2003.

[BSM+03] F. Budinsky, D. Steingerg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework : A Developer's Guide*. Addison-Wesley, 2003.

[Béz05] Jean Bézivin. On the unification power of models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.

[CCMN04] Girish Chafle, Sunil Chandra, Vijay Mann, and Mangala Gowri Nanda. Decentralized orchestration of composite web services. In *In Proceedings of the Alternate Track on Web Services at the 13th International World Wide Web Conference (WWW 2004)*, pages 134–143. ACM Press, 2004.

[CDNM06] Massimiliano Colombo, Elisabetta Di Nitto, and Marco Mauri. Scene: A service composition execution environment supporting dynamic changes disciplined through rules. In Springer Berlin / Heidelberg, editor, *Service-Oriented Computing – ICSOC 2006*, volume 4294/2006 of *Lecture Notes in Computer Science*, pages 191–202. Springer Berlin / Heidelberg, November 2006.

[CF04] Carine Courbis and Anthony Finkelstein. Towards an aspect weaving bpel engine. In Y. Coady and D. H. Lorenz, editors, *the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Lancaster, United Kingdom, March 2004.

[Cha07] David Chappell. Introducing sca. Avalaible at http://www.davidchappell.com/articles/Introducing_SCA.pdf, July 2007.

[CIJ+00] Fabio Casati, Ski Ilnicki, Lijie Jin, Vasudev Krishnamoorthy, and Ming chien Shan. Adaptive and dynamic service composition in eflow. HPL-2000-39 Technical Report, March 2000.

[Cla02] Siobhan Clarke. Extending standard uml with model composition semantics. *Science of Computer Programming*, 44(1):71–100, July 2002.

[CM04a] A. Charfi and M. Mezini. Hybrid web service composition: business processes meet business rules. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 30–38, New York, November 2004. ACM Press.

[CM04b] Anis Charfi and Mira Mezini. Aspect-oriented web service composition with AO4BPEL. In *European Conference on Web Services*, pages 168–182, 2004.

[CSW08] Tony Clark, Paul Sammut, and James Willans. *Applied Metamodelling A Foundation For Language Driven Development*. Ceteva, second edition, 2008.

[Des07] M. Desertot. *Une architecture adaptable et dynamique pour les serveurs d'applications*. PhD thesis, University Joseph Fourier Grenoble, 2007.

[DLC+07] Xinguo Deng, Ziyu Lin, Weiqing Cheng, Ruliang Xiao, Lina Fang, and Ling Li. Modeling web service choreography and orchestration with colored petri nets. *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, 2:838–843, August 2007.

[EAS08] Wolfgang Emmerich, Mikio Aoyama, and Joe Sventek. The impact of research on the development of middleware technology. *ACM Transactions on Software Engineering and Methodology*, 17(4):1–48, August 2008.

[EDA98] J. Estublier, S. Dami, and M. Amiour. APEL: A graphical yet executable formalism for process modeling. *Automated Software Engineering: An International Journal*, 5(1):61–96, January 1998.

[EDSV09] J. Estublier, I. Dieng, E. Simon, and G. Vega. Flexible composite and automatic component selection for service-based applications. In *In Proceedings of 4th International Conference on Evaluation of Novel Approaches to Software Engineering*, 2009.

[Fab08] Fabric3 Project. The Fabric3 Project, 2008. http://www.fabric3.org/-overview.html.

[Fav04a] Jean-Marie Favre. Foundations of model (driven) (reverse) engineering : Models - episode i, stories of the fidus papyrus and of the solarus. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2004.

[Fav04b] J.M. Favre. Towards a basic theory to model driven engineering. In *3rd Workshop in Software Modeling Engineering*, 2004.

[Fav05] Jean-Marie Favre. Foundations of meta-pyramids: Languages vs. metamodels – episode ii: Story of thotus the baboon1. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[FBV06] Marcos Didonet Del Fabro, Jean Bézivin, and Patrick Valduriez. Weaving models with the eclipse amw plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen, Germany*, 2006.

[FEB06] J-M. Favre, J. Estublier, and M. Blay. *L'ingenénierie dirigér par les modèles*. Edition Hermes, 2006.

[FFR+07] Robert France, Franck Fleurey, Raghu Reddy, Benoit Baudry, and Sudipto Ghosh. Providing support for model composition in metamodels. In *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*, pages 253 – 264, 2007.

[FJ05] Marcos Didonet Del Fabro and Frédéric Jouault. Model transformation and weaving in the amma platform. In *Proceedings of the Generative and Transformational Techniques in Software Engineering (GTTSE'05), Workshop*. Centro de Ciências e Tecnologias de Computação, Departemento de Informatica, Universidade do Minho, 2005.

[FKN94] Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh. *Software Process Modelling and Technology*. Advanced Software Development Series. John Wiley & Sons, September 1994.

[FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. *Future of Software Engineering, 2007. FOSE '07*, pages 37–54, May 2007.

[FRGG04] R. France, I. Ray, G. Georg, and S. Ghosh. Aspect-oriented approach to early design modelling. *Software, IEE Proceedings* -, 151(4):173–185, Aug. 2004.

[FS05] D. F. Ferguson and M. L. Stockton. Service-oriented architecture: Programming model and product architecture. *IBM Systems Journal*, 44(4):753–780, 2005.

[FWK02] Paul Fremantle, Sanjiva Weerawarana, and Rania Khalaf. Enterprise services. *Communications of the ACM*, 45(10):77–82, 2002.

[GHM+05] O. Gruber, B. J. Hargrave, J. McAffer, P. Rapicault, and T. Watson. The Eclipse 3.0 platform: adopting OSGi technology. *IBM Syst. J.*, 44(2):289–299, 2005.

[GHS95] Dimitrios Georgakopoulos, Mark F. Hornick, and Amit P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[Gro01] Object Management Group. Corba component model version 4.0. Specification available at http://www.omg.org/docs/formal/06-04-01.pdf, April 2001.

[Har02] M. Harsu. A survey on domain engineering. Disponible at http://practise2.cs.tut.fi/pub/papers/domeng.pdf, 2002.

[HC01] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together (ACM Press)*. Addison-Wesley Professional, June 2001.

[HHKR+07] C. Herrmann, H. Holger Krahn, B. Rumpe, M. Schindler, and S. Völkel. An algebraic view on the semantics of model composition. In *Model Driven Architecture-Foundations and Applications*, LNCS, pages 99–113, 2007.

[HM08] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv.*, 40(3):1–28, 2008.

[IEE90] IEEE. Ieee standard glossary of software engineering terminology, 1990.

[JCSS01] Li-jie Jin, Fabio Casati, Mehmet Sayal, and Ming-Chien Shan. Load balancing in distributed workflow management system. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 522–530, New York, NY, USA, 2001. ACM.

[JGMB05] J-M. Jezequel, S. Gerard, C. Mraidha, and B. Baudry. Approche unificatrice par les modèles. – rapport final : Action spécifique cnrs sur l'ingénierie dirigée par les modèles,. Available at http://www-adele.imag.fr/mda/as/, 2005.

[JKH+04] Matjaz B. Juric, Bostjan Kezmah, Marjan Hericko, Ivan Rozman, and Ivan Vezocnik. Java rmi, rmi tunneling and web services comparison and performance analysis. *SIGPLAN Not.*, 39(5):58–65, 2004.

[KBA02] I. Kurtev, J. Bézivin, and M. Aksit. Technological spaces: An initial appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, 2002.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP 2001 — Object-Oriented Programming*, LNCS, pages 327–353. Springer-Verlag, 2001.

[KKL+05] M. Kloppmann, D. Koening, F. Leymann, G. Pfau, A. Rickayzen, C. von Riegen, P. Schmidt, and I. Trickovic. WS-BPEL extension for people - BPEL4People. A Joint White Paper by IBM and SAP, July 2005.

[KL03] R. Khalaf and F. Leymann. On web services aggregation. In Springer Berlin / Heidelberg, editor, *Technologies for E-Services*, volume 2819/2003 of *Technologies for E-Services*, pages 1–13. Springer–Verlag, Septembre 2003.

[Kle08] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, 1997.

[KML+04] G. Karsai, M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits. Composition and cloning in modeling and meta-modeling. *Control Systems Technology, IEEE Transactions on*, 12(2):263–278, March 2004.

[KPP06a] Dimitrios Kolovos, Richard Paige, and Fiona Polack. The epsilon object language (eol). *Model Driven Architecture – Foundations and Applications*, pages 128–142, 2006.

[KPP06b] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Merging models with the epsilon merging language (eml). In *Proc. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems*, pages 215–229, 2006.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture(TM): Practice and Promise*. Object Technology Series. Addison-Wesley, April 2003.

[Lee00] M. H. Lee. Model-based reasoning: a principled approach for software engineering. *Software - Concepts & Tools*, 19(4):179–189, 2000.

[Ley01] F. Leymann. Web Service Flow Language (WSFL 1.0). IBM, Specification available at http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, May 2001.

[LR97] Leymann and D. Roller. Workflow-based applications. *IBM Systems Journal*, 36(1):102–123, 1997.

[Lud03] Jochen Ludewig. Models in software engineering – an introduction. *Software and Systems Modeling*, 2(1):5–14, March 2003.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.

[Mic06a] Microsoft Corporation. Devices Profile for Web Services, February 2006. http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf.

[Mic06b] Sun Microsystems. Enterprise JavaBeans Specification Version 3.0. Specification available at http://java.sun.com/products/ejb/docs.html, May 2006.

[MM05] F. Montagut and R. Molva. Enabling pervasive execution of workflows. In *CollaborateCom 2005, 1st IEEE International Conference on Collaborative*

Computing:Networking, Applications and Worksharing, page 10 pp. IEEE Computer Society, 2005.

[MMWF93] Raul Medina-Mora, Harry K. T. Wong, and Pablo Flores. Actionworkflow as the enterprise integration technology. *IEEE Data Eng. Bull.*, 16(2):49–52, 1993.

[MPP02] Massimo Mecella, Francesco Presicce, and Barbara Pernici. Modeling e-service orchestration through petri nets. In A. Buchmann et al., editor, *Proceedings of the International VLDB Workshop on Tehcnologies for E-Services*, pages 38–47, 2002.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–40, September 1992.

[MRB03] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Rondo: a programming platform for generic model management. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 193–204, New York, NY, USA, 2003. ACM.

[New08] Newton Project. The Newton Project, 2008. http://-newton.codecauldron.org/site/index.html.

[Ngu08] T. Nguyen. *Codèle : Une approche de composition de modèles pour la Construction de Systèmes à Grande Échelle*. PhD thesis, University Joseph Fourier Grenoble, 2008.

[NSC+07] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. Matching and merging of statecharts specifications. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 54–64, Washington, DC, USA, 2007. IEEE Computer Society.

[NvLKL07] J. Nitzsche, T. van Lessen, D. Karastoyanova, and F. Leymann. Bpel-light. In *Business Process Management*, volume 4714 of *Lecture Notes in Computer Science*, pages 214–229. Springer, September 2007.

[OAS04] OASIS. Universal description, discovery and integration specification. Specification available at http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm, October 2004.

[OAS06] OASIS. Web services security: Soap message security 1.1. Specification available at http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf, February 2006.

[OAS07a] OASIS. Web services atomic transaction (ws-atomictransaction) version 1.1. Specification available at http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec-errata-os.pdf, July 2007.

[OAS07b] OASIS. Web Services Business Process Execution Language. Specification available at http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf, april 2007.

[Obj08] Object Management Group (OMG). CORBA 3.1, 2008. http://-www.omg.org/spec/CORBA/3.1/.

[OMG03a] The Object Management Group OMG. Common warehouse metamodel. Available at http://www.omg.org/spec/CWM/1.1/, 2003.

[OMG03b] The Object Management Group OMG. Omg mda guide. Available at http://www.omg.org, 2003.

[OMG07] The Object Management Group OMG. Unified modeling language. Available at http://www.omg.org, 2007.

[OMG08] The Object Management Group OMG. Software & systems process engineering metamodel specification. Available at http://www.omg.org/spec/SPEM/2.0/, 2008.

[Ost87] L. Osterweil. Software processes are software too. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 2–13, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[OT01] H. Ossher and P. Tarr. Hyper/j(tm): Multi-dimensional separation of concerns for java(tm). *Software Engineering, International Conference on*, 0:8–21, 2001.

[PA04] Cesare Pautasso and Gustavo Alonso. From web service composition to megaprogramming. In *Proceedings of the 5th VLDB Workshop on Technologies for E-Services (TES-04)*, Toronto, Canada, 29/08/2004 2004.

[PA05] Cesare Pautasso and Gustavo Alonso. The jopera visual composition language. *Journal of Visual Languages and Computing (JVLC)*, 16:119–152, 2005.

[Pap03] M.P. Papazoglou. Service-oriented computing: concepts, characteristics and directions. *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12, Dec. 2003.

[PBM03] Paulo F. Pires, Mário R. F. Benevides, and Marta Mattoso. Building reliable web services compositions. In *In International Workshop Web Services Research, Standardization, and Deployment*, pages 59–72. Springer-Verlag, 2003.

[PHA06] Cesare Pautasso, Thomas Heinis, and Gustavo Alonso. Jopera: Autonomic service orchestration. *IEEE Data Engineering Bulletin*, 29(3), September 2006.

[PHA07] Cesare Pautasso, Thomas Heinis, and Gustavo Alonso. Autonomic resource provisioning for software business processes. *Information and Software Technology*, 49(1):65 – 80, 2007.

[PvdH07] M. Papazoglou and W. van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, July 2007.

[RGF+06] Y.R. Reddy, S. Ghosh, R.B. France, G. Straw, J.M. Bieman, N. McEachen, E. Song, and G. Georg. Directives for composing aspect-oriented design class models. *Transactions on Aspect-Oriented Software Development*, pages 75–105, 2006.

[RM89] A. Rochfeld and J. Moréjon. *La Méthode Merise, tome 3 : Gamme opératoire*. Editions d'Organisation, 1989.

[Ros09] RosettaNet. Rosettanet. Available at http://www.rosettanet.org/, 2009.

[Rum91] J. Rumbaugh. *Object Oriented Modeling and Design*. Prentice Hall, 1991.

[SBDM02] Quan Z. Sheng, Boualem Benatallah, Marlon Dumas, and Eileen Oi-Yan Mak. Self-serv: a platform for rapid composition of web services in a peer-to-peer environment. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 1051–1054. VLDB Endowment, 2002.

[Sei03] E. Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, September 2003.

[SGM02] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional. 2nd Edition, England, 2002.

[Suna] Sun Microsystems. Glassfish: Open Source Application Server. https://-glassfish.dev.java.net/.

[Sunb] Sun Microsystems. Jini.org. http://www.jini.org/wiki/Main_Page.

[SWI09] SWIFT. Swiftnet. Available at http://www.swift.com/, 2009.

[Tay98] David A. Taylor. *Object technology (2nd ed.): a manager's guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[Tha01] S. Thatte. XLANG: web services for business process design. Microsoft, Specification available at http://www.gotdotnet.com/team/xmlwsspecs/xlang-c/default.htm, June 2001.

[Thi98] S. Thibault. *Langagés Dédies: Conception, Implémentation et Application*. PhD thesis, Université de Rennes 1, France, Octobre 1998.

[UPn08] UPnP Forum. UPnP Device Architecture 1.1, October 2008. http://-www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf.

[vdA03] M. ter Hofstede A.H.M. van der Aalst, W.M.P. Dumas. Web service composition languages: old wine in new bottles? In *Euromicro Conference, 2003. Proceedings. 29th*, pages 298–305, 2003.

[vdA04] W. M. P. van der Aalst. Business process management demystified: A tutorial on models, systems and standards for workflow management. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 1–65. Springer-Verlag, Berlin, 2004.

[vdAH05] W. M. P. van der Aalst and Ter A. H. M. Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[vdAtHKB03] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[Veg05] G. Vega. *Développement d'Applications à Grande Echelle par Composition des Méta-Modèles*. PhD thesis, University Joseph Fourier Grenoble, 2005.

[Vil03] J. Villalobos. *Fédération de Composants : une Architecture Logicielle pour la Composition par Coordination*. PhD thesis, University Joseph Fourier Grenoble, 2003.

[W3C02a] W3C. Web Services Choreography Interface (WSCI). Specification available at http://www.w3.org/TR/wsci/, August 2002.

[W3C02b] W3C. Web Services Description Language (WSDL). Specification available at http://www.w3.org/2002/ws/desc/, 2002.

[W3C04] W3C. Web Services Architecture Specification. Specification available at http://www.w3.org/TR/ws-arch/, February 2004.

[W3C05] W3C. Web services choreography description language version. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/, November 2005.

[W3C06] W3C. Web services policy 1.2 - framework. Specification available at http://www.w3.org/Submission/WS-Policy/, 2006.

[WE98] Reisig. Wolfgang and Rozenberg. Grzegorz (Editors). *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. 1998.

[Wes07] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer Verlag, 2007.

[WFM94] WFMC. Workflow management coalition: Workflow reference model. (wfmc-tc-1003). Technical report, Workflow Management Coalition, 1994.

[WFM02] WFMC. Workflow process definition interface – xml process definition language (xpdl) (wfmc-tc-1025). Technical report, Workflow Management Coalition, 2002.

[WG97] G. Wiederhold and M. Genesereth. The conceptual basis for mediation services. *IEEE Expert*, 12(5):38–47, Sep/Oct 1997.

[WPDH02] Petia Wohed, Wil M. P, Aalst Marlon Dumas, and Arthur H. M. Ter Hofstede. Pattern based analysis of bpel4ws. Technical report, Queensland University of Technology, 2002.

[WWWD96] Dirk Wodtke, Jeanine Weisenfels, Gerhard Weikum, and Angelika Kotz Dittrich. The mentor project: Steps toward enterprise-wide workflow management. In *Proceedings of the International Conference in Data Engineering*, pages 556–565, 1996.

[YP04] Jian Yang and Mike P. Papazoglou. Service components for managing the life-cycle of service compositions. *Information Systems*, 29(2):97–125, 2004.