RNTL FAROS
Composition de contrats pour la Fiabilité d'ARchitectures Orientées Services

Livrable F-1.1
Coordonnateur : Philippe COLLET

État de l'art sur la contractualisation et la composition

Projet FAROS

Août 2006

Projet RNTL FAROS : `http://www.lifl.fr/faros`

Financé par ANR — alicante — EDF R&D — france telecom — i3S sophia antipolis — IRISA — lifL

CHAPITRE 4

Développement par composition

Coordonnateur : Mireille Blay-Fornarino.
Rédacteurs : Pierre Combes, Laurence Duchien, Tristan Glatard, Philippe Lahire, Stéphane Lavirotte, Clémentine Nemo, Audrey Occello, Renaud Pawlak, Anne-Marie Pinna-Dery, Lionel Seinturier, Jean-Yves Tigli.

4.1 Introduction

Depuis plusieurs années, l'ingénierie du logiciel tend à se rapprocher du développement "hardware" par l'assemblage de "composants logiciels". La programmation par "composition" s'est alors intensifiée pour répondre aux problèmes de répartition et d'hétérogénéité des composants logiciels, pour décroître la complexité de développement des applications, pour favoriser la gestion des aspects techniques. En quoi les différentes formes de compositions répondent-elles aux problèmes d'adaptabilité et de fiabilité des applications ? Comment impactent-elles l'adaptabilité et la fiabilité des systèmes construits par composition ? Dans ce chapitre, nous visons à répondre à ces questions par l'étude des travaux relatifs à la composition en phase de développement i.e. de la conception à l'exécution.

Composition : Action de former une application par assemblage ou combinaison de plusieurs éléments.
A cette définition informelle sous-tend l'existence de deux formes de composition, l'assemblage et la combinaison. Il s'en suit différentes interprétations du terme composition selon les approches.

Dans ce chapitre, nous étudions la composition selon les axes suivants :

- l'assemblage de composants logiciels : dans ce contexte nous abordons l'interopérabilité et les architectures logicielles (cf. 4.2),
- la composition par combinaison de codes et de modèles (cf. 4.3) : nous nous focalisons plus précisément sur la séparation des préoccupations,
- la construction d'assemblage dans les architectures orientées services (cf. (cf. 4.4) : nous revenons dans cette partie sur l'usage de la programmation par séparation des préoccupations dans les architectures orientées services.

Pour chacun de ces axes, après une présentation de travaux qui lui sont relatifs, nous l'analysons au regard de l'adaptation et de ces moyens et capacités à garantir la fiabilité des applications.

La composition des contrats a été abordée au chapitre précédent, et la composition des modèles sera détaillée au chapitre suivant. Nous les effleurons seulement dans ce chapitre lorsque un rapprochement explicite avec ces travaux nous semble nécessaire.

4.2 Assemblage de composants et interopérabilité

Assembler : Mettre ensemble.

La composition des composants intervient au niveau de l'assemblage des composants pour exprimer les interactions entre les composants.
La diversité des modèles à composants et des interprétations qui leur sont associées nous a conduit à commencer cette partie en posant notre vocabulaire (cf. 4.2.1), avant de présenter la notion de plates-formes à composants (cf. 4.2.2). Nous rappelons ensuite la gestion des assemblages par des langages de définition d'architectures (ADL) (cf. 4.2.3) une description plus détaillée est donnée au chapitre 3, avant de présenter différents travaux relatifs à l'assemblage de composants réalisés dans nos équipes : fractal, Wcomp et le système Satin.

4.2.1 Modèles à composants

Bien qu'il n'existe pas de consensus sur la définition de "composant", les avis semblent unanimes sur les points suivants : *Un composant est une unité logicielle qui spécifie clairement ses interactions avec l'extérieur en définissant les services qu'elle offre et ceux qu'elle requiert*.
L'assemblage consiste alors à "connecter" les composants. Nous présentons brièvement les protocoles élémentaires qui interviennent dans une application à base de composants.

Création de composants

Sous-jacent aux principaux langages de composants, nous trouvons un langage de classes. Le terme de composant est alors utilisé à la fois pour parler des instances et des classes. La notion d'instances peut correspondre à un ensemble d'objets (adaptateurs, objet intermédiaire, objet d'implémentation, squelette). Dans les modèles à composants les plus évolués, la création des instances est associée à la définition d'une "Home" qui joue le rôle de fabrique et détient la faculté de créer des instances ou de renvoyer des références vers des instances existantes. Cette entité joue un rôle très important pour la gestion de pool d'objets, de références, de l'initialisation, de la persistance, du déploiement, etc. [DmYK01, obj05]. Dans la suite de ce travail, nous préciserons par instances ou classes la signification du terme composant en cas d'ambiguïté.

Liaisons entre composants

La nature des liaisons entre les instances de composants dépend du modèle de composant. Ainsi un des modèles les plus riches, CCM[obj05], supporte les communications par envoi de messages et émission d'événements. Ces derniers sont particulièrement bien appropriés à l'adaptation dynamique des assemblages de composants en autorisant plusieurs émetteurs ou récepteurs d'information sur un même port.
D'autres formes de liaisons existent qui mettent en jeu le contrôle sur les composants, c'est le cas du modèle Fractal (cf. 4.2.4).
La définition explicite des communications entre les composants est un des principaux avantages des modèles à composants sur les modèles à objets. Certains travaux cherchent aujourd'hui à assurer que seules des communications explicites entre des composants sont utilisées afin de valider l'intégrité des communications [LCL06, ACN02].
Les liaisons jouent un rôle important en particulier lorsque les composants sont distants, voir définis dans des plates-formes hétérogènes. Dans ce cas, l'approche la plus générale de gestion de l'hétérogénéité repose sur le bus logiciel sous-jacent (RMI, RMI-IOOP, CORBA, .Net). Dans [Kra03], Krakowiak explicite les différents patterns de programmation utilisés pour gérer les références et l'interopérabilité. Pour accroître l'abstraction et favoriser la mise en place

de formes diverses de communication, différents travaux enrichissent la notion de liaisons conduisant à la mise en œuvre de connecteurs [Car03, MB05].
De manière non explicite et donc plus proches des travaux sur les aspects présentés dans la section 4.3, les travaux sur les interactions entre instances de composants séparent la gestion des interactions et de l'interopérabilité entre les composants et l'attribue à un mécanisme extérieur dit serveur d'interactions [BFCE+04].

Substituabilité des composants

Pour répondre aux objectifs de réutilisation, la substituabilité d'un composant par un autre est une propriété essentielle des modèles à composants, qui est en particulier utilisée pour l'adaptation d'une application[GHC99]. Le plus souvent défini par une simple validation de type, différents travaux abordent ce point en associant aux composants les notions de rôles[OPD04b], de contrats de qualité, de validité des protocoles d'usage [DUVH06, RRB02, Rap02, TFS04], etc.

Recherche et Découverte des composants

Des services de nommage aux services "vendeurs" [MMGL01], de nouveaux protocoles tels que UPnP, OSGI proposent des mécanismes de découverte de composants qui favorisent l'adaptation dynamique des applications en fonction du "contexte".
Le choix du mode de découverte des services actif/passif, centralisé ou non, modifie le processus d'adaptation dynamique.
Dans le cas des services de nommage ou vendeurs, l'application interroge le service pour obtenir un nouveau composant qui réponde mieux à ses besoins. Ces travaux introduisent alors les notions de qualité de service pour déterminer en fonction des demandes du clients et du contexte d'exécution [BDH01]. Les travaux de Colombe Herault et Sylvain Lecomte étendent cette démarche en introduisant des coordinateurs qui gèrent via des contrats en fonction des modifications du contexte d'usage le choix des services techniques qui sont représentés par des composants [HL04].
A l'inverse l'utilisation de protocoles basés sur la déclaration à tous les points de contrôle de l'existence d'un nouveau service place la découverte de service au centre d'un processus d'adaptation réactif (cf. http ://www.upnp.org/) qui ne dépend plus d'un composant centralisé.

Encapsulation et hiérarchie de composants

Un autre point important de l'approche à composants est la possibilité donnée par un modèle à composants tel que Fractal de structurer les composants par encapsulation, créant ainsi des entités de réutilisation de plus gros grain. L'architecture et la substituabilité des composants doivent alors prendre en compte cette nouvelle forme de composants.
Dans le domaine des IHMs, nous trouvons d'autres formes de compositions des composants. Utilisé par dans un cadre d'adaptation d'IHMs multimodales, un modèle particulièrement intéressant, est celui de l'approche ICARE[BNB04], dans laquelle 4 sortes de composants de composition sont définis permettant de combiner les données de 2 à n composants : un pour la Complémentarité (par exemple pour fusionner les données de composants donnant l'orientation et la position d'un utilisateur), un pour la Redondance (parole et geste pour désigner un lieu sur une carte), un pour l'Equivalence (cliquer sur un bouton ou dire un mot clef est équivalent) et un pour la Redondance/Equivalence.

4.2.2 Plates-formes à composants

Comme nous l'avons vu précédemment à la création d'une instance de composant sont associées à la fois les notions de "home", d'ensemble d'objets, de pools, de proxies, etc. Ces différentes entités vont constituer l'infrastructure de la plate-forme [MDBF06]. Le modèle CCM étend cette prise en compte du cycle de vie des composants par le déploiement et des politiques de gestion des références sophistiquées via le Portable Object Adaptator (POA).
C'est sur l'infrastructure que repose l'intégration des services dits non fonctionnels tels que la sécurité, les transactions, la persistance. Les plates-formes à composants supportent à la fois dans leur spécification (CCM, EJB, Fractal) et dans leurs mises en oeuvre l'intégration des services non-fonctionnels. Via des fichiers de configuration, les générateurs de code prennent en charge l'intégration des codes relatifs aux services. L'intégration de services non standard repose alors sur des extensions des spécifications et en particulier des fichiers de configuration.
Tandis que les spécifications des plates-formes à composants stipulent ces différents aspects du cycle de vie d'un composant, les implémentations qui les supportent s'appuient sur des infrastructures très différentes[NBF04].

4.2.3 Architectures

Un assemblage de composants va constituer une application. Le langage Piccola est dédié à l'assemblage des composants [AN01]. Différents travaux visent à décrire les architectures et à les valider, les langages supports sont alors dit langage de définition d'Architectures (ADL) (Pour une présentation détaillée cf.3.4).

Langages de description d'architecture(ADL)

Les langages de description d'architecture (ADL) [MT97] sont des notations destinées à représenter l'architecture d'un système logiciel en vue de son analyse. L'utilisation des ADLs se situe principalement au moment de la conception d'un système.
Dans les ADLs, l'architecture d'un système est décrite principalement en terme de composants qui implémentent des interfaces, de connecteurs (interconnexions entre composants) et de leurs configurations (ou compositions). Un composant est présenté de manière grossière dans le cadre des ADLs comme une unité de calcul ou un entrepôt de données. Une interface spécifie les services que le composant fournit. Les connecteurs modélisent les interactions entre les composants à travers leurs interfaces ainsi que les règles qui gouvernent ces interactions. Une configuration (ou composition) représente un graphe de composants connectés entre eux à l'aide de connecteurs.
Les ADLs sont généralement accompagnés d'une panoplie d'outils permettant la modélisation contrôlée des architectures (en limitant les possibilités d'actions en fonction de l'état courant de la modélisation de l'architecture[1]), l'analyse de l'architecture (model checkers, parsers, ...), la génération de code, la simulation de l'architecture.
La définition d'interfaces répond de manière très élémentaire à l'expression de la sémantique des composants. Pour permettre d'utiliser les outils décrits précédemment (vérification de contraintes, simulation des architectures), il est nécessaire de disposer d'un modèle définissant la sémantique des composants. Par exemple, Rapide repose sur l'ordonnancement partiel d'événements, et définit la sémantique comportementale [LV95]. Wright utilise le formalisme CSP pour analyser les connexions entre connecteurs et composants afin de détecter des deadlocks [All97].

[1] Par exemple, la sélection de composants dont les interfaces ne sont pas couramment utilisées dans l'architecture peut être interdite.

La majorité des ADLs existants ne s'intéressent qu'aux configurations statiques. Les exceptions sont C2SADEL [MORT96], Darwin [MK96], Rapide [LV95], et Weaves [GR91]. Darwin et Rapide supportent seulement les modifications dynamiques d'architecture qui sont connues à l'avance. C2SADEL et Weaves permettent l'ajout, la suppression et le ré-assemblage de composants.

Les ADLs sont exploités pendant la phase de conception des applications et sont rarement disponibles pendant la phase d'exécution de celles-ci. Aussi sont-ils insuffisants dans l'état actuel pour garantir que des changements dynamiques seront appliqués au système d'une façon sûre. La plupart des ADLs garantissent, en effet, la validité d'assemblage que pour les constructions initiales. D'après Medvidovic [MT97, Med96], même les ADLs capables de modéliser des changements dynamiques (anticipés ou non) sont insuffisants pour garantir que les changements sont appliqués de manière sûre. Medvidovic précise que les problèmes de sûreté dus aux adaptations dynamiques tel que le problème de la consistance des assemblages, n'entrent pas dans le cadre des ADLs et doivent être pris en charge par des outils d'analyse spécifiques aux environnements d'exécution.

4.2.4 Fractal : du modèle aux implémentations

Le modèle de composants Fractal a pour base les notions de composant, interface, liaisons et contrats [BCL+04b]. Les originalités principales de ce modèle résident dans une composition hiérarchique des composants qui autorise le partage et sur une prise en charge du contrôle à la fois par réflexivité et via des contrôleurs [BCS02].

Un composant est une entité exécutable qui est conforme au modèle de composant Fractal. Les composants fractal peuvent être de toute taille et représenter aussi bien des composants métiers que techniques.

Les interfaces sont les seuls points d'interaction entre les composants. Elles expriment les opérations requises ou fournies par un composant.

Une liaison est un canal de communication établis entre des composants. Une liaison primitive unit des interfaces de composants définies dans le même espace d'adressage. Elle est implémentée par exemple, par une référence Java ou un pointeur C. Une liaison composite est une configuration de liaisons primitives et de composants de liaisons. Un composant de liaison est un composant dédié à la gestion de la communication entre par exemple des composants qui ne se trouvent pas dans le même espace d'adressage.

Le contrôle des composants s'appuie sur l'existence d'une "membrane" qui contrôle le contenu du composant. Ce contrôle réflexif a plusieurs points d'impacts : contrôle du contenu et des sous-composants, contrôle des envoi et réception de messages, contrôle de l'exécution, du cycle de vie, de la qualité de services, etc.

Un composant peut contenir récursivement d'autres composants, dans la figure 4.2.4, le composant C est Primitif tandis que les composants (A, B1, B2) sont composites. Un composant peut être contenu dans plusieurs composants. Dans ce cas, le composant qui le contrôle est le plus petit composant qui englobe l'ensemble des composants qui l'englobent(A pour C). Les contrats sont des obligations réciproques entre des éléments tels que les composants, les interfaces, les liaisons, etc. Le modèle de composants Fractal spécifie différents contrats qui doivent être respectés dont (i) il ne doit pas y avoir de cycle dans la relation de contenance des composants composites (ii) une liaison ne doit être établie qu'entre des interfaces "compatibles",(iii) des composants ne peuvent communiquer que si une liaison a été établie entre eux, etc.

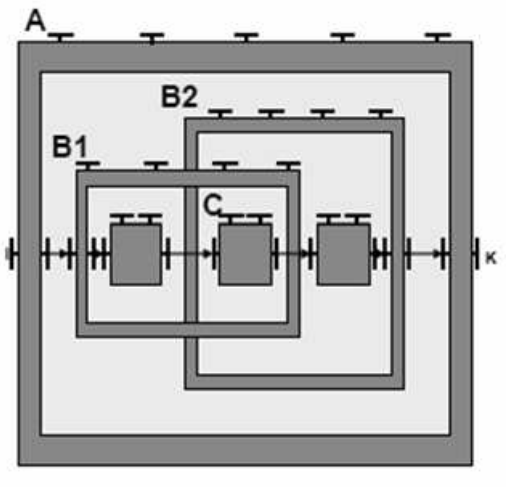

FIG. 4.1 – Composants Composites. Les rectangles noir représentent la membrane du composant, i.e. le contrôle du composant. Les formes en T associées aux rectangles correspondent aux interfaces internes ou externes du composant. Lorsque les interfaces externes sont positionnées au dessus du composants, elles correspondent à des interfaces de contrôles. Les flèches représentent les liaisons.

Mises en œuvre Il existe différentes implémentations du modèle de composants Fractal. L'implémentation de référence est "Julia"[2].
Il y a plusieurs autres implémentations (cf. le site de fractal[3]) :

AOKell est une implémentation du modèle de composant fractal, basée sur AspectJ. L'approche est décrite dans le paragraphe 4.3.6.

FracTalk [4] est une implémentation en Smalltalk du modèle de composant fractal.

FractNet [5] est une implémentation en .Net du modèle de composant fractal.

Plasma [6] est une implémentation en C++ du modèle de composant fractal, dédiée aux applications multimédias.

ProActive est une implémentation asynchrone et distribuée du modèle de composant fractal, qui adresse la gestion de grille de calculs.

Think est une implémentation en C dédiée au développement des systèmes d'exploitation.

Langages et Outils Différents Langages et outils ont été définis et développés autour du modèle Fractal. Nous noterons tout particulièrement le langage de configuration Fractal ADL (cf. http ://fractal.objectweb.org/tutorials/adl/index.html) qui permet de décrire les types, composants, interfaces , etc., conduisant au déploiement automatique de configurations à base de composants.
Les composants Fractal JMX permettent d'administrer à distance des applications fractal par l'utilisation d'agents JMX (cf. http ://java.sun.com/products/JavaManagement/).

[2]http ://fractal.objectweb.org/tutorials/julia/index.html
[3]http ://fractal.objectweb.org/index.html
[4]http ://csl.ensm-douai.fr/FracTalk
[5]http ://www-adele.imag.fr/fractnet/
[6]http ://www.inria.fr/rapportsactivite/RA2004/sardes/uid43.html

4.2.5 Wcomp : Approche multi-designs du développement par composition

Wcomp est une plate-forme à composants pour le développement rapide de prototypes d'applications multi-dispositifs devant évoluées en cours d'exécution[CFWTLR06].
Son architecture s'organise autour de *Containers* et de *Designers*. L'objectif des Containers est de prendre en charge dynamiquement la gestion de comportements tels que l'instanciation, la désignation, la destruction de composants logiciels fonctionnels et de liaisons. Les Designers permettent la manipulation dynamique des assemblages de composants en utilisant les formalismes adaptés. Un Designer graphique d'architecture comme Bean4WComp permet par exemple de composer manuellement des assemblages de composants à partir d'une représentation graphique des flots d'événements. Il est particulièrement adapté à la description de l'application. Un Designer d'aspects d'assemblage comme ISL4WComp permet quant à lui, par le biais d'une évolution du langage ISL (Interaction Specification Language), de décrire des schémas d'interaction[BFCE$^+$04]. Ces derniers sont alors sélectionnables, applicables et tissables [CFWBFT$^+$05]et permettent ainsi d'adapter dynamiquement l'application précédemment décrite à son contexte. Les relations entre la plate-forme et les différents designers a été étudiée en terme de transformations entre métamodèle dans [CFWBFT$^+$06].
Enfin, dans un soucis permanent de mise en œuvre et d'expérimentation, WComp est au cœur d'un environnement expérimental original : une plate-forme d'étude des usages des équipements informatiques mobiles en environnement simulé, appelé « Ubiquarium[7] Informatique ». L'Ubiquarium est constitué pour cela de divers dispositifs, comme autant de services découvrables et composables dynamiquement par WComp. Ces dispositifs peuvent être à la fois des dispositifs virtuels (objets d'une scène 3D dans laquelle l'utilisateur est immergé), et des dispositifs réels portés sur lui ou présents dans son environnement. Un tel environnement est un cadre idéal à l'évaluation des nouvelles applications de l'informatique mobile et ambiante telles que les usages des ordinateurs portés ou « wearable computers ».

4.2.6 Contrôle dynamique de l'évolution des assemblages : le système SATIN

L'approche Satin est basée sur un modèle d'adaptations dynamiques indépendant plate-forme sur lequel des propriétés de sûreté sont formalisées et validées. Les propriétés de sûreté permettent d'anticiper à partir d'une description d'adaptation si la mise en œuvre de celle-ci ne remettra pas en cause la sûreté de fonctionnement de l'application.
Certaines propriétés de sûreté servent à garantir en particulier que les modifications d'assemblages à l'exécution sont sûres. Ainsi, une fonctionnalité ne peut pas être supprimée tant qu'il existe des liaisons de composants utilisant cette fonctionnalité pour garantir la consistance des assemblages. De même, un composant ne peut remplacé un autre composant que s'il est capable de remplir un rôle similaire pour garantir la conservation du contexte d'utilisation des composants et de l'assemblage (le problème du transfert d'état n'est pas abordé dans Satin). Enfin, toute modification d'assemblage introduisant cycles ou points de non-déterminisme peut être interdite.
Le modèle [OPD04a] est projeté sous la forme d'un service de sûreté que les plates-formes peuvent interroger pour déterminer si une adaptation donnée risque de briser la sûreté de fonctionnement de l'application. Les propriétés de sûreté sont alors évaluées, sous la forme de contraintes OCL, à chaque demande d'adaptation. Le service de sûreté permet d'aider l'adaptateur à décider si une adaptation peut être réalisée sans perturber l'exécution de l'application. Les propriétés de sûreté ayant été définies de manière orthogonale, l'utilisateur du service a la possibilité de choisir un sous-ensemble des propriétés qu'il souhaite voir préservées vis-à-vis de la plate-forme d'adaptation considérée.

[7] du Latin Ubique, en toute chose et tout être, avec le suffixe rium signifiant lieu ou structure. Donc Ubiquarium Informatique : " lieu ou structure dans laquelle l'informatique est en toute chose et tout être "

4.2.7 Analyse de la composition par assemblage

Dans [Szy96], l'auteur souligne déjà combien il est nécessaire de faire évoluer les démarches de développements pour rendre les systèmes vraiment extensibles ce qui inclut (i) de prendre en charge des capacités d'extensions séparées des codes (nous aborderons ce point au paragraphe 4.3), (ii) de tenir compte de la nécessité de travailler dans un monde ouvert, (iii) d'associer plus d'informations sémantiques aux entités à composer.
Les composants répondent partiellement à ces points.

Adaptation

Les modèles à composants intègrent par nature une dimension adaptation des assemblages. Cette adaptation se caractérise à la fois par la modification des assemblages, la substitution de composants et dans les modèles plus évolués tels que Fractal, par une adaptation des contrôles. L'approche par composants vise également à favoriser l'intégration de services dits techniques tels que les transactions, la sécurité, la persistance. L'évolution des applications dans ce contexte d'assemblage et de génération de code est alors facilitée améliorant la productivité et la réutilisation des composants. C'est ainsi que l'approche à composants vise aujourd'hui à la production et l'utilisation de composants sur étagère (Components-Off-The-Shelf « COTS »).
Les langages de descriptions des architectures permettent de "prévoir" les adaptations et de les valider mais pas de les diriger. La prise en compte d'adaptations non anticipées et la prise en compte du contexte pour demander et valider des adaptations à l'exécution reste problématique alors que les nouveaux protocoles de découvertes des composants intensifient les besoins dans ce domaine.

Validation

La validation des assemblages de composants est laissée aux langages sous-jacents par typage par les modèles à composants. Les langages de définition des architectures (ADL) étendent très largement ces possibilités de validation en permettant de vérifier les assemblages et de simuler les exécutions. Différents travaux présentés plus largement au chapitre 3 renforcent les capacités de validation des assemblages par exemple en intégrant des informations sur le comportement des composants [BR06].
Lorsque les assemblages peuvent évoluer dynamiquement, ces formes de vérifications ne suffisent plus et des systèmes de validation doivent alors être mises en place comme nous l'avons vu avec le système SATIN (cf. 4.2.6).

4.3 Composition par combinaison

Combinaison : Union, dans des proportions définies, de deux ou plusieurs éléments donnant un nouvel élément ayant des propriétés différentes de celles de ses composants[8].

Jusqu'à présent nous avons essentiellement abordé la composition comme extérieure aux entités composées.
Nous proposons à présent de nous intéresser aux compositions qui modifient les entités mises en jeux. Cette forme de composition est alors plus proche de la notion de "loi de composition" en mathématiques.

[8] Cette définition est une adaptation de la définition suivante relative à la chimie : Union, dans des proportions définies, de deux ou plusieurs corps donnant un nouveau corps ayant des propriétés différentes de celles de ses composants.

Loi de Composition

En mathématiques, une loi de composition, ou loi tout court, est une relation ternaire qui est aussi une application. C'est donc une application d'un produit cartésien de deux ensembles E et F dans un troisième ensemble G, avec G égal à E ou à F. [extrait de wikipedia]

Après une présentation générale des travaux relatifs à la programmation et modélisation par séparation des préoccupations, nous détaillons différents systèmes et langages, réalisés dans nos équipes et qui apportent des éléments à la problématique FAROS.

4.3.1 Programmation par séparation des préoccupations

Aspect-oriented software development is a new technology for separation of concerns (SOC) in software development. The techniques of AOSD make it possible to modularize crosscutting aspects of a system. (cf. http ://aosd.net/)
La complexité croissante des applications et l'expertise nécessaire dans les différents aspects techniques ont conduit à une nouvelle approche du développement de logiciels basée sur la "séparation des préoccupations" tandis que des outils dits "Tisseurs" (Weavers en anglais) se chargent de valider la cohérence du tout et leur composition.
On peut distinguer deux approches du développement par séparation des préoccupations : le développement par *sujets* et par *aspects*.
La notion de sujet est proche de la notion de vue et introduit un découpage vertical. Les mécanismes de composition s'appuient en particulier sur la fusion.
Les aspects introduisent un découpage transversal et la composition repose sur des points d'entrelacements des codes. Parmi les aspects souvent traités nous trouvons : la gestion des utilisateurs (authentification), l'archivage des données (persistance), la programmation concurrentielle (multi-threading), l'information pendant l'exécution du logiciel (trace), l'application de patterns de programmation, etc.

Après une rapide présentation du vocabulaire que nous utilisons relativement à la programmation par séparation des préoccupations, nous précisons les différentes formes de programmation par aspects, puis nous décrivons quelques langages supports à cette forme de programmation. Nous concluons ce paragraphe en énonçant les limites de la programmation par séparation des préoccupations.

Principe et vocabulaire

Le développeur spécifie la partie métier de son application indépendamment des préoccupations techniques. Celles-ci sont prises en charge par des aspects. Un aspect est spécifié de manière autonome, en principe indépendamment des applications auxquelles il sera appliqué. Pour cela, un aspect spécifie les appels à effectuer à certains *points d'exécution (Joinpoint en anglais)*. Parmi les points d'exécutions les plus fréquents, nous trouvons l'appel à une méthode, l'exécution d'une méthode, l'affectation d'un attribut, la levée d'une exception, etc.
Un *point d'interception (poincut en anglais)* précise les points d'exécution satisfaisants aux conditions d'activation d'un aspect. Il permet par exemple d'identifier l'invocation d'une méthode particulière, sur l'instance d'une classe bien précise. Il peut être exprimé sous la forme d'un modèle, par exemple en utilisant un système d'expressions régulières. AspectJ permet d'utiliser des caractères génériques (* et ..) pour identifier un ensemble de points d'interception de manière concise.
A un point d'exécution peut être introduit du code dit *greffon (en anglais, advice)*. Celui-ci précise l'association entre un point d'interception, du code et la manière dont ce code contrôle

l'interception. En AspectJ, les greffons sont de nature before, after ou around.
A un aspect peuvent également être associées des *introductions (intertype declarations)* qui consistent à ajouter des références de sous-classages, des attributs ou des méthodes, etc.
L'insertion dans le code de base des greffons est dite tissage ou tramage (en anglais, weaving). Le tissage statique procède par instrumentation du code source ou du pseudo-code machine intermédiaire (bytecode java, IL) tandis que le tissage dynamique intervient lors de l'exécution du logiciel (implémenté par exemple par JAC[PSDF01], Noah).

Programmation par Aspects

Il existe différentes mises en oeuvre de la programmation par aspects [BSL01] : Approche par transformation de programme (AspectJ, programmation par attributs), Approche par transformation d'interprète ([RBY+04]), Approches hybrides (Javassist[9], etc.). Ces travaux ont trait à la programmation générative que nous ne détaillerons pas dans ce document si ce n'est au travers des langages étudiés. Nous invitons le lecteur intéressé par ces aspects à se référer aux travaux de Yannis Smaragdakis et Don Batory [SB00].

Il existe en particulier deux approches permettant d'implémenter la programmation par aspects selon que l'on considère le développement de l'application comme totalement ou non indépendant des aspects techniques.
La première consiste à séparer le code des aspects du code sur lequel ils sont appliqués (appelé code de base). Un langage tiers permet d'établir les relations exactes qui existent entre le code de base et les aspects. Cette approche est celle suivie par AspectJ (cf. http ://www.eclipse.org/aspectj/). Elle favorise l'évolution de l'application et des aspects [KM05], mais rend assez difficile la compréhension du comportement de l'application : "quel aspect s'applique à une classe ?", "Quel est le code résultant ?".
La seconde approche consiste à annoter le code de base afin de préciser les points où les aspects doivent s'appliquer. Cette approche a l'avantage de préciser les points sur lesquels un traitement par aspects est attendu, mais résiste moins bien à l'évolution des aspects. Cette seconde approche est suivie par Microsoft avec l'intégration en standard des Attributes .NET. La mise en oeuvre de cette approche des "aspects", au delà des discussions sur leur validité en tant que Programmation Orientée Aspects, est bien moins puissante que celle proposée par AspectJ. Un travail particulièrement intéressant dans ce domaine est relatif à l'usage des attributs pour faciliter la programmation par composants [RPPM06].

Langages pour la programmation par séparation des préoccupations

AspectJ [KHH+01a] et *HyperJ* [OT00] sont des langages dédiés respectivement à la programmation par aspects et par sujets. Leurs possibilités conjuguées en terme de tissage de préoccupations sont remarquables pourtant quand on les étudie indépendamment on relève un certain nombre de lacunes. Il est particulièrement important pour une meilleure réutilisabilité de définir un protocole de composition qui soit indépendant des futurs contextes d'intégration ; cependant Hyper/J ne le permet pas et Aspect/J seulement partiellement[10]. Par rapport aux adaptations qui sont supportées pour décrire la composition de préoccupations, on peut noter qu'AspectJ ne fournit pas d'adaptations pour fusionner les classes ou les méthodes. Par ailleurs, dans Hyper/J les adaptations relatives à l'interception de primitives sont pauvres : pas d'adaptations pour les attributs et des adaptations insuffisantes pour les méthodes (très

[9] http ://www.csg.is.titech.ac.jp/ chiba/javassist/
[10] Il n'est en particulier pas possible de le faire lorsque les points d'impacts des adaptations sont des classes ou des attributs.

peu d'information contextuelle). Enfin chacun de ces langage n'offre qu'un mode de composition, *in situ* (modification d'une préoccupation existante) pour AspectJ et *ex situ* (production d'une nouvelle préoccupation) pour Hyper/J.
Parmi les autres langages ou approches traitant de AOP et SOP on peut mentionner *ConcernJ* [Car01] ; il s'appuie sur des filtres de composition pour mettre en oeuvre la composition de préoccupations. Il y a aussi *César* [MO03] qui mélange l'AOP avec la programmation orientée composant de telle façon que la spécification des aspects (c'est-à-dire les interfaces) est séparée de leur mise en oeuvre et de leur déploiement. *JAC* [PSDF01] s'inspire aussi des composants mais utilise une réification pour la spécification d'un aspect et ainsi ne nécessite aucune extension de langage. *Jiazzi* [MH03] permet de programmer par des rôles ou par des points de vue, mais la séparation et donc la composition des préoccupations ne peut être réalisée que dans la même classe.
Dans [Qui04, LQ06], les auteurs proposent une approche pour composer les préoccupations d'une application. Cette proposition s'appuie tout d'abord sur l'expressivité que l'on retrouve dans la plupart des langages à objets et qui semble suffisante pour décrire une préoccupation. Elle définie ensuite un métamodèle de composition qui s'inspire à la fois de la programmation par sujets et de la programmation par aspects pour fournir des solutions appropriées pour composer des préoccupations fonctionnelles, non fonctionnelles ou hybrides. Ce métamodèle propose en particulier le concept d'*adaptateur*. Chaque adaptateur est identifié par un nom unique, peut être abstrait ou concret et peut hériter (au sens de la spécialisation) d'un autre adaptateur. Chaque adaptateur référence des cibles d'adaptation (i. e. les entités sur lesquelles les adaptations vont s'appliquer). Chaque adaptation est typée en fonction du type de l'entité sur laquelle elle doit s'appliquer. Il y a donc des adaptations pour les classifieurs, pour les méthodes, pour les attributs, etc. Ces adaptations peuvent être concrètes ou abstraites : les abstraites permettent une spécification *a posteriori* des cibles, au moment de l'application de l'adaptation, quand la cible réelle est connue. Lorsqu'une cible est abstraite, il est possible de lui associer des contraintes qui devront être vérifiées lors de sa concrétisation. Pour concrétiser une cible d'adaptation, trois options s'offrent aux programmeurs : citer une cible, citer une liste de cibles ou encore donner une expression régulière qui identifie un ensemble de cibles. L'intérêt de cette approche par rapport à des approches classiques est qu'elle permet d'associer à une préoccupation réutilisable un protocole de composition qui guidera et contrôlera la composition.

Limites de la séparation des préoccupations

Un des objectifs de la Programmation par aspects est de rendre transparent aux utilisateurs l'intégration de contrôles. Or la composition des aspects intervenant sur un même point d'interception ne peut pas toujours être gérée sans intervention de l'utilisateur. Cette tâche est alors d'autant plus compliquée que tous les aspects n'ont pas nécessairement été définis par lui et que les conflits peuvent apparaître alors qu'il définit simplement de nouvelles classes ou de nouveaux aspects [TBB04]. La prise en compte de l'ordre des adaptations va alors à l'encontre de la séparation puisque par définition les acteurs d'une adaptation ne se connaissent pas nécessairement. Il apparaît donc nécessaire de définir les adaptations selon des algèbres qui gèrent la composition indépendamment de l'ordre des déclarations et détecte les conflits éventuels. Les travaux relatifs aux interactions entre aspects s'intéressent tout particulièrement à ce dernier point [FF05, PSDB04, DFS04, DFL$^+$05]. D'autres travaux relatifs aux compositions de transformations tels que [MKR06] apportent également des solutions à la détection et l'ordonnancement des flots de propagation non déterministes.
Les nouvelles architectures permettent l'introduction de nouveaux composants et par conséquent de nouveaux " types ". Les adaptations en particulier l'introduction de nouvelles fonctionnalités imposent de prendre en compte la possibilité qu'une absence de sous-type com-

mun ne signifie pas qu'il n'y en aura pas, conduisant ainsi au problème du produit cartésien énoncé par Szyperski dans [Szy96].
L'invalidité d'une composition peut impliquer le rejet d'une ou plusieurs adaptations. Il convient donc d'être capable à la fois d'adapter le système en introduisant de nouveaux aspects mais également en retirant des aspects.
Dans [KG02], les auteurs démontrent que les aspects ne permettent pas de gérer les transaction et la concurrence celles-ci n'étant pas des propriétés séparées des entités à contrôler. La limite entre les codes qui sont le métier et ceux relatifs aux aspects reste encore difficile à établir.

4.3.2 Aspects et Modélisation

On peut considérer le tissage de préoccupations aussi bien au niveau du code comme nous venons de le voir que des modèles. On parle alors de tissage de modèles (cf. 4.3.2). Inversement dans un souci de comparaison, complétude et interopérabilité, différents travaux se sont intéressés à la modélisation des langages d'aspects (cf. 4.3.2).

Tissage de modèles

La modélisation des différentes facettes d'une entreprise peut conduire à la mise en oeuvre de modèles de grande taille avec les mêmes inconvénients que ceux mis en évidence pour la conception d'applications de grandes tailles. Il est donc important d'introduire aussi une séparation des préoccupations au niveau des modèles. UML est souvent cité comme un métamétamodèle pour la conception de modèles métiers, Philippe Desfray dans [AD05] propose d'utiliser la fusion de paquetage (*Package merge*) pour composer les différentes préoccupations d'un modèle. La fusion des entités est contrôlée par des préconditions de fusion tandis que la description de la transformation à réaliser pour implémenter la fusion est décrite par les postconditions qui lui sont associées. Dans UML ces préconditions et postconditions de fusion sont prédéfinies et la fusion n'est opérée qu'entre des entités de même nom ce qui entraine de nombreuses limitations. C'est pourquoi les auteurs proposent une extension qui permet en particulier d'étendre le nombre d'entités prises en compte (*components*, *state machines*, etc.) et de permettre à l'utilisateur de renommer des entités et de surcharger les préconditions et postconditions de fusion d'UML par la définition de contraintes OCL définies à cet effet.
Les travaux de S. Clarke [Cla02] s'appuient largement sur la conception par sujets et proposent d'étendre UML afin d'y intégrer une relation de composition. Plus précisément, cette extension modifie la hiérarchie des classes du métamodèle d'UML pour introduire plusieurs catégories d'élément composable (*ComposableElement*) ; les propriétés, les associations, les opérations, les classifieurs et les sujets (spécialisation d'un paquetage) en font notamment partie. La hiérarchie des relations offertes dans UML est, elle aussi, modifiée afin d'intégrer plusieurs catégories de relations de composition à partir desquelles il est possible d'établir une correspondance entre les entités à composer. Naturellement comme pour tout élément du métamodèle UML, la description de leur sémantique est complétée par la définition de contraintes et de règles de validité. Une fois la correspondance établie entre deux éléments composables (ils peuvent avoir ou non les mêmes noms), il est possible d'associer à la composition le comportement d'une fusion ou d'un remplacement. Lorsque les éléments composables sont constitués d'autres éléments (c'est le cas des classifieurs) et si la correspondance entre certains de ces éléments n'est pas explicite alors c'est le nom qui détermine si ces éléments sont candidats à être remplacés ou fusionnés. Lorsque les noms diffèrent, ils sont ajoutés sans changement au modèle résultat. Concernant la fusion il est proposé plusieurs solutions pour résoudre les conflits (précédence, transformation, redéfinition, etc.). Cependant la fusion des assertions

d'une opération ou d'une classe n'est pas vraiment abordée et surtout la composition basée sur une description graphique peut se révéler complexe et n'est pas réutilisable.
D'autres travaux ont pour point de départ UML ou le MOF mais proposent l'ajout de mécanismes complémentaires. Par exemple [CCMV03, CCMV04, MCCV05] proposent une approche basée sur l'idée de paquetage générique (*Template package*). Suivant cette approche, un modèle peut être générique et donc exposer un certain nombre de paramètres qui sont requis. Ces paramètres peuvent être des classes, des propriétés, des opérations ou bien des associations. Ces modèles pourront être composés (opérateur *apply*) successivement avec d'autres modèles, génériques ou non. La correspondance entre les paramètres formels du modèle à composer et les entités du modèle cible est établie au moment de la composition tandis que les autres entités sont fusionnées dans ce même modèle.
Catalysis est une approche pour la modélisation par rôle, basé sur l'UML [DW99]. Catalysis propose de séparer les modèles de conception en utilisant des plans horizontaux et verticaux. Les plans verticaux décomposent les modèles selon le point de vue des catégories d'utilisateur. Les plans horizontaux permettent une décomposition des modèles par rapport aux infrastructures techniques et aux protocoles de communications. Dans Catalysis, la composition de spécifications est basée par défaut sur des entités de même nom. Lorsque l'on veut réaliser une composition sur des entités de noms différents il faut soit faire un renommage d'une des entités soit le spécifier par l'intermédiaire d'invariants supplémentaires et indiquer le nom de l'entité dans le modèle résultant. Lors de la fusion, les assertions quelque soit leur catégorie sont fusionnées par des opérateurs *ET*.
Les compositions abordées dans [FRGG04, SGS$^+$04] se rapprochent de celles de [MCCV05] mais contrairement à ces travaux ainsi qu'à ceux de [Cla02], elles portent plutôt sur des préoccupations transverses. Les modèles représentant des aspects sont en fait vus comme des patrons. Ces derniers sont décrits par des modèles génériques (*UML Model Template*) dont il faut instancier les paramètres pour établir la correspondance avec le modèle à composer. Par défaut les entités de même nom et de même type sont fusionnées en une seule entité mais il est possible d'utiliser un ensemble de directives de composition (suppression, renommage, création, ajout, etc.).
L'approche proposées dans [CL06] s'appuie sur les travaux des mêmes auteurs décrits dans la section 4.3.1. Elle a des objectifs similaires à ceux proposés par [SGS$^+$04] ou [DW99] mais elle s'en distingue notamment par *i)* des différences relativement aux types d'adaptation proposés et *ii)* les mécanismes mis en oeuvre pour que la description de la composition soit considérée comme un savoir faire qu'il faut pouvoir réutiliser le plus rapidement et sûrement possible.
Les travaux de Bernstein and co. dans ce domaine visent à généraliser la composition des modèles en s'appuyant sur la définition de correspondances données ou déduites : opération match qui retourne une correspondance (mapping) entre deux modèles donnés. Les auteurs distinguent alors l'opération "merge" qui retourne un nouvel élément à partir de l'application de la correspondance entre deux modèles et l'opération "Compose" qui retourne la composition de deux correspondances comme une nouvelle correspondance[Pre03, Ber03].
Différents travaux s'intéressent à valider au niveau des modèles les compositions d'aspects portant sur le comportement [MV05]. Il s'agit alors en particulier de caractériser les points de coupes au niveau des modèles. Le travail de Jacques Klein et Franck Fleurey [KF06, KHJ06] qui porte sur l'intégration d'aspects au niveau des scénarii est particulièrement intéressant sur ce point en caractérisant les conséquences en terme de composition des choix de sélection.

Modélisation des langages d'aspects

L'engouement pour la programmation par séparation des préoccupations et plus particulièrement par aspects a conduit à proposer différentes approches (métamodélisation et profils) pour prendre en charge les aspects.

Parmi ces travaux, certains ciblent des langages d'aspects précis tels que AspectJ [BS03, SHU02], tandis que d'autres proposent une approche plus générale qui devrait permettre de représenter au niveau des modèles des aspects sans supposer du langage d'aspect sous-jacent [AEB03]. Notons que lorsque ces approches sont basées sur des extensions d'UML, il semble à la fois y avoir contradiction (un aspect n'est pas vraiment une classe ou un paquetage) et les approches divergent étendant soit classe soit package.

Dans [OH05] les auteurs proposent trois extensions du modèle UML. La première est indépendante d'AspectJ et HyperJ tandis que les deux autres sont respectivement dédiées à ces deux langages. Des transformations de modèles mettent en oeuvre le passage du métamodèle général vers les métamodèles pour Hyper/J et AspectJ.

4.3.3 Hétérogénéité et Aspects : Le système Noah

Noah est un framework dans lequel les entités logicielles hétérogènes peuvent interagir sans se connaître[BFCE+04]. Les interactions entre composants sont décrites dans un langage indépendant des cibles : Interaction Specification Language (ISL). Le langage ISL permet de décrire des aspects au moyen de règles. Chaque règle spécifie, pour un message reçu donné ou un ensemble de messages via le caractère *, le comportement attendu. Le corps d'une règle se définit en utilisant les opérateurs suivants ont été définis : l'envoi de message, la séquence, la concurrence, l'affectation, la condition, la gestion d'erreurs, etc.
Noah peut être vu comme un repository d'aspects dynamiques dont le tissage s'appuie sur un mécanisme de composition qui respecte la commutativité et l'associativité, i.e. que quelque soit l'ordre d'application d'un ensemble de contrôle en un même points d'interception le résultat est équivalent[Ber01].
Le service Noah peut gérer des objets Pur Java, des composants Java RMI components, des composants Jonas 2.5 EJB, des composants Fractal et des composants Microsoft .NET. D'anciennes mises en oeuvre gênèrent du code pour C++/Corba (Micado), Java /RMI (JavaInt) et Distributed Smalltalk (DSTInt).
Les avantages de Noah, autres que ceux traditionnels à une approche par aspects, sont :
– des aspects dynamiques entre instances. Un aspect peut être défini à tous moments pendant l'exécution d'une application. L'utilisateur peut poser et retirer des aspects sur des instances données en cours d'exécution ;
– l'indépendance des langages des cibles. Le langage de spécification des aspects (ISL) est indépendant des plates-formes sur lesquels les composants contrôlés sont définis ;
– l'hétérogénéité des composants. Il est possible de faire interagir des composants EJB, .net, RMI.
L'intégration des principes Noah dans une architecture orientée Services et son implémentation pour des web services .Net est en cours d'étude.

4.3.4 Architecture et Séparation des préoccupations : le système Transat

TranSAT (Transformations for Software Architecture) propose de gérer les évolutions d'une architecture logicielle décrite sous forme d'une architecture à base de composants suivant le modèle proposée par SafArchie [BD04] [BMP05] [BLMD06] (cf. 3.2.4). TranSAT est un cadre de conception pour l'intégration de nouvelles fonctionnalités au sein d'une architecture logicielle existante. Inspiré par le développement de logiciels par aspects, TranSAT intègre une dimension supplémentaire dans la description d'une architecture logicielle par la définition d'un patron d'architecture. Celui-ci représente une unité de structuration pour les préoccupations transverses à intégrer. Il contient (i) un assemblage de composants, appelé plan, qui

prend en charge les fonctionnalités nouvelles liées à la préoccupation à intégrer, (ii) une description des éléments devant être présents dans l'architecture de base afin de pouvoir tisser le plan, appelé masque de point de jonction et (iii) un ensemble de règles de transformation qui définissent les modifications à apporter sur le plan initial pour l'intégration de la préoccupation. Le patron d'architecture définit un mécanisme de transformation qui permet d'associer deux plans d'architecture, l'un de base et l'autre représentant la nouvelle préoccupation à intégrer. Ce mécanisme de transformation contient à la fois des primitives de transformation et qui modifient l'assemblage et des primitives de transformation qui sont intrusives et qui modifient l'interface et le comportement des composants. Ce patron est fait pour être réutilisable pour différents plans de base grâce à une spécification claire des attentes du patron vis-à-vis de son contexte d'intégration via le masque de point de jonction. De plus, l'intégration se fait par un mécanisme de tissage dont la sémantique est explicitée par des règles de transformation.

4.3.5 Spoon : Transformation de programmes dirigée par les annotations

Spoon[Paw05] est un framework pour la transformation et l'analyse statique de programmes Java 5 ou compatibles. Plus précisément, Spoon est un compilateur Java extensible et ouvert, lui-même écrit en Java, et qui utilise les techniques de réflexivité à la compilation. Il propose une API générale de traitement de programme, incluant une spécialisation pour analyser les annotations Java 5. L'une des caractéristiques de Spoon est l'utilisation des *generics* Java, qui permettent le typage des "processeurs". Ce typage fort permet un meilleur support des environnements de développement (vérifications syntaxique ou sémantique, complétion, documentation, navigation, etc.), et le support d'un mécanisme de *templates* en Java standard, qui est utilisé comme base de la programmation des transformations de programme. Ainsi, Spoon permet la métaprogrammation à base de templates correctement typés (typique de la Programmation Générative), à la Haskell ou C++, sans pour autant nécessiter l'utilisation de langages autres que le Java 5 standard. Cette caractéristique le rend plus simple à intégrer dans un processus de développement, et en particulier pendant les phases d'implémentation des composants. Par exemple, Spoon est utilisé pour l'implémentation de composants Fractal écrits en Java et annotés par des annotations définies par le DSL Fraclet.
Spoon comprend aussi un ensemble d'extensions ou de sous-projets qui permettent de cibler des utilisations plus spécifiques que la transformation de programme en général. Spoon-AOP est une extension de Spoon qui permet la définition d'aspects en Java 5 (programmes annotés) qui sont particulièrement efficaces et correctement typés. AVal est une application de Spoon qui permet la validation syntaxique et sémantique d'un ensemble d'annotations formant un DSL (par exemple Fraclet, annotations EJB3, JSR 181 - pour les Web Services). JDiet est un outil Spoon qui transforme des programmes J2SE vers des programmes compatibles avec J2ME CLDC (Connected Limited Device Configuration), ce qui permet de déployer des programmes Java vers des PC de poche.

4.3.6 AOKell

AOKell [SPDC06] est une implémentation du modèle de composants Fractal [BCL+04a]. Par rapport aux autres implémentations de ce modèle, AOKell est originale dans sa façon de traiter, à l'aide d'aspects, la composition des codes fonctionnels et non fonctionnels.
Deux niveaux sont en général pris en compte dans une application Fractal : le niveau de base qui implémente la partie métier, et le niveau de contrôle qui est chargé de la supervision et de la gestion du niveau précédent. Ce niveau est défini dans une membrane dite de contrôle. Celle-ci est composée d'un ensemble d'entités élémentaires, les contrôleurs, qui fournissent au composant des services techniques non fonctionnels. La granularité de ces services peut

être quelconque, allant d'un simple service de nommage, à un service de cycle de vie ou à un service transactionnel.
Le rôle d'un framework conforme aux spécifications Fractal est donc de composer ces services non fonctionnels avec la partie métier. Pour effectuer cette composition, AOKell utilise des aspects. Chaque contrôleur est associé à un aspect et l'ensemble des aspects est tissé sur le code des composants. Les aspects effectuent soit des injections de code, soit de l'interception du code existant. Il s'agit, soit d'augmenter le comportement du composant, soit de le modifier en fonction de la définition de la partie non fonctionnelle.
Initialement, les aspects utilisés par AOKell pour effectuer cette composition ont été développés avec AspectJ qui est un compilateur d'aspect pour Java [KHH+01b]. Plus récemment, une seconde version de ces aspects a été développée avec le transformateur de code Spoon (voir 4.3.5). Cette dernière version fournit des temps de compilation et d'exécution meilleurs qu'avec la version AspectJ.
Une deuxième caractéristique originale d'AOKell réside dans sa façon de développer le niveau de contrôle. Alors que les implémentations existantes du modèle Fractal utilisent des objets pour cela, AOKell a introduit la notion de composant de contrôle. Les services non fonctionnels associés à un composant sont ainsi implémentés sous la forme d'un assemblage de composants de contrôle qui est tissé sur les composants de base à l'aide d'aspects.

4.3.7 FAC

FAC (Fractal Aspect Components) [PSDC06] est un modèle de programmation qui étend le modèle Fractal et permet de combiner les styles de programmation à base de composants et d'aspects. Alors que jusqu'à présent la notion d'aspect a essentiellement été étudiée en relation avec les objets, la proposition FAC est originale au sens où elle envisage cette notion au niveau des composants.
FAC introduit trois artefacts pour développer des applications à base de composants et d'aspect : composant d'aspect, domaine d'aspect et liaison d'aspect. Ces trois artefacts n'introduisent pas de concepts supplémentaires dans le modèle Fractal. Le composant d'aspect est un composant primitif qui implémente le comportement d'un aspect (i.e. le code advice). Le domaine d'aspect est un composant composite qui regroupe un composant d'aspect et l'ensemble des composants qui sont impactés par cet aspect. La liaison d'aspect est une liaison entre un composant et le composant d'aspect : elle réifie le chemin de communication emprunté lorsqu'un aspect intercepte un point de jonction et y applique son comportement.
Deux implémentations du modèle FAC existent : une pour Julia, l'implémentation de référence du modèle Fractal, et une pour AOKell. Le principe de ces deux implémentations est identique : un contrôleur est fourni pour gérer le tissage et l'ADL est étendu pour décrire les composants à tisser et les localisations où ces composants doivent être tissés.

4.3.8 MicM : Intégration de services indépendamment des plates-formes

Les plates-formes à composants permettent de développer des briques logicielles réutilisables. Ces briques logicielles contiennent le code métier de l'application tandis que la plate-forme d'exécution se charge de fournir et gérer le code technique (authentification, transactions, persistance, notification etc.). Le besoin croissant des applications en terme de nouveaux services fait émerger un nouvel acteur : le fournisseur de services. Son rôle est d'intégrer de nouveaux services dans les plates-formes à composants. Le fournisseur de services doit faire face à l'hétérogénéité des plates-formes à composants, à la complexité des générateurs qui produisent le code technique et à la composition des différents services. Il ne bénéficie d'aucun support pour intégrer de manière homogène un service dans différentes plates-formes à composants ni pour composer les services. Dans la thèse de Olivier Nano[Nan04], l'auteur propose un

modèle d'intégration de services indépendant des plates-formes à composants qui permet de décrire de manière abstraite l'intégration de services. Ce modèle supporte un système de composition automatique des intégrations de services qui permet de détecter des conflits d'intégration et un processus de projection des intégrations de services dans les différentes plates-formes à composants.

4.3.9 Analyse de la composition par combinaison de code

Cette forme de programmation connaît un engouement certain. En effet, elle apporte un niveau d'abstraction supplémentaire, qui favorise à la fois la robustesse du développement, la rapidité de production et la réutilisation des codes [KM05]. Ces critères sont vrais pour autant que l'on associe à ces techniques des outils de validation et que l'utilisateur est confiance dans leur production, i.e. les codes générés et les compositions des codes.

Adaptation

La maintenabilité est accrue parce que le code métier et les codes techniques relatifs aux aspects sont maintenus indépendamment des autres. On peut donc s'attendre à une meilleure réutilisabilité puisque la définition de l'application est sensée être indépendante des aspects et que les aspects peuvent également être réutilisés indépendamment de l'environnement d'exécution. La définition d'aspects abstraits dans aspectJ renforce ce point. L'évolution des applications est donc facilitée puisque chaque aspect se préoccupe d'une fonctionnalité précise. Peu de travaux abordent néanmoins l'adaptabilité dynamique des applications par introduction d'aspects.

Validation

Les aspects sont souvent utilisés pour introduire le code nécessaire au monitoring et au contrôle des applications. En cela, ils sont une aide à la validation des contrats.
La difficulté est d'analyser le code généré lors des phases de mise au point des logiciels (débogage, test). Les outils autour des langages tels que ceux définis autour de AJDT, basé sur AspectJ, permettent néanmoins de passer de façon transparente, en mode débogage, du code d'une classe à celui d'un aspect.
La composition des aspects reste également problématique car centrée sur une approche "programmative" qui rend difficile le développement par réutilisation de codes aspectisés ou sans connaissance des autres [FF05]. Il s'agit dans le cadre d'un usage des aspects pour notifier la validation des contrats de proposer des compositions adaptées des aspects.

Au niveau des modèles, la conception par séparation des préoccupations favorise la collaboration et l'enrichissement du modèle de base, tout en opérant des contrôles non supportés par les ateliers standard de modélisation qui reposent sur la seule sémantique d'UML. En enrichissant ces capacités de validation, la modélisation par séparation des préoccupations laisse envisager une intensification de cette forme de développement. Elle allie déclarativité et génération de code, de la sorte que l'expressivité et la robustesse des applications y gagnent. Néanmoins cette puissance potentielle ne sera réalité que si les outils développés assurent des propriétés de composition "intuitive", démontrables et probablement non dépendantes d'une relation d'ordre des déclarations. La détection des conflits de composition est alors essentielle.

Nous n'avons pas abordé dans cette section l'application de la programmation par séparation des préoccupations aux Web Services nous l'aborderons au paragraphe 4.4.5.

4.4 Composition dans les architectures orientées services

Collaboration : Action de travailler de concert avec un ou plusieurs autres (wikipedia).
Une fois généralisé à l'ensemble du système d'information, le dispositif de communication universelle proposé par l'architecture SOA permet en principe de réutiliser et de combiner à loisir les applicatifs métier, au sein de processus par exemple, et ceci de façon très réactive. En effet, l'intérêt principal d'une structuration d'un S.I. en services est d'autoriser la composition de services pour atteindre un objectif fonctionnel. Cette composition constitue un processus dont le contexte est plus ou moins étendu (en général, ce contexte est plus restreint que pour les processus métier qui constituent l'ossature du S.I., voire limité à réaliser un objectif "local", placé sous la responsabilité d'un seul acteur).
Ainsi l'approche SOA favorise l'enchaînement des services selon un *mode d'orchestration* : les services sont en couplage faible (loosely coupled) contrairement à l'approche orientée objet et composants qui procèdent par propagation [Bon05]. Cette dernière favorise l'enchaînement des services par des appels directs ou via des ports requis à un autre service et ainsi de suite suivant un niveau de profondeur non limité. Les services sont alors en couplage fort (tightly coupled). Les deux modes de séquencement doivent cohabiter : une implémentation complète en orchestration n'est pas envisageable car le niveau de complexité des fonctions d'orchestration deviendrait trop important.Nous avons déjà présenté le couplage fort au paragraphe 4.2. Nous nous intéressons dans cette partie à la composition par orchestration de services et à l'usage de la programmation par séparation des préoccupations pour prendre en compte l'introduction de propriétés transversales dont la qualité de services.
Vis à vis de la composition des services, nous pouvons considérer deux sortes de services :
– Atomic Service : Il se définit par une unique invocation qui déclenche son exécution sans nécessité de nouvelles interventions. Lorsque ce service n'invoque pas un autre service, on parle de "component service".
– Workflow (conversional) Service : lorsque les informations en entrées ne peuvent pas être fournies en une seule fois, le service se présente comme une machine à états finis.

Nous aborderons la composition dans les architectures orientées services selon les axes suivants :
– en 4.4.1, workflow : vocabulaire, objectifs, problématiques,
– en 4.4.2, composition dans le contexte des Web services (orchestration et chorégraphie),
– en 4.4.3, dynamique dans les assemblages de services,
– en 4.4.4, prise en compte de la qualité de service dans les orchestrations,
– en 4.4.5, travaux relatifs à l'utilisation des aspects dans les orchestrations.

4.4.1 Workflow

(Voir dans Reference Model for Service Oriented Architecture 1.0 Public Review Draft 1.0, 10 February 2006) [MLM+06]
Le terme anglais " workflow " veut littéralement dire " flux de travail ". Cette forme de travail implique un nombre de personnes limité devant accomplir, en temps limité, des tâches articulées autour d'une procédure. Il y a deux grands types de workflows : le " workflow humain " et le " workflow programmatif ". Le premier assure de manière transparente le suivi des tâches et leur traçabilité : qui fait quoi, quand et comment. Un non informaticien peut gérer ce type de workflow à travers un outil performant. Le second type, plus complexe, s'occupe de l'enchaînement nécessaire au cours de l'ensemble du processus. Nous nous intéressons uniquement à ce type de workflow dans ce document.
La description d'un workflow caractérise les relations temporelles entre les interactions sur les services. La description d'un workflow se fait à l'aide d'un langage dédié. Certains travaux, particulièrement dans le domaine des grilles de calcul, étendent ces notions élémen-

taires pour intégrer : des stratégies d'itération, des contraintes de synchronisation et la gestion des erreurs.

Le workflow a la responsabilité de :

– la gestion de contexte : maintien d'information entre les services.
  Le worflow prend en charge la conservation de certaines données qui sont nécessaires tout au long de la durée de vie d'un enchaînement. Par exemple, il peut s'agir de mémoriser un résultat qui est réutilisé plus tard dans le workflow. Les données mémorisées forme un contexte. Il est créé au moment du lancement du workflow et se détruit lorsqu'il se termine. Selon le principe du couplage faible, chaque élément assemblé ignore l'existence des autres. Le contexte est ainsi composé de l'ensemble des éléments de l'infrastructure, des *processes*, des contrats qui interviennent dans une interaction avec un service. Dans le cadre des applications mobiles, la notion de contexte est essentielle et intervient en conséquence fortement la composition des services. Du côté des Web services, le contexte sera à terme traité par la spécification WS-Context(http ://xml.coverpages.org/WS-ContextCD-9904.pdf).
– Gestion transactionnelle : commit à deux phases.
  Les éléments assemblés lors de l'orchestration peuvent émettre des mises à jour dans les bases de données. Puisque ces éléments ne se connaissent pas entre eux, ils sont incapables de se synchroniser. C'est le workflow qui prend en charge la gestion d'une unité transactionnelle globale.
– Gestion de la logique applicative.
  Au-delà des rôles techniques de gestion de contexte et de transaction, le workflow assure un rôle applicatif. Il contient la logique fonctionnelle d'assemblage des éléments et uniquement cette logique. Les règles métiers restent localisées dans les éléments assemblés.

Parallélisme d'une application

Dans le cas d'une exécution d'une application sur une grille de calcul, sa description sous la forme d'un workflow en fournit une parallélisation naturelle. Trois types de parallélisme sont exploitables. Le premier et le plus direct d'entre eux est le *parallélisme de workflow*. Il correspond à l'exécution parallèle de deux services indépendants du workflow. De plus, l'exécution d'un même service sur des données différentes et indépendantes conduit au *parallélisme de données*. Ce type de parallélisme est fréquent dans application tirant parti des grilles de calcul, où les services doivent gérer de grandes masses de données. Enfin, deux services liés séquentiellement peuvent s'exécuter en parallèle sur plusieurs jeux de données indépendants. On parle alors de *parallélisme des services* ou de *pipelining*. Exploiter ces différents types de parallélisme permet un déploiement simple et efficace d'une application sur une grille de calcul [GMP+06].

Gestion des erreurs

Le traitement des erreurs est un cas critique en ingénierie du logiciel. Il en est de même dans le monde des workflows où il est essentiel de prendre en compte les échecs des activités, surtout si leurs exécutions sont liées à la continuation de l'exécution. Il est donc important pour la robustesse du workflow qu'il permette la définition de mécanismes de gestion des erreurs. Ces mécanismes peuvent simplement détecter l'erreur et la traiter en soumettant à nouveaux les tâches échouées ou mettre en oeuvre des actions de compensation. Ces actions de compensations permettent en cas d'erreur d'annuler ou de réparer les effets d'une tâche ayant échouée. Le traitement des erreurs permet de garder l'exécution du workflow cohérente et consistance même en présence d'erreurs.

Langages de description d'un workflow

Outre les langages dédiés à la composition de Web services (cf. 4.4.2), des langages de flots ont été définis par la communauté e-Science, tels que Scufl (Simple Concept Unified Flow Language) ou MoML (Modeling Markup Language). Ces langages sont conçus pour la description du flux de données. Des opérateurs de composition de données d'entrée tels que *dot* et *cross product* permettent en particulier une description précise de la manière dont un service est itéré sur ses entrées. A la différence des langages issus de la communauté du e-Business tels que BPEL, ce type de langages n'intègre que quelques opérateurs de contrôle sommaires, la logique de l'application étant décrite à l'intérieur des services eux-mêmes.

Moteur de workflow

L'exécution de la description d'un workflow sur un jeu de données d'entrée est assurée par un moteur de workflow, responsable en particulier de la transmission des données entre les différents services. De nombreux moteurs de workflow sont utilisés pour le déploiement d'expériences scientifiques "in-sillico" dans des domaines applicatifs variés. Taverna [OAF+04], issu de la communauté bioinformatique, permet la composition d'applications intégrant des services de nature variée et est interfacé avec des outils permettant leur découverte sémantique. Kepler [LAB+05], issu de l'environnement de composition Ptolemy, offre différents modes d'invocation (directors) qui permettent par exemple la simulation de workflows s'exécutant en temps discret. Triana [TWSM05], développé à l'origine par un projet de détection d'ondes gravitationnelles, se distingue par différents modèles d'exécution sur grille de calcul (parallèle, peer-to-peer). Enfin, MOTEUR [GMP+06], développé dans l'équipe Rainbow, se concentre sur l'exploitation des différents types de parallélismes pour le déploiement d'applications sur une grille de calcul.

Formalisation des Workflows et Diagrammes d'activités

Différentes techniques formelles peuvent être utilisées pour modéliser et valider des workflow : algèbres de processus [ABV04], LOTOS (CADP) [GS04] ou automates [EBR00], si ce n'est la théorie des actions [DBM04, DBLM03].
Une technique de modélisation qui semble particulièrement appropriée est les diagrammes d'activités d'UML2 [OMG04]. Ces diagrammes permettent de spécifier un ensemble d'activités et leurs enchaînements. La notion de swim lanes permet de regrouper dans un même bloc vertical les activités qui seraient supportées par la même plate-forme, ou ensemble de plates-formes (ou composants). La notion d'activités composites est également particulièrement intéressante, elle permet de donner d'une activité une vue boîte noire (dans une activité englobante) et une vue boîte blanche détaillant le workflow interne. Par comparaison, les diagrammes d'états donnent une vue seulement partielle du comportement d'un système. Un diagramme d'état montrant en général le comportement d'un seul process, il est alors très difficile de comprendre l'enchaînement entre processes en parallèle. Les diagrammes d'états sont alors peu appropriés à l'orchestration, et surtout à la chorégraphie. Les diagrammes de séquences montrent bien le comportement sur l'ensemble d'un système, mais ils deviennent vite illisibles lors de l'introduction de conditions ou de boucles. Différents travaux ont donné une sémantique formelle aux diagrammes d'activités, sémantique en général basée sur des variantes des réseaux de Petri ou des systèmes de transition [Esh99, EW04, JLGC04]. Les Réseaux de Petri conviennent particulièrement aux diagrammes d'activités. On peut cependant souligner des restrictions importantes apportées aux diagrammes d'activités, en particulier sur les activités composites, les Forks (Lancement en parallèle de tâches) et Join (resynchronisation de tâches).

Donner une sémantique formelle et exécutable aux diagrammes d'activités oblige à mettre des restrictions sur l'utilisation de toutes les possibilités d'UML2. Une telle sémantique doit être basée sur un compromis entre possibilités d'expression et possibilités de calcul. Le projet RNRT/PerSiForm [Per] s'emploie à fournir une sémantique aux diagrammes d'activités basée sur une variante des RDP stochastiques, colorées et temporisées, sémantique qui inclus l'utilisation des activités composites.

4.4.2 Orchestration et Chorégraphie dans le cadre des Web Services

Les Web Services sont aussi utilisés comme axes d'échanges entre des sous-systèmes hétérogènes du S.I d'entreprises (voir précédemment ESB). Dans ce contexte d'utilisation deux points de vues sont possibles :
- la spécification externe, qui décrit l'enchaînement des WS et les rôles attachés à l'utilisation d'un WS : c'est la *chorégraphie*,
- la réalisation interne des échanges entre WS contribuant, pour le compte d'un partenaire donné, à la réalisation de la chorégraphie, que l'on appelle *orchestration*.

La présentation des orchestrations et des chorégraphies a été faite au chapitre 2. Dans ce chapitre, nous ne distinguerons pas ces deux formes d'assemblages en nous intéressant de manière plus générale à leur composition.

Composition des orchestrations

Comme nous l'avons vu précédemment (cf. 2.3.2) une orchestration est une composition de services. Les orchestrations, en particulier si l'on considère les éléments non-fonctionnels (sécurité, tracing, cryptographie, etc.) comme des services, mettent en jeux des connaissances et entrelacements de services. Elles sont alors particulièrement difficiles à composer pour prendre en compte la gestion d'erreurs, de contextes, d'ordonnancement des appels, etc.
Cette composition peut être vue sous deux angles.
- Le premier en considérant les éléments du système (services et orchestration) comme des boîtes noires accessibles uniquement via leur interface. Dans ce cas, la définition du résultat d'une orchestration comme nouveau service permet une composition récursive des orchestrations [KMW03]. Cependant cette composition pose des problèmes d'appels multiples aux services communs, d'absence de partage du contexte de gestion des erreurs délocalisées, etc. [Nem06]
- Le second en considérant les orchestrations comme des boîtes blanches qu'il s'agit de composer par tissage de code. Dans ce cas, la composition d'orchestration est une tâche plus complexe, qui suppose une bonne connaissance des services et orchestrations existantes.

Approches formelles des langages d'orchestrations

Afin de permettre la validation des assemblages de services, différents travaux tendent à dériver du langage BPEL4WS des vérifications formelles.
Parmi ces travaux, dans [CCCV05, Mar05] les auteurs proposent une transformation du langage WSBPEL vers la notation CSS pour l'un et les réseaux de pétri pour l'autre. Les auteurs utilisent cette formalisation pour vérifier la "compatibilité des services" à assembler, ou la propriété d'interchangeabilité des services afin de remplacer des services en prenant en compte à la fois les aspects "syntaxiques" et le comportement des services. Notons que ce travail exige une formalisation de l'ensemble des services mis en jeu dans une orchestration[Vir04, FUMK04].

4.4.3 Adaptation dynamique des assemblages

Lorsque les assemblages de services sont définis statiquement, ce qui est aujourd'hui le cas avec les approches relatives à BPEL, il n'est pas possible d'adapter la liaison à un service. Or cette adaptation est nécessaire pour, par exemple, sélectionner le service approprié en fonction des choix utilisateurs tels que le choix du service le moins cher, ou prendre en compte de nouveaux services[CIJ+00].
Dans le monde des services, nous retrouvons donc les mêmes paradigmes relatifs à la découvertes des services que dans le monde des composants. Il s'agit alors de définir des règles de sélection des services. A notre connaissance, ce point est aujourd'hui peu étudié dans le cadre des Web Services, l'adaptation reposant principalement sur la construction de nouveaux services par la définition d'orchestrations. Néanmoins les travaux relatifs au web sémantique abordent cette problématique via l'usage d'ontologies [PSSN03] qui permettent de déterminer en fonction d'une tâche les services les mieux adaptés ou en cas d'échec pour substituer de nouveaux services répondant mieux aux exigences utilisateur[BW03]. En fonction des approches, il s'agit donc de prendre en compte soit de l'adaptation par l'utilisateur soit de l'auto-configuration, les auteurs parlent alors d'auto-organisation [PB05].

4.4.4 Qualité de Service dans les compositions

Qualité de Service et les Web services

Dans le cas des Web services, La "Qualité de Service" (QoS) est une notion cruciale qui réfère aux multiples propriétés de qualité lors du fonctionnement du Web service. Il s'agit principalement de propriétés telles que la performance, la confiance, le passage à l'échelle, la capacité, la robustesse, la sécurité, etc. Avec la prolifération des Web services, la QoS devient un critère décisif dans le choix du service approprié parmi des offres de plus en plus compétitives et aux fonctionnalités souvent similaires. Dès lors elle constitue une priorité pour les fournisseurs de services et leurs partenaires, puisque chaque service possède ses propres caractéristiques de QoS, et chaque client possède des exigences spécifiques.

Toutefois, contrairement aux normes bien établies dans le domaine fonctionnel des Web services (WSDL, SOAP, UDDI), il n'existe pas de spécifications officiellement reconnues par la communauté en ce qui concerne la QoS. En particulier, il n'existe donc pas de standards en ce qui concerne la publication de la QoS d'un Web service, ni en ce qui concerne l'expression de la requête du client. Pour autant, de nombreux travaux se sont déjà essayés à proposer des formalismes et des infrastructures pour répondre à cette problématique. Il s'agit notamment des travaux effectués autour des SLA et de leur infrastructure de gestion.

Qualité de Service dans les Compositions de Web services

Lorsque l'on compose des Web services disponibles sur le Web, il devient particulièrement délicat de prédire la QoS de l'assemblage obtenu. En effet, les propriétés de QoS d'un service composite résultent de la complexe agrégation de la QoS des services sous-jacents. Pour autant la composabilité des Web services est un de leurs principaux atouts, et la vision actuelle du marché s'oriente vers la création de services complexes créés à partir de services de base. Les enjeux liés à la garantie de QoS des compositions de services sont donc tout aussi important que l'assemblage fonctionnel des services. Par exemple, si un service d'achat d'actions en bourse dépend d'autres services qui sont trop longs à invoquer ou bien même qui peuvent échouer quand il y a trop de requêtes, alors l'assemblage sera aussi inutile qu'un service qui

utiliserait des valeurs inexactes.

La garantie de la QoS des compositions de services constitue un verrou technologique majeur auquel la Recherche tente d'apporter des réponses. De nombreux travaux se sont intéressés aux langages dédiés à l'exécution de processus métier. Ces langages appartiennent à la classe des "Business Process Execution Language" (BPEL). De part son expressivité limitée, la spécification commune qui en est issue, le langage BPEL4WS, dédié à l'aspect fonctionnel des compositions ne prend pas en compte les paramètres de QoS. Ainsi, de la même façon qu'il n'existe toujours pas de spécifications bien établies dans le domaine de la QoS entre service et client, il n'existe pas non plus ni langage, ni protocole d'interaction, ni infrastructure encore clairement identifiés pour spécifier la QoS dans les compositions de Web services. Ce constat est le cœur du projet FAROS.

4.4.5 Application des aspects aux orchestrations

AO4BPEL

Les travaux d'Anis Charfi et Mira Mezini [CM04] portent sur l'élaboration d'un mécanisme de composition faisant intervenir des aspects rédigés sur la base du langage BPEL4WS. Leur analyse se fonde sur le constat de deux lacunes du langage BPEL4WS : d'une part les compositions manquent de modularité, notamment en ce qui concerne les préoccupations "éparpillées", et d'autre part les compositions demeurent figées du fait que le BPEL4WS spécifie des informations statiques. Pour apporter des solutions à ces limitations, une extension de BPEL4WS est envisagé en introduisant des mécanismes orientés aspect. Le résultat est l'élaboration du langage AO4BPEL (Aspect Oriented for Business Process Execution Language) offrant plus de modularité pour la spécification de compositions de Web services et un support d'adaptation dynamique des compositions. Pour que les aspects définis avec ce langage puissent être tissés dynamiquement dans le document BPEL4WS, les auteurs ont choisi d'utiliser les "activités" du BPEL comme possibles join points. Étant donné que ce langage est spécifié en XML, le langage de coupe est basé sur XPath qui permet de faire des requêtes sur des documents XML. Une coupe peut donc référencer de multiples activités, tandis que les advices encapsulent des instructions BPEL4WS. Ce choix, plutôt que celui d'un autre langage d'advice, est justifié par le fait que les auteurs ne souhaitent pas corrompre la portabilité du BPEL4WS. Pour palier au manque d'expressivité de ce langage pour l'implémentation des advices, ils évoquent la possibilité de faire des appels vers un "Web service d'infrastructure" qui contiendrait la véritable logique de l'advice. Plus récemment, les auteurs ont travaillé sur la possibilité de faire intervenir un aspect de sécurité dans les compostions de Web services [CM05b, CM05a] en ayant également recours à des artifices tels que l'appel à des Web services d'infrastructure. L'idée de ce travail est de rajouter des appels vers un service qui s'occupe de gérer la sécurité et d'autres paramètres de QoS.

Tissage d'aspects dans un moteur BPEL

Les travaux de Carine Courbis et d'Anthony Finkelstein [CF05a, CF05b] sont similaires à l'approche de l'AO4BPEL. L'origine de ces travaux repose sur le souhait des auteurs de construire un moteur BPEL aussi minimal que possible, mais facilement configurable et extensible. Les nouvelles fonctionnalités apportées par ce moteur permettent d'étendre et modifier facilement son comportement, sélectionner et remplacer aisément les Web services après le déploiement de la composition, tisser ou bien détisser des préoccupations transversales, et en-

fin permettre le remplacement à chaud du workflow (et ainsi autoriser des compositions à la volée).
Ces travaux évoquent la possibilité de mélanger des contraintes de QoS avec la composition (notamment en sélectionnant le service approprié). Comme pour l'AO4BPEL, un langage d'aspect (ou domain-specific aspect language) a été développé spécifiquement pour le BPEL4WS. La syntaxe est différente de celle de l'AO4BPEL mais la différence la plus importante vient du fait que deux types d'aspects peuvent être implémentés. D'une part des aspects "statiques" dont le rôle est d'injecter dans le moteur BPEL des préoccupations de type monitoring, débogage, sélections de Web services après déploiement. Les advices de ces aspects sont alors rédigés en Java. Le second type d'aspect, dit "dynamique", a pour objectif de modifier la structure de la composition à l'exécution. Cette fois-ci le tissage ne s'effectue plus sur le moteur BPEL mais sur le document BPEL lui même.

Web Service Management Layer

La Vrij Universiteit Brussel a réalisé une plate-forme de communication entre client et Web service. Cette couche nommée "Web Service Management Layer" [VVJ06] a pour mission d'encapsuler tout le code client relatif à des préoccupations de gestion, comme par exemple la sélection du service le moins cher, la communication asynchrone ou encore l'optimisation du trafic. Les auteurs justifient la création de ce dispositif par le fait que le lien client service est un lien statique ne faisant pas intervenir d'autres paramètres que la description fonctionnelle du service. La couche WSML fait intervenir la programmation par aspect pour la mise en place des préoccupations transverses.

Il est dans les intentions des auteurs d'étendre ce travail à la gestion de la composition de Web services, notamment dans le cadre d'orchestrations spécifiées en BPEL4WS. Dans cette perspective, une première proposition pour la prise en charge de règles métier au sein des compositions est exposée dans [CV05]. Les auteurs ont pu observer qu'il n'existe aucun support pour la définition de règles métier au sein des langages de composition. Ici encore, le manque d'expressivité et de modularité du BPEL4WS ont orienté leurs efforts vers l'utilisation de l'AOP pour tenter de résoudre les lacunes. Plus exactement, l'approche empruntée consiste à isoler la logique des règles métier dans des aspects JasCo [VSV+05].

Projet Self-Serv

Le projet Self-Serv de Quan Z.Sheng, Boualem Benattallah et Marlon Dumas [BSD03] souligne tout d'abord la grande difficulté actuelle dans l'élaboration de compositions de Web services, qui souffrent d'un besoin constant de développement *ad hoc*. Ils jugent cette lacune inacceptable compte tenu de la taille et de la volatilité du Web. Leur solution consiste en une plate-forme permettant une composition plus aisée des Web services. Pour cela, les compositions sont rédigées avec un langage déclaratif basé sur les diagrammes à état, tandis que le concept de "communauté de services" crée un intermédiaire entre la composition et le service applicatif, en sélectionnant le fournisseur le plus approprié. Une communauté de services est une collection de Web services avec des fonctionnalités similaires mais des QoS différentes. L'exécution des services composites est contrôlée par des composants "coordinateurs" dont le but est d'initialiser, contrôler et *monitorer* l'état de la composition qui lui est associé.

4.4.6 Analyse de la composition dans les architectures orientées services

Les architectures orientées services sont nées de l'intensification de l'usage de l'informatique et des réseaux, et plus particulièrement du Web. De plus en plus d'entreprises utilisent cette technologie pour délivrer de nouveaux services. De la sorte, le développement par composition des services est particulièrement productif puisque de plus en plus de services se trouvent accessibles et que les exigences en matière de plate-formes d'accueil se limitent au support de XML et éventuellement soap. Dans ce contexte les langages de composition jouent un rôle essentiel.
Dans ce chapitre, nous nous sommes plus particulièrement intéressés à l'adaptation des assemblages et à leur validité. En particulier, nous constatons aujourd'hui que les problèmes récurrents dans le contexte des compositions de Web services sont la prise en compte de propriétés d'authentification, la validation des assemblages, la récupération d'erreur en fonction du contexte et aujourd'hui une nécessaire adaptation avec la découverte dynamique de services, la sélection de services et la prise en compte des interactions avec l'utilisateur afin de rester compétitifs dans un environnement hautement dynamique et pour autoriser une spécialisation en fonction des clients.
La standardisation de BPEL n'est pas finalisée et les vendeurs proposent des extensions par exemple pour faciliter la gestion des interactions avec l'utilisateur. Dans ce contexte, nous devons aborder adaptation et validation des architectures orientées services à la fois en nous appuyant sur les standards et en nous en détachant, via des modèles, pour assurer la pérennité de notre travail.

Adaptation

La démarche SOA présente l'avantage de proposer une méthodologie de développement dans laquelle les couches de services se construisent en minimisant, en principe, les dépendances entre les services, dit couplage lâche. Le paradigme SOA et l'utilisation de modèles de processus sont donc particulièrement appropriés pour gérer l'évolution rapide des services métiers et des processus métiers qui les orchestrent. Ils supportent en particulier la réutilisation des services pour créer de nouveaux services par composition. L'utilisation des langages de spécifications de workflow favorise une meilleure compréhension des activités et de leur enchaînements. Les orchestrations BPEL jouent un rôle important sur ce point par l'expression de l'ensemble des partenaires (Web services) via leur "identité" indépendamment de leurs implémentations. Les implémentations actuelles ne supportent cependant que la liaison statique aux différents partenaires, ce qui est quand même contradictoire avec les principes des Web services.
Idéalement les assemblages de services devraient pouvoir s'adapter aux modifications de l'environnement et aux besoins des différents utilisateurs. Il est en effet nécessaire de pouvoir dynamiquement choisir la bonne version d'un Web service selon le contexte d'exécution du processus. En particulier, l'introduction des utilisateurs dans le workflow implique de permettre une réorganisation dynamique des assemblages en fonction des interventions des utilisateurs, voire de procéder par auto-organisation [PB05].
Face à la complexité des réseaux de services, la durée de vie des interactions, la difficulté d'un contrôle de l'intégrité des transactions sur les applications propriétaires sous-jacentes aux services, la compensation est un mécanisme d'adaptation qu'il parait essentiel de prendre en compte [ZDGH05].
Les solutions basées sur les moteurs de workflow présentent une centralisation du contrôle qui répond alors mal au passage à l'échelle, à la gestion des pannes, à la flexibilité, etc. Aussi, dans [CCMN04], les auteurs discutent-ils l'intérêt de décentraliser certaines parties du réseaux de Web services pour éviter les goulots d'étranglements, réduire les échanges de don-

nées, distribuer les contrôles. A l'instar des travaux sur les grilles de calcul [GEM06], il s'agit de déterminer le découpage optimal de l'application, sachant que en décentralisant la propagation et la gestion des erreurs est encore plus difficile à maîtriser.

Validation

Alors que BPEL prend ses racines dans les travaux relatifs aux algèbres de processus, qui offrent des outils pour valider les assemblages, les Web services présentent avec WSDL une sémantique trop faible pour permettre une réelle validation des compositions de services. Le respect des protocoles relatifs aux services est alors un défi dans un contexte général. Les annotations associées aux définitions WSDL ne permettent pas telles que une validation automatique. Les contrats vont donc jouer un rôle primordial pour assurer la validité des assemblages (cf. 3) tandis que les travaux sur la qualité de services apportent des éléments de réponses qu'il s'agit d'exploiter.
L'article de Nahrstedt et Balke [NB04] met particulièrement bien en avant ce point en décrivant les problèmes relatifs à la composition des services dans le cadre de systèmes multimédia. Dans ce contexte, ils défendent en particulier la faiblesse sémantique des Web services actuels en matière de qualité de service et les besoins en technologies pour supporter les négociations.

4.5 Conclusion : complémentarité des formes de composition

Le développement des applications repose encore aujourd'hui sur la construction de systèmes monolithiques qui doivent être maintenus et adaptés pour chaque évolution quand bien même il s'agit de développer des applications similaires.
Le développement par composition de composants et de services représente une solution potentielle en autorisant une décomposition des fonctionnalités en entités plus petites et si possibles indépendantes.
En terme de prise en compte des contrats au niveau des compositions dans les plates-formes à composants, nous avons constaté (i) des besoins pour contrôler et valider les assemblages : les ADL y répondent partiellement de manière déclarative pour les assemblages prévus tandis que les interfaces de contrôles de Fractal ou container Wcomp tentent d'y répondre de manière "programmative" pour les évolutions dynamiques des assemblages, (ii) une grande hétérogénéité des infrastructures qui supportent les composants avec en particulier un rôle important accordé aux liaisons pour la prise en compte de l'interopérabilité entraînant une frontière floue entre le support à la distribution et l'infrastructure des plates-formes à composants.
La combinaison des codes est une forme de programmation particulièrement bien adaptée à l'introduction des contrôles dans les applications. La combinaison des modèles répond au problème des compositions de points de vues. Nous avons également constaté que cette forme de développement nécessite des supports formels pour valider les compositions et assurer la cohérence des résultats. La prise en charge de la validation des contrats soit au niveau des modélisation soit à l'exécution semble trouver là un bon support d'intégration pour autant que nous soyons à même d'assurer qu'ils n'introduisent pas de failles.
Le développement des architectures orientées services est particulièrement bien adapté à l'adaptation des applications ne serait-ce que parce que les contraintes des infrastructures de support sont beaucoup moins fortes qu'avec les approches à composants. Les nombreux rapprochements entre algèbres de processus et le langage d'orchestration BPEL4WS témoignent d'un besoin de validation des assemblages. Cependant, plus la richesse des workflows et des données échangées est importante, plus cette démarche apparaît difficile. La faible sémantique associée aux Web services, la difficulté de capturer les contextes d'exécution et la

nécessité de calculer la qualité de service au niveau des assemblages constituent assurément des freins à un usage sensé des architectures à base de Web services dans un contexte professionnel.
De l'étude menée dans ce chapitre nous pouvons retenir que la composition est un élément essentiel à la réalisation des applications futures, mais qu'elle requiert un support important afin de répondre aux besoins de validation des assemblages (passage à l'échelle, propriétés de sûreté, vivacité, etc.), de programmation proche des spécificités des programmeurs (Langages d'architectures, orchestrations, algèbre de processus, langages d'aspects, etc.) et de flexibilité pour assurer une programmation indépendante des supports et adaptable en fonction des particularités du contexte d'exécution.

Bibliographie

[ABV04] E. Pimentel A. Brogi, C. Canal and A. Vallecillo. Formalizing web service choreographies. In *Proc WS-FM 2004*, 2004.

[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 Press.

[AD05] Samir Ammour and Philippe Desfray. A concern-based technique for architecture modelling using the UML Package Merge. pages 1–13, Nuremberg, Germany, november 2005.

[AEB03] O. Aldawud, T. Elrad, and A. Bader. Uml profile for aspect-oriented software development, 2003.

[Aks03] Mehmet Aksit, editor. *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, Boston, Massachusetts, USA, March 2003. ACM Press.

[All97] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, May 1997. CMU Technical Report CMU-CS-97-144.

[AN01] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts — A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[BCL+04a] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. An open component model and its support in Java. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE-7)*, volume 3054 of *LNCS*, pages 7–22. Springer, May 2004.

[BCL+04b] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in java. In *Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2003)*, Edinburgh, Scotland, May 2004.

[BCS02] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing, 2002.

[BD04] Olivier Barais and Laurence Duchien. Safarchie studio : An argouml extension to build safe architectures. In *Workshop on Architecture Description Languages (WADL 2004)*, Toulouse, France, August 2004.

[BDH01] Y. Bardin, S. Damy, and B. Herrmann. Un service de médiation pour les applications réparties à base de composants. In *Journées Composants : flexibilité du système au langage*, pages 67–76, Besançon, France, October 2001.

[Ber01] L. Berger. *Mise en oeuvre des interactions en environnements distribués, compilés et fortement typés : le modèle MICADO*. PhD thesis, Université de Nice-Sophia Antipolis, octobre 2001.

[Ber03] Philip Bernstein. Applying model management to classical meta data problems. In *Conf. on Innovative Database Research (CIDR)*, Asilomar, CA, USA, jan 2003.

[BFCE+04] M. Blay-Fornarino, A. Charfi, D. Emsellem, A-M. Pinna-Dery, and M. Riveill. Software interaction. *Journal of Object Technology*, 3(10) :161–180, z 2004.

[BLMD06] Olivier Barais, Julia Lawall, Anne-Françoise Le Meur, and Laurence Duchien. Safe integration of new concerns in a software architecture. In *13th Annual IEEE International Conference on Engineering of Computer Based Systems ECBS'06*, Potsdam, Germany, mar 2006.

[BMP05] Olivier Barais, Alexis Muller, and Nicolas Pessemier. Extension de fractal pour le support des vues au sein d'une architecture logicielle. In *Numéro spécial de la revue L'OBJET-RSTI*, volume 11. Hermès Sciences, 2005.

[BNB04] Jullien Bouchet, Laurence Nigay, and Didier Balzagette. Icare : a component-based approach for multimodal interaction. In *UbiMob '04 : Proceedings of the 1st French-speaking conference on Mobility and ubiquity computing*, pages 36–43, New York, NY, USA, 2004. ACM Press.

[Bon05] Pierre Bonnet. Cadre de référence, architecture SOA, meilleures pratiques. http ://www.orchestranetworks.com/fr/soa/, February 2005. Orchestra Networks.

[BR06] Mikael Beauvois and Michel Riveill. An Extension of Fractal for Behavioural Composition. In *Fractal CBSE Workshop*, Nantes, July 2006.

[BS03] M. Basch and A. Sanchez. Incorporating aspects into the uml, 2003.

[BSD03] Boualem Benatallah, Quan Z. Sheng, and Marlon Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1) :40–48, 2003.

[BSL01] Noury M. Bouraqadi-Saâdani and Thomas Ledoux. Le point sur la programmation par aspects. *Technique et Science Informatique*, 20(4), 2001.

[BW03] Wolf-Tilo Balke and Matthias Wagner. Cooperative discovery for user-centered web service provisioning. In Zhang [Zha03], pages 191–197.

[Car01] P.S. Caro. *Adding Systemic Crosscuting and Super-Imposition to Composition Filters*. Ph.D Thesis, University of Twente, Netherlands, 2001.

[Car03] Eric Cariou. *Contribution à un Processus de Réification d'Abstractions de Communication*. PhD thesis, Université de Rennes, June 2003.

[CCCV05] Javier Camara, Carlos Canal, Javier Cubo, and Antonio Vallecillo. Formalizing WSBPEL Business Processes Using Process Algebra. In *Foundations of Coordination Languages and Software Architectures (FOCLASA)*, San Francisco (CA), August 2005. Springer.

[CCMN04] Girish B. Chafle, Sunil Chandra, Vijay Mann, and Mangala Gowri Nanda. Decentralized orchestration of composite web services. In *WWW Alt. '04 : Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 134–143, New York, NY, USA, 2004. ACM Press.

[CCMV03] Olivier Caron, Bernard Carré, Alexis Muller, and Gilles Vanwormhoudt. A framework for supporting views in component oriented information systems. In *in OOIS 2003*, volume 2817 of *Lecture Notes in Computer Science*, pages 164–178, Geneva, Switzerland, September 2003. Springer Verlag.

[CCMV04] Olivier Caron, Bernard Carré, Alexis Muller, and Gilles Vanwormhoudt. An ocl formulation of uml 2 template binding. In *in Proceedings of UML'2004 :7th International Conference on UML Modeling Languages and Applications*, volume 3273 of *Lecture Notes in Computer Science*, pages 27–40, Lisbon, Portugal, October 2004.

[CF05a] Carine Courbis and Anthony Finkelstein. Towards aspect weaving applications. In *ICSE '05 : Proceedings of the 27th international conference on Software engineering*, pages 69–77, New York, NY, USA, 2005. ACM Press.

[CF05b] Carine Courbis and Anthony Finkelstein. Weaving aspects into web service orchestrations. In *ICWS*, pages 219–226. IEEE Computer Society, 2005.

[CFWBFT+05] Daniel Cheung Foo Woo, Mireille Blay-Fornarino, Jean-Yves Tigli, Anne-Marie Pinna-Déry, David Emsellem, and Michel Riveill. Langage d'aspects pour la composition dynamique de composants embarqués. Lille, France, sep 2005.

[CFWBFT+06] Daniel Cheung Foo Wo, Mireille Blay-Fornarino, Jean-Yves Tigli, Stéphane Lavirotte, and Michel Riveill. Adaptation dynamique d'assemblages de dispositifs par des modèles. In *2èmes Journées sur l'Ingénierie Dirigée par les Modèles (IDM)*, June 2006.

[CFWTLR06] Daniel Cheung Foo Wo, Jean-Yves Tigli, Stéphane Lavirotte, and Michel Riveill. Wcomp : a Multi-Design Approach for Prototyping Applications using Heterogeneous Resources. In *17th IEEE International Workshop on Rapid System Prototyping (RSP)*, June 2006.

[CIJ+00] Fabio Casati, Ski Ilnicki, LiJie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in eFlow. In *Conference on Advanced Information Systems Engineering*, pages 13–31, 2000.

[CL06] Pierre Crescenzo and Philippe Lahire. De la réutilisabilité des applications vers celle des modèles. *Numéro spécial de la revue L'Objet*, page 23, juin/spetembre 2006.

[Cla02] Siobhàn Clarke. Extending standard uml with model composition semantics. *Sci. Comput. Program.*, 44(1) :71–100, 2002.

[CM04] Anis Charfi and Mira Mezini. Aspect-oriented web service composition with ao4bpel. In *ECOWS*, volume 3250 of *LNCS*, pages 168–182. Springer, 2004.

[CM05a] Anis Charfi and Mira Mezini. An aspect-based process container for bpel. In *AOMD '05 : Proceedings of the 1st workshop on Aspect oriented middleware development*, pages 1–6, New York, NY, USA, 2005. ACM Press.

[CM05b] Anis Charfi and Mira Mezini. Middleware services for web service compositions. In *WWW '05 : Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 1132–1133. ACM Press, May 2005.

[CV05] María Agustina Cibrán and Bart Verheecke. Dynamic Business Rules for Web Service Composition. In *Proceedings of the Second Dynamic Aspects Workshop (DAW05)*. Research Institute for Advanced Computer, March 2005.

[DBLM03] G. De Giacomo D. Berardi, D. Calvanese, M. Lenzerini, and M. Mecella. A foundational vision of e-services. In *Proc of CASA 2003 Workshop on Web Services, e-Business and the semantic WEB*, 2003.

[DBM04] G. De Giacomo D. Berardi, D. Calvanese and M. Mecella. Reasoning about actions for e-service composition. In *Proc ICALP 2004*, 2004.

[DFL+05] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc S&#233 ;gura-Devillechaise, and Mario S&#252 ;dholt. An expressive aspect language for system applications with arachne. In *AOSD'05 : Proceedings of the 4th international conference on Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2005. ACM Press.

[DFS04] Rémi Douence, Pascal Fradet, and Mario S&#252 ;dholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04 : Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150, New York, NY, USA, 2004. ACM Press.

[DmYK01] Linda G. DeMichiel, L. Ümit Yalçinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification, Version 2.0*. Sun Microsystems Inc., version 2.0 edition, August 2001.

[DUVH06] Nicolas Desnos, Christelle Urtado, Sylvain Vauttier, and Marianne Huchard. Assistance à l'architecte pour la construction d'architectures à base de composants. In Rousseau et al. [RUV06], pages 37–52.

[DW99] Desmond D'Souza and Alan Wills. *Objects, Components and Frameworks With UML : The Catalysis Approach.* Addison-Wesley, 1999.

[EBR00] A. Cavarra E. Borger and E. Riccobene. An asm semantics for uml activity diagrams. In *Proc Algebraic Methodology and Software Technology (AMAST 2000), LNCS 1816*, 2000.

[Esh99] R Eshsuis. Semantics and verification of uml activity diagrams for workflow modeling, phd, twente university, 1999.

[EW04] R. Eshuis and R. Wieringa. Tool support for verifying uml activity diagrams. In *IEEE Transactions on Software Engineering, vol 30*, 2004.

[FF05] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. pages 21–35. Addison-Wesley, Boston, 2005.

[FRGG04] Robert B. France, Indrakshi Ray, Geri Georg, and Sudipto Ghosh. Aspect-oriented approach to early design modelling. *IEE Proceedings - Software*, 151(4) :173–186, august 2004.

[FUMK04] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Compatibility verification for web service choreography. In *ICWS '04 : Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, page 738, Washington, DC, USA, 2004. IEEE Computer Society.

[GEM06] Tristan Glatard, David Emsellem, and Johan Montagnat. Generic web service wrapper for efficient embedding of legacy codes in service-based workflows. In *Grid-Enabling Legacy Applications and Supporting End Users Workshop (GELA'06)*, Paris, France, June 2006.

[GHC99] Jr. Grady H. Campbell. Adaptable components. In *ICSE '99 : Proceedings of the 21st international conference on Software engineering*, pages 685–686, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[GMP+06] Tristan Glatard, Johan Montagnat, Xavier Pennec, David Emsellem, and Diane Lingrand. MOTEUR : a data-intensive service-based workflow manager. Technical Report I3S/RR-2006-07-FR, I3S, Sophia Antipolis (France), March 2006.

[GR91] M. M. Gorlick and R. R. Razouk. Using weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering (ICSE13)*, pages 23–34, Austin, TX, USA, May 1991.

[GS04] A Chirichiello G Salaun, A Ferrara. Negotiation among web services using lotos/cadp. In *Proc ECOWS, LNCS 3250*, 2004.

[HL04] Colombe Herault and Sylvain Lecomte. Gestion dynamique des services techniques pour modele a composants. *CoRR*, cs.NI/0411089, 2004.

[JLGC04] J Merseguer JP Lopez-Grao and J Campos. From uml activity diagrams to stochastic petri nets : Application to software performance engineering. In *Proc WOSP 2004*, 2004.

[KF06] Jacques Klein and Franck Fleurey. Tissage d'aspects comportementaux. In *Langages et Modèles à Objets : LMO'06*, Nimes, France, March 2006.

[KG02] Jörg Kienzle and Rachid Guerraoui. Aop : Does it make sense ? the case of concurrency and failures. In *ECOOP '02 : Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 37–61, London, UK, 2002. Springer-Verlag.

[KHH+01a] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings of ECOOP'01*, LNCS(2072), Budapest, Hungaria, June 2001. Springer Verlag.

[KHH+01b] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *LNCS*, pages 327–353. Springer, June 2001.

[KHJ06] Jacques Klein, Loic Hélouet, and Jean-Marc Jézéquel. Semantic-based weaving of scenarios. In *proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, Bonn, Germany, March 2006. ACM.

[KM05] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2005.

[KMW03] Rania Khalaf, Nirmal Mukhi, and Sanjiva Weerawarana. Service-oriented composition in bpel4ws. In *WWW (Alternate Paper Tracks)*, 2003.

[Kra03] Sacha Krakowiak. Patrons et canevas pour l'intergiciel, August 2003.

[LAB+05] Bertram Ludäscher, Ikay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation : Practice & Experience*, 2005.

[LCL06] Marc Léger, Thierry Coupaye, and Thomas Ledoux. Contrôle dynamique de l'intégrité des communications dans les architectures à composants. In Rousseau et al. [RUV06], pages 21–36.

[LQ06] Philippe Lahire and Laurent Quintian. New perspective to improve reusability in object-oriented languages. *Journal Of Object Technology (JOT)*, 5(1) :117–138, 2006.

[LV95] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, pages 717–734, September 1995.

[Mar05] Axel Martens. Analyzing Web Service based Business Processes. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 3442, Edinburgh (Scotland), April 2005. Springer-Verlag.

[MB05] Selma Matougui and Antoine Beugnard. How to implement software connectors ? a reusable, abstract and adaptable connector. In Lea Kutvonen and Nancy Alonistioti, editors, *DAIS*, volume 3543 of *Lecture Notes in Computer Science*, pages 83–94. Springer, 2005.

[MCCV05] Alexis Muller, Olivier Caron, Bernard Carré, and Gilles Vanwormhoudt. On some properties of parameterized model applications. In *Proceedings of ECM-DA'05 : First European Conference on Model Driven Architecture - Foundations and Applications*, page 16 pages, Nuremberg, Germany, november 2005.

[MDBF06] Raphaël Marvie, Laurence Duchien, and Mireille Blay-Fornarino. *Au delà du MDA : l'Ingénierie Dirigée par les Modèles*, chapter Les plates-formes d'exécution et l'IDM. Hermès, 2006.

[Med96] N. Medvidovic. Adls and dynamic architecture changes. In ed. A. L. Wolf, editor, *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 24–27, San Francisco, USA, October 1996.

[MH03] S. McDirmid and W.C. Hsieh. Aspect-Oriented Programming with Jiazzi. In Aksit [Aks03].

[MK96] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of ACM SIGSOFT'96 : Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 3–14, San Francisco, CA, USA, October 1996.

[MKR06] Tom Mens, Günter Kniesel, and Olga Runge. Transformation dependency analysis. A comparison of two approaches. In *Langages et Modèles à Objets (LMO)*, pages 167–182, Nimes, March 2006. Hermes.

[MLM+06] Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter Brown, and Rebekah Metz. Reference Model for Service Oriented Architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS, February 2006.

[MMGL01] Raphael Marvie, Philippe Merle, Jean-Marc Geib, and Sylvain Leblanc. Torba : Trading contracts for corba. In *Proceedings of the 6th USENIX Conference on Object- Oriented Technologies and Systems (COOTS 2001)*, pages 1–14. ACM/IFIP/USENIX, jan 2001.

[MO03] M. Mezini and K. Ostermann. Conquering Aspects with Casear. In Aksit [Aks03], pages 90–99.

[MORT96] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of ACM SIGSOFT'96 : Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 24–32, San Francisco, CA, USA, October 1996.

[MT97] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1) :70–93, 1997.

[MV05] Farida Mostefaoui and Julie Vachon. Modélisation et vérification formelle de la composition des aspects. In Lionel Seinturier, editor, *actes de la 2ième Journée Francophone sur le Développement de Logiciels Par Aspects*. Hermès, Septembre 2005.

[Nan04] O. Nano. *Un modèle de réécriture pour l'intégration de services*. PhD thesis, Université de Nice - Sophia Antipolis, 2004.

[NB04] Klara Nahrstedt and Wolf-Tilo Balke. A taxonomy for multimedia service composition. In *MULTIMEDIA '04 : Proceedings of the 12th annual ACM international conference on Multimedia*, pages 88–95, New York, NY, USA, 2004. ACM Press.

[NBF04] Olivier Nano and Mireille Blay-Fornarino. Annotations et transformations de modèles pour l'intégration de services. *Conférence Langages et Modèles à Objets - LM0 2004, Lille France - publié dans la revue RTSI - série l'Objet (Lavoisier Eds) - ISBN : 2-7462-0887-3*, 10(2-3) :175–188, 15 au 17 mars 2004.

[Nem06] Clémentine Nemo. Vers la composition d'orchestrations de services. Master dissertation, DEA PLMT, Nice (France), June 2006.

[OAF+04] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna : A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics journal*, 17(20) :3045–3054, 2004.

[obj05] objectweb. The OMG's CORBA Components Technology. http ://openccm.objectweb.org/doc/ccm.html, 2005.

[OH05] D. Bardou O. Hachani. Modélisation par aspects et transformation vers aspectj et hyper/j. In *Actes de LMO 2005, Langages et Modèles à Objets dans la revue l'objet*, volume 11, pages 127–142, Berne, Suisse, mars 2005. Hermes-Lavoisier.

[OMG04] OMG. Uml 2.0 superstructure. http ://www.omg.org/cgi-bin/doc ?ptc/2004-10-02, october 2004.

[OPD04a] Audrey Occello and Anne-Marie Pinna-Déry. Safe runtime adaptations of components : a UML metamodel with OCL constraints. In *First International Workshop on Foundations of Unanticipated Software Evolution (FUSE)*, Barcelona (Spain), March 2004.

[OPD04b] Audrey Occello and Anne-Marie Pinna-Déry. An Adaptation-safe Model for Component Platforms. In *13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE)*, Nice (France), jul 2004.

[OT00] H. Ossher and P. Tarr. Hyper/J : Multi-Dimentionnal Separation of Concern for Java. In Carlo Ghezzy, editor, *Proceedings of ICSE'00*, Limerick, Ireland, June 2000. ACM Press.

[Paw05] Renaud Pawlak. Spoon : annotation-driven program transformation — the aop case. In *AOMD '05 : Proceedings of the 1st workshop on Aspect oriented middleware development*, pages 1–6, New York, NY, USA, 2005. ACM Press.

[PB05] C. Prehofer and C. Bettstetter. Self-organization in communication networks : principles and design paradigms. *Communications Magazine, IEEE*, 43(7) :78–85, 2005.

[Per] Projet rnrt/persiform.

[Pre03] Christian Prehofer. Plug-and-play composition of features and feature interactions with statechart diagrams. In Daniel Amyot and Luigi Logrippo, editors, *FIW*, pages 43–58. IOS Press, 2003.

[PSDB04] Nicolas Pessemier, Lionel Seinturier, Laurence Duchien, and Olivier Barais. Une extension de fractal pour l'aop. In *Première journée Francophone sur le Développement de Logiciels par Aspects (JFDLPA 2004)*, Paris, France, September 2004.

[PSDC06] N. Pessemier, L. Seinturier, L. Duchien, and T. Coupaye. A model for developing component-based and aspect-oriented systems. In *Proceedings of the 5th International Symposium on Software Composition (SC'06)*, Lecture Notes in Computer Science. Springer, March 2006.

[PSDF01] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Dynamic Wrappers : Handling he Composition Issue with JAC. In *proceedings of TOOLS'01*, pages 56–65, 2001.

[PSSN03] Massimo Paolucci, Naveen Srinivasan, Katia P. Sycara, and Takuya Nishimura. Towards a semantic choreography of web services : From wsdl to daml-s. In Zhang [Zha03], pages 22–26.

[Qui04] L. Quintian. *JAdaptor : Un modèle pour améliorer la réutilisation des préoccupations dans le paradigme objet*. Thèse de doctorat, Université de Nice-Sophia Antipolis, France, juillet 2004.

[Rap02] Pascal Rapicault. *Modèles et techniques pour spécifier, développer et utiliser un framework : une approche par méta-modélisation*. PhD thesis, Université de Nice - Sophia Antipolis, May 2002.

[RBY+04] R. Razavi, N. Bouraqadi, J. W. Yoder, J. F. Perrot, and R. Johnson. Language support for adaptive object-models using metaclasses. In *Research Track of the ESUG 2004 Smalltalk Conference*, Köthen (Anhalt), Germany, September 2004. Selected for publication in the Elsevier international journal "Computer Languages, Systems and Structures".

[RPPM06] Romain Rouvoy, Nicolas Pessemier, Renaud Pawlak, and Philippe Merle. Using attribute-oriented programming to leverage fractal-based developments. In *Proceedings of the 5th International ECOOP Workshop on Fractal Component Model (Fractal'06)*, Nantes, France, jul 2006. To appear.

[RRB02] P. Rapicault, J-P. Rigault, and L. Bourlier. Model, notation, and tools for verification of protocol-based components assembly. In Judith Bishop, editor, *Component Deployment CD*2002, number 2370 in LNCS, pages 257–268. IFIP / ACM Working Conference, Springer-Verlag, June 2002.

[RUV06] Roger Rousseau, Christelle Urtado, and Sylvain Vauttier, editors. *Langages et Modèles à objets*, Nîmes (France), March 2006. Hermès - Lavoisier.

[SB00] Yannis Smaragdakis and Don Batory. Application generators. *Encyclopedia of Electrical and Electronics Engineering*, 2000. J.G. Webster (ed.), John Wiley and Sons.

[SGS+04] G. Straw, G. Georg, E. Song, S. Ghosh, R. France, and J. M. Bieman. Model composition directives. In *in Proceedings of UML'2004 :7th International Conference on UML Modeling Languages and Applications*, volume 3273 of *Lecture Notes in Computer Science*, pages 84–97, Lisbon, Portugal, October 2004.

[SHU02] Dominik Stein, Stefan Hanenberg, and Rainer Unland. A uml-based aspect-oriented design notation for aspectj. In *AOSD '02 : Proceedings of the 1st international conference on Aspect-oriented software development*, pages 106–112, New York, NY, USA, 2002. ACM Press.

[SPDC06] L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A component model engineered with components and aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, Lecture Notes in Computer Science. Springer, June 2006.

[Szy96] Clemens Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia, 1996.

[TBB04] Francis Tessier, Mourad Badri, and Linda Badri. A model-based detection of conflicts between crosscutting concerns : Towards a formal approach. In Minhuan Huang, Hong Mei, and Jianjun Zhao, editors, *International Workshop on Aspect-Oriented Software Development (WAOSD 2004)*, September 2004.

[TFS04] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. Préservation de choix architecturaux lors de l'évolution d'un composant. In *Proceedings of OCM-SI'04 workshop (Objets, Composants et Modèles pour les Systèmes d'Information), held in conjunction with INFORSID'04*, Biaritz, France, May 2004.

[TWSM05] Ian Taylor, Ian Wand, Matthew Shields, and Shalil Majithia. Distributed computing with Triana on the Grid. *Concurrency and Computation : Practice & Experience*, 17(1–18), 2005.

[Vir04] Mirko Viroli. Towards a formal foundation to orchestration languages. *Electr. Notes Theor. Comput. Sci.*, 105 :51–71, 2004.

[VSV+05] Wim Vanderperren, Davy Suv&#233 ;e, Bart Verheecke, Mar&#237 ;a Agustina Cibr&#225 ;n, and Viviane Jonckers. Adaptive programming in jasco. In *AOSD '05 : Proceedings of the 4th international conference on Aspect-oriented software development*, pages 75–86, New York, NY, USA, 2005. ACM Press.

[VVJ06] Bart Verheecke, Wim Vanderperren, and Viviane Jonckers. Unraveling crosscutting concerns in web services middleware. *IEEE Software*, 23(1) :42–50, 2006.

[ZDGH05] Olaf Zimmermann, Vadim Doubrovski, Jonas Grundler, and Kerard Hogg. Service-oriented architecture and business process choreography in an order management scenario : rationale, concepts, lessons learned. In *OOPSLA '05 : Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 301–312, New York, NY, USA, 2005. ACM Press.

[Zha03] Liang-Jie Zhang, editor. *Proceedings of the International Conference on Web Services, ICWS '03, June 23 - 26, 2003, Las Vegas, Nevada, USA*. CSREA Press, 2003.