SELECTA : une approche de construction d'applications par composition de services.

Idrissa Abdoulaye Dieng

UNIVERSITE DE GRENOBLE

THESE

Pour obtenir le grade de

DOCTEUR de l'Université de Grenoble

Discipline : INFORMATIQUE

ÉCOLE DOCTORALE Mathématiques Informatique Sciences et Technologies de l'Information

Présentée et soutenue publiquement par

IDRISSA ABDOULAYE DIENG

Le 31 Mai 2010

SELECTA : une approche de construction d'applications par composition de services.

Directeur de thèse :

Jacky ESTUBLIER

JURY :

Jean-Pierre Giraudin, Professeur à l'Université Pierre Mendès France	(Président)
Laurence Duchien, Professeur à l'Université de Lille 1	(Rapporteur)
Jean-Claude Royer, Professeur à l'Ecole de Mines de Nantes	(Rapporteur)
Xavier Blanc, Maître de Conférence à l'Université Pierre et Marie Curie	(Examinateur)
Yves Ledru, Professeur à l'Université Joseph Fourier	(Examinateur)
Jacky Estublier, Directeur de recherche au CNRS	(Directeur de thèse)

Thèse préparée au sein du Laboratoire LIG – Equipe ADELE

Remerciements

Je tiens tout d'abord à remercier tous les membres du jury qui m'ont fait l'honneur de participer à ma soutenance de thèse. Je remercie particulièrement Mme Laurence Duchien et M. Jean-Claude Royer d'avoir accepté de rapporter mon travail de thèse, d'en avoir fait une lecture détaillée, et d'avoir émis des commentaires pertinents. Je remercie également M. Xavier Blanc, M. Jean-Pierre Giraudin et M. Yves Ledru d'avoir accepté d'examiner ce travail.

J'adresse mes sincères remerciements à Jacky Estublier, Philippe Lalanda et Pierre-Yves Cunin qui mon accueilli au sein de l'équipe de recherche ADELE. Je remercie aussi l'ensemble des membres de l'équipe ADELE, particulièrement les membres du groupe de travail CADSE (German, Etienne, Stéphane, Thomas, …), ceux du groupe SAM (Diana, Eric, Mehdi, …) et ceux du Club Pause Pepsi (Bassem, Gabriel, Issac, Johan, Jonathan, Kiev, Lionel, Mariam, Pierre, Pierre-Alain, Walter, ...) pour les bons moments passés ensemble.

Je tiens à adresser ma gratitude à **ma famille** (mes parents, mes sœurs et mon frère) qui m'a toujours soutenu. Finalement, un grand merci à tous mes amis, et particulièrement ceux du Sénégal, de Grenoble ou d'ailleurs en France.

Une mention particulière à : Jacky d'avoir accepté de m'encadrer, qui m'a accordé sa disponibilité et son soutien tout au long de ce travail. **Gabriel** pour m'avoir supporté durant ce travail en tant que collègue de bureau. **German** pour ses conseils, la relecture du manuscrit et son aide lors de la réalisation de l'implémentation de SELECTA. **Stéphane** pour sa disponibilité lors de la réalisation de l'implémentation de SELECTA.

Merci, Thanks, Jarajëf (Wolof, la langue nationale du Sénégal),

On jaaraama ! (Pulaar, ma langue maternelle), Gracias …

Je dédie ce travail à ma chère maman (Hapsatou SY) & à mon père (Abdoulaye DIENG)

Résumé

On appelle composition le mécanisme permettant la réalisation d'applications logicielles par intégration de services. Les applications à service exigeant souvent des propriétés telles que la sélection dynamique ou non déterministe des services, le concept d'application doit être défini différemment, et le processus de composition est très difficile à réaliser manuellement. Le processus de composition devrait être automatisé par des outils et des environnements logiciels. Ces outils devraient permettre d'une part la construction d'applications flexibles et d'autre part garantir la cohérence et la complétude de la composition.

Cette thèse s'intéresse à la construction d'applications depuis leur conception jusqu'à leur exécution. Dans cette thèse, nous proposons une approche de composition de services et un prototype nommé SELECTA permettant de répondre aux besoins et aux défis de la composition de services, tels que l'augmentation du niveau d'abstraction de la spécification de la composition, la construction incrémentale de composites, la gestion de la cohérence et de la complétude de composites, la définition de langages de contraintes de services ou d'algorithmes de sélection de services.

Mots clés : Composition de services, Construction incrémentale, Sélection de services, Cohérence de la composition, Modèle d'application, Contraintes de services, Environnements logiciels, Ingénierie dirigée par les modèles.

Abstract

We call composition the mechanism for building software applications by service integration. Service-based applications often require properties like dynamic or non-deterministic service selection that require a new definition of software application, which turns out to be very difficult to perform manually. There is a need to automate this process by providing tools and environments. These tools should allow to build flexible software applications and to ensure the consistency and completeness properties of the composition.

This thesis focuses on building software applications, from design to execution. In this thesis, we propose a service composition approach and a prototype named SELECTA that meets the needs and challenges of service composition, such as increasing the abstraction level for application specification, incremental building of composites, consistency management and completeness of composites, definition of service constraints languages or service selection algorithms.

Keywords: Service composition, Incremental building, Service selection, Consistency of the composition, Application model, Service constraints, Software environments, Model-driven engineering.

Table des Matières

Table des Figures

Liste des Tableaux.

1 INTRODUCTION.

1.1 CONTEXTE ET PROBLEMATIQUE.

Le génie logiciel est une discipline informatique, née dans les années 70, qui cherche à faire du développement et de la maintenance des systèmes logiciels un processus d'ingénierie. Cette discipline couvre plusieurs activités et domaines métiers, et fournit des méthodes, des outils et des ateliers de support pour la construction de logiciels, afin d'optimiser les coûts et les délais de réalisation de ces logiciels et d'améliorer la qualité de leur production.

Le génie logiciel doit cependant faire face à un ensemble de besoins et défis majeurs afin d'appréhender un certain nombre de problématiques auxquelles sont confrontées les entreprises fournissant des logiciels en général, et les concepteurs et/ou développeurs de systèmes informatiques en particulier. Ces problématiques concernent par exemple :

- l'amélioration de la qualité des logiciels,
- la maîtrise de la croissante complexité des systèmes logiciels,
- la prise en charge de la diversité et de l'hétérogénéité des systèmes logiciels, en termes de langages de programmation, de technologies de conception (et de réalisation) ou de plates-formes d'exécution,
- la gestion de l'évolution de ces systèmes informatiques, avant et durant leur étape d'exécution.

Pour répondre à ces problématiques, de nombreux travaux de recherche ont été menés dans le domaine de l'ingénierie des logiciels. Ces travaux ont souvent aboutis à de nouveaux paradigmes tels que les approches orientés objet [Tay98], composant [HC01] ou service [PvdH07], préconisant la construction de systèmes logiciels par réutilisation et intégration de briques logicielles, ou bien l'ingénierie dirigée par les modèles [BBB+05] qui fournit des principes et des mécanismes permettant d'augmenter le niveau d'abstraction de la modélisation des logiciels pour faire face à leur croissante complexité.

Les systèmes logiciels ont beaucoup évolué au cours de ces dernières années. En effet, le processus de développement de ces systèmes et leur structure étaient monolithiques et centralisés en prenant des hypothèses très fortes : les besoins de l'entreprise sont stables et la structure des systèmes logiciels est statique. Actuellement, les systèmes deviennent particulièrement distribués et peuvent être dynamiquement fédérés à partir d'un ensemble d'entités logicielles qui sont fournies par l'entreprise. Les systèmes ont besoin d'être plus flexibles, dynamiques et facilement adaptables afin de répondre aux nouveaux besoins et exigences de l'entreprise.

La technologie des services, comme son prédécesseur celle des composants logiciels, se base sur une séparation claire entre l'interface (décrivant les fonctionnalités fournies) et sa mise en œuvre. Cependant, le paradigme à service met l'accent sur le fait que les services pouvant être utilisés dans une application :

- peuvent être disponibles (localement ou à distance) et peuvent être fournis par des tiers,
- peuvent être disponibles dynamiquement c'est-à-dire les services peuvent apparaître ou disparaître à tout instant du cycle de vie de l'application,
- peuvent être sélectionnés tardivement ; leur sélection peut être réalisée à tout moment du cycle de vie de l'application, y compris à l'exécution,
- divers services fournissant les « mêmes » fonctionnalités peuvent simultanément être utilisés dans la même application et peuvent être partagés par différentes applications.

La technologie à service augmente ainsi la **flexibilité** dans le choix des services logiciels requis par une application et améliore aussi leur **réutilisation** et leur **dynamisme**. Toutes ces caractéristiques font que cette approche est particulièrement bien adaptée à de nouveaux domaines d'applications comme la téléphonie ou la domotique (qui utilise un ensemble d'équipements ou d'appareils électroniques). La technologie à service se généralise, et il existe actuellement un nombre important d'applications, se basant sur cette technologie, en cours de développement. Malgré ce succès, la réalisation d'applications à base de services, est aujourd'hui une tâche assez difficile, parce que ces applications ont souvent des caractéristiques et des contraintes inhabituelles, ce qui nécessiterait d'adapter ou de « repenser » les méthodes et/ou les outils nécessaires pour assister leur réalisation jusqu'à leur exécution.

Le mécanisme permettant de réaliser l'intégration de services logiciels est appelé une **composition**, et le résultat est en général un nouveau service appelé un **service composite**. Le paradigme à service n'indique pas comment utiliser divers services, par intégration, pour effectuer le processus de la composition. C'est pourquoi, plusieurs approches, langages et outils ont été proposés, tant dans le monde académique qu'industriel, afin de réaliser la composition de services. Dans cette thèse, l'étude réalisée concernant les approches et outils existants pour la composition de services a révélé de nombreuses lacunes, et nous avons identifié un ensemble de besoins et de défis de la composition de services. En effet, la réalisation d'une composition est actuellement une tâche manuelle pouvant être assez complexe parce qu'elle se fait généralement avec un bas niveau d'abstraction.

Les propriétés essentielles lors du processus de la composition de services, telles que la gestion de la **cohérence** et de la **complétude** de la composition ou la prise en compte des aspects non fonctionnels ou la gestion de la **disponibilité dynamique** des services qui participent à la composition, ne sont souvent pas assurées par les approches et outils actuels

de composition. Il faudrait cependant **automatiser** le processus de construction d'un service composite pour simplifier et faciliter sa réalisation et son exécution. Ainsi, nous ressentons le besoin de réduire la complexité du processus de composition en fournissant des langages, modèles et outils ayant un haut niveau d'abstraction. Les concepteurs (et/ou les développeurs) d'applications requièrent des concepts avec un haut niveau d'abstraction, des mécanismes, des outils et des environnements logiciels qui supportent les propriétés requises par les applications à service.

1.2 OBJECTIFS DE LA THESE.

Notre principal objectif est de faciliter la réalisation d'applications à base de services logiciels. Dans la pratique, une application est souvent modélisée sous la forme d'un composite. Dans ce travail, nous avons comme objectif de proposer une approche définissant, avec un niveau d'abstraction assez élevé, le concept de composite. Notre intention est de fournir un ensemble d'outils et d'environnements logiciels spécialisés en fonction des activités associées à chaque phase du cycle de vie de l'application depuis le développement « rapide » d'applications à service jusqu'à leur exécution.

Les propriétés des services font que les applications à service possèdent de nombreuses caractéristiques particulières : les services utilisés par l'application sont dynamiques dans le sens où ils peuvent apparaître ou disparaître à tout instant, ils peuvent être partagés dans divers contextes par plusieurs applications et ils peuvent aussi être distribués à travers un réseau, par exemple Internet ou Intranet.

C'est la raison pour laquelle, nous souhaitons offrir la capacité d'exprimer tous ces aspects au niveau de la spécification de composites afin de répondre aux besoins des applications à service, en fournissant des concepts, des techniques et des mécanismes, des outils et des environnements logiciels permettant la conception, la réalisation et l'exécution d'applications à service qui requièrent un niveau élevé de flexibilité et de dynamisme. Le travail consiste également à fournir une plateforme d'exécution qui est chargée d'assurer que les propriétés et caractéristiques des composites sont effectivement satisfaites à l'exécution, et les environnements logiciels permettant de décrire et valider les descriptions de composites.

En résumé, les objectifs de ce travail sont les suivants :

Nos objectifs.

- *Faciliter la réalisation et l'exécution d'applications à service.*
- *Définir le concept de composite prenant en compte les besoins identifiés pour la composition comme :*
  - *Augmenter le niveau d'abstraction de la spécification de composites.*
  - *Définir des composites par intention, à partir des propriétés et contraintes à satisfaire.*
  - *Définir des composites automatiques et flexibles.*
- *Automatiser le processus de réalisation de la composition :*
  - *Définir un langage d'expression de contraintes et de sélection des éléments requis par un composite.*
  - *Fournir des mécanismes, des outils et un algorithme de sélection de services.*
  - *Gérer et garantir la cohérence et la complétude des composites.*
- *Fournir des environnements logiciels spécialisés dans les tâches/activités de génie logiciel qui sont liées à chaque phase du cycle de vie des applications ; et en particulier, des environnements de développement et de composition de services, et ainsi que les plates-formes d'exécution associées.*

1.3 ORGANISATION DU MANUSCRIT.

Outre l'introduction, ce rapport de thèse est composé de deux parties principales : la première intitulée « Etat de l'art et de la pratique », introduit le contexte dans lequel se place cette thèse et dresse un état de l'art de quelques langages, approches et outils qui sont liés au contexte scientifique de cette thèse. La seconde partie du document, intitulée « Contributions », présente notre approche de composition de services et sa mise en œuvre à travers le canevas SELECTA.

PREMIERE PARTIE : « ETAT DE L'ART ET DE LA PRATIQUE »

Cette partie est organisée en deux chapitres :

- Le chapitre 2 « **Le contexte : Préliminaires** » introduit les approches, les langages et les modèles qui sont en relation avec le contexte scientifique dans lequel se déroule cette thèse.

- Le chapitre 3 « **Composition de services** » décrit en détail le processus de la composition de services, les catégories de composition de services qui existent, les avantages et limitations des approches et des environnements actuels de composition de services.

Nous concluons la première partie du manuscrit de cette thèse, en présentant les besoins et les défis de la composition de services.

DEUXIEME PARTIE : « CONTRIBUTIONS »

Cette partie est organisée en trois chapitres :

- Le chapitre 4 **« SELECTA : une approche de composition »** met en évidence notre proposition pour répondre aux besoins et aux défis de la composition. Dans ce chapitre, nous présentons nos besoins et nos objectifs, nos principaux concepts et mécanismes et le support fourni pour automatiser le processus de la composition.

- Le chapitre 5 **« SELECTA dans le contexte de SAM »** décrit notre approche de composition au sein de la machine à services SAM. Dans ce chapitre, nous présentons d'abord l'architecture et les concepts de la machine SAM. Nous détaillons ensuite le concept de composite pour faciliter la réalisation et l'exécution d'applications à service. Nous introduisons également les apports de notre approche par rapport à l'état de l'art.

- Le chapitre 6 **« Réalisation et Expérimentations »** présente la mise en œuvre de notre approche en décrivant notre environnement de composition de services, nommé SELECTA, et sa plateforme d'exécution associée SELECTA RUNTIME qui se charge d'exécuter et de gérer les services composites. Nous introduisons également les expérimentations réalisées et la validation de notre approche.

Le chapitre 7 **« Conclusion et Perspectives »** expose les contributions de cette thèse et introduit les perspectives du travail.

Première partie :
État de l'art et de la pratique.

2

LE CONTEXTE : PRELIMINAIRES.

Ce chapitre a pour objectif de présenter plusieurs approches et modèles qui sont en relation avec le contexte scientifique dans lequel se déroule cette thèse, le chapitre 3 abordera le sujet central de cette thèse : la composition de services.

Dans ce chapitre, nous présentons d'abord l'approche à composant en identifiant ses concepts et principes de base. Notre objectif n'est pas de fournir un état de l'art détaillé sur l'approche à composant, mais plutôt d'introduire ses principaux concepts, ses avantages et limitations qui nous permettrons d'aborder les autres approches présentées dans ce chapitre. Nous décrivons ensuite les principes fondamentaux de l'approche à service apparue pour faire face à certaines limitations de l'approche à composant. En même temps, nous introduisons différentes technologies ou plates-formes à services mettant en œuvre le paradigme orienté service.

Nous présentons par la suite le concept de composant orienté service dont l'objectif est de réaliser l'approche à service à travers le paradigme composant pour palier les limites de l'approche à service, tout en bénéficiant des avantages offerts par l'approche orientée composant. Finalement, nous introduisons la notion d'architecture logicielle pouvant être utilisée comme concept central lors de la conception de systèmes logiciels suivant les approches présentées précédemment. Avant de conclure ce chapitre, nous introduisons les concepts et les propriétés de base de l'ingénierie dirigée par les modèles.

2.1 APPROCHE ORIENTEE COMPOSANT.

La réutilisation a toujours été un défi majeur depuis le début du génie logiciel. C'est pourquoi, de nombreuses approches [Tay98][Boo93][SGM02][HC01] ont été proposées pour apporter des solutions à cette problématique de réutilisation. Ainsi, l'approche à composant a été définie pour améliorer la réutilisation du code des applications logicielles. Pour ce faire, cette approche promeut la construction d'une application à partir d'un ensemble de briques logicielles bien définies et indépendantes, appelées composants [SGM02][HC01].

2.1.1 Les principes et caractéristiques de base.

L'objectif de l'approche à base de composant est d'améliorer les limitations de son prédécesseur en l'occurrence l'approche orientée objet [Tay98][Boo93]. Ainsi, elle cherche à améliorer la réutilisation grâce au développement d'applications par assemblage de composants, à fournir des mécanismes permettant aux développeurs de se concentrer uniquement sur les besoins métiers de l'application, à faciliter la maintenance, l'évolution et l'administration des applications logicielles.

En effet, l'approche à composant propose d'encapsuler les données et les fonctionnalités de base d'une application à l'intérieur d'un ensemble d'éléments appelés des composants. Les fonctionnalités fournies par un composant sont uniquement visibles et accessibles à travers ses interfaces publiques. Une séparation claire est effectuée entre ces interfaces et le code réalisant leur implémentation. De ce fait, lors de l'assemblage des composants constituant une application, des connexions seront établies entre ses composants en termes d'interfaces pour spécifier l'architecture de l'application. L'acteur (c'est-à-dire l'assembleur) réalisant cet assemblage ne connaît pas la structure interne des composants qu'il utilise du fait que selon son point de vue ces derniers sont des « boites noires ».

La séparation des préoccupations entre les aspects fonctionnels et non fonctionnels tels que la sécurité, les transactions ou la distribution d'un composant fait partie des principes de base de l'approche orientée composant. Le développeur d'un composant réalise seulement le code correspondant à la logique applicative liée aux fonctionnalités offertes par le composant, et les propriétés non fonctionnelles sont gérées et assurées par la plateforme d'exécution associée. Plusieurs plates-formes à composant proposent des mécanismes pour gérer un ensemble de propriétés non fonctionnelles. Généralement, le concept de conteneur est utilisé pour réaliser la gestion de ces aspects non fonctionnels. L'ensemble des propriétés non fonctionnelles est extensible dans la mesure où le développeur peut en définir d'autres en écrivant le code associé.

A l'heure actuelle, il n'existe pas encore de véritable consensus sur la définition du concept de composant mais les définitions suivantes sont les plus citées et acceptées dans la littérature :

Clements SZYPERSKI [2002]

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

Dans la définition proposée par SZYPERSKI [SGM02], nous pouvons identifier les caractéristiques suivantes d'un composant :

- Un composant est une unité de composition. Il peut donc être assemblé dans une application par un tiers.
- Un composant possède des interfaces bien définies permettant de spécifier les fonctionnalités fournies par le composant.
- Un composant peut exprimer des dépendances explicites vers d'autres composants en vue de spécifier ses besoins.
- Un composant peut être une unité de déploiement et il peut être déployé de manière indépendante.

George HEINEMANN [2001]

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

D'après cette définition [HC01], un composant devrait être conforme à un modèle à composant. De nos jours, il existe plusieurs modèles à composant proposés tant dans le monde académique qu'industriel. Comme exemple, nous pouvons citer les modèles à composant CORBA (CCM), EJB et le canevas .NET issus de l'industrie ou le modèle Fractal issu de la recherche. Chaque modèle à composant possède sa propre infrastructure d'exécution capable d'exécuter des composants qui lui sont conformes.

Dans la définition proposée par HEINEMANN [HC01], nous retrouvons les propriétés suivantes :

- Un composant peut être une unité de déploiement pouvant être déployée indépendamment des autres composants.
- Un composant peut être composé dans une application avec d'autres composants, ceci sans aucune modification préalable du composant. Le mécanisme de composition est défini par le modèle à composant auquel est conforme le composant.

En résumé, à partir des définitions citées précédemment, nous pouvons noter qu'un composant est une unité logicielle qui est bien spécifiée, autonome, réutilisable dans plusieurs applications ou contextes, et qui peut être déployée de manière indépendante. Selon cette définition, pour qu'un composant puisse être déployé indépendamment, il faudrait que sa définition soit bien séparée de sa plateforme d'exécution et des autres composants avec lesquels il pourrait être assemblé [SGM02]. En outre, un composant devrait être clairement spécifié (en exprimant ce qu'il fournit et ce qu'il requiert) et il devrait être auto-contenu (*self contained* en anglais) dans le but de faciliter son assemblage avec d'autres composants. En d'autres termes, un composant a besoin d'encapsuler son implémentation et il doit interagir avec sa plateforme d'exécution uniquement à travers des interfaces bien définies qui constituent ses points d'accès.

La figure suivante illustre le concept de composant :

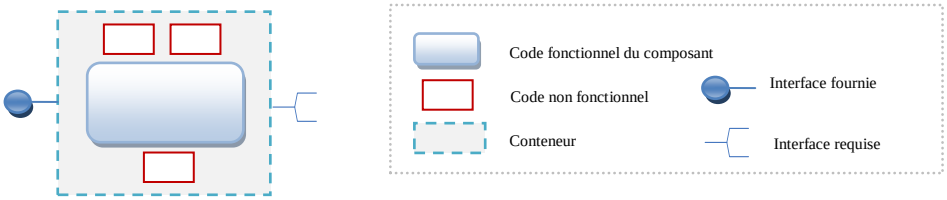

figure 1. Exemple d'une instance de composant.

2.1.2 Les avantages de l'approche à composant.

L'approche orientée composant possède plusieurs avantages dans la mesure où :

- Elle favorise la réutilisation de code grâce au développement d'applications par assemblage de briques logicielles préexistantes.
- Elle facilite le développement incrémental d'applications parce que certains composants d'une application donnée peuvent être développés puis intégrés plus tard.
- Elle effectue une séparation claire entre les aspects fonctionnels et non fonctionnels d'une application en fournissant des mécanismes permettant aux développeurs de se concentrer uniquement sur les besoins de l'application, les aspects non fonctionnels associés au composant étant gérés par la plateforme d'exécution du composant (en général, par un conteneur).
- Elle améliore l'extensibilité, l'évolution et la maintenance des applications puisque ces dernières ne sont plus monolithiques.
- Elle effectue une séparation claire entre les tâches des développeurs et des assembleurs.

2.1.3 Synthèse.

Dans l'intention de réduire les délais et les coûts de réalisation des logiciels, l'approche à composant propose d'assembler des composants préexistants et réutilisables. Malgré les avantages de cette approche, il existe toujours plusieurs obstacles à surmonter dans le développement d'applications avec le paradigme composant.

Un composant est une brique logicielle qui est composée d'une ou plusieurs interfaces et le code réalisant les fonctionnalités qui sont fournies au travers de ces interfaces. L'interface constitue le seul point d'accès pour utiliser un composant ; donc elle devrait fournir en plus des fonctionnalités offertes par ce composant toute l'information sur ses dépendances c'est-à-dire les services requis pour la réalisation de ses fonctionnalités. Toutefois, dans les modèles à composant existant les dépendances ne sont pas explicitement spécifiées. Ce qui constitue une limitation de cette approche lors de l'assemblage des composants d'une application.

Un composant est réalisé en vue d'être assemblé dans une application avec d'autres composants. Le mécanisme d'assemblage est souvent appelé une composition. D'ailleurs un langage de composition est nécessaire pour la réalisation d'une composition mais la plupart des modèles à composant comme EJB[1] [Mic06b] ou CCM[2] [Obj01] ne disposent pas de langages de composition.

Dans l'approche orientée composant, la structure de l'application est en général définie statiquement lors de la conception. Ainsi, il s'avère très difficile de substituer un composant par un autre à l'exécution. Une limitation majeure de cette approche est le fait qu'elle ne fournit pas de support pour la gestion du dynamisme des composants utilisés (c'est-à-dire le fait d'avoir des composants pouvant être disponibles ou pas).

[1] Enterprise Java Beans

[2] Corba Component Model

Le tableau suivant résume les avantages et les limites de l'approche à composant :

Avantages	Limites
Favorise la réutilisation.	Couplage fort (dépendances entre composants).
Exprime explicitement ses fonctionnalités.	Elle ne fournit pas de support pour la gestion du dynamisme. Architecture statique.
Séparation des aspects fonctionnels & non fonctionnels.	La plupart des modèles à composant ne disposent pas de langage de composition.
Facilite le développement incrémental d'applications.	
Gère le cycle de vie des instances de composant.	

tableau 1. Récapitulatif des avantages et limites de l'approche à composant.

2.2 APPROCHE ORIENTEE SERVICE.

La section précédente a mis en évidence les limitations de l'approche à composant concernant la gestion du dynamisme des composants d'une application à l'exécution et le fort couplage entre ses composants. Pour faire face à ces limitations, l'approche orientée service, est apparue récemment en génie logiciel dans l'objectif de faciliter la réalisation d'applications logicielles.

En effet, ce paradigme à service a été conçu à l'origine pour apporter une solution à la problématique d'intégration de systèmes informatiques hétérogènes, par exemple pouvant être réalisés avec différents langages de programmation. Ainsi, l'approche à service cherche à faciliter la réalisation de l'intégration de ces systèmes. En outre, elle propose des moyens pour supporter le développement d'applications flexibles et faciliter l'évolution indépendante des éléments constituant l'application. Dans cette approche, les éléments d'une application peuvent apparaître ou disparaître à tout instant, et ils peuvent être remplacés, même à l'exécution, par d'autres éléments qui sont disponibles.

Dans cette section, nous allons introduire d'abord les concepts de base de l'approche à service. Ensuite nous identifions les avantages de cette approche. Puis avant de conclure par une synthèse, nous présentons la plateforme à service OSGi[1] [OSG05a] qui met en œuvre l'approche orientée service.

2.2.1 Les concepts de base.

La notion de service constitue le concept de base de l'approche à service. Cette approche promeut la construction de systèmes informatiques par intégration de services logiciels. Un service logiciel correspond à une entité logicielle fournissant un ensemble de fonctionnalités et qui est mise à disposition par divers fournisseurs en vue d'être utilisée par un tiers (un autre service ou une application) considéré comme le client du service. Dans la littérature, nous trouvons un ensemble de définitions du concept de service et parmi elles :

[1] Open Services Gateway initiative

M. P. PAPAZOGLOU [2003]

"Services are self-describing, platform-agnostic computational elements…."

M. P. PAPAZOGLOU [2007]

"Services are autonomous, platform-independent entities that can be described, published, discovery, and loosely coupled in novel way."

Dans ces définitions, PAPAZOGLOU [Pap03][PvdH07] présente un service comme une entité logicielle possédant une description clairement spécifiée et qui est indépendante de sa plateforme d'exécution. De cette manière, un service ne connaît pas son contexte d'utilisation. En plus, il peut être mis à disposition par un fournisseur et un utilisateur peut l'utiliser en ayant connaissance uniquement sa description, sans connaître réellement les détails liés à la technologie utilisée lors de sa réalisation, ce qui offre un couplage faible entre les fournisseurs et utilisateurs de services.

A. ARSANJANI [2004]

"A service is software resource (discoverable) with an externalized service description. This service description is available for searching, binding, and invocation by a service consumer. Services should ideally be governed by declarative policies and thus support a dynamically reconfiguration architectural style."

Cette définition [Ars04] spécifie qu'un service doit posséder une description accessible depuis l'extérieur. Cette description permet aux consommateurs (utilisateurs) du service de la rechercher, la trouver, se lier et invoquer le service correspondant à cette description. Ainsi, cette définition décrit le modèle d'interaction existant entre les fournisseurs et les consommateurs de services. Elle précise aussi qu'un service peut disposer d'un ensemble de politiques (propriétés) exprimées de façon déclarative, ce qui offre la possibilité de décrire des propriétés non fonctionnelles et permet la reconfiguration dynamique de services.

En résumé, un service possède les caractéristiques suivantes :

- Un service possède une description rendue accessible par des fournisseurs pour des éventuels utilisateurs.
- La description d'un service peut contenir l'ensemble des fonctionnalités fournies par ce service, et ainsi que ses propriétés.
- La description de service ne contient pas les détails liés à la technologie utilisée, tel que le langage de programmation.
- Un utilisateur de service est capable de rechercher une description correspondant à un service donné pour l'invoquer à tout moment.

L'approche à service définit un modèle d'interaction basé sur différents acteurs permettant l'intégration facile d'applications hétérogènes sans les modifier et facilite aussi

leur évolution d'une manière indépendante. Ce modèle d'interaction est composé de trois acteurs :

- *Fournisseur de service* : un fournisseur de service réalise un ensemble de fonctionnalités en termes de services qui sont décrites dans une description appelée spécification de service. Une spécification de service contient l'ensemble des informations nécessaires pour l'utilisation du service. Ces informations peuvent exprimer les caractéristiques fonctionnelles du service ou les propriétés non fonctionnelles telles que la qualité de service.
- *Consommateur de service* : un service peut être utilisé par un ou plusieurs utilisateurs appelé(s) consommateur(s). Un consommateur de service utilise une spécification de service pour exécuter les fonctionnalités fournies par ce service.
- *Annuaire de services* (ou courtier de services) : les fournisseurs de services publient leurs spécifications de service dans un registre appelé annuaire de services. En plus des spécifications de service, l'annuaire de services contient des références vers les fournisseurs de services. Le rôle de l'annuaire est essentiellement de permettre aux consommateurs de localiser les fournisseurs des spécifications de service qui correspondent à leurs besoins.

En d'autres termes, l'annuaire de services joue le rôle d'intermédiaire entre les fournisseurs et les consommateurs de service. Dans un premier temps, les fournisseurs publient dans l'annuaire leurs spécifications de service. Ensuite, les consommateurs de service peuvent les rechercher à partir de cet annuaire pour les découvrir, puis les utiliser même sans connaître forcément les fournisseurs de ces services. Ainsi, l'approche à service permet d'obtenir un couplage faible entre les consommateurs et les fournisseurs de services dans la mesure où ces derniers ne partagent que la spécification de service. La figure suivante illustre le modèle proposé par l'approche orientée service décrivant les interactions entre les différents acteurs.

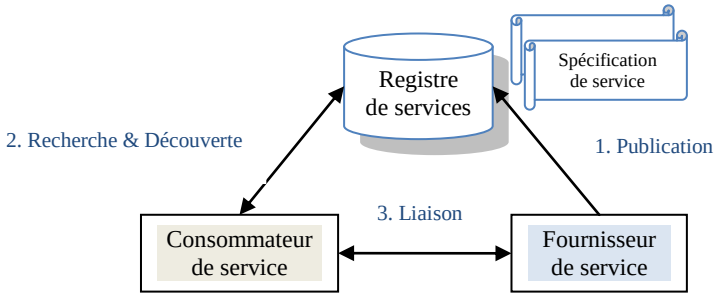

figure 2. Modèle d'interaction de l'approche à service.

Dans la figure 2, nous notons également que l'approche à service propose trois primitives permettant de réaliser les communications nécessaires du modèle d'interaction. Ces primitives sont :

- *Publication de service* : permet aux fournisseurs de service d'enregistrer les services qu'ils fournissent. Cette interaction s'effectue entre les fournisseurs et l'annuaire de service.
- *Recherche et découverte d'un service* : permet aux consommateurs de service de rechercher et découvrir les fournisseurs de service qu'ils requièrent. Cette interaction a lieu entre le consommateur et l'annuaire de service.

- *Liaison à un service* : une fois que le service requis est découvert par le consommateur, ce dernier peut se lier à un fournisseur de ce service pour utiliser les fonctionnalités qu'il fournit. Cette interaction s'effectue entre le consommateur et le fournisseur de service.

La réalisation des primitives d'interaction peut se faire à n'importe quel moment du cycle de vie de l'application. Elle peut se faire à la phase de conception (développement), au déploiement ou à l'étape d'exécution de l'application. Ainsi, l'approche à service permet une liaison tardive entre les fournisseurs et les consommateurs de service. Cette propriété de liaison tardive offre la possibilité de sélectionner dynamiquement (à l'exécution) les fournisseurs des services requis par l'application.

La réalisation des primitives de publication, découverte et liaison à l'exécution permet de supporter le dynamisme dans l'approche à service. Ainsi, nous parlerons d'approche à service dynamique. Cependant, il faudrait ajouter deux nouvelles primitives pour la gestion du dynamisme des services. Ces primitives sont :

- *Retrait de service* : dans l'approche à service dynamique, le fournisseur doit retirer du registre ses services si ces derniers ne sont plus disponibles.
- *Notification* : les consommateurs de service sont notifiés des départs de services qu'ils utilisent et qui ne sont plus disponibles. De même, ils sont notifiés de l'arrivée de nouveaux services correspondant à leurs besoins.

La figure suivante décrit le principe de retrait et d'ajout de service :

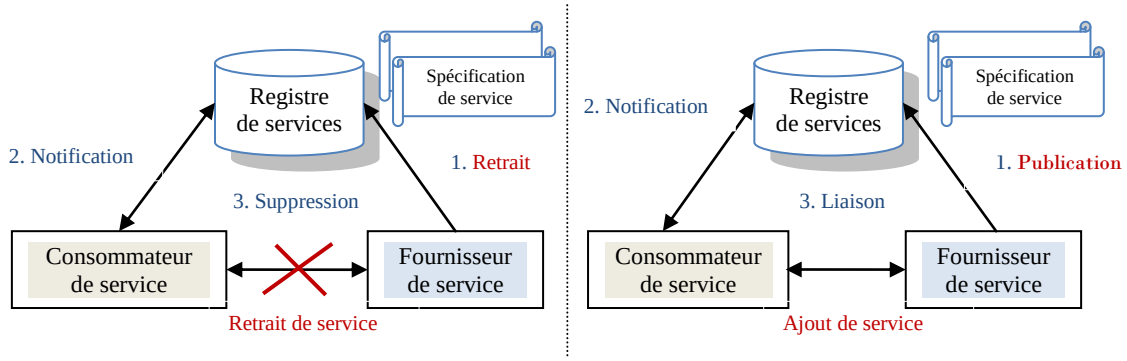

figure 3. Principe de retrait et d'ajout de service.

2.2.2 Les avantages de l'approche à service.

L'architecture à service apporte les avantages majeurs suivants :

- **Réutilisation** : elle offre la possibilité de mettre en œuvre de nouveaux services à partir de ceux existants. Les services sont ainsi réutilisables dans une application.
- **Couplage faible et liaison retardée** : le modèle d'interaction proposé par l'approche à service offre un couplage faible entre les consommateurs et les fournisseurs de service. En plus, l'approche à service offre la capacité de liaison tardive entre les fournisseurs et les consommateurs de service.
- **Evolutivité et souplesse** : elle offre la possibilité de modifier ou de substituer, à l'exécution, les services utilisés par l'application pour satisfaire par exemple les

nouveaux besoins de l'entreprise, supportant ainsi le dynamisme au sein des applications.

2.2.3 La plateforme à service OSGi.

Il existe actuellement plusieurs plates-formes à service, issues de l'industrie ou de la recherche, telles que les services web [ACKM03][W3C04b] ou la technologie OSGi [OSG05a]. Dans ce document, nous présentons seulement la technologie OSGi parce que celle des services web est connue et très répandue. La technologie OSGi est une spécification définie en 1999 par le consortium *OSGi Alliance* [OSG05a] regroupant plusieurs entreprises telles que IBM, Oracle et Motorola. Ce consortium fournit un ensemble de spécifications spécifiant une plateforme à service dynamique, un modèle de déploiement et d'exécution de service et un ensemble de services techniques [OSG05b].

A l'origine, la spécification OSGi a été conçue pour des passerelles résidentielles. Toutefois, elle est utilisée aujourd'hui dans d'autres domaines : les serveurs d'applications comme OW2 Jonas [Obj07], les systèmes embarqués tels que les téléphones portables ou les véhicules, ou bien les outils de développement intégrés comme Eclipse [GHM+05][Eclc].

La plateforme à service proposée est basée sur le langage Java et elle est centralisée. Un service OSGi est spécifié sous la forme d'une interface Java avec un ensemble de propriétés (attributs/valeurs) décrivant les informations du service par exemple son fournisseur. Conformément à l'approche à service, la spécification OSGi définit un annuaire de services qui lui est spécifique. Les fournisseurs enregistrent leurs services fournis dans cet annuaire et ont la possibilité de les retirer ou de modifier les propriétés des services à tout moment. Un consommateur de service peut rechercher, dans l'annuaire, un ensemble de fournisseurs d'un service donné dans le but de les invoquer. Dans OSGi, la recherche de service peut être sophistiquée en utilisant des filtres exprimés dans le langage LDAP.

La plateforme OSGi réalise une approche à service dynamique en offrant aux fournisseurs de service la possibilité de retirer leurs services et aux consommateurs de services d'être notifiés quand un service utilisé n'est plus disponible ou quand un nouveau fournisseur de service apparaît. Ainsi, la plateforme OSGi permet de gérer le dynamisme des services. Cependant, les consommateurs de service doivent explicitement libérer les services utilisés lorsqu'ils ne sont plus disponibles. Ce qui constitue une contrainte forte pour les développeurs d'applications.

La figure 4 illustre l'architecture à service OSGi :

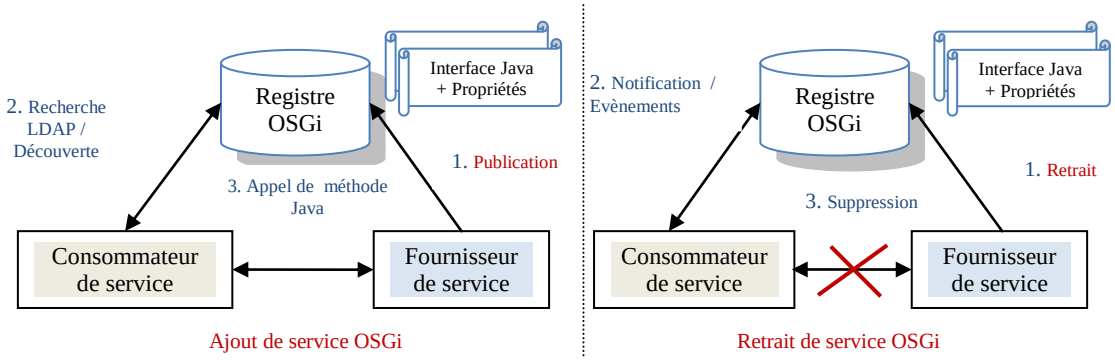

figure 4. Architecture des services OSGi.

En plus du dynamisme, c'est-à-dire la gestion de l'apparition et de la disparition de fournisseurs de service, la spécification OSGi définit un modèle de paquetage (*packaking*) et de déploiement de service OSGi. Ce modèle spécifie l'unité de déploiement appelée *bundle*. Un bundle peut fournir ou consommer des services. Les *bundles* peuvent être interconnectés entre eux pour réaliser une tâche particulière et ils peuvent aussi être administrés, à l'exécution, par la plateforme d'exécution. La plateforme d'exécution gère également le cycle de vie des *bundles*. Ainsi, un *bundle* peut être installé, démarré, arrêté, supprimé ou mis à jour à l'intérieur de la plateforme d'exécution, sans obligatoirement interrompre l'exécution de cette dernière.

Le tableau 2 contraste la plateforme OSGi avec les services web.

	Services web	OSGi
Description de service	Interface WSDL[1].	Interface Java (avec des propriétés).
Courtier de service	Registre UDDI[2] (peu utilisé en pratique). Peut être décentralisé.	Registre Spécifique OSGi. Centralisé.
Communications	Protocole SOAP[3] (synchrone & asynchrone).	Appels de méthodes (Java).
Notifications	Pas de notifications aux clients pour le retrait ou l'ajout d'un service web.	Notification par des évènements de l'arrivée ou du départ d'un service.

tableau 2. Caractéristiques des plates-formes à services.

2.2.4 Synthèse.

En conclusion, nous pouvons noter que l'approche à service améliore la réutilisation grâce à la réalisation de nouveaux services à partir de ceux qui existent et qui sont disponibles. L'architecture à service définit un ensemble d'acteurs et de capacités natives de communication entre ces acteurs. Elle permet de découpler les fournisseurs des consommateurs de services au niveau de leurs interactions. Ainsi, elle apporte plus de flexibilité pour la construction d'applications et permet de résoudre les problèmes d'interopérabilités entre ces applications.

La gestion du dynamisme, l'arrivée et le départ de service, est une caractéristique majeure de l'approche à service qui offre plus de souplesse aux applications. Actuellement, il existe plusieurs plates-formes à service mettant en place l'approche à service. Nous avons étudié particulièrement les services OSGi.

La plateforme OSGi fournit un modèle à service dynamique. Cependant, il s'avère difficile de gérer le dynamisme (c'est-à-dire l'arrivée et le départ de services) parce que les consommateurs de services doivent explicitement relâcher les références des services qu'ils utilisent quand ces derniers ne sont plus mis à leur disposition. En plus, l'implémentation du

[1] Web Service Description Language [W3C02]

[2] Universal Description, Discovery and Integration [OAS04]

[3] Simple Object Access Protocol [W3C00]

service contient à la fois le code fonctionnel et celui concernant les aspects non fonctionnels tels que la gestion du dynamisme ou du cycle de vie des services. Enfin, la technologie OSGi est centralisée et ne résout pas le problème d'intégration de systèmes informatiques hétérogènes. Le tableau suivant récapitule les avantages et les limites des technologies à service :

Avantages	Limites
Favorise la réutilisation.	Elle ne fournit pas de mécanisme de création d'applications (composition).
Offre un couplage faible entre les fournisseurs et consommateurs de service.	OSGi : ne permet pas l'intégration de systèmes hétérogènes.
Capacité de liaison tardive.	Services web : ne gère pas le cycle de vie des services ni leur disponibilité dynamisme.
Assure la flexibilité, l'évolutivité et l'interopérabilité des applications.	OSGi : plateforme centralisée et non interopérable.
Offre une souplesse et la gestion du dynamisme.	La séparation entre les interfaces fournies et requises n'est pas exprimée explicitement dans la spécification de service (comme WSDL).

tableau 3. Récapitulatif des avantages et limites de la technologie à service.

2.3 COMPOSANTS ORIENTES SERVICES.

Les sections précédentes ont mis en évidence les avantages et limitations des approches à composant et à service. En effet, l'approche à composant ne permet pas de gérer la disponibilité des composants (comme dans l'approche à service). Elle offre cependant l'avantage de séparer les aspects fonctionnels et non fonctionnels du composant et de gérer son cycle de vie. Ces aspects ne sont pas supportés par l'approche à service.

Les approches à composant et à service sont cependant complémentaires [Krä08]. Dans cette section, nous introduisons un concept nommé composant orienté service, apparu pour combiner les avantages des approches à composant et à service et améliorer ainsi la construction d'applications à service. Nous présentons aussi quelques modèles mettant en œuvre le concept de composant orienté service.

2.3.1 Motivations et objectifs.

Le concept de composant orienté service a été proposé en 2004 [Cer04] dans l'intention de combiner les avantages majeurs des approches à composant et à service. Ainsi, il permet de mettre en œuvre l'approche à service tout en ayant les propriétés essentielles de l'approche à composant. En fait, le concept de composant orienté service propose de faciliter la réalisation de services en reprenant des composants l'idée de la séparation de préoccupation et la définition explicite des fonctionnalités requises.

De ce fait, le développeur réalise seulement le code fonctionnel (le contenu) des fonctionnalités métiers offertes et les aspects non fonctionnels tels que la gestion de la disponibilité dynamique ou du cycle de vie des services (composants) sont gérés par le mécanisme de conteneur. Dans certaines technologies à service comme OSGi, le code qui gère la disponibilité dynamique des services est assez complexe et est à la charge du développeur. L'utilisation de conteneurs, inspiré des composants, pour gérer des aspects non fonctionnels

comme la disponibilité dynamique, va simplifier la tâche des développeurs parce qu'ils gèrent seulement les aspects fonctionnels.

Le tableau 4 décrit les avantages du concept de composant orienté service :

Avantages du concept de composant orienté service
Rend explicite les fonctionnalités fournies et requises par un composant : le composant peut utiliser des services et il peut en fournir.
Dépendances entre composants sont spécifiées en termes de services : permet de réduire les dépendances (directes) entre composants et faciliter ainsi leur évolution. L'assemblage est réalisé en termes de services ; un fournisseur de service peut être substitué par un autre fournissant le même service.

tableau 4. Avantages du concept de composant orienté service.

2.3.2 Approches et plates-formes existantes.

Il existe diverses technologies permettant de mettre en place des modèles à composant orienté service. Dans cette section nous en présentons deux parmi les plus utilisées : la spécification *SCA*[1] [OSO07][BII+05][Cha07] et le modèle *iPOJO*[2] [Esc08, EH07, EHL07].

Service Component Architecture (SCA)

SCA est issue d'un projet de la collaboration OSOA[3] dont l'objectif est de fournir un ensemble de spécifications incluant d'une part un modèle pour la création de composants et d'autre part, un modèle de programmation permettant de construire des applications logicielles basées sur l'architecture à service. Dans cette section, nous introduisons uniquement le modèle qui permet de créer des composants logiciels. L'autre modèle sera présenté dans le prochain chapitre (Cf. Chapitre 3).

Dans la spécification SCA, les fonctionnalités métiers des applications sont fournies par un ensemble de composants offrant des services. En d'autres termes, toujours avec l'ambition de satisfaire un ensemble de besoins métiers spécifiques, une application peut être construite en assemblant des composants SCA.

Étant basé sur l'architecture orientée service, SCA bénéficie de tous les avantages offerts par ce dernier, par exemple le découplage de la logique métier des détails techniques en séparant les spécifications de service de leurs implémentations possibles. L'un des avantages majeurs de SCA est le fait que sa spécification supporte plusieurs langages comme Java, C++ ou BPEL[4] [OAS07] pour l'écriture des implémentations de services. La spécification SCA identifie plusieurs concepts tels que *service*, *implémentation*, *composant*, *propriété*, *binding* et *référence*. Le tableau suivant définit les concepts du modèle à composant SCA :

[1] Service Component Architecture

[2] injected POJO (Plain Old Java Object)

[3] Open Service Oriented Architecture

[4] Business Process Execution Language

Concepts	Définitions
Service & Binding	Un *service* est la définition de fonctionnalités métiers, elle comprend, entre autre une interface Java ou WSDL composée de plusieurs opérations, et un ensemble de *bindings* c'est-à-dire les différents mécanismes d'accès à utiliser pour appeler le service.
Implémentation	Une *implémentation* est un bout de code réalisant zéro, un ou plusieurs services. Elle peut être écrite avec plusieurs langages comme Java, C++ ou BPEL.
Composant	Un *composant* est constitué au plus d'une implémentation qui est configurée dans un fichier de configuration décrit avec le langage SCDL[1] qui basé sur XML. La configuration d'une implémentation consiste à affecter des valeurs aux propriétés éditables qu'elle définit.
Propriété	Dans SCA, les *propriétés* sont typées et elles peuvent être de type simple ou complexe, par exemple les types définis dans XML Schéma.
Référence	Les composants SCA offrent leurs fonctionnalités métiers en termes de services et ils peuvent aussi utiliser ou dépendre d'autres services. Ces dépendances sont appelées des *références*.

tableau 5. Concepts du modèle à composant SCA.

La figure 5 décrit un exemple de composant SCA :

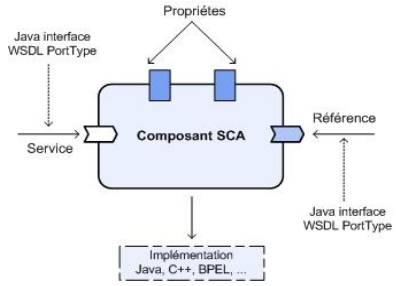

figure 5. Exemple de composant SCA.

La figure 6 illustre les relations entre les concepts SCA :

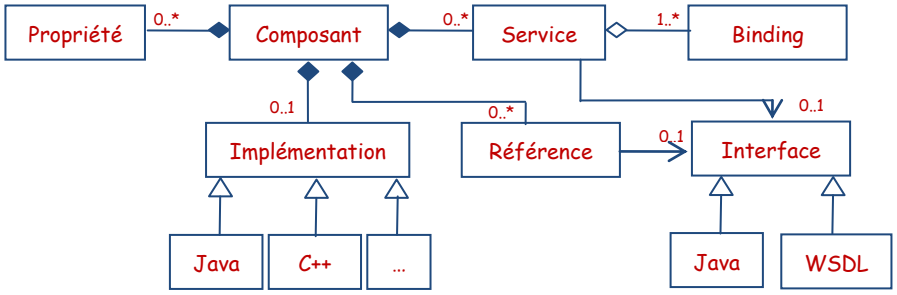

figure 6. Relation entre les concepts SCA.

La spécification SCA ne contient pas de mécanisme pour la gestion de la disponibilité dynamisme de composants (offrant des services), ce qui constitue ainsi une limitation

[1] Service Component Definition Language

majeure de SCA. De nos jours, il existe plusieurs implémentations de la spécification SCA telles que *Apache Tuscany* [Apa08], *IBM WebSphere* [IBM], *Newton* [New08] ou *FraSCati* [OW2][SMF+09]. Certaines plates-formes SCA comme *Newton* ou *FraSCati* gèrent le dynamisme en s'appuyant sur les technologies à service dynamique comme OSGi ou bien en étendant le modèle fourni par la spécification SCA. Toutefois, la plupart des implémentations de SCA ne fournissent pas de support pour gérer la disponibilité dynamique de composants.

injected Plain Old Java Object (iPOJO)

iPOJO est un modèle à composant orienté service développé par l'équipe ADELE du laboratoire LIG (Laboratoire d'Informatique de Grenoble) [lig]. Le modèle iPOJO est développé au dessus de la technologie OSGi et il est intégré actuellement au projet Apache Felix [Apa] (une plateforme qui met en œuvre la spécification OSGi). L'objectif du modèle iPOJO est de fournir une infrastructure de développement et d'exécution d'applications dynamiques suivant le paradigme à service. Pour cela, le modèle iPOJO se base sur plusieurs concepts tels que *spécification de service*, *propriété*, *type de composant* (ou *composant*), *instance* et *conteneur*.

Dans le modèle iPOJO, une *spécification* représente la description d'un service en termes d'interface Java et potentiellement avec un ensemble de propriétés. Un *composant* peut fournir des fonctionnalités à travers une ou plusieurs spécifications de service et il peut aussi exprimer ses dépendances vers d'autres spécifications. Ces dépendances sont exprimées en fonction des spécifications de service, ce qui offre la possibilité de retarder (jusqu'à l'exécution) la réalisation des liaisons entre un composant et les autres fournissant ses spécifications requises. La réalisation (ou la résolution) dynamique des dépendances de services fait partie des points forts du modèle iPOJO. Un composant peut posséder plusieurs *instances* pouvant être configurées différemment.

Le modèle iPOJO utilise les principes de la séparation de préoccupations en se servant du concept de *conteneur* qui est défini dans l'approche à composant pour effectuer une séparation entre la logique métier et les aspects non fonctionnels des composants. Chaque composant iPOJO est associé à un conteneur chargé de la gestion de ses aspects non fonctionnels. Un conteneur est composé de plusieurs «*handlers* » et chaque « *handler* » gère un seul aspect non fonctionnel. L'usage du concept de conteneur permet de faciliter la tâche des développeurs dans la mesure où le conteneur peut gérer, entre autre, tous les aspects techniques liés à la technologie de services, par exemple la publication, la recherche, la découverte et la sélection de services. En effet, le développeur réalise son composant comme un simple POJO. Cette facilité au niveau du développement des composants constitue aussi un point fort du modèle iPOJO.

La plateforme d'exécution associée au modèle iPOJO possède un ensemble de « *handlers* » prédéfinis, par exemple *Service Dependency Handler* qui assure la réalisation des dépendances dynamiques entre les services, *Lifecycle Callback Handler* qui gère le cycle de vie des instances de composants ou bien *Configuration Handler* qui permet la configuration et la reconfiguration des instances de composants. Le modèle iPOJO est extensible parce que d'autres *handlers* peuvent être ajoutés pour gérer d'autres aspects non fonctionnels comme la sécurité, les transactions ou les temps de réponses des services.

Un composant iPOJO peut spécifier des politiques de liaison (*binding policy*) sur ses dépendances de services avec les propriétés *aggregate*, *mandatory*, *dynamic* et/ou *filtered*. Ces propriétés indiquent respectivement : si le composant requiert un ou plusieurs fournisseurs

de service, s'il s'agit d'une dépendance qui est obligatoire ou optionnelle, si la liaison est statique ou dynamique et/ou s'il peut filtrer l'ensemble des fournisseurs disponibles.

Le tableau 6 résume les concepts de base du modèle iPOJO :

Concepts	Définitions
Spécification de Service	Description de service sous la forme d'interface Java et potentiellement avec un ensemble de *propriétés* (paires d'attributs/valeurs).
Composant	Il peut fournir des fonctionnalités à travers une ou plusieurs spécifications de service et il peut aussi exprimer ses dépendances vers d'autres spécifications.
Instance	Un composant peut posséder plusieurs *instances* pouvant être configurées différemment.
Conteneur	Le conteneur gère les aspects non fonctionnels. Il est composé de plusieurs « handlers » gérant chacun un aspect non fonctionnel.

tableau 6. Concepts du modèle iPOJO.

Les figures (figure 7 et figure 8) décrivent respectivement un exemple de composant iPOJO et les relations existantes entre les concepts iPOJO :

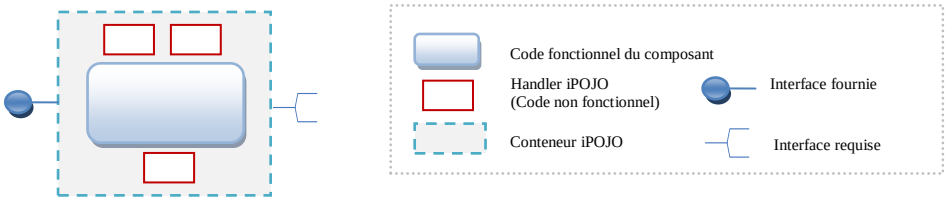

figure 7. Exemple de composant iPOJO.

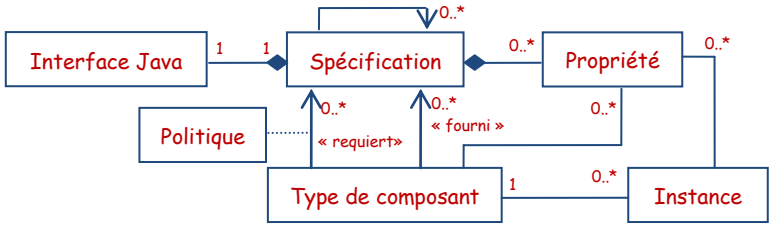

figure 8. Relations entre les concepts iPOJO.

En plus du modèle de développent, iPOJO définit un langage de description d'applications dynamiques. Nous allons présenter ce langage dans le chapitre 3.

2.3.3 Synthèse.

Les modèles à composant orienté service combinent les principes des approches à composant et à service pour simplifier la création d'applications. Dans les sections précédentes, nous avons introduits quelques modèles (SCA et iPOJO) qui sont actuellement les plus connus et utilisés.

Le modèle iPOJO simplifie le développement des composants de l'application en utilisant le concept de conteneur. En fait, le développeur réalise uniquement le code métier et les aspects non fonctionnels sont gérés par un conteneur composé de plusieurs *handlers* (prenant en charge une propriété non fonctionnelle). Le modèle iPOJO est extensible parce

qu'en plus des *handlers* prédéfinis le développeur peut en ajouter d'autres dans l'infrastructure d'exécution.

L'avantage majeur de SCA est le fait de supporter (1) l'hétérogénéité c'est-à-dire divers langages de réalisation de composants et (2) l'utilisation de plusieurs protocoles de communication tels que SOAP ou RMI pour assurer les interactions entre les composants. La spécification SCA ne décrit pas de mécanisme pour la gestion de la disponibilité dynamique de composants. Ainsi, chaque plateforme mettant en place la spécification SCA est libre de spécifier la manière d'assurer cette propriété de dynamisme de composants (un point essentiel pour la création d'applications dynamiques).

Les modèles SCA et iPOJO ne fournissent pas d'environnements complets et spécialisés pour la réalisation de composants ou d'applications dynamiques. Un environnement a pour objectif de simplifier la tâche des développeurs en guidant la génération du code nécessaire et faciliter le déploiement (paquetage) des composants. Toutefois, il existe quelques outils fournis sous la forme de plug-ins Eclipse pour le développement de composants SCA ou iPOJO. Mais en général ces outils ne sont pas assez complets et seul un utilisateur avancé peut facilement les utiliser.

Dans le modèle iPOJO, nous retrouvons quelques limitations de la technologie OSGi dans la mesure où il est développé au dessus de cette technologie. Par exemple, iPOJO est une plateforme centralisée. Par contre, la spécification SCA supporte une exécution distribuée de l'application.

Quelques avantages et limitations des modèles SCA et iPOJO sont décrits dans le tableau suivant :

	SCA	iPOJO
Hétérogénéité (langages …)	Oui.	Java seulement.
Dynamisme	La spécification ne dit rien.	Oui.
Support de création d'applications	Non.	Non.
Protocoles de communication	SOAP, RMI,…	Appel de méthode Java.
Propriétés non fonctionnelles	Oui, prédéfinis	Oui, ouvert (*handlers*).
Exécution distribuée	Oui.	Non.

tableau 7. SCA versus iPOJO.

2.4 ARCHITECTURES LOGICIELLES.

L'ingénierie logicielle possède plusieurs objectifs et défis dans l'intention d'améliorer la réalisation et l'exécution de systèmes logiciels complexes. Parmi ces objectifs, nous pouvons citer :

- maîtriser la complexité des systèmes logiciels en augmentant leur niveau d'abstraction lors de leur spécification et conception,
- favoriser la réutilisation de briques logicielles lors de la construction de systèmes logiciels,

- améliorer (ou assister) la conception, l'analyse (la compréhension) ou le déploiement de systèmes complexes,
- favoriser leur évolution dans le temps, avant ou pendant leur exécution.

Les approches présentées dans les sections précédentes (Cf. Sections 2.1, 2.2 et 2.3) de ce chapitre proposent des solutions répondant aux problématiques liées à la construction de systèmes complexes favorisant la réutilisation d'entités logicielles (composants, services ou composants à service). En général, la conception de ces systèmes repose sur la notion commune d'architecture logicielle qui décrit l'ensemble des entités logicielles constituant le système et les différentes interactions existantes entre ces entités.

L'augmentation du niveau d'abstraction de la spécification des systèmes logiciels est un point important lors de leur conception et réalisation. D'après GARLAN et SHAW [GS93][SG95], la description d'une architecture logicielle est une étape nécessaire permettant d'augmenter le niveau d'abstraction des systèmes logiciels lors de leur modélisation, conception et réalisation.

Une architecture logicielle offre une vision globale de l'application modélisée [Ves93]. Elle correspond à la description « abstraite » de l'application c'est-à-dire un modèle « abstrait » de cette dernière permettant aux architectes de se concentrer uniquement sur l'analyse des besoins et des décisions capitales pour mieux structurer leurs applications répondant aux exigences requises. La plupart du temps, l'architecture d'une application est spécifiée de manière informelle, souvent en termes de « boites » et de « flèches », sans sémantique précise associée.

C'est la raison pour laquelle, de nombreux langages de description d'architecture (ADLs[1]) sont apparus dans le but de fournir une réponse à cette problématique. Un ADL permet de définir un vocabulaire « commun » entre les différents acteurs (architectes/concepteurs, développeurs ou assembleurs) participant depuis la conception jusqu'à l'exécution du système logiciel. Il permet de spécifier l'ensemble des composants d'une manière « abstraite » c'est-à-dire indépendamment des technologies de réalisation des composants, les interactions pouvant explicitement exister entre ces composants et les critères fonctionnels et/ou non fonctionnels à respecter. En d'autres termes, un langage de description d'architecture permet de définir la structure et les propriétés (ou contraintes) de l'application modélisée [SG95].

Les rôles de la description d'architecture logicielle sont [GS93] :

- permettre de décrire la structure du logiciel,
- faciliter l'analyse et la compréhension du logiciel à partir de sa description,
- définir les invariants du système et servir de pivot pour son évolution,
- permettre la génération de code,
- améliorer la réutilisation.

Après avoir introduit la notion d'architecture logicielle, nous présentons dans la suite de cette section, ses principaux concepts, les besoins et défis à surmonter dans le

[1] Architecture Description Languages

domaine des architectures logicielles et un tour d'horizon de quelques langages et outils de description d'architectures.

2.4.1 Concepts de base d'une architecture logicielle.

Dans le monde de la recherche, il n'existe pas de définition universelle d'une architecture logicielle mais un « consensus » sur les concepts essentiels [MT00]. D'après les définitions et les canevas proposés dans la littérature [PW92] [GAaCB95] [Cle96] [Wol97] [MT00], les principaux concepts d'une architecture logicielle sont *composant*, *connecteur* et *configuration* :

- **Composant** : correspond à une unité de calcul ou de stockage [MT00] qui expose ses fonctionnalités à travers une interface. L'interface du composant est le point d'accès et de communication du composant avec le monde extérieur. Elle devrait décrire les fonctionnalités fournies et requises.
- **Connecteur** : correspond à un élément architectural qui permet de modéliser les interactions entre les composants d'une architecture. Un connecteur peut disposer d'un ensemble de règles de contraintes gouvernant ces interactions, par exemple les protocoles de communication entre les composants.
- **Configuration** : élément architectural qui exprime la structure d'un système, sous la forme ensemble de composants et de connecteurs. Il s'agit du résultat de l'assemblage des composants et peut servir par exemple lors de l'analyse et de la validation de l'architecture (par exemple les correspondances entre les interfaces fournies et requises ou la validation du respect des règles de contraintes).

Nous retrouvons certains concepts de l'approche à composant dans les approches de description d'architectures logicielles. C'est pourquoi, les travaux sur les modèles à composant et la description d'architectures logicielles ont vite convergés vers la description de composants composites, appelés des configurations.

Nous avons retenu la définition suivante d'une architecture logicielle ; une architecture décrit la structure globale et les propriétés du système modélisé :

> M. SHAW et D. GARLAN [1995]
>
> *"Software architecture is the level of software design that addresses the overall structure and properties of software systems."*

2.4.2 Problèmes, besoins et défis.

L'assemblage des composants constituant l'application peut être une tâche complexe dans la mesure où les composants peuvent par exemple être conçus pour différents contextes. La description des composants et celle de l'architecture logicielle sont indépendantes ; souvent les descriptions des composants sont spécifiées « à la main » et l'assemblage des composants est peu assisté.

Ainsi, les modèles décrivant les composants et l'architecture logicielle de l'application sont peu sûrs, et leur maintenance est problématique. Généralement, ce sont des modèles descriptifs (versus opérationnels/exécutables) et c'est surtout ce qui a provoqué l'abandon des langages d'ADLs, parce qu'il existe un véritable gap entre la spécification et

la réalisation concrète de l'architecture [POR98]. Dans la plupart des cas, la spécification de l'architecture n'est pas manipulable à l'exécution de l'application.

La principale motivation pour fournir des langages formels de description d'architectures logicielles est de faciliter l'analyse, le raisonnement et la manipulation d'architectures grâce à des outils logiciels. L'utilité d'un ADL est donc souvent liée à l'outillage fourni comme support lors de la conception et/ou de l'analyse de l'architecture ou la génération du code exécutable de l'application à partir de sa description architecturale. Nous ressentons ainsi un ensemble de besoins en termes d'outils logiciels pour :

- **spécifier l'architecture (et les composants logiciels)**. Cette spécification devrait être « abstraite » par rapport aux systèmes modélisés. Une solution est d'utiliser les principes de l'ingénierie dirigée par les modèles (détaillés dans la section 2.5 de ce chapitre) pour la définition d'architectures à partir de modèles indépendants des technologies de réalisation de l'architecture,
- **faciliter l'analyse et le raisonnement** sur l'architecture de l'application modélisée [dPJC00] [RM07],
- **offrir plusieurs facettes** (vues) de l'architecture aux différents acteurs (par exemple les architectes/concepteurs, développeurs ou assembleurs) participant à la réalisation de l'application. Ces acteurs partagent l'architecture logicielle de l'application ; souvent ils ont besoins d'en avoir différentes visions en fonction de leurs connaissances métiers spécifiques et de leurs rôles dans l'application [Das07],
- **raffiner une architecture** : réduire la distance entre la spécification et la réalisation concrète de l'architecture en fournissant des outils pour guider le raffinement d'architectures logicielles. Le processus de raffinement d'architecture peut être complexe parce qu'il faudrait assurer la cohérence et la conformité (l'exactitude) de l'architecture, entre sa spécification et sa réalisation ; ces propriétés ne sont souvent pas garanties,
- **garantir la fiabilité et la cohérence** des descriptions d'architectures et des composants,
- **prendre en compte le caractère évolutif de l'architecture** : fournir la capacité d'évolution des systèmes existants pour s'adapter à de nouveaux besoins ; cette évolution est souvent contrainte par la disponibilité dynamique des composants ou services utilisés. L'évolution d'une architecture consiste par exemple à ajouter de nouveaux composants, ajouter de nouvelles fonctionnalités, substituer, adapter ou mettre à jour un composant de l'architecture, s'adapter à l'environnement d'exécution de l'application pour assurer ses propriétés désirées, etc. [BMDL08][Bar05][GS09][Cha02]. L'évolution de l'architecture doit aussi tenir compte de celle des composants (diverses versions d'un composant peuvent exister), ce qui rend complexe la prise en charge de l'évolution de l'architecture,
- **construire d'une manière incrémentale l'architecture** : fournir des supports facilitant la spécification fonctionnelle de l'architecture (modèle métier) vers sa concrétisation [Bar05], ce qui constitue l'une des limitations des ADLs existants (la réalisation est souvent manuelle) et l'architecture n'est pas toujours exécutable [POR98],
- **supporter des architectures logicielles dynamiques** : la réalisation et l'exécution d'applications dynamiques, où les composants peuvent être disponibles ou pas à tout moment, implique de considérer la modification à l'exécution de

l'architecture : ajout ou suppression de composants, création ou suppression de connecteurs entre les composants, etc. C'est pourquoi, plusieurs travaux de recherche [Bra04][BCDW04] [BPPT06][And00][YML05] tentent de donner des solutions pour gérer les architectures logicielles dynamiques. Nous avons retenu la définition suivante d'une architecture logicielle dynamique :

Architecture dynamique.

Une architecture logicielle est dynamique si elle est modifiable lors de l'exécution du système modélisé. Une modification de l'architecture peut concerner par exemple l'ajout ou la suppression de nouveaux composants, la création ou la suppression de connecteurs entre les composants.

Un système logiciel possède un cycle de vie complexe qui est composé de plusieurs phases : spécification, conception, réalisation, déploiement, assemblage, exécution, etc. La description de l'architecture logicielle d'un système, en termes de configurations spécifiant la structure globale du système, est souvent insuffisante pour faciliter l'usage, l'évaluation et la validation de cette description d'architecture tout au long du cycle de vie du système [GAaCB95]. Par exemple, l'évaluation et la validation d'une architecture peuvent consister à capturer les modifications ultérieures du système, puis à vérifier et assurer qu'elles ne violent pas les propriétés requises par le système. C'est pourquoi, dans [GAaCB95] les auteurs proposaient d'étendre les modèles d'architectures existants pour mieux supporter le cycle de vie des systèmes logiciels.

Nous ressentons ainsi le besoin de fournir des supports logiciels, tout au long du cycle de vie du système, pour faciliter et simplifier par exemple le déploiement des éléments logiciels de base et l'analyse, la validation ou l'administration du système.

2.4.3 Synthèse : comparaison de quelques langages et outils.

Dans cette section, nous présentons un tour d'horizon sur un ensemble de langages et outils de description d'architectures logicielles (ADLs). Nous réalisons une étude comparative de ces langages et outils en nous basant sur divers critères que nous jugeons pertinents pour la description d'architectures logicielles. Les critères considérés permettent d'évaluer les capacités du langage de description d'architectures et de présenter les caractéristiques fournies par les outils associés :

Architecture et supports durant tout le cycle de vie du logiciel.

- Spécification d'architectures logicielles : Existe-t-il un outil ou un éditeur (textuel ou graphique) dédié à la spécification d'architectures logicielles ?
- Raffinement d'architectures : Est-ce que les langages offrent la capacité de construire d'une manière incrémentale une architecture logicielle ? Existe-t-il des supports pour raffiner (rendre concrète) une architecture logicielle ?
- Construction d'applications : Existe-t-il des supports logiciels pour guider et faciliter la création d'applications « exécutables » à partir de l'architecture ?
- Packaging et déploiement : Existe-t-il des supports logiciels pour faciliter le packaging et le déploiement concret des applications ?
- Administration : Existe-t-il des outils pour observer et/ou analyser l'exécution d'une application modélisée par une architecture ?

- Evolution et dynamisme de l'architecture à l'exécution : Est-ce que l'évolution d'une architecture est prise en compte dans le langage ? L'architecture peut-elle être reconfigurée pour faire face à de nouvelles exigences ? De nouveaux éléments logiciels (ou connecteurs) peuvent-elle être ajoutés (ou supprimés) dans l'architecture ?

Analyse, vérification, validation et exécution d'architectures.

- Analyse et validation syntaxique et sémantique : Existe-t-il des outils pour vérifier et valider la syntaxe et la sémantique de l'architecture ?
- Vérification de la complétude de l'architecture : Existe-t-il des outils pour vérifier et garantir la complétude d'une architecture (tous les composants qui sont nécessaires sont présents) ?
- Vérification de la cohérence de l'architecture : Existe-t-il des outils pour vérifier et garantir la cohérence d'une architecture logicielle : les contraintes définies à la conception sont-elles toujours vérifiées ? Les incohérences sont-elles notifiées ?
- Disponibilité de l'architecture à l'exécution : L'architecture est-elle consultable à l'exécution (introspection) ? Est-t-elle modifiable (reconfiguration) ? Est-elle exécutable ?

Le tableau 8 illustre le résultat de notre étude comparative de quelques langages et outils de description d'architectures logicielles. Nous divisons ces langages et outils en deux catégories à savoir les langages et les modèles à composant qui fournissent la capacité de description d'architectures logicielles :

- Langages et outils : CommUnity [Com], Wright/Dynamic Wright [Wri][All97], Darwin [JMK95][MK96] et ZCL [dP99][dPJC00].
- Modèles à base de composant et outils : ArchJava [ACN02], ArchStudio [Arc] [Das07], Fractal [BCS04], SOFA [KT02], SafArchie [BD05].

		Supports fournis							Vérifie ou garantie		
		Creation / Specification d'architectures	Validation d'architectures	Raffinement d'architectures	Construction d'applications	Packaging & Deploiement	Monitoring	Dynamisme	Completude	Coherence	Disponibilite de l'architecture a l'execution
Langages	**CommUnity** (2003)	OUI Extension			**NON**		**débuggeur**				
	Dynamic Wright (1997)	**Pas d'outils**	OUI (Extension externe)	**NON :** langage fournit la capacité mais aucun support existe actuellement	**NON**	**NON**	**NON**	OUI	OUI	OUI	**NON**
	Darwin (1995)	OUI	**NON**	OUI	**NON**	OUI (répartition des instances de composants)	**NON**	OUI (contraintes)	**NON**	**NON**	
	ZCL (1999)	OUI	OUI	**NON**	**NON**						
Modèles à composant	**ArchJava** (2002)	OUI	OUI (Architecture statique)			**Pas spécifié**	**NON**	OUI (limité)		OUI	**NON** Pas de plateforme d'exécution
	ArchStudio	OUI	OUI (syntaxique)	OUI (sélection des variantes et version)	OUI		**NON**				
	Fractal (2004)	OUI	OUI (Extension)	OUI	OUI	OUI Placement manuel ; Extension FROGi : approche à service ;	OUI contrôle de l'exécution (introspection)	OUI			OUI
	SOFA (2002)	OUI	OUI (statique)		OUI	OUI	OUI (extension DCUP)	OUI - restreinte (remplacement à l'exécution de composants)			
	SafArchie (2005)	OUI : Etend ArgoUML	OUI	OUI (Extension TranSAT)	OUI	OUI Déploiement simple.		OUI		OUI	Oui (supervision)

tableau 8. Tableau comparatif de quelques langages et outils d'ADLs.

Le tableau 9 détaille la gestion du dynamisme et de l'évolution de l'architecture des langages et outils dédiés à la description d'architectures logicielles que nous avons étudiés dans le tableau 8 :

		Dynamisme / Evolution de l'architecture				
		Composants dynamiques		**Connexions dynamiques**		**Evolution de la structure de l'architecture**
		AJOUT DE COMPOSANTS	SUPPRESSION DE COMPOSANTS	AJOUT DE CONNEXIONS	SUPPRESSION DE CONNEXIONS	ENSEMBLE DES COMPOSANTS VARIABLE
Langages	**CommUnity** (2003)	OUI	OUI	OUI	OUI	**NON** (ensemble d'éléments fixe)
	Dynamic Wright (1997)	OUI	OUI	OUI	OUI	**NON**
	Darwin (1995)	OUI	OUI (Extension externe)	**NON** Pas applicable (connexions implicites)	**NON** Pas applicable (connexions implicites)	**NON**
	ZCL (1999)	OUI	OUI	OUI	OUI	**NON**
Modèles à composant	**ArchJava** (2002)	OUI (instances)	OUI (instances)	**NON** (instances)	**NON** (instances)	**NON** Architecture statique
	ArchStudio					**NON**
	Fractal (2004)	OUI (Extension FScript)	OUI (Extension FScript)	OUI (Extension FScript)	OUI (Extension FScript)	
	SOFA (2002)	OUI		OUI		**NON** (ensemble d'éléments fixe)
	SafArchie (2005)	OUI (instances)	OUI (instances)	OUI (instances)	OUI (instances)	

tableau 9. Dynamisme et évolution de l'architecture dans les langages d'ADLs.

En conclusion, la description formelle d'architectures logicielles permet d'améliorer la spécification et la conception d'applications complexes, de favoriser la réutilisation lors de la conception d'applications et de faciliter leur analyse et validation. Les tableaux précédents ont mis en évidence les limitations des langages et outils de description d'architectures logicielles :

- la structure de l'architecture est statique en général,
- la complétude et la cohérence de l'architecture ne sont pas toujours garanties,
- le raffinement d'architecture : la concrétisation de l'architecture n'est pas souvent assisté ; l'architecture n'est souvent pas « exécutable »,
- dans la plupart du temps, la spécification d'une architecture logicielle n'est pas manipulable à l'exécution. Il existe un véritable écart entre la spécification d'une architecture et sa réalisation [POR98],
- la gestion du dynamisme limité : les langages et modèles fournissent peu de mécanismes pour spécifier et contraindre le dynamisme de l'architecture. Dans le cas où le dynamisme est géré dans l'architecture, le passage de sa spécification à sa réalisation concrète n'est pas évident. L'utilisation de l'approche à service

donne de nouvelles perspectives qui sont explorées par quelques travaux de recherche [YML05],

- la plupart des approches ne fournissent pas d'outils logiciels par exemple pour la création d'applications, leur déploiement et administration, la sélection des composants constituant l'application, leur localisation ou leur disponibilité.

2.5 L'INGENIERIE DIRIGEE PAR LES MODELES.

L'*Object Management Group* (OMG) a proposé en novembre 2000 une approche nommée MDA[1] [OMG][KWB03] pour tenter de faire face à la prolifération des technologies et langages hétérogènes qui sont utilisés dans les systèmes informatiques. L'OMG essaye d'unifier les technologies qu'il propose et de rendre les systèmes informatiques interopérables. L'approche MDA permet de séparer, lors de la modélisation d'applications, les aspects liés aux plates-formes technologiques des applications et ceux qui sont indépendants des technologies. Elle définit ainsi deux types de modèles :

- Les modèles indépendants des plates-formes (ou en anglais PIM[2]) : décrivant les aspects de l'application qui sont indépendants des plates-formes d'exécution des applications.
- Les modèles dépendants des plates-formes (ou en anglais PSM[3]) : décrivant les aspects de l'application qui sont liés à une technologie (ou un langage) spécifique ; donc à une plate-forme d'exécution particulière.

L'objectif de la séparation de ces types de modèles dans l'approche MDA est de rendre indépendante la spécification de l'application et faciliter la génération du code de l'application vers une technologie cible particulière par une succession de transformations de modèles. L'ingénierie dirigée par les modèles (IDM) reprend les concepts et les principes de base de l'approche MDA dans l'intention d'améliorer et d'automatiser le processus de réalisation des applications logicielles. L'IDM est une généralisation de l'approche MDA [FEBF06]. Ce chapitre introduit la vision de l'IDM et présente succinctement les concepts de base de cette approche.

2.5.1 Origine et motivations.

L'ingénierie dirigée par les modèles [BBB+05] [FEBF06] [BG01] vise à faire face à la complexité, la croissance, l'évolution rapide et l'hétérogénéité des applications logicielles [FR07]. Pour ce faire, elle propose de mettre l'accent sur la modélisation des applications en les spécifiant selon différents points de vue (plusieurs aspects du système). Chaque point de vue ou aspect de l'application correspond à un ou plusieurs modèles (pouvant être exprimés dans divers formalismes ou langages), par exemple les modèles métiers, exprimant un aspect du système avec différents niveaux d'abstraction qui sont souvent assez élevés.

[1] Model Driven Architecture

[2] Platform Independent Models

[3] Platform Specific Models

L'IDM propose ainsi de concevoir les applications logicielles non par programmation en utilisant un langage de programmation générique, mais à partir d'un ensemble de modèles permettant de générer l'artefact exécutable, ce qui permet de réduire le temps de développement des logiciels et d'augmenter le niveau d'abstraction de leur spécification. Dans les approches traditionnelles de développement d'applications, le code source est l'élément central lors de la réalisation des applications. Par contre, dans l'IDM la notion de modèle joue le rôle central dans le processus de développement des logiciels. L'IDM cherche à améliorer le processus de réalisation des applications logicielles et à offrir des moyens permettant de couvrir et gérer « tout » le cycle de vie des applications.

L'IDM est une approche générative c'est-à-dire qu'une partie de l'application (ou « tout ») est générée à partir de modèles. Les modèles doivent donc être explicitement définis et facilement interprétables par un programme pour faciliter la génération automatique du code exécutable de l'application. L'ingénierie dirigée par les modèles cherche à être productive en fournissant des outils et environnements pour automatiser le développement d'applications et qui permettent par exemple de créer des modèles, les transformer, valider des modèles (assurer la cohérence entre modèles) ou générer le code exécutable de l'application à partir des modèles.

L'ingénierie dirigée par les modèles se base sur plusieurs concepts : *modèle*, *langage de modélisation*, *métamodèle*, *transformation* etc. Dans cette section, nous n'aborderons pas le concept de transformation seules les notions de *modèles* et *métamodèles* seront introduites.

2.5.2 Concepts de modèle et de métamodèle.

Le concept de *modèle* n'est pas nouveau et plusieurs définitions de ce concept ont été proposées. Dans les définitions ci-dessous, nous trouvons les caractéristiques d'un modèle :

J. BEZINVIN & O. GERBE [2001]

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

Cette définition [BG01] exprime les caractéristiques suivantes d'un modèle :

- Simplification : un modèle est une simplification d'un système réalisé dans un but particulier. Un modèle peut donc être considéré comme une abstraction d'un système en exprimant une partie de ses informations (en masquant certains détails).
- Simulation (Représentation) : un modèle représente un système et devrait être en mesure de répondre à des questions à la place du système.

A. KLEPPE et al. [2003]

"A model is a description of (part of) a system written in a well-defined language."
"A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer."

Dans cette définition [KWB03], nous trouvons les propriétés suivantes :

- Description (Représentation) : un modèle décrit un système (ou une partie du système). Nous avons donc la relation de représentation entre le modèle et son système correspondant.
- Formalisation : la représentation d'un système doit être exprimée dans un langage bien défini permettant de formaliser explicitement les systèmes informatiques et d'automatiser les traitements sur les modèles.

En résumé, un modèle est une description, une abstraction, une représentation ou une simplification d'un système, exprimé dans un langage bien défini (formalisation).

Dans l'ingénierie dirigée par les modèles, les modèles doivent être clairement et formellement définis pour faciliter leur manipulation par un ordinateur. Le concept de métamodèle est utilisé pour définir le modèle du langage d'expression (ou bien de modélisation) de modèles [OMG][Fav05][JGMB05]. Un modèle doit être conforme à son métamodèle, ce qui permet de garantir qu'un modèle est correct et de pouvoir automatiser les traitements sur le modèle lui même :

OMG [2002]

"A meta-model is a model that defines the language for expressing a model."

La figure 9 décrit les relations entre les concepts de modèle, métamodèle et le système modélisé :

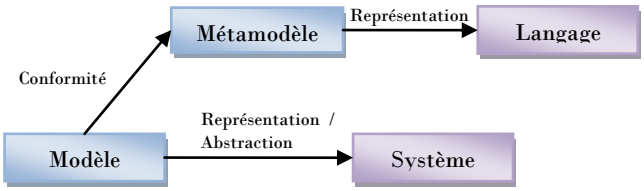

figure 9. Relations entre les concepts de l'IDM.

2.5.3 Langages spécifiques à un domaine particulier.

L'ingénierie dirigée par les modèles cherche à faciliter et automatiser le processus de développement des applications. Comme souligné précédemment, l'IDM propose de définir d'abord l'application d'une manière « abstraite » à travers divers modèles, puis d'effectuer une génération automatique du code exécutable de l'application à partir de ces modèles. La définition abstraite se réalise avec l'usage d'un langage de modélisation qui est souvent exprimé sous la forme de métamodèle.

La recherche et l'industrie, proposent de plus en plus d'outils et de langages de modélisation spécifiques (ou dédiés) à un domaine particulier [Wil01]. L'expression d'un domaine consiste à l'identification et à la définition des concepts liés au domaine et des relations entre ces concepts [Veg05]. Le langage d'expression du modèle de domaine peut être exprimé sous forme textuelle ou graphique afin de faciliter la spécification de modèles par les utilisateurs.

L'utilisation des concepts et principes essentiels de l'IDM dans un contexte particulier permet d'une part de simplifier et faciliter le développement d'applications qui

sont en relation avec ce contexte, et d'autre part de réutiliser l'existant lors de la conception d'applications comme dans les approches orientées ligne de produits [IEE03].

L'approche proposée dans cette thèse se base sur les principes de l'ingénierie dirigée par les modèles dans un domaine particulier ; celui de la construction et de l'exécution d'applications dynamiques à base de services en fournissant un langage de description d'applications.

2.5.4 Synthèse.

L'ingénierie dirigée par les modèles est un domaine de recherche émergeant et de nombreux travaux sont en cours. L'IDM permet d'augmenter le niveau d'abstraction de la définition d'une application en la spécifiant avec divers modèles ce qui permet de séparer les préoccupations (par exemple les aspects fonctionnels et non fonctionnels) de l'application. Elle offre et présente ainsi plusieurs aspects ou points de vue aux utilisateurs en fonction des activités à réaliser. L'IDM propose que les modèles décrivant l'application soient productifs et clairement spécifiés dans un langage bien défini pour faciliter la génération automatique du code exécutable de l'application à partir des modèles.

Les concepts de *modèle* et *métamodèle* permettent respectivement de décrire (ou représenter) d'une manière « abstraite » l'application et de capturer toutes les informations ou propriétés essentielles du langage de modélisation utilisé pour décrire les modèles d'un système. Le métamodèle est un modèle du langage de modélisation ; il permet d'assurer la conformité des modèles (un modèle est conforme à un ou plusieurs métamodèles) et de faciliter le traitement automatique des modèles.

Le tableau suivant synthétise les avantages de l'ingénierie dirigée par les modèles :

Avantages de l'ingénierie dirigée par les modèles
Le concept de modèle permet d'augmenter le niveau d'abstraction de la spécification de l'application indépendamment des technologies existantes.
L'IDM contribue à la maîtrise de la complexité des systèmes informatiques en proposant de les spécifier avec divers modèles. Elle offre la possibilité de séparer les préoccupations (plusieurs aspects ou points de vue de l'application sont modélisés sous forme de modèles).
L'IDM est une approche productive et générative parce que les modèles permettent de générer totalement ou partiellement le code exécutable de l'application.
L'IDM permet d'automatisation le processus de réalisation d'applications et offre des supports pour gérer « tout » le cycle de vie de ces applications.

tableau 10. Avantages de l'ingénierie dirigée par les modèles.

L'IDM est confrontée à de nombreux défis. Par exemple, la définition de langages, métamodèles ou modèles doit être flexible afin de permettre de les étendre ou les adapter à de nouveaux besoins ou fonctionnalités.

3

COMPOSITION DE SERVICES.

Le chapitre précédent a introduit les approches à composant et à service. Ces approches offrent la possibilité de construire des applications en assemblant (ou composant) un ensemble d'éléments exécutables, indépendants et préexistants. Ainsi, elles permettent d'intégrer et de réutiliser des éléments logiciels (appelés composants ou services) et le mécanisme réalisant l'assemblage de ces éléments est appelé une ***composition***. Dans le paradigme orienté service, il s'agit d'une composition de services et en général le résultat est un nouveau service nommé service composite.

Dans ce chapitre, nous décrivons dans un premier temps les différentes phases du processus de réalisation d'une composition de services. Ensuite, nous présentons plusieurs approches, langages et outils issus du monde académique ou industriel qui sont dédiés à la composition de services. Pour terminer, nous présentons une synthèse sur la composition de services. Dans cette synthèse, nous identifions un ensemble de besoins et de défis importants au niveau de la composition de services, en vue d'une part de simplifier et de faciliter le processus de la construction de services composites, et d'autre part de supporter d'autres caractéristiques comme le dynamisme ou l'automatisation du processus de composition.

3.1 PROCESSUS DE LA COMPOSITION DE SERVICES.

La composition de services permet de développer rapidement des applications en réutilisant les services existants pour créer d'autres services plus complexes. Le processus de réalisation d'une composition de services consiste à sélectionner des services, les combiner puis les exécuter afin de réaliser un objectif particulier. L'approche à service spécifie le schéma d'interaction entre les fournisseurs et consommateurs (ou clients) de services afin de les publier, les rechercher et/ou les découvrir. Cependant, l'approche à service n'indique pas comment utiliser divers services par composition, intégration ou assemblage pour réaliser le processus de composition. C'est pourquoi plusieurs approches, langages et outils ont été proposés, tant dans le monde académique que industriel, pour réaliser la composition de services.

Dans [YP04] les auteurs ont identifiés quatre phases décrivant le cycle de vie du processus d'une composition. Ces phases sont les suivantes :

- **Phase de définition** : elle permet de définir la composition d'une manière abstraite, c'est-à-dire indépendamment des services concrets qui seront choisis puis utilisés plus tard dans la composition.
- **Phase de planification** : cette phase identifie l'ensemble des services potentiels qui sont conformes à la description abstraite définie dans la phase précédente, et elle réalise aussi la préparation nécessaire pour leur exécution.
- **Phase de construction** : elle permet de sélectionner des services parmi ceux identifiés dans l'étape de planification. Cette phase peut aussi utiliser des mécanismes ou des langages de sélection afin de sélectionner les meilleurs services c'est-à-dire ceux qui correspondent le mieux aux besoins décrits dans la définition de la composition. La définition concrète de la composition est réalisée à ce niveau.
- **Phase d'exécution** : cette phase réalise les connexions ou les liaisons entre les services sélectionnés et exécute le service composite.

En d'autres termes, le cycle de vie du processus de composition de services consiste d'abord à définir un service composite abstrait en fonction des descriptions des services requis (par exemple les interfaces ou spécifications), ensuite à découvrir et à sélectionner les services fournissant ces interfaces parmi ceux qui sont disponibles, et enfin à réaliser les liaisons entre ces services sélectionnés. La sélection de services peut se faire à la phase de conception du composite ou à l'exécution et ainsi on parlera respectivement de sélection *statique* ou *retardée*.

TAXINOMIE.

Dans la suite de ce chapitre, nous utilisons la terminologie suivante :

Composition statique.

*Une composition est dite **statique** si les artéfacts participant à la composition sont sélectionnés avant la phase d'exécution.*

Composition retardée.

*Une composition est dite **retardée** si les artéfacts participant à la composition peuvent être sélectionnés et liés à la phase d'exécution. En général, une fois les liaisons effectuées elles restent figées.*

Composition dynamique.

*Une composition est dite **dynamique** si les artéfacts participant à la composition peuvent apparaître et/ou disparaître à tout instant pendant l'exécution. De nouveaux artéfacts peuvent être sélectionnés et/ou de nouvelles liaisons peuvent être réalisées.*

Composition autonomique.

*Une composition est dite **autonomique** si les sélections ou les liaisons (des artéfacts participant à la composition) déjà réalisées peuvent être modifiées suite à une évolution du contexte d'exécution.*

Composition incrémentale.

*Une composition est dite **incrémentale** si les artéfacts participant à la composition peuvent être déterminés et ajoutés au fur et à mesure durant tout le cycle de vie de la conception à l'exécution.*

3.2 APPROCHES DE COMPOSITION.

Il existe plusieurs approches, technologies, langages et outils de composition de services [DS05]][Pel03a]. En fonction de la manière de définir des services composites, les approches de composition peuvent être divisées en deux catégories : la composition basée sur un procédé et la composition structurelle. Cette section présente les différentes catégories de composition de services en introduisant leurs caractéristiques, leurs avantages et leurs limites.

3.2.1 Composition basée sur un procédé.

Les termes d'orchestration et de chorégraphie sont les plus utilisés pour désigner les approches de composition de services basées sur un procédé [RK06][Pel03a][Pel03b] [RK06]. Dans ces approches, 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 procédé est décrit par un ensemble d'activités et par le flot de contrôle et de données entre ces activités. Le flot de contrôle indique l'ordre d'enchaînement des invocations et d'exécution des activités du procédé. Dans le cadre de l'orchestration de services web, la réalisation de chaque activité correspond à l'invocation d'un service web qui se chargera de réaliser la tâche correspondant à l'activité associée. Les approches basées sur un procédé rendrent explicite le flot de contrôle qui devient externe par rapport aux services utilisés car il est décrit en dehors de ces services.

a. Approches et langages de composition.

Divers langages ont été conçus pour l'orchestration de services web tels que la proposition de IBM *Web Services Flow Language* (WSFL) [Ley01], la spécification de Microsoft XLANG [Tha01] et le standard de fait *Business Process Execution Language* (WS-BPEL4 ou tout simplement BPEL) [OAS07] proposé par IBM, Microsoft, BEA, SAP et Siebel. La spécification BPEL est le fruit de la combinaison des standards XLANG et WSFL, et elle sera détaillée dans le reste de cette section parce qu'elle est devenue le standard de fait reconnu pour l'orchestration de services web.

La spécification BPEL définit une grammaire conçue au dessus du langage XML[1] permettant de décrire la logique de contrôle nécessaire pour orchestrer des services web. A l'exécution, un moteur BPEL interpréte les modèles de procédés qui sont conformes à cette grammaire. Le résultat d'un procédé BPEL est un nouveau service web possédant une interface WSDL [W3C02] qui contient l'ensemble des opérations correspondants aux activités du procédé. Dans un procédé BPEL, les types de données WSDL sont utilisés pour décrire les informations pouvant circuler dans le procédé. Au niveau de la description du procédé ces données sont indiquées par les balises « *variable* ».

BPEL fournit un support pour la construction de procédés exécutables et abstraits. Un procédé est dit exécutable dans le cas où ses services concrets requis sont désignés dans sa définition. Il est dit abstrait s'il indique seulement les messages échangés entre les différents participants mais pas explicitement ces derniers [Pel03a]. Un procédé abstrait peut être utilisé par exemple pour produire plusieurs implémentations d'un service composite à partir d'une même description.

Plusieurs implémentations de qualité industrielle de la spécification BPEL sont disponibles pour le développent et la gestion de l'exécution de services composites, par exemple nous pouvons citer les outils Eclipse BPEL [Ecla], ou bien Orchestra [Orc] de Bull ou encore ActiveBPEL [Act] développé par l'entreprise *Active endpoints*. Ces outils fournissent des environnements graphiques pour faciliter la phase de définition de la composition, et ensuite ils génèrent la description XML correspondant au procédé BPEL.

D'autres approches et outils utilisant des modèles de procédés ont aussi été proposés par exemple nous avons la proposition de HP[2] eFlow [FCS00] ou bien Self-Serv [BSD03] publié à l'université *University of New South Wales*.

Dans Self-Serv, une composition est spécifiée par un diagramme d'états DSC (*Diagram State Chart* en anglais) ; à l'exécution un modèle d'orchestration basé sur un système pair à pair (*peer-to-peer*) est utilisé pour permettre le passage à l'échelle (la scalabilité) [BSD03]. Le système Self-Serv est capable de composer des services web mais aussi d'autres types de services comme Corba [Obj01] à condition que ces derniers possèdent une interface décrite en WSDL.

Dans eFflow, une composition est définie par un graphe dont les nœuds représentent les invocations de services ou les décisions à prendre, et les arcs décrivent le flot de contrôle entre les services. Le graphe proposé dans e-Flow est très similaire aux diagrammes d'activités d'UML. Le système eFlow ne compose que des services de type e-services apparus grâce à l'importante utilisation des systèmes de commerces électroniques [PS01].

b. Exemple de procédé.

La figure suivante illustre un exemple de procédé BPEL décrivant un processus de réservation de billets de spectacles avec la possibilité de restauration sur place. Dans notre exemple, le processus de réservation est composé de quatre activités nommées *RechercherSpec*, *ReserverSpec*, *Restauration et Paiement* permettant respectivement de

[1] Extensible Markup Language : http://www.w3.org/XML/

[2] Site officiel de Hewlett Packard : http://www.hp.com/

rechercher des spectacles en fonction des dates indiquées, de réserver pour ces spectacles, puis d'effectuer une réservation de table(s) au niveau du restaurant, et enfin de confirmer et payer la réservation.

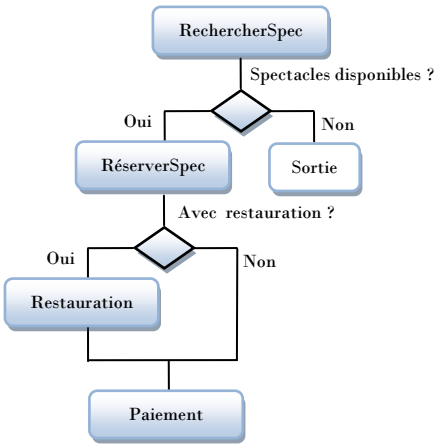

figure 10. Exemple de procédé de réservation de billets de spectacles.

c. Défauts et limites.

Les approches actuelles de composition de services, et la spécification BPEL en particulier, possèdent plusieurs limites car elles ont pris des hypothèses simplificatrices ; par exemple, tous les services composés sont des services web qui sont décrits par des interfaces WSDL. Par conséquent, ces approches n'offrent pas la possibilité d'orchestrer d'autres types de services comme ceux d'OSGi ou bien simplement des services implémentés comme des POJO (*Plain Old Java Object*), ou encore des applications patrimoines. En outre, dans BPEL, le processus de construction d'une composition est complexe et il requiert à la fois des connaissances métiers et techniques de la part du développeur (ou de l'utilisateur).

Ainsi, il est avéré que dans BPEL le niveau d'abstraction de la spécification est assez bas et que la plupart des solutions actuelles pour le développement et la gestion de services web composites sont des activités manuelles assez complexes [BOP03]. C'est pourquoi, récemment plusieurs travaux ont été réalisés dans le but d'augmenter le niveau d'abstraction du processus de construction des services composites.

La plupart de ces travaux s'appuient sur l'ingénierie dirigées par les modèles [FFR+07] afin d'augmenter le niveau d'abstraction de la composition (Cf. Section 2.5), et utilisent des techniques de génération pour obtenir la définition concrète de la composition à partir de ces modèles [PE08].

L'outil FOCAS[1] [PE08][Ped09] développé dans notre équipe de recherche fait partie des approches basées sur des modèles pour la réalisation de services composites. FOCAS utilise l'idée de la séparation de préoccupations pour spécifier un procédé. Dans FOCAS, chaque préoccupation est modélisée par un méta-modèle indépendant des autres et l'orchestration de services est réalisée par la composition de ces différents méta-modèles [PDE08]. L'outil FOCAS est extensible parce que d'autres préoccupations peuvent être ajoutées pour décrire d'autres types d'applications basés sur des procédés de manière

[1] Framework for Orchestration, Composition and Aggregation of Services

générale. L'outil est capable d'orchestrer plusieurs types de services, par exemple des services web, des services OSGi ou iPOJO, des POJOs, des services UPnP[1] [UPn08] ou DPWS[2] [Mic06a] etc.

3.2.2 Composition structurelle.

Une composition structurelle consiste à décrire la structure de l'application en identifiant ses différents composants et en précisant les connexions existantes entre ces composants. De nos jours, la spécification SCA [OSO07] et l'approche iPOJO [Esc08][EHL07] (présentées dans le précédent chapitre) fournissent des modèles à composants qui sont conformes aux principes de l'architecture orientée service et permettent d'effectuer une composition structurelle de services. La suite de cette section détaillera ces modèles de composition structurelle de services.

a. La spécification SCA.

La spécification SCA [OSO07][BII+05][Cha07] propose deux modèles pour la réalisation de composants (qui peuvent offrir des services) et la description de l'assemblage (la composition) de composants pour spécifier l'application. Le chapitre précédent (Cf. Section 2.3) a introduit le modèle de programmation pour la création de composants SCA en soulignant ses avantages et limitations. Dans cette section, nous présentons en détail le modèle de composition proposé par la spécification SCA.

La spécification SCA possède plusieurs avantages comme le support de plusieurs langages pour la réalisation de composants. La spécification SCA supporte aussi SDO[3] qui est un standard permettant de représenter et de manipuler des données indépendamment de leur structure réelle, ce qui constitue un énorme avantage pour SCA puisque facilitant la composition de services hétérogènes sans se préoccuper de la structure des données réelles.

La spécification SCA identifie plusieurs concepts tels que *service*, *implémentation*, *composant*, *propriété*, *référence*, *connecteur*, *binding* et *composite*. Dans le tableau 5 (du chapitre précédent) nous avons introduit les concepts de *service*, *implémentation*, *composant*, *propriété*, *binding* et *référence*. En plus de ces concepts, nous ajoutons dans le tableau suivant, la définition des concepts *connecteur* et *composite*.

[1] Universal Plug and Play

[2] Devices Profile for Web Services

[3] Service Data Objects : http://www.osoa.org/display/Main/Service+Data+Objects+Home

Concepts	Définitions
Service & Binding	Un *service* est la définition de fonctionnalités métiers, elle comprend, entre autre une interface Java ou WSDL composé de plusieurs opérations, et un ensemble de *bindings* c'est-à-dire les différents mécanismes d'accès à utiliser pour appeler le service.
Implémentation	Une *implémentation* est un bout de code réalisant zéro, un ou plusieurs services. Elle peut être écrite avec plusieurs langages comme Java, C++ ou BPEL.
Composant	Un *composant* est constitué au plus d'une implémentation qui est configurée dans un fichier de configuration décrit avec le langage SCDL qui basé sur XML. La configuration d'une implémentation consiste à affecter des valeurs aux propriétés éditables qu'elle définit.
Propriété	Dans SCA, les *propriétés* sont typées et elles peuvent être de type simple ou complexe, par exemple les types définis dans XML Schéma.
Référence	Les composants SCA offrent leurs fonctionnalités métiers en termes de services et ils peuvent aussi utiliser ou dépendre d'autres services. Ces dépendances sont appelées des *références*.
Connecteur	Le concept de *connecteur* ou « *wire* » permet l'établissement des liaisons entre les *références* et les *services*.
Composite	Un composite SCA est un assemblage de composants SCA qui offrent des services.

tableau 11. Les concepts de la spécification SCA.

Les concepts SCA sont décrits dans le méta-modèle de la figure 11 proposé par la communauté OASIS[1] (*Organisation for the Advancement of Structured Information Standards*) :

[1] Site officiel d'OASIS : http://www.oasis-open.org/home/index.php

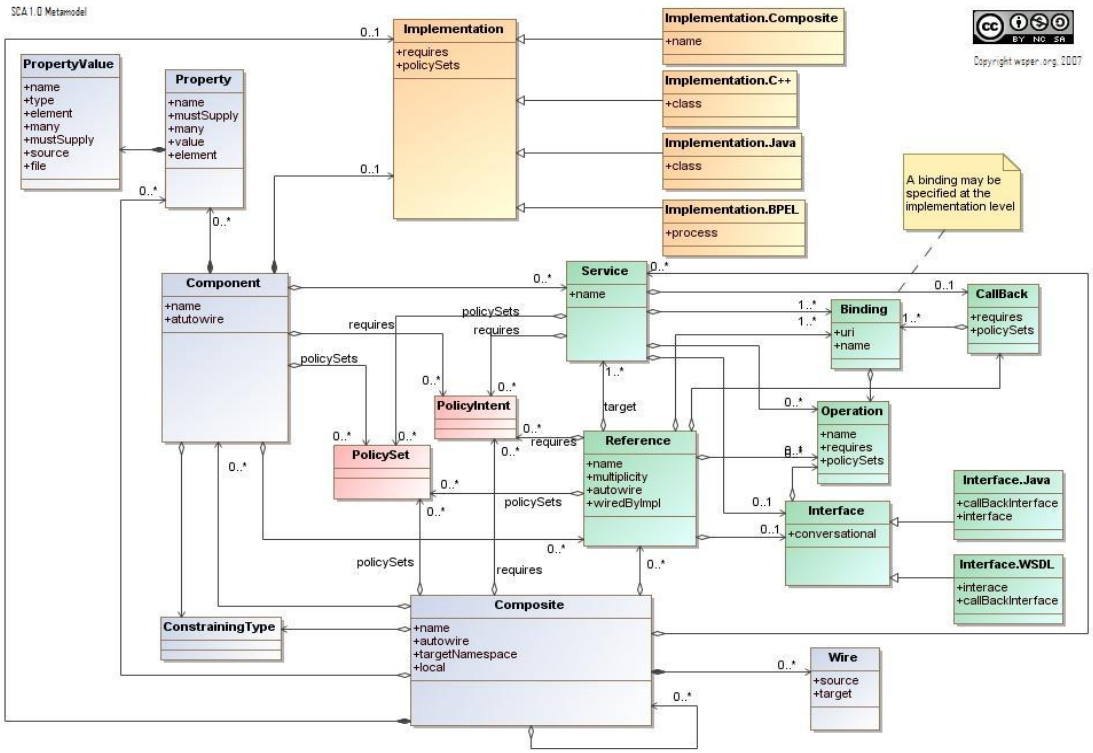

figure 11. Méta-modèle de SCA (http://www.wsper.org/sca10.jpg)

SCA décrit le contenu et les liaisons entre les composants d'une application en terme d'assemblages nommés des *composites*. Un composite SCA est un assemblage pouvant contenir des composants, des services, des références de services, des déclarations de propriétés permettant la configuration de ses composants, et des liaisons spécifiant les connexions entre les composants. Dans SCA, les composites peuvent aussi contenir d'autres composites pouvant être référencés et utilisés comme des composants SCA. Ainsi, la spécification SCA permet de construire des composites hiérarchiques.

Dans SCA, les services contenus dans un composite peuvent être accessibles à distance. Pour assurer les communications distantes entre les services, SCA offre la possibilité d'utiliser un protocole décrit dans le « *binding* » spécifié au sein du service et/ou de l'implémentation, par exemple les protocoles JMS (*Java Message Service*), RMI (*Remote Method Invocation*) ou bien SOAP (*Simple Object Access Protocol*) pour réaliser des communications synchrones ou asynchrones. La figure 12 décrit la structure d'un composite SCA :

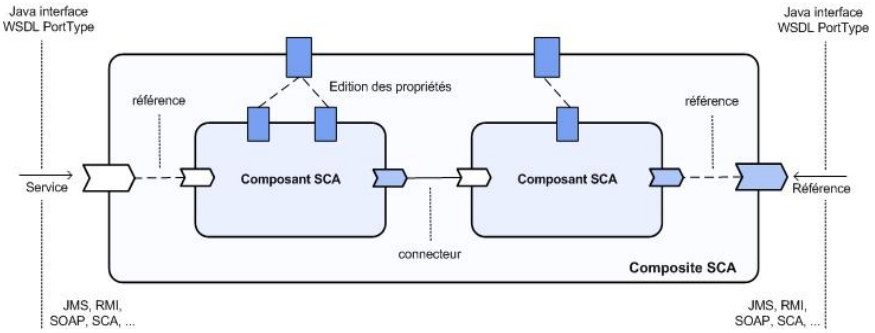

figure 12. Un exemple de composite SCA.

Dans le but de contraindre un composant (y compris un composite) et son implémentation, SCA définit le concept de *ConstrainingType* pouvant contenir des services, des références de services, des propriétés et des politiques abstraites de qualité de services. Un *ConstrainingType* permet de spécifier l'ensemble des éléments à implémenter et les contraintes à satisfaire. Ainsi, la spécification SCA permet de définir la structure des composites en précisant les composants nécessaires avant la réalisation même de leurs implémentations associées. Ceci offre la possibilité de rendre indépendants les composants de leurs implémentations, et l'avantage de cette indépendance est le fait de pouvoir construire plusieurs composites « concrets » à partir de la même spécification « abstraite » servant par exemple de « *template* ». Cette vision permet de réaliser des composites avec une vision *Top-Down* (de l'abstrait vers le concret).

Un composite SCA peut être une unité de déploiement qui sera configurée et déployée à l'intérieur d'un domaine. Un *domaine* SCA permet de définir la visibilité des composites, en regroupant par exemple les composites qui sont contrôlés et gérés par une division de l'entreprise ou bien ceux qui sont fournis par le même vendeur. Le fait de regrouper les composites par vendeur est un aspect important parce que tous les vendeurs SCA ne sont malheureusement pas compatibles. Les contributions SCA indiquent les artéfacts qui sont déployés dans chaque domaine, et ils peuvent utilisés plusieurs formats par exemple ZIP ou JAR. La figure 13 illustre un exemple de domaine SCA et les communications entre domaines :

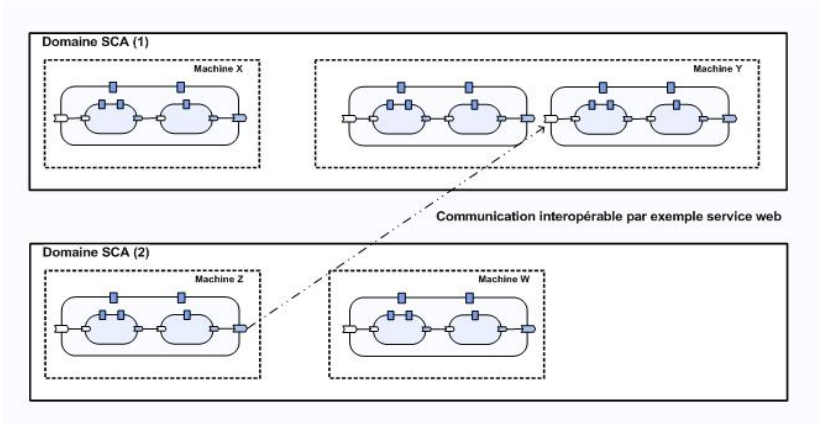

figure 13. Exemple de domaine SCA.

L'expression de besoins non fonctionnels des services, tels que la sécurité ou les transactions, est un aspect important lors de la définition d'applications à service. Une composition de services doit tenir compte de ses aspects non fonctionnels dans la mesure où ils peuvent impacter le cycle de vie de ses composants. C'est la raison pour laquelle, SCA fournit un canevas « *framework* » permettant de spécifier des contraintes et des critères de qualité de services sur une composition. La spécification des aspects non fonctionnels se fait en termes de politiques et elle peut être réalisée à plusieurs niveaux dans la composition, allant de la phase de conception au déploiement concret des composants.

Afin de rendre indépendante la définition des politiques de leurs techniques d'implémentation, SCA identifie deux concepts à savoir *PolicyIntent* et *PolicySet* permettant respectivement : (1) de définir les critères abstraits de qualité de services par exemple l'intégrité ou la confidentialité sur les composants (ou composites) indépendamment des technologies, et (2) de spécifier les politiques concrètes en indiquant les technologies utilisées. Au déploiement par exemple, le « déployeur» sera chargé de choisir les politiques concrètes

appropriées qui sont conformes aux critères abstraits définis. Le fait d'effectuer les choix technologiques au déploiement permet de mieux réutiliser les composants.

La spécification SCA ne spécifie pas la manière de mettre en œuvre tous ses concepts pour fournir une implémentation compatible à SCA. Récemment, plusieurs implémentations de SCA ont été proposées dans le monde open source ou par des vendeurs industriels. Parmi ces implémentations nous pouvons citer : *Tuscany* [Apa08] réalisée par Apache, *Newton* [New08] qui utilise la technologie OSGi pour la gestion dynamique des instances et/ou des dépendances de services, ou bien *Fabric3* [Fab08]. Chaque infrastructure d'exécution de composites SCA doit fournir un ensemble de conteneurs permettant de gérer les différents langages qu'elle supporte, par exemple les conteneurs Java, BPEL [OAS07], ou Spring [Spr] pour l'exécution des implémentations de composants écrites dans ces langages.

b. Le modèle iPOJO (injected Plain Old Java Object).

iPOJO est un modèle à composant orienté service (chaque composant peut fournir un ou plusieurs services). Il est développé au dessus de la technologie OSGi. L'un des objectifs de iPOJO est de faciliter le développement de composites en fournissant un langage de description d'architectures (ADL) (Cf. Section 2.4) permettant de réaliser une composition structurelle de services.

Le modèle iPOJO se base sur plusieurs concepts tels que *spécification de service*, *type de composant* (ou *composant*), *instance*, *propriété*, *conteneur* et *composite* (Cf. section 2.3.2). Le métamodèle iPOJO décrivant les relations entre ces concepts n'est pas explicitement défini. Pour ce faire, nous avons réalisé un métamodèle (présenté à la figure 14) illustrant quelques concepts de base de la composition de services dans le modèle iPOJO.

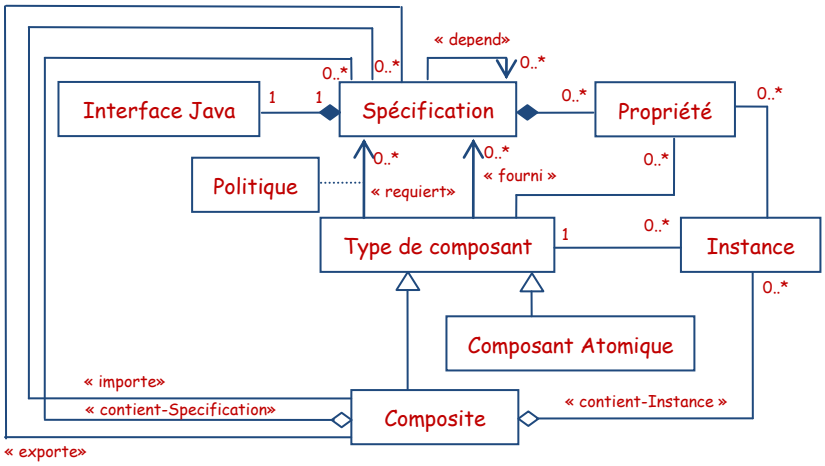

figure 14. Métamodèle de la composition dans iPOJO.

Comme indiqué dans le chapitre 2, dans le modèle iPOJO une spécification de service correspond à une description de service sous la forme d'une interface Java et un ensemble de propriétés. Dans iPOJO, une spécification de service peut dépendre d'autres spécifications. L'expression de ces dépendances entre spécifications de service permet de faciliter la composition structurelle de services et le support du dynamisme. Dans une application qui utilise un service (spécification) la substitution d'une implémentation de ce service par un autre devient ainsi possible. Dans l'approche à service classique, les dépendances entre services ne sont pas exprimées au niveau de leur spécification. Sur ce point, le modèle iPOJO se distingue des principes de l'approche à service.

Un composant iPOJO peut fournir des fonctionnalités à travers une (ou plusieurs) spécification(s) de service et il peut requérir d'autres spécifications. Une fabrique (*factory*) est associée à chaque composant iPOJO et elle permet de créer les instances du composant. Les instances d'un composant peuvent être configurées différemment. Les fabriques sont déclarées sous la forme de services et ainsi ils pourront être disponibles dynamiquement dans le registre de services, ce qui constitue un avantage lors de la création et de l'exécution d'applications dynamiques.

Dans le modèle iPOJO on distingue deux types de composants : les composants atomiques et les composites. La réalisation d'une composition structurelle se fait à travers les composites iPOJO. Un *composite* iPOJO est un composant donc il peut offrir des services et il peut dépendre d'autres spécifications de services. En plus, il peut contenir des spécifications de services et/ou des instances de composants. Les spécifications fournies ou requises par un composite peuvent être déclarées à partir des éléments qu'il contient. Généralement, ceci se fait avec les mécanismes d'export et d'import de spécifications de service.

La notion de *contexte* est un point essentiel dans iPOJO. En effet, les services fournis par les instances de composants ne sont visibles qu'à l'intérieur d'un contexte (*scope*). Ainsi, les services fournis par les instances contenues dans un composite iPOJO ne sont exploitables que par d'autres instances se trouvant dans le même composite. De cette façon, le modèle iPOJO fournit une propriété essentielle qui est l'isolation de services. Le contexte est une forme d'annuaire de services qui est privé au composite. L'approche à service ne définit pas de mécanisme d'isolation de services. Le modèle iPOJO possède cependant une autre caractéristique particulière par rapport à l'approche à service.

Le modèle iPOJO propose un format basé sur le langage XML pour décrire les services contenus dans la composition ainsi que leurs propriétés. Le résultat de la composition peut être publié comme un service. A l'exécution, le conteneur associé au composite réalisera les liaisons dynamiques entre les différents participants de la composition.

La figure 15 décrit un exemple de composite iPOJO :

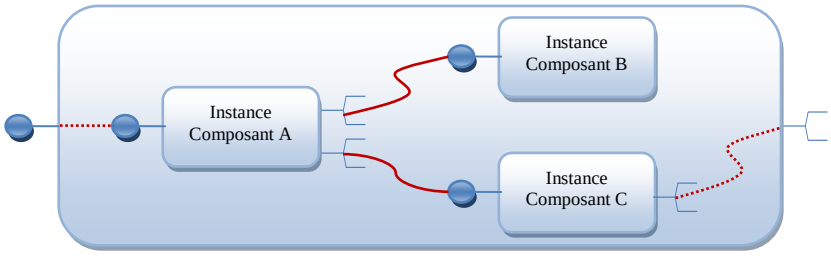

figure 15. Exemple de composite iPOJO.

c. Défauts et limites des modèles SCA et iPOJO.

Les modèles de composition proposés par la spécification SCA et iPOJO possèdent plusieurs défauts et limitations. Le modèle SCA est limité par exemple sur ces aspects :

- **Interopérabilité** : la spécification SCA propose un ensemble de concepts pour faciliter la réalisation et l'assemblage de composants. Mais elle ne définit pas la façon de mettre en œuvre ces concepts. Il existe plusieurs implémentations mettant en place SCA. Malheureusement, en pratique, toutes les implémentations de SCA n'interopèrent pas facilement. En général, chaque

implémentation couvre quelques langages de programmation pour la réalisation de composants SCA et fournit peu de protocoles de communication pour leur interaction.

Le concept de domaine proposé par SCA permet de résoudre partiellement le problème d'interopérabilité entre les fournisseurs de technologies SCA parce qu'il permet d'isoler les éléments SCA par exemple par fournisseur et de supporter des communications distantes et interopérables. Chaque vendeur fournissant une plateforme compatible à la spécification SCA définit (ou réutilise) un langage pour la description de services (fournis par les composants SCA). L'idéal serait de disposer d'un langage de description commun pour toutes les plates-formes SCA.

- **Liaisons statiques** : dans la description de composite SCA, les liaisons entre les composants contenus sont généralement définies explicitement et statiquement c'est-à-dire qu'elles sont spécifiées à la phase de développement ou de déploiement et elles ne changent plus à l'exécution (composition statique). La spécification SCA permet de ne pas spécifier dans la description de l'application les liaisons entre les composants. L'information est généralement déduite de la description de composants (« autowire »). Les technologies SCA offrant cette possibilité de non définition explicite de liaison n'offrent pas une vision globale et précise de l'architecture de l'application. La spécification SCA présente une vision surtout statique du composite parce que sa structure ainsi que les protocoles de communication (« binding ») une fois choisis restent figés, ce qui constitue un inconvénient pour une reconfiguration ultérieure de l'application.
- **Dynamisme** : la spécification SCA ne spécifie pas de mécanisme pour supporter la disponibilité dynamique de composants. Ainsi, elle ne facilite pas la description et l'exécution de composites où certains composants peuvent être disponibles ou pas, ou bien être substitués y compris à l'exécution. La spécification SCA ne décrit pas l'infrastructure nécessaire pour créer et exécuter des applications dynamiques. Chaque plateforme SCA est libre de réaliser ou pas la gestion de la disponibilité dynamique et il y a un risque d'incompatibilité entre les plates-formes. Certaines plates-formes comme *Newton* ou *FraSCati* offre la possibilité de changer les liaisons entre composants SCA ou bien de substituer ces derniers.
- **Aspects non fonctionnels** : SCA permet de spécifier un ensemble de besoins non fonctionnels d'un service en termes de politiques lors de la définition d'applications. Elle fournit ainsi un canevas pour l'expression de ces politiques. Mais les politiques sont exprimées au même niveau que la description de la composition. L'idéal serait d'effectuer une séparation claire entre la définition de la composition et l'expression des politiques associées aux services.

Le modèle de composition iPOJO est limité quant à lui par exemple sur les points suivants :

- **Interopérabilité** : Le modèle iPOJO ne gère pas la composition de services hétérogènes qui peuvent être spécifiés avec différents langages de description (et de réalisation) ou bien s'exécutant sur diverses plates-formes à services. L'approche iPOJO ne règle pas ainsi la problématique d'hétérogénéité des applications.

- **Environnements de composition** : le modèle iPOJO propose un langage pour la description de la composition et une infrastructure d'exécution de services atomiques ou composites. Il fournit un environnement sous la forme de plug in Eclipse pour assister les utilisateurs dans l'étape de développement de services. Mais cet environnement est minimal et ne fournit pas de supports permettant à des utilisateurs non experts à la technologie iPOJO de réaliser facilement leurs applications. En d'autres termes, seuls les experts iPOJO sont capables d'utiliser « facilement » l'environnement actuel.
- **Découverte et sélection de services** : dans le modèle iPOJO une spécification de service correspond à son interface (le nom) et un ensemble de propriétés. La spécification de service n'est pas assez riche par rapport aux informations contenues. De plus, le modèle iPOJO ne permet pas aux utilisateurs d'exprimer facilement à travers un langage les propriétés (fonctionnelles et non fonctionnelles) désirées sur les services devant participer à la composition. Ainsi, le modèle iPOJO manque d'outils pour spécifier les besoins fonctionnels et non fonctionnels, découvrir et sélectionner automatiquement les services requis à la composition en fonction de la description des besoins.
- **Typage des propriétés et des contraintes** : le modèle iPOJO, comme la plateforme OSGi, utilise des expressions LDAP pour la définition de contraintes. Les expressions LDAP ne sont pas typées donc à partir d'une expression la déduction du type de service (ou des services) concerné(s) n'est pas fournie.

Les modèles SCA et iPOJO ne garantissent pas la cohérence de la composition en vérifiant et validant les propriétés et contraintes des services qui participent dans la composition.

3.3 ENVIRONNEMENTS ET OUTILS DE COMPOSITION.

3.3.1 Motivations.

De nombreux acteurs peuvent intervenir dans le processus de réalisation d'une application logicielle allant de sa spécification à son exécution. Dans le cas d'une application à base de services, les acteurs peuvent, par exemple spécifier des services, réaliser leurs implémentations, composer ou déployer des services, etc.

Les différents acteurs effectuent un ensemble de tâches particulières pouvant être assez complexes. Ils ont besoins d'assistance et de supports pour automatiser la réalisation de leurs tâches. Un environnement logiciel doit offrir une aide pour simplifier le travail des acteurs participant à la réalisation de l'application. Ainsi, un environnement de composition de services permet de faciliter l'intégration de services en fournissant un ensemble d'outils spécialisés pour l'automatisation du processus de composition.

Un environnement de composition est généralement constitué d'un ensemble d'outils logiciels comme :

- **Un éditeur** : permettant d'assister les développeurs dans leur(s) tâche(s) de composition à travers l'usage d'un éditeur pour décrire le service composite. L'assistance peut être de divers types, par exemple, l'éditeur peut fournir des supports de coloration syntaxiques des mots clés du langage de composition associé ou de complétion automatique. Il peut être sous forme textuelle ou graphique. Les éditeurs graphiques permettent la spécification de la

composition à partir d'éléments visuels et ils offrent la capacité de génération automatique de code pour cacher la complexité technique liée à l'écriture de ce dernier.

- **Un validateur ou compilateur** : permettant de valider les erreurs lors de l'écriture du code décrivant le modèle de composition. Dans le cas d'un langage compilé, l'environnement de composition peut disposer d'un compilateur permettant de compiler et détecter ainsi les erreurs de compilation.
- **Un débogueur** : permettant le déboguage des erreurs sur les modèles de composition.
- **Un interpréteur** : permettant d'interpréter les modèles décrivant la composition pour pouvoir les exécuter.
- **Un support d'exécution** : capable d'exécuter les modèles décrivant la composition.

Les environnements de composition sont en général spécifiques à un langage de composition particulier ou à une technologie particulière par exemple les services web. Ils fournissent un sous-ensemble des outils cités précédemment et ils peuvent être de deux types : les environnements de modélisation (spécification) de composition et ceux d'exécution de services composites.

3.3.2 Quelques environnements et outils de composition.

Il existe de nombreux environnements et outils de qualité industrielle pour la composition de services en général et pour l'orchestration en particulier. Ces environnements permettent de spécifier et gérer l'exécution de procédés ou de services composites suivant le type de composition associé (orchestration ou composition structurelle). Nous pouvons citer les outils issus de l'industrie ou du monde académique comme Eclipse BPEL [Ecla], Orchestra [Orc], FOCAS [PDE09], DoCoSOC[1] [Mar08], Self-Serv [BSD03] ou les outils intégrés dans Eclipse pour les applications à service (SOA Tools Plateform) [Eclb].

Eclipse BPEL

Eclipse BPEL est constitué d'un ensemble d'outils permettant l'édition, le déploiement, le test et le déboguage de procédés BPEL sous l'environnement de développement intégré Eclipse. Les outils intégrés dans Eclipse BPEL sont les suivants :

- **Editeur de modèles** : Eclipse BPEL fournit un éditeur graphique pour la spécification de procédés BPEL. Cet éditeur est basé sur le canevas GEF[2] et offre une interface graphique sophistiquée pour faciliter la réalisation de procédés BPEL. Il permet la génération automatique de la description XML de procédés dans la syntaxe proposée par la spécification BPEL.
- **Validateur de modèles** : le validateur de modèles intégré dans Eclipse BPEL analyse un modèle de procédé pour signaler les erreurs et « warnings » sur ce dernier.

[1] Domain Configurable Service Oriented Computing

[2] Graphical Editing Framework

- **Infrastructure d'exécution** : Eclipse BPEL intègre aussi une infrastructure pour le déploiement et l'exécution de modèles de procédés BPEL à l'intérieur d'un moteur BPEL.
- **Débogueur** : il fournit un support de déboguage pour offrir aux développeurs de procédés BPEL une option permettant de déboguer l'exécution de leurs procédés.

L'environnement Eclipse BPEL est extensible c'est-à-dire il supporte l'ajout de nouvelles fonctionnalités et/ou de moteurs BPEL par exemple pour spécialiser l'éditeur associé ou bien étendre l'infrastructure d'exécution de procédés.

Orchestra

Orchestra est un moteur d'orchestration de services web fourni par Bull et qui est conforme à la spécification BPEL. Il permet de spécifier et exécuter des procédés métiers. Le moteur Orchestra est réalisé sous la forme d'une application J2EE et profite donc des avantages comme la gestion des transactions et de la persistance offerts par les serveurs d'applications en général et JOnAS (Java Open Application Server) en particulier [Obj07].

Orchestra fournit un ensemble de fonctionnalités à travers des outils pour la spécification et l'exécution de procédés BPEL :

- **Editeur et validateur de modèles** : Orchestra fournit un éditeur spécialisé permettant de spécifier des modèles de procédés métiers et cet éditeur peut être intégré dans l'environnement Eclipse.
- **Console d'administration** : Orchestra fournit une console d'administration pour la gestion de l'exécution de procédés métiers qui sont conformes à la spécification BPEL.
- **Outil d'exécution** : Orchestra intègre un moteur capable d'exécuter les modèles de procédés. L'exécution de procédés peut se réaliser avec ou sans la gestion de la persistance. En plus, Orchestra offre un mécanisme de reprise sur panne lors de l'exécution de procédés, ce qui constitue un grand avantage pour la gestion de l'exécution des procédés métiers.

FOCAS (Framework for Orchestration, Composition and Aggregation of Services)

FOCAS [PDE09] est un canevas de composition de services qui se base sur l'approche d'orchestration. Il fournit un environnement d'orchestration et d'exécution de services sous Eclipse [Eclc]. FOCAS offre la capacité d'orchestrer des services web, OSGi ou iPOJO et propose une approche d'orchestration de services qui est composée de deux étapes :

- **Définition abstraite de l'orchestration** : cette étape permet de spécifier d'une manière abstraite l'orchestration de services en utilisant les différents modèles identifiés dans FOCAS pour exprimer la logique métier de l'application. L'orchestration est définie ainsi en termes de spécifications de service.
- **Réalisation concrète de l'orchestration** : cette étape réalise l'orchestration concrète des services en faisant correspondre à chaque spécification de service un service concret qui est disponible. FOCAS facilite la génération de l'orchestration concrète en fournissant les outils présentés ci-dessous.

FOCAS fournit un environnement de développement et d'exécution d'orchestration de services qui est composé d'outils comme :

- **Editeur de modèles** : FOCAS intègre un ensemble d'éditeurs qui sont spécifiques à chaque modèle : contrôle, données et services. Par exemple, FOCAS intègre in éditeur graphique basé sur le canevas GEF pour la définition de modèle de procédés.
- **Outils de composition de modèles** : l'orchestration de services est spécifiée avec plusieurs modèles qui sont indépendants et nécessaires. FOCAS intègre un ensemble d'outils spécialisés pour faciliter la composition de ces modèles.
- **Générateur de code de l'orchestration** : FOCAS assiste les utilisateurs dans leurs tâches en offrant des capacités de génération automatique et partielle du code de l'orchestration.
- **Moteur d'exécution** : FOCAS intègre un moteur d'exécution de l'orchestration concrète de services.

L'environnement est extensible c'est-à-dire des aspects fonctionnels (nouveaux modèles et fonctionnalités) ou non fonctionnels (par exemple la sécurité [CL08] ou les transactions) peuvent être ajoutés.

DoCoSoC (Domain Configurable Service Oriented Computing)

[Mar08] propose une approche qui fournit un environnement de composition de services appelé DoCoSOC. L'objectif est d'automatiser la composition de services à partir d'une spécification abstraite. Cette spécification abstraite prend en compte les informations qui sont relatives au domaine métier et spécifiques à l'application. Les utilisateurs définissent leurs applications en termes de concepts du domaine, ce qui permet de développer « rapidement » des applications sans disposer de connaissances techniques. Pour ce faire, DoCoSOC propose une approche de composition composée de deux étapes :

- **Définition abstraite de la composition** : cette étape permet de spécifier d'une manière abstraite la composition exprimant la logique métier de l'application. La composition abstraite est définie avec les concepts du domaine applicatif et elle permet aux utilisateurs non experts de spécifier leurs applications parce que la spécification n'est pas liée aux aspects spécifiques d'une technologie à service particulière. Elle est exprimée en termes de spécifications de service et d'un ensemble de connecteurs décrivant leurs interactions.
- **Réalisation concrète de la composition** : cette étape effectue la composition concrète de l'application. Elle se fait à partir d'une génération de la composition vers une technologie à service spécifique en fonction des services qui sont disponibles dans un registre. La phase de génération est automatisée dans DoCoSOC et elle s'effectue d'une manière « transparente » (l'utilisateur prend l'initiative de générer la composition).

DoCoSOC est une approche orientée ligne de produits et qui se base sur l'approche à service. L'environnement de développement fourni se base sur l'ingénierie dirigée par les modèles et le canevas EMF[1] [BSM+03] pour la génération de l'éditeur de modèles de

[1] Eclipse Modeling Framework

composition à partir du métamodèle (spécifiant l'architecture à service) du domaine. Chaque éditeur généré est donc spécifique à un domaine applicatif particulier. En résumé, l'environnement DoCoSOC fournit des outils comme :

- **Editeur de modèles** : spécifique à un domaine particulier et qui est généré à partir du métamodèle du domaine.
- **Générateur de code de la composition** : génération du code de la composition pour une technologie spécifique à service comme OSGi ou les services web.
- **Plateforme d'exécution** : basée sur la technologie OSGi pour l'exécution d'applications à service.

Self-Serv (composing web accessible information and business services)

L'environnement Self-Serv proposé dans [BSD03] permet la composition de services web en utilisant un modèle d'orchestration qui est basé sur un système pair à pair. La composition est spécifiée sous la forme d'un diagramme d'états. L'environnement fournit des outils comme :

- **Editeur des services composites** : permettant de spécifier la composition de services à partir d'un ensemble de services qui sont disponibles dans un registre.
- **« Déployeur » de services** : un outil de déploiement de services dans l'infrastructure d'exécution fournie par l'environnement Self-Serv. L'outil génère et déploie les tables de routage pour tous les états du composite décrit par le diagramme d'états.
- **Système de découverte de services** : cet outil s'appuie sur les standards proposés par la technologie des services web. Dans l'implémentation actuelle, le registre de services est un registre UDDI [OAS04] qui est centralisé. L'outil permet de découvrir et sélectionner automatiquement les services requis à partir des politiques spécifiées.
- **Infrastructure d'exécution de services** : capable d'exécuter la composition de services.

Eclipse SOA Tools Platform (STP)

STP [Eclb] est un projet Eclipse dont la mission est de fournir un ensemble de canevas et d'outils pour la conception, la configuration, l'assemblage (la composition), le déploiement et la gestion d'applications à base de services. Le projet STP utilise la spécification SCA comme modèle à service de référence et il fournit des outils pour spécifier et gérer les composites SCA. Il intègre les outils suivants :

- **Editeur et validateur de modèles** : STP/SCA fournit un éditeur graphique basé sur le canevas GMF[1] pour la spécification de composites SCA. Il fournit aussi un éditeur pour la visualisation des modèles XML décrivant les composites SCA et intègre un validateur de modèles permettant de signaler des erreurs et « warnings » de modélisation. Les éditeurs sont fournis dans l'environnement de développement intégré Eclipse [Eclc].

[1] Graphical Modeling Framework

- **Générateur de code** : STP/SCA contient un générateur de code Java à partir de la description de composites SCA.
- **Support d'exécution** : STP/SCA fournit une infrastructure pour exécuter et débuguer sous l'environnement Eclipse des projets Java correspondant à des composites SCA.

3.3.3 Synthèse et limitations.

Nous avons présenté quelques environnements et outils dédiés à la composition de services. Ces outils possèdent un ensemble de concepts (composants) et mécanismes communs. La figure suivante décrit le principe de fonctionnement des outils de compositions de services en introduisant les liens entre les éléments qui leur composent :

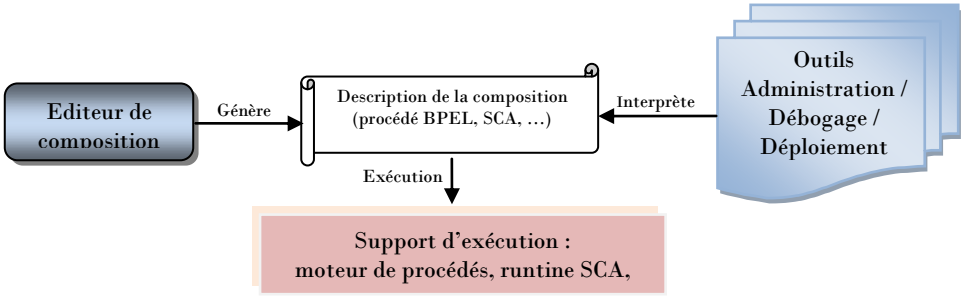

figure 16. Composants des outils de composition de services.

Le tableau 12 synthétise les caractéristiques des environnements et outils de composition étudiés précédemment :

	Editeur & validateur de modèles	*Déploiement, Débogage & Administration*	*Support d'exécution*	*Extensibilité*
Eclipse BPEL	Editeur graphique de modèles et un validateur détectant les erreurs et « warnings » sur les modèles.	Support de déploiement, de tests et de débogage de procédés BPEL.	Oui.	L'environnement est extensible : ajout de nouvelles fonctionnalités, moteurs …
Orchestra	Editeur graphique de modèles de procédés BPEL avec un validateur.	Console d'administration avec la gestion de la reprise en panne de l'exécution.	Oui.	Non spécifiée.
FOCAS	Editeur graphique de modèles de procédés ; d'autres éditeurs pour les aspects liés à la composition.	Ne fournit pas de support de déploiement, de tests et de débogage de l'exécution de procédés.	Oui.	Environnement extensible : ajout d'aspects non fonctionnels, de nouvelles fonctionnalités …
DoCoSOC	Editeur de modèle de domaine. Assez simple.	Ne fournit pas de support de déploiement, de tests et de débogage de l'exécution de procédés.	Oui.	Environnement extensible parce l'approche est dirigée par les lignes de produits. **Inconvénient** : il faut régénérer les éditeurs de modèles de domaine à chaque fois.
Self-Serv	Editeur de services composites	Infrastructure de découverte, sélection et de déploiement de services web.	Oui.	Non spécifiée.
STP/SCA	Editeur graphique de modèles de composites. Editeur de politiques conforme à la spécification WS-Policy pour exprimer la sécurité par exemple sur la composition.	Infrastructure de déploiement.	Oui.	Non spécifiée.

tableau 12. Synthèse sur quelques outils de composition.

Les environnements et outils actuels de composition de services possèdent de nombreuses limitations. Certains d'entre eux héritent des défauts et limitations de la spécification BPEL par exemple en ne composant que des services web.

Une application à services possède un cycle de vie composé de phases comme la spécification, la conception, la composition, le déploiement et l'exécution de services. Dans chaque phase, beaucoup d'acteurs peuvent participer et manipuler un ensemble de concepts qui sont relatifs aux activités à accomplir. Les environnements et outils étudiés offrent la capacité de modéliser, déployer, exécuter et administrer les applications (qui sont exprimées sous la forme de composition de services). Mais, dans ces outils, la transition entre les différentes phases du cycle de vie de l'application n'est pas claire et les tâches ne sont pas bien séparées.

Un environnement devrait offrir un ensemble de vues spécifiques aux types d'acteurs, ce qui permet à ces derniers de ne voir et manipuler que les concepts et fonctionnalités pertinents par rapport à leurs tâches. Ces vues doivent contenir des concepts de haut niveau (abstraits) pour permettre aux utilisateurs de spécifier la composition indépendamment des détails techniques d'implémentation. Les environnements de composition sont en général spécifiques à une technologie à service. La composition de services devrait être ainsi automatique. Pour ce faire, l'environnement doit inclure des outils permettant de rechercher et sélectionner des services à partir d'un ensemble de critères (fonctionnels et/ou non fonctionnels) et des moyens (par exemple des langages) pour la spécification de ces critères de sélection. La plupart des environnements étudiés ne disposent pas de tels outils, ce qui constitue une limitation majeure de ces outils pour automatiser la composition.

La majorité des environnements d'exécution de services ne gère pas le dynamisme c'est-à-dire la disponibilité dynamique des services participant à la composition. En plus, une fois les connecteurs (entre services) spécifiés dans l'environnement de modélisation (ou au déploiement) ils ne changent plus à l'exécution. Tout cela constitue un handicap pour la gestion d'applications dynamique à services.

3.4 SYNTHESE.

Nous avons présenté deux types de composition de services : la composition basée sur un procédé qui est généralement appelée orchestration et la composition structurelle de services. Ces deux types de composition de services sont les plus utilisés ; ils sont complémentaires et possèdent plusieurs avantages :

- L'orchestration de services permet de rendre explicite la logique du flot de contrôle et de données de l'application : les interactions entre les services, les messages (données) échangés et l'ordre d'invocation des services ;
- Le flot de contrôle et de données est externe aux services participant à la composition ;
- La composition structurelle offre une vision de la structure de l'application : les éléments qui la compose et l'ensemble des connecteurs entre eux. L'orchestration de services est plutôt orientée vers une vision comportementale de l'application.
- La composition structurelle offre la plupart du temps la capacité de composition hiérarchique (les services utilisés dans la composition peuvent être des services composites).

C'est pourquoi, une approche de composition peut combiner à la fois les deux types de composition. Dans ce cas, une composition structurelle sera utilisée généralement pour la description de la structure de l'application et ensuite avant la phase d'exécution de l'application les outils proposés peuvent générer un procédé métier, par exemple au format du langage BPEL, qui sera exécuté par un moteur de procédés. Etant donnée qu'il existe déjà des outils libres ou industriels fournissant des supports (moteurs) d'exécution conformes à la spécification BPEL, la génération de procédés BPEL permet de réutiliser ces outils et de profiter de leurs avantages.

Il existe également d'autres types de composition comme :

- **Composition sémantique ou par but (goal)** : consiste à sélectionner, combiner et exécuter un ensemble de services, généralement des services web, pour satisfaire un objectif. Les utilisateurs humains réalisent souvent des tâches manuelles pour essayer d'atteindre leurs objectifs en composant par exemple des services web qui sont disponibles. Le web sémantique propose d'ajouter des informations liées à la sémantique des services web dans leurs descriptions WSDL (décrivant uniquement les informations syntaxiques). L'ajout d'informations sémantiques dans la description de services permet de faciliter la découverte et sélection automatique de services, par des utilisateurs non humains comme un programme ou un ordinateur. La composition sémantique se base sur le concept d'ontologie qui spécifie l'ensemble des termes et concepts (les informations) relatifs à un domaine particulier : le modèle du domaine métier. Il existe plusieurs travaux proposés ou qui sont en cours sur la composition sémantique de services web [SAS03] basée sur des langages ou ontologies comme OWL-S[1] [W3C04a] ou WSMO[2] [RKL+05]. Mais, la composition sémantique n'est malheureusement pas encore trop répandue et utilisée.
- **Composition basée sur la qualité de service ou des règles de contraintes** : d'autres approches ou canevas comme StarWSCoP [HSZ03][ZT07][BOP03][NCF05] [MDSR07][AAC07][Cud05] proposent de composer des services en se basant sur un ensemble de contraintes d'une manière générale, et particulièrement la qualité de service (comme le temps de réponse ou d'exécution). Ces contraintes correspondent aux critères de sélection des services requis dans la composition.

Dans les sections précédentes, nous avons souligné le manque d'outils de modélisation et d'exécution de services composites permettant de simplifier réellement la tâche des utilisateurs. Les environnements actuels de composition de services manquent généralement de langages formels permettant d'exprimer les besoins applicatifs (fonctionnels ou non), d'outils facilitant la spécification de ces besoins et qui automatisent la phase de sélection des services requis dans la composition et qui répondent aux besoins spécifiés.

Les tableaux suivants (tableau 13 et tableau 14) synthétisent les caractéristiques de quelques modèles, langages ou outils de composition de services en fonction d'un ensemble de critères comme le type de composition ou les types de services composés :

[1] Web Ontology Language for Web Services

[2] Web Service Modeling Ontology

LANGAGES OU MODELES	TYPES DE COMPOSITION	COMPOSITION HIERARCHIQUE EXPLICITE	AUTOMATIQUE	SELECTION DYNAMIQUE	LIAISON RETARDEE / DYNAMIQUE	COMPOSITION ABSTRAITE	TYPES DE SERVICES COMPOSES	RESULTAT
BPEL	Procédé	OUI	Non	Non spécifié	Non spécifié	OUI avec les « portTypes » des interfaces WSDL	Services web	Service web
SCA	Structurelle	OUI	Non	Non spécifié (Extensions)	OUI avec le concept de « autowires »		Services web, Java, C++, composite, …	Le résultat n'est pas forcément un service
iPOJO	Structurelle	OUI	Non	OUI	OUI	OUI	iPOJO et OSGi	Le résultat n'est pas forcément un service
WSFL	Procédé	NON	Non	NON	NON		Services web	
XLANG	Procédé	NON	Non	NON		Service web		

tableau 13. Positionnement de quelques langages et modèles de composition.

CANEVAS / OUTILS	TYPES DE COMPOSITION	COMPOSITION HIERARCHIQUE EXPLICITE	AUTOMATIQUE / DYNAMIQUE	SELECTION DYNAMIQUE	LIAISON RETARDEE / DYNAMIQUE	COMPOSITION ABSTRAITE	TYPES DE SERVICES COMPOSES	RESULTAT
FOCAS	Procédé	NON	Manuelle	OUI usage de la machine SAM. (Cf. Section 5.1**Erreur ! Source du renvoi introuvable.**)	OUI	OUI	Services web, iPOJO, OSGi, Java …	Le résultat n'est pas un service
DoCoSOC	Structurelle		Manuelle	OUI	OUI	OUI	OSGi, iPOJO	
Self-Serv	Composition déclarative : orchestration P2P	Oui WSDL.		Environnements dynamiques : Sélection par attributs	OUI Les conteneurs sont des services		Services web	
eFlow (HP)	Procédé	OUI	changements adaptations dynamiques.	OUI. Découverte dynamique. Découplage sélection / procédé. Pas de sélection.	OUI La liaison ne peut pas être modifiée		E-services	Le résultat est un service.
StarWSCoP (SUN)	Composition basée sur la qualité de service	NON	Manuelle	OUI Découverte		OUI PortType de s WSDL	Services web	Service web

tableau 14. Positionnement de quelques outils de composition.

BILAN DE L'ETAT DE L'ART ET DE LA PRATIQUE.

La réalisation d'une composition de services doit être simple, rapide et à moindre coût. Actuellement, elle est une tâche manuelle assez complexe parce qu'elle se fait généralement avec un niveau d'abstraction bas. Il faudrait automatiser le processus de construction d'un service composite pour simplifier et faciliter sa réalisation et son exécution. Ainsi, nous ressentons le besoin de réduire la complexité du processus de composition en fournissant des langages, modèles et outils ayant un haut niveau d'abstraction. [MM04] identifie les besoins qui doivent être satisfaits par une composition de

services web : la connectivité des services, la qualité de services, le passage à l'échelle et la cohérence de la composition. La propriété de connectivité des services est assurée par toutes les approches de composition de services ; par contre les autres propriétés ne sont pas toujours garanties. En plus de ces propriétés, une composition de services devrait garantir un ensemble d'aspects non fonctionnels tel que la sécurité (pas seulement la qualité de services), exprimer et gérer le dynamisme et la substitution des services participant dans une composition.

La définition de composites respectant un ensemble de critères fonctionnels et/ou non fonctionnels telle que la qualité de services est un aspect important pour la réalisation d'applications à service. C'est pourquoi dans la littérature, des approches de composition de services basées sur des règles de contraintes [BOP03] [HWH08], la qualité de services [MDSR07] [JM07][YMBM08] ou le web sémantique [AAS02] ont été proposées. En général, ces approches se fondent sur les catégories de composition que nous avions identifiées précédemment. En outre, les approches de composition sémantique utilisent par exemple le langage OWL-S [W3C04a] (au lieu de WSDL pour la description de services) pour fournir des supports de découverte de services qui sont basés sur leurs aspects fonctionnels.

La plupart des approches actuelles de composition ne disposent pas de mécanismes ou de langages permettant d'effectuer la sélection des services requis par une composition donnée. Généralement, la sélection de services est réalisée manuellement et statiquement à la conception d'un service composite et une fois que le choix des services est effectué il reste inchangé. Pour offrir plus de flexibilité au niveau des choix, la composition doit disposer de langages de sélection pour exprimer ses besoins et les critères de sélection, et aussi des algorithmes permettant de sélectionner les services d'un composite. La sélection de services devrait être possible lors des phases de conception, de déploiement ou d'exécution des composites, et l'infrastructure d'exécution devrait assurer les liaisons dynamiques entre ces services à l'exécution.

L'automatisation du processus de réalisation d'une composition permet le développement rapide de composites en facilitant l'interaction avec le développeur. Idéalement, le développeur devrait fournir une description (un but) pour exprimer les besoins et les contraintes de la composition. Ensuite, un moteur va évaluer cette description, sélectionner les services requis pour fournir d'une manière « transparente » un composite cohérent par rapport au but spécifié. La composition dynamique fait parti des défis de la composition de services et implique de disposer à l'exécution des mécanismes de sélection permettant de sélectionner les services nécessaires. Certaines approches comme [NCF05] réduisent la composition retardée de services à un problème de satisfaction de contraintes.

La définition de ***composites totalement « abstraits »*** c'est-à-dire qui sont exprimés seulement en termes des descriptions des services potentiels facilite la réutilisation des spécifications de composites pour la construction de différents ***composites « concrets ».*** En plus, la possibilité d'ajouter graduellement des informations sur la description d'un composite (***composite incrémental***) peut permettre de gérer tout le cycle de vie d'une application allant de sa conception à son exécution en passant par son déploiement concret. Plusieurs catégories d'informations peuvent être ajoutées par plusieurs acteurs en fonction de leurs tâches et usages, par exemple les informations concernant la distribution des composants de l'application ou bien celles qui sont liées d'une manière générale à leur déploiement.

Deuxième partie :
Contributions.

4

SELECTA : UNE APPROCHE DE COMPOSITION.

Le chapitre 3 a mis en évidence les avantages et les limitations des approches et environnements actuels dédiés à la composition de services. Nous avons identifié également quelques besoins et défis à surmonter afin d'améliorer le processus de la réalisation de la composition. Dans cette thèse, nous proposons une approche de composition, nommée SELECTA, répondant à un ensemble de besoins et défis tels que la sélection automatique des éléments (services ou composants) requis par un composite ou la construction de composites flexibles. Dans ce chapitre, nous présentons les objectifs, les principes et les mécanismes de base de notre approche. Nous introduisons ensuite un langage d'expression de contraintes et un algorithme générique de sélection afin d'automatiser le processus de la composition. Nous concluons ce chapitre par une synthèse.

4.1 OBJECTIFS ET BESOINS.

4.1.1 Objectifs.

Le processus de la réalisation d'une application logicielle possède un cycle de vie assez complexe qui est constitué de plusieurs phases telles que :

- l'analyse des besoins du logiciel,
- la spécification du logiciel,
- la conception et le développement des briques de base du logiciel,
- le test et le déploiement de ces briques,
- l'assemblage des briques constituant le logiciel,
- l'exécution du logiciel,
- l'administration (« monitoring ») de l'exécution du logiciel, etc.

Dans chaque phase du cycle de vie d'une application, nous trouvons divers acteurs manipulant un ensemble de concepts afin de réaliser la (ou les) tâche(s) qui sont liée(s) aux activités de la phase. Ces acteurs peuvent être les mêmes ou distincts durant tout le cycle de vie de l'application et ils peuvent manipuler des concepts pouvant être très différents en fonction du niveau d'abstraction de la phase et de leurs connaissances métiers.

Notre principal objectif est de faciliter la réalisation d'applications logicielles en couvrant et en assurant leur cycle de vie, allant de la conception à l'exécution des applications. Pour ce faire, nous souhaitons identifier les liens existants entre les différentes étapes du cycle de vie d'une application dans le but de faciliter les transitions entre ces étapes et d'optimiser le coût de réalisation des applications. Ainsi, nous avons l'ambition d'utiliser le concept de composite comme élément pivot et central du cycle de vie des applications.

Notre défi.

Utiliser le concept de composite pour *couvrir et gérer le cycle de vie des applications logicielles : conception, développement, composition, déploiement, exécution,…, monitoring, évolution, autonomique …*

Dans cette thèse, nous avons comme objectif de définir avec un niveau d'abstraction assez élevé, les concepts de composite et d'application. Notre intention est de fournir des environnements logiciels spécialisés en fonction des activités associées à chaque phase du cycle de vie dans le but de faciliter le développement d'applications de la conception à l'exécution.

Les applications à base de services logiciels possèdent des caractéristiques particulières : les services utilisés sont dynamiques dans le sens où ils peuvent apparaître ou disparaître à tout instant, ils peuvent être partagés dans divers contextes par plusieurs applications et ils peuvent être distribués à un travers un réseau, par exemple Internet ou Intranet. C'est la raison pour laquelle, nous souhaitons offrir la capacité d'exprimer tous ces aspects au niveau de la spécification de composites et nous souhaitons offrir des supports pour automatiser certaines tâches liées à la réalisation et à l'exécution des applications à service.

La figure suivante illustre notre vison :

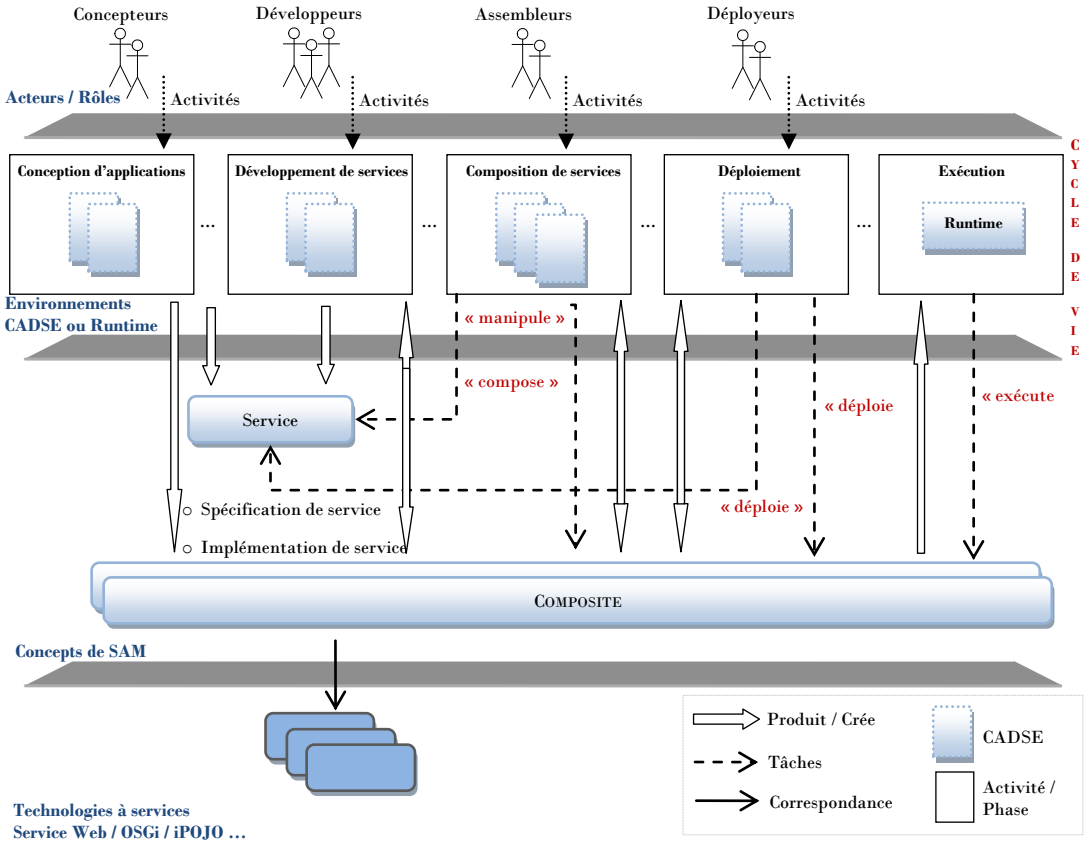

figure 17. Notre vision pour supporter le cycle de vie des applications.

Dans notre approche, chaque phase du cycle de vie peut disposer de plusieurs environnements spécialisés nommés CADSE(s)[1] [Adea][EVLL08] permettant de faciliter l'accomplissement des tâches associées à la phase. Les experts métiers définissent les concepts et les fonctionnalités du domaine métier des environnements en utilisant notre éditeur d'environnements nommé CADSEg[2]. Ces concepts sont spécifiés (ou traduits) en termes de concepts « génériques » pour réaliser des composites utilisables par les autres environnements du cycle de vie. Par exemple, les environnements de composition ou de déploiement de services manipulent des services atomiques ou composites et ils produisent en sortie des composites.

A l'exécution, nous fournissons une plate-forme d'exécution (Cf. Sections 5.3 et 6.3) qui est capable d'une part d'interpréter et d'enrichir la description des composites pour les exécuter, et d'autre part de garantir et satisfaire les caractéristiques des applications tels que les propriétés qu'elles définissent ou le dynamisme des éléments utilisés par ces applications.

[1] Computer Aided Domain Specific Engineering environment qui sera introduit dans la section 6.1

[2] Computer Aided Domain Specific Engineering environment generator

4.1.2 Besoins : support du cycle de vie des applications.

Nous souhaitons utiliser le concept de composite pour supporter et gérer le cycle de vie des applications. Pour cela, la spécification d'une composition devrait être définie, dans un premier temps, d'une manière « abstraite » et si possible elle devrait être « totalement » abstraite c'est-à-dire décrite en termes de descriptions (ou spécifications) de ces entités. De cette manière, nous permettrions de décrire l'architecture d'une application à construire en donnant la description des éléments qui la constituent, par exemple lors de la phase de conception.

Dans chaque phase du cycle de vie, la description d'un composite peut être enrichie au fur et à mesure pour indiquer plus d'informations, par exemple des propriétés qui sont liées aux activités réalisées dans la phase. La construction de composites d'une manière itérative est un point essentiel dans notre ambition d'utiliser le concept de composite pour supporter le cycle de vie des applications. Par conséquent, nous devrions permettre le raffinement itératif d'une composition en partant d'une spécification abstraite vers une composition qui sera exécutable.

Ainsi, nous ressentons le besoin de pouvoir « résoudre » partiellement une composition c'est-à-dire offrir la possibilité d'ajouter graduellement des informations dans sa définition ou bien de faire certains choix ultérieurement dans le cycle de vie. Ces aspects deviennent nécessaires dans la mesure où les éléments requis peuvent ne pas être disponibles à un moment donné, ou dans une étape donnée nous ne disposions pas de toute l'information nécessaire pour effectuer des choix sur les éléments, ou bien pour profiter au mieux des capacités des plates-formes dynamiques pour prendre en compte d'autres possibilités d'éléments.

La réutilisation d'éléments existants et disponibles est un point important qui facilite la réalisation « rapide » et automatique (ou semi-automatique) d'une composition. La sélection et la liaison de ces éléments peuvent être dynamiques c'est-à-dire elles peuvent être effectuées à l'exécution. Pour cela, il faudrait offrir des mécanismes permettant de sélectionner automatiquement, avant l'exécution et durant l'exécution, les éléments requis par une composition.

Une composition devrait décrire l'ensemble des propriétés et des contraintes souhaitées sur ses éléments contenus. Ces contraintes sont appelées des contraintes contextuelles parce qu'elles sont pertinentes que pour un composite. En effet, les acteurs de chaque phase du cycle de vie d'une application peuvent spécifier des propriétés fonctionnelles et/ou non fonctionnelles et ils peuvent ajouter des contraintes qui sont spécifiques à leurs tâches. Ainsi, la définition et/ou l'usage de langages, de mécanismes, des algorithmes et des outils deviennent indispensables pour exprimer les caractéristiques des composites c'est-à-dire leurs propriétés et leurs contraintes, et surtout pour réaliser la sélection automatique des « meilleurs » éléments.

Cependant, la plateforme d'exécution que nous proposons devrait être capable d'assurer la cohérence des propriétés et des contraintes contextuelles (parce qu'elles sont spécifiques aux composites) définies par les composites. En outre, pour permettre la réutilisation des éléments en général et des composites en particulier, notre plateforme devrait garantir que leurs caractéristiques soient conformes à celles définies par les applications qui les réutilisent. Ainsi, nous pourrions fournir un support permettant la

construction automatique (ou semi-automatique) de composites et assurer la cohérence globale des applications.

4.1.3 Génie logiciel et Composites.

Le génie logiciel couvre plusieurs activités et domaines métiers, et fournit des méthodes, des outils et des ateliers de support pour la conception de logiciels dans le but d'optimiser le coût de réalisation des logiciels et d'améliorer la qualité de leur production. Le génie logiciel identifie les différentes phases du cycle de vie des logiciels. Il décrit un ensemble d'activités qui sont réalisées par des acteurs et qui sont liées aux phases du cycle de vie. Généralement, une activité de génie logiciel s'effectue dans un ou plusieurs environnements logiciels, en vue de fournir une assistance aux acteurs pour la réalisation de leurs activités.

Nous proposons de considérer un projet de génie logiciel comme une succession de phases dont le but est de sélectionner, d'adapter ou de développer au fur et à mesure des briques logicielles (composants ou services dans le cas des applications à base de composant ou service) jusqu'à ce que la structure et le contenu de l'application à réaliser soient complètement définis. Dans chaque phase du processus logiciel, le travail est effectué par des acteurs humains avant l'étape d'exécution, ou bien par des machines à l'exécution.

A l'exécution, le système ne peut réaliser que des sélections d'éléments, par opposition au développement de code pouvant être effectué dans les phases antérieures à l'exécution. Les éléments qui sont automatiquement sélectionnés par le système pour une application doivent satisfaire sa description exprimant explicitement les propriétés et les contraintes à remplir par les éléments requis. Nous faisons l'hypothèse que notre système dispose d'un (ou de plusieurs) *repository* dans lequel seront disponibles un ensemble d'éléments.

En s'appuyant sur le concept de composite, nous fournissons un ensemble d'environnements logiciels et de supports d'exécution qui facilitent l'accomplissement des différentes tâches des acteurs intervenant dans le cycle de vie, de la conception à l'exécution, des applications logicielles. Grâce à ces environnements, nous assistons les acteurs dans leurs activités de génie logiciel en guidant l'interaction utilisateur. Ainsi, nous fournissons aux acteurs, des supports lors de la réalisation des activités de génie logiciel, et nous facilitons les transitions entre les phases du cycle de vie des logiciels.

Les experts des domaines métiers identifient et définissent l'ensemble des concepts et fonctionnalités de base qui sont pertinents par rapport aux activités associées à la phase sur laquelle ils interviennent. Ces définitions se font au niveau des environnements logiciels CADSE(s). Dans chaque environnement CADSE, les concepts manipulés peuvent ne pas être les mêmes et ils peuvent posséder différents niveaux d'abstraction. Par exemple, les concepts d'application (ou de système), de sous application (ou de sous système), de module ou de composant peuvent être définis pour être manipulés dans un CADSE décrivant par exemple l'architecture d'une application. Un environnement de composition de services utilisant par exemple une approche d'orchestration peut définir les concepts de procédé, activité, sous activité, connecteur, port, etc.

Comme exemple d'environnements spécialisés CADSE nous pouvons citer :

- **Environnement de développement de services** : permettant de réaliser des services qui sont conformes à l'approche orientée service. L'environnement CADSE SAM

CORE que nous allons présenter plus loin offre une possibilité de définir des services ;

- **Environnement de composition de services** : cet environnement [EDSV09] permet de spécifier des services composites pouvant intégrer des aspects et besoins comme la sélection et la composition « automatique » de services, et les langages permettant d'exprimer les contraintes permettant de construire les composites par intention (à partir d'un but qui exprime leurs propriétés et contraintes) ;
- **Environnement de déploiement** : il s'agit de l'environnement qui permet aux spécialistes du déploiement de définir les concepts et les stratégies de packaging et de déploiement concret des services composites ;
- **FOCAS** : il s'agit d'un environnement décrivant une manière de définir une composition de services en utilisant des concepts qui sont liés à l'approche d'orchestration de services [Ped09][PDE09].

Les concepts de base définis dans les environnements sont exprimés (ou bien seront traduits) en terme de composites, d'expressions de propriétés et de contraintes spécifiées dans notre langage qui seront pertinents lors des étapes de sélection des différents éléments participant à la composition de l'application.

4.2 PRINCIPES ET MECANISMES.

Les besoins identifiés dans la section 4.1.2 impliquent de disposer de deux principaux mécanismes :

- un mécanisme de définition de **composites par intention** c'est-à-dire une construction de composites à partir d'une description (ou un but) qui spécifient les propriétés et les contraintes à satisfaire par les éléments participant à la composition,
- un mécanisme supportant le concept de **composite partiel** c'est-à-dire une construction de composites par raffinement itératif et de sélection partielle des éléments nécessaires à la composition.

Le principe de construction de composites par intention entraîne la disposition d'un mécanisme de sélection automatique des éléments nécessaires à la composition. En effet, dans chaque phase du cycle de vie, de la conception à l'exécution des composites, notre outil devrait évaluer leurs descriptions, ensuite calculer les différents éléments participant à la composition en effectuant une sélection des « meilleurs » éléments parmi ceux qui sont disponibles et qui correspondent aux caractéristiques désirées.

Composite par intention.

Un composite par intention est défini à partir d'un but exprimant les propriétés et les contraintes à satisfaire par les éléments *pouvant participer à la composition.*

Composite partiel.

Un composite est dit partiel si à un instant donné certains des éléments *qu'il devrait contenir ne sont pas encore sélectionnés ou spécifiés. Dans ce cas, seule une sélection partielle ou retardée de ses éléments est réalisée.*

Sélection retardée.

Une sélection est dite retardée si certains choix d'éléments sont laissés ouverts pour une étape de sélection ultérieure (dans la même phase ou dans une autre phase du cycle de vie de l'application).

Notre solution se base sur plusieurs concepts à savoir les notions de **groupe d'équivalence** (ou groupe) et de **résolution** permettant de définir les propriétés qui facilitent la capacité de distinguer les entités et de les sélectionner dans une composition.

4.2.1 Groupe d'équivalence.

Un **groupe d'équivalence** (ou **groupe**) est défini comme un ensemble d'éléments qui sont indiscernables pour un certain point de vue. Un **point de vue** est défini comme un ensemble de propriétés (et de relations) que possèdent tous les membres d'un groupe d'équivalence. Le point de vue d'un groupe spécifie :

- **les propriétés communes** : il s'agit des propriétés (avec les mêmes valeurs) dont disposent tous les membres du groupe,
- **les propriétés variables** : il s'agit des définitions de propriétés que possèdent tous les membres du groupe mais avec des valeurs pouvant être différentes. Ces propriétés permettant de distinguer les membres du groupe.

Les membres d'un groupe peuvent aussi définir des propriétés (et des relations) qui leurs sont spécifiques. Il s'agit des **propriétés propres** à chaque membre du groupe d'équivalence. Si seules les propriétés communes sont pertinentes, alors les membres du groupe sont indiscernables. Le point de vue et le groupe, sont définis par les propriétés communes. Un groupe d'équivalence est constitué d'un **représentant** et de zéro (0), un (1) ou plusieurs **membre**(s). Dans notre approche, nous faisons les hypothèses suivantes :

- les propriétés communes d'un groupe (avec leurs valeurs associées) sont définies par un élément, appelé le représentant du groupe ;
- le représentant d'un groupe est un **type** pour les membres du groupe et définit les propriétés communes (avec leurs valeurs) et les propriétés variables ;
- les membres d'un groupe ne peuvent pas modifier les valeurs des propriétés communes ;
- les propriétés variables sont généralement utilisées lors de la sélection (résolution) des membres du groupe ;
- un membre d'un groupe dispose de ses propriétés propres, des propriétés communes (et leurs valeurs) et des propriétés variables, qui sont spécifiées par le représentant du groupe.

Un groupe d'équivalence est spécifié à partir d'un type de groupe. Un **type de groupe** (*TG*) est défini par le type *RT* du représentant du groupe et au moins par une relation spécifique *RGD*, dite relation de groupe, entre *RT* et le type *MT* de ses membres. Les membres d'un groupe peuvent être de différents types.

Pour mettre en usage nos concepts, nous donnons les définitions simples suivantes d'un métamodèle et d'un modèle. Un metamodèle est défini comme un ensemble de définitions de type (*TD*) et de définitions de relations (*RD*) entre les *TD* :

Métamodèle MM.

MM = <Types, Relations> avec Types = {TD} et Relations = {<RD, GroupAnnotation>} où

GroupAnnotation est un booléen précisant si RD est une relation de groupe et

RD = <TD_{src}-RelationName-TD_{dest}> /

TD_{src}, TD_{dest} ∈ Types ∧ TD = <Att, Ctr > / Att = {AD}, Ctr = {Constraint}

AD est une définition d'attribut où AD = <AName, AType>,
Constraint est une expression de contrainte (Cf. Section 4.3.2).

Notations :

Source (TD, RD) = vrai => TD est le type source de RD.

Destination (RD, TD) = vrai => TD est le type destination de RD.

Attributs (TD) = {AD} : retourne l'ensemble des définitions d'attributs de TD.

Un modèle M est défini comme un ensemble d'objets *O* et de liens *L* tels que M est conforme à un métamodèle MM.

Modèle M conforme à MM.

M = <O, L, MM> où MM = <Types, Relations> ∧

O = {o} ∧ L = {<o-r-d> / o, d ∈ O ∧ r ∈ Relations}

o = <type, {attribut}> / type ∈ Types ∧ attribut = <AD, value> ∧ AD ∈ Attributs (type).

Nous utilisons les notations suivantes :

Notations.

Soit M un modèle vu comme un graphe constitué d'éléments (typés), reliés par des liens (binaires, orientés, typés). On note :

EE(M) est l'ensemble des entités de M, EL(M) est l'ensemble des liens de M.

<o-r-d> ∈ M ⇔ il existe un lien de type r partant de o vers d.

Type (t) = T : t est une entité de type T.

Type de groupe TG.

Soit MM = <Types, Relations> un métamodèle.

isGroupType (RT) $\Leftrightarrow$ $\exists$ *<RT-RGD-MT>* $\in$ *Relations* $\wedge$ *isRelGroup (RGD)*

$\forall$ *<RT-RGD-MT> / isRelGroup (RGD)* $\Leftrightarrow$ *isMemberType (RT, MT)*

Notations :

memberType (RT) = {MT} / isMemberType (RT, MT).

isMemberType (RT, MT) : MT est un type des membres du type de groupe RT.

isRepresentantType (RT) : *RT est le type de représentant du type de groupe RT.*

isRelGroup (RD) : RD est une définition de relation avec l'annotation « group ».

Groupe d'équivalence G.

Soient MM = <Types, Relations> un métamodèle et M un modèle conforme à MM.

G est un groupe d'équivalence noté

$G = <R, \{M_i\}> \Leftrightarrow \exists\ RT \in Types\ /\ isGroupType\ (RT) \wedge Type\ (R) = RT.$

$\{M_i\}$ = *les membres du groupe G.*

$\forall\ M_i \in G : \exists\ MT \in Types\ /$

$Type\ (M_i) = MT \wedge \exists\ <R\text{-}RGD\text{-}M_i> \in EL\ (M) \wedge\ isRelGroup(RGD)\ ;$

$R = <\{R_{AV}\}, \{R_{Link}\}, \{RAD\}, \{RLD\}>$ *avec*
$\{R_{AV}\}$, *un ensemble d'attributs valués de R*
$\{R_{Link}\}$, *un ensemble de liens* / $\forall\ r \in \{R_{Link,}\}\ \exists\ D\ /\ <R\text{-}r\text{-}D> \in M$
$\{RAD\}$, *un ensemble de définition d'attributs*
$\{RLD\}$, *un ensemble de définition de relations* /
$\forall$ relT $\in \{RLD\}$, $\exists$ MT / MT $\in$ *memberType(RT)* $\wedge$
Source (MT, relT)

$M_i = <\{R_{AV}\}, \{R_{Link'}\}, \{M_{AV}\}, \{M_{Link}\}, \{MP_{AV}\}, \{MP_{Link}\}>$ *où*
M_{AV} = *une valeur de la définition d'attribut RAD,*
M_{Link} = *un lien de type RLD,*
R_{AV} *et* R_{Link} *sont en lecture seule,*

$Type\ (M_i) = MT\ /\ MT = <\{MT_{AD}\}, \{MT_{RD}\}>$

MP_{AV} = *une valeur de la définition d'attribut* MT_{AD} *et* MP_{Link} *un lien de type* MT_{RD}.

$\forall < M_i\text{-}RL\text{-}D> \in \{R_{Link'}\} \Rightarrow \exists\ <R\text{-}RL\text{-}D> \in \{R_{Link}\}.$
$\forall\ <R\text{-}RL\text{-}D> \in \{R_{Link}\} \Rightarrow \exists < M_i\text{-}RL\text{-}D> \in \{R_{Link'}\}.$

Notations :

Soit $G = <R, \{M_i\}>$ *un groupe*
isGroup (R) ;
$\forall\ M_i : R = group\ (M_i)\ ;$

group $(M_i) = R$: M_i *est un membre du groupe représenté par R.*

Définitions :

CommonProperties (G) $= \{R_{AV}\} \cup \{R_{Link}\}$

VariableProperties $(M_i) = \{M_{AV}\} \cup \{M_{Link}\}$

OwnProperties $(M_i) = \{MP_{AV}\} \cup \{MP_{Link}\}$

MemberProperties (M_i) = ***CommonProperties (group*** (M_i)***)*** $\cup$ ***VariableProperties*** $(M_i) \cup$ ***OwnProperties*** (M_i)

Les attributs valués {R_{AV}} et les relations {R_{Link}} désignent les **propriétés communes** du groupe. Les ensembles {RAD} et {RLD} désignent les définitions des **propriétés variables** des membres du groupe. Chaque membre du groupe spécifie les valeurs de ces propriétés dans {M_{AV}} et {M_{Link}}. {MT_{AD}} et {MT_{RD}} correspondent aux définitions d'attributs et de relations spécifiées par le type MT des membres M. Les **propriétés propres** d'un membre du groupe sont définies par {MP_{AV}} et {MP_{Link}}. La figure 18 illustre le principe des concepts de groupe et de type de groupe.

La figure 19 illustre deux exemples de groupe d'équivalence. *Fax* est un représentant de groupe (un type) définissant les propriétés que doivent disposer les membres du groupe : *WorkCentre_4260X* et *Brother_FAX_2920* fournissant les fonctionalités d'un télécopieur (*Fax*). Ces propriétés sont les suivantes :

- **Propriété commune** : l'interface décrivant les fonctionnalités fournies par un télécopieur.
- **Propriétés variables** : les propriétés printer, print_speed et transmission_speed sont définies par le représentant du groupe (Fax) et permettent à chaque membre du groupe (les appareils fournissant les fonctionnalités d'un Fax) de spécifier respectivement s'il dispose d'une option d'imprimante, les vitesses d'impression et de transmission de documents. Les membres du groupe peuvent avoir différentes valeurs pour les propriétés variables.

En plus des propriétés communes et variables, les membres du groupe Fax, à savoir WorkCentre_4260X et Brother_FAX_2920, possèdent d'autres propriétés qui leurs sont spécifiques (par exemple la propriété screen spécifiée par le télécopieur Brother_FAX_2920).

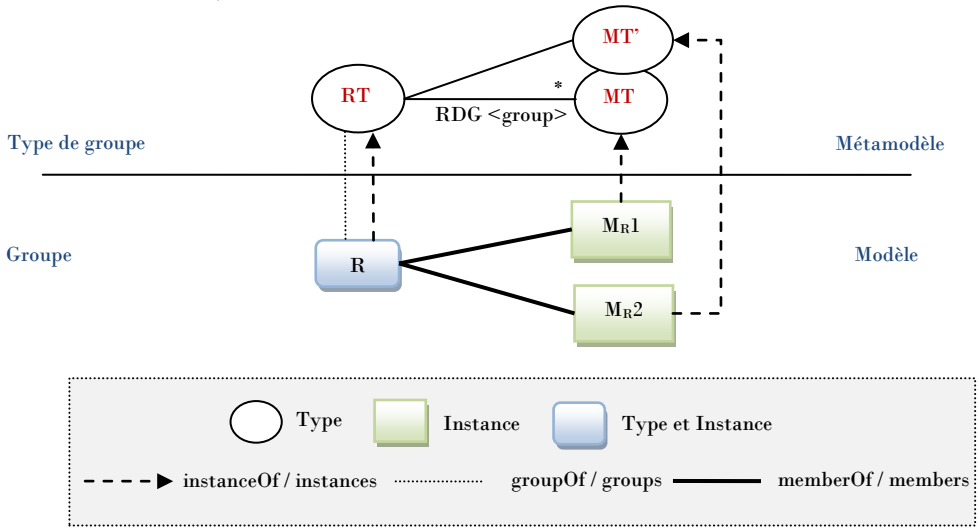

figure 18. Principe de la notion de groupe d'équivalence.

Un membre de groupe peut être un représentant d'un autre groupe. Comme exemple, nous pouvons citer *Brother_FAX_2920* (Cf. figure 19) qui est à la fois un membre du groupe *Fax* et le représentant du groupe dont les membres sont *B1 2920* et *B2 2920*. Notons que les propriétés dont dispose un élément *E* qui est à la fois membre et représentant de groupe, deviennent des propriétés communes pour les membres du groupe représenté par *E*.

Membre et Représentant de groupe.

Soient R = <{R_{AV}}, {R_{Link}}, {RAD}, {RLD}> un représentant de groupe et

M = <{R_{AV}}, {R_{Link}}, {M_{AV}}, {M_{Link}}, {MP_{AV}}, {MP_{Link}}> un membre du groupe représenté par R.

Dans le cas où M est un représentant d'un autre groupe on note :

M = <{RM_{AV}}, {RM_{Link}}, {RMAD}, {RMLD}> avec :

{RM_{AV}} = {R_{AV}} ∪ {M_{AV}} ∪ {MP_{AV}},

{RM_{Link}} = {R_{Link}} ∪ {M_{Link}} ∪ {MP_{Link}},

{RMAD} et {RMLD} sont des définitions d'attributs et de relations (les propriétés variables des membres du groupe représenté par M.

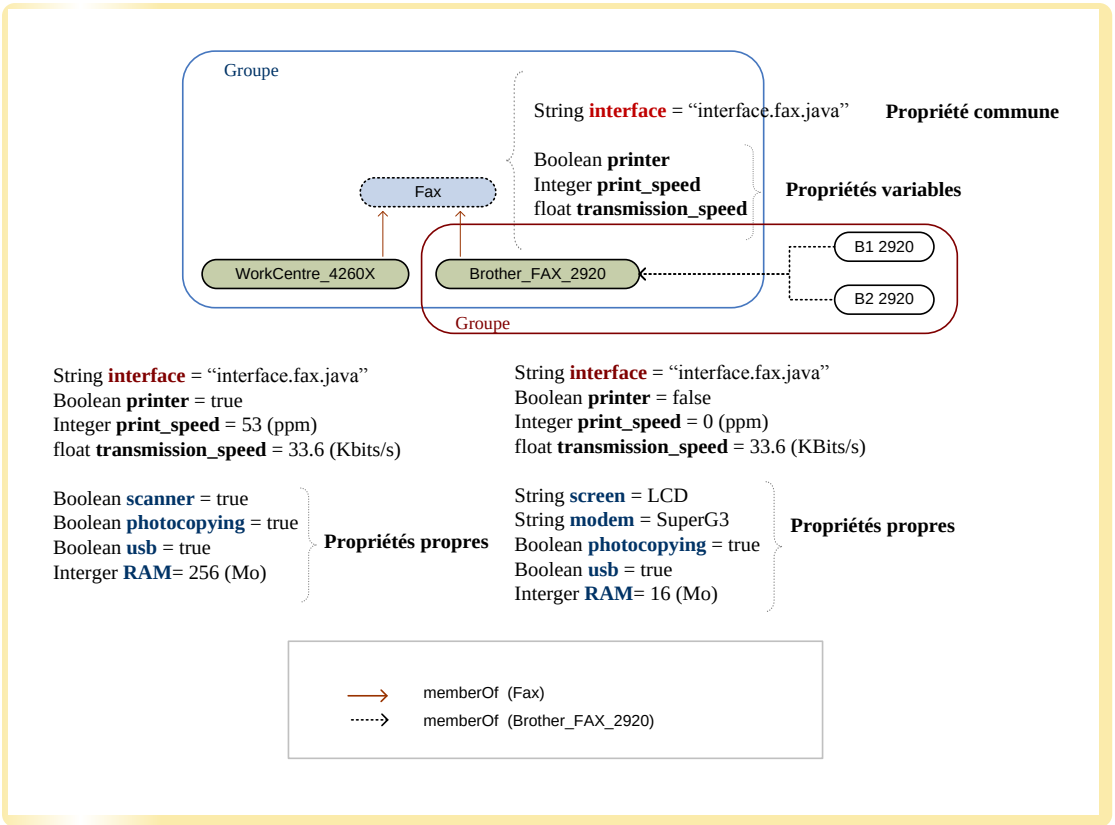

figure 19. Illustration de la notion de groupe d'équivalence.

4.2.2 Sélection et résolution de groupe.

La **résolution** d'un groupe d'équivalence consiste à sélectionner un ou plusieurs membres du groupe. Nous disposons d'une fonction nommée *Resolve* qui permet de naviguer la relation *members* (définie dans la figure 18) lors de la résolution d'un groupe G représenté par R et de retourner « max » membres de G sélectionnés qui remplissent les critères de sélection :

Résolution.

Soit $G = <R, \{M_i\}>$: $\forall M_i$: *group* $(M_i) = R$.

Resolve $(G, max, Exp, c) = \{M_1, \ldots, M_m\} \vee \emptyset$ *avec* $0 < m <= max$, *Exp une expression LDAP et* $c = \{element\}$ *représentant le contexte d'évaluation des critères de sélection* $\Rightarrow$

$\forall M_i$ *Verify* $(M_i, Exp(c))$

// *Si le groupe est dejà résolu*

Resolve $(G, max, Exp, c) = resolutions\ (c, G)$ *si* $resolutions\ (c, G) \neq G$.

Définitions et Notations.

resolutions (c, G) : si G est un groupe, retourne les membres du groupe G contenus dans l'ensemble c, sinon le prédicat retourne G.

resolutions (c, G) = ({gi} ⊂ c / group (gi) = G) ∨ (G si {gi} =∅)

Verify (E, Exp) retourne vrai si E respecte l'expression logique LDAP (*Exp*).

4.3 AUTOMATISATION DU PROCESSUS DE LA COMPOSITION.

Les caractéristiques de l'approche à service dynamique (Cf. Section 2.2) font que le contexte des applications dynamiques à service ne correspond pas à la manière habituelle de description d'applications sous forme d'une entité composite composé d'un ensemble d'éléments et de connections entre ces éléments, par exemple avec un langage de description d'architecture (ADL[1]) (Cf. Section 2.4) ou un procédé décrivant une orchestration de services. Cette manière de décrire un composite est assez rigide et ne supporte pas facilement la flexibilité requise par les applications dynamiques à service.

La description « traditionnelle » d'un composite en fournissant une liste exhaustive de tous ses éléments est contraignante, inadéquate, voir même impossible, pour un certain nombre de raisons parmi lesquelles :

- certains éléments requis par le composite peuvent ne pas exister à un instant donné dans un (ou plusieurs) repository disponible(s),
- certains éléments requis par le composite peuvent exister ailleurs mais inconnus ou indisponibles à un moment donné,

[1] Architecture Description Language

- on peut vouloir utiliser les éléments disponibles dans la machine lors de l'exécution de l'application.

Bien sûr il est possible de décrire et réaliser, avec les technologies traditionnelles comme les modèles à base de composants logiciels, des applications qui répondent aux besoins et hypothèses de l'approche à service dynamique tels que ceux identifiés ci-dessus. Cependant, les concepteurs et les développeurs doivent faire face à une complexité de réalisation due à la faiblesse du niveau d'abstraction des technologies qu'ils utilisent pour concevoir et réaliser leurs applications. Ils ne disposent généralement pas d'outils logiciels et de méthodes permettant de les assister dans la réalisation de leurs tâches. De nos jours, nous ressentons le manque de tels outils et environnements logiciels pour la réalisation et l'exécution d'applications.

L'ensemble des propriétés mentionnées fait que la sélection des éléments d'un composite, et la gestion des composites en général peuvent être ardues, et quasi impossible à faire à la main pour les composites complexes. C'est la raison principale pour laquelle il faudrait disposer de mécanismes de sélection automatique. L'un de nos objectifs est de proposer un mécanisme permettant de construire et gérer des composites répondants aux besoins des applications dynamiques et flexibles à service. Nous mentionnons ci-dessous les propriétés à garantir lors du processus de construction automatique de composites.

4.3.1 Propriétés à garantir.

Une composition devrait garantir les propriétés suivantes :

- **L'automatisation** : nous avons vu que construire manuellement des composites complexes est une source potentielle d'erreurs et que spécifier un composite par la liste exhaustive de ses éléments est parfois impossible. Une approche de composition devrait cependant offrir des supports facilitant l'automatisation du processus de réalisation de la composition.
- **La complétude** : un mécanisme de composition devrait assurer que tous les éléments nécessaires, sont sélectionnés ou spécifiés manuellement par l'utilisateur. En plus, un composite devrait disposer de la capacité d'être explicitement dans l'état « incomplet », ce qui est souvent le cas si certains de ses éléments seront identifiés plus tard lors d'une étape de sélection ultérieure ou pour profiter au mieux des capacités des plates-formes d'exécution comme OSGi ou iPOJO en découvrant à l'exécution les éléments requis par le composite.
- **La cohérence** : un mécanisme de composition devrait garantir que seuls les éléments qui satisfont les critères du composite (par exemple ses propriétés et contraintes) et qui sont comptables entre eux, sont sélectionnés. Ainsi, il devrait être capable de signaler les incohérences et les contraintes non respectées.
- **La sélection minimale** : un mécanisme de composition devrait assurer que seuls les éléments nécessaires sont sélectionnés.
- **L'évolution** : un composite peut être dans l'état « incomplet ». Il devrait cependant pouvoir facilement être complété. Ainsi, un composite devrait pouvoir être construit d'une manière incrémentale (itérative) en lui ajoutant au fur et à mesure de nouveaux éléments respectant ses critères.
- **L'optimalité** : le processus de sélection des éléments devrait être optimal afin de permettre la construction de la « meilleure » configuration possible, en prenant

compte des contraintes et des dépendances des éléments qui sont disponibles. Le système devrait être capable de calculer un ensemble optimal et cohérent d'éléments requis (par le composite) qui respectent les contraintes et propriétés du composite. Il faudrait cependant définir les critères d'optimalité de configurations de composites.

4.3.2 Langage de contraintes.

Dans cette section, nous détaillons notre langage d'expression de contraintes qui sont spécifiées comme dans le langage de contraintes OCL[1] [OMG06] sous la forme de prédicats permettant d'indiquer les besoins d'un élément. Le langage de contraintes que nous proposons permet de naviguer sur les relations définies entre les concepts d'un métamodèle donnée. Notre langage est une extension du langage de recherche LDAP[2] proposé dans [The]. Il diffère du langage OCL car dans notre langage, les contraintes peuvent être associées à la fois à un type ou à des instances. En plus, dans notre système les instances peuvent aussi être des types ; il s'agit de types dynamiques qui sont les représentants de groupe (Cf. Notion de groupe d'équivalence détaillée dans la section 4.2). Le concept de groupe et le typage dynamique ne sont pas reconnus par le langage OCL.

Dans notre langage, les expressions de contraintes peuvent contraindre les éléments en fonction de leurs propriétés fonctionnelles ou non fonctionnelles, et peuvent être obligatoire ou non (il s'agit de contraintes préférentielles). Nous avons défini un langage générique de contraintes qui est capable d'une part de manipuler les concepts d'un méta-modèle et d'autre part de naviguer à travers les relations existantes entre ces concepts. La syntaxe et la sémantique du langage seront détaillées plus loin dans cette section.

Les contraintes définies par un élément peuvent servir à assurer l'intégrité et la cohérence du *repository* ou bien guider la sélection des éléments pouvant être utilisés par un composite. Nous détaillons ci-dessous ces deux types de contraintes.

CONTRAINTES DE COHERENCE DU REPOSITORY.

Une expression de contrainte peut servir à assurer l'intégrité et la cohérence du *repository* contenant les éléments qui sont disponibles. Le groupe *Fax* peut définir une contrainte permettant de contraindre tous les membres souhaitant offrir les fonctionnalités d'un télécopieur. Ainsi, lors de la création de la relation « *memberOf* » (Cf. figure 19) permettant de spécifier les fonctionnalités fournies par un membre, les contraintes définies par ces éléments seront évaluées, si elles ne sont pas respectées alors la relation « *memberOf* » ne pourra pas être créée. Nous pouvons ainsi assurer les propriétés suivantes :

- tous les membres d'un groupe sont conformes aux contraintes spécifiées par ce groupe,
- garantir la cohérence et l'intégrité du repository.

---

[1] Object Constraint Language

[2] RFC 1960: A String Representation of LDAP Search Filters. Disponible à http://www.ietf.org/rfc/rfc1960.txt

CONTRAINTES DE SELECTION.

Une expression de contrainte définie par un élément peut servir à filtrer les éléments pouvant être choisis pour participer à une composition. En effet, un élément peut déclarer des contraintes sur d'autres éléments avec lesquels il pourrait être utilisé dans une application. Ce type de contraintes sert par exemple à imposer des propriétés fonctionnelles ou non fonctionnelles à d'autres éléments. Ces contraintes doivent donc être respectées à tout moment, dans n'importe quel contexte ou application, et sont appelées des **contraintes intrinsèques.** Ce type de contraintes peut être utilisé pour :

- automatiser le processus de sélection des éléments nécessaires à un composite,
- valider la cohérence des éléments sélectionnés pour participer à la composition,
- réduire les risques d'erreurs lors de la phase d'exécution (on assure que les éléments sélectionnés avant l'exécution sont conformes aux exigences du composite et des autres éléments utilisés par le composite).

Contrainte intrinsèque.

*Une contrainte est dite intrinsèque si elle est définie par un élément et elle doit être respectée à tout moment dans n'importe quel contexte ou application utilisant l'*élément *associé.*

Un composite peut également requérir (ou imposer) des propriétés ou contraintes aux éléments existants (pouvant être utilisés par le composite). Ces contraintes ne sont pertinentes que pour le composite lui même. Ces relations et propriétés sont appelées des caractéristiques contextuelles : **contraintes et propriétés contextuelles**.

Contrainte contextuelle.

Une contrainte est dite contextuelle si elle est définie par un composite et elle n'est valide et pertinente que pour le composite qui la spécifie.

SYNTAXE ET SEMANTIQUE DE NOTRE LANGAGE DE CONTRAINTES.

La syntaxe et la sémantique de notre langage d'expression de contraintes et de sélection sont détaillées dans l'annexe A. Le langage proposé s'appuie sur le concept de groupe (Cf. Section 4.2), d'expression de chemins et sur les expressions logiques LDAP. Une contrainte de sélection possède une forme générale ressemblant à (Cf. Annexe A) :

```
Context (Select | Optional) <PathExp> [<max>] (<Expression-LDAP>)
```

Context est implicite ; c'est l'élément auquel est associé l'expression. Le mot clé "*Self*" désigne cet élément et <*Contains*> désigne les éléments contenus dans "*Self*" dans le cas où *Self* est un composite. *Select* et *Optional* sont des mots clés du langage désignant respectivement des expressions de sélection obligatoires ou optionnelles. *PathExp* permet de définir la portée de la contrainte c'est-à-dire l'ensemble des entités sur lequel va porter la contrainte. *PathExp* retourne un ensemble d'éléments de même type ou un ensemble vide. <*max*> exprime la cardinalité de la contrainte c'est-à-dire le nombre maximal d'entités que

la contrainte doit retourner. Par défaut, *max* vaut 1 lorsque *PathExp* retourne des groupes ; l'infini sinon. *Expression-LDAP* correspond à une expression logique LDAP étendue avec quelques opérateurs (Cf. Annexe A).

```
<PathExp> = ( T | "Self" )[{["." | ".."] ri}] i = 1 ... n
     avec T un type ou un groupe, et ri des types de relations.
```

Notre langage se base sur le concept de groupe d'équivalence qui fournit un typage (dynamique). Les expressions de contraintes sont donc typées (les types et les attributs utilisés dans les expressions de contraintes peuvent être statiquement validés). Nous illustrons notre langage de contraintes par les exemples ci-dessous :

```
Select Fax [2]( & (printer = true) (print speed >= 25) ) ;
```

Etant donné que *Fax* est un groupe (Cf. figure 19), l'expression implique la sélection de deux de ses membres au plus (résolution du groupe représenté par *Fax*) disposant des propriétés suivantes :

- printer = true : indiquant que le télécopieur dispose d'une option d'imprimante,
- print_speed >= 25 : le télécopieur permet d'imprimer des documents avec une vitesse d'impression supérieure ou égale à 25 ppm.

```
Optional Device (shared = "true") ;
```

Device est un type défini dans un métamodèle correspondant aux types des télécopieurs, par exemple *WorkCentre_4260X* (Cf. figure 19). Cette contrainte indique de sélectionner de **préférence** des membres du type *Device* ayant la propriété (*shared = true*) indiquant qu'elles peuvent être partagées par plusieurs composites.

Le calcul du chemin (*PathExp*) ressemble aux nombreux formalismes d'expression de chemins, comme dans OCL, XPath[1], FPath[2] [Dav05] ou OQL[3] [ASL89]. Le calcul du chemin retourne un ensemble d'éléments de même type, mais si le long du chemin, un élément est un groupe déjà résolu dans le contexte d'évaluation de l'expression, alors ce sont les résolutions c'est-à-dire les membres de ce groupe qui sont utilisées à la place du groupe lui-même. Lorsqu'un groupe à été résolu une fois dans un contexte donné, le choix effectué est respecté par toute évaluation qui « passe par » ce groupe. Le contexte d'évaluation d'une contrainte, pour un élément donné, est constitué d'un ensemble d'éléments ($c = \{élément\}$) avec lesquels il va être composé.

La façon précédente de calculer et de naviguer les chemins, les concepts de group et de résolution, et le typage dynamique ne sont pas reconnus dans les langages de navigation cités ci-dessus. C'est pourquoi ces langages ne peuvent pas directement être utilisés dans

---

[1] XML Path : un langage de navigation proposé par le consortium W3C et qui permet d'adresser des parties de documents XML (http://www.w3.org/TR/xpath/)

[2] Ce langage permet de naviguer dans une architecture Fractal et de sélectionner des éléments (composants, connecteurs, etc.). Sa syntaxe ressemble à celle du langage XPath.

[3] Object Query Language : une extension du langage SQL qui fournit un langage de requêtes pour manipuler des bases de données orientées objets.

notre approche. La sémantique du prédicat *Satisfies*, permettant de dire si un élément respecte ou pas une contrainte, est décrite ci-dessous :

Vérification de contrainte.

Exp (c) ≡ "*Self*" ("Select" | "Optional") <PathExpression> [<max>] (<Expression-LDAP>) ";" avec Expression-LDAP est une expression LDAP et **c = {element}** le contexte.

PathExpression = {mi} ∨ ∅ ≡ T {["."] | [".."] <relation$_n$>} 0<n<=k où

(T ∈ Types ∨ isGroup (T) ∨ T = "Self") ∧ ∀ **n <relation$_n$> ∈ Relations** :

Soit p PathExpression avec p= T.r$_1$…….r$_n$

// p dans le contexte (c) retourne l'ensemble {elt}.

Soit p (c) = {elt}

// pi (c) est la valeur de p (c) de longueur i dans le contexte c
Soit p$_i$ (c) = T.r$_1$….r$_i$ = {mi$_j$} avec 0<=i<=n et j = 1…q

// La valeur p$_i$ (c) est calculée à partir de celle de p$_{i-1}$ (c) avec une navigation
// directe (.) ou inverse (..) sur la relation r$_i$
qi (c) = p$_{i-1}$ (c).r$_i$ ∨ p$_{i-1}$ (c)..r$_i$ avec p$_{i-1}$ (c) = {mi-1$_j$} j=1…q
p$_{i-1}$ (c).r$_i$ = ∪ Destination (mi-1$_j$, r$_i$) pour j = 1..q // Union de tous les ens dest de r$_i$
p$_{i-1}$ (c)..r$_i$ = ∪ Source (r$_i$, mi-1$_j$) pour j = 1...q // Union de tous les ens src de r$_i$
// Si T est un type alors p$_0$ est l'ensemble des instances de T
p$_0$ (c) = extention (T) si T est un type
// Si T est un groupe résolu dans c alors p$_0$ est l'ensemble des membres de T
// contenus dans c
p$_0$ (c) = resolutions (c, T) si T est un groupe résolu dans c
// p$_0$ = T si T est un groupe et c ne contient aucun membre de T
p$_0$ (c) = T si isGroup (T) ∧ resolutions (c, T) = ∅

// Sinon p$_0$ vaut la valeur « Self »
p$_0$ (c) = "Self" sinon.

// Substitution du groupe G par ses résolutions gi

Soit qi (c) = {qij} ∨ ∅ avec j = 1…r

//Ne considérer que les résolutions de groupe déjà effectuées
isGroup (qi(c)) ⇒ p$_i$ (c) = $\coprod_{j=1}^{j=r}$ **resolutions (c, qij)**
¬isGroup (qi(c)) ⇒ p$_i$ (c) = qi (c)
resolutions (c, G) = ({gi} ⊂ c / group (gi) = G) ∨ (G si {gi} =∅)

// <max> vaut par défaut « 1 » lorsque *PathExpression* retourne des groupes
isGroup (p (c)) ∧ undefined (max) ⇒ max = 1

// <max> vaut par défaut l'infini sinon
¬isGroup (p (c)) ∧ undefined (max) ⇒ max = ∞

Vérification de contrainte (Suite).

// L'expression retourne un ensemble contenant au maximum <max> éléments
// vérifiant l'expression logique LDAP correspondante

On a : p (c) = {s_i} avec i = 1…o

¬isGroup (p (c)) ⇒ Exp (c) = {s_i} ∨ ∅ / Verify (s_i, Expression-LDAP)

isGroup (p (c)) ⇒ Exp (c) = $\coprod_{i=1}^{i=o}$ resolve (si, max, Expression − LDAP, c)

// Un élément s respecte une contrainte Exp (c) si l'expression de chemin de Exp ne contient pas l'élément s ou bien s appartient à l'ensemble des éléments de Exp (c)
Satisfies (s, Exp (c)) ⇒ s ∉ p (c) ∨ s ∈ Exp (c) avec p le "PathExpression" de Exp.

Notations.

Destination (s, r) : l'ensemble des éléments destination de la relation r issue de s.
Source (r, d) : l'ensemble des éléments source de la relation r pointant sur d.

extention (T) : l'ensemble des instances du type T.
undefined (max) : la valeur <max> n'est pas spécifiée dans l'expression de contrainte.

4.3.3 Algorithme générique de sélection.

Nous introduisons un algorithme générique de sélection qui se base sur tout métamodèle et sur les notions de groupe d'équivalence et de résolution. Nous définissons le concept de type de composite (TC) comme un cas particulier d'une définition de type (TD) fournissant les informations qui sont nécessaires lors de la résolution de composites de type TC. Ces informations concernent :

- La définition des navigations (*PathDefinition*) indiquant pour un type donné les types de relation à suivre lors de l'exécution de l'algorithme de sélection.
- L'ensemble des types de groupe à résoudre (*ResolvedGroupType*) durant l'exécution de l'algorithme de sélection.

Type de composite TC.

Un type de composite TC ∈ Types est noté :

TC = <{AD}, {RD}, {PathExpression}, {ResolvedGroupType}> où
{AD} est l'ensemble des definitions d'attribut de TC
{RD} est l'ensemble des definitions de relation de TC
PathExpression = $T.r_1 \ldots. r_n$ *//voir ci-dessus*

ResolvedGroupType = {GT} où isGroupType (GT) = true

Un composite C de type TC possède une **définition** qui fournit :

- ses caractéristiques c'est-à-dire ses propriétés et ses contraintes (*Constraints*) qui sont exprimées avec notre langage de contraintes SELECTA (Cf. Section 4.3.2),
- les informations concernant les groupes à ne pas résoudre c'est-à-dire les groupes retardés de sélection (*Delay*) et les groupes à ignorer (*Ignore*),
- les représentants et les membres de groupe qu'il contient initialement (*Contains*).

Définition de composite CD.

CD = <TC, Contains, Delay, Ignore, Incomplete, Constraints > où :
Type(CD) = TC,
Delay, Ignore, Incomplete = $\{R_i / isGroup\ (R_i)\}$,
Contains = List (E) où E ∈ EE(M) avec M un modèle.
Constraints = {Expression} où Expression = une contrainte de sélection du composite.

A partir de la définition CD du composite, l'algorithme complètera la liste des éléments de *Contains* et calculera l'ensemble des groupes non résolus (*Incomplete*).

Contains.

Soient **TC = <{AD}, {RD}, {PathExpression}, {ResolvedGroupType}>** un type de composite et **C = <TC, Contains, Delay, Ignore, Incomplete, Constraints >** un composite.

// Les contraintes du composite + celles des éléments contenus dans Contains et Delay
// constraints (m) retourne l'ensemble des contraintes définies par m
AllConstraints = Constraints $\cup$ ($\cup$ **Constraints** (**m**))$\forall$ **m** $\in$ **Contains** $\cup$ **Delay**

// m_i doit être contenu par le composite C s'il existe un chemin de C vers m_i et
// s'il respecte les contraintes définies dans AllConstraints.

Contains = List (m_1, …, m_n) /

$\forall$ **m_i avec i** $\in$ **{1, …, n}**

m_i.isInComposite (AllConstraints, C)

$\wedge$ **m_i** $\notin$ **Delay** $\cup$ **Ignore** // m_i n'est pas contenu dans Delay ni Ignore ;

$\nexists$ **m** $\notin$ **Contains / m.isInComposite (AllConstrains** $\cup$ **Constraints (m), C)**

(il n'exite pas un élément m / m.isComposite vrai et m n'est pas contenu dans Contains)

Définition :

m.isInComposite (AllConstraints, C) où *C* est un composite de type TC, *Contains* est la liste d'éléments contenus par C et *AllConstraints* un ensemble de contraintes. *PathExpression* est une liste de chemins définie par TC.

m.isInComposite (AllConstraints, C) $\equiv$

$\exists$ **p** $\in${ **PathExpression**} **/**
m $\in$ **p (Contains)** // il existe un chemin vers m

$\wedge$ $\forall$ **Exp** $\in$ **AllConstraints : Satisfies (m, Exp (Contains))** // m respecte les contraintes

Note. L'ensemble *Contains* est defini, entre autre, comme étant un ensemble d'entités vérifiant toutes les contraintes des éléments de ce même ensemble. En théorie, il faudrait calculer toutes les combinaisons d'éléments de l'ensemble de départ et sélectionner les ensembles qui vérifient toutes les contraintes. Ce problème est NP complet, l'algorithme que nous proposons par la suite (Cf. Chapitre 5) utilise des heuristiques et des hypothèses simplificatrices.

Un composite SELECTA possède un état pouvant être :

- **Complet ou incomplet** : un composite est dit complet si tous les groupes qui ne sont pas explicitement retardés ont été résolus. Le composite est dans l'état incomplet s'il contient des groupes pour lesquelles aucun membre respectant les contraintes du composite n'est trouvé dans un (ou plusieurs) repository.
- **Retardé** (delayed) : il existe des groupes contenus par le composite (*Delay*). La sélection d'un (ou plusieurs) membre(s) de ces groupes est retardée, par exemple pour profiter au mieux des membres qui seront ultérieurement disponibles ou simplement ils seront réalisés plus tard.
- **Conflit** (conflict) : un composite est dans l'état « conflit » s'il y a des conflits de contraintes à une étape du processus de sélection de ses entités contenus (*Contains*). Un composite en conflit est nécessairement incomplet.

4.4 SYNTHESE.

Dans ce chapitre, nous avons présenté les concepts, les principes et les mécanismes « génériques », comme la notion de groupe d'équivalence ou de résolution, qui sont à la base de notre approche de composition nommée SELECTA. Nous avons idendifié les propriétés qui devraient être garanties par une composition. Notre approche fournit des solutions face aux besoins et aux défis de la composition (Cf. Chapitre 3) pour l'automatisation du processus de la composition.

Afin d'automatiser le processus de la composition, nous avons proposé un langage d'expression de contraintes et un algorithme « générique » de sélection des entités partcipant à une composition. Une expression de contrainte spécifiée avec notre langage peut être **intrinsèque** à l'élément qui la définie c'est-à-dire la contrainte devrait être respectée dans n'importe quel contexte ou application utilisant cet élément. Une expression peut aussi être **contextuelle** ; dans ce cas elle n'est pertinente uniquement dans le contexte d'un composite donné qui utilise les éléments concernés par la contrainte.

Les contraintes peuvent servir à :

- assurer la **cohérence du *repository*** contenant les éléments disponibles ou
- guider la **sélection** des entités pouvant participer à une composition.

Le chapitre suivant introduit notre approche de composition dans un contexte spécifique de construction d'applications à base de services, utilisant un métamodèle particulier celui de la machine SAM (Service Abstract Machine, Cf. Section 5.1).

5

SELECTA DANS LE CONTEXTE DE SAM.

Ce chapitre définit le concept de composite pour un métamodèle spécifique, celui de la machine SAM[1]. Nous fournissons un algorithme spécifique à ce métamodèle pour construire des configurations cohérentes de composites. Avant de présenter notre proposition dans le contexte de SAM, nous introduisons tout d'abord la plateforme SAM, une machine abstraite à services que nous utilisons pour l'exécution d'applications dynamiques à service. Nous décrivons par la suite l'architecture, les concepts et les différentes caractéristiques de notre approche de composition de services au sein de la machine SAM. Puis, nous introduisons brièvement notre plateforme d'exécution de composites. Nous concluons ce chapitre par une synthèse.

5.1 SAM : LA MACHINE ABSTRAITE A SERVICES.

5.1.1 Objectifs et motivations.

Il existe de nos jours diverses plates-formes à services. Nous pouvons citer celles des technologies de services web [ACKM03][W3C04b] ou OSGi[2] [OSG05a]. Les plates-formes à services qui existent à l'heure actuelle sont homogènes et supportent généralement un seul type de service, ce qui explique que la plupart de ces plates-formes à services ne sont pas compatibles et interopérables.

Pour faire face à ce problème d'hétérogénéité des plates-formes à services, notre équipe de recherche ADELE [ADEc] propose une approche et un système nommé SAM [ES09][ADEb] dont l'objectif est de fournir un canevas intégrant un ensemble

[1] Service Abstract Machine

[2] Open Services Gateway initiative

d'environnements et d'outils qui permettent d'une part de réaliser et d'exécuter des services, et d'autre part de cacher l'hétérogénéité des plates-formes à services existantes. La machine SAM n'est pas une plateforme à services et elle ne cherche pas à remplacer celles qui existent. En effet, il s'agit d'une machine abstraite à services dont le rôle est de déléguer dynamiquement l'exécution des services à des plates-formes concrètes à services.

L'approche SAM fournit un ensemble de concepts et des fonctionnalités de base à travers une interface de programmation (API[1] en anglais) de haut niveau d'abstraction :

- permettant de rechercher des services qui sont disponibles à partir d'un ensemble de critères,
- fournissant des supports et des facilités pour le déploiement et la distribution « transparente » de services,
- gérant le comportement dynamique des services c'est-à-dire l'apparition et la disparition des services.

5.1.2 Architecture et concepts.

L'architecture proposée dans SAM est constituée de trois couches (Cf. figure 20) :

- **La couche physique** : elle contient diverses technologies et plates-formes à services telles que les services web, OSGi, DPWS[2] [Mic06a], UPnP[3] [UPn08] ou Java EE[4].
- **La couche logique** : elle contient un modèle de l'exécution contenant des services représentant ceux qui tournent sur les plates-formes concrètes de la couche physique.
- **La couche SAM Core** : elle se charge de déléguer dynamiquement l'exécution des services qui sont spécifiés dans le modèle d'exécution vers la ou les plates-formes à services qui supportent l'exécution de ce type de service. En effet, la délégation dynamique de l'exécution de services est réalisée en fonction du type de service à exécuter et des capacités offertes par les plates-formes sous jacentes.

La figure 20 illustre les différentes couches de l'architecture de la machine SAM :

[1] Application Programming Interface

[2] Devices Profile for Web Services

[3] Universal Plug and Play

[4] Java Enterprise Edition

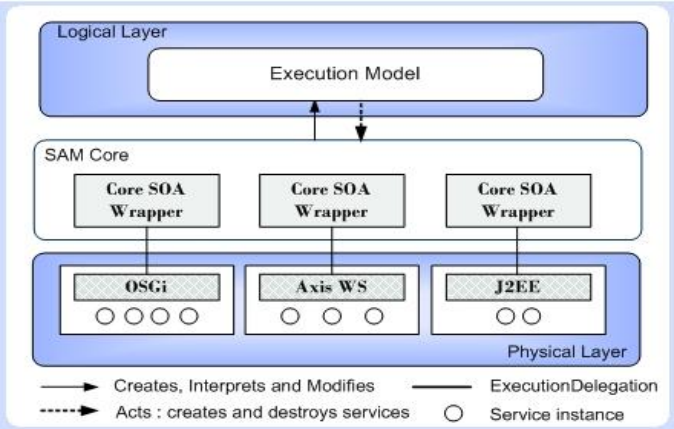

figure 20. Architecture de l'approche SAM.

Les éléments du modèle d'exécution sont spécifiés en termes de concepts de la machine SAM qui supporte les concepts et les fonctionnalités essentielles de l'approche à service (Cf. Section 2.2). SAM spécifie donc un ensemble de concepts et de fonctionnalités qui sont définis dans la plupart des plates-formes à services. Les concepts définis dans SAM sont « abstraits » et sont mappés aux concepts « concrets » des plates-formes à services sous jacentes. La figure suivante décrit les concepts de base de la machine SAM et les relations entre ces concepts :

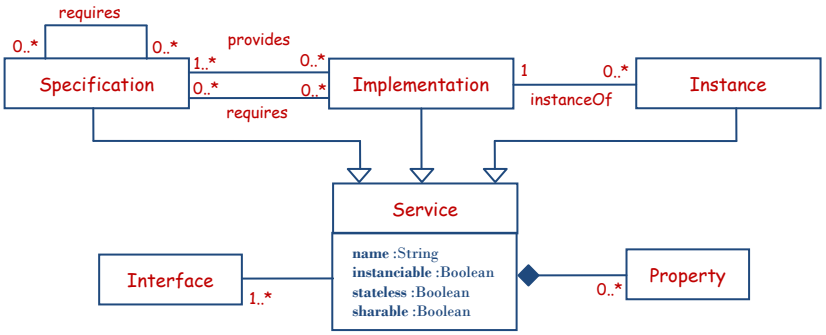

figure 21. Concepts de base de la machine SAM.

La notion de service correspond au concept central de la machine SAM. Dans SAM, un service est une entité contenant une (ou plusieurs) interface(s) (dans la version actuelle, il s'agit d'une interface Java) avec un ensemble de propriétés. Les propriétés d'un service sont définies sous la forme d'attributs et de valeurs permettant d'exprimer les caractéristiques du service. Dans l'approche SAM, un service possède trois matérialisations : il peut s'agir d'une *spécification*, d'une *implémentation* ou d'une *instance*.

LE CONCEPT DE SPECIFICATION DE SERVICE.

Une *spécification* représente un service pouvant indiquer à travers des relations *requires* l'ensemble de ses dépendances vers d'autres spécifications c'est-à-dire ses services requis. Une spécification de service ne contient pas de détails liés à une technologie (ou une plateforme) particulière. La capacité de définition de dépendances entre les spécifications de service offre la possibilité de spécifier des services composites structurels comme dans l'approche SCA [OSO07][BII+05], en termes de spécifications de service.

Pour illustrer les concepts de SAM, nous introduisons dans la figure 22 un exemple décrivant un ensemble de services SAM qui sont liés à différents domaines (métiers ou

techniques) tels que la réservation en ligne de billets de spectacles ou de restaurant. Ces services peuvent être utilisés dans différents contextes par diverses applications. Dans notre exemple, *Search_Show* est une spécification de service qui offre une fonctionnalité permettant aux clients de rechercher des spectacles à partir d'un ensemble de critères spécifiés comme la date, le lieu ou le type de spectacle (concert, festival, théâtre, …). *Booking_Show* et *Booking_Restaurant* sont des services qui permettent respectivement la réservation de spectacles et de table(s) au niveau d'un restaurant avoisinant la salle où se déroulera le (ou les) spectacles désirés(s) par le client.

Bank est une spécification de service qui décrit une banque quelconque et ses services fournis. Le service *Payment* offre une fonctionnalité permettant de réaliser une opération de paiement en ligne. Un paiement en ligne est accepté uniquement dans le cas où le client paye sa réservation par carte bancaire ; c'est pourquoi, dans notre exemple le service *Payment* requiert les services *Credit_Card* et *Bank,* ce qui donne la capacité aux clients de préciser leur moyen de paiement, le type de carte bancaire à utiliser et la banque qui a délivrée cette carte.

Le service *Security* définit les fonctionnalités permettant d'assurer les propriétés de sécurité lors de la réalisation de transactions bancaires et le service *Log* permet de sauvegarder l'historique des actions réalisées lors d'une transaction. Le service *BD* définit les fonctionnalités d'une base de données. Le service *GUI* offre une interface graphique pouvant être utilisée pour afficher des informations à travers un site Internet (ou Intranet). Ce service peut servir par exemple pour visualiser les fonctionnalités de réservation en ligne d'une application Web offrant cette possibilité.

LE CONCEPT D'IMPLEMENTATION DE SERVICE.

Une *implémentation* est une entité (un bout de code) qui fournit une ou plusieurs spécifications de service en réalisant les fonctionnalités définies dans leurs interfaces associées. Une spécification de service peut être fournie par zéro ou plusieurs implémentations. Dans SAM, une implémentation hérite de toutes les dépendances de services (spécifications) définies par la spécification de service qu'elle fournit. Par exemple, la spécification *Payment* (Cf. figure 22) dépend des spécifications *Bank* et *Credit_Card* alors toutes les implémentations *de Payment* dépendront aussi des spécifications *Bank* et *Credit_Card*. En plus de ces dépendances, une implémentation a la possibilité d'ajouter d'autres dépendances qui lui sont propres vers des spécifications de service.

Dans la figure 22, nous avons deux banques nommées *Alpha* et *Beta* qui sont disponibles et qui fournissent les services associés à une banque. *Visa_Card* et *CB_Card* sont des implémentations de service définissant respectivement une carte bancaire (le service *Credit_Card*) de type « visa » et « carte bleue ». Trois implémentations du service *Payment* sont aussi disponibles :

- *Payment_Secure_Log* : assure un service de paiement en ligne sécurisé. Il requiert deux autres services à savoir *Security* et *Log*.
- *Payment_Secure* : réalise un service de paiement en ligne qui est sécurisé mais qui ne sauvegarde pas l'historique des actions réalisées pendant la transaction.
- *Payment_Simple* : réalise un service de paiement en ligne non sécurisé.

LogImpl_1 est une implémentation qui fournit un service de *Log*. Elle dépend du service *BD* (une base de données pour la sauvegarde d'informations). Les services *MySQL* et *HSQLDB* sont deux implémentations du service *DB*. Dans notre exemple, nous avons

plusieurs implémentations disponibles du service *GUI*. Ces implémentations sont caractérisées en fonction du langage de programmation utilisé : HTML avec le service *HTML* ou Java avec le service *JSP*. Le service *Device_GUI* est une autre implémentation qui fournit le service *GUI* et qui est compatible et utilisable par un appareil électronique comme un téléphone portable ou un baladeur numérique.

LE CONCEPT D'INSTANCE DE SERVICE.

Une *instance* est une entité qui représente l'exécution d'une implémentation de service. Une instance de service hérite par définition les valeurs des propriétés et les relations qui sont définies par son implémentation, et par transitivité celles de sa spécification.

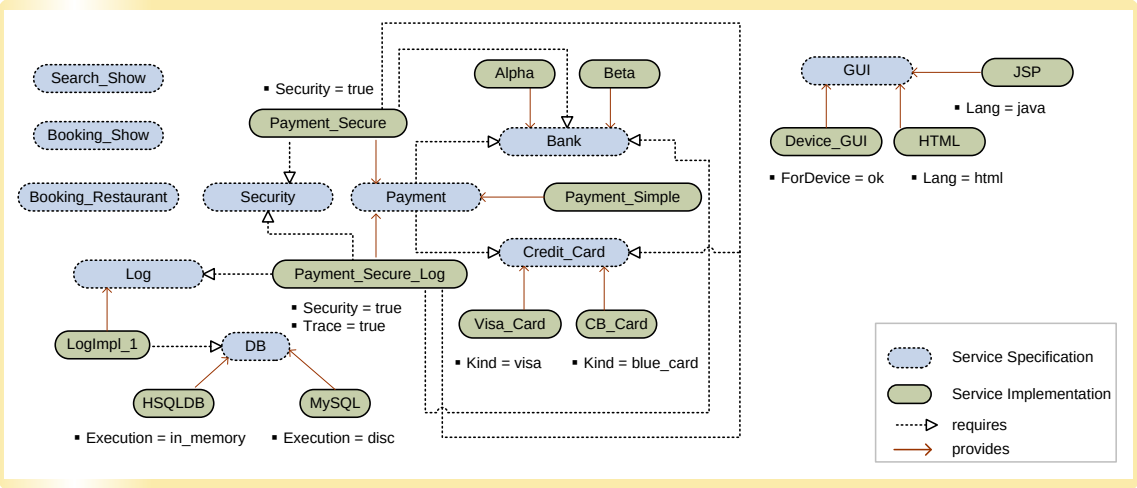

figure 22. Exemple de repository SAM.

PROPRIETES INTRINSEQUES DES SERVICES.

Dans SAM, un service dispose d'un ensemble de propriétés qui sont supportées par quelques (ou bien toutes) les plates-formes à services existantes. Ces propriétés sont appelées des **propriétés intrinsèques** des services. Le tableau suivant décrit **quelques-unes** des propriétés prédéfinies dans SAM :

Propriétés	Sémantique
@instanciable [*true*|*false*]	Un service peut avoir des instances. Le processus de création d'instances est appelé une instanciation. Dans SAM, un service est dit non instanciable si l'instanciation est réalisée par des procédures d'administration externes (manuelles ou non) et non pas lors de l'exécution d'une application contenant le service. Un service web ou bien un appareil de type DPWS ou UPnP ne sont pas instanciable. Dans SAM, la valeur par défaut de cette propriété est *true*.
@sharable [*true*|*false*]	Cette propriété indique si un même service peut être simultanément utilisé par plusieurs applications, à l'intérieur d'une machine SAM. Dans le cas où le partage n'est pas autorisé, un autre service (implémentation ou instance) est créé à chaque demande. La valeur par défaut est *true*.
@stateless [*true*|*false*]	Cette propriété indique si le service possède un état ou non. La valeur par défaut de cette propriété est *true*.

tableau 15. Propriétés intrinsèques des services.

En plus des propriétés intrinsèques, un service SAM (une spécification, une implémentation ou une instance) peut définir d'autres propriétés qui lui sont propres et qui permettent de spécifier ses caractéristiques.

La machine SAM étend les plates-formes à services existantes en intégrant un ensemble de concepts, de propriétés et de fonctionnalités qui sont nécessaires pour l'exécution d'applications dynamiques à service (certains des services qui composent une application peuvent apparaître et/ou disparaître à tout moment). La machine SAM propose un système qui garantie une distribution « transparente » des services, ce qui constitue l'un de ses grands avantages.

Le système SAM est extensible c'est-à-dire des concepts et fonctionnalités peuvent être intégrés. Il n'est cependant pas défini d'une manière monolithique. Par exemple, une extension spécifiant un support de déploiement ou un mécanisme de composition de services pour faciliter la construction « automatique » d'applications à base de service peut être intégré dans le système SAM, appelé SAM CORE. Dans la suite de ce chapitre, nous présenterons une extension du système SAM qui propose une manière de réaliser la composition dynamique de services. Le tableau 16 illustre les caractéristiques et les avantages de la machine SAM :

Caractéristiques de la machine SAM

Hétérogénéité	Elle résout le problème d'hétérogénéité des plates-formes à services comme les services web, OSGi, UPnP. L'exécution des services est réalisée par les plates-formes à services sous-jacentes. Cette exécution est déléguée dynamiquement vers ces plates-formes par le système SAM.
Abstraction	SAM est une machine abstraite à services. Elle est générique et abstraite car elle ne fournit pas (directement) les services qu'elle propose.
Distribution	Le système SAM est distribué. Plusieurs machines SAM distantes peuvent communiquer entre elles. La distribution des services SAM est gérée d'une façon « transparente » par le système.
Extensibilité	Le système SAM est extensible. D'autres concepts et fonctionnalités peuvent être ajoutés dans SAM. Dans cette thèse, nous proposons une extension permettant de construire et d'exécuter des applications flexibles et dynamiques : les éléments qui les composent peuvent apparaître ou disparaître à tout moment, ils peuvent être substitués par d'autres ayant au moins les mêmes propriétés, etc.
Dynamisme	Le système SAM gère et assure le comportement dynamique des services tel que leur apparition et/ou leur disparition. Il permet ainsi d'exécuter des applications dynamiques à base de service.

tableau 16. Caractéristiques du système SAM.

5.2 COMPOSITES SELECTA.

5.2.1 Architecture, concepts et caractéristiques de SELECTA.

Dans notre système [EDS09], nous définissons le concept de composite comme une extension de la machine SAM présentée dans la section 5.1. L'extension Composite SELECTA [EDS09][EDSV09] que nous proposons est l'une des extensions possibles du noyau de base SAM CORE. Elle propose une manière de définir un composite prenant compte des besoins identifiés précédemment (Cf. Section 4.1.2). D'autres extensions définissant le concept de composite peuvent être définies. La figure suivante illustre l'architecture de notre système :

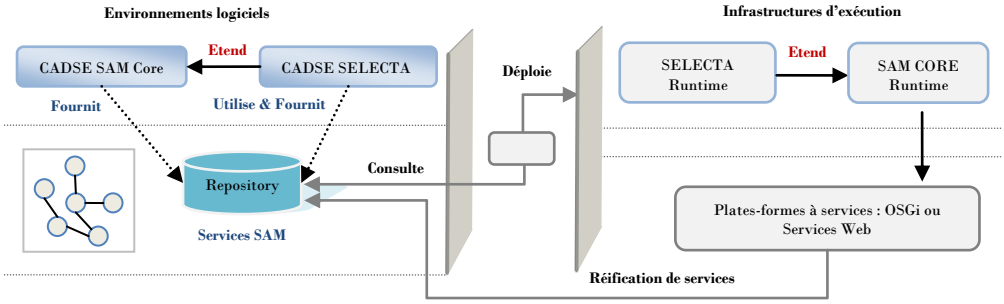

figure 23. Architecture de notre système.

Notre système est constitué de deux parties : les environnements logiciels CADSE dans lesquels s'effectuent les activités de génie logiciel, avant l'exécution, et les infrastructures d'exécution qui sont capables d'exécuter des services atomiques et composites. Nous proposons plusieurs environnements CADSE en fonction de l'étape du cycle de vie de l'application et des tâches de génie logiciel à réaliser, par exemple nous pouvons citer :

- CADSE SAM CORE : cet environnement [EDSV09] est utilisé lors de la phase de conception et/ou de développement de services. Il permet de développer des services (pouvant être disponibles dans un ou plusieurs repository SAM) qui respectent les principes de l'architecture orientée service. L'environnement offre la possibilité de développement de services avec un niveau d'abstraction assez élevé en cachant les détails techniques liés aux technologies à services, par exemple les services web ou OSGi, pour faire face à l'hétérogénéité des plates-formes à services ;
- SELECTA : cet environnement [EDSV09][EDS09] permet la spécification de services composites en intégrant des aspects et besoins de la composition comme la sélection et la composition automatique de services, et les langages permettant d'exprimer les contraintes permettant de construire les composites par intention (à partir d'un but qui exprime leurs propriétés et contraintes). Notre environnement de composition SELECTA est une extension du CADSE SAM CORE, ce qui permet d'une part de réutiliser ses concepts et fonctionnalités, et d'autre part de spécifier de nouveaux concepts à partir de ceux qui existent. Les services composites SELECTA possèdent diverses caractéristiques qui seront présentées dans la section suivante. L'environnement de composition sera détaillé plus loin dans ce manuscrit dans les chapitres suivants ;

- CADSE SAM DEPLOIEMENT : il s'agit de l'environnement qui permet aux spécialistes du déploiement de définir les concepts et les stratégies de packaging et de déploiement concret des services composites. Des outils spécialisés au déploiement de services peuvent être définis. Ces outils ou environnements de déploiement consultent généralement dans un premier temps les services qui sont disponibles dans un ou plusieurs repository de services. Ensuite, en fonction des politiques et des stratégies de packaging et de déploiement qui sont spécifiées ils réalisent le déploiement réel des services.

Dans ce manuscrit, seuls les environnements logiciels CADSE SAM CORE (présenté dans la section 5.1 et SELECTA (Cf. Section 6.2) sont présentés en détail. L'infrastructure d'exécution des composites, nommée SELECTA RUNTIME, étend la plateforme d'exécution SAM et s'appuie sur les fonctionnalités fournies par ce dernier. Elle sera détaillée plus loin dans ce manuscrit (Cf. Section 6.3). Dans la suite de cette section, nous introduisons les extensions apportées par SELECTA dans le noyau SAM CORE.

EXTENSION DU CONCEPT DE SERVICE.

Nous proposons une extension du concept de service SAM (spécification, implémentation ou instance) permettant aux services de définir des contraintes qui sont spécifiées avec notre langage de contraintes (Cf. Section 4.3.2). Le concept de contrainte (*Constraint*, voir la figure 24) offre à un service la capacité d'exprimer un ensemble de caractéristiques qui sont nécessaires pour son exécution correcte. Il s'agit des contraintes intrinsèques des services et elles peuvent servir par exemple pour imposer une plateforme d'exécution particulière ou des propriétés fonctionnelles ou non fonctionnelles comme la sécurité (la confidentialité) ou la performance.

La figure suivant illustre le concept de service SAM (Cf. Section 5.1) :

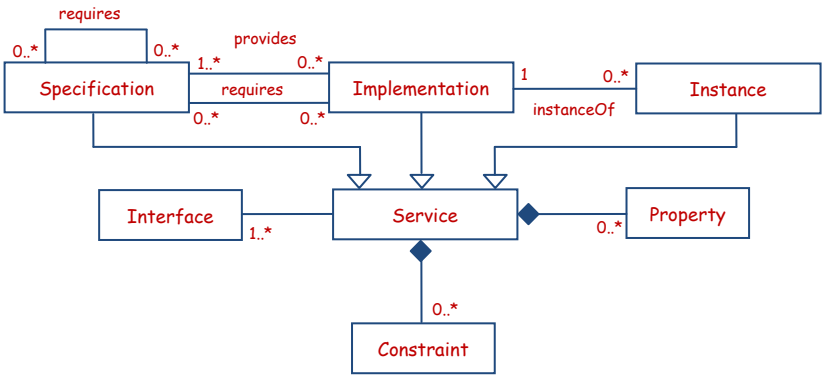

figure 24. Extension du concept de service SAM.

CONCEPT DE COMPOSITE.

Dans les approches traditionnelles de construction d'applications logicielles, une application est souvent décrite sous la forme d'une entité composite constituée de tous les éléments (composants) qui la composent. Une application est spécifiée en fait comme un ensemble d'éléments et de connections entre ces éléments. Cette manière de définir une application n'est pas adéquate pour les applications modernes qui se basent par exemple sur le paradigme à service, car les services pouvant être utilisés par l'application peuvent ne pas être disponibles ou même être inconnus à un instant (avant ou pendant la phase d'exécution de l'application).

Dans notre approche de composition, comme le montre la figure 25, un composite SELECTA est une implémentation de service pouvant contenir des spécifications (grâce à la relation *containsSpec*), implémentations (grâce à la relation *containsImpl*) ou instances (avec relation *containsIns*) de services. Les implémentations de service qui sont contenues dans un composite peuvent être atomiques (c'est-à-dire des services élémentaires définis avec le noyau SAM CORE) ou composites (c'est-à-dire des services définis par exemple avec notre extension de SAM CORE, nommée Composite SELECTA), qui offre la possibilité de construire des composites hiérarchiques. De cette façon, nous donnons la capacité de spécifier une composition structurelle de services.

Composite SELECTA.

*Un **composite SELECTA** est une implémentation de service pouvant contenir d'autres services (spécifications, implémentations ou intances de services).*

De même que dans les approches traditionnelles de composition, un composite SELECTA peut être spécifié par la liste exhaustive des services qu'il contient (spécifications et/ou implémentations de service) en mentionnant explicitement les relations *containsSpec* et *containsImpl* qui sont associées aux services qui sont nécessaires au composite.

Un composite SELECTA est avant tout une implémentation de service. Il n'est cependant pas nécessairement autonome (en anglais, *self-contained*). Il peut donc avoir des dépendances vers d'autres services à travers des relations *requires* (voir la figure 25). De la même manière, la structure d'un composite peut ne pas être complètement définie à un moment donné. Ainsi, un composite devrait être capable d'indiquer ses éléments (services) incomplets en définissant explicitement les choix de services à effectuer lors d'une étape ultérieure du processus de sélection des services requis.

Par conséquent, la sélection d'un service S pour un composite C ne doit pas être réalisée tant qu'une relation *delaySpec* ou *delayImpl* existe entre C et S. Une sélection de services retardée doit être effectuée ultérieurement, par exemple lors de la phase de développement, au déploiement et au plus tard pendant l'exécution de l'application. La stratégie à adopter est spécifiée par l'utilisateur. Le système SELECTA offre les mécanismes et moyens permettant d'effectuer ou de retarder des sélections de services.

Le système SELECTA étend le noyau SAM CORE en définissant de nouveaux concepts (par exemple le concept de composite) et de nouvelles relations entre les concepts (par exemple les relations *containsSpec*, *containsImpl* ou *delaySpec*). Le concept *Characteristics* (de la figure 25) permet de spécifier les contraintes contextuelles d'un composite. Il contient ainsi l'ensemble des contraintes de sélection qui seront évaluées lors du processus de réalisation de la composition : les services sélectionnés devront respecter les contraintes définies par le composite. Ainsi, ce concept permet de définir une partie du but (propriétés et contraintes) du composite et il est primordial pour la construction d'un composite par intention.

La relation « *wires* » (de la figure 25) est une relation contextuelle indiquant la liaison directe entre deux implémentations, mais seulement dans le contexte du composite. En effet, une implémentation peut être liée à une autre dans une application donnée, et ne pas l'être dans une autre application.

La relation « *refines* » entre les composites offre la possibilité de définir un composite en raffinant un ou plusieurs composites. Le processus de raffinement d'un composite consiste par exemple à :

- ajouter de nouveaux services ou de nouvelles relations au composite à raffiner,
- spécifier de nouvelles propriétés du composite,
- indiquer ou imposer des contraintes sur les services pouvant être utilisés (ou sélectionnés) pour participer à la composition.

La construction de composites par raffinement itératif peut être utilisée pour la gestion du cycle de vie des applications en utilisant le concept de composite. En effet, dans chaque phase du cycle de vie de l'application, les acteurs manipulent un (ou plusieurs) composite(s) pour y ajouter des informations et propriétés qui sont liées à leurs connaissances et compétences et produisent en sortie des composites. Lors de la phase de déploiement, ils peuvent ajouter par exemple des informations liées au packaging et au déploiement des services contenus dans un composite.

Un composite SELECTA possède un état pouvant être **complet**, **incomplet**, **retardé** ou **conflit** (Cf. Section 4.3.3). Dans notre approche, seuls les services composites qui sont dans l'état complet peuvent être directement exécutés par l'infrastructure d'exécution.

La figure 25 décrit le concept de composite SELECTA :

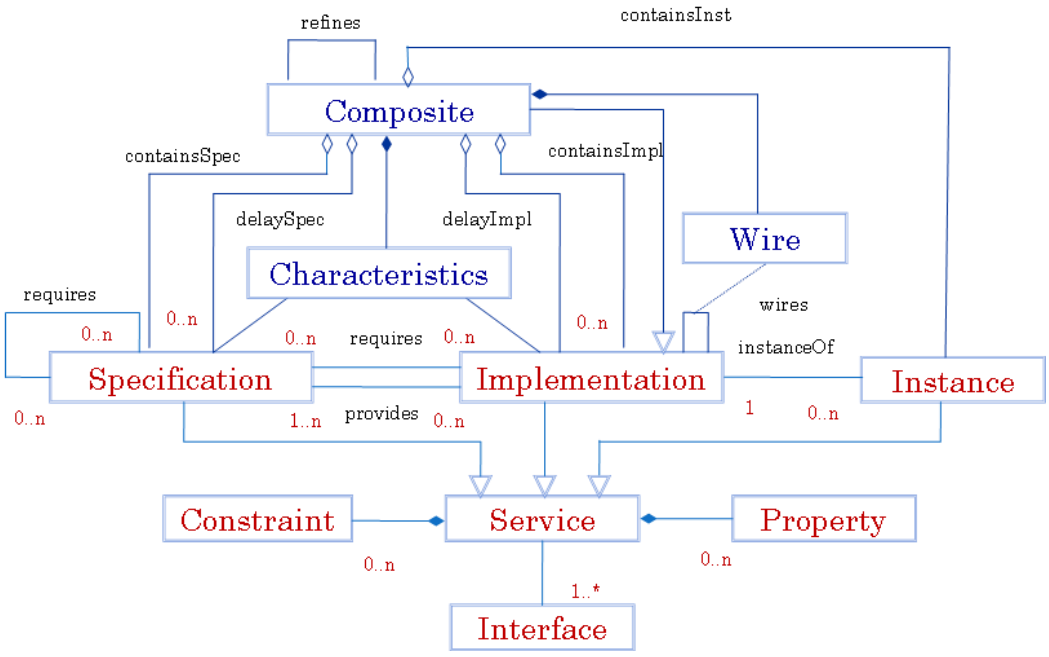

figure 25. Métamodèle de composition dans SAM.

Le tableau 17 résume les caractéristiques du concept de composite SELECTA :

Caractéristiques du concept de composite SAM
Un composite SELECTA peut contenir des services de tout type (spécifications, implémentations ou instances). Il peut avoir des dépendances vers d'autres services et il n'est donc pas forcément autonome.
Un composite peut être explicitement incomplet : les sélections retardées sont définies grâce aux relations *delaySpec* et *delayImpl*.
Un composite peut donner aux services des caractéristiques visibles uniquement par ce composite : propriétés, relations et contraintes contextuelles. Il peut être construit à partir des caractéristiques (propriétés et contraintes) à satisfaire par les services pouvant participer à la composition.
Un composite SELECTA possède un état : complet, incomplet, retardé ou conflit. Seuls les composites complets peuvent être exécutés par l'infrastructure d'exécution.

tableau 17. Caractéristiques des composites SELECTA.

5.2.2 Algorithme de sélection spécifique à SAM.

Nous proposons un analyseur et un interpréteur de contraintes spécifiques à SAM et qui sont capables d'une part de valider la syntaxe des expressions de contraintes et de sélection, et d'autres part d'évaluer et connaître leurs sémantiques. L'algorithme permettant de sélectionner les services qui sont requis par un composite et qui sont conformes aux critères du composite s'appuie sur ces outils précédemment cités.

Un type de composite SELECTA est défini comme suit :

Type de composite SELECTA.

$TC_{SELECTA}$ = <{*complete, delayed, conflict*}, {<*C-requires-S*>,<*C-containsSpec-S*>, <*C-delaySpec-S*>, <*C-containsImpl-I*>, <*C-delayImpl-I*>, <*C-containsInst-Inst*>}, {*S.requires, I.requires*}, {*S, I*}> *avec S = Specification, I = Implementation, Inst = Instance et C = Composite.*

Un composite SELECTA (de type $TC_{SELECTA}$) possède les attributs (*complete*, *delayed* et *conflict*) précisant son état et les relations (*requires*, *containsSpec*, *containsImpl*, *containsInst*, *delaySpec* et *delayImpl*) indiquant son contenu, ses dépendances et les sélections retardées (Cf. figure 25). **PathDefinition** = {S.requires, I.requires} définit les chemins à naviguer. Les types de groupe à résoudre (**ResolvedGroupType**) sont *Spécification* et *Implémentation*.

La définition précédente précise que lors du calcul des services nécessaires pour un composite SELECTA, l'algorithme sélectionnera des membres du type de groupe *Spécification* et *Implémentation*. La navigation se fera uniquement sur les relations *requires* à partir des services de type *Spécification* ou *Implémentation*. Lors de la création d'un service composite, l'utilisateur doit indiquer les informations suivantes $CD_{SELECTA}$ = <$TC_{SELECTA}$, Contains, Delay, Ignore, Incomplete, Constraints> :

- la ou les spécifications de services qui sont fournies par le composite (relation « provides),
- il peut imposer (sélection manuelle) certains services (spécifications et/ou implémentations) contenus initialement par le composite (grâce aux liens « containsSpec » et/ou « containsImpl »), (*Contains*)

- il peut indiquer les dépendances externes (*Ignore*) du composite (ses services requis grâce aux relations « requires »),
- il peut spécifier les services (spécifications ou implémentations) à partir desquels la résolution (sélection) ne sera pas réalisée (sélections retardées (*Delay*) grâce aux relations « delaySpec » et « delayImpl »)
- il peut déclarer, s'il le désire, un ensemble de contraintes (*Constraints*) exprimant la définition en intention du composite, par exemple les propriétés à satisfaire par les services à sélectionner durant le processus de sélection de ses services.

En fonction des relations existantes et des contraintes spécifiées par le composite, l'algorithme permet de :

- calculer (ou sélectionner) l'ensemble des services (*Contains*) constituant une configuration du composite ; les services doivent être compatibles entre eux et respecter les contraintes du composite,
- construire les relations comme « containsSpec » ou « containsImpl » indiquant les services qui ont été sélectionnés lors de la résolution,
- garder l'information (*Incomplete*) concernant les résolutions incomplètes de services (par exemple, les spécifications de service pour lesquelles aucune implémentation n'est trouvée dans le repository de services),
- construire la description complète de la configuration du composite résumant toutes les actions (sélections) qui ont été réalisées.

5.3 COMPOSITES A L'EXECUTION : SELECTA RUNTIME.

Les composites doivent être exécutables. Pour ce faire, nous fournissons une plate-forme d'exécution nommée SELACTA RUNTIME (Cf. Section 6.3) qui est chargée d'une part d'interpréter et d'enrichir la description des composites dans le but de les exécuter, et d'autre part de garantir le respect des propriétés et des contraintes définies par les composites et la disponibilité dynamique des services utilisés par ces composites.

En d'autres termes, nous offrons un support d'exécution dont le rôle est d'exécuter et de gérer les composites SELECTA. SELECTA RUNTIME prend en entrée la (ou les) description(s) de composites produite(s), en général, à l'aide de notre environnement SELECTA. SELECTA RUNTIME effectue ensuite toutes les actions nécessaires pour l'exécution correcte du (ou des) composite(s) associé(s) aux descriptions, tout en garantissant à tout instant, la conformité et la cohérence de l'exécution des composites par rapport à leurs descriptions (propriétés et contraintes).

5.3.1 Composites et modèles d'architectures exécutables.

Le concept de composite que nous proposons permet d'assister les architectes, concepteurs et/ou assembleurs durant la spécification et l'assemblage des composants de l'architecture modélisant une application. Le support logiciel que nous apportons permet d'assurer la sûreté et la cohérence des modèles d'applications construits avec notre environnement SELECTA. Notons que nous utilisons les mêmes modèles, langages, algorithmes de sélection et outils depuis la spécification/conception jusqu'à l'exécution de l'application, et que les modèles des applications sont disponibles et manipulables c'est-à-dire consultables et/ou modifiables à l'exécution. Il existe quelques travaux [FR07][GBF09] (nommés *models@runtime*) qui utilisent et étendent les principes et mécanismes de

l'ingénierie dirigée par les modèles pour des environnements d'exécution chargés d'exécuter des systèmes logiciels. D'autres travaux de recherche sont en cours de réalisation au sein de notre équipe de recherche pour la spécification et la gestion de la conformité entre les modèles d'architectures (décrivant les applications) et les modèles d'exécution (reflétant l'exécution réelle de l'application).

5.3.2 Composites dynamiques.

Nous avons vu que les composites SELECTA peuvent être dans divers états : *complets*, *incomplets*, *retardés* ou en *conflits*, et que la structure de leur contenu peut évoluer en permanence, ceci en fonction des choix de services effectués lors du processus de sélection de services et de la disponibilité de ces services. Il nous faudrait cependant disposer d'un ensemble de mécanismes, avant et/ou pendant la phase d'exécution, pour assurer diverses propriétés telles que :

- réaliser la sélection automatique des services requis par un composite,
- effectuer les sélections retardées de services,
- gérer l'évolution des composites (en termes de structure, de propriétés et de contraintes),
- garantir à tout moment la cohérence et la complétude des composites, etc.

Nous avons défini un **langage générique** de définition de contraintes et un **algorithme générique** de sélection que nous avons présenté dans la section 4.3. Nous avons réalisé un **interpréteur** de ce langage et un **post processeur** spécifique à la machine SAM. Le post processeur est chargé de compléter le travail réalisé par l'interpréteur en créant des relations particulières, comme *containsSpec* ou *containsImpl*, à partir du composite vers les services sélectionnés et en réalisant les connecteurs (des relations « wires ») entre les services contenus par le composite. Les relations créées sont conformes au métamodèle de composition décrite dans la figure 25.

COMPOSITES INCOMPLETS ET SELECTIONS AUTOMATIQUES.

La plateforme d'exécution SELECTA RUNTIME exécute les services du composite qui ont été sélectionnés et se charge de compléter les services qui ne sont pas encore sélectionnés en les choisissant dynamiquement, s'ils sont disponibles et remplissent les propriétés et les contraintes du composite. SELECTA RUNTIME réalise ainsi, à l'exécution, des sélections de services afin de compléter les services manquants qui sont nécessaires au composite.

A l'exécution, les sélections retardées de services doivent être effectuées s'il y a des services requis par un composite qui sont disponibles et conformes à sa description. Les composites qui sont dans l'état *incomplet* peuvent également être éxécutés, mais risquent d'échouer si les services manquants ne sont toujours pas disponibles lors de l'exécution.

SELECTA RUNTIME transforme et complète graduellement la description des composites jusqu'à ce qu'ils ne contiennent que des instances de services qui sont reliés entre eux par des connecteurs (des relations « *wires* », voir la figure 25) et qui respectent les contraintes des composites. La sélection de services est cependant guidée par :

- le contexte d'exécution c'est-à-dire par les services qui sont disponibles et qui tournent sur la plateforme avec les valeurs de leurs propriétés,
- les implémentations de service qui sont disponibles dans un (ou plusieurs) *repository* de services,

- les implémentations de services qui sont actuellement définies (ou sélectionnés) dans la description du composite,
- les contraintes et les propriétés du composites,
- les contraintes des services déjà sélectionnés (qui sont contenus par le composite) lors d'une étape antérieure dans le processus de sélection de services.

COMPOSITES ET ARCHITECTURES LOGICIELLES DYNAMIQUES.

Les travaux sur les architectures logicielles dynamiques fournissent une base pour supporter l'évolution ou la reconfiguration dynamique des logiciels à l'exécution. L'approche à service peut être utilisée pour supporter l'évolution et le dynamisme de l'architecture des logiciels. Des travaux comme [YML05] proposent des solutions basées sur le paradigme à service et le principe consiste généralement à réifier les concepts de l'architecture en termes de concepts de l'approche à service.

Nous utilisons à l'exécution, les mêmes langages, algorithmes et outils lors de la sélection et de la résolution de services. Nous garantissons à l'exécution, les mêmes propriétés que dans les phases antérieures du cycle de vie de l'application telles que la gestion de sa cohérence ou de sa complétude.

5.4 SYNTHESE.

Nous proposons une approche de composition qui fournit :

- des langages et modèles ayant un haut niveau d'abstraction décrivant la composition et exprimant les critères de sélection. Notre démarche se base sur l'ingénierie dirigée par les modèles afin d'augmenter le niveau d'abstraction de la spécification de la composition. L'idée est de faciliter la spécification d'une composition indépendamment des technologies à services comme les services web ou OSGi. Nous proposons également un langage d'expression de contraintes et un algorithme pour sélectionner automatiquement les éléments requis par un composite à partir des besoins spécifiés.
- Un environnement SELECTA pour définir et exécuter des composites flexibles. L'environnement intègre un ensemble d'outils permettant d'automatiser la sélection des services requis par le composite, de construire et d'exécuter d'une manière itérative et incrémentale des composites flexibles, avec une option offrant la possibilité de retarder certaines sélections de services, voir même jusqu'à l'exécution, de compléter les sélections, etc.

Notre système apporte des solutions face aux besoins des architectures logicielles que nous avons identifiés dans la section 2.4.2. Notre système garantit des propriétés telles que la cohérence et la complétude de la composition, mais ne garantit pas la validation sémantique de l'architecture. Nous proposons des outils et environnements par exemple pour la spécification, la construction et l'exécution d'applications.

6

REALISATION ET EXPERIMENTATIONS.

Dans ce chapitre, nous présentons en détail la mise en œuvre de notre proposition à travers le système nommé SELECTA. Nous introduisons tout d'abord notre éditeur d'environnements spécialisés CADSE [Adea]. Nous utilisons la technologie CADSE pour fournir des environnements spécialisés pour la réalisation de tâches spécifiques à chaque phase du cycle de vie d'une application, par exemple la conception et le développement des éléments de base (services ou composants), le déploiement des éléments ou bien la composition de ces éléments pour constituer l'application. Nous introduisons ensuite l'architecture et les caractéristiques essentielles du système SELECTA. Puis, nous présentons la plateforme d'exécution associée à SELECTA. Nous concluons ce chapitre par une synthèse des expérimentations réalisées et de la validation de notre approche.

6.1 CADSE : ENVIRONNEMENTS SPECIALISES.

6.1.1 Objectifs et motivations.

Lors de la réalisation d'applications logicielles, les développeurs doivent souvent prendre en compte un ensemble d'aspects et de contraintes qui sont liés aux applications à réaliser. Ces aspects peuvent être de deux catégories : il peut s'agir de propriétés fonctionnelles ou non fonctionnelles tels que la performance ou la sécurité des briques (éléments) de base de l'application. Ces propriétés permettent de spécifier l'ensemble des caractéristiques et contraintes à satisfaire lors de la réalisation des logiciels. La réalisation des logiciels et ces derniers eux même deviennent ainsi de plus en plus complexes.

Par conséquent, les développeurs doivent faire face à cette complexité et tenter de la maîtriser. De nombreux besoins découlent de la maîtrise de cette complexité des logiciels. Parmi les besoins nous pouvons citer : l'augmentation du niveau d'abstraction de la spécification des applications et les environnements et outils pour réaliser les applications.

Environnements et automatisation.

Le besoin d'environnements de développement d'applications se fait ressentir pour automatiser et faciliter ainsi la réalisation « rapide » d'applications prenant en charge l'ensemble des aspects fonctionnels et/ou non fonctionnels nécessaires. Dans le chapitre 3 (Cf. Section 3.3), nous avons identifié quelques limitations et besoins des environnements actuels de composition de services dédiés à la réalisation d'applications à service. Les environnements de développement d'applications possèdent des besoins concernant :

- **Description des applications à haut niveau d'abstraction**. L'environnement doit permettre de décrire les applications en termes de concepts « abstraits » du domaine métier indépendamment des technologiques cibles.
- **La séparation des préoccupations**. Un environnement doit fournir un ensemble de points de vue (ou vues) permettant de séparer les concepts et fonctionnalités des utilisateurs en fonction de leurs rôles et des tâches qu'ils doivent réaliser : par exemple la conception, la réalisation, le déploiement ou la composition des briques de base de l'application.
- **L'extensibilité**. Les environnements doivent être facilement extensibles pour simplifier l'introduction de nouveaux concepts et/ou de nouvelles fonctionnalités dans les environnements existants, etc.

CADSE [Adea][EVLL08] est un projet de notre équipe de recherche ADELE[1] [ADEc]. L'objectif de CADSE est d'assister les développeurs lors de la réalisation d'applications spécifiques à un domaine particulier. CADSE est une approche utilisant les principes de l'ingénierie dirigée par les modèles (Cf. Section 2.5) et fournit un ensemble d'outils incluant des éditeurs de modèles, des validateurs de ces modèles et des générateurs capable de générer du code à partir des modèles qui décrivent une (ou plusieurs) application(s).

CADSE est conçu au dessus de l'environnement de développement intégré Eclipse [Eclc] et il étend ce dernier avec un ensemble de concepts et de fonctionnalités. L'environnement CADSE est libre et il est fournit sous la forme de *plug-ins* et de *features* pour la plateforme Eclipse. CADSE est un environnement générique offrant la possibilité de spécifier des outils dédiés à un domaine particulier et qui permettent de créer des applications spécifiques à ce domaine.

La spécification d'un domaine consiste à identifier et à définir des concepts qui sont liés au domaine et les relations entre ces concepts. Dans CADSE, les utilisateurs manipulent les concepts du domaine (modèles d'applications) pour définir leurs applications, et non pas directement les artéfacts Eclipse comme les projets, répertoires ou fichiers. CADSE augmente ainsi le niveau d'abstraction de spécification des applications. De cette façon, des utilisateurs qui ne sont pas des experts « techniques » peuvent définir des applications en termes de concepts du domaine qui leur sont familiers.

L'environnement CADSE se charge de la génération des artéfacts Eclipse et de la correspondance entre les concepts du domaine et ces artéfacts. Il gère aussi la synchronisation et assure la cohérence entre les artéfacts Eclipse et les concepts

[1] Environnements et outils pour le Génie Logiciel Industriel.

correspondant du domaine. CADSE fournit un support d'interaction aux utilisateurs à travers un ensemble d'éditeurs de modèles et de vues.

6.1.2 Architecture, concepts et modèles de base.

La réalisation d'environnements spécialisés pour un domaine n'est pas une tâche simple et facile à accomplir ; elle peut prendre du temps et elle peut être complexe. Le défi est de fournir des environnements spécialisés pour un domaine avec un temps et un coût de réalisation raisonnable. C'est la raison pour laquelle, dans l'approche CADSE nous proposons un CADSE nommé CADSEg[1] qui permet de spécifier et de générer des environnements spécialisés pour un domaine particulier et permettant de définir des applications dédiées au domaine.

CADSEg : UN EDITEUR D'ENVIRONNEMENTS SPECIALISES POUR UN DOMAINE.

CADSEg permet de spécifier un CADSE et de générer à partir de cette spécification le CADSE correspondant. Les experts du domaine expriment à l'aide de CADSEg les concepts de base du domaine applicatif dans un modèle de données. Ils peuvent aussi définir les relations de correspondance entre les concepts du modèle de données et les artéfacts Eclipse. Par exemple, une relation de correspondance peut être établie entre un concept du domaine et un fichier ou un projet Java. Les experts du domaine peuvent définir des pages de création et de propriétés des éléments (concepts) de l'application. A partir de cette description, CADSEg génére le CADSE à utiliser pour le développement d'applications qui sont conformes et spécifiques au domaine.

Chaque environnement CADSE est associé à un espace de travail (*workspace*) contenant l'ensemble des éléments du domaine et les artéfacts de l'environnement de développement intégré sous-jacent correspondant à ces éléments. Dans la version actuelle de CADSE, il s'agit des artéfacts de base de la plateforme Eclipse : fichier, répertoire, projet (Java, AspectJ, …), etc.

Selon ses objectifs, le CADSE peut générer le code exécutable de l'application et les artéfacts nécessaires à partir des modèles d'applications. Il est capable de générer l'arborescence c'est-à-dire la structure complète des projets associés aux concepts du domaine et offre la possibilité d'étendre le code généré afin de préciser le comportement souhaité de la logique applicative. CADSE gère l'espace de travail en assurant la synchronisation et la cohérence entre les éléments des modèles et les artéfacts Eclipse.

La figure suivante décrit l'architecture de l'approche CADSE :

[1] Computer Aided Domain Specific Engineering environment generator

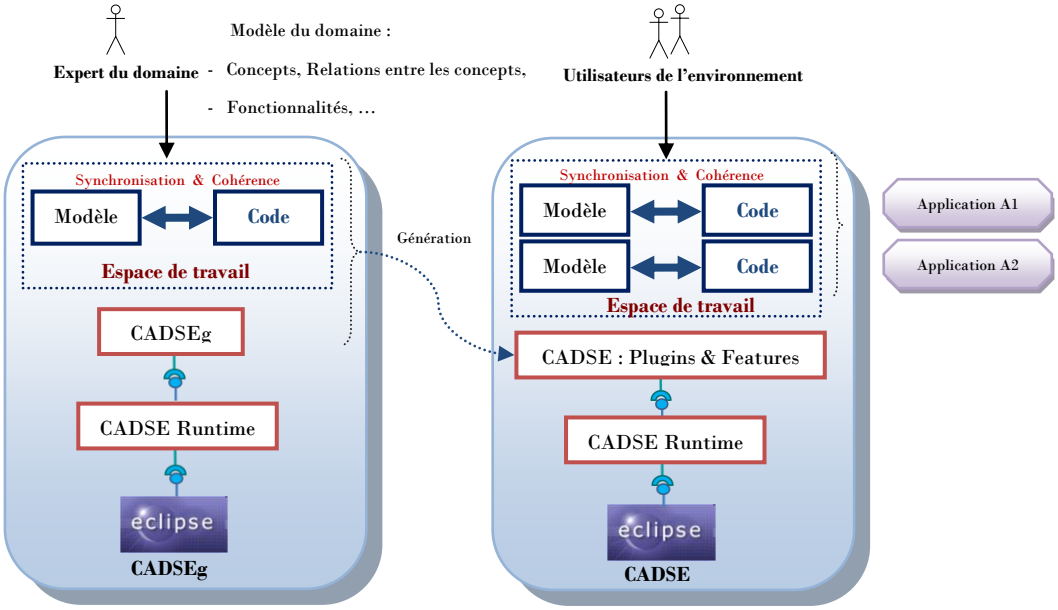

figure 26. Architecture de CADSE.

L'outil CADSEg est composé d'un ensemble de modèles et de concepts de base. Les modèles de base de CADSEg sont les suivants :

- **Modèle de données** : permet de spécifier l'ensemble des concepts du domaine et les relations existants entre eux. Pour augmenter le niveau d'abstraction et la granularité des entités, il est possible de définir dans le modèle de données des entités composites regroupant un ensemble de concepts. Les concepts du modèle de données sont spécifiques au domaine.
- **Modèle de correspondances** : permet de définir les correspondances entre les concepts du modèle de données et les artéfacts Eclipse. Un concept du domaine peut ne pas disposer de correspondance vers Eclipse.
- **Modèle des interactions** : CADSE intègre un ensemble d'éditeurs de modèles et des validateurs de ces modèles. Il offre la possibilité de définir des vues spécifiques en plus de celles qui sont fournies par défaut, par exemple la vue permettant de créer et d'afficher l'ensemble des entités du domaine qui existent dans un espace de travail. Un environnement CADSE peut avoir des vues spécialisées par exemple pour développer les entités d'une application, tester et déployer ces entités, les composer (assembler), etc.
- **Modèle d'évolution** : CADSE dispose d'un outil pour gérer les versions des entités d'un espace de travail. Cet outil étend les systèmes de gestion de versions existants comme Subversion[1] (svn). Les politiques d'évolution des entités sont définies dans le modèle d'évolution [ELV09].

La figure suivante illustre les modèles de base de CADSEg :

[1] Site officiel : http://subversion.tigris.org/

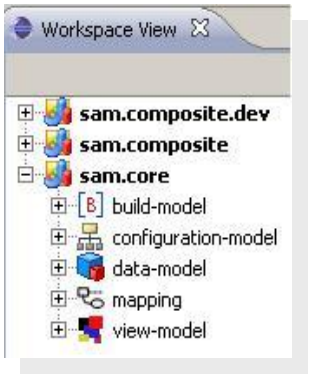

figure 27. Les modèles de base de CADSEg.

Le modèle de données permet de spécifier l'ensemble des concepts de base d'un domaine et les relations entre ces concepts. Il permet ainsi de définir un modèle du domaine. Il se base sur un ensemble de concepts tels que *ItemType* et *LinkType* :

- *ItemType* : il s'agit d'un type d'items correspondant au type d'un ensemble d'entités appelées des items qui sont définies dans un CADSE. Un *ItemType* peut définir les attributs que doivent disposer tous les items de ce type. Les attributs sont typés et peuvent être de divers types : string, entier, réel, booléen ou énumération (une liste de valeurs). Ils peuvent être obligatoires ou optionnels, en lecture seule ou non (readOnly), etc.
- *LinkType* : les items peuvent avoir des relations entre eux. Ces relations sont spécifiées entre les types de ces items avec le concept de *LinkType* (un type de lien). Deux items peuvent ainsi être liés grâce à un lien (une instance de *LinkType*). Un lien est caractérisé par la notion de cardinalité qui permet de définir le nombre (le minimum et le maximum) d'items requis comme origine ou destination du lien. Un lien (ou un type de lien) peut définir des propriétés caractérisant sa nature :
  - *Require* : permet de définir une relation de dépendance entre les items. Si la cardinalité minimale du lien vaut un « 1 » alors il s'agit d'un lien qui est obligatoire. Cette propriété est souvent utilisée pour spécifier par exemple une dépendance de compilation au sens Java.
  - *Part* : un item peut contenir d'autres items. Un lien de type « part » permet de gérer le cycle de vie des items contenus par un autre. Si deux items sont liés par un lien de type « part » alors l'item destination du lien ne peut pas exister sans l'existence de l'item source de ce lien. En plus, la destruction de la source du lien implique de détruire l'item destination du même lien.
  - *Composition* : permet de structurer la hiérarchie des items composites pouvant contenir d'autres items (grâce aux liens de type « composition »).
  - *Annotation* : indique que la destruction de l'item destination implique de détruire aussi l'item source.
  - *Group* : permet de définir une relation de groupe entre deux types (Cf. notion de groupe dans la section 4.2).

La figure 28 illustre les concepts de base de l'approche CADSE :

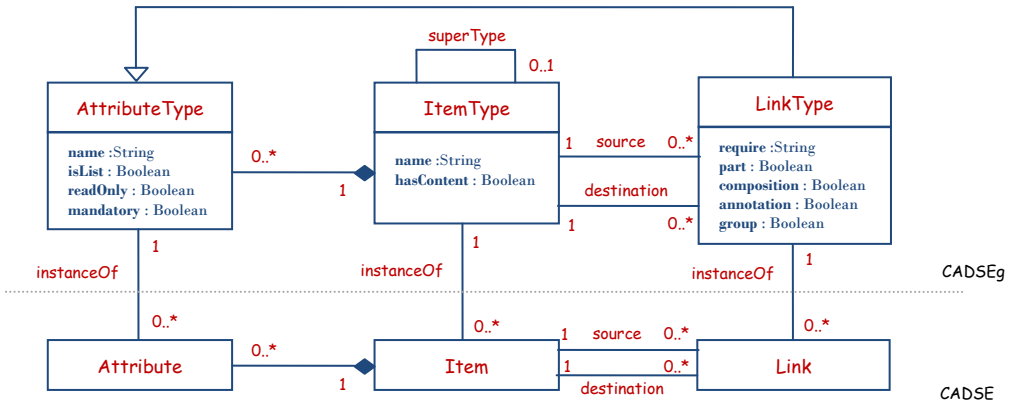

figure 28. Les concepts de base de CADSE.

Dans la figure 29, nous présentons un exemple de CADSEg permettant de spécifier l'environnement de développement de services de la machine SAM présentée ci-dessous (Cf. Section 5.1). Le CADSE SAM permet de définir et de développer des services qui sont conformes au métamodèle de la machine SAM.

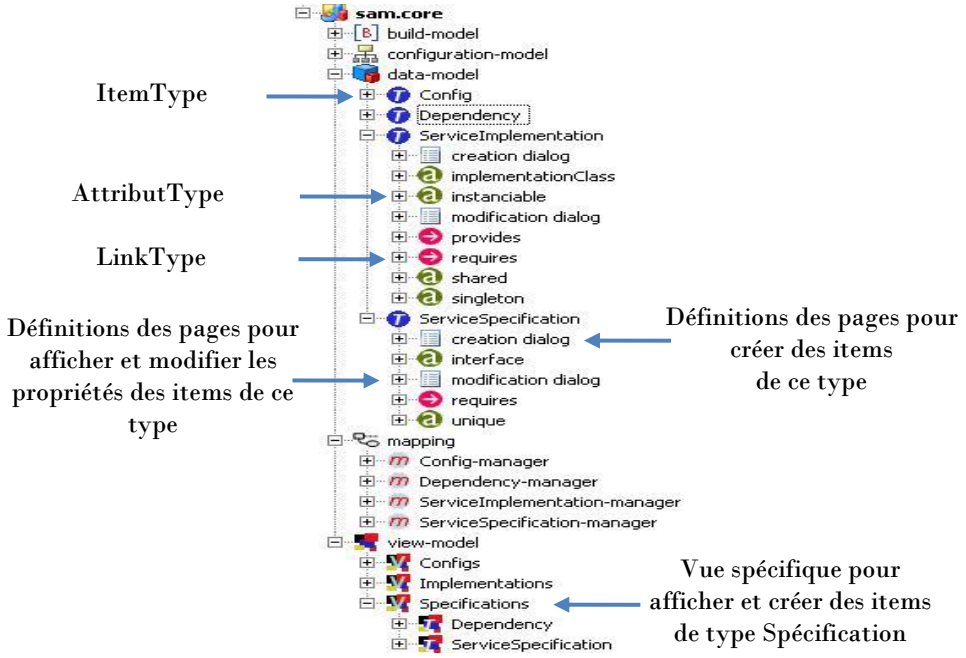

figure 29. Les concepts de base de CADSEg : l'exemple de la machine SAM.

En résumé, l'approche CADSE fournit des fonctionnalités et des mécanismes puissants, au dessus de la plateforme Eclipse, qui permettent de faciliter la réalisation d'environnements spécialisés pour un domaine spécifique. Dans CADSE, nous proposons une approche fournissant un éditeur d'environnements logiciels spécialisés afin de réduire leur temps de réalisation et diminuer les efforts de développement des environnements.

Avantages de l'approche CADSE
Augmente le niveau d'abstraction : les utilisateurs manipulent les concepts du domaine et pas nécessairement les artéfacts Eclipse.
CADSE gère la correspondance entre les concepts du domaine et les artéfacts Eclipse. Il assure aussi leur synchronisation et leur cohérence. Il est capable de générer automatiquement du code exécutable de l'application à partir des modèles d'applications.
CADSE gère l'interaction des utilisateurs en fournissant des pages de création et de propriétés, des vues, …
Un environnement CADSE est extensible par nature. On peut y ajouter de nouveaux concepts, relations, attributs, …
Le mécanisme de composition permet de définir de nouveaux environnements (qui sont complexes) à partir de ceux qui existent.

tableau 18. Avantages de l'approche CADSE.

6.2 SELECTA : CADSE DE COMPOSITION DE SERVICES.

6.2.1 SELECTA : un système modulaire et extensible.

L'une des contributions de cette thèse est la réalisation d'un atelier de génie logiciel, nommé SELECTA, dédié à la construction et à l'exécution d'applications à service. Le système SELECTA est basé sur une architecture **modulaire** et **extensible** comme l'illustre la figure 30.

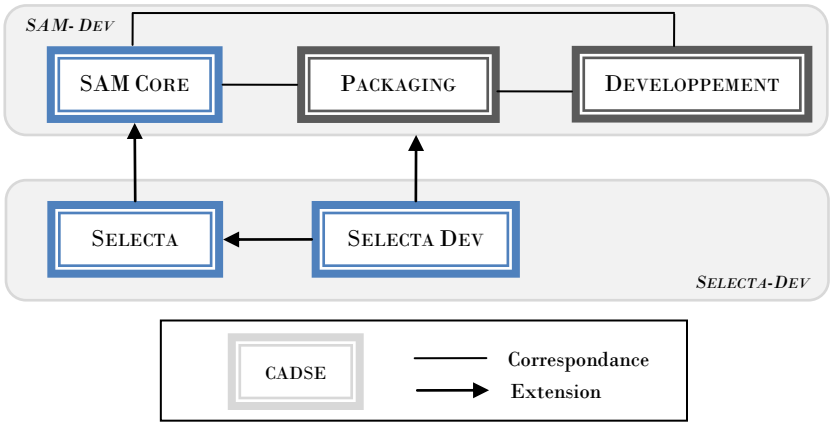

figure 30. SELECTA : une architecture modulaire.

Dans la figure 30, nous distinguons les environnements logiciels CADSE suivants :

- SAM CORE : permettant de spécifier des services (spécifications et/ou implémentations) et des relations conformes au métamodèle du système SAM défini dans la figure 21 (Cf. Section 5.1).
- DEVELOPPEMENT : permettant de définir les artéfacts de développement (modules, packages, etc.) représentant les principaux concepts du noyau SAM CORE lors de la phase de développement. Grâce au mécanisme de composition d'environnements offert par CADSE (Cf. Section 6.1), nous définissons les correspondances entre les concepts de SAM CORE et ceux de l'environnement DEVELOPPEMENT. Les concepts de spécification et d'implémentation de service

sont par exemple représentés, pendant la phase de développement, sous la forme de modules. Concrètement, un module peut par exemple correspondre à un projet Java, un fichier JAR[1] ou un bundle OSGi [OSG05a].

- PACKAGING : permettant de définir les concepts d'unités de déploiement, de packaging et de transfert qui sont pertinents pendant la phase de packaging et de déploiement des services ou des applications logicelles. Les concepts de SAM CORE et de l'environnement de DEVELOPPEMENT auront des correspondances vers les concepts du CADSE de PACKAGING afin de spécifier la manière de les déployer sur une plateforme à service cible, par exemple OSGi [OSG05a] ou iPOJO [EHL07].

La composition des environnements logiciels présentés ci-dessus (SAM CORE, DEVELOPPEMENT et PACKAGING) donne comme résultat un nouvel environnement CADSE qui permet de définir, de développer et de déployer des services conformes à SAM. Cet environnement est appelé SAM-DEV.

- SELECTA : SELECTA est un logiciel CADSE qui étend l'environnement SAM CORE et permet de définir des services composites conformes au métamodèle de la figure 25 décrivant les concepts de base de notre approche de composition (Cf. Chapitres 4 et 5). Cet environnement intègre un ensemble d'outils concernant par exemple la définition et la gestion des contraintes de services ou la sélection des services requis par un composite.
- SELECTA-DEV : étend les environnements SELECTA et SAM-DEV afin d'intégrer les concepts correspondants aux services (composites ou atomiques) lors des phases de développement et de composition. Cette séparation offre la possibilité de réutiliser, lors de la phase d'exécution, les concepts, mécanismes et algorithmes réalisés dans les environnements SAM CORE et SELECTA.

La figure suivante illustre les relations entre les concepts SAM CORE et de DEVELOPPEMENT :

[1] Java Archive (JAR) permet de regrouper plusieurs fichiers en un seul fichier archive. La documentation officielle est disponible sur http://java.sun.com/docs/books/tutorial/deployment/jar/

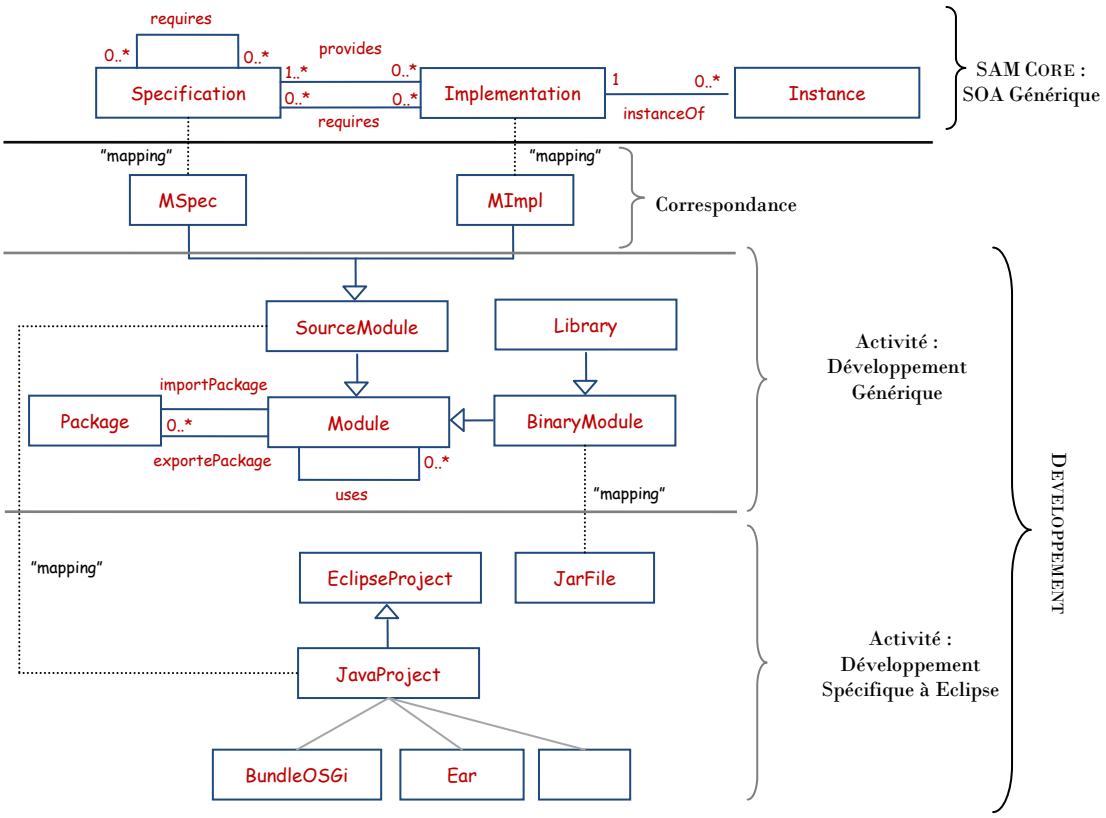

figure 31. Relations entre les concepts de SAM et de développement.

Durant la phase de développement, le concept de service (spécification ou implémentation) est représenté sous forme de module. Nous distinguons deux types de modules : les modules sources (par exemple le code source Java) et les modules binaires (par exemple, une librairie). Un module peut contenir des packages et peut en importer ou exporter (au sens Java du terme). Un module peut dépendre d'autres modules. Les dépendances entre les modules sont déduites à partir des celles des services correspondants à ces modules, par exemple si un service *S1* requière un autre *S2* alors les modules représentant ces services seront liés par une relation « *uses* » exprimant une dépendance entre les modules.

Nous séparons les concepts liés à la phase de développement en deux couches. La première couche spécifie les concepts (ou artéfacts) de développement d'une manière générique (module, package, …) c'est-à-dire indépendamment des plates-formes de développement, comme exemple nous pouvons citer la plateforme Eclipse [Eclc]. La deuxième couche définit les artéfacts de développement (projet Java, fichier simple, fichier JAR, …) qui sont liés à la plateforme de développement Eclipse. Des relations de correspondances seront établies ensuite entre les concepts de ces deux couches. Cette séparation nous offre la capacité de développer des services sur d'autres plates-formes qu'Eclipse.

Idéalement, les couches présentées ci-dessus devraient être modélisées avec différents environnements logiciels CADSE qui pourront être composés par la suite pour fournir un environnement de développement de services spécialisé par exemple pour la plateforme Eclipse. Dans la version actuelle, les deux couches de l'environnement de développement sont spécifiées dans le même environnement CADSE appelé DEVELOPPEMENT.

La figure 32 introduit les correspondances entre les concepts de développement et de packaging :

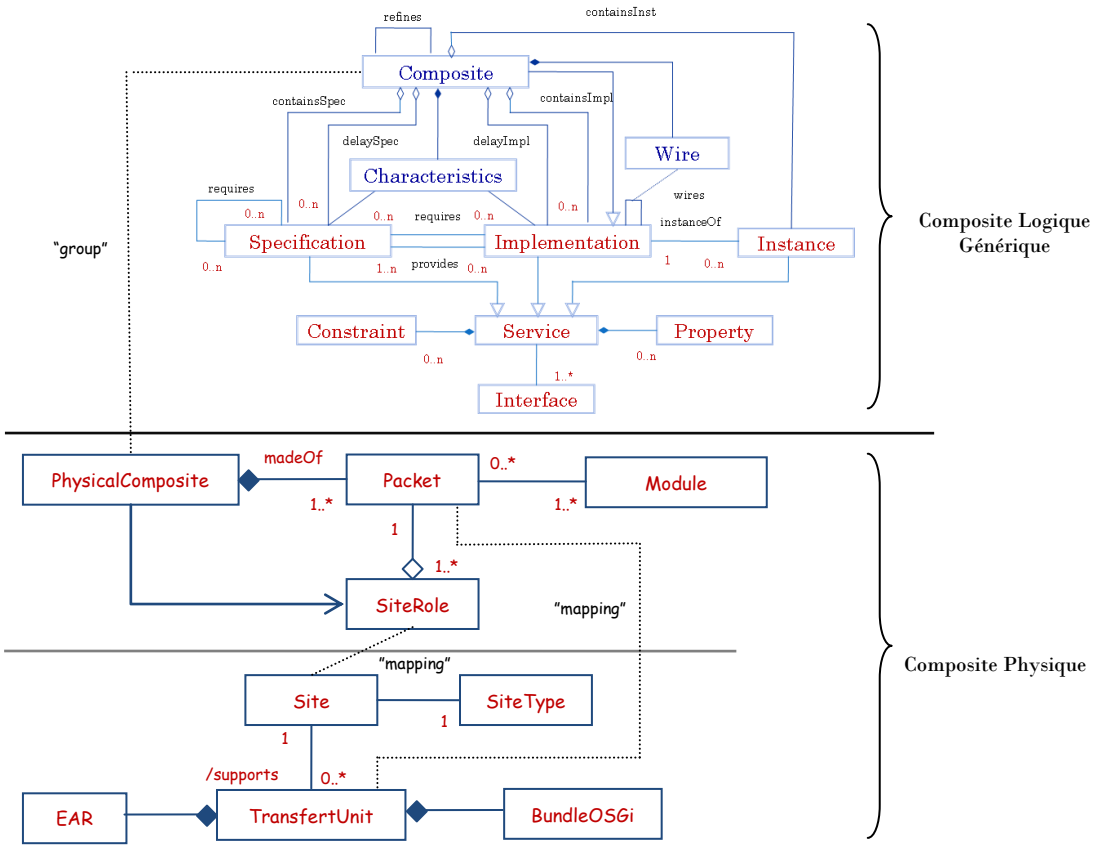

figure 32. Correspondances entre les concepts de développement et de packaging.

Les modules sont regroupés sous la forme d'entités logiques appelées Packet correspondant aux unités de packaging et de transfert sur une plateforme cible à services. Un *Packet* peut contenir des *bundles* comme ceux reconnus par les plates-formes OSGi ou iPOJO et/ou des fichiers EAR[1] ou WAR[2] par exemple pour une application Web. Dans notre système, une unité de transfert est un fichier ZIP contenant des ressources associées aux services (bundles, fichiers binaires ou exécutables) et optionnellement des informations (décrites sur des fichiers XML) requises par les services contenus dans le fichier ZIP. Dans la couche *Composite Logique Générique*, les composites sont exprimés uniquement en termes de concepts génériques de SAM CORE et de notre extension SELECTA. Un composite « logique » est associé à un ensemble de composites « physiques » par une relation « *group* » qui permet de transférer les valeurs des attributs définis par le composite « logique » vers chaque composite « physique » (Cf. la notion de groupe d'équivalence décrite dans la section 4.2). Par conséquent, les composites « physiques » contiennent tous les attributs et les relations

[1] Enterprise ARchive

[2] Web Application aRchive

spécifiés par le composite « logique », et ajoutent aussi des informations concernant la structuration des modules et les sites sur lesquels ils vont s'exécuter.

Dans la suite de cette section, nous introduisons l'architecture et les choix techniques effectués lors de la réalisation du système SELECTA :

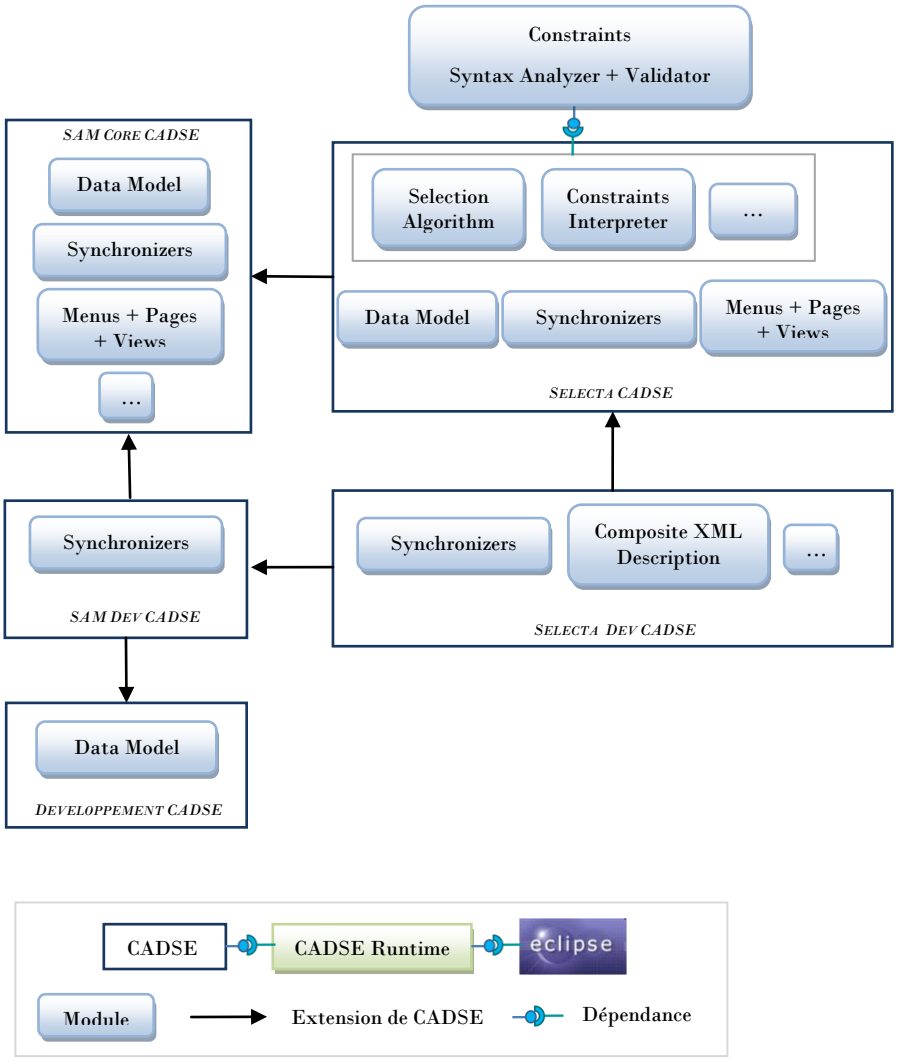

figure 33. Architecture globale de SELECTA.

LE MODULE POUR LA VALIDATION SYNTAXIQUE DES EXPRESSIONS DE CONTRAINTES.

Le module *Constraints Syntax Analyser & Validator* (CSAV) exprime formellement (avec le métalangage EBNF[1]) la syntaxe de notre langage d'expression de contraintes de services (Cf. Section 4.3.2), et fournit un analyseur et un validateur syntaxique de ces expressions de contraintes. Nous avons choisi l'outil JavaCC[2] (précisément la version 5.0)

[1] Extended Backus-Naur Form

[2] Java Compiler Compiler – Site officiel : https://javacc.dev.java.net/

pour la mise en œuvre du langage de contraintes que nous proposons dans SELECTA. L'outil JavaCC est un générateur de parseurs pour Java qui prend en entrée la spécification de la grammaire (un fichier possédant l'extension *jj*) de notre langage de contraintes et génère un parseur (un programme Java) pour la validation syntaxique des expressions de contraintes. Le module CSAV est fourni sous la forme de *plugin* Eclipse et est réutilisable en dehors de SELECTA dans la mesure où la syntaxe du langage est générique et indépendante du système SAM.

ENVIRONNEMENT SAM CORE.

L'environnement SAM CORE est constitué de plusieurs modules tels que :

- ***Data Model*** (DM) contenant la description du métamodèle SAM (les concepts, les propriétés et les relations entre les concepts).
- ***Menus, Pages & Views*** (MPV) spécifiant un ensemble de menus par exemple pour la création de services (spécifications ou implémentations), des pages comme celles permettant de créer des services ou d'afficher leurs propriétés, et de diverses vues pour regrouper par exemple l'affichage des services en fonction de leurs types. Nous avons défini des vues spécifiques pour visualiser les spécifications ou les implémentations de service et une vue globale pour la visualisation de tous les services qui existent dans un espace de travail.
- ***Synchonizers*** contenant un ensemble de synchronisateurs qui permettent de gérer la notion de groupe d'équivalence au sein de l'environnement SAM CORE. En effet, ces synchronisateurs offrent la possibilité par exemple de transférer les définitions de propriétés (et des valeurs communes) spécifiées par une spécification de service (S) lors de la création d'une implémentation de service (I) fournissant les fonctionnalités de S ou de propager les dépendances de services définies par S au niveau de l'implémentation I (Cf. Section 5.1).

ENVIRONNEMENT DEVELOPPEMENT.

Comme souligné précédemment, l'environnement nommé DEVELOPPEMENT définit les artéfacts de base qui sont liés à la phase de développement de services. Ces artéfacts et les relations entre eux sont spécifiés dans un modèle de données CADSE appelé *Data Model*.

ENVIRONNEMENT SAM-DEV.

SAM-DEV étend les environnements SAM CORE et DEVELOPPEMENT en établissant les relations de correspondances entre les concepts de ces deux environnements CADSE. SAM-DEV ajoute des synchronisateurs permettant de générer les artéfacts de développement correspondant aux concepts de spécification ou d'implémentation de service. Lors de la création d'une spécification de service *S*, le synchronisateur génère un projet Java contenant une interface Java et un fichier de description des propriétés définies par *S*. Lors de la création d'une implémentation de service *I* fournissant la spécification *S*, le synchronisateur génère un projet Java contenant une classe qui implémente (au sens Java du terme) les fonctionnalités définies dans l'interface fournie par le service *S* et fixe les dépendances entre les projets associés à *S* et *I* (gestion de la *ClassPath*).

ENVIRONNEMENT SELECTA.

Le système SELECTA est constitué de plusieurs modules comme :

- ***Data Model*** qui étend le module DM de l'environnement SAM CORE en intégrant les concepts (composite, constraint, characteristic, wire, etc.) du métamodèle de la figure 25 décrivant notre extension qui est dédiée à la composition de services (Cf. Chapitre 5).
- ***Menus, Pages & Views*** (MPV) permettant d'ajouter un ensemble de menus et de vues par exemple pour l'exécution de l'algorithme de sélection des services requis par un composite ou la réalisation des sélections retardées. Ce module intègre aussi de nouvelles pages comme celles permettant de visualiser dans une fenêtre la description d'une configuration de composite et son état (complet, retardé et/ou incomplet) obtenus après l'exécution de l'algorithme de sélection de services (Cf. figure 38). Cette fenêtre indique les services sélectionnés, les services manquants, les connecteurs (wires) entre les services, les contraintes qui ont permis la sélection de ces services, l'état du composite, etc.
- ***Synchronizers*** ajoute de nouveaux synchronisateurs, par exemple celui qui permet d'enlever les connecteurs (wires) entre les services (implémentations) réalisées lors de l'exécution de l'algorithme de sélection des services d'un composite dans le cas où une implémentation utilisée par ce composite n'existerait plus dans le repository de services.
- ***Constraints Interpreter*** (CI) dépend du module CSAV et permet de spécifier la sémantique des expressions de contraintes. Le module CI est spécifique au système SAM parce qu'il connaît la sémantique des concepts (spécification, implémentation, …), des attributs (*unique*, *shared*, …) et des relations (*requires*, *provides*, …) définis dans le métamodèle SAM (Cf. Section figure 21). Nous avons personnalisé, à ce niveau, les messages d'erreurs générés par l'outil JavaCC lors de la validation des expressions de contraintes, et nous avons ajouté aussi d'autres types de messages, appelés des « warning » qui permettent d'alerter l'utilisateur si les services ou les propriétés (des services) utilisés dans les expressions de contraintes n'existent pas (ou ne sont pas encore définis) dans le repository de services.
- ***Selection Algorithm*** (SA) réalise la sélection des services requis par un composite en se basant sur sa description (les services déjà contenus ou sélectionnés, les contraintes définies par le composite et celles des services contenus, les services retardés de sélection, etc.). Le module SA utilise l'API[1] du CADSE SELECTA afin de connaître la sémantique de ses concepts et des relations.

ENVIRONNEMENT SELECTA-DEV.

SELECTA-DEV contient un module ***Synchronizers*** qui étend le module *Synchronizers* défini dans l'environnement SAM-Dev gérant les correspondances entre les concepts du système SAM et les artéfacts de développement qui leurs sont associés. L'extension consiste à générer un projet Eclipse contenant un fichier de description du

[1] Application Programming Interface

composite qui sera utilisé par exemple lors du déploiement. En fait, il s'agit d'un fichier XML dont la structure est bien définie. Nous utilisons la technologie JET[1] pour modéliser le « *template* » de génération et automatiser la génération de la description XML du composite à partir du modèle CADSE. Le module ***Composite XML description*** contient le « *template* » et le code lié à la génération de la description du composite. L'annexe B décrit un exemple de fichier de description d'une configuration de notre exemple de système d'alarme.

6.2.2 Un exemple : mise en œuvre du système d'alarme dans SELECTA.

Considérons notre exemple de système d'alarme d'une usine de production que nous avons décrit dans l'annexe C. Le cas d'utilisation choisi consiste à utiliser un ensemble de capteurs (par exemple, des capteurs de température et d'humidité) pour collecter des mesures de données qui sont liées à l'environnement physique de l'usine à surveiller. Les données collectées sont ensuite agrégées, analysées et si nécessaire sauvegardées sur des supports comme une base de données ou fichier électronique afin de garder l'historique des mesures reçues. Des traitements (ou des actions) spécifiques, tels qu'une alerte vocale ou un envoie de messages SMS ou électroniques, peuvent ensuite être réalisés en cas de détection d'anomalies sur les données collectées.

L'application à réaliser est modélisée dans le système SELECTA comme un composite et dispose de nombreux besoins tels que :

- **Gestion de l'hétérogénéité** : les collecteurs de données (les capteurs) peuvent être hétérogènes dans la mesure où ils peuvent être réalisés dans diverses technologies ou peuvent utiliser différents protocoles de communication tels que ZigBee ou UPnP.
- **Gestion de la cohérence et de la complétude** : les services utilisés par l'application doivent être compatibles entre eux et respecter les contraintes et les propriétés spécifiées par le composite (l'application). Tous les services requis par l'application doivent être sélectionnés (ou manuellement spécifiés), s'il y en a qui ne sont pas encore disponibles l'application devrait être dans un état incomplet. En d'autres termes, le système doit gérer la complétude et garantir la cohérence de l'application.
- **Gestion de l'évolution et du dynamisme** : les services utilisés par l'application peuvent évoluer et peuvent apparaître ou disparaître à tout moment. Dans ce cas, l'application devrait être capable d'utiliser d'autres services (de nouveaux capteurs par exemple) qui sont disponibles et qui correspondent à ses critères (contraintes et propriétés).

Le système SELECTA facilite la création de services en générant les artéfacts de développement nécessaires à leur réalisation concrète. Le développeur est chargé de réaliser la mise en œuvre concrète des fonctionnalités attendues par les services. Les figures suivantes décrivent la définition des fonctionnalités fournies par la spécification de service *MeasureAggregator* et le code métier de l'une de ces implémentations nommée *MAa*

[1] Java Emitter Template

fournissant les fonctionnalités spécifiées par la spécification *MeasureAgregator* (Cf. Annexe C).

```
package fr.lig.adele;

import java.util.List;

public interface MeasureAggregator {

        /**
            @generated
        */

        String AS_ID = "MeasureAggregator";

        // Ruturns the maximum required measure number to be collect for aggreagating
        public int getRequiredMeasureNumber ();

        // Returns a Temperature measure
        public int getTemperature ();

        // Returns a Humidity measure
        public int getHumidity ();

        // Returns a list containing the average of the temperature and humidity measures
        public List <Double> calculAverage ();
}
```

figure 34. Exemple de l'interface *MeasureAggregator*.

```
package fr.lig.adele.selecta.demo;

import java.util.ArrayList;
import java.util.List;

import fr.lig.adele.HumiditySensor;
import fr.lig.adele.MeasureAggregator;
import fr.lig.adele.TemperatureSensor;


public class MAa implements MeasureAggregator {

        /**
         * @generated
         */

        // It requires a Humidity sensor
        HumiditySensor HumiditySensorDependency;

        /**
         * @generated
         */

        // It requires a Temperature sensor
        TemperatureSensor TemperatureSensorDependency;

        // the maximum number of measures to collect for aggregating
        int measure_Nunber_Max = 5;

        // the initial value; will be incremented in the calculAverage() method
        int current_Measure_Number = 0;

        … …

        @Override
        public int getHumidity() {
            return HumiditySensorDependency.getHimidity();
        }

        @Override
        public int getTemperature() {
            return TemperatureSensorDependency.getTemperature();
        }

        public int getCurrent_Measure_Number() {
            return current_Measure_Number;
        }

        @Override
        public List<Double> calculAverage() {

        … … …

        }

        @Override
        public int getRequiredMeasureNumber() {
            return measure_Nunber_Max;
        }
}
```

figure 35. Extrait du code du service *MAa*.

Le service (implémentation) *MAa* requiert les spécifications de service *HumiditySensor* et *TemperatureSensor* pour la récupération des mesures de données liées à la température et à l'humidité de l'usine (Cf. Annexe C). Les variables pour la manipulation de ces dépendances sont générées automatiquement dans le squelette du code de mise en œuvre du service *MAa*. L'implémentation *MAa* offre la fonctionnalité de calcul de la moyenne (la méthode *calculAverage*) de **cinq** mesures reçues (température et humidité).

Les services réalisés dans l'environnement SELECTA peuvent être utilisés par diverses applications et sont disponibles dans un ou plusieurs *repository* de services. La figure suivante illustre notre exemple de *repository* dans SELECTA (Cf. Annexe C) :

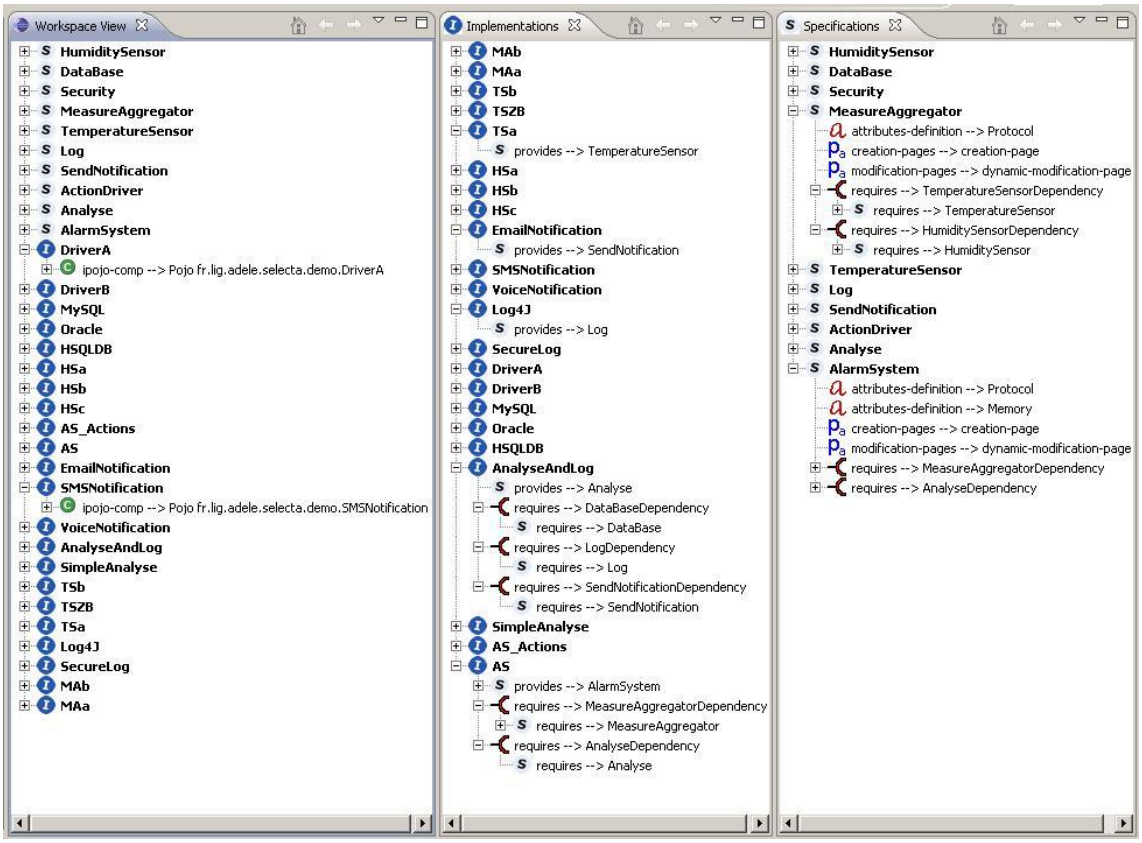

figure 36. Mise en œuvre du *repository* de services dans SELECTA.

Notre exemple de système d'alarme :

- fournit les fonctionnalités définies par la spécification AlarmSystem,
- contient initialement la spécification SendNotification pour l'envoie d'alerte en cas de détection d'anomalies sur les mesures de données collectées,
- les services qui sont requis et utilisés par le système doivent remplir un ensemble de contraintes. La figure 37 illustre les contraintes du système d'alarme en utilisant notre éditeur de contraintes. L'éditeur de contraintes permet de spécifier les contraintes du composite (décrivant le système), de détecter les erreurs syntaxiques et de signaler des messages de types « warnings » si les services et/ou les attributs utilisés dans les expressions de contraintes sont inconnus dans le repository de services.

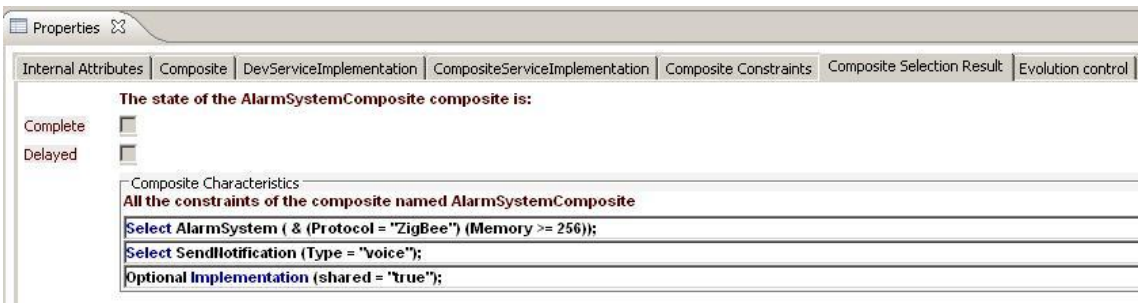

figure 37. Contraintes du système d'alarme dans SELECTA.

Nous fournissons un ensemble d'outils dans le but de faciliter et de guider la sélection des services requis par un composite et qui sont conformes à ses critères (contraintes et propriétés). La figure 38 décrit un exemple de configuration du système d'alarme obtenu après l'exécution de notre algorithme de sélection de services à partir de sa description et de l'état du *repository* de services (les services contenus par le composite, les contraintes définies, les services qui sont disponibles et qui correspondent à ses critères, etc.).

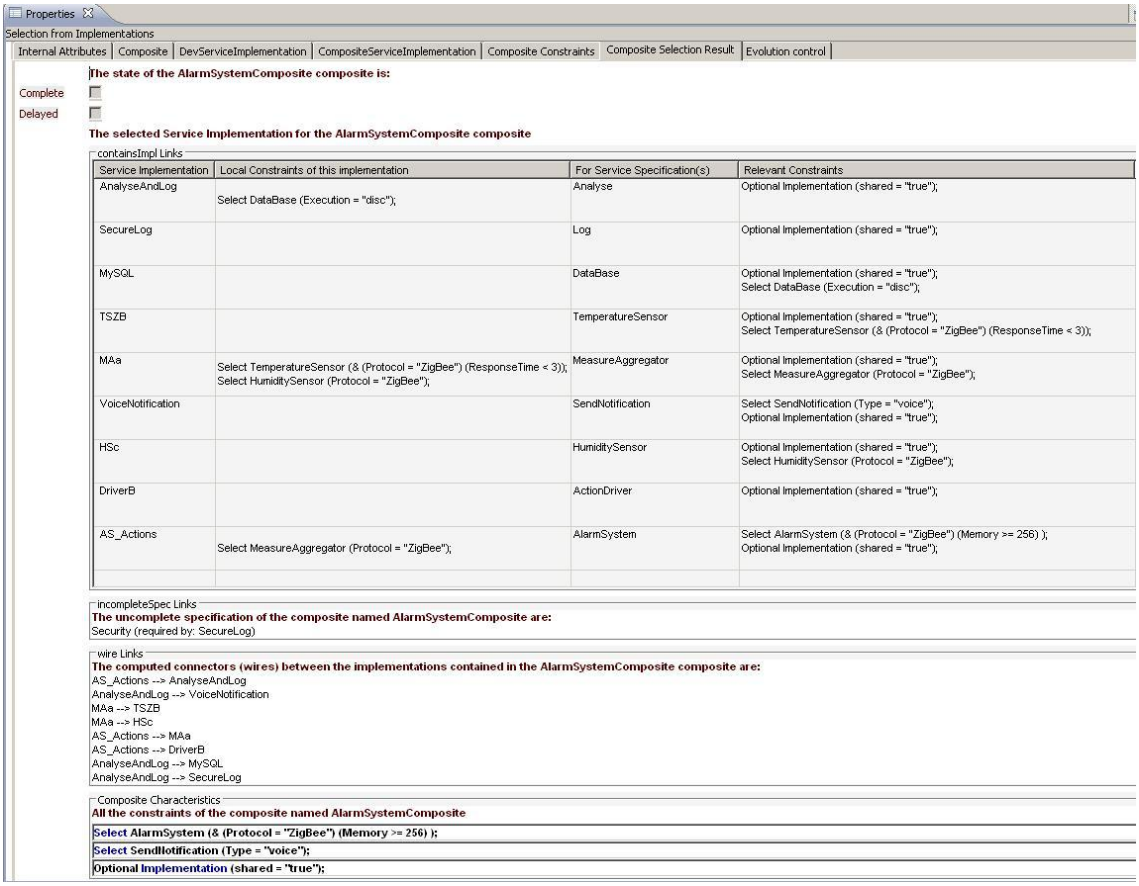

figure 38. Une configuration du système d'alarme dans SELECTA.

6.3 PLATEFORME D'EXECUTION DE COMPOSITES.

Nous avons introduit dans la section 5.3 notre plateforme d'exécution de composites nommée SELECTA RUNTIME. Dans cette section, nous présentons l'architecture générale et les technologies et/ou les plates-formes sur lesquelles se base la plateforme SELECTA RUNTIME. La figure 39 décrit l'architecture de SELECTA RUNTIME :

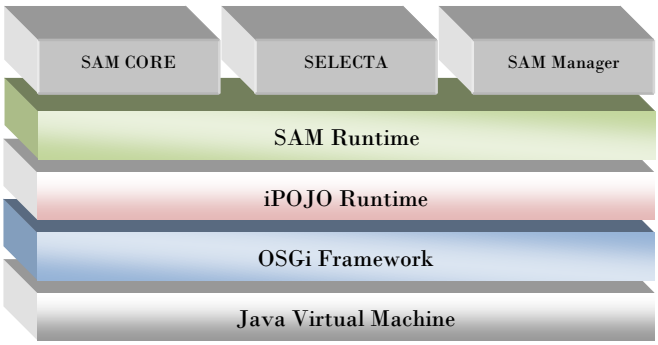

figure 39. Architecture de SELECTA RUNTIME.

Le principal socle de SELECTA RUNTIME est la plateforme d'exécution SAM RUNTIME (Cf. Section 5.1). Cette dernière s'éxécute au desssus de la plateforme OSGi (par exemple, Felix[1] ou Equinox[2]) et requièrt la plateforme d'exécution qui est fournie par le modèle iPOJO afin de bénéficier de ses avantages (et de ceux de la plateforme OSGi) tels que :

- la réalisation d'une plateforme (SAM RUNTIME) en termes de composants orientés services en se basant sur un modèle extensible et supportant le dynamisme,
- un support facilitant le déploiement de composants et la gestion du cycle de vie de leurs instances,
- la gestion des dépendances entre les services est assurée par le modèle iPOJO.

Nous utilisons les modèles et les mécanismes définis dans les CADSE SAM CORE et SELECTA. Il a fallut cependant effectuer une adaption d'une partie du code afin de le rendre indépendant de la plateforme Eclipse (parce que CADSE est une extension de la plateforme d'Eclipse) et de pouvoir l'exécuter directement sous une plateforme mettant en œuvre la spécification OSGi, nous avons choisi la plateforme Felix comme environnement cible.

Le module *SAM Manager* se charge d'une part d'observer l'état de la plateforme (les services qui y tournent, ceux qui sont disponibles, etc.) et d'audre part d'assurer la cohérence et la validité de l'exécution entre la description de l'application (spécifiée comme un composite) et l'état courant de la plateforme. Il se charge aussi si nécessaire de réaliser des actions (changements) sur la plateforme. Nous signalons que les travaux sur la gestion de la conformité entre le modèle de l'application et le modèle décrivant l'état réel de la plateforme sont en cours de réalisation. Ils ne seront pas traités dans ce manuscrit.

[1] Site officiel : http://felix.apache.org/

[2] http://www.eclipse.org/equinox/

6.4 EXPERIMENTATIONS ET VALIDATION.

Nous avons validé les principes et les notions proposées dans notre approche par la mise en œuvre d'un prototype nommé SELECTA. Nous introduisons dans cette section quelques chiffres concernant la réalisation du système SELECTA. Nous présentons tout d'abord la taille, en termes du nombre de lignes de code et de l'effort de documentation éffectué, dans les différents environnements logiciels qui constituent SELECTA et la qualité du code. Nous montrerons ensuite le gain de productivié pouvant être obtenu en utilisant notre système.

Nous avons utilisé l'outil Sonar (la version 1.6) qui est disponible en open source permettant de mesurer et de gérer la qualité du code. Les figures suivantes (figure 40 et figure 41) illustrent la qualité du code des environnements SAM DEV et SELECTA.

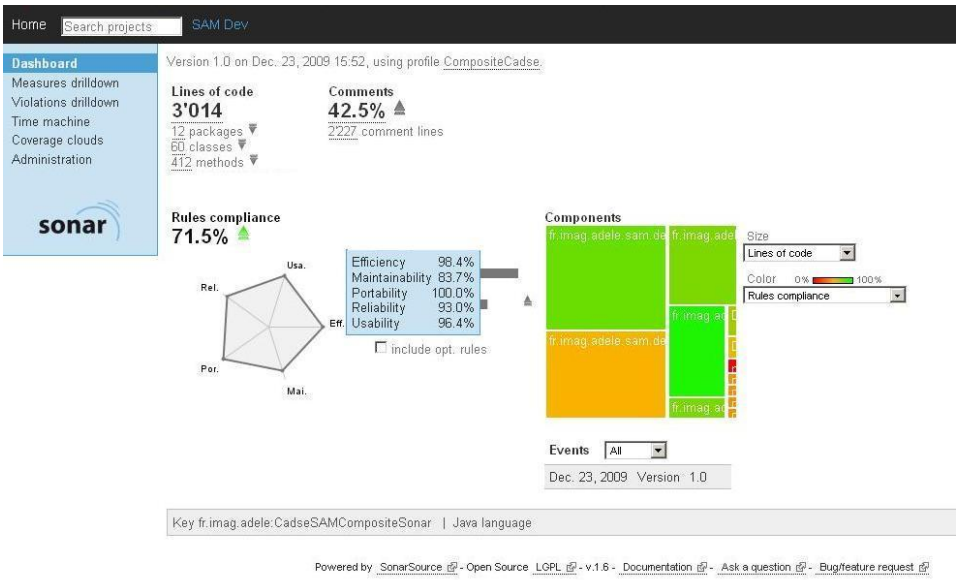

figure 40. Quelques chiffres sur le système SAM CORE.

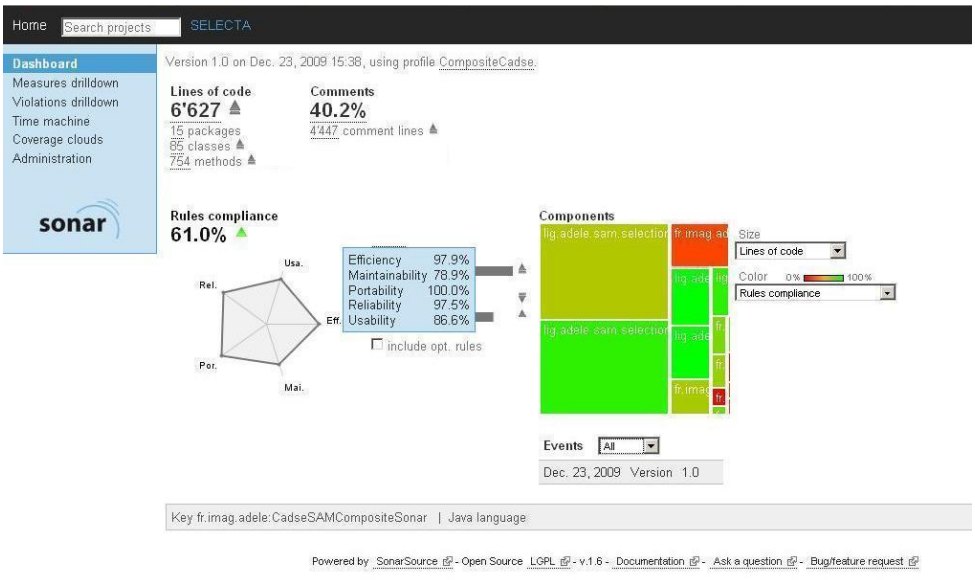

figure 41. Quelques chiffres sur le système SELECTA.

Le tableau suivant résume le nombre de lignes de code des environnements logiciels proposés dans cette thèse :

Environnements	Nombre de lignes de code	Commentaire / Documentation
SAM CORE	876 lignes	910 lignes
SELECTA	6627 lignes	4447 lignes
SAM-DEV	3014 lignes	2227 lignes
SELECTA-DEV	789 lignes	167 lignes
Autres	4949 lignes	3777 lignes

tableau 19. Nombre de lignes de code des environnements.

Le résultat du tableau 19 montre l'effort de documentation de nos différents environnements logiciels, ce qui peut faciliter leur extension dans la mesure où nous nous basons sur une démarche modulaire et qui favorise l'extensibilité. Nous notons dans les figures précédentes un effort de maximisation du respect des qualités de code en génie logiciel comme :

- **La portabilité (portability)**. Cette qualité consiste à minimiser l'effort pour porter le code dans un autre environnement logiciel. Notre code est portable à cent pour cent (100%) dans des environnements réalisés avec le langage Java.
- **La maintenabilité (Maintainability)**. Cette qualité consiste à minimiser l'effort à réaliser pour détecter et corriger les annomalies (bugs). Nous avons obtenu par exemple 83.7 % et 78.9 % respectivement pour les environnements SAM-DEV et SELECTA. Nous avons fait un effort de modularité et de consistance des différents environnents, ce qui facilite aussi leur maintenabilité.
- **La maniabilité (Usability)**. Cette qualité consiste à minimiser l'effort nécessaire pour l'apprentissage de l'outil, la mise en œuvre des données en entrée, l'analyse et l'exploitation des résultats fournis par l'outil. Nous fournissons diverses interfaces graphiques aux utilisateurs permettant de faciliter la saisie, le formatage et la visualisation des données. En plus, nous avons une documentation de nos outils et nous avons par exemple les chiffres 86.3 %, 96.4 % et 86.6 % pour respectivement les environnements SAM CORE, SAM-DEV et SELECTA.
- **La réutilisabilité et l'extensibilité**. Nos environnements peuvent être étendus « facilement », ceci s'explique par la démarche choisie qui consiste à l'entension et la composition de différents environnements afin de construire un autre plus complexe et spécialisé. Le code des environnements est réutilisable dans d'autres contextes ou environnements.

En réalisant manuellement une configuration de notre exemple de système d'alarme, nous nous rendons compte souvent que :

- il y a un grand risque d'oublier de choisir des services nécessaires ou de résoudre les dépendances de services choisis. En d'autres termes, la propriété de la complétude du composite n'est toujours pas garantie,
- il y a un grand risque de choisir des services qui ne sont pas compatibles entre eux parce qu'ils ont des contraintes ou des propriétés incompatibles ou bien qu'ils ne remplissent pas les critères du composite à réaliser. En d'autres termes, la propriété de la cohérence du composite ne peut pas toujours être garantie,

- on ne sait pas prendre en compte les sélections retardées de services ; À quel moment les réaliser ?
- on ne sait pas justifier les choix de services effectués (sur quels critères),
- on ne sait pas identifier facilement quelle implémentation a été choisie pour quelle spécification de service,
- on se sait pas évaluer l'état du composite (complet, incomplet, retardé, conflit).

En résumé, l'automatisation du processus de la composition permet d'augmenter le gain de la productivité des configurations de composites. Notre système facilite la réalisation de configurations de composites en garantissant les propriétés identifiées précédemment.

Notre solution permet de réduire le coût de la réalisation de l'environnement logiciel. Par exemple dans les environnements SAM CORE et SELECTA, environ 720 et 1554 lignes de code ont été générées respectivement parmi les 876 et 6627 soit 82 % du code du noyau SAM CORE et 23 % du code de SELECTA. Le code généré est déduit à partir des modèles des environnements CADSE.

6.5 SYNTHESE.

Nous avons introduit l'architecture globale et les propriétés essentielles de notre système SELECTA et de sa plateforme d'exécution qui lui est associée. Nous avons fait un effort de modélisation en se basant sur une démarche de génie logiciel. En effet, le système SELECTA est construit sur une architecture modulaire et extensible, ce qui permet de respecter les principes de base du génie logiciel dans le domaine de la conception de systèmes logiciels.

Ce chapitre nous a permis de présenter aussi les choix techniques effectués lors de la conception du système SELECTA, de mettre en œuvre un exemple d'application en utilisant SELECTA et d'introduire les expérimentations réalisées au sein de notre système. Le tableau suivant résume les propriétés de notre système :

Propriétés du système SELECTA

Modularité	SELECTA est un système modulaire, ce qui permet de réutiliser une partie de son architecture dans d'autres systèmes. Nous proposons une approche dirigée par les modèles : la modularité est guidée par différents métamodèles.
Extensibilité	SELECTA étend différents environnements logiciels CADSE et peut être étendu par d'autres CADSE.
Séparation des préoccupations	Les aspects, les concepts et les mécanismes liés à chaque étape du cycle de vie sont spécifiés dans différents environnements. En d'autres termes, SELECTA est construit par composition de différents environnements. En plus, le système SELECTA définit plusieurs vues en fonction des tâches et des rôles des développeurs d'applications.

tableau 20. Quelques propriétés du système SELECTA.

7

CONCLUSION ET PERSPECTIVES.

Nous concluons ce manuscrit par une synthèse des contributions de la thèse fournissant des réponses par rapport aux besoins et défis de la composition en générale et de la composition de services en particulier. Notre proposition permet la construction « rapide » et l'exécution d'applications fléxibles et dynamiques. Nous introduisons ensuite les travaux futurs qui annoncent les perspectives de recherche de ce travail.

7.1 SYNTHESE DES CONTRIBUTIONS.

Pour faire face aux besoins et défis de la composition, nous avons proposé dans cette thèse un système nommé SELECTA dont l'objectif est de faciliter la réalisation et l'exécution d'applications flexibles et dynamiques. Nous avons introduit notre proposition en deux temps : dans un premier temps nous avons décrit les aspects, les principes et les mécanismes génériques de notre approche composition s'appliquant sur tout métamodèle (Cf. Chapitre 4), puis nous présenté la mise en œuvre de notre approche dans le cadre d'une composition de services (Cf. Chapitre 5). Les besoins et les défis d'une composition concernent par exemple :

- l'augmentation du niveau d'abstraction de la spécification de la composition,
- l'automatisation du processus de la réalisation de la composition en fournissant des environnements logiciels,
- la construction incrémentale de composites,
- la définition de langages et de modèles permettant de spécifier les contraintes des éléments,
- la définition de composites flexibles par intention c'est-à-dire à partir de leurs propriétés et contraintes,
- l'automatisation de la sélection des éléments d'un composite,

- la construction « automatique » de configurations de composites qui sont valides et cohérentes tout en gérant leur complétude (tous les éléments nécessaires sont présents) et les sélections retardées, etc.

Nous avons identifié diverses caractéristiques d'une composition afin de présenter la taxinomie utilisée tout au long du manuscrit. Une composition peut donc être définie d'une manière *statique*, *retardée*, *dynamique*, *incrémentale* ou *autonomique* (Cf. les définitions proposées dans la section 3.2). Cette thèse apporte une solution permettant de réaliser ces différentes compositions en fournissant des modèles, des langages, des algorithmes et des outils logiciels appropriés, de la conception à l'exécution des applications. Nous insistons sur le fait qu'actuellement SELECTA ne supporte pas la composition autonomique basé sur l'état (et les évènements) du contexte d'exécution de la plateforme pour construire des configurations cohérentes. Nous fournissons cependant un début de solution et des mécanismes qui nous permetterons de construire des systèmes autonomiques (ou autonomes).

Nous proposons ainsi un langage d'expression de contraintes, un interpréteur et un validateur de ces expressions de contraintes, un algorithme de sélection et un ensemble d'outils et d'environnements logiciels dédiés à la construction d'applications. Le système SELECTA respecte un ensemble de principes de base du génie logiciel. En effet, il s'agit d'un système qui est basé sur une architecture **modulaire** et **extensible**.

Le système SELECTA utilise une démarche d'ingénierie dirigée par les modèles et est bati sur le principe de la **séparation des préoccupations**. En effet, SELECTA est constitué de différents environnements logiciels identifiant les concepts et les aspects liés à chaque étape du cycle de vie des applications et utilise des mécanismes puissants fournis par l'approche CADSE pour la composition de ces environnements. SELECTA offre diverses vues aux developpeurs d'applications pour séparer les aspects et les points de vue en fonction des tâches à réaliser.

Notre proposition utilise un ensemble de concepts et combine les avantages de différents domaines tels que les approches à service ou à composant orienté service et l'architecture logicielle. La composition structurelle se base sur les principes de l'architecture logicielle et du paradigme à composant (ou bien à composant à service). Le système SELECTA permet de réaliser une composition structurelle de services. Nous avons montré aussi comment notre système peut être utilisé pour répondre à quelques besoins des architectures logicielles.

Nous avons montré comment mettre en valeur tous les concepts et les avantages de l'ingénierie dirigée par les modèles. La figure 42 décrit les domaines concernés par les travaux de cette thèse :

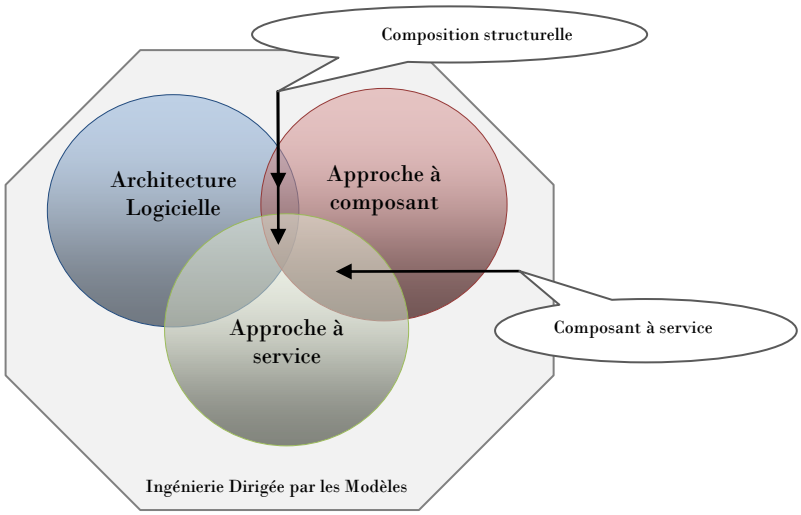

figure 42. Principaux domaines concernés par notre proposition.

7.2 PERSPECTIVES DU TRAVAIL.

Plusieurs axes de recherche dérivent de la proposition de cette thèse. Ces axes de recherche concernent par exemple :

- la gestion du cycle de vie des applications,
- la gestion de la conformité et de la validité de modèles,
- l'amélioration de notre algorithme de sélection,
- la réalisation d'un environnement complexe et complet pour le déploiement,
- la mise en œuvre du concept de composite autonomique dans le système SELECTA,
- l'utilisation de SELECTA pour la construction de familles d'applications logicielles.

Notre principal défi (Cf. Section 4.1), à long terme, est de fournir des environnements logiciels spécialisés pour la réalisation des tâches spécifiques aux activités associées à chaque phase du cycle de vie des applications (*conception*, *composition*, *déploiement*, *exécution*, *monitoring*, *évolution*, etc.), en utilisant le concept de composite comme élément central. Dans cette thèse, nous proposons un concept de composite permettant de fournir les notions et les mécanismes nécéssaires pour atteindre notre défi, et notre proposition répond aux besoins des applications logicielles.

GESTION DE LA CONFORMITE ET LA VALIDITE DE MODELES A L'EXECUTION.

D'autres travaux sont en cours de réalisation au sein de notre équipe de recherche concernant la gestion de la conformité et de la validité de modèles. En effet, la description de l'application est spécifiée en intention dans un modèle appelé *modèle d'application* et un *modèle d'état* représente l'état à un instant donné de la platefome d'exécution (les services qui y tournent, ceux qui sont démarrés et qui sont disponibles, etc.). Le travail consiste à garantir que :

- l'exécution soit toujours conforme à la description de l'application décrite dans le modèle d'application,

- les évolutions des modèles restent valides par rapport à un modèle d'évolution qui décrit les politiques d'évolution, etc.

EXTENSION DU LANGAGE DE CONTRAINTES ET DE L'ALGORITHME DE SELECTION.

Nous avons proposé un algorithme de sélection permettant de choisir les éléments participant à la composition afin d'automatiser le processus de sélection des éléments. Nous garantissons dans notre approche plusieurs propriétés nécessaires lors de la composition telles que la *complétude*, la *sélection minimale* d'éléments ou la *cohérence* d'un composite. Notre solution ne garantit cependant pas la construction de *configurations optimales*, pour cela il faudrait définir explicitement les critères d'optimalité. Notre langage de contraintes peut être étendu avec le concept de poids afin d'indiquer la priorité des expressions de contraintes. En fonction du poids et des éléments ayant la probabilité maximale de s'exécuter correctement ensemble, nous pourrions déterminer la « meilleure » configuration posssible des composites.

Dans certain cas, l'algorithme de sélection peut échouer dans le choix des éléments correspondant aux contraintes et aux propriétés requises par un composite. En effet, dans une étape de sélection donnée, si plusieurs éléments remplissent les critères de sélection alors l'algorithme choisira au hasard un (ou plusieurs) élément(s). Le ou les choix réalisés dans une étape de sélection peuvent s'avérer être des mauvais choix lors d'une autre étape. Une solution face à ce problème est de fournir un algorithme avec une option de retour en arrière (*backtraking*). Nous avons réalisé, en option, quelques cas de *backtraking* dans notre algorithme de sélection. Une perspective du travail serait de mettre en place les autres cas restants.

ENVIRONNEMENT COMPLET SPECIALISE POUR LE DEPLOIEMENT.

Dans le chapitre 6, nous avons spécifié les concepts liés à la phase de packaging et de déploiement de services ou des applications. Toutefois, il faut noter que nous n'avons introduit que les concepts « minimaux » nécessaires au déploiement et qu'il faudrait les étendre afin de fournir des supports logiciels plus complexes et puissants spécialisés pour le déploiement. Une perspective du travail serait donc de fournir un environnement logiciel CADSE spécialisé pour le déploiement.

Une autre perspective de travail serait la mise en œuvre du concept de composites autonomiques dans le but de supporter la réalisation et l'exécution de systèmes autonomes qui sont sensibles au contexte d'exécution et pouvant s'administrer tout seul en fonction de ce contexte.

LIGNES DE PRODUITS LOGICIELS ET COMPOSITES.

L'ingénierie des lignes (ou familles) de produits logiciels [IEE03][Car] favorise la réutilisation d'entités logicielles, non pas pour la réalisation d'un produit particulier mais plutôt pour concevoir et développer une famille de produits logiciels spécifiques à un domaine. Elle permet ainsi aux concepteurs de systèmes logiciels, de faire face à des préoccupations telles que (1) la réduction des coûts et délais de réalisation des systèmes et (2) l'amélioration et l'augmentation de la productivité de ces systèmes. Cette approche est de plus en plus adoptée et a connu un véritable succès dans l'industrie [CN03]. Elle possède un ensemble de besoins pour améliorer la réalisation de systèmes qui se basent sur cette approche. C'est pourquoi, plusieurs travaux de recherche ont été menés (ou sont en cours de réalisation) dans le domaine des lignes de produits [PKGJ08][DSB05][CN03]. Nous pouvons identifier les besoins suivants :

- **La gestion de la variabilité** : la variabilité designe les propriétés qui différencient les membres de la famille de produits. La gestion de la variabilité est un aspect important dans les approches de lignes de produits. Plusieurs travaux existent actuellement pour la modélisation et la gestion de la variabilité entre les produits d'une famille [IJJ97][DSGB03].
- **La gestion de la cohérence et de contraintes** : il faudrait assurer que les modèles de produits sont cohérents par rapport à l'architecture de la famille de produits. L'utilisation de la notion de métamodèle proposée dans l'ingénierie dirigée par les modèles permet d'obtenir cette propriété de cohérence. Il faudrait cependant pouvoir exprimer les contraintes de cohérence des modèles. Ces contraintes sont utiles lors de la dérivation des produits à partir de l'architecture de la famille et des éléments logiciels réutilisables qui existent.
- **L'automatisation de la dérivation** : le processus de la construction d'un produit de la famille devrait être automatisé [PKGJ08][GS07][McG05]. Cette automatisation possède des conséquences comme : la gestion de la variabilité des produits, la vérification et la validation de la cohérence des modèles, la disponibilité d'un *repository* contenant les artéfacts logiciels disponibles et l'usage de mécanismes de sélection d'artéfacts qui sont nécessaires à la configuration d'un produit [GS07]. En général, les conséquences des sélections réalisées ne sont pas claires (les artéfacts logiciels peuvent avoir des dépendances entre eux, le fait de sélectionner un élément va impliquer de sélectionner *aussi ses* dépendances) [DSB04]. Il faudrait aussi assurer la cohérence des sélections effectuées. L'assemblage de composants est une tâche complexe.
- **Des environnements flexibles et évolutifs** : le coût de production des environnements logiciels pour la conception de systèmes logiciels n'est pas négligeable. Il faudrait donc des environnements logiciels moins coûteux en termes de conception et de développement. Ces environnements devraient être assez flexibles et évolutifs afin de faciliter leur utilisation, réutilisation et extensibilité.

L'ingénierie des lignes de produits logiciels est compatible avec beaucoup de technologies. Actuellement, un certain nombre de technologies émergentes, telles que l'ingénierie dirigée par les modèles ou le paradigme orienté service, ont influencé les approches de lignes de produits logiciels [LMN07]. Nous remarquons que les approches (et les mécanismes) de composition et celles des lignes de produits logiciels ont des préoccupations communes, qui sont liées aux problématiques de réutilisation et d'intégration d'entités logicielles lors de la conception d'applications.

L'environnement SELECTA proposé dans cette thèse peut être utilisé pour la conception de familles d'applications logicielles. Il fournit des solutions pour répondre aux besoins de l'ingénierie des lignes de produits logiciels. Certains besoins comme l'expression et la gestion de la variabilité sont cependant traités différemment dans notre approche. En effet, nous utilisons la notion de groupe d'équivalence (Cf. Section 4.2) pour exprimer les propriétés communes et celles pouvant différencier les membres (services) d'un groupe d'équivalence. Ainsi, plusieurs services fournissant les mêmes fonctionnalités peuvent disposer de différentes valeurs pour les propriétés du groupe d'équivalence. Par contre, dans l'ingénierie des lignes de produits la variabilité est exprimée en termes de caractéristiques (appelées des *features*) qui expriment des propriétés.

Une configuration correspondant à un produit logiciel peut être modélisée sous la forme de composite SELECTA. En utilisant notre approche, l'architecture de la famille d'applications peut être décrite comme une entité composite, en termes de spécifications de service et potentiellement avec des contraintes. Ce qui permet ainsi de rendre « abstraite » la spécification de l'architecture de la famille d'applications. Plusieurs produits qui sont exprimés sous la forme de composites peuvent raffiner le même composite. Les membres de la famille peuvent être dérivés à partir de l'architecture du composite de référence, avec l'usage du concept de *refines* (figure 25) et de *résolution* (Cf. Section 4.2) que nous proposons. Notre solution permet de résoudre les éléments abstraits (spécifications de service) pour les concrétiser en sélectionnant des composants (implémentations de service) respectant les contraintes spécifiées (Cf. Annexe C). Par exemple, dans la grande famille des téléphones portables, on pourrait définir les téléphones « bas de gamme », les téléphone 3G, les « smart phones » etc. sous forme de composites abstraits qui se raffinent les uns les autres ; les membres de chaque sous-famille étant des composites complets qui raffinent le composite abstrait représentant la sous-famille. La figure suivante décrit notre mécanisme de construction de ligne de produits logiciels :

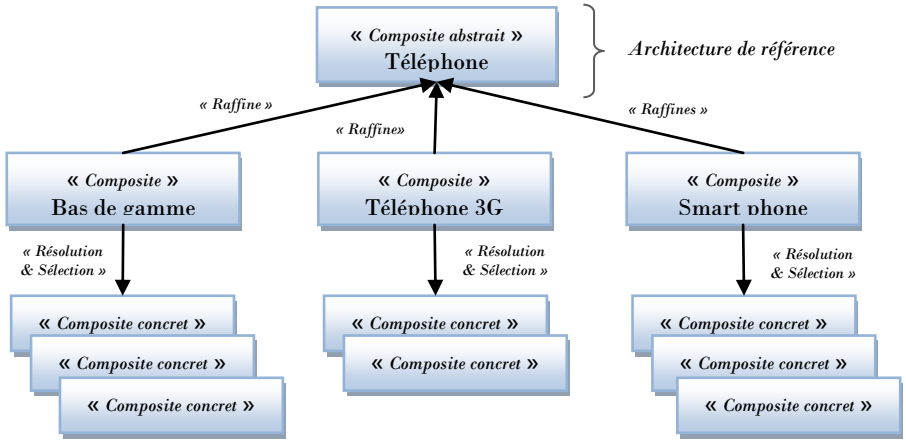

figure 43. Ligne de produits des téléphones portables.

Dans notre approche, nous proposons un langage permettant d'exprimer les contraintes souhaitées et nous fournissons un ensemble d'outils et d'environnements logiciels offrant la capacité par exemple de sélectionner automatiquement les composants nécessaires aux produits (applications), de gérer la cohérence et la complétude des applications (configurations partielles ou complètes). Nous proposons ainsi des supports logiciels pour construire et exécuter d'une manière flexible des configurations de produits et des mécanismes pour effectuer (ou retarder) des sélections de composants. Le tableau suivant résume les réponses que nous apportons pour concevoir des familles d'applications logicielles en utilisant l'environnement SELECTA :

SELECTA

Gestion de la variabilité.	Oui, avec le concept de groupe d'équivalence.
Gestion de la cohérence et de contraintes.	Oui.
Automatisation de la dérivation de produits.	Oui.
Environnement flexible et extensible.	Oui.
Feature	Non.

tableau 21. SELECTA pour la construction de familles d'applications.

Nous fournissons des solutions face à certains besoins de l'ingénieire des lignes de produits. Cependant, nous sommes conscients qu'il reste plusieurs questions à aborder afin de fournir une « veritable » approche ligne de produits logiciels. Une perspective du travail serait donc d'essayer de modeliser explicitement le concept de *feature* (de l'ingénierie des lignes de produits) [KCHN90] dans Selecta, de préciser en détail les similitudes et les différences entre les concepts de *feature* et des propriétés de groupe de SELECTA et de spécifier la manière de composer les *features* dans le contexte de SELECTA.

BIBLIOGRAPHIE.

[AAC07] Jonatha Anselmi, Danilo Ardagna, and Paolo Cremonesi. A QoS-based selection approach of autonomic grid services. In *SOCP '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, pages 1–8, New York, NY, USA, 2007. ACM Press.

[AAS02] J. R. Hobbs O. Lassila D. Martin D. McDermott S. A. McIlraith S. Narayanan M. Paolucci T. Payne A. Ankolekar, M. Burstein and K. Sycara. DAML-S: Web Service Description for the SemanticWeb. Sardinia, Italy, June 2002. In The Semantic Web - ISWC 2002: First International Semantic Web Conference.

[ACKM03] G. Alonso, F. Casati, H. Kuno, and H. Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer Verlag, 2003.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM.

[Act] ActiveBPEL. Activebpel website. http://www.active-endpoints.com/active-bpel-engine-overview.htm.

[Adea] Adele Team. CADSE Website. http://cadse.imag.fr/.

[ADEb] ADELE TEAM. SAM Website. http://sam.ligforge.imag.fr/.

[ADEc] ADELE TEAM. Website. http://www-adele.imag.fr/.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture*. Ph.d. thesis, Carnegie Mellon, School of Computer Science, May 1997. Technical Report Number: CMU-CS-97-144.

[And00] J. Andersson. Issues in Dynamic Software Architectures. In *Proc. of the Int. Software Architecture Workshop*, pages 111–114. IEEE, 2000.

[Apa] Apache. Apache felix website. http://felix.apache.org/.

[Apa08] Apache Software Foundation. The Apache Tuscany Project, 2008. http://tuscany.apache.org/.

[Arc] ArchStudio. Archstudio website. http://www.isr.uci.edu/projects/archstudio/index.html.

[Ars04] A. Arsanjani. Service-oriented modeling and architecture. how to identify, specify, and realize services for your soa. http ://www-128.ibm.com/developerworks/webservices/library/wssoa-design1/, 2004.

[ASL89] A. M. Alashqur, S. Y. W. Su, and H. Lam. OQL: a query language for manipulating object-oriented databases. In *VLDB '89: Proceedings of the 15th international conference on Very large data bases*, pages 433–442, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[Bar05] O. Barais. *Construire et Maîtriser l'évolution d'une architecture logicielle à base de composants, Ph. D. Thesis, , Lille, France, November* . Phd thesis, Laboratoire d'Informatique Fondamentale de Lille, Lille, France, November 2005.

[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.

[BCDW04] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33, New York, NY, USA, 2004. ACM.

[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.

[BD05] Olivier Barais and Laurence Duchien. Safarchie studio: Argouml extensions to build safe architectures. In Springer Boston, editor, *Architecture Description Languages*, volume 176 of *IFIP International Federation for Information Processing*, pages 85–100, 2005.

[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.

[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.

[BMDL08] O. Barais, A. F. Le Meur, L. Duchien, and J. Lawall. Software Architecture Evolution. pages 233–262. Springer, 2008.

[Boo93] G. Booch. *Object-Oriented Analysis and Design with Applications (2nd Edition)*. Object-Oriented Software Engineering. Addison-Wesley Professional, 1993.

[BOP03] Jian Yang Bart Orriens and Mike P. Papazoglou. Model driven service composition. In LNCS, editor, *International Conference on Service-Oriented Computing (ICSOC)*, volume 2910, pages 75–90. Springer, 15-18 Dec. 2003.

[BPPT06] Antonio Bucchiarone, Andrea Polini, Patrizio Pelliccione, and Massimo Tivoli. Towards an architectural approach for the dynamic and automatic composition of software components. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 12–21, New York, NY, USA, 2006. ACM.

[Bra04] J. S. Bradbury. Organizing definitions and formalisms for dynamic software architectures. Technical report 2004-477, School of Computing, Queen's University, March 2004.

[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.

[Car] CarengieMellon. Software engineering institute. http://www.sei.cmu.edu/.

[Cer04] Humberto Cervantes. *Vers un modèle à composants orienté services pour supporter la disponibilité dynamique*. PhD thesis, Université Joseph Fourier - Grenoble I, March 2004.

[Cha02] C. Chaudet. *Pi-Space : Langage et outils pour la description d'architectures évolutives à composants dynamiques. Formalisation d'architectures logicielles et industrielles*. Phd thesis, Université de Savoie, Savoie, France, December 2002.

[Cha07] David Chappell. Introducing sca. Avalaible at http://www.davidchappell.com/articles/Introducing_SCA.pdf, July 2007.

[CL08] Stéphanie Chollet and Philippe Lalanda. Security specifcation at process level. In *SCC '08: Proceedings of the 2008 IEEE International Conference on Services Computing*, pages 165–172, Washington, DC, USA, July 2008. IEEE Computer Society.

[Cle96] Paul C. Clements. A survey of architecture description languages. In *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, page 16, Washington, DC, USA, 1996. IEEE Computer Society.

[CN03] P. C. Clements and L. Northrop. A framework for software product line practice – version 4.2. Pittsburgh, USA, 2003. Carnegie Mellon, Software Engineering Institute.

[Com] CommUnity. Community web site. http://www.fiadeiro.org/jose/CommUnity.

[Cud05] M. Lutfiyya H. Cuddy, S. Katchabaw. Context-aware service selection based on dynamic and static service attributes. volume 4, pages 13– 20.

International Conference on Wireless And Mobile Computing, Networking And Communications, 2005. (WiMob'2005),, IEEE, 22-24 Aug. 2005.

[Das07] Eric M. Dashofy. *Supporting Stakeholder-driven, Multi-view Software Architecture Modeling*. PhD thesis, University of California, Irvine, 2007.

[Dav05] Pierre-Charles David. *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation*. PhD thesis, Université de Nantes, Juillet 2005.

[dP99] Virginia C. de Paula. *ZCL: A Formal Framework for Specifying Dynamic Distributed Software Architectures*. Ph.d. thesis, Computer Science Department, Federal University of Pernambuco, Brazil, August 1999.

[dPJC00] Virginia C. de Paula, G. R. Ribeiro Justo, and P. R. Freire Cunha. Specifying and verifying reconfigurable software architectures. In *PDSE '00: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, page 21, Washington, DC, USA, 2000. IEEE Computer Society.

[DS05] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1:1–30, 2005.

[DSB04] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Experiences in software product families: Problems and issues during product derivation. In *SPLC*, pages 165–182, 2004.

[DSB05] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74(2):173–194, 2005.

[DSGB03] Sybren Deelstra, Marco Sinnema, Jilles Van Gurp, and Jan Bosch. Model driven architecture as approach to manage variability in software product families. In *In Proceedings of the Workshop on Model Driven Architectures: Foundations and Applications*, pages 109–114. Springer, 2003.

[Ecla] Eclipse. Eclipse BPEL oficial website. http://www.eclipse.org/bpel/.

[Eclb] Eclipse SOA. Soa tools platform project. http://www.eclipse.org/stp/.

[Eclc] Eclipse Web site. http://www.eclipse.org/.

[EDS09] Jacky Estublier, Idrissa A. Dieng, and Eric Simon. Automating Component Selection and Building Flexible Composites for Service-Based Applications. 2009.

[EDSV09] Jacky Estublier, Idrissa A. Dieng, Eric Simon, and German 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.

[EH07] Clément Escoffier and Richard S. Hall. Dynamically Adaptable Applications with iPOJO Service Components. In Markus Lumpe and Wim Vanderperren, editors, *6th International Symposium on Software Composition (SC 2007)*, volume 4829 of *Lecture Notes in Computer Science*, pages 113–128. Springer, December 2007.

[EHL07] Clément Escoffier, Richard S. Hall, and Philippe Lalanda. iPOJO: an Extensible Service-Oriented Component Framework. *IEEE International Conference on Services Computing (SCC 2007)*, pages 474–481, July 2007.

[ELV09] Jacky Estublier, Thomas Lévêque, and German Vega. Evolution control in MDE projects: Controlling model and code co-evolution. April 2009.

[ES09] J. Estublier and E. Simon. Universal and extensible service-oriented platform feasibility and experience: The service abstract machine. In *Second IEEE International Workshop on Real-Time Service-Oriented Architecture and Applications (RTSOAA 2009)*, Seattle, July 2009.

[Esc08] Clément Escoffier. *iPOJO : Un modèle à composant à service flexible pour les systèmes dynamiques*. PhD thesis, Université Joseph Fourier, December 2008.

[EVLL08] Jacky Estublier, German Vega, Philippe Lalanda, and Thomas Lévêque. Domain Specific Engineering Environments. In *APSEC*, pages 553–560. IEEE, December 2008.

[Fab08] Fabric3 Project. The Fabric3 Project, 2008. http://www.fabric3.org/overview.html.

[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.

[FCS00] LiJie Jin Vasudev Krishnamoorthy Fabio Casati, Ski Ilnicki and Ming-Chien Shan. Adaptive and dynamic service composition in eflow. In LNCS, editor, *Advanced Information Systems Engineering*, volume 1789, pages 13–31. Springer, March 2000.

[FEBF06] J-M. Favre, J. Estublier, and M. Blay-Fornarino. *L'ingenénierie dirigér par les modèles au delà du MDA*. Hermes Lavoisier, 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.

[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.

[GAaCB95] Cristina Gacek, Ahmed Abd-allah, Bradford Clark, and Barry Boehm. On the definition of software system architecture. In *Proceedings of the First International Workshop on Architectures for Software Systems*, page 85–94, Seatle, WA, April 1995.

[GBF09] Nelly Bencomo Gordon Blair and Robert R. France. Models@run.time. In *4th Workshop on Models@run.time at MODELS 09*, Denver, USA, October 2009.

[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.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In V.Ambriola and G.Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume I, pages 1–40, New Jersey, 1993. World Scientific Publishing Company.

[GS07] Hassan Gomaa and Michael E. Shin. Automated software product line engineering and product derivation. In *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, page 285a, Washington, DC, USA, 2007. IEEE Computer Society.

[GS09] David Garlan and Bradley Schmerl. AEvol: A tool for defining and planning architecture evolution. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 591–594, Washington, DC, USA, 2009. IEEE Computer Society.

[HC01] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together (ACM Press)*. Addison-Wesley Professional, June 2001.

[HSZ03] Bin Zhou Haiyan Sun, Xiaodong Wang and Peng Zou. Research and implementation of dynamic web services composition. In Springer, editor, *Advanced Parallel Processing Technologies*, volume 2834 of *LNCS*, pages 457–466, 2003.

[HWH08] Willem-Jan van den Heuvel Hans Weigand and Marcel Hiel. Rule-based service composition and service-oriented business rule management. Proceedings of ReMoD 2008, 2008.

[IBM] IBM Websphere. http://www-01.ibm.com/software/websphere/.

[IEE03] IEEE Software staff. Product line engineering: The state of the practice. volume 20, pages 52–60, Los Alamitos, CA, USA, 2003. IEEE Computer Society Press.

[IJJ97] M. Griss I. Jacobson and P. Johnson. *Software Reuse: Architecture, Process and Organization for Business Success*. 1997.

[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.

[JM07] Michael C. Jaeger and Gero Mühl. Qos-based selection of services: The implementation of a genetic algorithm. In *In KiVS 2007 Workshop: Service-Oriented Architectures und ServiceOriented Computing (SOA/SOC*, pages 359–370, 2007.

[JMK95] S. Eisenbach J. Magee, N. Dulay and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference, ESEC'95*, September 1995.

[KCHN90] K. Kang, S. Cohen, J. Hess, and S. Novak, W.and Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute (SEI), November 1990.

[Krä08] Bernd J. Krämer. Component meets service: what does the mongrel look like? In Springer London, editor, *Innovations in Systems and Software Engineering*, volume 4, 2008.

[KT02] Tomás Kalibera and Petr Tuma. Distributed component system based on architecture description: The sofa experience. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 981–994, London, UK, 2002. Springer-Verlag.

[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.

[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.

[lig] LIG website. http://www.liglab.fr/.

[LMN07] John Bergey Gary Chastek Sholom Cohen Patrick Donohoe Lawrence Jones Robert Krut Reed Little John McGregor Liam O'Brien Linda M. Northrop, Paul C. Clements with Felix Bachmann. A framework for software product line practice, version 5.0. *Product Line Practice Initiative*, 2007.

[Mar08] Cristina Marin. *Une approche orientée domaine pour la composition de services*. PhD thesis, Université Joseph Fourier, May 2008.

[McG05] J. McGregor. Preparing for automated derivation of products in a software product line. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005. CMU/SEI-2005-TR-017, ADA448223.

[MDSR07] Arun Mukhija, Andrew Dingwall-Smith, and David S. Rosenblum. QoS-Aware Service Composition in Dino. In *ECOWS '07: Proceedings of the Fifth European Conference on Web Services*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[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.

[MK96] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, page 3–14, San Fransisco, CA, October 1996.

[MM04] N. Milanovic and M. Malek. Current solutions for web service composition. volume 8, pages 51– 59. Internet Computing, IEEE, 2004.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. volume 26, pages 70–93, Piscataway, NJ, USA, 2000. IEEE Press.

[NCF05] Abdul Wasim Shaikh Nizamuddin Channa, Shanping Li and Xiangjun Fu. Constraint satisfaction in dynamic web service composition. pages 658– 664. Proceedings. Sixteenth International Workshop on Database and Expert Systems Applications, 2005., 22-26 Aug 2005.

[New08] Newton Project. The Newton Project, 2008. http://newton.codecauldron.org/site/index.html.

[OAS04] OASIS. Universal description, discovery and integration specification. Specification available at http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm, October 2004.

[OAS07] 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.

[Obj01] Object Management Group. Corba component model version 4.0. Specification available at http://www.omg.org/docs/formal/06-04-01.pdf, April 2001.

[Obj07] Objectweb. JOnAS: Java Open Application Server, 2007. http://wiki.jonas.objectweb.org/xwiki/bin/view/Main/WebHome.

[OMG] OMG. Model Driven Architecture. Available at http://www.omg.org/mda/.

[OMG06] OMG. Object constraint language specification, version 2.0, 2006.

[Orc] Orchestra IBM. Orchestra official website. http://orchestra.objectweb.org/xwiki/bin/view/Main/WebHome.

[OSG05a] OSGi Alliance. OSGi Service Platform Core Specification. http://www.osgi.org, 2005.

[OSG05b] OSGi Alliance. OSGi Service Platform Service Compendium. http://www.osgi.org, 2005.

[OSO07] OSOA. *SCA Service Component Architecture : Assembly Model Specification*, 15 March 2007. http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications.

[OW2] OW2 Consortium. FraSCati Website. https://wiki.ow2.org/frascati/.

[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.

[PDE08] G. Pedraza, I. Dieng, and J. Estublier. Multi-concerns composition for a process support framework. In *Proceedings of the ECMDA Workshop on Model Driven Tool and Process Integration*, Berlin, June 2008. FOKUS.

[PDE09] G. Pedraza, I. A. Dieng, and J. Estublier. Focas: An enginering environment for service-based applications. In *Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, Milan, Italy, 9-10 May 2009. INSTICC Press.

[PE08] G. Pedraza and J. Estublier. An extensible services orchestration framework through concern composition. In *Proceedings of the 1st International Workshop on Non-functional System Properties in Domain Specific Modeling Languages - NFPinDSML-2008*, volume 394 of *CEUR Workshop Proceedings*, Toulouse, September 2008.

[Ped09] Gabriel Pedraza. *FOCAS : un canevas extensible pour la construction d'applications orientées procédé*. PhD thesis, University Joseph Fourier Grenoble, 2009.

[Pel03a] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, October 2003.

[Pel03b] Chris Peltz. web ser vices orchestration: a review of emerging technologies, tools, and standards. 2003.

[PKGJ08] Gilles Perrouin, Jacques Klein, Nicolas Guelfi, and Jean-Marc Jézéquel. Reconciling automation and flexibility in product derivation. In *SPLC '08: Proceedings of the 2008 12th International Software Product Line Conference*, pages 339–348, Washington, DC, USA, 2008. IEEE Computer Society.

[POR98] R. Taylor P. Oreizy, N. Medvidovic and D. Rosenblum. Software architecture and component technologies: Bridging the gap. In *OMG-DARPA-MCC Workshop on Compositional Software Architectures*, Monterey, CA, January 1998.

[PS01] Giacomo Piccinelli and Eric Stammers. From e-processes to e-networks: an e-service-oriented approach. pages 549–553. Proceedings of Web Services Workshop, 2001.

[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.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[RK06] F. Leymann R. Khalaf, A. Keller. Business processes for web services: principles and applications. *IBM Systems Journal*, 45:425 – 446, January 2006.

[RKL+05] D. Roman, U. Keller, H. Lausen, J. Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.

[RM07] R. Rouvoy and P. Merle. Description et de vérification de motifs d'architecture avec fractal adl. In *Proceedings of the French Conference on Langages et Modèles à Objets (LMO'07)*, Toulouse, France, march 2007.

[SAS03] Siegfried HANDSCHUH Sudhir AGARWAL and Steffen STAAB. Surfing the service web. In LNCS, editor, *International semantic web conference*

(ISWC), volume 2870, pages 211–226, Sanibel Island FL , ETATS-UNIS, 2003. Springer.

[SG95] Mary Shaw and David Garlan. Formulations and formalisms in software architecture. In Springer-Verlag, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 307–323, 1995.

[SGM02] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional. 2nd Edition, England, 2002.

[SMF+09] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J-B. Stefani. Reconfigurable SCA Applications with the FraSCati platform. In *SCC '09: Proceedings of the 2009 IEEE International Conference on Services Computing*, pages 268–275, Washington, DC, USA, 2009. IEEE Computer Society.

[Spr] Spring source. Spring framework. http://www.springframework.org.

[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.

[The] The Internet Engineering Task Force (IETF). Rfc 1960: A string representation of ldap search filters. http://www.ietf.org/rfc/rfc1960.txt.

[UPn08] UPnP Forum. UPnP Device Architecture 1.1, October 2008. http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf.

[Veg05] G. Vega. *Développement d'Applications à Grande Echelle par Composition des Méta-Modèles*. PhD thesis, University Joseph Fourier Grenoble, 2005.

[Ves93] S. Vestal. A cursory overview and comparison of four architecture description languages. Technical report, Honeywell Technology Center, February 1993.

[W3C00] W3C. Simple Object Access Protocol (SOAP) 1.1, May 2000. http://www.w3.org/TR/soap/.

[W3C02] W3C. Web Services Description Language (WSDL). Specification available at http://www.w3.org/2002/ws/desc/, 2002.

[W3C04a] W3C. Owl-s: Semantic markup for web services. http://www.w3.org/Submission/OWL-S/, 2004.

[W3C04b] W3C. Web Services Architecture Specification. Specification available at http://www.w3.org/TR/ws-arch/, February 2004.

[Wil01] D. S. Wile. Supporting the DSL spectrum. *Journal of Computing and Information Technology*, 9(4):263–287, December 2001.

[Wol97] Alexander L. Wolf. Succeedings of the second international software architecture workshop (isaw-2). volume 22, pages 42–56, New York, NY, USA, 1997. ACM.

[Wri] Wright. Wright website. http://www.cs.cmu.edu/afs/cs/project/able/www/wright/.

[YMBM08] I-Ling Yen, Hui Ma, Farokh B. Bastani, and Hong Mei. Qos-reconfigurable web services and compositions for high-assurance systems. volume 41, pages 48–55, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press.

[YML05] Ping Yu, Xiaoxing Ma, and Jian Lu. Dynamic software architecture oriented service composition and evolution. In *CIT '05: Proceedings of the The Fifth International Conference on Computer and Information Technology*, pages 1123–1129, Washington, DC, USA, 2005. IEEE Computer Society.

[YP04] Jian Yang and Mike P. Papazoglou. Service components for managing the life-cycle of service compositions. *Information Systems*, 29(2):97–125, 2004.

[ZT07] Haiyan Zhao and Hongxia Tong. A dynamic service composition model based on constraints. pages 659–662. Sixth International Conference on Grid and Cooperative Computing, 2007. GCC 2007., 16-18 Aug 2007.

ANNEXE A : SYNTAXE DE NOTRE LANGAGE DE CONTRAINTES.

```
(<item-name>|<relation-name>|<variable-name>)::= <string>
<maximum> ::= <number> | "*"
<number> ::= <numeral> | <numeral> <number>
<numeral> ::= "0" | "1" | "2"| … | "9"

// cardinality = 1 ou 0-1 ou 1-* ou 0-* ou n avec n un entier >0
// La valeur par défaut est 1
<cardinality> ::= "1"| "0-1" | "1-*" | "0-*" | <maximum>\"0"

// Pour les services composites SELECTA, X désigne "Specification" |
"Implementation" …
<class> ::= any type X | "Self"
<selExpression> ::= <expression-LDAP>
<set> ::= (<class>|"#"<variable-name>) [<selExpression>]
<rule> ::=<set> {<navigation>} | <selExpression>

// <rule> retourne un ensemble d'items
// . = navigation directe ; .. = navigation inverse
<navigation>::= (". " | "..") <relation-name> ["["<cardinality>"]"]
[<selExpression>]
```

```
// Nous avons des expressions de contraintes et de sélection
// <variable-definition> permet de définir des variables

<expression> ::= <constraint-Expression> | <selection-Expression> |
<variable-definition>

// Nous distinguons les contraintes intrinsèques (des services) et celles
// (contextuelles) liées au composite

<constraint-Expression> ::= <intrinsic-constraint> | <composite-
constraint>

// Contraintes intrinsèques aux services (Self désigne le service
// définissant la contrainte)
// Exemple : (equals (Self.requires, Self..provides.requires))

<intrinsic-constraint> ::= <selExpression> ";"
<composite-constraint> ::= "Constraint" <rule> ";"

// « Select » designe une expression obligatoire de sélection
// « Optional » désigne une expression préférentielle
<selection-Expression> ::= ("Select" | "Optional") <rule> ";"
<variable-type> ::= <class> - {"Self"}

//définition de variables
<variable-definition>::= "def" <variable-type> <variable-name> "=" <rule>
";"

// On va étendre le langage LDAP avec les opérations <inclusion> &
// <equals> et la navigation sur les relations.
<inclusion> ::= "includes" "(" <rule> "," <rule> ")"
<equals> ::= "equals" "(" <rule> "," <rule> ")"

// Pour filtrer un ensemble avec une expression
<filterSet> ::=(<class>|"#"<variable-name>) <selExpression>
```

```
// Syntaxe des expressions LDAP étendues par le choix sur les éléments
// obtenus, les opérations d'inclusion et d'égalité d'ensembles.

<expression-LDAP> ::= "("<filtercomp> ")"
<filtercomp> : := <and> | <or> | <not> | <type> |
<and> ::= "&" <filterlist>
<or> ::= " | " <filterlist>
<not> ::= " ! " <expression-LDAP>
<filterlist> ::= <expression-LDAP> | <expression-LDAP> <filterlist>
<type> ::= <simple> | <navigation> | <filterSet> | <inclusion> | <equals>
<simple> ::= <attribute> <filtertype> <content>
<filtertype> ::= <equal> | <approx> | <greater> | <less>
<equal> ::= "="
<approx> ::= "~="
<greater> ::= ">="
<less> ::= "<="
```

ANNEXE B : EXEMPLE DE DESCRIPTION D'UNE CONFIGURATION DE COMPOSITE.

```
<?xml version="1.0" encoding="UTF-8"?>
<compositeservice name="AlarmSystemComposite" init="false" complete="false" delayed="false" conflict="false">

	<provides>

		<specification name="AlarmSystem" unique="false" />

	</provides>

	<containsSpec>

		<specification name="TemperatureSensor" unique="false" />

		<specification name="MeasureAggregator" unique="false" />

		<specification name="HumiditySensor" unique="false" />

		<specification name="SendNotification" unique="false" />

		<specification name="Security" unique="false" />

		<specification name="Log" unique="false" />

		<specification name="ActionDriver" unique="false" />

		<specification name="DataBase" unique="false" />

		<specification name="Analyse" unique="false" />

	</containsSpec>
```

```
<containsImpl>

        <implementation name="AnalyseAndLog" instanciable="true">

                <!-- These are the constraints defined by this service implementation item -->
                <local-constraints>

                        <constraint>Select DataBase (Execution = disc);</constraint>

                </local-constraints>


                <!-- These are the constraints which permit to choose this service
                        implementation item
                -->
                <relevant-constraints>

                        <constraint>Optional Implementation (shared = true) ;</constraint>

                </relevant-constraints>

                <!-- This item was chosen for these service specification(s) -->
                <for-specification>

                        <specification name="Analyse" unique="false" />

                </for-specification>

        </implementation>

        <implementation name="SecureLog" instanciable="true">


                <!--
                        These are the constraints which permit to choose this service
                        implementation item
                -->
                <relevant-constraints>

                        <constraint>Optional Implementation (shared = true) ;</constraint>

                </relevant-constraints>

                <!-- This item was chosen for these service specification(s) -->
                <for-specification>

                        <specification name="Log" unique="false" />

                </for-specification>

        </implementation>

        <implementation name="MySQL" instanciable="true">

                <!--
                        These are the constraints which permit to choose this service
                        implementation item
                -->
                <relevant-constraints>

                        <constraint>Optional Implementation (shared = true) ;</constraint>

                        <constraint>Select DataBase (Execution = disc) ;</constraint>

                </relevant-constraints>

                <!-- This item was chosen for these service specification(s) -->
                <for-specification>

                        <specification name="DataBase" unique="false" />

                </for-specification>

        </implementation>
```

```
<implementation name="TSZB" instanciable="true">

        <!--
                These are the constraints which permit to choose this service
                implementation item
        -->
        <relevant-constraints>

                <constraint>Optional Implementation (shared = true) ;</constraint>

                <constraint>Select TemperatureSensor (& (Protocol = ZigBee)
                        (ResponseTime < 3)) ;</constraint>

        </relevant-constraints>

        <!-- This item was chosen for these service specification(s) -->
        <for-specification>

                <specification name="TemperatureSensor" unique="false" />

        </for-specification>

</implementation>

<implementation name="MAa" instanciable="true">

        <!--
                These are the constraints defined by this service implementation
                item
        -->
        <local-constraints>

                <constraint>Select TemperatureSensor (& (Protocol = ZigBee)
                        (ResponseTime < 3));</constraint>

                <constraint>Select HumiditySensor (Protocol = ZigBee);
                </constraint>

        </local-constraints>

        <!--
                These are the constraints which permit to choose this service
                implementation item
        -->
        <relevant-constraints>

                <constraint>Optional Implementation (shared = true) ;</constraint>

                <constraint>Select MeasureAggregator (Protocol = ZigBee) ;
                </constraint>

        </relevant-constraints>

        <!-- This item was chosen for these service specification(s) -->
        <for-specification>

                <specification name="MeasureAggregator" unique="false" />

        </for-specification>

</implementation>

<implementation name="VoiceNotification" instanciable="true">

        <!--
                These are the constraints which permit to choose this service
                implementation item
        -->
        <relevant-constraints>

                <constraint>Select SendNotification (Type = voice) ;</constraint>

                <constraint>Optional Implementation (shared = true) ;</constraint>

        </relevant-constraints>


        <!-- This item was chosen for these service specification(s) -->
        <for-specification>

                <specification name="SendNotification" unique="false" />

        </for-specification>

</implementation>
```

```
<implementation name="HSc" instanciable="true">

        <!--
                These are the constraints which permit to choose this service
                implementation item
        -->
        <relevant-constraints>

                <constraint>Optional Implementation (shared = true) ;</constraint>

                <constraint>Select HumiditySensor (Protocol = ZigBee) ;
                </constraint>

        </relevant-constraints>

        <!-- This item was chosen for these service specification(s) -->
        <for-specification>

                <specification name="HumiditySensor" unique="false" />

        </for-specification>

</implementation>

<implementation name="DriverB" instanciable="true">

        <!--
                These are the constraints which permit to choose this service
                implementation item
        -->
        <relevant-constraints>

                <constraint>Optional Implementation (shared = true) ;</constraint>

        </relevant-constraints>

        <!-- This item was chosen for these service specification(s) -->
        <for-specification>

                <specification name="ActionDriver" unique="false" />

        </for-specification>

</implementation>

<implementation name="AS_Actions" instanciable="true">

        <!--
                These are the constraints defined by this service implementation
                item
        -->
        <local-constraints>

                <constraint>Select MeasureAggregator (Protocol = ZigBee);
                </constraint>

        </local-constraints>

        <!--
                These are the constraints which permit to choose this service
                implementation item
        -->
        <relevant-constraints>

                <constraint>Select AlarmSystem (& (Protocol = ZigBee) (Memory >=
                        256) ) ;</constraint>

                <constraint>Optional Implementation (shared = true) ;</constraint>

        </relevant-constraints>

        <!-- This item was chosen for these service specification(s) -->
        <for-specification>

                <specification name="AlarmSystem" unique="false" />

        </for-specification>

</implementation>

</containsImpl>
```

```
        <!-- Wires management -->
        <wires>

                <wire implementationItemSource="AS_Actions"
                        implementationItemDestination="AnalyseAndLog" />

                <wire implementationItemSource="AnalyseAndLog"
                        implementationItemDestination="VoiceNotification" />

                <wire implementationItemSource="MAa"
                        implementationItemDestination="TSZB" />

                <wire implementationItemSource="MAa"
                        implementationItemDestination="HSc" />

                <wire implementationItemSource="AS_Actions"
                        implementationItemDestination="MAa" />

                <wire implementationItemSource="AS_Actions"
                        implementationItemDestination="DriverB" />

                <wire implementationItemSource="AnalyseAndLog"
                        implementationItemDestination="MySQL" />

                <wire implementationItemSource="AnalyseAndLog"
                        implementationItemDestination="SecureLog" />

        </wires>

        <!-- Incomplete service specification item(s) -->
        <uncompleteSpec>

                <specification name="Security" unique="false" />

        </uncompleteSpec>

        <!--
                These are the constraints defined in the characteristics attribute of
                the composite item
        -->
        <constraints>

                <constraint>Select AlarmSystem (& (Protocol = ZigBee) (Memory >=
                        256) ) ;</constraint>

                <constraint>Select SendNotification (Type = voice) ;</constraint>

                <constraint>Optional Implementation (shared = true) ;</constraint>

        </constraints>

</compositeservice>
```

ANNEXE C :
EXEMPLE D'UN SYSTEME D'ALARME.

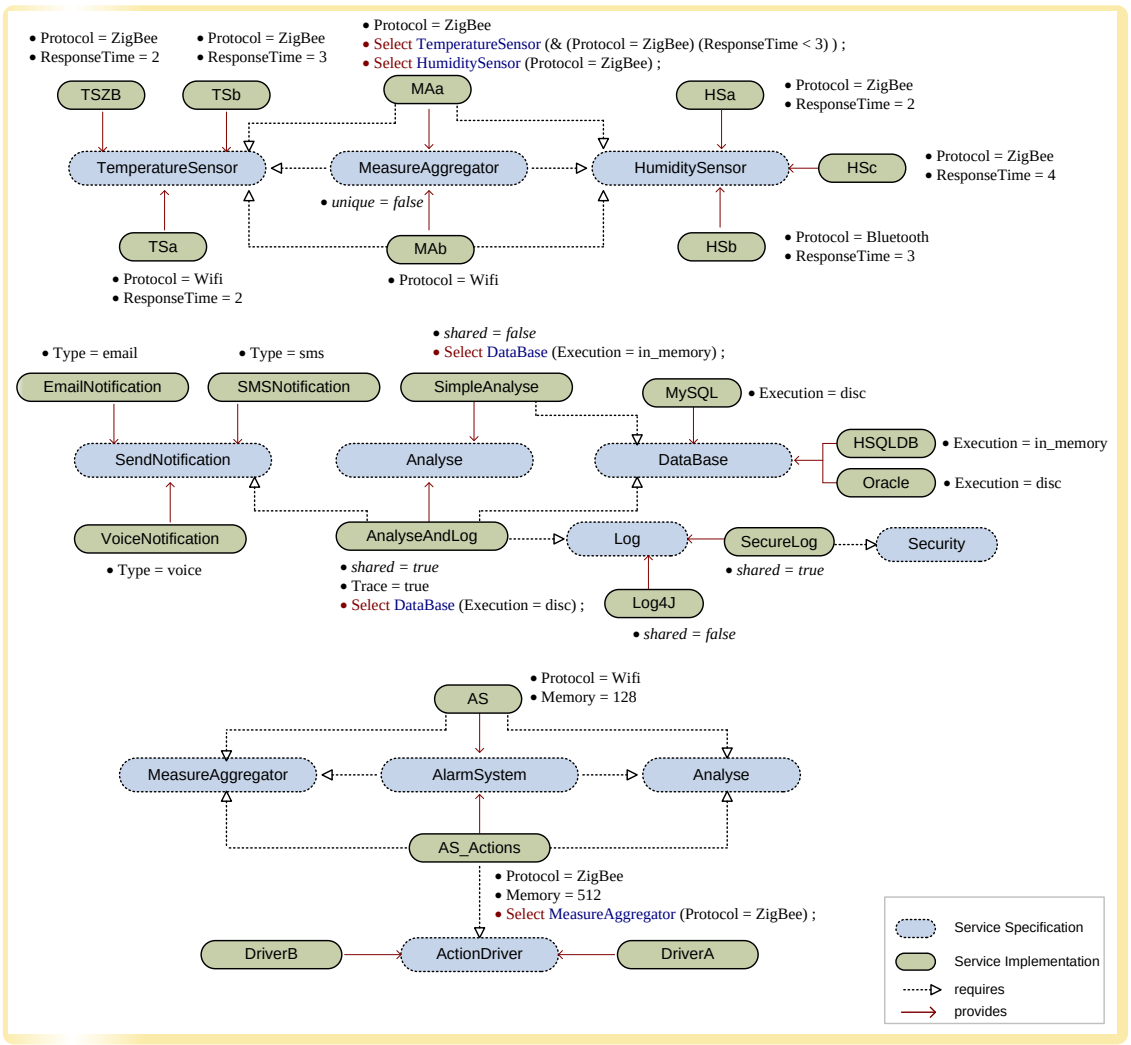

figure 44. Exemple de *repository* de services : système d'alarme d'une usine de production.

Dans la figure 44, nous trouvons dix (10) spécifications de services et vingt deux (22) implémentations de service fournissant ces spécifications. Les tableaux suivants décrivent les fonctionnalités fournies par ces services :

Spécifications de service	Description
TemperatureSensor	Elle offre les fonctionnalités permettant de récupérer des mesures de données liées à la température d'un environnement.
HumiditySensor	Elle offre les fonctionnalités permettant de récupérer des mesures de données liées à l'humidité d'un environnement.
MeasureAggregator	Elle fournit les fonctionnalités qui permettent de collecter et de remonter les données récupérées à partir d'un ensemble de capteurs, par exemple de température ou d'humidité. Cette spécification requiert *TemperatureSensor* et *HumiditySensor*, ce qui signifie que toutes les implémentations (par exemple *MAa*) fournissant la spécification *MeasureAggregator*, requièrent aussi les spécifications *TemperatureSensor* et *HumiditySensor*. La spécification *MeasureAggregator* possède la propriété *Unique* = *false* qui indique la possibilité d'utiliser simultanément plusieurs implémentations fournissant la spécification *MeasureAggregator* dans la même application. *Unique* est un attribut prédéfini dont la sémantique est connue par le système (sa valeur par défaut est « true »).
Analyse	Elle offre les fonctionnalités permettant d'analyser les données qui sont collectées, par exemple le calcul de la température moyenne ou un autre traitement plus complexe.
AlarmSystem	Elle fournit les fonctionnalités pour la réalisation de systèmes d'alarme. Cette spécification requiert *MeasureAggregator* et *Analyse* afin de pouvoir collecter un ensemble de mesures de données et les traiter ensuite pour signaler les incohérences.
SendNotification	Elle permet d'envoyer une alerte en cas d'incohérences dans le système. La notification peut se faire par exemple par un envoie de message électronique ou une alerte vocale.
Log	Elle fournit les fonctionnalités permettant de sauvegarder l'historique des actions (ou évènements) réalisées, par exemple dans un fichier ou une base de données.
Security	Elle offre les fonctionnalités permettant de garantir les aspects qui sont liés à la sécurité de l'application tels que l'intégrité ou la confidentialité de l'information.
DB	Elle offre les fonctionnalités fournies par les systèmes de gestion de base de données qui sont utilisables pour la sauvegarde et la manipulation de données.
ActionDriver	Elle intègre les fonctionnalités permettant de réagir en fonction des incohérences identifiées dans le système (en utilisant un ensemble de *drivers*).

Implémentations de service	Description
TSZB	Il s'agit d'une implémentation qui fournit les fonctionnalités offertes par la spécification *TemperatureSensor*. Elle possède les propriétés suivantes : *Protocol* = *ZigBee*[1] (indiquant le type du protocole de communication supporté) et *ResponseTime* = *2* (indiquant le temps maximal de réponse du service en secondes).
TSa	Elle fournit les fonctionnalités offertes par la spécification *TemperatureSensor* et possède les propriétés suivantes : *Protocol* = *Wifi* et *ResponseTime* = 2.
TSb	Elle offre les fonctionnalités fournies par la spécification *TemperatureSensor* et possède les propriétés suivantes : *Protocol* = *ZigBee* et *ResponseTime* = 3.
HSa	Il s'agit d'une implémentation qui fournit les fonctionnalités offertes par la spécification *HumiditySensor* avec les propriétés (*Protocol* = *ZigBee* et *ResponseTime* = *2*).
HSb	Elle offre les fonctionnalités fournies par la spécification *HumiditySensor* et possède les propriétés suivantes : *Protocol* = *Bluetooth* et *ResponseTime* = 3.
HSc	Elle fournit les fonctionnalités offertes par la spécification *HumiditySensor* et possède les propriétés suivantes : *Protocol* = *ZigBee* et *ResponseTime* = 4.
MAa	Il s'agit d'une implémentation qui fournit les fonctionnalités offertes par la spécification *MeasureAggregator* avec la propriété (*Protocol* = *ZigBee*). Cette implémentation requiert les spécifications *TemperatureSensor* et *HumiditySensor*. Elle déclare deux contraintes spécifiant les caractéristiques des implémentations (qui fournissent ses spécifications requises) pouvant être utilisées en même temps que *MAa*.
MAb	Elle fournit les fonctionnalités offertes par la spécification *MeasureAggregator* avec la propriété (*Protocol* = *Wifi*). Cette implémentation requiert les spécifications *TemperatureSensor* et *HumiditySensor*.
SimpleAnalyse	Il s'agit d'une implémentation fournissant les fonctionnalités du service *Analyse*. Elle possède la propriété *shared* = *false* exprimant le fait que les instances de cette implémentation ne peuvent pas être partagées par diverses applications. *Shared* est une propriété prédéfinie dont la sémantique est connue par le système (la valeur par défaut de cette propriété est « false »). Cette implémentation requiert une base de données (*DataBase*). Dans cet exemple, elle possède une contrainte indiquant qu'elle utilise une base de données proposant une gestion en mémoire vive (par exemple, *HSQLDB*).
AnalyseAndLog	Il s'agit d'une autre implémentation fournissant les fonctionnalités du service *Analyse* et possède les propriétés *Shared* = *true* et *Trace* = *true* (indiquant qu'elle réalise des traces de logging ; d'où la dépendance vers un service de *Log*). Cette implémentation requiert une base de données (*DataBase*) et impose une contrainte spécifiant un type de base de données (proposant une gestion en disque, par exemple *MySQL*). Elle requiert aussi les services *Log* et *SendNotification* (pour l'envoie des alertes).

[1] ZigBee est un protocole de communication basé sur la norme IEEE 802.15.4 qui est utilisable pour les réseaux à dimension personnelle (WPAN : Wireless Personal Area Network).

EmailNotification	Elle permet d'envoyer les alertes par courrier électronique (*Type* = *email*).
SMSNotification	Elle permet d'envoyer les alertes en utilisant un service de messagerie SMS (Type = *sms*).
VoiceNotification	Elle permet d'alerter les incohérences du système par voix (Type = *voice*).
MySQL	Elle offre les fonctionnalités du service *DataBase* avec une gestion sur des fichiers disques (*Execution* = *disc*).
HSQLDB	Elle offre les fonctionnalités du service *DataBase* avec une gestion en mémoire vive (*Execution* = *in_memory*).
Oracle	Elle offre les fonctionnalités du service *DataBase* avec une gestion sur des fichiers disques (*Execution* = *disc*).
Log4J	Il s'agit d'une implémentation fournissant les fonctionnalités d'un service de *Log*. Elle possède la propriété *shared* = *false* exprimant le fait que les instances de cette implémentation ne peuvent pas être partagées par diverses applications. *Shared* est une propriété prédéfinie dont la sémantique est connue par le système (la valeur par défaut de cette propriété est « false »).
SecureLog	Elle offre les fonctionnalités d'un service de *Log*. Elle requiert un service de sécurité (*Security*) pour assurer un log sécurisé et possède la propriété (*shared* = *true*).
AS	Elle offre les fonctionnalités du service *AlarmSystem* et requiert les services *MeasureAggregator* et *Analyse*. Elle possède les propriétés suivantes : *Protocol* = *Wifi* (pour les communications) et *Memory* = *128* (indiquant sa capacité de stockage interne).
AS_Action	Il s'agit d'une autre implémentation qui fournit les fonctionnalités du service *AlarmSystem* et requiert les services *MeasureAggregator*, *Analyse* et *ActionDriver* (pour pouvoir agir sur le système en cas de problèmes). Elle possède les propriétés suivantes : *Protocol* = *ZigBee* (pour les communications) et *Memory* = *512* (indiquant sa capacité de stockage interne), et impose une contrainte concernant le protocole de communication (ZigBee) des services de collecteur de données (*MeasureAggregator*) pouvant être utilisés en même que *AS_Action*.
DriverA	Elle intègre les fonctionnalités permettant d'agir sur un système en cas d'incohérences dans ce dernier.
DriverB	Elle intègre les fonctionnalités permettant d'agir sur un système en cas de problèmes.

tableau 22. Caractéristiques des services décrits dans le *repository* SAM de la figure 44.

DEFINITION INITIALE DU SYSTEME D'ALARME.

Un système d'alarme est modélisé comme un composite fournissant la spécification *AlarmSystem*. Dans notre exemple, nous souhaitons réaliser un système d'alarme utilisant le service *SendNotification* (pour l'envoie des alertes d'incohérences du système) ayant les contraintes suivantes :

```
Select AlarmSystem (&(Protocol = "ZigBee") (Memory >= 256));
```

Cette contrainte impose de sélectionner une implémentation fournissant la spécification *AlarmSystem* qui utilise le protocole ZigBee pour assurer les communications

du système et possède une capacité de mémoire de stockage interne supérieure ou égale à 256 Mo.

```
Select SendNotification (Type = "voice") ;
```

Cette contrainte de sélection impose de choisir une implémentation fournissant le service *SendNotification* capable de signaler les incohérences du système par une alerte vocale (*Type* = *voice*).

```
Optional Implementation (shared = "true") ;
```

En plus, la contrainte précédente permet de choisir, de préférence s'il y'en a parmi les services disponibles et qui respectent les contraintes de l'application, des implémentations disposant la propriété suivante (*shared* = *true*) ; *shared* est un attribut prédéfini dont la sémantique est connue par le système SELECTA et indique le fait qu'un service peut être partagé par diverses applications (composites).

La figure suivante décrit la définition initiale du système d'alarme :

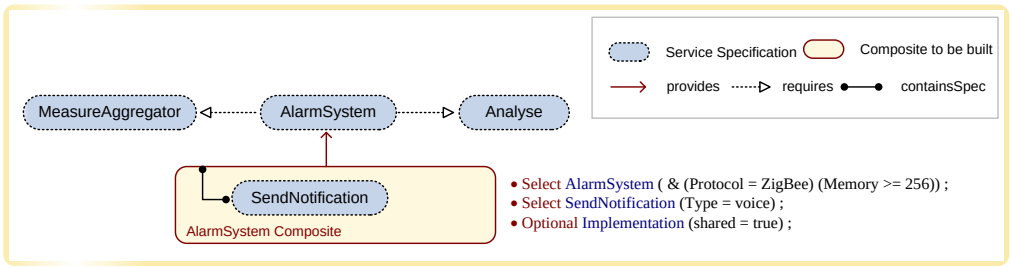

figure 45. La description initiale de notre système d'alarme.

EXEMPLE DE CONFIGURATION DE NOTRE SYSTEME D'ALARME.

La figure 46 illustre une configuration de notre système d'alarme obtenue après l'exécution de l'algorithme de sélection :

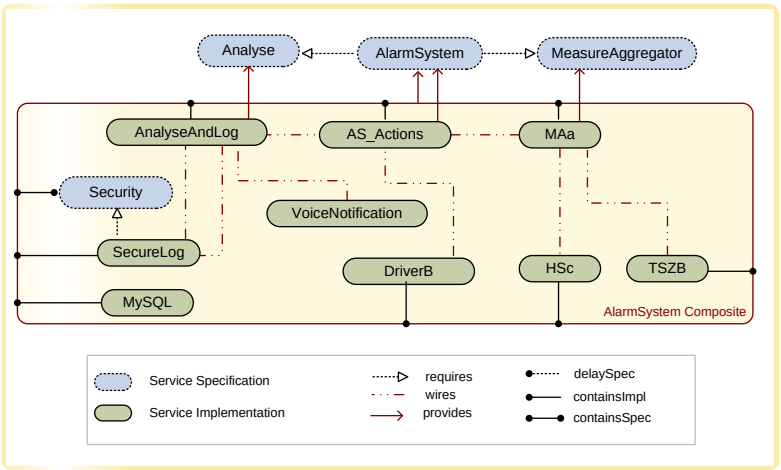

figure 46. Une configuration du système d'alarme.

Le composite décrit dans de la figure 46 est dans l'état incomplet dans la mesure où le groupe représenté par *Security* n'a pas pu être résolu (aucune implémentation fournissant la spécification *Security* n'est disponible dans le *repository*).