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

Table des matières

CHAPITRE 1

Introduction

Ce rapport est le résultat de l'étude de l'état de l'art dans les domaines relatifs au projet RNTL FAROS (composition de contrats pour la Fiabilité d'ARchitectures Orientées Services) : les services, les contrats, la composition et l'ingénierie des modèles. Il constitue le premier livrable (F-1.1) du premier lot de ce projet. Il sera notamment utilisé pour déterminer les spécifications d'une architecture pour la contractualisation des services (livrable F-1.2).

1.1 Contexte

Pour faire face à l'évolution de leur métier, les entreprises doivent pouvoir intégrer anciens et nouveaux systèmes d'information par composition. De plus, l'arrivée des assistants numériques et des téléphones portables amène les entreprises à proposer un accès aux services via ces nouveaux terminaux. Ces terminaux sont des environnements contraints qui nécessitent une plus forte prise en compte des propriétés extra-fonctionnelles au niveau applicatif, ce qui constitue une difficulté supplémentaire. Cette intégration par composition de services est actuellement pour elles un enjeu crucial. Quelle que soit leur taille, leur système d'information comprend la plupart du temps de services autonomes et hétérogènes. Le verrou technologique, que ce projet se propose de lever, consiste à mettre en place des garanties contractuelles sur les assemblages et les compositions de ces services.
Actuellement, sous le vocable SOA pour Services Oriented Architecture, se développe un style d'architecture dont l'objectif est de proposer un faible couplage entre composants et services logiciels par des moyens de composition au moment de la conception et des moyens d'orchestration au moment de l'exécution tout en s'appuyant sur des règles métiers. Cette approche récente a besoin de s'appuyer sur des concepts et des démarches de conception d'applications et également d'être outillée.
Le projet FAROS a pour objectif de définir un environnement de composition pour la construction fiable d'architectures orientées services pour des applications destinées à des environnements nomades. Le projet FAROS complète les travaux sur l'intégration d'applications par la prise en compte d'éléments contractuels permettant une composition cohérente des services. Ces éléments contractuels devraient être de différents niveaux : fonctionnels, extra-fonctionnels, liés à la qualité de service, locaux ou globaux. Ils permettront l'élévation du niveau de confiance dans l'assemblage des composants et seront exprimés à l'aide de langages formels. Ils viendront compléter les modèles métiers et seront utilisés lors de la composition, lors de la projection vers les plates-formes d'exécution, mais également lors de l'orchestration des services sur ces plates-formes.

1.2 Démarche

A travers cet état de l'art, les objectifs vis-à-vis du projet FAROS sont doubles. Il s'agit d'une part d'établir un vocabulaire commun, en envisageant les différentes définitions et approches qui ont été proposées dans les travaux connexes pour les éléments constitutifs du projet : services, architectures orientées services, contrats, compositions. D'autre part, l'étude de l'existant doit permettre de mieux cerner les verrous à lever, dans les résultats attendus et dans la manière de réaliser ce projet.

Ainsi, la présente étude se doit de faire une synthèse précise des différentes approches pour composer des services et des approches pour établir des contrats de confiance dans le développement logiciel. Les approches de composition seront particulièrement étudiées par leur capacité à garantir des propriétés de fiabilité (description syntaxique, protocole d'utilisation d'interface, sémantique comportementale, propriétés de synchronisation, définition de qualité des services). Les approches contractuelles étudiées couvriront la garantie de propriétés fonctionnelles et extra fonctionnelles dans les systèmes orientés objets, composants et services. Par ailleurs, la mise en œuvre de l'architecture pressentie dans le projet repose fortement sur l'ingénierie des modèles et les techniques de transformation associées. Cette étude doit donc s'intéresser à ces techniques et aux moyens de valider les différentes transformations prévues.

1.3 Plan

La suite de ce rapport s'organise de la manière suivante. Le chapitre 2 étudie les principales approches qui définissent des services et les architectures associées. Le chapitre 3 effectue une synthèse des différentes approches contractuelles dans les modèles à objets et à composants. Ce chapitre étudie aussi les différents formes de contrat que l'on peut trouver actuellement dans les architectures orientées services. Dans le chapitre 4, les différents moyens de composition, de services ou d'autres entités informatiques similaires, sont étudiés. Le chapitre 5 présente l'approche générale de l'ingénierie des modèles. Les objectifs et les apports de l'ingénierie des modèles sont expliqués, puis les grandes familles d'usages sont présentés. L'accent est mis sur les difficultés de la conception de métamodèles et de transformations de modèles, en particulier sur les questions de puissance d'expression des langages de transformation et de testabilité. Les différentes catégories d'outils sont ensuite établies. Le chapitre s'achève sur la description du rôle que jouera l'ingénierie des modèles au sein du projet FAROS. Le chapitre 6 conclut ce rapport en synthétisant les études effectuées et en mettant en avant des éléments directeurs pour la suite du projet. Une bibliographie spécifique est fournie à la fin de chacun des chapitres 2 à 5. Enfin, une liste des principaux acronymes relatifs au domaine est donnée en annexe.

CHAPITRE 2

Services

Coordonnateur : Anne-Françoise Le Meur.
Rédacteurs : Fabien Baligand, Mireille Blay-Fornarino, Philippe Collet, Jean Perrin, Nicolas Rivierre.

Ce chapitre a pour but d'introduire les concepts de base associés aux services (partie 2.1) et aux architectures orientées services (partie 2.2). Puis la partie 2.3 traite des Web Services, qui constituent une mise en œuvre possible, et souvent utilisée, d'une architecture orientée services. Enfin, la partie 2.4 aborde les thèmes qui seront développés dans les chapitres suivants.

2.1 Des objets aux services

Une des problématiques fondamentales de l'ingénierie informatique réside dans la capacité à rationaliser le développement des applications. En particulier, la question de la réutilisabilité des briques logicielles déjà développées constitue un enjeu majeur. Il y a déjà une dizaine d'années, l'approche objet donnait un premier élément de réponse à cette problématique. Les objets ont apporté un ensemble de principes de conception parmi lesquels l'abstraction (un objet possède un type de données et un comportement bien spécifié), l'encapsulation (l'interface et l'implémentation sont séparées), le polymorphisme (capacité de prendre des formes différentes à l'exécution), l'héritage (possibilité d'enrichir un objet avec des extensions), l'identité (capacité de distinguer un objet parmi les autres). Techniquement, les objets sont dépendants des langages et d'un environnement d'exécution.
Les composants sont apparus pour permettre des regroupements cohérents et réutilisables d'objets [Szy98] : "Un composant est une unité de composition avec des interfaces spécifiées à l'aide de contrats et de dépendances au contexte seulement. Un composant logiciel peut être déployé indépendamment et est sujet à être composé par des tiers". Objets et composants partagent un certain nombre de motivations communes : ils possèdent tous deux des propriétés encapsulées, ils sont accessibles par des interfaces bien spécifiées et ils facilitent la réutilisation du logiciel. Les objets représentent des entités du monde réel, tandis que les composants décrivent des fonctionnalités du monde réel. Techniquement, les composants se déploient et s'exécutent dans des contextes qui leur sont spécifiques. Par ailleurs, l'approche composant permet, par le biais de ces contextes, de mettre en œuvre certains paramètres d'exécution tels que la sécurité, la persistance, les mécanismes transactionnels, etc.
Le concept de service est actuellement le sujet de définitions très variées. L'objectif du livrable suivant de spécifications des besoins est notamment de poser une définition commune aux partenaires et pertinentes vis-à-vis de la problématique définie. Nous donnons ici plusieurs définitions, qui ne sont pas *a priori* contradictoires, mais qui se placent selon différents points de vue[1] :

– *A service represents some functionality (application function, business transaction, system service, etc.) exposed as a component for a business process [Dod04].*
– *Service : The means by which the needs of a consumer are brought together with the capabilities of a provider [MLM+06].*

[1] les définitions sont ici données en anglais pour éviter toute *interprétation* lors de la traduction.

– *A service in SOA is an exposed piece of functionality with three properties : (1) the interface contract to the service is platform-independent, (2) the service can be dynamically located and invoked, (3) the service is self-contained, that is, the service maintains its own state [Has03].*

Indépendamment de ces définitions, plusieurs caractéristiques se distinguent. Les interfaces et les données exhibées par le service sont exprimées en termes métiers. Les aspects technologiques ne sont plus essentiels car les services sont autonomes, c'est-à-dire qu'ils sont indépendants du contexte d'utilisation ainsi que de l'état des autres services, et interopèrent via des protocoles standardisés. Un service définit une entité logicielle (*e.g.*, ressource, application, module, composant logiciel, etc.) qui communique via un échange de messages et qui expose un contrat d'utilisation. De façon similaire aux approches par objet ou par composant, l'approche service cherche à fournir un niveau d'abstraction encore supérieur, en encapsulant des fonctionnalités et en permettant la réutilisation de service déjà existant. La propriété de couplage faible que cette approche induit est à l'origine des architectures agiles que sont les architectures orientées services.

2.2 Architectures orientées services

2.2.1 Qu'est qu'une architecture orientée services ?

Une architecture orientée services (SOA pour *Service Oriented Architectures*) est un style d'architecture fondée sur la description de services et de leurs interactions. Les caractéristiques principales d'une architecture orientée services sont le couplage faible, l'indépendance par rapport aux aspects technologiques et l'extensibilité. La propriété de couplage faible implique qu'un service n'appelle pas directement un autre service, les interactions sont gérées par une fonction d'orchestration. Il est donc plus facile de réutiliser un service puisqu'il n'est pas directement lié à d'autres services. L'indépendance par rapport aux aspects technologiques est obtenue grâce aux contrats d'utilisation associés et qui sont indépendants de la plate-forme technique utilisée par le fournisseur du service. Enfin, l'extensibilité est rendue possible par le fait que de nouveaux services peuvent être découverts et invoqués à l'exécution.
La notion d'architecture orientée service n'est pas nouvelle, elle est apparue dès le développement des approches client/serveur. Ici encore les définitions sont nombreuses et nous retiendrons les suivantes :

– *A service-oriented architecture is a style of multitier computing that helps organizations share logic and data among multiple applications and usage modes*, donnée en 1996 par le groupe Gartner [SN96] ;
– *Service Oriented Architecture is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations*, établie dans le modèle de référence OASIS [MLM+06] ;
– *SOA enables flexible integration of applications and resources by : (1) representing every application or resource as a service with standardized interface, (2) enabling the service to exchange structured information(messages, documents, "business objects"), and (3) coordinating and mediating between the services to ensure they can be invoked, used and changed effectively* [Dod04].

Malgré le manque de spécification officielle pour définir une architecture orientée services, trois rôles clés sont communément identifiés : producteur de services, répertoire de services et consommateur de services. L'interaction entre ces trois rôles est décrite par la Figure 2.1. Le producteur de services a pour fonction de déployer un service sur un serveur et de générer une description de ce service qui précise à la fois les opérations disponibles et leur mode d'invocation. Cette description est publiée dans un répertoire de services, aussi appelé annuaire. Les consommateurs de services peuvent découvrir les services disponibles et obtenir leur description en lançant une recherche sur le répertoire. Un consommateur peut alors utiliser

la description du service obtenue pour établir une connexion avec le fournisseur et invoquer les opérations du service souhaité.

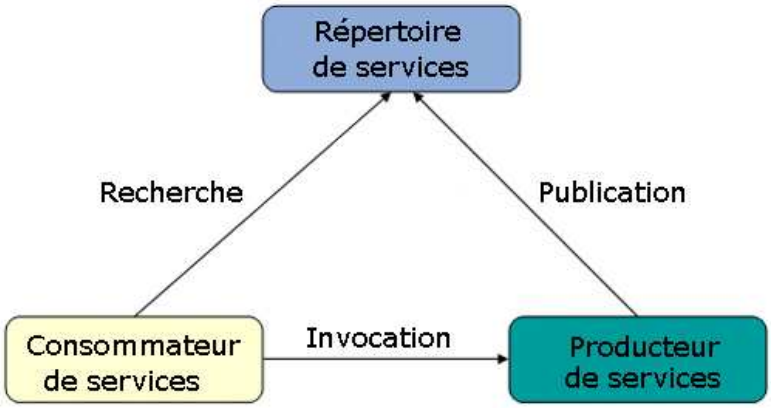

FIG. 2.1 – Architecture orientée services

Bien que les SOAs n'aient vraiment pris leur essor que ces dernières années, il existe des propositions plus anciennes sur ce même modèle telles que CORBA ou DCOM. Les SOAs sont en effet plus connues sous leur version Web services (Web Services Oriented Architectures ou WSOA) car elles reposent alors sur des technologies basées sur des standards XML et offrent ainsi une solution indépendante des implémentations, plates-formes, langages ou encodage de données propriétaires. Les trois protocoles de base sont UDDI, WSDL et SOAP. WSDL permet de décrire un service, UDDI de l'enregistrer et de le découvrir, et SOAP est utilisé comme protocole de transport pour envoyer les messages entre les consommateurs et les producteurs de services. Ces trois standards sont décrits plus en détail dans la partie refsec :standards.

2.2.2 La notion de couplage faible

Tout comme le concept de SOA, le concept de couplage faible ne donne pas lieu à une définition officielle qui fasse l'unanimité. L'objectif communément admis est cependant d'introduire le minimum de dépendances entre services pour permettre d'assembler ceux-ci aisément. Dans le contexte SOA, il s'agit notamment de favoriser la réutilisabilité de sous systèmes existants, déjà déployés, et de leur combinaison afin de répondre rapidement, et à faible coût, à de nouveaux besoins métier. Pour atteindre cet objectif, un certain nombre de règles d'ingénierie, spécifiques ou non au contexte SOA, ont été identifiées [Dod04, WF04, DWvH04, PvdH06, MJ05, Col04] comme favorisant le couplage faible et la réutilisabilité.

Du paradigme orienté objet sont repris les principes d'encapsulation et d'abstraction. L'idée est de cacher aux utilisateurs l'information contenue dans un objet et de ne proposer qu'une interface stable mettant l'accent sur les détails jugés nécessaires à sa manipulation. L'objet est vu de l'extérieur comme une boîte noire. Cela permet de découpler sa spécification (ce qu'on voit) de l'implémentation pratique de ces spécifications. On peut ainsi changer son implémentation sans changer son comportement externe.

Des systèmes distribués ont notamment retenu le modèle Publication/Souscription et le principe de la granularité à gros grain.

– Publication/Souscription est un modèle de communication où des producteurs de données publient des messages auxquels souscrivent des consommateurs. Ce modèle, mis en oeuvre par des brokers tels que par example les bus EAI (Entreprise Application Integration) et ESB (Entreprise Service Bus), favorise le découplage entre producteurs et consommateurs distribués, le passage à l'échelle et le déploiement incrémental des services.

– L'idée de la granularité à gros grain est que les opérations exposées doivent refléter des besoins métier - une notion relative mais qui vise à améliorer la réutilisabilité, l'efficacité et la robustesse des communications permettant de réaliser un besoin métier, en évitant de multiplier les échanges distribués et de gérer un état transitoire (*e.g.*, une seule opération pour passer un ordre d'achat, au lieu d'accesseurs individuels suivis d'une validation).

Les règles suivantes sont plus spécifiques au contexte SOA :
– L'indépendance aux aspects technologiques (*e.g.*, pas de références distribuées). Les contrats d'utilisation qu'exposent les services doivent être auto-descriptifs et indépendants de la plate-forme technique utilisée par le fournisseur du service. Par exemple, le langage WSDL (Web Services Description Language) permet d'exposer les fonctionnalités des Web Services en assurant cette indépendance.
– L'absence d'état. En toute rigueur, un service bien formé est sans état. Il reçoit les informations nécessaires dans la requête d'un client et ne le connaît plus une fois la réponse renvoyée. Cette règle qui peut sembler très contraignante doit être nuancée. Il est recommandé que la conservation des états (gestion de contextes) ainsi que la coordination des actions (transactions) soient localisées dans une fonction spécifique de l'architecture SOA, telle que l'orchestration. C'est le processus qui est avec état et non pas les services orchestrés. L'application d'une telle règle favorise la réutilisabilité, le passage à l'échelle et la robustesse des services.
– La cohésion. Cette règle, délicate à définir, traduit le degré de proximité fonctionnel des opérations exposées par un service. Elle vise à favoriser la facilité de compréhension et la réutilisabilité d'un service en y regroupant des opérations homogènes constituant une unité métier, ou de même catégorie comme des alternatives pour effectuer un paiement.
– L'idempotence. Un service idempotent ignore les réceptions multiples d'une même requête. Celui-ci est rendu correctement que la requête arrive une ou plusieurs fois. L'idée est que l'utilisation d'un tel service permet de relâcher les hypothèses de fiabilité sur la couche de communication.

On notera que, si certaines règles sont bien maîtrisées (*e.g.*, encapsulation et indépendances aux aspects technologiques), d'autres sont plus contraignantes (*e.g.*, idempotence) et à ce titre moins appliquées.

2.2.3 Infrastructures et modèles d'exécution pour SOA.

SOA est souvent présentée comme un style architectural permettant aux entreprises de créer rapidement de nouvelles applications, en transformant leurs ressources existantes en services réutilisables et en les composant aisément avec d'autres services en fonction des besoins. Si l'accent est effectivement mis sur les notions de services « métier » et de couplage faible, dans la pratique le besoin d'infrastructures et de modèles d'exécution facilitant l'intégration et l'administration d'applications orientées services, ainsi que la prise en compte de services techniques tels que la sécurité, les transactions, ou la QoS, s'est rapidement fait sentir. Plusieurs initiatives sont apparues pour répondre à ce besoin, parmi lesquelles on peut citer SOC, ESB et SCA que nous introduisons brièvement ci-dessous. Nous nous intéresserons spécifiquement aux Web Services dans la partie suivante.
SOC (Service Oriented Computing) [GP03] est une initiative académique qui vise à étendre SOA pour permettre d'administrer et composer les services de façon flexible. L'architecture proposée distingue trois niveaux. Le premier couvre SOA avec ses fonctions minimales de publication, découverte et liaison de services. Le deuxième porte sur la composition dynamique de services. Il est chargé d'adapter en cours d'exécution les services composites (processus, workflow, etc.), pour tenir compte d'observations telles que des mesures de QoS sur les applications coopérantes, de leurs contraintes (SLA, etc.) ou de la découverte de nouveaux ser-

vices. Le troisième couvre les fonctions d'administration nécessaires à la supervision globale des applications. Une conférence annuelle (ICSOC) fédère les développements et applications de l'approche dans des contextes variés, comme la découverte et la composition dynamique de services dans les environnements mobiles.

ESB (Enterprise Service Bus) [Bal05] est une réponse industrielle aux approches SOA qui porte plus spécifiquement sur le domaine de l'intergiciel et de l'intégration des systèmes d'information. Apparu fin 2002, le concept donne lieu à plusieurs implémentations industrielles ou open-source pour les Web services. Les services minimaux d'un ESB sont :

– de faciliter les communications et interactions de services. Un ESB supporte au moins un style de communication par message (*e.g.,* request/response, publish/subscribe, etc.), un protocole et un format de définition d'interfaces (*e.g.,* SOAP/WSDL). Il gère le routage, l'adressage, la transparence à la localisation, la substitution de services, et permet de découpler la vue utilisateur d'un service de son implémentation.
– de faciliter l'intégration et la supervision de services. Un ESB gère les adapteurs permettant d'interopérer avec des systèmes *legacy*. Il permet d'ajouter des fonctions de supervision, par injection au niveau de points d'interception ou de dérivation, sans actions intrusives auprès des applications.

Les ESBs plus avancés intègrent des services à valeur ajoutée (mais propriétaires) allant de mécanismes variés pour la QoS tels que l'écrêtage de flux, plage d'ouverture ou de remontée d'alarmes, au traitement sémantique des messages tel que la réécriture et l'agrégation d'information. Ils peuvent également intégrer des fonctions d'orchestration.

SCA (Service Computing Architecture) [IA05] est une initiative récente de plusieurs éditeurs logiciels, incluant notamment BEA, IBM, IONA, Oracle et SAP, qui vise à proposer un modèle à composants pour construire des applications dans un contexte SOA. SCA encourage une organisation basée sur des composants explicites qui implémentent la logique métier, et communiquent au travers d'interfaces de services fournies et requises masquant tout détail ou dépendance sur des APIs techniques. Le modèle distingue l'étape d'implémentation des composants de celle d'assemblage d'un ensemble de composants, consistant à lier leurs services. Il est récursif comme les modèles à composants de type Fractal. Les spécifications SCA couvrent également le moyen de packager des composants formant une unité de déploiement, et s'intègrent avec SDO (Service Data Object) qui complètent l'architecture SCA par une solution commune d'accès à différents types de données. Cette initiative est intéressante car - d'une part elle illustre la complémentarité entre les approches à composants et services plutôt que de les opposer, comme souvent résumé dans les débats sur le couplage fort et faible - d'autre part, elle propose une réponse à des besoins réels du marché et des utilisateurs en matière de plates-formes d'exécution et de déploiement SOA.

2.2.4 Le modèle de référence de OASIS

OASIS travaille sur un modèle de référence pour les architectures orientées services [MLM+06]. Cet effort a pour volonté de définir un cadre conceptuel commun qui pourra être utilisé de façon consistante à travers les différentes implémentations. Le but est donc d'établir les définitions qui devraient s'appliquer à toutes les SOAs, d'unifier les concepts pour expliquer les patrons de conception génériques sous-jacents, et de donner une sémantique qui pourra être utilisée de façon non ambiguë lors de la modélisation de solutions données.

OASIS définit une SOA comme étant un *"paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains"*. Plus précisément, le modèle de référence de OASIS repose sur les notions de besoins (needs) et de capacités (capabilities). Ainsi, des personnes ou des organisations créent des capacités afin de résoudre et d'apporter une solution à leurs problèmes. Les besoins d'une personne peuvent être adressés par les capacités offertes par une autre personne. Il n'y a pas de relation une à une entre les besoins et

les capacités. De plus la granularité des besoins et des capacités est guidée par le métier. Un besoin donné peut nécessiter de combiner plusieurs capacités, et une capacité peut répondre à plus qu'un seul besoin. Une architecture orientée service offre donc un cadre pour faire correspondre des besoins à des capacités et pour combiner des capacités pour répondre à ces besoins.

Service et dynamicité

Dans le modèle de OASIS le concept central est évidemment le service. Le modèle de référence souligne que cette notion de service communément correspond implicitement soit à la capacité d'effectuer une tâche pour un tiers, soit à la spécification de la tâche qui est offerte par un tiers, soit enfin à l'offre d'effectuer une tâche pour un tiers. Plus précisément, ils définissent un service comme étant un mécanisme qui permet l'accès à un ensemble constitué d'une ou de plusieurs capacités, où l'accès est rendu possible à travers une interface définie et est effectué de façon cohérente avec les contraintes et politiques spécifiées par la description de service. La personne qui offre un service est un fournisseur de service et celui qui l'utilise un consommateur. Un service est considéré comme étant opaque dans le sens où son implémentation est cachée au consommateur de service.

Autour de la notion de service, les concepts clés pour décrire le paradigme SOA et qui sont introduits dans une perspective de dynamicité sont la visibilité, l'interaction et l'effet. La visibilité fait référence à la possibilité pour les fournisseurs et les consommateurs de se "voir" mutuellement en vue d'interagir. Cela implique notamment la mise en place de mécanismes de découverte de service ainsi que la mise à disposition de toutes les informations nécessaires pour qu'un consommateur potentiel puisse prendre connaissance de l'existence et des capacités d'un service donné. L'interaction correspond à l'activité d'utilisation d'une capacité. Cela s'effectue typiquement à travers l'échange de messages. Enfin le but principal lors de l'utilisation d'une capacité est de réaliser un ou plusieurs effets.

En plus de ces concepts liés au support dynamique des interactions avec des services, le modèle OASIS identifie des concepts liés aux services eux-mêmes. Ces concepts regroupent notamment la description de service, ainsi que les contrats et politiques qui sont liés aux services et aux fournisseurs et consommateurs de services.

Description de service

Une description de service représente les informations nécessaires afin d'utiliser le service et facilite la visibilité et l'interaction entre les consommateurs et fournisseurs de services. Le modèle de référence d'OASIS précise qu'il n'existe pas qu'une seule "bonne" description ; les éléments d'une description dépendent du contexte et des besoins des différentes parties impliquées. De façon générale, une description de service doit au moins spécifier des informations nécessaires afin qu'un consommateur puisse décider d'utiliser ou non un service. Ainsi le consommateur a besoin de savoir :

- que le service existe et est accessible,
- que le service effectue une fonction donnée ou un ensemble de fonctions,
- que le service fonctionne selon un ensemble de contraintes et politiques données,
- que le service sera en accord avec les contraintes imposées par le consommateur,
- comment interagir avec le service, ce qui inclut le format et le contenu des informations échangées ainsi que la séquence des informations échangées qui doit être respectée.

Politiques et contrats

Une politique représente une contrainte ou une condition définie par un consommateur ou un producteur sur l'utilisation, le déploiement ou la description d'une entité qu'il possède. Ainsi, une politique peut par exemple spécifier que tous les messages reçus sont cryptés. Plus largement, une politique peut être appliquée sur différents aspects d'une SOA tels que la sécurité, la confidentialité, la qualité de service, ou même des propriétés métier. Garantir qu'une politique est respectée peut nécessiter d'empêcher certaines actions non autorisées ou le passage dans un état donné. Il peut aussi être nécessaire de mettre en oeuvre des actions pour compenser lorsqu'une violation de politique a été détectée. Détecter une violation de politique peut être réaliser sous la forme d'assertions qui doivent être compréhensibles et traitables de préférence de façon automatique.

Un contrat représente, quant à lui, un accord entre au moins deux parties. Tout comme les politiques, les contrats portent sur les conditions d'utilisation des services. Un contrat doit lui aussi être exprimé sous une forme interprétable, et cela afin de faciliter la composition automatique de services. Le respect d'un contrat, contrairement aux politiques où le bon respect d'une politique est la responsabilité du propriétaire de la politique, peut nécessiter la résolution de désaccords qui éventuellement est à la charge d'une autre entité faisant autorité.

2.3 Les Web services

Les Web services constituent une approche pour mettre en oeuvre le paradigme de service. Les Web services [Pap04] ont été proposés initialement par IBM et Microsoft, puis en partie standardisés sous l'égide du W3C. Le groupe de travail du W3C a produit un glossaire [Not04] dans lequel il définit un Web service comme :

"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialisation in conjunction with other Web-related standards."

Techniquement, les Web services se présentent comme des composants sans état dont l'enjeu majeur est de promouvoir un couplage faible. Pour cela les normes de communication associées aux Web services sont standardisées et entièrement spécifiées en XML, langage permettant une grande interopérabilité entre les systèmes. La large implication des industriels dans l'élaboration des normes à tous les niveaux témoignent de l'ambition collective de mieux faire communiquer les systèmes informatiques par le biais du Web.

L'architecture des Web services s'articule autour des trois standards XML suivants :

- SOAP (Simple Object Access Protocol) [GHM+03] est le principal protocole utilisé pour l'invocation de services (Figure 2.1).
- WSDL (Web Service Definition Language) [CCMW01] est le principal format de description de services.
- UDDI (Universal Description, Discovery and Integration of Web Services) [OAS02] est le principal standard permettant la publication et la découverte de services.

Ces trois standards sont détaillés dans la partie 2.3.3.

2.3.1 Architecture en couches

Une architecture Web services est une architecture en couche qui repose sur plusieurs protocoles [CBD05]. De nombreuses organisations telles que WebServices.Org, The Stencil group, IBM conceptual Web services stack (WSCA), W3C Web Services, Microsoft, Sun One (webservice architectures), Oracle, Hewlett-Packard, BEA Systems ou Borland proposent leur propre modèle [Mye02]. A titre d'exemple, la Figure 2.2 illustre la pile de protocoles proposée par le W3C, appelée la "Web Service Architecture" (WSA).

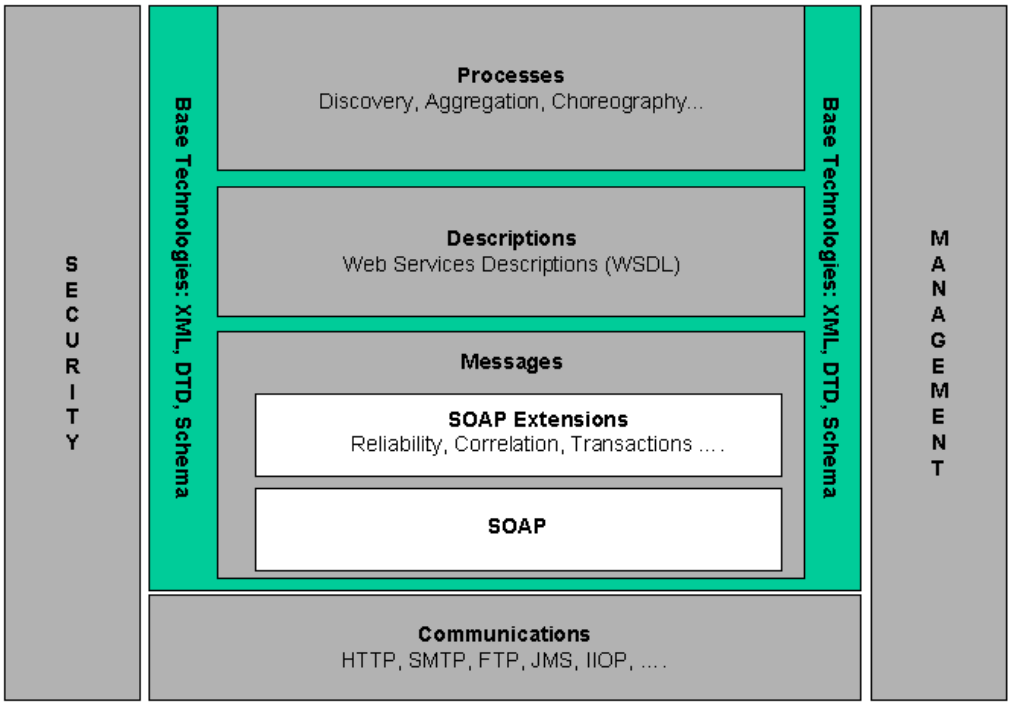

FIG. 2.2 – Architecture des Web services du W3C

Il y a trois couches dans la WSA : communication, message, gestion des processus. Chacune de ces couches se subdivise en domaine de granularité plus fine (*e.g.*, les normes de sécurité font partie d'un domaine de la couche message).

La couche communication adresse les aspects liés au transport des messages. Il est souvent possible de spécifier le style, le mode et l'encodage d'une communication. On distingue au moins deux styles de communication : le style RPC (Remote Procedure Call) et le style Document. En RPC, le Web service est similaire à l'approche distribuée traditionnelle. Cependant, de nombreuses variantes existent. Le style Document correspond à une communication où le client interagit avec le serveur non plus sous la forme de procédures mais de documents XML auto-descriptifs qui sont échangés. Trois mode de communication peuvent être envisagés : RPC ou mode requête-réponse, one-way messaging et asynchronous callback. Enfin l'encodage, qui spécifie comment les types sont représentés en XML peut être Literal (*i.e.*, suit littéralement les définitions de XML Schema) ou SOAP encoded (*i.e.*, suit la spécification de SOAP). Enfin le protocole de communication support le plus souvent utilisé est HTTP mais il peut être aussi SMTP, FTP, etc.

La couche message décrit la sémantique des messages SOAP. Composées fondamentalement par le trio SOAP, WSDL et UDDI, les spécifications de cette couche ont également pour objectif d'offrir aux applications distribuées des mécanismes complexes de sécurité, de garantie de livraison, de transaction, de description, etc. Ces spécifications évoluent régulièrement. Des organismes, tels que le WS-I, sont en charge de prendre en compte toutes les propositions et

de valider l'interopérabilité.
La couche gestion des processus regroupe tout ce qui est notamment lié à la composition et au management des services. Le management des Web services peut être défini comme un ensemble de capacités telles que la découverte de l'existence, la disponibilité, la santé, les performances, le contrôle, la configuration des ressources à l'intérieur des architectures à base de Web services. C'est le Web Services Distributed Management (WSDM) Technical Committee qui est en charge des normes de la gestion standardisée des Web services. La composition de services peut être simplement définie comme étant "le procédé consistant à combiner des services existants pour former de nouveaux services". Combiner des services peut se résoudre de deux façons distinctes : soit par le biais d'une chorégraphie, soit via une orchestration. Une chorégraphie décrit le flux de messages échangés par un Web service lors de son interaction avec d'autres services. Il s'agit donc d'une manière décentralisée de gérer une composition puisque chaque Web service est responsable d'une partie du workflow. La norme "Web Services Choregraphy Interface" (WSCI) [SISM02] permet de spécifier le comportement du Web service par rapport au reste de la composition. Une orchestration comprend un élément central responsable de la composition dans son ensemble. Le processus devient alors la somme de ses sous processus et l'orchestrateur gère, seul, les échanges de messages. De nombreux travaux se sont intéressés aux langages dédiés à l'exécution de processus métier, parmi lesquels WSFL d'IBM et XLANG de Microsoft. Ces efforts ont ensuite été fusionnés pour former une spécification commune nommée "Business Process Execution Language for Web Services" (BPEL4WS) [ISM+05].

2.3.2 Orchestration et Chorégraphie

Les Web services sont apparus dans le contexte des échanges entre entreprises sur Extranet, voire Internet. Ils sont aussi utilisés comme axes d'échanges entre des sous-systèmes hétérogènes du système d'information d'entreprises (voir précédemment ESB). Dans ce contexte d'utilisation deux points de vues sont possibles :
– la spécification externe, qui décrit l'enchaînement des Web services et les rôles attachés à l'utilisation d'un Web service : c'est la *chorégraphie*,
– la réalisation interne des échanges entre Web services contribuant, pour le compte d'un partenaire donné, à la réalisation de la chorégraphie, que l'on appelle *orchestration*.
La *chorégraphie* n'est pas encore standardisée (voir WS-Choreography spécifiée par le W3C), alors que la proposition Business Process Execution Language for Web Services (BPEL4WS) définit un dialecte XML destiné à exprimer l'orchestration de Web services.

Orchestration

Une orchestration exprime les conditions et les enchaînements des invocations WS et plus généralement aux services. C'est le modèle des interactions que doit suivre un service pour réaliser sa fonctionnalité. Dans le cadre des architectures orientée services, ces orchestrations sont vues comme des compositions de services définissant des applications.

BPEL4WS *BPEL4WS* décrit des processus exécutables qui sont des réalisations privées - *i.e.*, limitées au système d'information d'un partenaire - de processus collaboratifs. Un modèle BPEL4WS comporte deux sections :
– la description de l'orchestration proprement dite (ordonnancement des activités),
– le lien avec les Web services qui prennent en charge ces activités.
Ces deux sections se basent sur la description externe des services par leur WSDL. Il s'agit bien donc d'une composition de contrats avec une vérification centralisée : on peut vérifier la compatibilité syntaxique des services successivement activés, ainsi que certains éléments

pragmatiques, mais il n'y a pas de vérification sémantique, en l'absence d'expression sémantique des WSDL. La définition du langage *OWL-S* (Ontology Web Langage for Service) vise à répondre à ce besoin de validation sémantique (c'est un élément du Web sémantique). BPEL définit différentes activités primitives qui permettent d'exprimer les compositions d'interactions (invoke, reply, receive, wait), les affectations de données (assign) , la levée d'exceptions (throw), la terminaison d'une composition (terminate) ou l'activité ne rien faire (empty). Ces activités peuvent être composées via des activités structurées (sequence, switch, while, pick, flow).

OWL-S Une expérimentation d'utilisation du Web sémantique pour construire un workflow à partir de la description sémantique de services a été menée à EDF R&D.
Elle a consisté à développer des ontologies spécifiques au domaine électrique (ETSO) pour exprimer le caractère sémantique des concepts utilisés dans les Web services qui étaient potentiellement à enchaîner. Une seconde ontologie plus spécifique a été spécifiée, concernant la description des paramètres d'entrée-sortie utilisés pour l'agencement de ces services.
La description des services a été faite dans le langage OWL-S. L'ensemble des services participant à la transaction ETSO se sont vus attribuer un fichier OWL-S, partiellement engendrés à l'aide d'une conversion de la description WSDL préexistante des Web services définis par ETSO. L'enchaînement de services s'est fait ensuite sur une mesure de leur compatibilité entre eux.

Chorégraphie

La complexité des interactions entre les Web services, notamment lors des transactions commerciales, rend leur modélisation compliquée. Plusieurs points de vue peuvent être abordés lors de la modélisation mais la modélisation d'un point de vue global permet de prendre en compte des situations de concurrences dans les environnements distribués. La chorégraphie va permettre en exprimant les conditions et les enchaînements des invocations aux Web services de donner une vue flexible du processus global. Les langages de description des chorégraphies tels que WS-CDL sont nécessaires, tout comme BPEL, pour assurer la cohérence des comportements des points d'entrées entre les services qui opèrent.

Interoperabilité

Les communications entre Web services sont basées sur des échanges de documents XML. Un consortium initié par IBM et Microsoft nommé WSI (Web Services interoperabilty) veille à la compatibilité des mises en œuvres des protocoles qui composent les Web services. Les deux objectifs principaux du WSIO est d'élaborer des tests de compatibilité des implémentations des Web services et de travailler sur la prochaine génération de protocoles devant renforcer la sécurité et l'intégrité des processus.

2.3.3 Standards industriels

Les Web services reposent sur un certain nombre de standards. Les trois standards principaux sont SOAP, WSDL et UDDI que nous détaillons ci-dessous.
Notons qu'il existe de nombreuses plates-formes pour les Web services comme par exemple le Web Services Platform de Sun ou le Web Services Toolkit d'IBM. Diverses implémentations pour standards sont aussi disponibles, comme par exemple Apache Axis pour SOAP, jUDDI pour UDDI qui sont tous les deux issus du Jakarta Apache Project, ou encore le SOAP Toolkit de Microsoft.

SOAP

Originellement appelé SOAP pour "Simple Object Access Protocol", le protocole d'échange de messages entre Web services est un standard ouvert et entièrement basé sur le langage XML. Ce protocole ne requiert aucune plate-forme spécifique, ni d'exigence particulière en ce qui concerne l'implémentation de l'application. Au contraire, ce mécanisme fournit une sémantique tout à fait indépendante des contraintes techniques et, de fait, compréhensible par tous. Dans la pratique, le transfert est le plus souvent assuré via le protocole HTTP. SOAP peut cependant reposer sur d'autres protocoles de transport comme par exemple SMTP ou JMS (Java Message Service).
Un message SOAP est un document XML dont la structure est spécifiée par des schémas XML. Plus précisément tout message SOAP se compose d'un unique élément *envelope* qui englobe une partie *header* et une partie *body*. La partie header contient les métadonnées concernant d'éventuelles propriétés non fonctionnelles du service (jeton de sécurité, contexte de transaction, certificat de livraison, etc.) tandis que la partie body regroupe les éléments métier tels que les appels de méthode avec transfert des données spécifiques.

Cette structure permet une séparation nette des différentes préoccupations auxquelles doit répondre le service. Généralement, les données header des messages SOAP sont traitées par des filtres (handlers) placés autour de l'implémentation fonctionnelle du service. Conceptuellement, le header offre une possibilité d'extension des messages en rajoutant des données spécifiques au contexte. Toutefois, le consommateur et le producteur du service doivent se comprendre sur la sémantique de ces données supplémentaires, d'où le rôle fondamental de normalisation des travaux de la "Web Service Architecture".

WSDL

Comme tout composant, les Web services sont atteignables par le biais d'interfaces. Il s'agit du contrat WSDL qui s'apparente aux Interface Definition Languages (IDL) déjà existants, puisqu'il s'intéresse à fournir une abstraction fonctionnelle du service. WSDL repose sur XML pour décrire de façon abstraite (et donc indépendante de l'implémentation du service) l'ensemble des opérations et des messages qui peuvent être échangés avec un service donné. Une description WSDL contient aussi des informations sur le protocole de communication utilisé, le format d'encodage des données, et l'adresse URL à laquelle le service est accessible. Un contrat rédigé en WSDL doit permettre au client de trouver l'adresse du service et de rédiger des messages compréhensibles par ce dernier.

Ainsi, le WSDL se compose en deux parties. Une première partie définit de manière abstraite les éléments, les opérations et les types de données, tandis qu'une seconde partie précise de manière concrète les adresses physiques de ces opérations ainsi que le mapping des données avec les protocoles de transport. Cette distinction est utile car elle permet de concevoir le service indépendamment de l'environnement de déploiement.

Plus précisemment, le WSDL définit les Web services à travers six éléments :
– types : décrivent sous la forme d'une spécification XML Schema les types des données échangées entre le client et le fournisseur de services,
– message : définit les informations échangées lors d'une requête ou d'une réponse. Un message a un nom et potentiellement des *parts* qui font référence à des paramètres et des valeurs de retour,

– portType : combine plusieurs messages pour former une opération. Il y a quatre types d'opération : one-way, request-response, solicit-response et notification,
– binding : spécifie le protocole de communication (le plus souvent HTTP, mais aussi SMTP, FTP, etc.) et le format d'encodage des données (encodage RPC, Literal Document, etc.) pour les opérations et messages définis par un type de port donné. Il est possible grâce à des extensions internes de WSDL de définir des binding SOAP,
– port : est un point d'accès au service identifié de manière unique par la combinaison d'un binding et d'une adresse internet,
– service : regroupe un ensemble de ports, chaque port offrant une alternative (différents protocoles, etc.) pour accéder au service.

UDDI

Il existe également un mécanisme de découverte des Web services : il s'agit de "Universal Discovery Description and Integration" (UDDI) qui peut être considéré comme un annuaire de Web services.
UDDI permet aux entreprises de s'inscrire et de publier leurs Web services. UDDI se comporte lui-même comme un Web service dont les méthodes sont appelées via le protocole SOAP. Les opérations pouvant être effectuées concernent la recherche, la navigation, l'ajout, et la suppression de services.
UDDI permet aux développeurs de découvrir les détails techniques d'un Web service (le WSDL) ainsi que les informations orientées métier. Par exemple, en utilisant UDDI, il devient possible de répertorier tous les Web services reliés aux informations boursières, implémentés sur HTTP, et qui sont gratuits d'utilisation. De plus, UDDI permet de fournir un niveau d'indirection permettant un couplage faible entre les services d'une composition. Concrètement un répertoire UDDI est un fichier XML qui décrit un business et les services qu'il offre.
Il existe plusieurs catégories de classification pour les Web services :
– Pages blanches : les Web services sont classés par fournisseurs de services,
– Pages jaunes : les Web services sont classés par catégories,
– Pages vertes : contiennent des informations techniques sur les Web services.

2.4 Problématique autour des services

Tandis que la démarche SOA préconise le faible couplage entre les services, les travaux relatifs aux Web services appuient cette indépendance sur le langage d'interface WSDL. Ce formalisme ne suffit pas à caractériser les différents constituants d'un service tels que la qualité de service ou les protocoles d'appels nécessaires (lorsque celui est conversationnel), etc. Le chapitre suivant 3 aborde cette problématique.
Une des avancées de la proposition SOA est de favoriser l'expression des interactions entre services par des langages adaptés. La mise en oeuvre de ces compositions, leur positionnement vis à vis de la problématique de la séparation des préoccupations, leur adaptation dynamique restent des points difficiles. Le chapitre 4 fait le point sur différents travaux relatifs à la problématique de la composition.
La démarche SOA et les standards relatifs aux Web services sont jeunes et en pleine évolution. Il est donc nécessaire d'amorcer tout travail autour de ces techniques en ne s'attachant pas à un standard particulier. Dans un même temps, il est nécessaire de valider nos travaux par une concrétisation, si possible efficace. La validation de nos recherches par la mise en oeuvre au niveau des applications exige d'atteindre des plates-formes particulières qui, dans notre projet, sont par ailleurs hétérogènes à la fois dans les langages ciblés (Java ou C#), les modes de composition (composants, aspects, annotations) et les supports embarqués ou non. C'est

pourquoi il est nécessaire dans ce contexte mouvant de suivre une démarche IDM présentée au chapitre 5.
L'adéquation entre les compositions ainsi définies et les capacités de contrats doivent également être vérifiées. C'est ce point qui constituera le coeur de recherche du projet FAROS.

Bibliographie

[Bal05] N. Balani. Model and build esb soa frameworks. *IBM, developerWorks, http ://www-128.ibm.com/developerworks/web/library/wa-soaesb/*, 2005.

[CBD05] CBDI. Web services protocols summary. http ://road-map.cbdiforum.com/reports/protocols/summary.php, July 2005.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) specification. http ://www.w3.org/TR/wsdl, March 2001.

[Col04] Mark Colan. Service-oriented architecture expands the vision of web services. *IBM, developerWorks*, 2004.

[Dod04] Mahesh H. Dodani. From objects to services : A journey in search of component reuse nirvana. *Journal of Object Technology*, 3(8) :49–54, 2004.

[DWvH04] N. Dokovski, I. Widya, and A. van Halteren. Paradigm : Service oriented computing. *(Freeband/AWARENESS/D2.7b) TI/RS/2004/145*, 2004.

[GHM+03] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. Frystyk Nielsen. Simple Object Access Protocol (SOAP) specification. http ://www.w3.org/TR/soap12-part1/, June 2003.

[GP03] D. Georgakopoulos and M. Papazoglu. Service-oriented computing. *Communications of the ACM, Vol. 46, No 1*, 2003.

[Has03] Sayed Hashimi. Service-oriented architecture explained. `http://www.ondotnet.com/pub/a/dotnet/2003/08/18/soa_explained.html`, August 2003. O'reilly on dot net.

[IA05] IBM and All. Service component architecture, building systems using a service oriented architecture, version 0.9. *Joint Whitepaper by BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, Sybase.*, 2005.

[ISM+05] IBM, BEA Systems, Microsoft, SAP AG, and Siebel Systems. Business process execution language for web services version 1.1. http ://www-128.ibm.com/developerworks/library/specification/ws-bpel/, July 2005.

[MJ05] Vinod Muthusamy and Hans-Arno Jacobsen. Small scale peer-to-peer publish/subscribe. In *International Workshop on Description Logics (DL2005), Edinburgh, Scotland*, 2005.

[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.

[Mye02] Judith M. Myerson. Web services architectures. http ://www.webservicesarchitect.com/content/articles/webservicesarchitectures.pdf, January 2002.

[Not04] W3C Working Group Note. Web services glossary. http ://www.w3.org/TR/2004/NOTE-ws-gloss20040211, February 2004.

[OAS02] OASIS. OASIS UDDI specification. http ://www.uddi.org/specification.html, July 2002.

[Pap04] Mike P. Papazoglou. Web services : Concepts, architectures, and applications. In *Springer Verlag*, 2004.

[PvdH06] M.P. Papazoglou and W.J. van den Heuvel. Service-oriented design and development methodology. *Int. J. of Web Engineering and Technology (IJWET)*, 2006.

[SISM02] BEA Systems, Intalio, SAP, and Sun Microsystems. Web service choreography interface (wsci) 1.0. http ://www.w3.org/TR/wsci/, August 2002.

[SN96] Roy W. Schulte and Yefim V. Natis. Service oriented architectures, part 1 & 2. http ://www.gartner.com, April 1996. Gartner Group.

[Szy98] C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[WF04] Guijun Wang and Casey K. Fung. Architecture paradigms and their influences and impacts on component-based software systems. In *37th Hawaii International Conference on System Sciences (HICSS)*, 2004.

CHAPITRE 3

Contrats

Coordonnateur : Philippe Collet.
Rédacteurs : Fabien Baligand, Hervé Chang, Pierre Combes, Laurence Duchien, Alain Ozanne, Anne-Marie Pinna-Dery, Nicolas Rivierre, Roger Rousseau, Bruno Traverson.

3.1 Introduction

Dans ce chapitre, nous étudions les différentes approches contractuelles qui ont été proposées dans les approches par objets, puis par composants, et dernièrement dans les architectures orientées services. Nous nous intéressons donc aux formes de contrats et à certaines techniques de spécification et de vérification dans ces trois paradigmes.
Historiquement, les propriétés d'un programme ont pu être exprimées à l'aide d'assertions qui sont parfois appelées aussi contrats. Une assertion dans un programme est une expression booléenne qui doit être satisfaite pour que le code associé soit exécuté correctement. Les assertions viennent des travaux de Floyd [Flo67] et de Hoare [Hoa69]. Elles servent essentiellement dans la spécification des programmes, des objets et plus récemment dans la spécification des composants. Les programmeurs utilisent des assertions dans la description ou le raffinement de contraintes de typage définies dans les interfaces de classe. Le travail le plus connu dans ce domaine est probablement celui d'Eiffel [Mey92]. Le langage Eiffel contient des éléments natifs pour exprimer des assertions dans des pré et postconditions et dans des invariants de classes pour décrire des contrats entre l'utilisateur et le développeur d'une classe. Cette forme de contrat définit ainsi des propriétés vérifiables à l'exécution. Chaque spécification peut être interprétée informellement pour déterminer qui est censée la garantir. En cas de violation, cela permet d'établir des responsabilités.
D'autres travaux utilisent une notion de contrat pour définir, un peu de la même manière, des propriétés à vérifier, avec une notion de responsabilité vis-à-vis de la garantie des propriétés. Ainsi, dans notre étude, nous pourrions aborder la grande majorité des techniques de vérification et de validation en essayant de dégager ces caractéristiques ; mais afin d'utiliser un spectre pertinent, nous ne conserverons que les approches qui se donnent une interprétation contractuelle ou qui sont directement en rapport avec les architectures orientées services.
Pour structurer notre étude, un point de départ pertinent est la taxonomie de différentes formes de contrat établie pour des composants logiciels par Beugnard et al [BJPW99]. Elle distingue les quatre niveaux de contrat suivants :

– basique, relatif aux propriétés syntaxiques (noms des méthodes et type des paramètres) et les propriétés sémantiques simples (e.g. les langages de définition d'interfaces avec les IDL).
– comportemental, relatif aux propriétés qui peuvent être spécifiées avec des préconditions, des postconditions, et des invariants, en incluant les contraintes sur l'historique. Ces propriétés sont des propriétés liées à une méthode, cette méthode étant vue comme une unité atomique.
– synchronisation : il s'agit ici des propriétés concernant les interactions des composants. Cela correspond à la prise en compte des interactions du composant avec l'extérieur, mais ayant une influence sur son comportement interne. On peut décrire leur exécution sous

forme d'une suite d'appels de méthodes selon un motif spécifié. Le principe de l'atomicité, ici, ne peut être conservé, l'indéterminisme des appels et la concurrence de l'exécution des opérations est à prendre en compte dans la description de ces propriétés.
– qualitatif, ainsi associé à toutes les propriétés extra-fonctionnelles, telles que la qualité de service, le temps de réponse.

Notre contexte se focalise sur les architectures orientées services et sur les nouvelles formes d'architectures logicielles à base de composants. Il nous faut ainsi tenir compte de l'importance des architectures logicielles pour la manipulation explicite de l'organisation des entités logicielles, ainsi que des spécificités des architectures orientées services. Un certain nombre de formes de contrats ont ainsi été proposées pour assurer la cohérence des architectures. Ces contrats peuvent ainsi être considérés comme basiques, car ils s'appuient sur des notions de type, ou comme extra-fonctionnels, car relatifs à l'architecture. Nous classifierons donc ces formes de contrats dans des contrats d'architecture. De même, différentes sortes de contrats ont été proposées pour décrire les aspects comportementaux. Un nombre important d'entre eux utilisent des formalismes développées pour la spécification concurrente, qui définissent de manière insécable un comportement et des contraintes de synchronisation. Nous préférons donc caractériser ces contrats comme des contrats comportementaux.

Nous adaptons ainsi un découpage différent de la classification précédente, en étudiant d'abord, dans les contextes objets et composants, toutes les formes de contrats comportementaux (partie 3.2), puis ceux relatifs à la qualité de services (partie 3.3), et enfin ceux portant sur l'architecture (partie 3.4). Les formes de contrats propres aux services et aux architectures orientées services seront ensuite étudiés (partie 3.5) Enfin, les caractéristiques les plus pertinentes des modèles, métamodèles et autres formalismes de contrats sous-jacents seront présentées (partie 3.6).

3.2 Les contrats comportementaux

Nous considérons ici les principaux modèles, formalismes et systèmes à base de contrats comportementaux, au sens large, appliqués aux objets et aux composants. Nous incluons ainsi différentes formes de description comportementale : assertions exécutables et dérivés (partie 3.2.1), programmes abstraits (partie 3.2.2), automates (partie 3.2.3), algèbres de processus (partie 3.2.4), et diagrammes de séquence (partie 3.2.5).

3.2.1 Assertions exécutables et dérivés

Eiffel

Bien que d'autres langages d'assertions aient été développés pour des langages procéduraux et modulaires, le langage Eiffel [Mey91] définit le premier une approche contractuelle complète basés sur des assertions. Ce langage de programmation inclut nativement des préconditions (vérifiées à l'entrée de la méthode) et postconditions (vérifiées en sortie) sur les méthodes, des invariants de classes, ainsi que des assertions internes aux corps des méthodes. Il permet d'interpréter ces assertions comme des contrats. Un contrat est ainsi défini par analogie au monde réel, comme une spécification d'obligations mutuelles entre deux parties au moins. Les contrats en Eiffel peuvent alors prendre deux formes [Mey92] :
– le *contrat de clientèle* qui lie un client et un fournisseur *de classe*, le client étant responsable des préconditions de la méthode qu'il appelle, le fournisseur étant responsable des postconditions de la méthode qu'il implémente ;
– le *contrat d'héritage* qui lie l'auteur initial d'une classe et les développeurs qui l'adaptent par héritage — une classe fille, peut ajouter des assertions à celle de sa classe parente en suivant

des règles prédéfinies pour les invariants (conjonction des invariants des classes parentes), ajout alternatif des préconditions (require-else), et conjonction des postconditions.

Dans les assertions, il est possible d'invoquer des méthodes et le contenu d'une expression antérieure à l'exécution d'une méthode peut être utilisé après celle-ci à l'aide du mot clé **old**. Les assertions peuvent aussi être dénotées par un nom, qui sera utilisé pour les désigner en cas de violation. Si une assertion n'est pas vérifiée, sa catégorie détermine l'endroit où se produit l'exception associée. Une violation de précondition génère une exception dans la méthode cliente, un échec de postcondition crée une exception dans la méthode fournie, un invariant dans la méthode du fournisseur ayant causé la rupture de l'invariant.

Adaptations à Java

De nombreux travaux et propositions définissent des assertions pour le langage Java. Le lecteur pourra se reporter à [Plö02] pour un tour d'horizon de ces travaux, qui appliquent tous les mêmes principes mais diffèrent surtout par leurs techniques d'implémentation. Parmi eux, iContract [Kra98] fut le premier prototype fonctionnel. Il était basé sur un préprocesseur (source vers source) et utilisait les annotations javadoc pour intégrer les assertions aux fichiers sources Java. On notera aussi ST-Class [DFF+99] qui fournit un système d'assertions en Java, mais utilise aussi la technique de classes auto-testables dans laquelle des tests générés à partir des assertions, sont placés directement dans les classes.

JML

JML [LBR99] se définit comme un *Behavioral Interface Specification Language* dans la lignée de Larch. Ce dernier insère des commentaires dans un code source pour lui associer un modèle algébrique abstrait développé séparément. Le modèle abstrait étant écrit dans un langage commun *Larch Shared Langage*, cela permet de réutiliser des outils de preuves indépendamment des langages traités. Ainsi une spécification JML se compose de :

– une spécification d'interface qui porte sur les éléments de langages que cette dernière présente, telles les méthodes,
– une description de comportement, ainsi que d'autres propriétés ou contraintes, reposant sur les éléments décrits dans l'interface, exprimée à l'aide d'un modèle.

Le lien entre le modèle et le programme auquel on l'applique est réalisé à l'aide de pré, postconditions et d'invariants. JML permet ainsi de décrire des assertions classiques de l'approche par contrat en Java. Il propose pour ce faire des quantificateurs de collection et des bibliothèques *ad hoc*. Il est aussi possible de spécifier les conditions correspondantes aux levées d'exception. L'héritage des spécifications est pris en compte par la vérification des affaiblissements et renforcements nécessaires. D'autre part, le modèle de comportement d'une classe peut aussi reposer sur la définition de variables dites de modèles. Celles-ci sont déclarées dans le code JML et visibles uniquement de celui-ci. Grâce à ces variables, la spécification peut s'appuyer sur la description de modèles plus abstraits et génériques que les contrats traditionnels. Par ailleurs, il est possible d'établir un lien entre les variables de modèles et l'implémentation des classes par la définition d'une fonction d'abstraction.

JML présente ainsi la particularité de combiner des vérifications dynamiques (par l'évaluation à l'exécution des assertions) et statiques. Différents outils d'analyse ont été greffés à la machinerie JML, comme ESC/Java et ESC/Java 2, qui effectuent des tests au delà de la vérification de type (pointeur nul, dépassement de limites, erreur de transtypage, effet de bord). L'outil Chase complète ESC/Java en testant les clauses `assignable` qui spécifient quelles variables doivent être exclusivement modifiées par les méthodes. D'autres outils transforment les annotations JML en entrées pour prouveurs de théorème (LOOP), ou produisent des obligations de preuve, pour le prouveur de théorème Coq (Krakatoa, pour un sous-ensemble de Java) ou

pour des prouveurs B (JACK). Enfin, d'autres outils facilitent la production de spécifications ou la génération de tests à partir des spécifications.
Il est à noter qu'une grande partie des mécanismes de JML ont été transférés dans le langage Spec# [BLS05], dédié au langage de programmation C#.

Contrats d'interaction entre objets

Dès 1990, les travaux de Helm et Holland [HHG90] définissent la spécification de la composition comportementale dans les systèmes orientés objets à l'aide de contrats. Il s'agit alors d'écrire des contrats entre les différents éléments en interaction. Le contrat est défini comme une construction de langage pour la spécification explicite de compositions comportementales, par la description d'un ensemble de *participants* qui communiquent et de leurs obligations contractuelles. Ces obligations étendent la signature usuelle des types pour saisir les dépendances comportementales entre objets. Le contrat définit aussi des préconditions sur les participants pour établir le contrat et des invariants qui doivent être maintenus par les participants.
Le langage de contrat est spécifique à l'approche, car il décrit les comportements en terme de séquences d'actions à déclencher et de conditions qui doivent être satisfaites. Le langage supporte l'envoi de message et la modification d'une variable (par un mécanisme différentiel proche des spécifications ensemblistes). Les actions peuvent être simplement séquencées, organisées en *if-then-else*, exécutées facultativement (opérateur `?`) ou un nombre de fois non précisé (opérateur `*`). Il faut enfin noter qu'il y a une visibilité complète entre les participants, les problèmes de référencement mutuel étant laissés à l'implémentation. Outre les différentes constructions syntaxiques, l'essentiel de ce langage est de pouvoir spécifier des obligations causales entre les envois de messages.
Entre les contrats exprimés, il y a une possibilité d'héritage qui correspond à une relation de sous-typage ou de réutilisation, en supposant une inclusion complète. Les contrats sont ainsi définis indépendamment des classes, mais l'implémentation des classes doit être mise en conformité avec les spécifications. Il faut donc effectuer des *déclarations de conformité* qui lie une classe — ou une classe et ses super-classes abstraites pour former un tout cohérent — à un participant de contrat : la liaison de conformité se fait par typage structurel (conformité des signatures) avec des contraintes entre classes et sous-classes (obligation pour une sous-classe de fournir, d'implémenter…). La liaison entre les participants pourrait se faire par nommage de type, mais ce serait effectivement moins général pour un contrat qui a vocation à s'adapter à différents participants.

Adaptations à .NET et J2EE

Des adaptations de l'approche par contrats pour les composants logiciels ont été proposées.
L'intégration la plus traditionnelle est réalisée par le ContractWizard [KA01]. Cet outil exploite l'intégration d'Eiffel dans la plate-forme .NET. Il permet d'appliquer des pré/post-conditions et invariants sur les classes des langages à objet supportés par .NET, en produisant pour chacune d'entre elles un adaptateur en langage Eiffel. Il présente l'avantage de ne pas nécessiter le code source des classes cibles. Ainsi à partir d'un *assembly* (unité de déploiement .NET), il produit les wrappers Eiffel associés et les range dans un nouvel *assembly* qui prend alors la place du précédent. Une autre forme de contrats pour des composants .NET a été développée [BS03], mais ces contrats reposent sur le langage AsmL d'annotation par programme abstrait (cf. partie suivante) pour spécifier séparément les interfaces. La vérification se fait alors à l'exécution, en interprétant le programme abstrait en parallèle avec le code du programme.

Une utilisation classique de l'approche par contrat appliquée aux EJB est décrite par [VTS02]. Le conteneur EJB est modifié pour inclure des intercepteurs dans les interfaces EJB qu'il produit, et le serveur (JBoss) est surveillé à travers son interface JMX. La vérification des pre/post conditions est alors déclenchée par les intercepteurs en amont et aval de l'exécution des services des EJB. Les invariants sont exprimés au niveau de l'application et leur évaluation peut être effectuée régulièrement dans le temps à l'aide des fonctionnalités JMX, ou bien déclenchée par des intercepteurs. A la différence de l'approche par contrat traditionnelle, l'évaluation des préconditions n'est pas réalisée coté client, mais effectuée par le conteneur EJB.

Autour d'OCL

Le langage OCL. Le langage OCL [Obj97] a été créé pour la version 1.1 de la notation UML, afin de fournir un moyen d'expression de contraintes dans les diagrammes de conception. OCL est un langage typé et fonctionnel qui fournit un support pour spécifier des invariants sur les classes et les types, pour décrire des préconditions et des postconditions sur des opérations et pour définir des gardes sur des transitions d'états. OCL ne s'occupe d'aucun détail d'implémentation et a d'ailleurs été créé au départ sans aucun souci d'exécutabilité. OCL utilise les types prédéfinis dans le métamodèle UML, comme des types de base. Les valeurs et des opérateurs prédéfinis sont alors disponibles. OCL fournit en plus des types pour les énumérations et tout *classifieur* UML peut être utilisé dans une expression OCL. Des collections génériques sont aussi disponibles, avec un certain nombre d'opérateurs (sélection, itération, quantification). OCL peut être utilisé pour annoter des diagrammes de classe, en exprimant des spécifications de classe. L'opérateur postfixé `@pre` permet de référencer une valeur avant l'appel de la méthode dans les postconditions. Des invariants sont aussi exprimables, depuis la version 1.3 d'UML, a l'aide du stéréotype `inv :`. Il est à noter qu'en voulant assurer l'absence d'effet de bord dans les expressions, les concepteurs d'OCL en ont fait un langage purement fonctionnel, ce qui implique que les objets manipulés n'ont pas d'identité, et qu'on ne peut donc comparer que des valeurs.

Vérifications de spécifications OCL. Afin de vérifier des spécifications OCL, plusieurs travaux se sont intéressés à l'exécutabilité d'OCL, notamment l'*OCL-compiler* [HDF00], qui fournit une plate-forme de manipulation de spécifications purement OCL. Les spécifications OCL sont analysées en fonction d'informations extraites de diagrammes UML. Cette plate-forme propose la génération de code de vérification des assertions en Java, selon les mêmes principes généraux que les prototypes d'intégration d'assertions en Java. Plus récemment une réalisation reposant la programmation par aspect a été produite [DBL05] qui supporte quasiment toute la syntaxe de la version 1.3 d'OCL (seules l'Enumeration et l'opération OclInState ne sont pas supportées). Les contraintes OCL de la modélisation UML sont dans un premier temps traduites en arbres de syntaxe abstraite. Ceux-ci sont par la suite transformés en code Java avec l'assistance de requêtes au modèle UML et au programme contraint. Puis AspectJ est utilisé pour leur tissage dans le code du système étudié.
D'autres travaux s'intéressent aussi à effectuer des vérifications plus statiques à partir de spécifications OCL. Il est par exemple possible de produire des parties de spécification de machines abstraites B [LS02] en traduisant assez facilement les types de base et les opérations sur collections OCL vers les éléments constitutifs de la méthode B. Dans [DKR00], les auteurs définissent le formalisme BOTL (*Object-Based Temporal Logic*), une extension objet de la logique temporelle par branchement (CTL), et utilisent ce formalisme logique pour définir formellement certaines parties du langage OCL. Comme leur formalisme supporte le *model-checking* [BBL+99], ils disposent alors d'outils de vérification automatique de leur sous-ensemble OCL. Cependant, leur modèle BOTL n'est pas entièrement orienté objet puisque, ni l'héritage, ni

même le sous-typage ne sont considérés. Le sous-ensemble retenu n'est alors qu'une version modulaire du langage OCL qui supporte l'encapsulation.

Quelques extensions d'OCL. Un grand nombre d'extensions d'OCL ont été proposées depuis sa création. La méthode Catalysis [DW98] s'est reposée sur la notation UML et une version étendue du langage OCL. Ces extensions concernent notamment un opérateur qui permet de dénoter qu'une opération doit forcément en appeler une autre. Ceci a servi de base à une des fonctionnalités ajoutées dans la version 2 d'OCL (voir ci-après). Une autre extension concerne l'introduction d'opérateurs de logique temporelle dans OCL. Le langage TOCL [ZG02] comporte des opérateurs pour exprimer des contraintes dans le temps futur (état suivant, quantifications temporelles). Une extension spécifique au domaine du temps-réel [FM02] montre comment spécifier plus d'aspects dynamiques à l'aide d'opérateurs temporels ajoutés à OCL, et dont la sémantique est définie en logique temporelle bornée par le temps, basée sur CTL (*Computational Tree Logic*). A OCL sont ajoutées des notions de configuration d'états et de chemins à travers des séquences d'exécution. A partir de n'importe quel objet OCL, il est alors possible d'obtenir les chemins temporellement précédents et suivants, ainsi que le suivant direct et toutes les configurations disponibles. A l'aide des opérateurs OCL existants comme `forAll` et `exists`, la sémantique de ces opérateurs est équivalente à la plupart des opérateurs créés dans TOCL et évidemment définissables dans une logique temporelle. Ici, un processus de validation est possible, en traduisant les expressions pour qu'elles soient traitées par un outil de *model-checking*.

OCL 2.0. La version 2.0 d'UML [OMG03] ne modifie pas fondamentalement la philosophie du langage OCL, mais propose un certain nombre d'améliorations. Une clause `body` permet de décrire des méthodes d'interrogation (fonction sans effet de bord) du modèle, en contenant la formule correspondante à la valeur à retourner. Ceci permet d'utiliser OCL non plus seulement comme un langage de contraintes, mais comme un langage de programmation interprétable sur des modèles UML. Dans OCL 2.0, il est aussi possible de contraindre l'évolution de la valeur d'attributs. D'autre part les structures de données ensemblistes ne sont plus aplaties lors de leur parcours, ce qui permet de disposer de listes de listes. Des quantificateurs universels ou existentiels portant sur les extensions de classes sont aussi disponibles. Enfin, OCL propose une entité réifiant un message, ainsi qu'un opérateur indiquant son envoi, nommé `hasSent`, et noté `^`. Ce dernier permet par exemple de stipuler la nécessité qu'un message ait été envoyé au cours de l'exécution d'une méthode. En outre,les messages réifiés d'OCL permettent la gestion de l'asynchronisme, chaque message dispose en effet d'une méthode `hasReturn` et d'une méthode `result`. Elles permettent de savoir respectivement si une réponse a été faite au message et, si oui, quelle en est sa teneur.

ConFract

Première version du système. ConFract [CRCR05] se présente comme un système pour organiser sous forme de contrats la spécification et la vérification de propriétés fonctionnelles et extra fonctionnelles sur des composants logiciels Fractal [BCL+04]. À partir de spécifications, le système construit des contrats lors des assemblages de composants. Ces contrats sont alors des objets de première classe pendant les phases de configuration et d'exécution des composants. Le système ConFract a été conçu pour séparer clairement les mécanismes contractuels de l'expression des spécifications. Dans cette première version, seul un langage d'assertions exécutables, nommé CCL-J (*Component Constraint Language for Java*), est intégré au système. Ce langage, partiellement inspiré d'OCL [Obj97], est dédié à Java et Fractal. Il permet notamment d'exprimer des assertions avec une portée globale à un composant, éventuellement composite, et non pas uniquement sur une interface.

Dans ConFract, trois grands types de contrats sont distingués. Un *contrat d'interface* est établi sur chaque connexion entre une interface requise et une interface fournie. Il regroupe les spécifications des deux interfaces, sachant que ces spécifications ne peuvent référencer que les méthodes de ces interfaces. Ce contrat s'apparente au contrat objet. Les *contrats de composition externe* sont posés sur la face externe de la membrane d'un composant. Ils sont formés des spécifications qui ne citent que des interfaces externes du composant. Ils expriment donc les règles d'utilisation et de comportement externe du composant. Les *contrats de composition interne* sont posés sur la face interne de la membrane d'un composant composite. Ils sont formés des spécifications qui ne citent que des interfaces internes du composant et des interfaces externes de ses sous-composants. Ils expriment les règles d'assemblage et de comportement interne d'implémentation.
Au cours de la réification d'un contrat, le système de ConFract détermine les responsabilités associées à chaque spécification, parmi la liste des composants *participants* au contrat. Lorsque les responsabilités d'une spécification sont toutes déterminées, celle-ci devient une **disposition** du contrat (ou spécification fermée). Ces responsabilités correspondent aux notions de *garant*, l'unique composant qui doit être prévenu en cas de négation de la disposition et qui a la capacité d'agir pour traiter le problème, le ou les *bénéficiaires*, les composants qui peuvent compter sur la véracité de la disposition et qui peuvent être prévenus en cas de changement d'état de celle-ci (invalidation ou rétablissement), d'éventuels *contributeurs*, qui sont des composants qui participent à la véracité de la disposition.
Lorsqu'un composant est introduit dans un assemblage, ConFract crée les contrats relatifs à ce composant et détermine les différents contributeurs de chaque clause. Lorsque tous les contributeurs d'une clause sont connus, celle-ci est alors vérifiable. Inversement, lors de reconfigurations dynamiques impliquant des changements de composants, le système ConFract met automatiquement à jour la liste des contributeurs des contrats impactés, et crée de nouveaux contrats si nécessaire (nouveaux contrats d'interface sur les connexions nouvellement établies). Les différents contrats sont gérés par des *contrôleurs de contrats*, placés sur la membrane de chaque composant. Chaque contrôleur de contrats d'un composant composite prend en charge le cycle de vie et l'évaluation du contrat de composition interne du composite sur lequel il est placé, du contrat de composition externe de chacun des sous-composants, du contrat d'interface de chaque connexion située dans son contenu.
L'implémentation actuelle de ConFract repose sur l'implémentation de référence de Fractal, Julia. Le contrôleur de contrats utilise le mécanisme de mixin fourni pour interagir avec les autres contrôleurs standards du modèle Fractal (`BindingController` et `ContentController` pour la construction et la mise à jour des contrats, `LifecycleController` pour la vérification d'invariants sur des composants). Des intercepteurs sont aussi placés sur les interfaces des composants afin de vérifier les pré et postconditions.

Extensions en cours. A partir de cette réalisation, plusieurs extensions sont en cours de développement.
Un modèle général de négociation pour ConFract a été proposé afin de prendre en compte la fluctuation importante de certaines propriétés, typiquement extrafonctionnelles, qui peuvent entraîner des violations de contrats. Dans ce modèle de négociation, l'initiateur de la négociation — le contrôleur de contrats — consulte les composants impliqués selon leur responsabilité afin d'obtenir des propositions alternatives. Ce processus est piloté par des politiques de négociation, et une première politique dite par relâchement a été proposée [CC05]. Son principe consiste à se tourner vers les composants bénéficiaires pour qu'ils proposent des alternatives équivalentes au relâchement des assertions ou à des reconfigurations de paramètres et d'attributs. Une seconde politique, par effort, est en cours de développement. Cette politique se base sur le principe que l'initiateur de la négociation consulte, cette fois-ci, l'unique garant de la clause en échec. Puis, le processus de négociation consiste à consulter des com-

posants à un niveau de hiérarchie donné, et dans les cas favorables, à propager le processus de négociation aux niveaux inférieurs de la hiérarchie. Pour pouvoir réaliser cette politique, la difficulté réside dans la capacité à propager la négociation de clauses qui portent sur des propriétés extra-fonctionnelles en l'orientant vers de nouveaux participants dans la hiérarchie des composants. Pour ce faire, une solution consiste à s'appuyer sur des patrons d'intégration pré-établis pour modéliser les propriétés extra-fonctionnelles, ainsi que sur des expressions compositionnelles qui permettent d'expliciter la réalisation des propriétés et de lier leur définition au niveau d'un composite à sa composition de composants [CC06].
Le métamodèle sous-jacent à la première version de ConFract a été construit avec une orientation opérationnelle. Des travaux sont ainsi en cours pour développer un modèle plus général en établissant les abstractions nécessaires selon plusieurs axes. Différents formalismes de spécification doivent pouvoir être utilisés pour construire les contrats. Dans le cadre d'assertions exécutables, le modèle de spécification est le même que celui de l'exécution, mais plus le langage de spécification devient abstrait, et formel, plus cette distance avec le système en exécution est importante. Il est alors nécessaire d'associer les spécifications avec une forme de contrat et un cycle de vie appropriés. Les mécanismes nécessaires à l'intégration d'un formalisme comme TLA ont été proposés dans [COR06b, COR06a]. L'intégration est notamment effectuée à l'aide d'un DSL pour spécifier les moments de vérification et les méthodes à utiliser. D'autres formalismes doivent être encore intégrés pour améliorer et valider les mécanismes proposés. Enfin, il reste à concevoir et implémenter les abstractions relatives aux entités contractualisées, afin de rendre le modèle applicable à des composants hiérarchiques et des services.

3.2.2 Programmes abstraits

Les techniques de spécification par programme abstrait sont relativement proches des techniques assertionnelles. Un programme abstrait peut être proche d'un programme classique avec des possibilités d'indéterminisme. Dans [BS03], le langage AsmL est utilisé sur la plateforme .NET. Comme AAL [KMJ02], ce langage de spécification repose sur une sémantique séparée pour annoter le programme. AAL (Alloy Annotation Language) permet, quant à lui, de décrire le comportement par de la logique du premier ordre, des ensembles et des relations. Ces spécifications sont ensuite analysées statiquement. L'approche par programmes abstraits est simplement une forme de spécification par modèle, qui peut être plus abstraite et puissante que les spécifications basées sur le langage, mais aussi plus difficile à appréhender par le développeur.

3.2.3 Automates

Dans la partie précédente, nous avons vu que beaucoup de systèmes à base d'assertions étaient étendus par des approches plus formelles, avec des possibilités de vérification statique de certaines propriétés. Nous nous intéressons maintenant à des approches qui privilégient plus, voire entièrement, cet aspect de vérification. Cette partie s'intéresse aux approches à base d'automates appliquées aux composants logiciels.

Interface Automata

Le formalisme des automates d'interfaces proposé dans [dAH01] permet de décrire à la fois les prérequis sur l'ordre d'appel des méthodes sur une interface, et les garanties sur l'ordre avec lequel le composant appellera des méthodes externes. A partir de ces définitions, il est possible de vérifier, de façon entièrement automatique, qu'une interface est compatible avec

une autre. Les auteurs définissent leur approche de composition des automates comme optimiste, dans le sens où deux composants sont compatibles s'il existe un environnement qui permet de les faire coopérer. Plus classiquement, la compatibilité des interfaces est définie par une notion de raffinement : une interface en raffine une autre si elle a des prérequis plus larges et fournit des garanties plus fortes.

Coconut/J

Les EJB, profitant de la réflexivité de Java, sont le cadre de la mise en oeuvre de CoCoNut/J, un outil d'adaptation du comportement des interfaces [Reu03]. Dans ce cadre l'approche par contrat est complétée par une modélisation du comportement des interfaces par des automates qui sont l'objet effectif de l'adaptation. Chaque interface est décrite par deux types d'automates :
– l'automate d'appel, qui définit un sous-ensemble de toutes les séquences d'appel valides sur l'interface,
– l'automate de fonction qui définit, pour chacune des fonctions de l'interface, toutes les traces possibles d'appel à des services extérieurs à son composant porteur.
Ainsi quand un composant A s'adapte à un composant B, son automate d'appel restreint ses fonctionnalités à celles compatibles avec les automates de B. Le système peut adapter les fonctionnalités offertes d'un composant au moment de son insertion.

3.2.4 Algèbres de processus

Bien que les méthodes algébriques soient développées et utilisées depuis longtemps, le développement des systèmes répartis à mis en avant les approches à base d'algèbres de processus. Ces approches définissent un processus comme un système auquel est associé un comportement décrit sous forme d'un ensemble fini d'événements ou d'actions. L'algèbre de processus est alors une structure mathématique qui, de la même manière que les groupes en algèbre, définit un ensemble stable d'entités (les processus) pour des opérateurs de composition tels que le choix entre deux, la mise en séquence, etc. Cette définition pourrait s'appliquer aux automates, mais la notion d'interaction entraîne les algèbres de processus dans la théorie de la concurrence. L'opérateur de mise en parallèle de processus et la communication deviennent les concepts essentiels des algèbres de processus.
La majorité des travaux trouvent leur inspiration dans les *Communicating Sequential Processes* (CSP) [Hoa85], un langage dans lequel un processus est une abstraction mathématique des interactions entre un système et son environnement. Le comportement d'un processus peut être alors spécifié ou surveillé par la trace de la séquence de ses actions. Ces actions peuvent être des communications, éventuellement modélisées par des canaux, etc. Des propriétés comme la surêté (rien d'incorrect ne peut se dérouler) et la vivacité (quelque chose d'attendu surviendra forcément) peuvent être vérifiées. Dans le domaine des composants et des services, les *Finite State Processes* (FSP), inspirés des CSP, sont aussi utilisés tels quels q ou étendus. Les processus sont alors définis grâces à des machines à états finis étendues.

Wright

L'ADL Wright utilise les CSP pour spécifier le comportement d'interfaces [AG97]. Du point de vue des vérifications statiques, Wright autorise la vérification d'absence d'inter-blocage ainsi que d'autres tests de cohérence. Il permet notamment de vérifier la compatibilité des parties connectées sur la base de la compatibilité de leurs protocoles d'échange de messages. Ces tests sont réalisés à l'aide d'un outil de model-checking, pour lequel les outils de Wright produisent les entrées. L'ensemble des vérifications se font lors de la phase de conception de l'architecture.

Darwin

Darwin [MKG99] est un ADL qui base la sémantique de ses descriptions structurelles sur le pi-calcul. Dans cet ADL, chaque composant est considéré comme une entité atomique d'exécution. La composition hiérarchique de composants est présentée comme la mise en parallèle des processus qu'ils représentent. Plus généralement, une architecture Darwin se traduit en un programme de pi-calcul dont la validité est synonyme de celle de l'architecture. Ainsi Darwin base sa description architecturale sur l'échange de messages entre processus. Le comportement des composants est spécifié à l'aide d'une autre algèbre de processus nommé FSP (Finite State Processes) [Mag99] et de diagrammes de transitions d'état exprimés en LTS (Labelled Transition System) et associés aux messages des FSP. Le diagramme intervient comme un outil de visualisation des comportements globaux qui d'après l'expérience des auteurs occupe un rôle important dans la vérification et la recherche d'erreurs dans la conception. Des outils associés ont été développés pour la vérification de propriétés de sûreté et de vivacité lorsque des modèles sont composés.

SafArchie

Le langage utilisé dans SafArchie [BD04, Bar05] repose sur un système à transitions étiquetées utilisé pour décrire la sémantique du langage de spécification SFSP (Simple Finite State Processes) qui est fortement inspiré de FSP [Mag99]. La description LTS du système constitue la forme graphique du langage et SFSP sa forme algébrique. Ce langage définit le comportement externe d'un composant comme une séquence finie d'actions quelconques. Contrairement à FSP qui travaille à partir de séquences finies d'actions quelconques, SFSP travaille sur une description du comportement externe du composant définie à partir de quatre types de messages : les requêtes émises, les réponses à une requête, les requêtes reçues, les réceptions de réponses à des requêtes émises. Le nombre d'opérateurs est limité à la notion de préfixe (description d'une séquence linéaire de messages), du choix indéterministe (alternative au sein d'une description d'un comportement), de l'itération (répétition d'un comportement une ou plusieurs fois), la composition parallèle (assemblage de plusieurs composants), le camouflage (réduction d'une spécification comportementale vers un sous-alphabet) et le ré-étiquetage (synchronisation de messages ayant des noms différents). Ce langage est volontairement peu expressif de façon à obtenir des traces exploitables.

SOFA

SOFA [PV02] est un modèle de composants hiérarchiques où chaque composant présente classiquement un ensemble d'interfaces requises et fournies. La communication entre composants est capturée de manière formelle à l'aide de *behavior protocols*. Ceux-ci reposent sur le principe selon lequel chaque appel ou retour de méthode forme un événement. Le *behavior protocol* d'une entité SOFA consiste en l'ensemble des traces d'événement qui peuvent être produites par l'entité. Un langage à base d'expressions régulières permet de former des expressions dénotant ces traces, à l'aide d'opérateurs pour désigner un appel de méthode, l'acceptation d'un appel de méthode, l'émission d'un retour ou l'acceptation d'un retour de méthode. Ces événements peuvent être associés à une instance d'interface et peuvent être combinés à l'aide d'un opérateur d'alternative et un autre de séquence. La définition d'un composant est donnée par deux notions, *frame* et *architecture*. La *frame* est la vision en boîte noire du composant avec ses interfaces externes, l'*architecture* est une implémentation spécifique d'un composant à l'aide de sous-composants. Une *frame* peut ainsi être implémentée par plusieurs architectures.
On peut alors associer un *behavior protocol* aux interfaces et aux *frames*. Le compilateur SOFA génère alors l'*architecture protocol* à partir des *frame protocols* des sous-composants. L'intérêt

de cette approche est double. Un compilateur vérifie la cohérence des spécifications des différentes entités en interaction, par exemple en vérifiant qu'un *frame protocol* est conforme aux protocoles de toutes ses interfaces, et qu'un *architecture protocol* est conforme au protocole de la *frame* englobante. La vérification de l'adhérence des spécifications aux implémentations se fait quant à elle à l'exécution à l'aide d'automates simples.

Il est à noter que le formalisme des *behavior protocols* a été récemment adapté au modèle Fractal [KAB+06].

3.2.5 Diagrammes de séquences

Les langages de scénarios les plus largement utilisés sont les Message Sequence Charts (MSC) [ITU99], standardisés par l'ITU, et les diagrammes de séquences, proposés par l'OMG [OMG02]. Les versions récentes de ces langages, telles les MSC2000, permettent l'introduction de propriétés temporelles : Times, durées des messages, temps d'exécution absolus ou relatifs. Les langages de scénario sont très utilisés pour modéliser l'échange de messages entre objets communicants, les objets étant représentés par des axes verticaux et les échanges de message par des arcs orientés entre ces axes verticaux. Pour modéliser des comportements plus complexes, des opérateurs sur les scénarios sont définis : alternative, séquence, boucle, exception, etc. Une sémantique formelle est donnée à ces diagrammes des séquences, souvent basée sur les algèbres de processus. Ces langages sont très utilisés pour les systèmes de télécommunication [AE03]. Cependant, ces langages permettent surtout d'exprimer des scénarios d'usage, ils sont particulièrement adaptés pour décrire des comportements possibles et voulus d'un système (ou, par analogie, des comportements interdits). Une description complète d'un système à l'aide de scénarios nécessite un nombre très important de diagrammes de séquence.

Un langage particulièrement intéressant, et qui permet de résoudre le problème du nombre de diagrammes à construire, est les Live Sequence Charts (LSC) [DH01, HKW05]. Ceux-ci distinguent deux types de diagrammes de séquence, les diagrammes existentiels et les diagrammes universels. Les diagrammes existentiels correspondent aux diagrammes des langages de scénarios tels les MSC, ils exprimeront un comportement possible du système, voulus ou non voulus. Les diagrammes universels se composent de deux parties : un pre-chart et un post-chart. Le post-chart sera exécuté si et seulement si le pre-chart est exécuté. Le pre-chart pourra alors exprimer les conditions dynamiques qui doivent impliquer l'exécution du post-chart. Ces diagrammes expriment alors tout naturellement une relation de cause à effet entre parties du comportement d'un système. La sémantique des LSCs est donnée sous la forme de logique temporelle, et des outils permettent de valider le système construit. Cette validation peut se faire par simulation/animation, les diagrammes universels étant exécutés et, pendant cette exécution, les diagrammes existentiels analysent si le comportement qu'ils représentent se produit ou ne se produit pas. Un outil de model checking est aussi disponible [HKMP02]. Son objectif est de trouver, parmi tous les comportements possibles des diagrammes universels, un chemin qui satisfasse un diagramme existentiel. Un intérêt est d'analyser qu'une propriété, exprimée par un diagramme existentiel, ne peut jamais être satisfaite. Les versions récentes de ces diagrammes permettent d'introduire des contraintes temporelles [HM02], des probabilités, des variables, et des opérations telles les boucles, les décisions, etc. Appliqués à une architecture à base de composants, les diagrammes universels permettent de décrire le comportement entre interfaces d'un même composant, ainsi que les comportements sur les connections [CHK05]. Les diagrammes existentiels peuvent permettre de vérifier qu'une propriété (un contrat dynamique) n'est jamais violée.

3.2.6 Bilan

Les contrats comportementaux que nous avons présentés sont de formes diverses. Une première famille de systèmes contractuels s'organise autour d'assertions exécutables et d'une forme de contrats assimilables à ceux définis par Meyer dans Eiffel. Les propriétés exprimées sont le plus souvent évaluables à l'exécution afin de vérifier l'adhérence de l'implémentation aux spécifications. D'autre part, une seconde famille se caractérise par des formalismes plus abstraits avec des capacités de vérification formelles. On trouve beaucoup de variations autour des algèbres de processus, qui ont la qualité de fournir un bon compromis entre leur expressivité et les vérifications possibles par *model-checking*. Ces vérifications sont relatives à la cohérence même des spécifications (absence d'interblocage, etc.) ou à la compatibilité entre deux spécifications. Nous remarquons aussi que les formalismes utilisés sont complémentaires dans les aspects spécifiés : les assertions s'appuient sur des événements définis autour des appels de service, et les formalismes plus abstraits permettent notamment d'exprimer des séquences d'appel attendus.
De manière transverse, l'aspect contractuel des approches étudiés est lui aussi diversifié. La contractualisation d'une propriété consiste généralement à affecter la responsabilité de la garantie à une personne ou une entité logicielle. Chaque échec d'une vérification donnée est théoriquement interprétable en termes de responsabilité. Cette notion de responsabilité est exploitée : dans le cadre de la conception par contrats, elle permet d'éviter la programmation défensive et d'attribuer la responsabilité de l'échec à un développeur précis. Les responsabilités sont parfois partagées, comme dans le contrat de clientèle dans les systèmes à objets. Néanmoins, cette notion de responsabilité est, de manière générale, implicite ou exprimée très informellement, comme une forme de guide méthodologique.

3.3 Les contrats de qualité de services

Cette partie présente les langages et systèmes les plus significatifs pour la spécification et la vérification de propriétés extra-fonctionnelles, principalement de Qualité de Service (QoS) dans les systèmes à objets et à composants. Elle commence par décrire des langages de spécification de la qualité de service (partie 3.3.1), avec une focalisation sur la phase de conception. Elle détaille ensuite des systèmes de contrats de QoS pour objets (partie 3.3.2), puis pour composants (partie 3.3.3). Une étude de Qos-ODP (partie 3.3.4) termine cette partie.

3.3.1 Spécification de qualité de service

QML

QML (QoS Modeling Language) [FK98] est un langage de spécification de contraintes de qualité de services. Les qualités de service visées par QML sont très générales, regroupées en *catégories* comme la fiabilité, la sécurité et les performances. Chaque catégorie définit une ou plusieurs *dimensions*, avec une *unité*, qui représente une métrique d'un des aspects de la catégorie. Les métriques sont numériques pour les aspects quantitatifs (quantités, pourcentages, seuils, etc.) ou ensemblistes pour les aspects qualitatifs (ensembles de valeurs ordonnées). Des précautions sont prises pour pouvoir comparer des métriques et déterminer si une valeur est *préférable* à une autre (*stronger than*), afin de déterminer des compatibilités (*conformance*) ou de permettre des négociations.
L'approche est contractuelle et applique des principes de séparation des préoccupations. Les *contrats* sont des instances personnalisées de *types de contrats*. Le lien entre les contrats non fonctionnels et les aspects fonctionnels (interfaces) est établi dans une construction appelée *profile*. Dans une application répartie, les clients peuvent personnaliser des contrats non fonc-

tionnels *requis*. Les serveurs proposent des contrats *fournis* qui peuvent être différents, mais doivent être conformes, préférables. Par exemple, un délai d'attente fourni `delay < 10 ms` est préférable à un délai requis `delay < 20 ms`, donc est conforme. Inversement, un débit `throughput > 100 mb/sec` est préférable à un débit `throughput > 10 mb/sec`, donc est conforme. QML se distingue par sa logique de mise en oeuvre de métrique qui a pour intérêt de rendre comparable des spécifications de manière statique. Compatibilité et substituabilité des entités spécifiées peuvent ainsi être vérifiées avant exécution.
Le langage QML possède de nombreuses qualités (généralité d'utilisation, lisibilité, séparation des préoccupations, réutilisabilité), mais aussi des défauts :
– les dimensions sont déclarées dans les contrats, mais pourraient être avantageusement définies dans des classes séparées, afin d'être mieux réutilisables, et personnalisables sur leurs ordres de grandeur et d'unités ;
– les contraintes exprimables sont relativement pauvres et non paramétrées par des arguments ;
– les profils établissent un couplage lisible, mais imparfait entre les contraintes de qualité et les aspects fonctionnels.

CQML

CQML [yAA01] reprend des traits caractéristiques de QML en les développant. Il s'articule autour de quatre concepts : la métrique, le contrat, l'association avec les entités d'implémentation et l'utilisation de catégories pour organiser les trois précédents concepts d'entités.
Une caractéristique de QoS correspond au minimum à une grandeur dont le domaine est un intervalle ou un ensemble, possiblement ordonné ou une énumération. Comme QML, CQML propose la dérivation qui consiste à définir une grandeur en fonction d'une autre. Les dérivations fournies sont essentiellement statistiques et sont plus nombreuses que dans QML. Les caractéristiques acceptent des paramètres et peuvent inclure dans leur définition des invariants, exprimés à l'aide de prédicats OCL. De plus, les valeurs possibles d'une grandeur peuvent être exprimées par une formule OCL, fonction de paramètres de la caractéristique.
La composition de composants peut être prise en compte du point de vue de la QoS. CQML distingue la relation entre un composant utilisé par un autre et la relation entre plusieurs composants eux-mêmes utilisés par un autre. Dans le premier cas, la composition est modélisée par l'opérateur "sequential" indiquant la dépendance d'un composant par rapport à un autre. Dans le second cas, les composants sont indépendants, néanmoins, vis-à-vis de leur utilisateur, ils sont susceptibles de devoir être utilisés conjointement ou non.
Les expressions de qualité de service regroupent sous un nom une liste de contraintes sur des caractéristiques de qos, à la manière des contrats de QML. Le corps de chaque contrainte est une expression OCL. Sa validité peut être modulée dans le cas de contraintes statistiques qui requièrent de manière globale plusieurs mesures avant de pouvoir être décidées.
Les profils QML et CQML ont en commun le principe d'attacher un contrat à une entité d'implémentation. Le profil CQML définit les spécifications offertes (provides) et requises (uses) pour un composant (ou objet) donné. Néanmoins, la spécification est moins fine que dans QML, CQML ne descend pas jusqu'au niveau de la méthode. Mais comme pour QML, il est possible d'attacher plusieurs contrats à une même entité d'implémentation, permettant ainsi la séparation des préoccupations. A la manière de QuO avec ses régions de fonctionnement, CQML fait reposer les négociations de QoS en phase de configuration ainsi que la gestion dynamique de celle-ci, sur la fourniture, pour chaque profil, d'un ensemble de sous-profils imbriqués.
Pour la négociation, les profils sont comparables à l'aide d'une relation de conformité semblable à celle de QML, et sont ordonnés. Enfin, en phase d'exécution, la transition d'un profil à un autre peut être associée à des réactions du système (à travers des callbacks). Du point de

vue de la composition des composants, les auteurs expliquent que les caractéristiques de QoS d'un assemblage peuvent se déduire de celles de ses composants mais placent ces travaux en dehors du champs d'étude de CQML.

CQML+

CQML+ [RZ03] est une extension de CQML. Il définit précisément un métamodèle d'exécution (cf. fin du chapitre) dans le cadre duquel les spécifications s'expriment. Ainsi, tant que les expressions de QoS ne font intervenir que des grandeurs du métamodèle, elles sont statiquement vérifiables, alors que la mise en oeuvre de grandeurs extraites du système cible implique la nécessité de vérifications dynamiques. Cette extension offre aussi dans le langage une prise en charge de la QoS des ressources distinctes de celle des composants. En effet le système de composants ne gère pas concrètement les ressources systèmes, à la différence des ressources des composants. Le langage CQML est aussi étendu par des caractéristiques de QoS composites. Enfin CQML+ permet de définir des dépendances explicites entre les clauses uses et provides des profils. Dans certains cas, un profil peut accepter plusieurs régions de fonctionnement. Il est possible de les décrire à l'aide d'un ensemble de sous-profils présentant les caractéristiques uses et provides correspondantes. Mais il peut être plus efficace de relier les caractéristiques uses et provides du profil par une ou plusieurs équations.

3.3.2 Systèmes à objets

Rusken

Le système Rusken [LPJ98] est un navigateur d'espace virtuel qui utilise un *framework* basé sur l'approche contractuelle. La qualité de service est réifiée par des objets Java, instances de classes qui encapsulent le code de contrôle de qualité de service des composants du réseau. Dans une approche collaborative, les applications distribuées envoient leurs messages sous le contrôle d'objets-contrats. Ceux-ci sont équipés de mécanismes de surveillance (monitoring) des temps de réponse, de la détection des défaillances, de la consistance des données, etc.
La violation des clauses contractuelles sont identifiées et leur traitement est paramétré. Ainsi par exemple, une application peut choisir une alternative si un fournisseur de service est défaillant dans un contrat de temps de réponse. La description des éléments de qualité de service par objets conduit à une description beaucoup moins *ad hoc* que dans les approches usuelles. Elle permet des descriptions de plus haut niveau, avec de meilleures possibilités de réutilisation et de paramétrage. Des classes de contrats prédéfinis fournissent aussi des aides à la formulation de spécification de contraintes de temps imposées.

MAQS

MAQS [BG98] consiste en une architecture, indépendante du domaine d'application, de gestion de la QoS. Elle est également indépendante de l'environnement réparti bien que, pour la mise en œuvre du prototype, l'environnement CORBA a été retenu. Dans cette architecture, la QoS est vue comme une extension du concept de contrat entre un client et un serveur. Il s'agit en fait d'étendre une offre de service traditionnelle en y ajoutant des propriétés de QoS. Un service de courtage (trading) permet de localiser le service qui satisfait au mieux la QoS attendue. Pour pallier les changements qui entraîneraient une potentielle rupture du contrat de QoS, les applications doivent être adaptatives.
Pour décrire la QoS, MAQS utilise un IDL étendu appelé QIDL (QoS IDL) avec lequel il est possible de :
– décrire un ensemble d'attributs (correspondant aux offres de QoS). décrire une interface CORBA à laquelle on attribue une QoS.

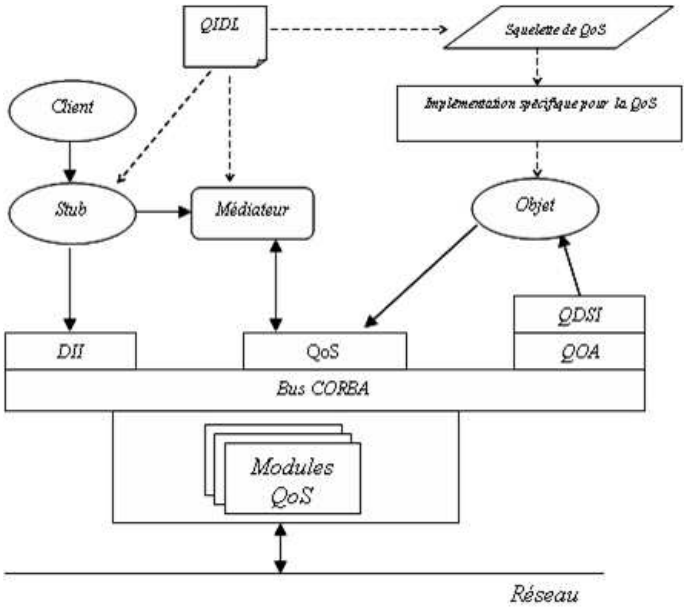

FIG. 3.1 – Le système MAQS

Cette description est ensuite compilée pour générer des amorces (souche et squelette) étendues pour CORBA. En plus des amorces, un autre élément est également généré : le médiateur. Le médiateur est utilisé par la souche lorsqu'une invocation est effectuée par le client. La souche délègue l'appel au médiateur qui va transmettre celui-ci en accord avec la QoS actuelle. Du côté serveur, toute invocation reçue soumise à la QoS est traitée par un adaptateur d'objet spécifique appelé QOA (QoS Object Adapter). Cet adaptateur délivre cette requête à l'implantation de l'objet via une interface DSI dédiée appelée QDSI (QoS DSI). Au niveau transport, MAQS introduit le concept de module de QoS. Un module permet la mise en œuvre de plusieurs moyens de transport qui peuvent être sélectionnés pour effectuer les invocations. Dans MAQS, le médiateur, le squelette ainsi que les modules doivent être développés par l'utilisateur par le biais d'un ensemble d'interfaces. Le médiateur est un élément qui est chargé dynamiquement lors de l'invocation et qui pourra donc être modifié dans l'avenir pour mieux satisfaire la négociation. En cas de changement de QoS, une notification est envoyée au client par le serveur qui peut alors modifier, par une approche " best effort ", ses besoins.

QuO

QuO [LSZB98, HLS04] est une architecture permettant l'inclusion de la QoS dans les applications réparties. Cette architecture est essentiellement fondée sur trois éléments :
- un délégué qui est utilisé du côté client. Il joue le rôle de souche, c'est-à-dire que c'est lui qui émet l'invocation distante. Cette invocation s'établit après vérification du contrat de QoS avant et après appel.
- un contrat de QoS entre le client et le serveur. C'est par l'intermédiaire de ce contrat qu'il est possible de spécifier les actions à engager quand le niveau de QoS change.
- un ensemble d'objets de condition système (CS). Il s'agit d'une interface entre le contrat et les ressources, les mécanismes, les objets et l'environnement réparti. Ces conditions système sont utilisées pour effectuer des mesures et du contrôle.

Lorsqu'une application cliente effectue une invocation, celle-ci est transmise au délégué. Ceci est totalement transparent pour l'utilisateur. Le délégué va alors vérifier le contrat de QoS en cours, ce qui nécessite de prendre éventuellement en compte les conditions système. Le

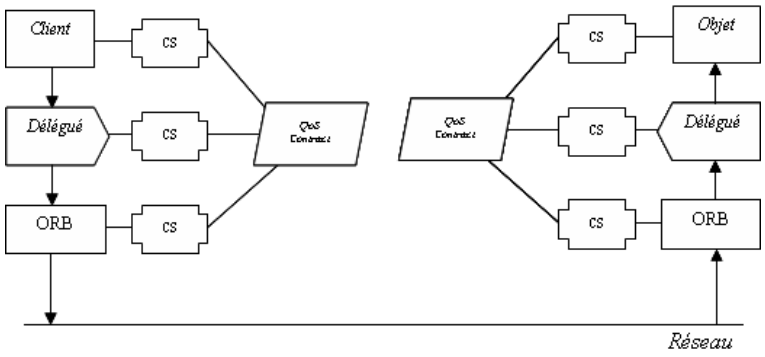

FIG. 3.2 – L'infrastructure QuO

contrat est un ensemble de régions de QoS où chaque région correspond à un état de QoS. Les changements d'état sont régis par un ensemble de règles de transitions et de prédicats. En prenant en compte les conditions système, le contrat détermine les régions actives qu'il transmet alors au délégué. A l'aide de ces informations, le délégué va choisir comment effectuer l'appel distant. Par exemple, le délégué peut choisir entre plusieurs méthodes alternatives (bloquer ou mettre dans un tampon) lorsque la QoS se dégrade. Au sein de QuO, il est également défini un langage de description appelé CDL (Contract Description Language). Ce langage est proche d'un langage de programmation comme C++ ou Java. Il permet de définir les régions de QoS, les prédicats, transitions et notifications vers le client en fonction de conditions système. Cette description est ensuite compilée pour générer le contrat de QoS. Dans cette approche, il n'y a pas d'établissement dynamique du contrat, il s'agit d'une approche statique qui identifie une relation entre deux entités.

JAMUS

JAMUS [LSG02] (Java Accommodation of Mobile Untrusted Software) est une plate-forme dédiée à l'hébergement de composants mobiles, capables d'exprimer leurs besoins propres en ressources. Chaque composant doit décrire ses besoins en ressources, de manière qualitative (e.g. des droits d'accès à des fichiers) ou quantitative (e.g. quantité d'espace mémoire disque ou centrale). La confrontation des ressources demandées et disponibles se fait de manière dynamique, au moment de l'admission d'un composant. Ensuite, la plate-forme surveille que l'utilisation effective des ressources est conforme aux demandes formulées et empêche tout débordement. Cette plate-forme est construite au dessus de l'environnement RAJE, une extension de JVM qui réifie certaines ressources système (CPU, mémoire, interface réseau, etc.). L'approche est contractuelle. Les composants s'engagent à ne pas consommer plus de ressources qu'ils n'ont déclaré et la plate-forme est tenue de les fournir, dès que l'admission est effective. La violation du contrat n'est pas équitable : la plate-forme peut sanctionner, mais pas le composant. Les *deadlocks* sont évités par la demande globale des ressources nécessaires, au moment de l'admission. Les contrats sont exprimés par des chaînes de caractères, dans un formalisme *ad hoc*, analysé dynamiquement par la plate-forme.

QMF

QMF [DTV99] définit une architecture logicielle respectant quatre propriétés fondamentales pour la gestion de la QoS : observation, négociation, composition et garantie. Cette architecture est générique puisqu'elle est applicable quel que soit le domaine et adaptable à n'importe quel environnement réparti. Lorsqu'une application souhaite entrer dans l'infrastructure de gestion de QoS, elle démarre par une phase de vérification de la compatibilité de ses besoins

vis-à-vis des exigences des applications évoluant déjà au sein de l'infrastructure. Il faut noter qu'en cas d'influence compatible c'est à dire d'une influence sur le comportement d'une application existante mais qui n'entraînerait pas de changement immédiat de son offre de QoS, l'infrastructure doit tout de même noter cette incidence au cas où celle-ci pourrait entraîner ultérieurement la rupture d'un contrat de QoS. Une fois la vérification effectuée, l'application peut alors exprimer son offre qui pourra, par la suite, être sollicitée par une application cliente. Une offre de QoS est une liste de propriétés qui peuvent être de plusieurs natures (de types simples ou complexes) mais dont la valeur peut également être qualifiée soit de constante, variable ou dépendante. Une valeur dépendante est une valeur qui n'est pas directement évaluable parce qu'elle dépend d'éléments extérieurs, soit des applications avec lesquelles l'application collabore, soit des ressources du système (mémoire ou débit réseau, par exemple).

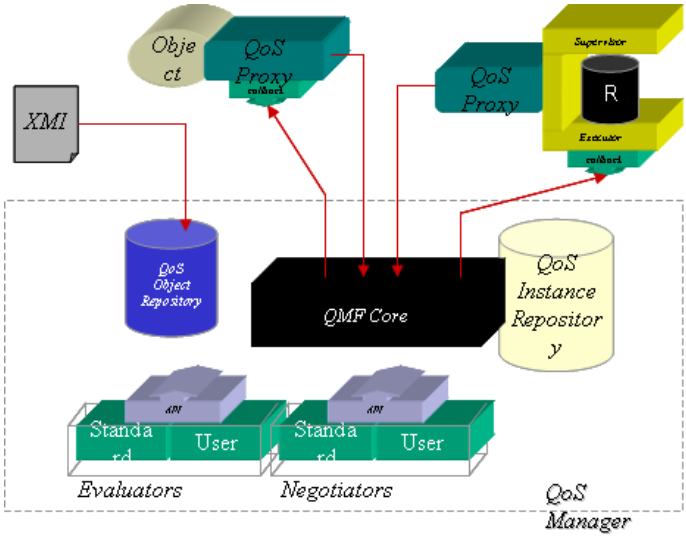

FIG. 3.3 – L'infrastructure QMF

L'évaluation de propriétés de QoS n'est donc pas un mécanisme systématique et s'appuie sur un " évaluateur " qui est un élément sollicité par l'infrastructure pour obtenir la valeur d'une propriété d'une offre de QoS. Un évaluateur standard est directement disponible au sein de la plate-forme mais l'utilisateur a également la possibilité de développer son propre évaluateur (pour des cas très spécifiques) en respectant une interface de connexion entre l'infrastructure de gestion de QoS et l'évaluateur lui-même. Lorsqu'un client souhaite établir une collaboration avec un serveur en fournissant ses besoins de QoS (sous forme de contraintes OCL sur les offres des objets à utiliser), celui-ci doit s'adresser à l'infra-structure qui va entrer dans une phase de sélection consistant à étudier un à un les objets serveurs potentiellement utilisables, c'est à dire ceux qui offrent le service demandé. L'infrastructure va alors appliquer la contrainte sur chaque objet pour vérifier si celui-ci remplit les exigences demandées. Ce processus est récursif lorsque l'offre d'un objet dépend elle-même d'un autre objet.
Au fur et à mesure de l'inscription des serveurs au sein de l'infrastructure, des communautés d'objets (objets reliés par des liens de QoS) sont répertoriées au niveau du gestionnaire de QoS. Le processus de composition consiste à partir d'un objet et à parcourir un à un tous les liens pour obtenir une offre de QoS globale qui pourra, par la suite, être soumise à la contrainte exprimée par le serveur. Il s'agit donc de parcourir les divers liens formant une communauté. Si une contrainte ne peut pas être respectée alors la demande du client échoue. Ce comportement par défaut implique, pour le client, le besoin de redéfinir ses besoins en diminuant certaines exigences puis à relancer le processus de sélection. Cette façon de procéder forme un mécanisme simple de négociation. Ce principe de négociation est celui fourni par

le " négociateur standard " qui est appelé par l'infrastructure pour mettre en œuvre la sélection d'un objet au sein des communautés. L'infrastructure permet également à l'utilisateur de fournir ses propres négociateurs en respectant une interface définie (sur le même principe que les évaluateurs). Lorsqu'une contrainte cliente peut être satisfaite, le contrat de QoS est scellé. Les divers participants au contrat sont alors informés de cet événement par l'intermédiaire d'une notification. De même, l'infrastructure note ce contrat et active un élément particulier appelé "exécuteur de QoS". Chaque objet et client peut enregistrer au sein de l'infrastructure un exécuteur de QoS (il s'agit d'un élément qui respecte là encore une interface spécifique) et qui sera appelé lorsqu'un contrat de QoS, mettant en jeu l'objet qui l'a enregistré, est scellé.
Lorsque le contrat est effectif, l'infrastructure de gestion de QoS doit régulièrement le vérifier c'est à dire observer les offres et vérifier si celles-ci respectent toujours la contrainte exprimée par le client. Cette observation est périodique et paramétrable. De plus, lorsqu'un objet fait évoluer son offre de QoS, il doit en informer l'infrastructure de gestion de QoS. Cette évolution va entraîner la vérification des contrats en cours (impliqués par cette modification) mais également tous les liens d'incidence notés lors de l'inscription d'un objet. Si un contrat est rompu, l'infrastructure fait appel au négociation (standard ou utilisateur) pour prendre une décision. Le négociateur standard va alors rechercher parmi les autres objets ceux qui seraient susceptibles de fournir un contrat valide (à condition que le client ait autorisé le changement transparent de contrat). Si aucun autre objet n'est capable de fournir le contrat souhaité, une notification est envoyée à l'initiateur du contrat.

3.3.3 Systèmes à composants

QCCS

QCCS (*Quality Controlled Component-based Software development*) [SAD+02] a été un projet européen IST qui a proposé une méthodologie et des outils pour la création de composants à l'aide de contrats et d'aspects. Sa méthodologie repose sur des extensions d'UML 1.4, particulièrement pour des éléments extra-fonctionnels incorporés dans le métamodèle d'UML. Les composants sont contractualisés par leurs interfaces, auxquelles sont associés des contrats fonctionnels (en OCL), et de contrats extra-fonctionnels exprimés en QML (cf. partie 3.3.1). Les contrats sont ensuite intégrés dans le code produit par des techniques de tissage.
Le métamodèle de QCSS est discuté dans la partie 3.6.2.

QosCL

QoSCL (QoS Constraint Language) est un modèle de contrat de QoS qui permet de modéliser des contrats de QoS et leurs dépendances en se reponsant sur UML 2.0. Son objectif est de pouvoir explicitement lier les contrats de QoS entre des interfaces requises et fournies. Les contrats exprimés en QoSCL [JDP03, DJP04] sont ainsi utilisés de deux manières :
– pour valider les composants individuellement, en tissant automatiquement le code de monitoring nécessaire dans les composants, à la manière de QCCS (cf. partie précédente) ;
– pour valider l'assemblage de composants, en inférant des contraintes de bout en bout à partir des contrats de chaque composant.

Cette dernière forme de validation exploite des techniques de résolution de contraintes pour déterminer des domaines de valeur acceptables sur des contraintes.
Le métamodèle de QoSCL est discuté dans la partie 3.6.2.

DRACO/CRuMB

Le système DRACO [BRBV04] est un *middleware* de surveillance de composants contractualisés pour systèmes embarqués. Il est associé à un outil de conception de composants et de

contrats extra fonctionnels, le tout formant une méthodologie pour développer des systèmes embarqués à base de composants. Ces composants sont des boîtes noires, qui sont explicitement connectées à travers des ports et qui communiquent par envoi de messages asynchrones. Chaque port est ainsi un canal de communication bidirectionnelle. Sur chaque port, il est possible de spécifier des contraintes de type et de synchronisation (ordre dans lequel les messages doivent apparaître). Des schémas de composant sont manipulés lors de la conception, alors que le système DRACO ne s'occupe que des instances résultantes à l'exécution.
Les contrats spécifient des aspects extra-fonctionnels généraux, mais réduits actuellement à des contraintes de temps et de bande passante (flux échangé) sur les communications entre composants. Ces contrats annotent les schémas de conception et sont parfois vérifiés lors de la conception ou par le système DRACO à l'exécution. En effet, la gestion des composants et de leurs interconnexions permet au système DRACO de piloter la surveillance des ces deux formes de contrats.
Associé au système DRACO, le système CRuMB (*Contract-based Resource Monitor and Broke*) s'occupe de la gestion contractualisée des ressources, en fournissant de quoi réifier les contrats et les négocier avec un système de règles.

3.3.4 QoS et ODP

Le but du modèle de référence ODP [ISO95] est d'offrir un cadre conceptuel pour spécifier une architecture de systèmes répartis ouverts. Dans cette partie, nous présentons des extensions génériques proposées à ce modèle pour le traitement de la Qualité de Service (QoS). Ces extensions n'ont pas pour objectif de définir ce qu'est la QoS ou ses dimensions (ponctualité, volumétrie, etc.). Elles visent à permettre de (i) spécifier et manipuler la QoS de façon modulaire dans une architecture ODP et de (ii) formaliser des garanties de QoS par composition de spécifications contractuelles.
Les concepts introduits ici sont développés et formalisés dans [LN97, Leb98] sur la base du modèle sémantique d'Abadi et Lamport [AL90], qui permet de raisonner sur les notions de contrat et de leur composition. Ils s'appliquent aux spécifications contractuelles du type *assume/guarantee*, qui permettent de séparer ce qui relève de la responsabilité d'une entité de celle de son environnement, et sont illustrés dans un contexte de contrôle d'admission réparti. Ils sont également utilisés dans [yA01] comme base formelle du langage CQML (Component Quality Modelling Language).

Quelques propriétés sont d'abord posées avec pour objectif de guider la construction d'un cadre architectural pour la QoS.
- **Propriété 1 :** Les énoncés de QoS doivent être suffisamment **modulaires** pour pouvoir être attachés à des objets individuels. Il devrait être possible de déduire la QoS attachée à une **composition** d'objets de la QoS attachée aux objets composants.
- **Propriété 2 :** la QoS doit pouvoir être **garantie** à certaines périodes durant la vie du système.
- **Propriété 3 :** le niveau de QoS atteint doit être **observable** de manière à permettre le développement d'applications de supervision ou de boucles de rétroaction.
- **Propriété 4 :** la QoS doit être **négociable**, dans le sens où pendant la vie du système, des entités peuvent quitter l'application alors que d'autres peuvent arriver avec des besoins différents. Le cadre doit être suffisamment flexible pour permettre d'implanter des stratégies de négociation de la QoS telles qu'une dégradation honorable.

Les définitions suivantes sont ensuite proposées comme des extensions aux "fondations" du modèle ODP. Autrement dit, il s'agit d'extensions génériques, indépendantes des points de vue ODP, qui s'appliquent au modèle objet primitif.

Relation de QoS. Une relation de QoS est une spécification contractuelle traduisant un engagement mutuel entre un objet et son environnement. Il s'agit d'une notion de base pour exprimer un énoncé de QoS ou un contrat négocié. Plus précisément, une relation de QoS s'appliquant à un objet A peut s'écrire sous la forme :

$$Exp(A) \multimap Obl(A)$$

où :
- $Obl(A)$ est une propriété fournie par A à son environnement. Celle-ci ne peut contraindre que les comportements de A (signaux sortant ou action interne modifiant l'état de A).
- $Exp(A)$ est une propriété requise par A de son environnement. Celle-ci ne peut contraindre que les comportements de l'environnement de A (signaux entrant ou action externe modifiant l'état de l'environnement de A).
- $\multimap$ est une forme d'implication logique qui exprime ici le fait que $Obl(A)$ ne peut devenir fausse avant $Exp(A)$. Les rôles sont donc bien séparés, l'objet A s'engage à satisfaire $Obl(A)$ tant que les objets constituant son environnement satisfont $Exp(A)$.

Dans un contexte de QoS, $Obl(A)$ et $Exp(A)$ sont généralement des propriétés de type sûreté (i.e. réfutables en un temps fini). Par exemple : tel trafic utilise au plus X% de la bande passante, la gigue de tel signal périodique est inférieure à Yms, ou des propriétés comportementales plus complexes. A noter qu'une relation de QoS peut représenter une pure hypothèse si $Obl(A) = TRUE$ ou une pure obligation si $Exp(A) = TRUE$.

Énoncés de QoS. Afin de décrire la QoS de systèmes répartis ouverts les énoncés suivants sont ensuite proposés. Ils s'expriment en utilisant la structure de relation de QoS introduite ci-dessus et peuvent être attachés aux objets durant la conception d'un système ou durant son exécution (dans le cas où la QoS est gérée dynamiquement).
- **Offre de QoS.** Cet énoncé permet de publier l'offre de QoS qu'un objet d'un système propose aux autres objets de ce système. Par exemple, un objet peut s'engager à fournir une certaine température tant qu'il reçoit une énergie suffisante (offre notée $(e > E1) \multimap (t > T1)$) et avoir une forme de rétroaction basique avec un autre objet qui s'engage à fournir une certaine énergie tant que la température est suffisante (offre notée $(t > T2) \multimap (e > E2)$).
- **Exigence de QoS.** Cet énoncé permet de contraindre la QoS d'un système. Ainsi, un utilisateur (représenté par un artefact) peut contraindre le système précédent en exprimant qu'il est satisfait tant que la température dépasse un seuil (exigence notée $(t > T3) \multimap TRUE$).
- **Contrat de QoS.** Cet énoncé est le résultat d'un accord. Un contrat de QoS définit l'ensemble des offres et des exigences de QoS que l'ensemble des objets d'une configuration se sont mis d'accord à fournir aux autres. Il s'exprime en utilisant une relation de QoS et peut résulter d'un mécanisme de négociation dynamique, ou être incorporés implicitement dans la phase de conception d'un système.

Dans un contexte de négociation dynamique, un objet peut disposer de plusieurs niveaux d'exigences ou d'offres de QoS. Dans le cas où les énoncés de QoS s'appliquent à une configuration d'objets, un mécanisme de raffinement pourra être utilisé pour obtenir l'ensemble des énoncés de QoS qui s'appliquent aux objets individuels de la configuration.

Composition. La QoS doit pouvoir se composer. Autrement dit, il doit être possible de prouver les caractéristiques de QoS d'un système à partir des caractéristiques de ses composants. Plusieurs travaux établissent des résultats de composabilité basés sur des spécifications contractuelles de type *assume/guarantee*. Un résultat de Abadi et Lamport [AL90, AL95] est ainsi utilisé dans [Leb98] pour montrer comment des énoncés de QoS modulaires, attachés à des objets plongés dans un environnement contraint, peuvent se composer.

Ce résultat est intéressant car il permet d'exprimer l'accord à vérifier pour garantir un contrat de QoS global entre le système composite résultant et son environnement. Il s'applique au système précédent (illustrant une pseudo interaction sur l'énergie) ainsi qu'à des systèmes plus complexes (eg. contrôle d'admission de flux multimédia, QoS de bout en bout). Il faut noter cependant qu'il ne s'applique pas à n'importe quelle type de composition mais à une catégorie importante, nommée composition conjonctive, où les objets sont composés par intersection de leurs spécifications contractuelles, et où les propriétés de QoS sont de type sûreté (i.e. réfutable en un temps fini).

Ressources. Les notions précédentes sont génériques. Leur utilisation est discutée dans le cadre du point de vue traitement du modèle ODP - qui fournit un modèle de programmation pour des entités réparties. Un modèle de ressources est notamment introduit comme un raffinement de ce point de vue. La motivation étant que la QoS n'est pas magique et que les ressources (CPU, réseau, etc.) doivent être réifiées pour pouvoir être traitées de façon contractuelle. Le modèle de ressources permet d'expliciter les capacités réelles de la plate-forme sous-jacente sous forme de spécifications contractuelles, faisant apparaître des tests d'admission dans leurs hypothèses, et pouvant être composées avec les offres et exigences d'objets applicatifs.

Négociation. Dans un contexte de systèmes répartis ouverts, la QoS doit pouvoir être négociée en cours d'exécution. Un processus de négociation est vu dans [Leb98] comme portant sur les objets d'une zone de négociation, où :

Une **zone de négociation** est une collection d'objets isolée en ce qui concerne la QoS. L'isolation signifie que les hypothèses de ces objets vis-à-vis de leur environnement ne s'appliquent qu'aux objets de la collection.

Le processus de négociation prend en entrée les offres et exigences de QoS correspondant aux objets de la zone de négociation. Il retourne un contrat de QoS consensuel décidé par la négociation. Un processus de négociation peut avoir lieu à la fois pendant la conception du système et pendant l'exécution.
L'auteur note que cette approche permet d'unifier des processus de négociation, tel un contrôle d'admission (vu comme une forme de négociation élémentaire de type oui/non) ou une dégradation honorable, en les ramenant à un problème d'accord sur un contrat de QoS global dans une zone de négociation. Les parties prenantes dans cette zone devant se mettre d'accord sur une collection d'offres et d'exigences de QoS. L'approche est illustrée dans un contexte de contrôle d'admission réparti, mais l'utilisation de stratégies plus évoluées permettant de trouver une issue positive à la négociation (eg. par concessions sur les offres) n'est pas développée.

3.3.5 Bilan

L'étude des approches pour contractualiser la QoS permet de distinguer les mêmes éléments discriminants que pour les contrats comportementaux. On trouve en effet des approches qui permettent d'exprimer la QoS sur des objets et des composants, et de vérifier, avant exécution, la compatibilité de différentes spécifications entre elles ou l'absence d'incohérence. D'autre part, d'autres systèmes s'attachent principalement à surveiller le système en cours d'exécution. La contractualisation se fait sur différentes notions de QoS et sur des propriétés extra-fonctionnelles. La vérification permet alors d'assurer la conformité du système gérant ou fournissant la QoS avec les spécifications attendues.

3.4 Les contrats d'architecture

Cette partie présente différents travaux qui se focalisent sur la contractualisation des architectures logicielles.

3.4.1 K-Component

Dans le modèle des K-Components [DC01], la configuration d'un système est réifiée en un graphe, manipulé et transformé par des programmes réflexifs nommés *contrats d'adaptation*. Le graphe de configuration est déduit automatiquement d'un système construit de manière programmatique. La reconfiguration est transactionnelle, et gèle l'exécution et l'état des composants. Les contrats sont décrits dans un langage d'adaptation spécifique, différent du langage de programmation des composants. Ils sont chargés/déchargés dynamiquement (en dehors des phases de configuration du système) par un "configuration manager" situé avec eux dans un métaniveau.
Chaque contrat contient un ensemble de règles spécifiant des contraintes architecturales et les opérations de reconfigurations à entreprendre en cas de violation. Ces contraintes sont évaluées à la perception d'événements de reconfiguration du système, produisant ainsi un découplage entre niveau de base et méta et une possibilité de mise en œuvre dynamique des contrats. Les règles dans les contrats d'adaptation sont interprétées comme une convention entre les participants (des composants) du contrat. Deux types de contrats d'adaptation sont distingués :
– de configuration, qui spécifie des contraintes architecturales entre composants d'un même système (techniquement limité à un espace d'adressage) ;
– de connexion, qui spécifie des contraintes sur une connexion individuelle entre le système et un composant dont il dépend et qui lui est extérieur. Techniquement, il peut s'agir d'un composant dans un autre espace d'adressage.
Le métamodèle des K-Components fournit le graphe de configuration, les protocoles et actions de reconfiguration de base agissant sur le graphe, et la gestion de la reconfiguration dynamique intégrée au sein des contrats, situés au métaniveau. Les contrats apparaissent donc comme des entités de premier ordre, non pas au niveau de l'application mais au niveau méta. Par ailleurs une entité, le Configuration Manager gère le couplage entre le cycle de vie des contrats et celui de l'application. Enfin les contrats s'appuient sur une vue de l'application au travers de son graphe de configuration, et suivent son activité au travers d'événements.

3.4.2 SATIN

SATIN [OPD04] est un service de sûreté que les plates-formes à composants peuvent interroger afin de déterminer si les adaptations dynamiques à mettre en oeuvre sont sûres. Ce service est construit à partir d'un modèle d'adaptation indépendant de toute plate-forme. Pour cela la notion d'adaptation est modélisée en UML : le sous-ensemble d'adaptations prises en compte inclut la modification comportementale, l'ajout de fonctionnalités, ou la création/modification d'assemblages de composants. Satin définit une sorte de contrat pour l'adaptation correspondant à un ensemble des propriétés que les adaptations doivent vérifiées pour préserver la sûreté de fonctionnement de l'application. Ces propriétés dites de sûreté sont listées ci-après. Notons que suivant les caractéristiques de la plate-forme, un sous-ensemble de propriétés à vérifier peut être choisi.
– P0 : Conservation du contexte d'utilisation. Un composant de remplacement doit pouvoir être utilisé de la même manière que le composant qui a été remplacé.
– P1 : *Consommation* des messages. Cette propriété permet d'éviter les erreurs dues à des

appels à des fonctionnalités inconnues. Entre autres, les fonctionnalités initiales d'un composant ne doivent pas être supprimées suite à une adaptation.
– P2 : Garantie de visibilité. Les fonctionnalités utilisées dans le cadre d'une adaptation doivent être visibles (depuis l'interface à laquelle l'adaptation a accès).
– P3 : Consistance des assemblages. Ce qui est utilisé par un composant dans le cadre d'une adaptation doit toujours être fourni afin qu'il n'y ait pas de composant requis manquant ou de type incorrect au sein d'un assemblage.
– P4 : Déterminisme du comportement. Cette propriété garantie la cohérence des adaptations successives d'un composant. Premièrement, l'application des adaptations doit être uniforme pour éviter que deux appels à une même fonctionnalité dans le même contexte (même paramètres, même état interne, mêmes ressources, mêmes adaptations, .) conduisent à deux résultats différents. D'autre part, les adaptations ne doivent pas engendrer de conflit de composition. Enfin, il ne doit pas y avoir de point de non-déterminisme à l'intérieur d'un assemblage de composants c'est à dire des points dans le flot d'exécution ou des fonctionnalités peuvent potentiellement être appelées dans n'importe quel ordre sur le même composant.
– P5 : Cycles. Il ne doit pas y avoir de cycle malin à l'intérieur d'un assemblage de composants c'est à dire de boucle infinie dans un flot d'exécution de méthodes.
– P6 : Retro-activité des adaptations. L'ajout, ultérieur à l'application d'une adaptation dont la coupe est non close, de fonctionnalités correspondant à la coupe de l'adaptation implique que celle-ci soit applicable à ces nouvelles fonctionnalités

L'approche adoptée pour formaliser ces propriétés de sûreté (P0 à P6) consiste à décrire des contraintes en OCL [Obj97] pour assurer ces propriétés. A titre d'exemple, les contraintes associées à la propriété P3 correspondent en langage naturel à vérifier que :

1. pour pouvoir appliquer une adaptation à un ensemble de composants, il faut que chaque composant impliqué remplisse le rôle attendu par l'adaptation (offre les fonctionnalités requises pour l'assemblage, par exemple).
2. une adaptation d'ajout de fonctionnalités sur un composant ne peut être défaite tant qu'il existe d'autres adaptations impliquant l'utilisation des fonctionnalités ajoutées du composant.

Le modèle de SATIN a été validé en prouvant la complétude des contraintes vis-à-vis de la propriété de sûreté à laquelle elles sont associées. La technique de validation par simulation consiste à construire explicitement des *snapshots* représentant des états d'un système. Deux étapes sont nécessaires pour prouver qu'un ensemble de contraintes n'est pas consistant par rapport à une propriété donnée. Premièrement, il faut vérifier que les contraintes ne soient pas trop faibles (aucun état indésirable ne doit être accepté par les contraintes). Deuxièmement, il faut vérifier que les contraintes ne soient pas trop fortes (aucun état valide ne doit être rejeté par les contraintes). Pour déterminer la non consistance d'un ensemble de contraintes vis-à-vis d'une propriété, il suffit de trouver au moins un contre-exemple. La validation par simulation est une forme de prototypage permettant d'assurer rapidement un certain degré de confiance dans la spécification et ceci à moindre coût puisqu'il n'est pas nécessaire d'implémenter le modèle et les contraintes. Cependant l'exactitude de la spécification est garantie seulement vis-à-vis des états analysés, Aussi la validation de la modélisation est complétée en utilisant la technique de preuve par théorème avec B [Abr96] pour vérifier en particulier la consistance, c'est à dire qu'il n'y ait aucune contradiction entre les contraintes OCL décrites.

3.4.3 Retour sur SafArchie

Le premier objectif des contrats dans SafArchie [BD04, Bar05] est de garantir la cohérence des assemblages et de détecter des incompatibilités entre composants assemblés. La première

vérification concerne le respect par l'architecture définie du méta-modèle de SafArchie. Cette base, c'est-à-dire le modèle structurel, est ensuite enrichie par des contrats. Dans SafArchie, un contrat est un moyen d'établir d'une façon explicite et non ambigüe les obligations et les droit d'un composant vis-à-vis de son environnement. Les obligations définissent une ou des contraintes garantissant le résultat du service fourni si et seulement si certains droits sont remplis.
Pour préciser les éléments de structure d'une architecture, deux contrats structurels sont définis : le contrat de port et le contrat d'assertion. Le contrat de port précise les conditions d'utilisation d'un port, c'est-à-dire les dépendances entre les opérations d'un port d'un composant. Il précise les opérations devant nécessairement être présentes pour le bon déroulement de l'interaction. C'est une expression logique évaluée au moment de la composition. Deux opérateurs sont définis : le + qui permet l'association de différentes opérations devant participer à l'interaction et le | qui autorise la définition de plusieurs sous-ensembles d'opérations admissibles Le contrat d'assertion définit, quant à lui, les types des paramètres des opérations. Ces contrats permettent d'améliorer la réutilisation des composants en précisant les types des paramètres des opérations. Ces contrats reprennent partiellement les résultats de Meyer. Il consiste à ajouter des contraintes sur les paramètres des opérations, les types de retour des opérations ou les attributs des composants. Dans SafArchie, ce langage est défini à partir d'OCL, qui a été adapté à la description des architectures.
Pour compléter ces contrats structurels, la notion de contrat comportemental est aussi introduite. En effet, une architecture est un système dit parallèle, c'est-à-dire composé de plusieurs processus pouvant s'exécuter simultanément et s'échangeant des informations. Une hypothèse asynchrone est placée, car le système ne possède pas d'horloge globale. Le comportement d'un composant primitif est alors décrit en termes de traces de messages qu'il échange avec son environnement. Pour ce faire, il est utilisé un langage algébrique permettant de décrire le comportement de chaque composant à l'aide d'un système à transitions étiquetées. Ce contrat décrit les échanges de messages émis et reçus par les instances de composant en fonction des enchaînements des opérations.
Deux remarques à cette association de contrat de comportement aux composants dans SafArchie. La première est que le langage est volontairement peu expressif. Il ne prend pas en compte les valeurs des paramètres des messages échangés ou les conditions associées aux choix internes au composants Ce manque d'expressivité permet d'obtenir une trace d'exécution compréhensible et utilisable par l'humain. La seconde est que le comportement n'est pas décrit au niveau des ports, mais au niveau du composant dans sa globalité, c'est-à-dire de façon à faire apparaître le comportement complet et de faciliter la réutilisation de celui-ci.

3.4.4 Retour sur ConFract

Dans le système ConFract (cf. partie 3.2.1), il est possible de définir des invariants de configuration sur un composant. Les assertions ainsi définies expriment des conditions indépendantes de toute exécution pour le composant. Les expressions peuvent référencer des attributs, mais aussi la description du composant Fractal par introspection (nom, type, cardinalité, présence des interfaces, etc.). Placées dans un contrat de composition interne, il est alors possible d'accéder de la même manière aux sous-composants d'un composite et de décrire ainsi des contrats d'architecture pour un niveau donné de la hiérarchie.
Comme ils ne référencent que des éléments de l'architecture, ces contrats sont vérifiables avant l'exécution. Dans ConFract, ils sont vérifiées juste avant que le composant Fractal soit démarré à l'aide son contrôleur de cycle de vie. Il est aussi à noter que ces contrat sont aussi toujours à jour vis-à-vis de l'architecture, car les contrats en ConFract sont mis à jour en fonction des reconfigurations dynamiques. Ainsi, lors d'un cycle de reconfiguration, les contrats d'architectures des composants arrêtés pour reconfiguration seront réévalués lors de leur re-

démarrage.

3.4.5 Contrat d'évolution

Dans [TFS05], les auteurs proposent d'assister le développeur dans l'évolution, à froid, de logiciels construits à base de composants en détectant les impacts d'un changement architectural sur certaines propriétés extra-fonctionnelles. Les propriétés visées sont uniquement des attributs de qualité auxquels sont associés des architectures tendant à les réaliser et des contraintes. Ces contraintes sont exprimées sur un métamodèle généraliste de composants logiciels à l'aide d'une variante d'OCL avec une expressivité proche de celle de CCL-J, le langage d'assertions fourni avec le système ConFract (cf. partie précédente).

3.4.6 Intégrité des communications dans ArchJava et Fractal

ArchJava [ACN02] étend le langage Java avec des concepts d'architecture à composants. Un compilateur est fourni et permet la vérification statique du typage de l'architecture. L'intégrité structurelle de l'architecture est notamment vérifiée à travers la garantie de l'intégrité des communications. Celle-ci consiste à établir que deux composants ne peuvent interagir que si une liaison architecturale explicite existe entre eux. Ces vérifications sont donc effectuées statiquement, mais elles sont de deux formes. Elles doivent vérifier que les interfaces ne sont référencées que par une déclaration architecturale explicite, mais aussi elles doivent assurer que le partage d'objets entre composants se fait correctement. Pour ce faire, ArchJava fournit un système d'annotation qui permet d'associer à chaque objet un domaine d'architecture et d'assurer l'intégrité en assurant qu'un aucun objet n'est partagé hors de son domaine. Cette solution impose néanmoins un modèle de programmation très contraignant (pas de d'échange de références de compoant par exemple), mais finalement relativement cohérent avec le cadre d'ArchJava, peu tourné vers la dynamicité.
Il est intéressant de noter que de récents travaux [LCL06] ont défini et implémenté ce contrôle de l'intégrité des communications dans la plate-forme Fractal/Julia. Cette vérification est alors faite dynamiquement, par le contrôle du flot d'exécution entre composants — pour vérifier que les communications entre les interfaces de composants sont bien conformes aux liaisons architecturales déclarées —, et par un modèle de partage d'objets entre composants — pour contrôler les communications entre les objets qui implémentent des composants Fractal en Java —.

3.4.7 Bilan

Les travaux présentés sont évidemment caractérisés par le fait qu'ils permettent la contractualisation d'éléments architecturaux dans les systèmes à base de composants. On y retrouve ainsi les différentes formes de contrat et les caractéristiques que nous avons déjà distinguées chez elles. Nous avons ainsi abordé des vérifications de typage et de cohérence de l'architecture, ainsi que des vérifications sur les messages circulant (intégrité des communications, forme comportementale reliée à l'architecture). De la même manière, les vérifications peuvent concerner la cohérence d'une spécification ou la compatibilité entre deux spécifications, ou lors de l'exécution, la vérification que l'implémentation est conforme à la spécification.

3.5 Les contrats spécifiques aux architectures orientées services

Cette partie étudie particulièrement les approches contractuelles proposées récemment dans les architectures orientées services. Dans la suite de cette partie, nous nous intéressons aux principales propositions académiques et industrielles qui permettent l'établissement de contrats

et le support des contrats signés. On trouve ainsi, dans le domaine académique, de nombreuses propositions qui se concentrent sur les spécifications des contrats et des offres/demandes par des langages divers, ainsi que sur l'élaboration d'infrastructures de gestion des contrats (construction, surveillance, des contrats, etc.). Dans le domaine industriel, les travaux portent plutôt sur l'exploitation des *Web services* pour l'intégration ouverte des applications et la mise en place de l'infrastructure nécessaire, du coté fournisseur, pour offrir et surveiller des services donnés.

3.5.1 Les éléments contractuels dans les SOA

Compte tenu de l'importance des services (service électroniques, *e-business*) et de la dynamicité des systèmes dans les architectures orientées services (services de nature hétérogène, relations fournisseurs-consommateurs *à la demande*, externalisation de systèmes, modifications transparentes des implémentations, etc.), il est nécessaire de spécifier les relations qui existent entre fournisseurs et consommateurs de services ou entre différents fournisseurs qui coopèrent. À la base, ces relations contractuelles ne sont pas réellement explicitées, et des contrats (ou *Service Level Agreement* - SLA) sont justement définis pour expliciter et identifier les besoins et les engagements des différentes parties sur la réalisation des services. Ces contrats sont généralement bilatéraux et traduisent ainsi d'une façon informelle le fait que, d'un coté, les fournisseurs s'engagent à fournir des services aux qualités spécifiées à la condition que les directives de l'utilisation des services soient suivies, et d'un autre coté, les consommateurs acceptent de respecter les directives d'utilisation pour pouvoir alors bénéficier des qualités de service garanties.

3.5.2 SLA : définitions et mises en œuvre

Les SLA ne constituent pas un concept nouveau et sont déjà traditionnellement utilisés dans le domaine de télécommunications lorsqu'il s'agit par exemple de contractualiser de la réservation de ressources. En revanche, il s'agit à la base de contrats établis manuellement, par des échanges de documents papiers. Le processus de création effective des SLA dépendait alors essentiellement des échanges et du degré d'automatisation de l'infrastructure de réservation de ressources. Appliqués dans les environnements de type SOA, ces SLA doivent maintenant prendre en compte les interactions diverses et dynamiques des différentes parties : de nouveaux services peuvent être intégrés, des SLA existants peuvent évoluer et des SLA peuvent être définis *à la demande*, y compris au-delà des frontières d'une même organisation. Ainsi, un défi majeur des SLA dans les SOA réside en la capacité d'automatiser le processus complet de contractualisation : de la création des contrats, dans des formats traitables en machine, jusqu'à leur surveillance et leur gestion au cours de l'exécution des services, en passant par des mécanismes automatisés permettant de négocier ces contrats pour établir ou maintenir des formes d'accord.

Contenu

Les contenus des SLA sont généralement structurés selon les trois classes d'informations suivantes :

1. *des informations légales*. Elles décrivent les différentes parties en présence (noms, adresses, signatures, etc.) ainsi que les dédommagements en cas de non satisfaction des termes techniques des contrats à la fois pour les fournisseurs et les consommateurs. Par exemple, cela est le cas lorsque des fournisseurs ne respectent pas des niveaux de qualité de services ou que des consommateurs en ont une mauvaise utilisation. Ces informations peuvent aussi être nécessaires à la validité légale des contrats. Par la suite, dans les cas

où les contrats ne sont pas respectés, les mesures à appliquer sont généralement établies au cas par cas selon la nature des relations fournisseur-consommateur (interruption de service et indemnités, gestion totale conférée aux fournisseurs, etc.).

2. *des informations organisationnelles*. Elles décrivent le déroulement des interactions entre les différentes parties lors de l'utilisation des services (nécessité de s'authentifier au préalable, modalités de gestion des problèmes divers, etc.).
3. *des informations techniques*. Elles décrivent les conditions dans lesquelles les services sont rendus. Ces conditions sont représentées sous la forme d'un niveau de service négocié, spécifiant par exemple des paramètres de QoS ou des valeurs garanties. Il existe de nombreux types de contraintes de service, qui s'étendent de ceux qui spécifient des aspects extrafonctionnels généraux (disponibilité, performance, etc.) jusqu'à ceux qui s'appliquent à des utilisateurs donnés dans des contextes bien précis (l'accès à des services spécifiques pour des employés d'une même entreprise). Ainsi, de telles contraintes peuvent porter sur la *disponibilité*[1] et la *performance* moyenne et/ou garantie des services (ex : «*un temps inter-pannes d'au plus 2 heures*», «*un taux de perte de données lors d'une transaction soit inférieure à 0.01 %*»), les mécanismes de mesures et de reports associés (modalités de mesures, autorité de contrôle tierce, etc.) ou encore les coûts financiers des services.

Établissement des contrats

Bien que les contrats électroniques aient pour but de formaliser des accords acceptés mutuellement par les deux parties, fournisseurs et consommateurs de services, le processus d'établissement des contrats reste cependant bien souvent asymétrique et contrôlé par les fournisseurs. Ce processus, sous sa forme la plus générale, se décompose selon le schéma suivant :
– les fournisseurs de services créent des gabarits de contrats (*contract template*) qui définissent notamment la liste des services offerts et les niveaux de services associés, et intègrent parfois les coûts financiers des services ainsi que des pénalités en cas de violation des termes du contrat. Ces gabarits de contrats sont envoyés aux consommateurs, puis,
– les consommateurs sélectionnent les services qui les intéressent et créent une instance de contrat. Ce contrat est renvoyé et soumis à validation par les fournisseurs.

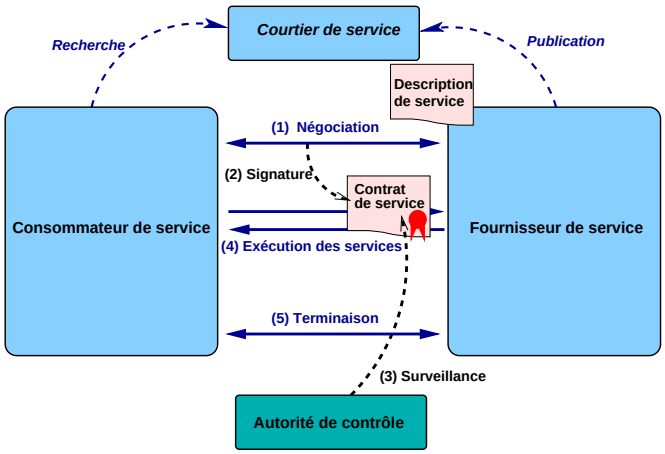

FIG. 3.4 – Cycle de vie des contrats.

[1] La 'disponibilité' est l'aspect extrafonctionnel le plus souvent mentionné dans les SLA, bien qu'il n'y ait pas de définition commune.

Des variations possibles de ce schéma proviennent du fait que les fournisseurs peuvent intégrer plus ou moins de flexibilité dans le gabarit de contrat initial en proposant le choix entre *(i)* différents services à des niveaux fixes, *(ii)* différents niveaux de services pour des mêmes services ou encore *(iii)* des paramétrisations sur d'autres aspects des contrats (coûts, pénalités, etc.). Ainsi, les fournisseurs et les consommateurs entrent alors dans un processus de négociation basé sur des échanges successifs de tels documents qui expriment les contrats souhaités par les consommateurs et ceux acceptés par les fournisseurs. Bien évidemment, cela implique que les fournisseurs de services aient les capacités pour supporter et proposer différents types des services, et que les deux parties puissent dialoguer durant ce processus de négociation. Par la suite, un ensemble de systèmes sont intégrés pour surveiller et gérer les contrats négociés lors de l'exécution des services.

3.5.3 Spécification d'accords et de contrats

Les travaux qui portent sur l'établissement des formes d'accords et les mécanismes de négociation s'attachent en grande partie à exprimer des offres et à élaborer des accords par le biais des langages de spécification. Ces langages se focalisent notamment sur l'expression des capacités des fournisseurs de services, l'expression des contraintes de niveaux de services et aussi sur la structure et la sémantique des messages XML.

WSOL

WSOL (*Web Service Offerings Language*) [TPP02] est un langage basé sur XML qui permet de spécifier des niveaux de services en étant rattachés aux descriptions WSDL des services. WSOL s'appuie sur la notion de classes de services qui représentent des SLA simples dans lesquels les consommateurs choisissent un service donné parmi ceux proposés par les fournisseurs. Le langage WSOL permet d'exprimer diverses contraintes :
– fonctionnelles : pré/postconditions et invariants ;
– de qualité de services ;
– de droits d'accès (simples) ;
– de prix.
Dans une offre de service WSOL peuvent aussi être spécifiées des entités responsables de surveiller une contrainte particulière sur un service. Typiquement, les grandeurs manipulées pour contraintes la QoS (temps de réponse, etc.) sont surveillées par des entités externes. Il est aussi possible de spécifier des relations entre les offres de service. En fait, cela permet simplement de spécifier, dans une offre de services, quelle autre offre de services doit être consulté si un contrat est violé sur une contrainte spécifique. Cela établit alors *en dur* les différents niveaux de service et la manière de négocier, en tentant simplement l'alternative référencée.

WSLA

Le canevas WSLA (*Web Service Level Agreement*) [KL03a] constitue une approche plus complète en proposant à la fois un langage de spécification de SLA ainsi qu'une infrastructure pour établir et surveiller les contrats lors de l'exécution des services. Le langage proposé [LKD+03], basé sur XML Schema, décrit les parties en présence, la description des services (caractéristiques et paramètres observables) et les obligations. Le métamodèle sous-jacent au canevas WSLA est présenté dans la partie 3.6.3.
Le langage WSLA est riche et introduit en particulier :
– des paramètres de SLA, qui sont des propriétés (extra-fonctionnelles) du service. Chaque paramètre a un nom, un type et une unité. Ces paramètres sont l'équivalent des QoS (disponibilité, temps de réponse, etc.)

– des métriques de QoS qui aggrègent d'autres métriques, récursivement. Une métrique peut être décrite par une fonction associée à un timer ou à un déclencheur associé à un événement. Mais une métrique peut aussi être définie par une directive de mesure, c'est-à-dire un bout de programme pour récupérer dans le système la valeur pertinente.

A partir de ces éléments, il est possible d'exprimer des contraintes de niveaux de services (*Service Level Objectives* - SLO) et des actions à effectuer. Les contraintes de niveaux de services sont exprimées dans une logique propositionnelle, donc avec une expressivité limitée par rapport à des assertions exécutables classiques. Les garanties d'action à effectuer concernent l'invocation d'opérations, mais aussi la notification des violations de certaines contraintes.
La phase de négociation s'appuie sur des échanges successifs de gabarits de contrats WSLA (*WSLA template*) entre fournisseurs et consommateurs de services. Le système de gestion quant à lui s'intéresse plus à la surveillance des contrats et à la réservation des ressources du coté fournisseur pour supporter les niveaux de services offerts.

SLAng

SLAng [SLE02] est aussi un langage de définition de SLA qui se focalise sur la définition de la QoS avec une focalisation sur les services réseaux et de middleware et sur le maintien de la QoS de bout en bout. On retrouve dans SLAng des caractéristiques classiques de définition de paramètres pour les SLA, et un schéma XML décrivant les parties, les contraintes, etc.
En plus, SLAng définit des responsabilités pour chaque sorte de SLA, avec la possibilité d'exprimer une responsabilité au client, au serveur, ou une responsabilité mutuelle. Hormis cela, les concepts manipulés sont très classiques pour le domaine. Dans [SLE04], la sémantique de SLAng est définie en UML et OCL et des notions de compositionnalité de SLAs sont introduites :

– composition inter-service (compatibilité entre niveaux de QoS des services requis et fournis),
– composition intra-service, pour laquelle l'expression de la QoS d'un service est liée à celle de ses composants.

La notion de compatibilité pour la composition inter-service est finalement assimilée par les auteurs à la comparaison utilisée en QML, c'est-à-dire la comparaison de niveaux prédéfinis. Tout autre test de compatibilité sur des expressions plus complexes (assertions généralistes) serait décidable, mais les auteurs semblent redécouvrir cette problématique. Enfin, la composition intra-service est uniquement évoquée et rien n'est vraiment fourni pour prédire la QoS.

Ws-Agreement

Ws-Agreement [AKCK+05] constitue un effort de standardisation et d'unification des nombreux langages existants dans le but de définir un langage et un protocole qui permettent d'annoncer les capacités des fournisseurs de services (templates), de créer les accords à partir des offres et de surveiller les contrats établis. Ws-Agreement s'attarde principalement sur la définition de la structure des documents représentant les accords mais n'abordent pas les mécanismes permettant de les atteindre. Le métamodèle et les concepts sous-jacents à la spécification Ws-Agreement sont détaillés dans la partie 3.6.3.

Ws-Negotiation

Ws-Negotiation [HLJ04] propose un langage déclaratif basé sur XML pour la négociation des SLAs. Ces travaux s'inspirent en partie des systèmes de négociation automatisés et introduisent les briques de base d'une négociation (messages, protocoles et stratégies), de façon

assez similaire aux systèmes de négociation introduits dans les systèmes multi-agents. Toutefois, ces travaux sont encore en cours d'élaboration et de standardisation auprès de la Global Grid Forum (GGF) [Glo] et dépendent en partie des avancées du Ws-Agreement.

Bilan

Les notions d'accords sont encore assez instables notamment du point de vue de la sémantique des propositions et de la structure des messages échangés. De plus, les mécanismes qui permettraient d'élaborer de tels accords, y compris par des formes de négociation automatisées, restent à l'heure actuelle peu aboutis et consistent bien souvent à des échanges simples de gabarits de contrats dont la description demande encore à être développée (définitions communes ?, réutilisations des gabarits ?). Les échanges entre fournisseurs et consommateurs conduisent à effectuer des choix de services à des qualités données ou des choix de valeurs de certains paramètres dans des échelles prédéfinies par les fournisseurs, par exemple par de la mise en correspondance des voeux de chaque partie (*match-making*) [MM02]. Il n'y a ainsi par réellement de protocoles d'interaction et de négociation dynamique entre fournisseurs et consommateurs pour l'établissement des contrats.
Par ailleurs, une tendance actuelle consiste aussi à exploiter des techniques de Web sémantiques afin de concevoir des langages de spécification plus expressifs (ontologies, définitions de nouvelles métriques de qualité, etc.) qui permettraient d'exprimer plus finement les notions d'accords et d'envisager la mise en œuvre de techniques d'inférence [SMS+02b, JW05].

3.5.4 Système de gestion de contrats

Pour mener à bien la phase d'exécution des contrats, les fournisseurs et consommateurs de services ont besoin d'une infrastructure permettant de spécifier, déployer et surveiller les contrats électroniques.

Infrastructure de gestion de WSLA

Dans l'infrastructure de gestion de WSLA [KKL+02, KL02], la gestion des contrats est effectuée par le biais d'un ensemble de services qui permettent de surveiller et mesurer les paramètres de QoS, et de déclencher des activités données pour résoudre les problèmes. Ces services sont souvent assurés par des autorités de gestion indépendantes à la fois des consommateurs et des fournisseurs (*Management Service Provider*). Les services qu'ils réalisent usuellement sont :
– la mesure des paramètres de QoS par interception des appels entre les clients et fournisseurs ;
– la vérification périodique de l'adéquation entre les qualités mesurées et celles indiquées dans les contrats ;
– le déclenchement d'actions d'adaptations en cas de violation des contrats. Ces actions consistent à *(i)* réaliser des politiques prédéfinies, *(ii)* alerter un administrateur ou encore *(iii)* signaler la violation au fournisseur ou au client impliqué.

WSML

Hewlett-Packard a développé un langage basé sur XML pour spécifier les SLA des Web services [AVM+02, AAV02, VAvMA02] : il s'agit du "Web Service Management Language" (WSML) qui se base sur WSDL et WSFL. Un document WSML consiste en la description d'une période de validité du SLA, des parties impliquées, et un ensemble de SLOs. Chaque SLO contient une description du jour et de l'heure durant lesquels le SLO est valide, ainsi

qu'un ensemble de clauses. Une clause décrit un ou plusieurs objets (par exemple des opérations) pour lesquels une mesure est effectuée, des temps ou évènements qui déclenchent l'évaluation, des échantillons de mesure utilisés pour l'évaluation, une fonction d'évaluation booléenne, et une action effectuée dans le cas où la fonction d'évaluation retourne un résultat négatif. La fonction d'évaluation capture la définition d'une métrique de QoS. Son objectif est de calculer cette métrique en fonction des échantillons de mesure et de vérifier une condition. Les documents rédigés avec le langage WSML peuvent être gérés par une infrastructure "Web Services Management Network" (WSMN) développée à cet effet.

Cremona

Cremona [LDK04] est une architecture dans laquelle les fournisseurs et consommateurs de services établissent des accords selon le standard Ws-Agreement. L'architecture consiste principalement en des entités de gestion des différentes phases définies dans un Ws-Agreement :
- Annonce des capacités des fournisseurs par des templates et choix d'une version appropriée par le client : Cremona fournit des entités séparées qui dialogue à la place du client et du fournisseur.
- Etablissement de l'accord : plusieurs entités de gestion s'occupent des rôles des parties, d'une stratégie de pilotage de haut niveau
- Surveillance : Le système définit un ensemble de moniteurs permettant de surveiller les services et les contrats ainsi que les systèmes de gestion des ressources des fournisseurs de services.

En revanche, cette architecture souffre aussi des limitations du standard Ws-Agreement : les termes des contrats ne peuvent être modifiés une fois qu'ils sont établis et les capacités de négociation dynamique et d'adaptation pour maintenir les accords établis sont encore peu abordées.

Autres systèmes industriels

D'autres systèmes industriels se focalisent sur l'acquisition et la surveillance des ressources et des données :
- SNAP (*Service Negotiation and Acquisition Protocol*) [CFK+02] est un protocole qui s'adresse davantage à l'acquisition de ressources dans un environnement de type grille et dans lequel les différents services offerts consistent directement à utiliser des niveaux de ressources donnés.
- HP Open View est une suite d'outils pour la surveillance des SLA qui se focalise essentiellement sur la signalisation des violations (reporting, audit, etc.) de SLA notamment par le biais de sa plate-forme surveillance (*Business Management Platform Agent*) [SMS+02a]. Les mécanismes de gestion dynamique des SLA ne sont pas encore abordés.
- IBM propose aussi son système de gestion couplé à WSLA [DDK+04]. Ce système permet de se connecter à différentes sources pour collecter des données et envoyer des événements relatifs aux SLA. Ces systèmes s'attachent actuellement à s'intégrer aux infrastructures existantes pour pouvoir y greffer des agents de collectes de données. La gestion des SLA consiste alors essentiellement à effectuer de la surveillance d'entités bien ciblées pour pouvoir rendre compte des violations de SLA et éventuellement appliquer des pénalités. En revanche, des mécanismes de gestion des contrats basés sur des interactions dynamiques entre fournisseurs et consommateurs de services ne sont pas encore d'actualité. Ces systèmes industriels s'attachent surtout pour l'heure à appliquer le modèle d'architecture orientée services pour remplacer les connections propriétaires par une approche basée sur l'intégration de services standardisés dans un contexte d'intégration ouverte d'applications d'entreprises.

3.5.5 Application d'approches existantes aux SOA

Les principales approches formelles, autour des automates et des algèbres de processus, ont récemment été adaptées à la problématique des architectures orientées services.

Web Service Interfaces

Dans [BCH05], les auteurs proposent de porter leur technique de spécification d'interfaces (cf. partie 3.2.3) aux Web services. Pour ce faire, trois types de contraintes sont associés à un service :
– des contraintes de signature, qui définissent les méthodes et leurs paramètres ;
– des contraintes de cohérence, qui sont des contraintes propositionnelles sur les appels et les résultats des méthodes
– des contraintes de protocole, qui spécifient des contraintes temporelles sur l'ordonnancement des appels de méthode.
Il est alors possible de vérifier la compatibilité et la substituabilité de deux interfaces. Le cadre de spécification et de vérification est vraiment identique à ce qui avait été développé pour les composants. Les deux approches raisonnent simplement sur une interface composée de méthodes.

Composition de services et FSP

Dans [FUMK03], les auteurs appliquent leur technique de spécification comportementale à base de FSP (cf. partie 3.2.4) à la vérification de la composition de Web services. L'approche se base sur BPEL4WS, un des standards pour spécifier et exécuter des spécifications de *workflow* pour composer des Web services. Le processus proposé s'articule de la façon suivante :
– Le comportement du workflow est spécifié à l'aide de diagrammes de séquence UML, ou plus précisément à l'aide de LTSA-MSCs, c'est-à-dire des *Message Sequence Charts* interprétés dans l'environnement *Labelled Transition System Analyzer*. Cet environnement permet de construire et d'analyser des modèles de spécification en FSPs.
– Une implémentation en BPEL4WS est développé et traduite dans une représentation en FSPs (la traduction s'appuie sur le déroulement des activités et ignore les détails techniques présents dans la description BPEL4WS).
– Une vérification d'équivalence de traces entre les deux modèles FSP est effectuée. Des correspondances d'abstraction sont d'abord faites entre les activités décrites de part et d'autre, puis la vérification d'équivalence peut faire apparaître des scénarios que le modèle décrit alors qu'ils n'ont pas été spécifiés par le développeur.
– Le processus peut être considéré comme itératif, jusqu'à ce qu'aucun inter-blocage ou violation ne soient détectés.
La principale contribution de ces travaux réside dans l'association des techniques connues des FSP à BPEL4WS. Néanmoins, les vérifications ne sont que des tests de compatibilité entre une spécification abstraite et une autre extraite d'une description BPEL4WS.

3.6 Métamodèles et formalismes de contrat

On présente dans cette partie un ensemble de modèles et de formalismes de contrat, en s'attachant aux propriétés qu'ils réifient. On s'intéressera ainsi à la représentation des responsabilités, à la définition du cycle de vie du contrat, aux liens du contrat avec l'architecture qu'il contraint, etc. On s'intéressera aussi à ce que chacun de ces modèles de contrat exprime. En effet, dans [BBB+00] ,le *Software Engineering Institute* distingue deux types de contrat. Un premier exprime l'adhérence d'un élément à sa spécification alors que le second concerne la

compatibilité d'un ensemble d'éléments collaborant via celle de leurs spécifications. Ces modèles et formalismes sont regroupés suivant la nature des éléments qu'ils contraignent : objets (partie suivante), composants (partie 3.6.2), services (partie 3.6.3) et agents (partie 3.6.4).

3.6.1 Objets

Les contrats sur les objets sont sans doute les premiers historiquement apparus. On peut distinguer les contrats exprimant qu'un objet satisfait à une spécification, des contrats exprimant la compatibilité entre deux spécifications. L'approche fondatrice du Design By Contract de Meyer [Mey92], permet à la fois la vérification de la satisfaction de pre post conditions par une classe et l'étude de la compatibilité entre ces assertions via celle du sous typage comportemental. Dans ce dernier cas, la compatibilité d'assertions portées par des classes parentes sur des méthodes identiques est étudiée, ce qui valide le polymorphisme du point de vue de cette approche.

OCL. Toutefois la métamodélisation d'OCL (figure 3.5) proposée par [RG99] bien qu'elle réifie les pré et post conditions semblables à celles du *Design by Contract* ne fait pas apparaître cette distinction. Elle présente les assertions et leur lien avec les objets qu'elles contraignent mais n'exprime pas le sous typage comportemental, ainsi que le montre la figure .

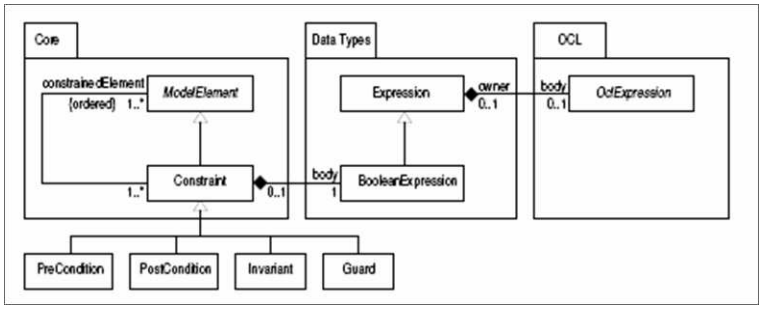

FIG. 3.5 – métamodèle OCL

Contrats et collaboration. Une autre approche contractuelle est mise en oeuvre dans [HHG90] (figure 3.6). Dans celle-ci les contrats sont explicitement réifiés, constitués de participants et d'obligations, et appliqués aux compositions d'objets. Les principes de satisfaction des participants à des spécifications et de compatibilité entre ceux-ci sont exprimés via des obligations de deux genres. Le premier principe est rendu par une catégorie d'obligations portant sur les propriétés de typage des participants. Le second principe est exprimé via des obligations "causales", capturant des dépendances comportementales entre objets, en contraignant les séquences de messages émis et reçus de chacun (clauses `supports`). Des invariants doivent aussi être satisfaits par les participants. Il faut noter que ces contrats supportent entre eux des relations de raffinement et d'inclusion (`includes`). Il est aussi intéressant de constater que l'instantiation des contrats peut être soumise à des conditions, on voit ici ainsi apparaître une notion de cycle de vie du contrat. Enfin, les auteurs proposent une notion de conformance pour rendre compte de l'acceptabilité d'une classe à un rôle défini dans un contrat. Néanmoins il faut remarquer que ce type de contrat spécifie la composition en elle même, et non la compatibilité entre les spécifications de ses participants.

Contrats et coordination. Une extrême de l'approche précédente est rendue dans les contrats de coordination définis par [AF99] (figure 3.7). Ces contrats, réifiés, possèdent des participants, auxquels sont appliqués des invariants exprimant leurs relations. Mais ces contrats contiennent aussi l'expression de la coordination entre leurs participants. Dérivés de classes d'associations, ils sont placés entre les instances de ceux-ci et interceptent de manière transparente leurs messages, déclenchant éventuellement ainsi des actions définies par des règles

```
contract AdjustView
    Viewer supports [
        Adjust(a:Adjustment) ↦ Perspective→SetValue(fcn(Picture→getShape(),a))
        ]
    Adjuster supports [
        a : Adjustment
        Attach(v:Viewer) ↦ {Viewer = v}
        Activate() ↦ Δa; Viewer →Adjust(a)
        ]
    Perspective supports [ ]
    Picture supports [
        shape : Shape
        getShape() : Shape ↦ return shape
        ]
    includes
        SubjectView(Views = {Viewer}, Subject = Perspective)
        ParentChild(Children = {Picture}, Parent = Viewer)
    instantiation
        Adjuster →Attach(Viewer)
end contract
```

FIG. 3.6 – Contrat d'interaction

"when <condition> do <actions> with <condition>". Dans ces contrats la garantie de la compatibilité entre les participants ne porte pas sur leurs comportements individuels mais est exprimée via des invariants. La garantie des spécifications individuelles n'est pas exprimée. Ainsi la composition n'est plus vérifiée sur la base du comportement des participants, mais programmée dans le contrat. Toutefois ces contrats sont explicitement positionnés entre leurs participants et sont transparents pour ceux-ci. Par ailleurs les auteurs de ce modèle en proposent une sémantique formelle.

```
contract Traditional package
        participants x : Account; y : Customer;
        constraints ?owns(x,y)=TRUE;
        coordination : when y.calls(x.withdrawal(z))
                       with x.Balance() > z
                       do call x.withdrawal(z);
end contract
```

FIG. 3.7 – Contrat de coordination

Contrats et responsabilités. Des contrats issus des travaux sur les agents sont utilisés dans [PBP99] pour modéliser de manière formelle les Use Cases de la conception objet. Ils reposent sur un langage simple définissant une algèbre de contrats, sachant que les auteurs définissent un contrat comme la description des voies par lesquelles un acteur peut modifier le système. Dans ce cadre le système et les acteurs sont considérés comme des agents et contraints par un langage dont la syntaxe de base est la suivante :

$$contract = ... \mid assert_a\ p \mid update_a\ R \mid contract1; contract2 \mid ...$$

"$assert_a$" spécifie que l'agent a doit vérifier p, "$update_a$" que l'agent a doit modifier le système de sorte que les états consécutifs du système soient liés par la relation R, " ;" met en séquence deux contrats etc. L'intérêt de cette approche est qu'elle réifie explicitement les responsabilités de chacun des acteurs et du système, ainsi que la composition des contrats. Elle se prête par ailleurs à des raisonnements formels sur les contrats, qui ont pour conséquence pratique d'évaluer si certaines actions d'un agent sont possibles d'après le contrat.

Discussion
Si le contrat tend à être réifié quand on s'éloigne du Design By Contract, les responsabilités

ne le sont en général pas, ni le cycle de vie du contrat. Hormis dans le dernier exemple il n'y a pas de dépendance exprimée entre les provisions du contrat, en conditionnant une par une autre. Le lien avec l'architecture est plutôt ténu, puisque la composition est considérée en dehors des relations individuelles entre les objets, et aucunement dans le dernier cas de figure des contrats.

3.6.2 Composants

Les contrats occupent une place explicite dans certaines définitions de composants, ainsi Szyperski considère que un composant est "a unit of composition with contractually specified interfaces and explicit context dependencies only". On constatera dans cette partie que les contrats sur les composants ne se limitent pas à la définition des interfaces mais contribuent à la garantie des assemblages.

QCCS. Un métamodèle de contrat (figure 3.8) est défini au sein des travaux relatifs à QCCS [SAD+02] dans le cadre de la modélisation UML version 1.4 des composants. Ce modèle adresse la garantie de propriété fonctionnelles et non fonctionnelles des interfaces de composants. En effet celles-ci sont complètement décrites par des contrats, tant pour leur signature que vis à vis de la qualité de service qui repose sur QML [FK98]. Pour ce faire ces contrats peuvent être composés. Leur mise en oeuvre est envisagé via un tissage dans le code à la manière des aspects. Ce modèle est intéressant par son intégration à UML, toutefois seuls les contrats en tant que satisfaction des composants à des spécifications sont envisagés. Par ailleurs, le contrat en lui même n'est pas détaillé, on ne voit ainsi par exemple ni ses clauses, ni son cycle de vie, etc.

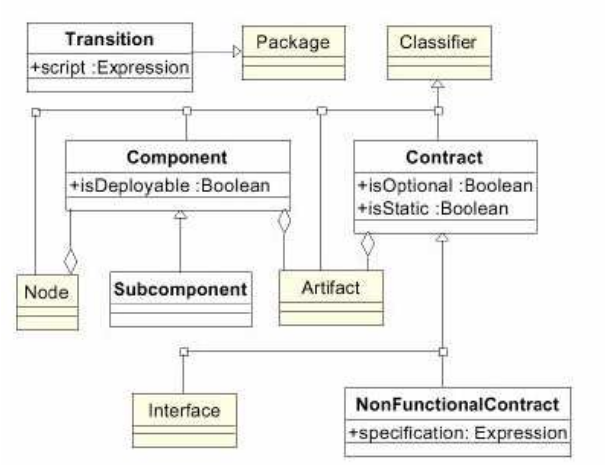

FIG. 3.8 – métamodèle QCCS de contrat

QoSCL. Un autre modèle de contrat, QoSCL [JDP03], est décrit dans le cadre de UML 2.0 (figure 3.9). Ce modèle intervient aussi bien au niveau de la garantie des spécifications des interfaces d'un composant, qu'à celui de la compatibilité de ces spécifications au sein d'un assemblage de composants. La notion de contrat reprend celle de QML, ainsi un type de contrat associe à une interface un ensemble de dimensions, c'est à dire de grandeurs non fonctionnelles contraintes. Ce contrat est vérifié à l'exécution via son tissage dans le code de l'application. Par ailleurs, les contrats sont des constituants d'un QoSComponent, dérivant de la classe UML Component, dont ils sont requis et fournis. Le formalisme permet d'exprimer qu'au sein d'un composant, il existe des relations entre les dimensions des contrats requis et fournis. Les auteurs présentent une méthode de traduction en langage logique de ces rela-

tions et dimensions pour chacun des composants. Ils appliquent ensuite à cette description de l'ensemble du système de composants, un solveur de contraintes qui fournit les domaines de valeur acceptables pour les dimensions des contrats. Ce modèle présente l'intérêt d'expliciter des relations entre clauses du contrat. Par ailleurs même si le contrat ne garantit que l'adhérence d'un composant à ses spécifications, l'assemblage de composants est garanti par la vérification de la composition de leurs contrats. Toutefois le problème des responsabilités n'est pas abordé et la garantie de l'assemblage n'est pas l'objet d'un contrat unique.

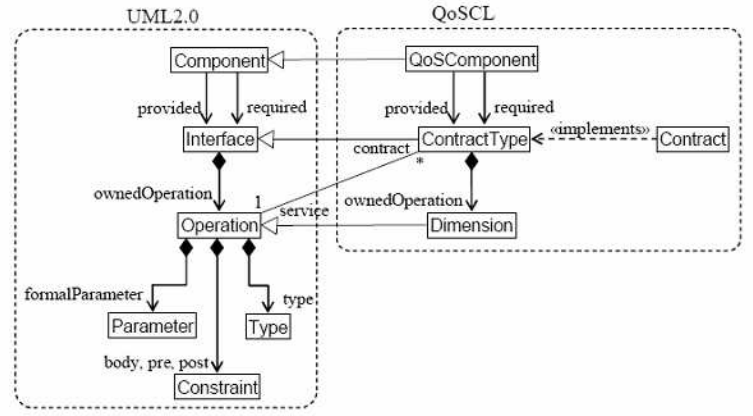

FIG. 3.9 – métamodèle QoSCL de contrat

GCCL. Dans le cadre des travaux [MPJ02] les auteurs définissent un langage de contrat destiné à vérifier la satisfaction des composants à leurs spécifications ainsi que la correction de leurs assemblages, tant du point de vue fonctionnel que non fonctionnel. Ce langage GCCL (Generalized Common Contract Language, figure 3.10) décrit les contrats comme des ensembles de contraintes exprimées à l'aide de pré, post conditions et d'invariants appliqués à divers éléments. Il faut noter que ce langage fournit un modèle de l'architecture à laquelle s'applique les contrats, un modèle des caractéristiques de QoS qu'ils contraignent, un modèle des événements ainsi que des cycles de vie des éléments de l'architecture contrainte. L'intégration des contrats dans l'architecture contrainte est ainsi complète. Le modèle événementiel permet de positionner le déclenchement de la vérification des contraintes par rapport aux cycle de vie des entités sur les caractéristiques desquelles elles portent. Comme les contrats prennent en paramètres les composants ou interfaces présentant les grandeurs de QoS qu'ils contraignent, l'adhérence des composants à leur spécification peut être vérifiée à l'exécution. Ces contraintes peuvent par ailleurs dépendre des variables que les contrats prennent en paramètre, et se retrouvées corrélées quand elles en ont en commun. D'autre part, le modèle de caractéristiques de QoS est défini de sorte qu'il soit possible d'appréhender les possibilités de traitement de leurs contraintes par des solveurs. La satisfaction des composants aux contrats peut ainsi être vérifiée statiquement. Finalement, des dépendances entre contrats peuvent être définies de diverses manières, par exemple via des paramètres qu'ils partagent. La compatibilité des éléments contractualisés et donc la validité des assemblages peut ainsi être vérifiée. Toutefois la notion de responsabilité n'est pas explicitée, ni les rôles des participants au contrat qui n'interviennent qu'en tant que "fournisseurs" de grandeurs contraintes.

Liaison entre comportements et architecture. Un modèle de contrat beaucoup plus orienté vers la garantie de l'assemblage des composants est décrit dans [TBD+04] (figure 3.11). Dans celui-ci sont réifiés les spécifications des ports des composants (`AssemblyContract`), mais aussi celles des liens entre ports (`Dependence`, `Synchronization`, `LinkBehavior`). Ainsi une connexion entre deux ports est validée par la vérification statique de la compatibilité de leurs `AssemblyContract` respectifs, constitués de pré et post conditions écrites dans un langage logique. Le contrat `Dependence`, exprime la dépendance comportementale entre ports

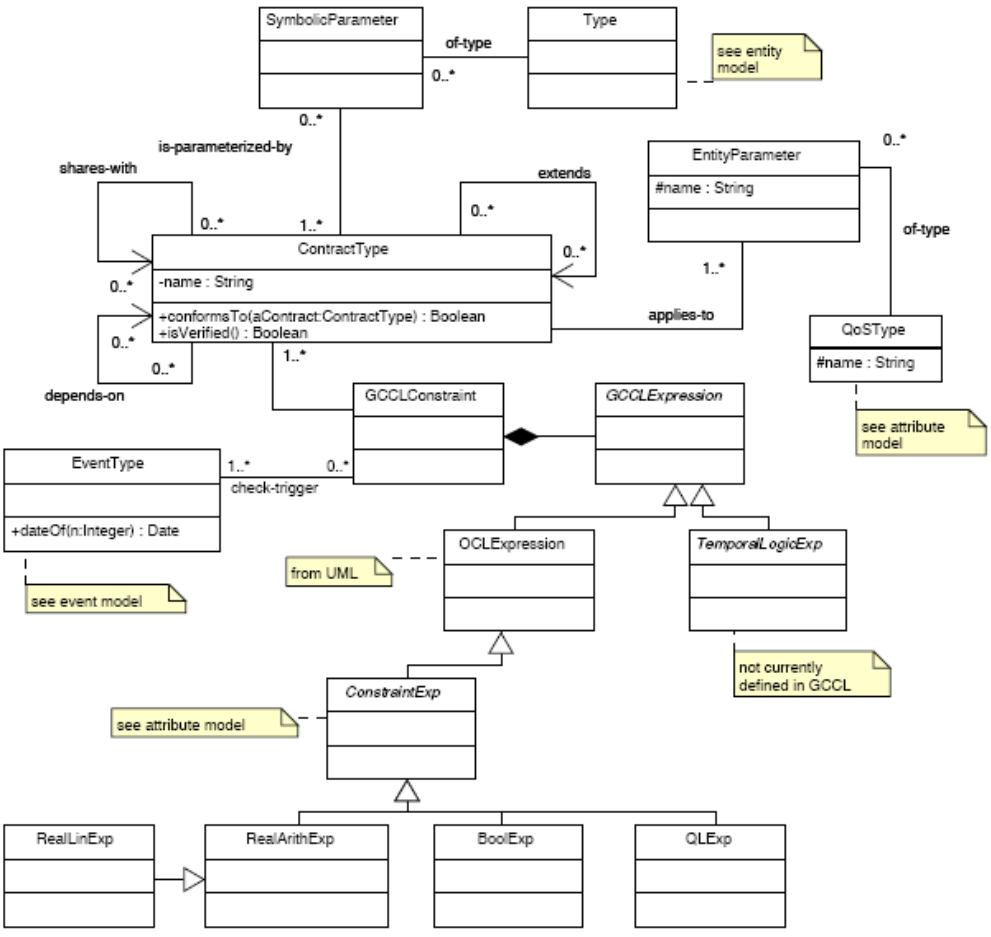

FIG. 3.10 – métamodèle GCCL de contrat

d'un même composant, le `LinkBehavior` entre ports de composants distincts. Ces contrats sont en fait des spécifications des comportements dont la composition fournit celui de l'assemblage de composants, tout en garantissant leur compatibilité. Ce modèle de contrat présente l'avantage d'être très proche de l'architecture, puisque à chaque relation architecturale correspond un type de contrat. Par ailleurs les contrats comportementaux locaux sont composables afin d'obtenir le contrat de la composition prise globalement. Toutefois la notion de responsabilité n'est pas abordée dans ces différents contrats. Par ailleurs, comme dans QoSCL, la validité de l'assemblage, c'est à dire la compatibilité de ses constituants est garantie par la composition des contrats mais pas par un contrat unique réifiant des rôles, responsabilités, etc., sur lequel raisonner.

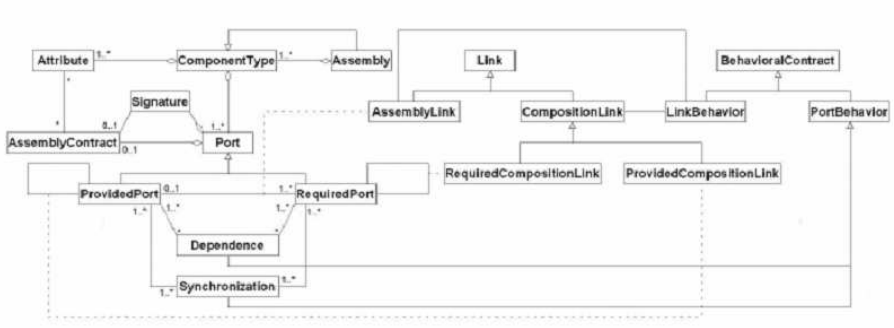

FIG. 3.11 – métamodèle pour l'analyse de la composition

CQML. CQML [yA01] est un formalisme de description de qualité de service dédié aux composants (figure 3.12). Il ne réifie pas de contrat mais décrit comment en définir et les vérifier sur la base des termes qu'il fournit. CQML permet de décrire des qualités (`quality`) qui sont des contraintes sur des grandeurs de qos (`quality_characteristic`). Un profil (`profile`) associe un ensemble de qualités à un composant, en distinguant les qualités qu'il requiert (`uses`) de celles qu'il fournit en échange (`provides`). Un contrat garantit dans ce cadre la validité d'une composition de composants : "Un contrat de QoS est un accord au moment du run-time entre des composants collaborant sur la QoS que chaque composant a la responsabilité de fournir". En résumé un contrat entre composants est l'association de l'ensemble de leurs profils. Pour que le contrat soit validé, il faut que ce que les uns fournissent corresponde à ce que les autres requièrent. Il est intéressant de voir apparaître la notion de responsabilité des composants et celle de garantie, inhérente aux contrats de la vie courante. Par ailleurs le principe de compatibilité entre les spécifications est explicitement mis en oeuvre entre celles fournies et celles requises. Toutefois, les correspondances entre profils se font sans tenir compte du détail de l'architecture, l'ensemble des spécifications fournies doivent satisfaire à chacune de celles qui sont requises.

Requirements and Assurances Contracts. Les Requirements/Assurances Contracts [Rau00] réalisent plus formellement les contrats décrits dans le cadre de CQML. En effet dans cette approche, la spécification des composants comprend des propriétés qu'ils requièrent et d'autres qu'ils fournissent. Il s'agit en particulier des interfaces des composants dont le comportement des méthodes est formellement spécifié. Les contrats formellement réifiés (figure 3.13) sur ces bases sont constitués de participants, les composants en interaction, et de correspondances entre propriétés requises et fournies par ces derniers. Le contrat est valide dans la mesure ou les correspondances entre propriétés requises et fournies sont satisfaites. Ces contrats présentent l'intérêt d'expliciter la notion de compatibilité entre les composants participants au contrat. Il faut noter que les correspondances sont établies sur le rapport entre spécifications de méthode requise et fournie, ces clauses de contrats ne sont donc à chaque fois que binaires, même si le contrat peut faire intervenir plus de deux participants. La portée des spécifications

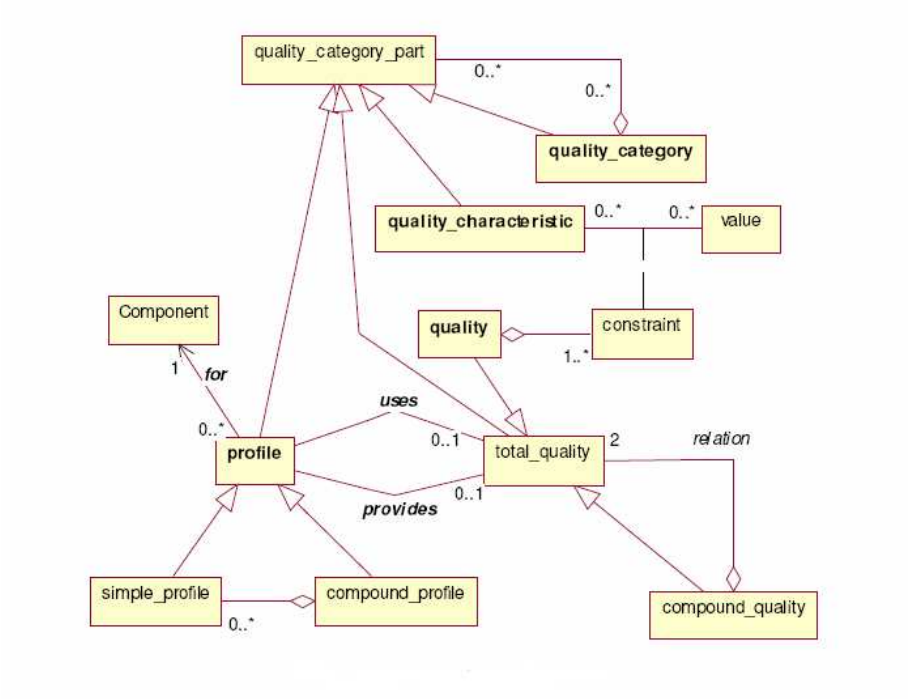

FIG. 3.12 – Modèle de description de la QoS dans CQML

des méthodes n'est pas clairement définie, il semble qu'elle recouvre le composant dans son ensemble.

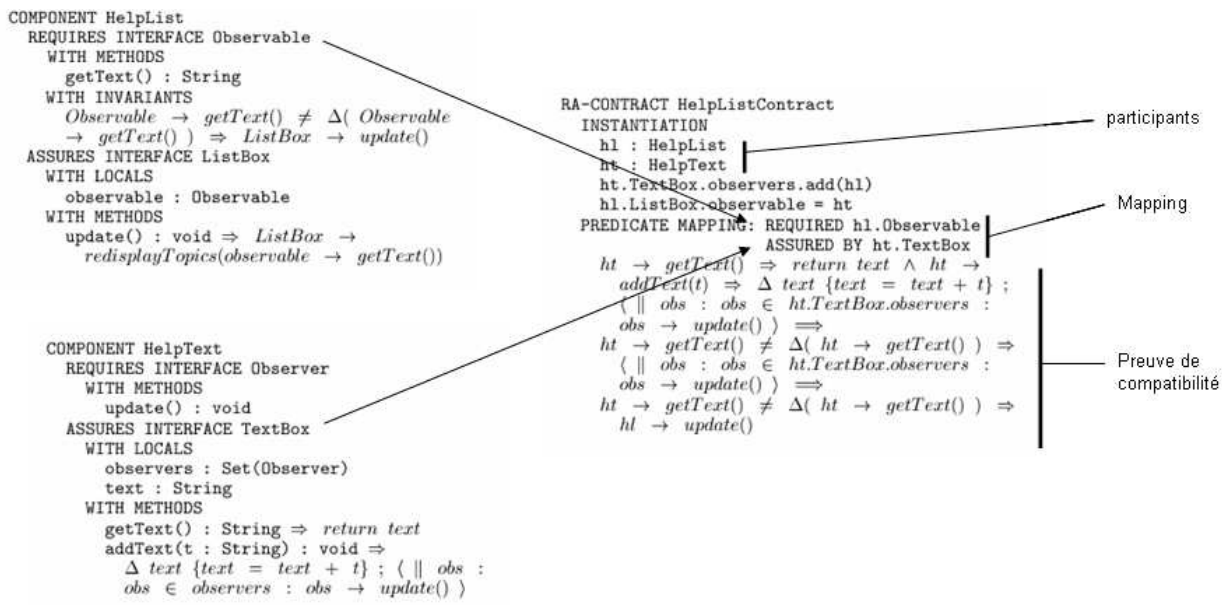

FIG. 3.13 – Requirements/Assurances Contracts

Discussion

Les responsabilités et rôles des composants ne sont pas réifiés dans les contrats rencontrés. Par contre l'architecture est prise en compte à divers niveaux. L'approche la plus simple consiste à constater que des composants collaborent et à vérifier que l'association de leurs spécifications n'est pas contradictoire, comme dans QoSCL, CQML. L'architecture est prise en compte plus finement dans les Requirements/Assurances Contract qui mettent en relation explicitement

les parties fournies et requises. Enfin le modèle [TBD+04] prend en compte chaque relation entre les composants, mais ne dispose pas d'un contrat global à l'assemblage.

3.6.3 Services

Dans le domaine des services, le contrat prend une importance particulière. En effet l'utilisation de services s'intègre dans des processus métier qui sont contraints par des critères tant non fonctionnels que fonctionnels. La simple description des interfaces ne suffit plus, l'objet des contrats est initialement dans ce cadre de limiter le risque que prend l'utilisateur d'un service tant du point de vue fonctionnel et que non fonctionnel. Nous étudions ici les métamodèles des approches les plus significatives actuellement.

SLA. Un métamodèle de contrat du type Service Level Agreement, SLA, est proposé dans les travaux [KKL+02] avec la préoccupation de s'appliquer à des services à l'évolution dynamique. Le contrat réifié (figure 3.14) est constitué de 3 types de composantes. Il possède une description du service ainsi que des participants au contrat, c'est à dire le fournisseur et les clients du service. Il contient aussi l'ensemble des contraintes (`Guarantee`), sur des grandeurs de QoS (`QoSParameterDefinition`), qui sont garanties par le fournisseur. Ce modèle a la particularité d'autoriser l'ajout et le retrait dynamique de contraintes (`Guarantee`). Toutefois, il n'exprime que la conformité d'un service à sa spécification et non une compatibilité entre des services. Par ailleurs, la responsabilité vis à vis des garanties est implicitement celle du fournisseur de service bien que d'autres acteurs soient présents dans le contrat.

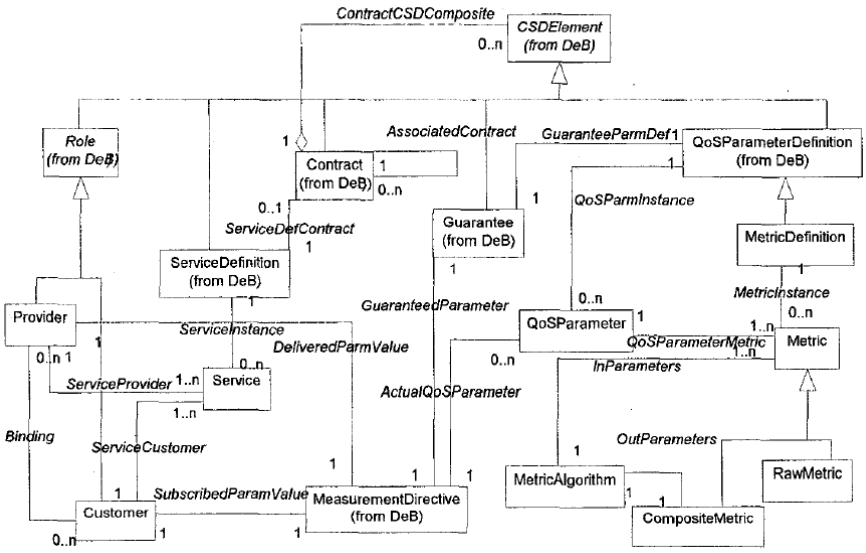

FIG. 3.14 – modèle de contrat pour des services dynamiques

WSLA. WSLA [LKD+03] est un langage de description de SLA pour les Web services conçu par les mêmes chercheurs que le précédent modèle mais plus évolué. A nouveau le contrat est composé de 3 parties (figure 3.15). Une partie contient les participants au contrat, qui en plus des clients et du fournisseur, comprend des services tiers intervenant dans la vérification des clauses du contrat. Une seconde partie du contrat contient l'objet du service contraint sous forme d'un ensemble de paramètres de qualité de service, qui peuvent être mesurés par des participants tiers. Enfin le contrat présente un ensemble d'obligations portant sur ces paramètres qui peuvent être évaluées par des participants tiers. Il faut noter que les obligations sont non seulement des contraintes sur les paramètres mais aussi des obligations d'actions. Par ailleurs ces obligations peuvent être associées à n'importe quel participant du contrat,

client ou fournisseur, le contrat exprime donc plus que la satisfaction d'un service à une spécification. On peut aussi remarquer que dans [KL03b], le cycle de vie complet du contrat est détaillé, depuis sa négociation jusqu'à sa terminaison.

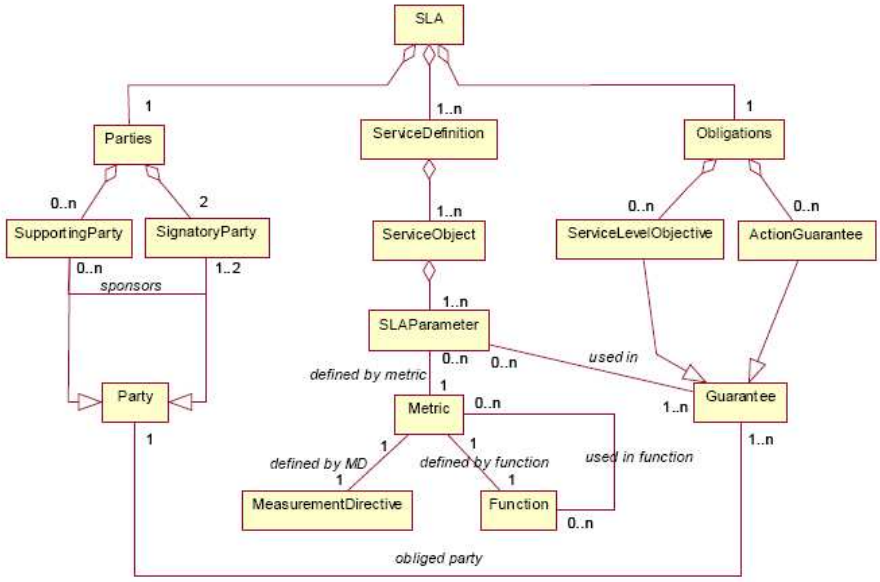

FIG. 3.15 – modèle de contrat WSLA

WS-Agreement. Le WS-Agreement [AKCK+05] est un langage basé sur XML destiné à exposer ce qu'un fournisseur de service propose, à permettre la passation d'un accord sur cette base et le monitoring de celui-ci à l'exécution du service. Un `agreement` contient trois parties. La première nommée `context` spécifie les participants de l'accord mais aussi entre autres sa durée. La seconde, les `Service Description Terms` décrit en particulier des grandeurs mesurables associées au service, par exemple un temps de réponse. Finalement la troisième partie d'un `agreement` contient les `Guarantee Terms` qui sont les contraintes sur les propriétés du service, éventuellement conditionnées par des `Qualifying Conditions`. C'est la même syntaxe qui est utilisé pour décrire ce qu'un fournisseur propose, un `agreement offer`, et ce que l'accord prévoit que le service effectue, l'`agreement`. Par exemple, dans le premier document peuvent se trouver des alternatives qui sont résolues pour aboutir au second document. Les travaux [AFM05] proposent une formalisation des WS-Agreements. En particulier les auteurs modélisent à l'aide d'un automate les différents états de l'`agreement` sur la base d'une formalisation de celui-ci. En effet l'`agreement` est défini comme composé d'un ensemble de termes, chacun combinaison d'un service et d'une garantie, ces derniers ayant leurs états propres dont on déduit celui du terme. Par ailleurs les auteurs proposent des termes de l'accord réagissant à l'ouverture de ce dernier pour en modifier d'autres et ainsi le rétablir. Le cycle de vie de l'accord est ainsi celui décrit dans la figure 3.16, un état `Revisited` représente l'accord modifié après rupture. Ainsi même si ce type de contrat ne porte que sur la satisfaction d'une spécification par un service, il est intéressant car présente des clauses de renégociation qui portent sur lui même et explicite son cycle de vie.

Contrats et composition. Dans [MSM04] les auteurs considèrent les contrats comme des outils de spécification des Web services. Ils proposent un langage de description des contrats, réifiant un large éventail de propriétés des services. Celles ci comprennent la description fonctionnelle du service, son comportement (pré et post conditions), ses performances, sa fiabilité, etc. L'intérêt de cette approche est qu'elle propose une méthodologie pour déduire le contrat d'une composition de services des contrats des services individuels. Ils traduisent pour ce

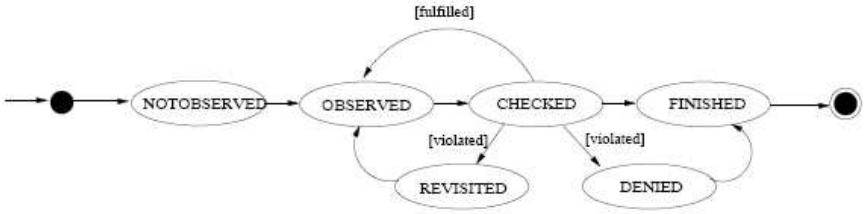

FIG. 3.16 – cycle de vie de WS-Agreement étendu

faire les contrats en machines B (figure 3.17) dont les termes, c'est à dire les propriétés du contrat, supportent des annotations décrivant leur évolution en cas de composition. Par la suite, pour chaque opérateur de composition, on définit pour chaque propriété, le résultat de sa composition suivant son annotation, afin de déduire la machine B résultante, et par suite le contrat, de la composition. Diverses vérifications doivent être effectuées pour valider la compatibilité des contrats, la satisfaction des invariants par celui résultant etc. Le trait remarquable de cette approche est qu'elle définit d'une part des opérations de composition sur les contrats, et d'autre part qu'elle établit une correspondance entre les opérations sur l'architecture et celles sur les contrats. Par ailleurs les opérateurs de composition ne sont pas limités, leur sémantique étant définie par rapport à l'évolution des propriétés composées.

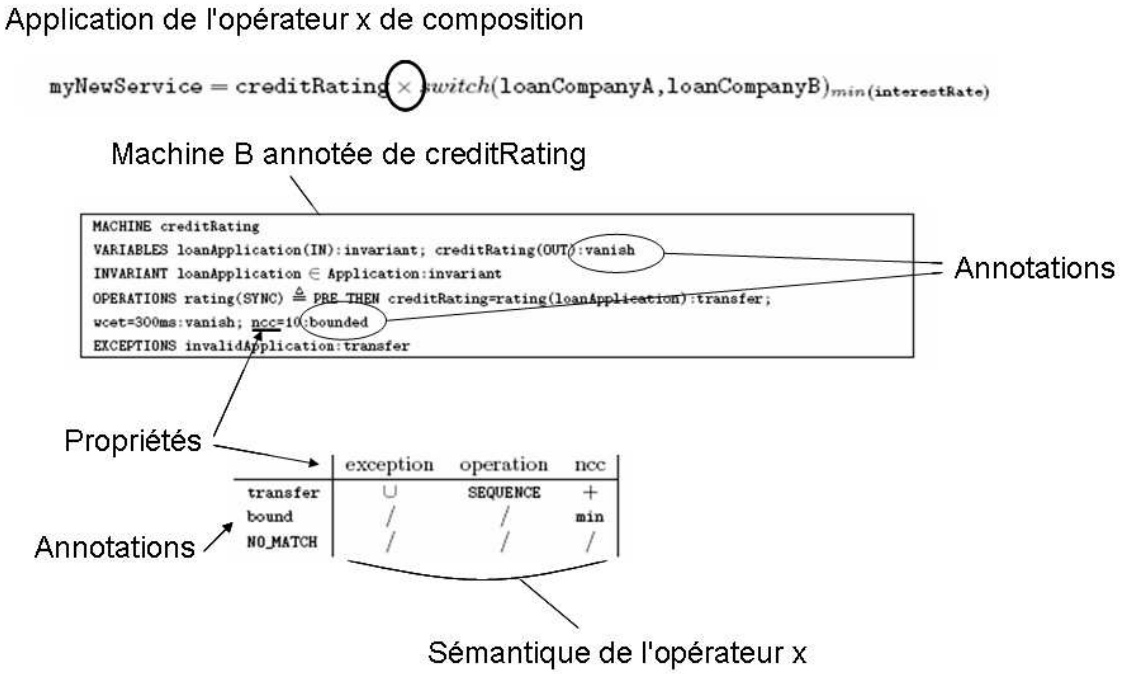

FIG. 3.17 – méthodologie de composition des contrats

Contrats et politiques. Dans [PG04] les contrats contraignent les relations entre les objets exposés par les services de processus métiers. Ils sont composés (figure 3.18) de participants qui y jouent des rôles (`Who`), de leurs objets qui sont les relations entre ce qu'exposent les participants (`What`), et des contraintes à vérifier sur ces relations pour que le contrat soit rempli (`How`). Il faut toutefois noter que les contraintes à vérifier sont exprimées sous formes de règles ECA associées à la relation qui doivent être exécutées pour que celle-ci soit satisfaite. Par exemple, une relation spécificiant qu'un service de réservation de billet (SRB) émet un billet est :

"émettre (SRB, billet)"

cette relation est contrainte sur sa date par :

Evénement : émettre (SRB, billet)

Contrainte : date (Evénement) < date_limite

Action : autoriser (émettre (SRB, billet))

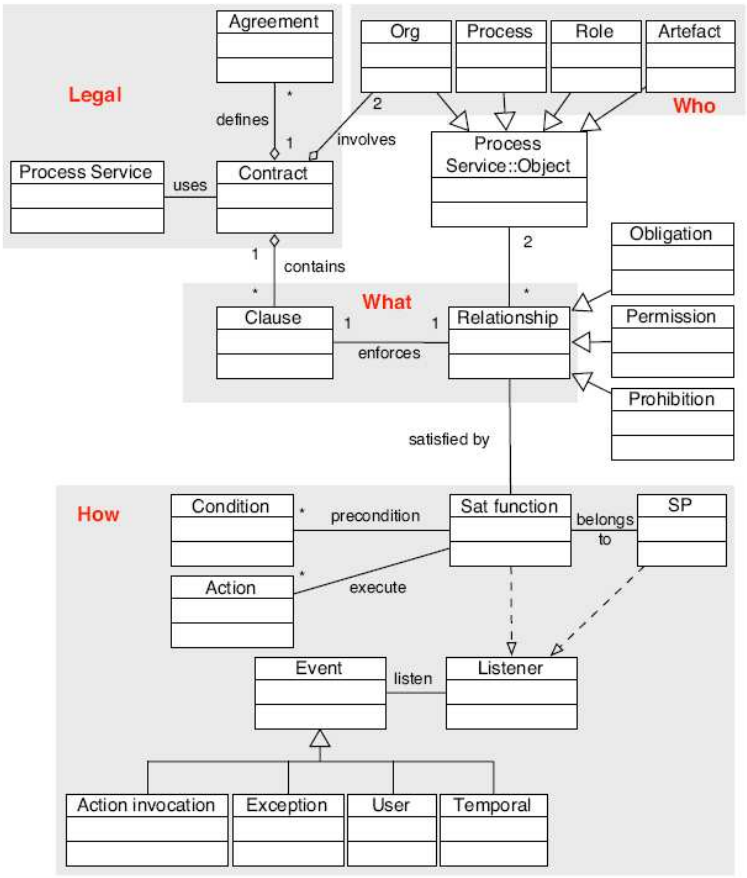

FIG. 3.18 – contrats et politiques

Ces contrats expriment la validité de relations, et donc la compatibilité, au sens large entre services et/ou objets des services. Mais ils se rapprochent plutôt de politiques car la vérification des relations guide des actions à effectuer sur le système.

Discussion
Par rapport aux contrats sur les objets et composants, ceux sur les services réifient clairement les rôles et les responsabilités des participants. Les cycles de vie de ces contrats sont aussi beaucoup plus clairement définis et des métriques de grandeurs contraintes preque systématiquement définies. Par contre, ils ne sont que rarement orientés vers la vérification de la compatibilité entre des services au sein d'une composition.

3.6.4 Agents

Par rapport aux différentes formes de contrat sur les objets, composants, services les contrats sur les agents sont ceux qui sans doute se rapprochent le plus de ceux de la vie quotidienne. Leurs formalisations réifient les acteurs du contrat et les clauses, en donnant à ces dernières

une sémantique déontique d'obligation, permission etc. Par ailleurs on peut noter que certaines distinguent et intègrent les contraintes sur les états du système de celles sur les actions des acteurs [KN02], [Bv97]. Plusieurs explicitent aussi le cycle de vie du contrat. Ainsi la durée d'application d'une clause est décrite dans [KN02], tandis que la composition même du contrat peut changer au cours du temps dans [DMF+02]. D'autres travaux expriment la différence entre violation de contrat et exception ou incorrection informatique [GvdT05]. Enfin, une algèbre de contrats est proposée par [Bv97], basée sur un formalisme capable d'exprimer que les agents assurent des garanties et comptent sur des hypothèses.

3.6.5 Bilan

On peut remarquer que les modèles de contrat ont été développés en rapport avec les préoccupations centrales des entités qu'ils contraignaient. Ainsi, les contrats appliqués aux objets réifient essentiellement des contraintes sur leurs comportements et interactions, mais la notion de garantie n'est pas clairement explicitée. Par contre les modèles de contrat appliqués aux services réifient clairement les responsabilités ainsi qu'un ensemble très divers de propriétés aussi bien fonctionnelles que non fonctionnelles auxquelles le service doit adhérer. Les modèles de contrat appliqués aux composants adressent le problème de la garantie de leur composition bien qu'ils ne la contraignent pas toujours explicitement. Il s'agit en effet parfois d'évaluer la compatibilité de contrats portés par des composants associés et non de confronter au sein d'un contrat leurs spécifications. Ils présentent toutefois souvent à des degrés variables des considérations architecturales. En effet la notion de propriété requise et fournie associée au paradigme composant facilite l'appréhension de leur collaboration en termes d'échanges et donc de contrat. Globalement, chacun des modèles de contrat réifie des principes nécessaires à la garantie de son objet et à son intégration dans l'architecture qu'il contraint, mais aucun ne les regroupe tous. Il est par ailleurs rare de trouver des clauses de « second ordre » c'est à dire contraignant le contrat lui même, comme par exemple le conditionnement d'une clause par la réalisation d'une autre.

Bibliographie

[AAV02] Sahai A., Durante A., and Machiraju V. Towards automated sla management for web services. Technical report, 2002.

[Abr96] Jean-Raymond Abrial. *The B-Book : Assigning Programs to Meanings*. Cambridge University Press (Cambridge U.K.), 1996.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava : connecting software architecture to implementation. In *Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 187–197. ACM, 2002.

[AE03] D. Amyot and A. Eberlein. An evaluation of scenario notations and construction approaches for telecommunications systems development. *Telecommunications Systems Journal 24(1)*, 2003.

[AF99] Luis Filipe Andrade and José Luiz Fiadeiro. Interconnecting objects via contracts. In *UML*, pages 566–583, 1999.

[AFM05] Marco Aiello, Ganna Frankova, and Daniela Malfatti. What's in an agreement ? an analysis and an extension of WS-agreement. In *ICSOC*, pages 424–436, 2005.

[AG97] R. J. Allen and D. Garlan. A formal basis for architectural connection. *ACM, Transactions on Software Engineering and Methodology*, 6, July 1997.

[AKCK+05] A. Andrieux, A. Dan K. Czajkowski, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement). Specification draft, Global Grid Forum (GGF), September Version 09/2005.

[AL90] M. Abadi and L. Lamport. Composing specifications. Research report 66, Digital, 1990.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3) :507–534, 1995.

[AVM+02] Sahai A., Machiraju V., Sayal M., van Moorsel A., and Casati F. Automated sla monitoring for web services. In *13th IFIP/IEEE International Workshop on Distributed Systems : Operations and Management, DSOM*, pages 28–41, Montreal, Canada, 2002. Springer-Verlag.

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

[BBB+00] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon Software Engineering Institute, May 2000. Volume 2.

[BBL+99] Béatrice Bérard, Michel Bidoit, François Laroussine, Antoine Petit, and Philippe Schoebelen. *Vérification de logiciels – Techniques et outils du model-checking*. Vuibert Informatique, 1999.

[BCH05] Dirk Beyer, Arindam Chakrabarti, and Thomas A. Henzinger. Web service interfaces. In *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, pages 148–159. ACM, 2005.

[BCL+04] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J-B Stefani. An Open Component Model and Its Support in Java. In *ICSE'2004 - CBSE7*. Springer - LNCS, 2004.

[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.

[BG98] Christian R. Becker and Kurt Geihs. Quality of service - Aspect of distributed programs. In *ICSE'1998*, Kyoto, Japan, 1998.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *IEEE Computer*, 32(7) :38–45, July 1999.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system : An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices : International Workshop, CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer Verlag, January 2005.

[BRBV04] Yolande Berbers, Peter Rigole, Stefan Van Baelen, and Yves Vandewoude. Components and contracts in software development for embedded systems. In *First European Conference on the Use of Modern Information and Communication Technologies*, pages 219–226, April 2004.

[BS03] Michael Barnett and Wolfram Schulte. Runtime verification of .net contracts. *Journal of Systems and Software*, 65(3) :199–208, 2003.

[Bv97] Ralph-Johan Back and Joakim Wright von. Contracts, games and refinement. Technical Report TUCS-TR-138, Turku Centre for Computer Science, Finland, 1997.

[CC05] H. Chang and P. Collet. Fine-grained Contract Negotiation for Hierarchical Software Components. In *EUROMICRO-SEAA'2005*. IEEE Computer, 2005.

[CC06] Hervé Chang and Philippe Collet. Eléments d'architecture pour la négociation de contrats extrafonctionnels. In *Première Conférence francophone sur l'Architecture Logicielle (CAL'2006)*. Hermes Science, 2006. A paraître.

[CFK+02] K. Czajkowski, I. Foster, C. Kesselman, V. Sander, and S. Tuecke. SNAP : A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*. Springer-Verlag LNCS, 2002.

[CHK05] P. Combes, D. Harel, and H. Kugler. Modeling and verification of a telecommunication application using live sequence charts and the play-engine tool. In *Proc 3rd Int Symp on Automated Technology for Verification and Analysis (ATVA05), LNCS 3707*, 2005.

[COR06a] Philippe Collet, Alain Ozanne, and Nicolas Rivierre. Enforcing different contracts in hierarchical component-based systems. In *Software Composition (SC'06)*, Lecture Notes in Computer Science. Springer Verlag, 2006.

[COR06b] Philippe Collet, Alain Ozanne, and Nicolas Rivierre. On contracting different behavioral properties in component-based systems. In *ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27*, 2006.

[CRCR05] P. Collet, R. Rousseau, T. Coupaye, and N. Rivierre. A Contracting System for Hierarchical Components. In *ICSE'2005 - CBSE8*. Springer - LNCS, 2005.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120, 2001.

[DBL05] Wojciech J. Dzidek, Lionel C. Briand, and Yvan Labiche. Lessons learned from developing a dynamic ocl constraint enforcement tool for java. In *OCL Workshop (MoDELS Satellite Events)*, volume 3844 of *Lecture Notes in Computer Science*, pages 10–19. Springer, 2005.

[DC01] Jim Dowling and Vinny Cahill. The k-component architecture meta-model for self-adaptive software. In *Metalevel Architectures and Separation of Crosscutting Concerns, Third International Conference, REFLECTION 2001, Kyoto, Japan, September 25-28, 2001, Proceedings*, volume 2192 of *Lecture Notes in Computer Science*, pages 81–88. Springer, 2001.

[DDK+04] A. Dan, D. Davis, R. Kearney, R. King, A. Keller, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web Services on demand : WSLA-driven Automated Management. *IBM Systems Journal, Special Issue on Utility Computing*, 43 :136–158, March 2004.

[DFF+99] D. Deveaux, R. Fleurquin, P. Frison, J.-M. Jézéquel, and Y. Le Traon. Composants objets fiables : une approche pragmatique. *L'objet*, 5(3-4) :469–494, December 1999.

[DH01] W. Damm and D. Harel. Breathing life into message sequence charts. *Formal Methods in System Design 19(1)*, 2001.

[DJP04] Olivier Defour, Jean-Marc Jézéquel, and Noël Plouzeau. Extra-functional contract support in components. In *Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings*, volume 3054 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2004.

[DKR00] D. Distefano, J. Katoen, and A. Rensink. Towards model checking OCL. In *Proceedings, ECOOP Workshop on Defining Precise Semantics for UML*, June 2000.

[DMF+02] Virginia Dignum, John-Jules Ch. Meyer, Frank, Dignum, and Hans Weigand. Formal specification of interaction in agent societies. In *FAABS*, pages 37–52, 2002.

[DTV99] Jérôme Daniel, Bruno Traverson, and Sylvie Vignes. Integration of quality of service in distributed object systems. In *International Working Conference on Distributed Applications and Interoperable Systems (DAIS)*, volume 143 of *IFIP Conference Proceedings*, pages 31–44. Kluwer, 1999.

[DW98] Desmond F. D'Souza and Alan C. Wills. *Object, Components and Frameworks with UML : The Catalysis Approach*. Addison-Wesley Publishing Co. (Reading, MA), 1998.

[FK98] Svend Frølund and Jari Koistinen. Quality of service in distributed object systems design. In *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), Santa Fe (New Mexico)*. USENIX, April 27-30 1998.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proceedings American Mathematical Society Symposium in Applied Mathematics*, volume 19, pages 19–31, 1967.

[FM02] Stephan Flake and Wolfgang Mueller. A UML profile for real-time constraints with the OCL. In *UML 2002 - The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 150–171. Springer Verlag, October 2002.

[FUMK03] Howard Foster, Sebastián Uchitel, Jeff Magee, and Jeff Kramer. Model-based verification of web service compositions. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*, pages 152–163. IEEE Computer Society, 2003.

[Glo] Global Grid Forum (Web Site). http ://www.gridforum.org/.

[GvdT05] Christophe Garion and Leendert van der Torre. Design by contract deontic design language for multiagent system. In *ANIREM*, 2005.

[HDF00] H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In *UML'2000 International Conference*, October 2000.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts : Specifying Behavioral compositions in Object-Oriented Systems. In Norman Meyrowitz, editor, *OOPSLA/ECOOP'90*, pages 169–180, Ottawa, Canada, October 1990.

[HKMP02] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioural requirements. In *Proc 4rd Int Symp on Formal Methods in Computer-Aided Design (FMCAD02), LNCS 2517*, 2002.

[HKW05] D. Harel, H. Kugler, and G. Weiss. Some methodological observations resulting from experience using lscs and the play-in/play-out approach. In *Proc Scenarios : Models, Algorithms and Tools, LNCS 3466*, 2005.

[HLJ04] P. C. K. Hung, H. Li, and J-J Jeng. WS-Negotiation : An Overview of Research Issues. In *Proc. of Hawaii International Conference on System Sciences (HICSS)*, 2004.

[HLS04] George T. Heineman, Joseph P. Loyall, and Richard E. Schantz. Component technology and qos management. In *Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings*, volume 3054 of *Lecture Notes in Computer Science*, pages 249–263. Springer, 2004.

[HM02] D. Harel and R. Marelly. Playing with time : on the specification and execution of time-enriched lscs. In *Proc 10rd IEEE/ACM International Symposium on Modeling, Analysis and simulation of Computer and Telecommunication Systems (MASCOT 2002)*, 2002.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, October 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[ISO95] ISO/IEC. Open distributed processing reference model - parts 1,2,3,4. Technical report, ISO 10746-1,2,3,4 or ITU-T X.901,2,3,4, 1995.

[ITU99] ITU. MSC : Message sequence charts, 1999.

[JDP03] Jean-Marc Jézéquel, Olivier Defour, and Noël Plouzeau. An mda approach to tame component based software development. In *Formal Methods for Components and Objects : Second International Symposium, FMCO 2003*, pages 260–275, 2003.

[JW05] H. Jin and H. Wu. Semantic-enabled Specification for Web Services Agreement. *International Journal of Web Services Practices*, 1(1-2), 2005.

[KA01] Raphael Simon Karine Arnout. the .net contract wizard : Adding design by contract to languages other than eiffel. In *TOOLS 39*, pages 14–23. IEEE Computer Society, 2001.

[KAB+06] J. Kofron, J. Adamek, T. Bures, P. Jesek, V. Mencl, P. Parizek, and F. Plasil. Checking fractal component bahaviour using behaviour protocols. In *5th Fractal Workshop, ECOOP 2006, Nantes, France*, 2006.

[KKL+02] A. Keller, G. Kar, H. Ludwig, A. Dan, and J.L. Hellerstein. Managing Dynamic Services : A Contract Based Approach to a Conceptual Architecture. In *Proc. of the 8th IEEE/IFIP Network Operations and Management Symposium (NOMS 2002)*, Florence, Italy, April 15-19 2002.

[KL02] A. Keller and H. Ludwig. Defining and Monitoring Service Level Agreements for dynamic e-Business. In *Proc. of the 16th USENIX Systems Administration Conference (LISA'02)*, Philadelphia, PA, November 3-8 2002.

[KL03a] A. Keller and H. Ludwig. The WSLA Framework : Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11(1), 2003.

[KL03b] Alexander Keller and Heiko Ludwig. The wsla framework : Specifying and monitoring service level agreements for web services. *Journal of Network and System Management*, 11(1), 2003.

[KMJ02] Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An analyzable annotation language. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 231–245. ACM Press, 2002.

[KN02] Martin J. Kollingbaum and Timothy J. Norman. Supervised interaction - a form of contract management to create trust between agents. In *Trust, Reputation, and Security*, pages 108–122, 2002.

[Kra98] Reto Kramer. iContract - the Java design by contract tool. In *International Conference on Technology of Object-Oriented Languages and Systems (Tools 26, USA'98)*. IEEE Computer Society Press (New York), Aug. 1998.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML : A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.

[LCL06] Marc Léger, Thierry Coupaye, and Thomas Ledoux. Contrôle dynamique de l'intégrité des communications dans les architectures à composants. In *LMO'2006 (Langages et Modèles à Objets)*, 2006.

[LDK04] H. Ludwig, A. Dan, and B. Kearney. Cremona : An Architecture and Library for Creation and Monitoring of WS-Agreements. In *International Conference on Service Oriented Computing (ICSOC)*. ACM Press, 2004.

[Leb98] L. Leboucher. *Algorithmique et modélisation pour la qualité de service des systèmes répartis temps réel*. PhD thesis, Ecole Nationale Supérieure de Télécoms, Paris, 1998.

[LKD+03] H. Ludwig, A. Keller, A. Dan, R. P. King, and R. Franck. WSLA Language Specification, Version 1.0. Technical Report, IBM Corporation, January 2003. http ://www.research.ibm.com/wsla/.

[LN97] L. Leboucher and E. Najm. A framework for real-time qos in distributed systems. *IEEE Workshop on Middleware for Distributed Real-Time Systems and Service, San Francisco, California*, December 1997.

[LPJ98] Stéphane Lorcy, Noël Plouzeau, and Jean-Marc Jézéquel. Reifying quality of service contracts for distributed software. In *26th Conference on Technology of Object-Oriented Systems (TOOLS USA'98)*, August 1998.

[LS02] Hung Ledang and Jeanine Souquières. Integration of UML and B specification techniques : Systematic transformation from OCL expressions into B. In *APSEC'2002*, December 2002.

[LSG02] Nicolas Le Sommer and Frédéric Guidec. JAMUS, une plate-forme sécurisée pour le code mobile. In *LMO'2002 (Langages et Modèles à Objets)*, pages 203–215. Hermes Science Publications, *L'objet*, volume 8, numéro 1-2/2002, Janvier 2002.

[LSZB98] Joseph P. Loyall, Richard E. Schantz, John A. Zinky, and David E. Bakken. Specifying and measuring quality of service in distributed object systems. In *1st International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98), 20-22 April 1998, Kyoto, Japan*, pages 43–52. IEEE Computer Society, 1998.

[Mag99] Jeff Magee. Behavioral analysis of software architectures using ltsa. In *ICSE*, pages 634–637, 1999.

[Mey91] Bertrand Meyer. *Eiffel : The Language*. The Object-Oriented Series. Prentice Hall Inc., 1991. Second revised printing, 1992.

[Mey92] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10) :40–51, October 1992.

[MKG99] Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. Behaviour analysis of software architectures. In *WICSA*, pages 35–50, 1999.

[MM02] S. Michael and S. Markus. A Matchmaking Component for the Discovery of Agreement and Negotiation Spaces in Electronic Markets. *Group Decision and Negotiation*, 11, 2002.

[MPJ02] Jacques Malenfant, Noël Plouzeau, and Jean-Marc Jézéquel. The design of gccl : a generalized common contract language. Technical Report RR-4502, INRIA, Rennes, July 2002.

[MSM04] Nikola Milanovic, Vladimir Stantchev, and Miroslaw Malek. Definition of contracts for service composition. In *The Second International Conference on Pervasive Computing, PERVASIVE 2004*, 2004.

[Obj97] Object Management Group, Inc. Object constraint language specification. Technical Report version 1.1, ad/97-08-08, IBM `www.software.ibm.com/ad/ocl`, September 1997.

[OMG02] OMG. UML v2, 2002.

[OMG03] OMG. UML 2 OCL final adopted specification. Technical Report ptc/03-10-14, Object Management Group, October 2003.

[OPD04] A. Occello and Anne-Marie Pinna-Dery. An adaptation-safe model for component platforms. In *Proceedings of the ISCA 13th International Conference on Intelligent and Adaptive Systems and Software Engineering, Nice, France, July 1-3, 2004*, pages 169–174. ISCA, 2004.

[PBP99] Luigia Petre, Ralph-Johan Back, and Ivan Paltor. Analysing uml use cases as contracts. In *UML*, pages 518–533, 1999.

[PG04] Oliver Perrin and Claude Godard. An approach to implement contracts as trusted intermediaries. In *The First Intl. Workshop on Electronic Contracting (WEC04)*, San Diego, CA, 2004.

[Plö02] Reinhold Plösch. Evaluation of Assertion Support for the Java Programming Language. In *Journal of Object Technology, Special issue : TOOLS USA 2002 proceedings*, volume 1,3, pages 5–17, 2002.

[PV02] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), November 2002.

[Rau00] Andreas Rausch. Software evolution in componentware using requirements/assurances contracts. In *ICSE*, pages 147–156, 2000.

[Reu03] Ralf Reussner. Automatic component protocol adaptation with the coconut/j tool suite. *Future Generation Comp. Syst.*, 19(5) :627–639, 2003.

[RG99] Mark Richters and Martin Gogolla. A metamodel for ocl. In *UML*, pages 156–171, 1999.

[RZ03] Simone Röttger and Steffen Zschaler. CQML$^+$: Enhancements to CQML. In J.-M. Bruel, editor, *Proc. 1st Int'l Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France*, pages 43–56. Cépaduès-Éditions, June 2003.

[SAD+02] A. Sassen, G. Amoros, P. Donth, K. Geihs, J. Jézéquel, K. Odent, N. Plouzeau, and T. Weis. Qccs : A methodology for the development of contract-aware components based on aspect-oriented design. In *Workshop on Early Aspects : Aspect-Oriented Requirements Engineering and Architecture Design (AOSD-2002)*, Enschede, The NetherLands, mar 2002.

[SLE02] J. Skene, D. Lamanna, and W. Emmerich. SLAng : A Language for Service Level Agreements. In *Proc. of Workshop on Future Trends in Distributed Computing Systems*. IEEE Computer Society, 2002.

[SLE04] J. Skene, D. Lamanna, and W. Emmerich. Precise Service Level Agreements. In *Proc. of International Conference on Software Engineering (ICSE)*, pages 179–188. IEEE Computer Society, 2004.

[SMS+02a] A. Sahai, V. Machiraju, M. Sayal, L. Jie Jin, and F. Casati. Automated SLA Monitoring for Web Services. Technical Report, HP Laboratories, January 2002.

[SMS+02b] A. Sahai, V. Machiraju, M. Sayal, A. P. A. van Moorsel, and F. Casati. Automated SLA Monitoring for Web Services. In *Proc. of International Workshop on Distributed Systems : Operations and Management (DSOM)*, volume 2506 of *Lecture Notes in Computer Science*, pages 28–41. Springer, 2002.

[TBD+04] Hanh-Middi Tran, Philippe Bedu, Laurence Duchien, Hai-Quan Nguyen, and Jean Perrin. Toward structural and behavioral analysis for component models. In *SAVBCS 2004, 12th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, NewPort Beach, California, USA, November 2004.

[TFS05] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. Nfrs-aware architectural evolution of component-based software. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 388–391. ACM, 2005. short paper.

[TPP02] V. Tosic, K. Patel, and B. Pagurek. WSOL - Web Service Offerings Language. In *International Workshop on Web Services, E-Business, and the Semantic Web (CAiSE/WES)*. Springer-Verlag, 2002.

[VAvMA02] Machiraju V., Sahai A., and van Moorsel A. Web services management network : An overlay network for federated service management. Technical report, 2002.

[VTS02] Gary Vecellio, William M. Thomas, and Rob Sanders. Containers for predictable behavior of component-based software. In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering : Benchmarks for Predictable Assembly*, Orlando, FL, May 2002.

[yA01] Jan Øyvind Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.

[yAA01] Jan Øyvind Aagedal Aag01. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.

[ZG02] Paul Ziemann and Martin Gogolla. An extension of OCL with temporal logic. In *Workshop on Critical Systems Development with UML*, September 2002.

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.

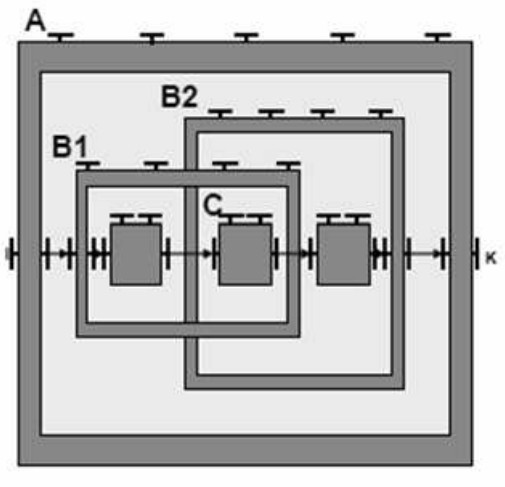

FIG. 4.1 – Composants Composites. Les rectangles noir représentent la membrane du composant, i.e. le contrôle du composant. Les formes en T associées aux rectangles correspondent aux interfaces internes ou externes du composant. Lorsque les interfaces externes sont positionnées au dessus du composants,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

CHAPITRE 5

Ingénierie des modèles

Coordonnateur : Noël Plouzeau.
Rédacteurs : Mireille Blay-Fornarino, Franck Chauvel, Franck Fleurey, Philippe Lahire, Pierre-Alain Muller.

5.1 Introduction

5.1.1 Rôle de l'ingénierie des modèles dans FAROS

Dans le contexte du projet FAROS, les méthodes et les techniques de l'ingénierie des modèles permettront de relier
– les modèles centrés sur la description des services (voir chapitre 2) ;
– les modèles centrés sur la description des plates-formes d'exécution.
L'une des préoccupations principales de FAROS est la spécification et le traitement des contrats sur des composants logiciels. L'ingénierie dans le contexte de FAROS aura pour fonction principale la traduction des contrats de services en contrats évaluables dans des plates-formes de composants. L'ingénierie des modèles a été adoptée dans FAROS comme moyen et non comme but, face à la situation désormais classique de la recherche de l'indépendance maximale entre problèmes métiers et problèmes technologiques.

5.1.2 Rôle général des modèles et de leur transformation

L'ingénierie dirigée par les modèles (IDM) [EFB+05] propose des pistes pour permettre aux organisations de surmonter la mutation des exigences du développement de logiciel. L'ingénierie des modèles est une forme d'ingénierie générative, par laquelle tout ou une partie d'une application informatique est générée à partir de modèles. Les idées de base de cette approche sont voisines de celles de nombreuses autres approches du génie logiciel, comme la programmation générative, les langages spécifiques aux domaines ((DSL pour *domain-specific language*) [vDKV00, CM98], le MIC (*Model Integrated Computing*), les usines à logiciels (Software Factories) [GSC+04], etc.
Dans cette nouvelle perspective, les modèles occupent une place de premier plan parmi les artefacts de développement des systèmes, et doivent en contrepartie être suffisamment précis afin de pouvoir être interprétés ou transformés par des machines. Le processus de développement des systèmes peut alors être vu comme un ensemble de transformations de modèles partiellement ordonné, chaque transformation prenant des modèles en entrée et produisant des modèles en sortie, jusqu'à obtention d'artefacts exécutables.
Pour donner aux modèles cette dimension opérationnelle, il est essentiel de spécifier leurs points de variations sémantiques, et donc aussi de décrire de manière précise les langages utilisés pour les représenter. On parle alors de métamodélisation.
L'intérêt pour l'ingénierie dirigée par les modèles a été fortement amplifié, en novembre 2000, lorsque l'OMG (*Object Management Group*) a rendu publique son initiative MDA™ [StOs00a] qui définit un cadre de normalisation pour l'IDM. Il existe cependant bien d'autres alternatives technologiques aux normes de l'OMG, par exemple *Ecore* dans le domaine technolo-

gique d'Eclipse, mais aussi les grammaires, les schémas de bases de données, les schémas XML. En somme, l'ingénierie des modèles dépasse largement le MDA, pour se placer plutôt à la confluence de différentes disciplines.
L'ingénierie des modèles peut jouer un rôle de médiation entre différentes disciplines, telles que la théorie des langages, la programmation générative, les bases de données, etc.
Ce chapitre s'articule en cinq parties. Les principaux domaines d'application de l'ingénierie des modèles sont présentés en premier lieu. Une deuxième partie présente différents méta-métamodèles qui vont permettre de décrire et de manipuler les modèles et les métamodèles. La troisième partie propose ensuite une classification des transformations de modèles pour séparer les différentes transformations qui vont venir outiller un métamodèle particulier. La quatrième partie se concentre sur l'outillage de métamodèles particuliers pour aboutir à des DSLs permettant à l'utilisateur de manipuler directement les concepts de son métier. Enfin, la cinquième et dernière section présente différentes techniques de validation de transformations de modèles.

5.1.3 Terminologie

Dans la suite de ce chapitre, nous utiliserons les définitions suivantes, extraites et adaptées du document *L'ingénierie dirigée par les modèles* [EFB+05].

Modèle Une description abstraite d'un système réel ou non. L'abstraction réalisée dépend des objectifs, mais l'un des objectif est l'équivalence simplifiée : pour une certaine catégorie de propriétés, le modèle possède les même propriétés que le système qu'il abstrait.

Langage Un ensemble de modèles.

Métamodèle Un modèle qui définit un langage pour exprimer des modèles.

Conformité Un modèle est conforme à un métamodèle s'il appartient à un langage défini par ce métamodèle.

5.1.4 Verrous scientifiques

Pour pouvoir gérer la complexité d'un système réel, l'ingénierie des modèles propose de raisonner sur des modèles pour éviter la complexité de la réalité. Un modèle est ici perçu comme une abstraction de la réalité, où la réalité est restreinte en fonction d'un point de vue particulier. Dans un système logiciel par exemple, les aspects liés à la performance ou à la qualité pourront faire l'objet de modèles spécifiques.
Dans ce cadre, la définition est un point critique de l'ingénierie des modèles. La définition des modèles nécessite de spécifier clairement les objectifs de la modélisation. Quelles sont les entités de la réalité à représenter ? Avec quel niveau de détail faut-il décrire la réalité ? Cependant, pour être exploitable, un modèle doit assurer un certain niveau de qualité : cohérence des données, complétude des données, sémantique, etc. La validation des modèles est donc un autre point critique de l'ingénierie des modèles.
Pour un même point de vue ou un même objectif, les modèles peuvent prendre plusieurs formes : graphiques, descriptions formelles, descriptions textuelles, etc. Manipuler les modèles requiert alors de définir les concepts principaux associés à la problématique sous-jacente. Les métamodèles vont définir ces concepts, qui permettent de raisonner sur les modèles et de construire un certain nombre d'outils adéquats.
La définition des métamodèles est ainsi au cœur de l'ingénierie des modèles et engendre un grand nombre de langages permettant de décrire les différents aspects de la réalité : modèle logique, modèle conceptuel, modèle physique, etc. Cette multitude de langages est imposée par la diversité des points de vue possibles et l'un des défis associés à l'ingénierie des modèles

est donc de gérer cette multiplication des langages. L'ingénierie des modèles doit donc fournir l'outillage nécessaire pour :
– comparer les modèles et les métamodèles entre eux et prouver leurs équivalences ;
– transformer les modèles pour passer d'un langage à un autre ;
– fusionner les modèles et les métamodèles pour intégrer les différents aspects d'une même réalité.

La promesse de l'ingénierie des modèles est donc de capitaliser sur les modèles et les métamodèles. L'emploi effectif de l'ingénierie des modèles ne sera toutefois envisageable que lorsque certaines questions auront été traitées :
– estimation du coût de développement des métamodèles (et des modèles) ;
– estimation de l'impact de l'ingénierie des modèles sur le cycle de développement en terme de productivité, de réactivité, de maintenance, etc. ;
– étude des procédés d'intégration d'architectures existantes dans un procédé d'ingénierie des modèles.

Dans le contexte du projet FAROS, le concept de contrat constitue la pierre angulaire des systèmes que le projet construira. La principale difficulté concernera très vraisemblablement la transition d'un métamodèle de contrat à l'autre. Plus précisément, comment sera-t-il possible de transformer des contrats exprimés dans un modèle de services métiers en des contrats adaptés à la supervision par des composants logiciels fondés sur telle ou telle technologie ?

5.1.5 Rapide panorama de l'ingénierie des modèles

Il existe actuellement de multiples langages dans le paysage de l'ingénierie des modèles. Une taxonomie de ces langages peut être organisée autour de quatre axes principaux :
– les langages de métadonnées avec par exemple Ecore d'IBM [BEG+03], EMOF de l'OMG [OMG06], ou les schéma XML du W3C [W3C04] ;
– les langages de transformation QVT issu de l'OMG [OMG05], ATL issu de l'INRIA [JK05], GreaT issu de l'université Vanderbilt NashVille, USA [gre03] ou XSLT issu du W3C [W3C05b] ;
– les langages de requête avec par exemple OCL issu de l'OMG [OMG03] ou XQUERY issu du W3C [W3C05a] ;
– les langages d'actions *Action Semantics* [OMG01] issu de l'OMG, Xion issu de Objexion™ [MSFB05].

Le langage Kermeta [MFJ05b] est un langage de métaprogrammation et pas uniquement un langage de métadonnées, comme Ecore ou eMOF. De même que Niklaus Wirth définissait les programmes comme des structures de données et des algorithmes, Kermeta présente les métamodèles comme des métadonnées et des actions[1]. Cette approche permet d'aller au-delà de la seule syntaxe et de pouvoir spécifier également la sémantique opérationnelles des métamodèles. Kermeta propose ainsi à la fois un langage de métamodélisation, un langage d'actions. Ce langage d'actions peut être utilisé aussi comme langage de transformation. Il a une syntaxe proche de Java mais permet de travailler directement des instances de modèles, et de manipuler les concepts définis au niveau du métamodèles comme des types de base.

5.2 Usages dans l'IDM

L'objectif de cette section est de présenter les principaux domaines d'application de l'ingénierie des modèles qui pourront nécessiter selon le cas l'utilisation de métamétamodèles différents et de transformations intra ou extra métamodèles. Les usages les plus fréquents sont :
– la réingénierie et l'évolution ;
– la réutilisation de modèles sur étagères ;

[1] Il propose ainsi une union entre la première et la dernière famille de langages présentées dans cette taxonomie.

– la conception par aspects (AOD), avec les procédés de tissage d'aspects ;
– la modélisation par sujets (*subject-oriented design*) ;
– la gestion des familles de produits (famille de modèles) ;
– l'interopérabilité (exemple PIM vers PIM ou PSM vers PSM) ;
– la portabilité (ex : PIM vers PSM).

La *réingénierie* (anglicisme venant de *refactoring*) est une opération de maintenance d'une application. Elle consiste à retravailler les différents modèles au cœur de l'application (code, architecture, schéma de données, etc.) non pour ajouter une fonction supplémentaire au logiciel mais pour améliorer sa lisibilité et simplifier sa maintenance. On parle alors aussi de transformation *isofonctionnelle*. L'évolution du logiciel est typiquement un aspect complémentaire de cette activité de maintenance. Elle concerne notamment l'ajout de services et elle nécessite de pouvoir faire un suivi des modifications apportées au logiciel. Il faut en particulier pouvoir disposer *i)* d'un historique des versions suivant différents critères et degrés de granularité (opération, classe, relation, système, etc.) et *ii)* d'un langage de requêtes pour pouvoir exploiter les informations suivant différents points de vue [GFD].

La *conception par aspects* (en anglais *aspect oriented design* - AOD) est un paradigme de conception qui permet de réduire fortement les couplages entre les différents aspects techniques d'un logiciel. La conception par aspects est un paradigme transverse et n'est pas liée uniquement à l'étape de programmation mais peut être mise en œuvre aussi bien avec un langage à objets comme Java qu'avec un quelconque métamodèle propre à un domaine de conception particulier, le seul prérequis étant l'existence d'un tisseur d'aspect pour le domaine cible. Ce tisseur utilise alors un certain nombre de primitives de transformation pour réaliser la fusion entre un aspect et un modèle de base. Par rapport à l'expressivité du métamétamodèle, il faut pouvoir :

1. faire la différence entre le modèle de base et un aspect ;
2. spécifier les correspondances entre les entités à composer ;
3. disposer de directives pour intégrer des préoccupations transverses [FRGG04, OH05] ; ces directives permettront à la fois de résoudre les conflits et de gérer l'ordre de composition si plusieurs aspects doivent être intégrés dans le modèle [SGS+04].

La *conception par sujets* est comme la conception par aspects une approche qui vise à améliorer la structuration de la mise en œuvre d'une application. Un sujet est un modèle ou un fragment de modèle exprimé à l'aide des paradigmes du domaine cible. La conception par sujets propose de composer un ensemble de sujets pour produire l'application finale [Cla02, OH05]. Ce processus de composition, nommé *intégration*, se fait en définissant des règles. Une règle implique au moins deux sujets et permet de définir la composition à l'aide de primitives de transformation simples (fusion, redéfinition, séquencement, etc.). La principale différence avec les approches de conception par aspects est le caractère symétrique de l'approche. En effet, les sujet sont tous vus au même niveau, aucun sujet n'est *a priori* dominant par rapport à un autre.

Une des raisons majeures de l'apparition des architectures dirigées par les modèles repose sur la volonté de décrire au mieux le savoir-faire ou la connaissance métier d'une organisation dans des modèles abstraits indépendants des plates-formes (PIM - *Platform Independent Models*) [StOS00b]. Ayant isolé le savoir-faire dans des PIM, on a besoin soit de transformer ces modèles en d'autres PIM (besoin d'interopérabilité), soit de produire ou de créer des modèles (PSM - Platform Specific Models) qui leur correspondent pour une plate-forme d'exécution spécifique (pour améliorer la portabilité et augmenter la productivité). Les travaux de [OH05] portent sur un modèle PIM pour décrire et composer des préoccupations indépendamment des plate-formes d'exécution, sur la séparation des préoccupations puis la proposition de deux nouveaux PIM, plus proches des plate-formes d'exécution (l'un pour la programmation par sujets et l'autre pour la programmation par aspects).

Parmi les domaines où l'ingénierie des modèles fait son entrée on trouve celui des IHM [SCF05]. La modélisation d'une IHM repose en particulier sur la description de six modèles : les modèles des *concepts*, des *tâches*, des *concepts-tâches*, des *espaces*, des *interacteurs* et de *programmes*. L'IDM doit proposer à la fois des métamodèles permettant la spécification de ces différents modèles et une approche pour rendre explicite les correspondances entre ces métamodèles. Un des objectifs est bien évidemment d'utiliser ces informations pour décrire les transformations associées et ainsi automatiser la construction d'IHM.
Les procédés de composition de fragments présentent un certain nombre de difficultés, énoncées ci-après.
Un fragment de modèle quelconque n'est pas nécessairement réutilisable d'emblée. En effet, la réutilisation des modèles sur étagère repose en général sur une bonne séparation des préoccupations et sur la possibilité de pouvoir adapter l'existant pour permettre son intégration dans le contexte où il est réutilisé. Un exemple typique de modèle qui doit pouvoir être réutilisé facilement est le *patron de conception*.
La sélection du modèle à composer présente également des difficultés. Deux orientations se dégagent pour analyser et sélectionner les modèles adaptés à un besoin donné : proposer un opérateur de comparaison fondé par exemple sur les aspects structurels ou dynamiques du modèle [Bar05a] ou un langage de requêtes dédié qui s'appuient sur un ensemble de critères permettant de caractériser les différents modèles. Les modèles à réutiliser [BG04, BGb05], notamment quand il s'agit de patrons de conception, nécessitent de mettre en place des mécanismes de composition, ou plus généralement de transformations entre PIM.
Parmi d'autres approches proposant des mécanismes de structuration, citons l'approche Theme [BC04] de Clarke *et al.* Cette approche repose sur la composition d'aspects tant lors de la phase d'analyse des besoins (Theme/Doc) que celle de la connexion de l'architecture (Theme/UML).
Un domaine particulier de réutilisation concerne les lignes de produits. En dehors de [Zia04] il y a peu d'approches complètes qui incluent à la fois les parties statiques et dynamiques des modèles, ainsi que la dérivation des produits. Par exemple, le document [Gom04] considère uniquement l'aspect statique du modèle, tandis que les travaux de la publication [JM02] sont fondés sur les cas d'utilisation. L'approche proposée par T. Ziadi [Zia04] consiste à :

1. utiliser un profil UML qui permet de décrire la variabilité, aussi bien dans les diagrammes statiques que dynamiques,
2. gérer des contraintes OCL pour exprimer à la fois des contraintes communes ou non à toutes les lignes de produits ;
3. dériver, par transformation de modèles, les modèles statiques et dynamiques des lignes de produits.

Dans cette approche, les mécanismes d'extension d'UML (stéréotypes, *tagged values*, etc) sont largement utilisés et la description de l'aspect dynamique du modèle est réalisée par des diagrammes de séquence (interactions entre les différentes entités du modèle) et par des machines à états pour la modélisation du comportement interne d'un objet. L'approche proposée dans [CLT06] est fondée sur la description d'actions encapsulées dans la partie méta des entités à modéliser et sur un mécanisme de composition de modèles qui seront ensuite assemblés en fonction des besoins. Une seconde différence est que, dans cette approche, les produits dérivés sont construits par instanciation de la partie générique de la ou des entités associées alors que l'approche de T. Ziadi les construit en s'appuyant sur le patron de conception *fabrique abstraite* [GHJV99] et sur MTL, un langage de transformations de modèles [Pro05].
Plus généralement il apparaît que la mise en œuvre de la variabilité des diagrammes statiques et dynamiques, c'est-à-dire l'aspect optionnel ou alternatif de certaines entités (propriétés, méthodes, classes,etc.), oblige à disposer d'un protocole ou d'un langage de composition dédié à la variabilité des lignes de produits.

5.3 Pouvoir d'expression des métamétamodèles

Les approches ou outils mis en œuvre pour réaliser les transformations sont décrits dans les sections 5.4.1 et 5.4.2. La richesse de ces transformations repose en particulier sur l'expressivité des métamétamodèles utilisés pour la description des métamodèles et des modèles qui leur sont associés. L'objectif de cette section est de donner un aperçu du pouvoir d'expression de ces métamétamodèles. On abordera en particulier les aspects structurels et comportementaux. Les comportements s'appuient naturellement sur des concepts tels que les automates, les réseaux de Petri, les diagrammes de séquence de la version 2 d'UML (catégorie de Hierarchical Message Sequence Charts).

5.3.1 Contraintes sur la structure

Voici ci-après quelques exemples de contraintes exprimées par un métamodèle et qui doivent être satisfaites par tous les modèles conformes à ce métamodèle.

1. Soit la propriété P_1 : *Aucune classe de modèle décrite à l'aide du métamodèle X ne contient plus de trois opérations* ;
   - si le métamodèle X est défini en UML, cette règle pourra s'exprimer en OCL,
   - si le métamodèle X est défini en XML, cette règle sera difficile à exprimer et devra être définie dans d'autres outils extérieurs tels que XSLT ;
2. Soit la propriété P_2 : *s'il existe deux classes qui ont le même identifiant alors on construit une classe qui résulte de la composition des deux classes, cette composition est définie au niveau du métamodèle ;*
   - si le métamodèle X est défini en UML, cette règle sera difficile à écrire et on peut envisager l'utilisation de langage dédié tel que J avec cependant une perte de l'information diluée dans le code J,
   - si le métamodèle X est défini en Kermeta, cette règle sera plus facile à écrire car on dispose du pouvoir d'expression d'un pseudo-langage.
3. Soit la propriété P_3 : *calculer l'ensemble des éléments qui contiennent dans leur identifiant le mot pk* ;
   - comme pour l'exemple précédent, les pseudo-langages facilitent l'extraction de l'information et sa réutilisation pour spécifier des propriétés. Par exemple, si le métamodèle X est défini en Kermeta, cette règle sera plus facile à écrire ; on dispose en effet d'un langage permettant des requêtes et des traitements sur les résultats de ces requêtes.

5.3.2 Exemples de langages de description de métamodèle

Voici ci-après une liste non exhaustive de moyens d'expression de métamodèles, autrement dit de métamétamodèles :
- le langage Kermeta [MFJ05a] ;
- *EMF [BEG+03], MOF* ;
- *UML* ;
- XMI ;
- logique de description ;
- grammaire, arbre de syntaxe abstraite.

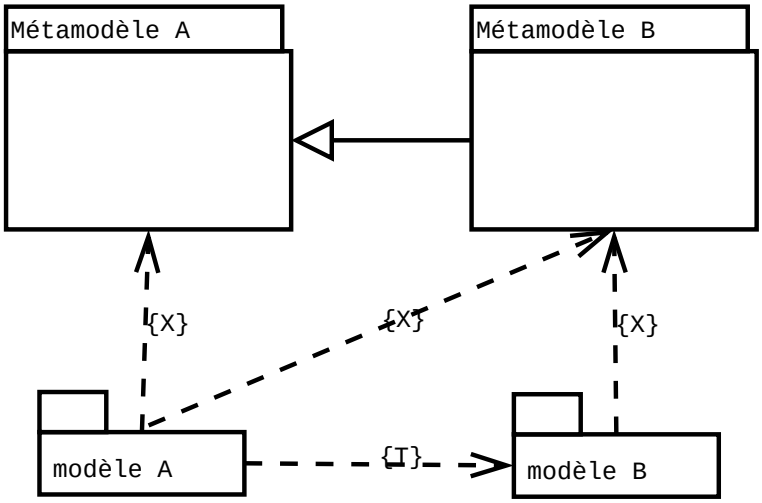

FIG. 5.1 – Description de transformations intra-modèles

5.4 Nature des transformations

Les transformations peuvent prendre des formes extrêmement variées. L'objet de cette section du document est de présenter quelques grandes familles importantes.
On distinguera ci-après deux grandes familles de transformations :

1. celles qui concernent le raffinement ou l'enrichissement de métamodèles ;
2. celles qui concernent le passage entre deux métamodèles complètement différents.

5.4.1 Transformations intra-métamodèle

Il s'agit des transformations qui travaillent à métamodèle constant : toutes les transformations opèrent sur des modèles dépendant d'un seul métamodèle.
La figure 5.1 montre une relation entre les métamodèles A et B. Cette relation peut être une relation d'enrichissement ou pas mais dans tous les cas il faudra que le modèle A à transformer, soit conforme aussi au métamodèle B. De même il peut y avoir plusieurs niveaux de relation (un métamodèle A peut lui même enrichir un métamodèle C). D'une manière générale l'idée est qu'il y a une intersection entre les métamodèles A et B.
Ces catégories regroupent soit des procédés (composition, aspects, sujets, thèmes) soit des applications (réingénierie, raffinement, familles de produits, évolutions).

5.4.2 Transformations extra-modèle (métamodèles hétérogènes)

Dans la figure 5.2 il n'y a pas de relation entre les métamodèles A et B et donc on ne peut rien dire dans le cas général. En particulier le modèle A ne sera pas conforme au métamodèle B. Les points importants sont :

1. d'une part l'existence d'un modèle B qui résulte de la transformation de A et qui soit conforme au métamodèle B ;
2. d'autre part l'existence de réponses à des questions comme T^{-1} existe-t-elle ou T est-elle injective ?

De nombreuses plates-formes d'exécution apparaissent régulièrement telles que Java ou Microsoft DotNET et avec elles, la nécessité de les intégrer dans des systèmes existants.

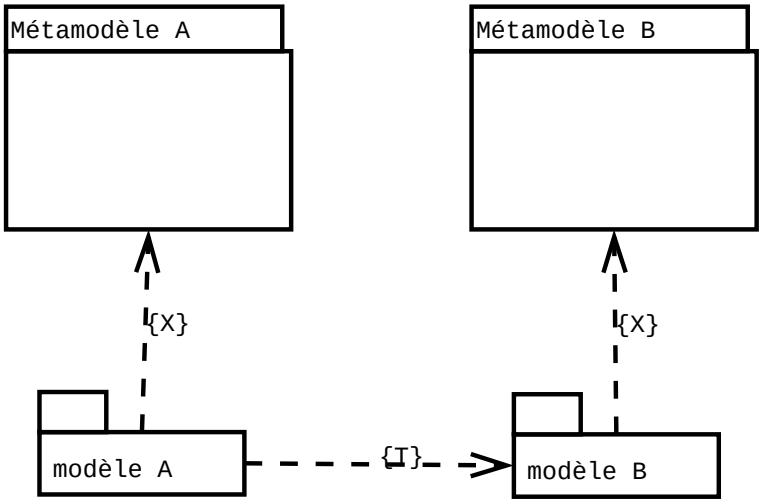

FIG. 5.2 – Description de transformation extra-modèles

Pour pallier ce problème, l'OMG a donc proposé l'architecture MDA (Model Driven Architecture). Le point clé du MDA est de décorréler les spécifications fonctionnelles d'un système de son implémentation sur une plate-forme donnée. Il s'agit ici de définir un modèle métier indépendant de toute technologie spécifique (PIM). Ce PIM va ensuite être projeté sur un modèle décrivant la plate-forme d'exécution choisie (PDM) pour obtenir un modèle décrivant l'application métier dans un contexte technologique particulier (PSM). On peut alors projeter un même système sur diverses plates-formes d'exécution sans avoir à modifier le modèle métier initial. Ce processus en "Y" repose sur un ensemble de transformations de modèles permettant de passer d'un PIM à PSM mais également de raffiner les PIM ou les PSM indépendamment. Dans ce processus, le PSM qui décrit une plate-forme d'exécution tend à éclater en une multitude d'aspects techniques transverses (sécurité, performance, etc.) qui vont être intégrés au modèle métier initial pendant le cycle de développement. Chacun de ces aspects fait l'objet d'un langage particulier spécifique de domaine, qu'il faut équiper des transformations de modèles adéquates pour pouvoir l'intégrer dans le modèle métier. Les contraintes et les propriétés des plates-formes technologiques opérant par composants sont décrites au chapitre 4.2.1.

5.4.3 Procédés et usages de transformation de modèles

L'objectif de cette section est de mettre en évidence l'existence des patrons de transformation. À l'instar des patrons de conception classiques, ces patrons de transformation correspondent à une solution classique à un problème de transformation. Un patron est caractérisé par le problème qu'il résout plus que par la façon dont il le résout.
Nous indiquons ci-après quelques applications typiques :
– opérateurs de transformation au sens de la méthode Catalysis (raffinement d'objets, d'actions, de modèles, d'opérations) ;
– relation avec les composants logiciels ;
– transformation de paquetage (généricité, application de thèmes) ;
– extraction de propriétés formelles et construction d'un modèle formel par abstraction ; l'objectif est d'alimenter un outil externe, par exemple pour faire une preuve (Vauchon, Plouzeau,etc.) ;
– composition de deux modèles ;
– mise en oeuvre des patrons de conception dans un modèle (Umlaut) ;
– travaux d'alexis Muller.

Patrons de transformation de Catalysis La méthode Catalysis ne préconise pas spécifiquement des techniques de transformation avec un langage de transformation, mais repose sur la notion de transformations de modèles comme support de la méthode. Plus précisément, Catalysis est centré sur la notion d'abstraction et de raffinement. Un modèle M_1 est une abstraction d'un autre modèle M_2 s'il existe une relation de *retrieval* entre M_1 et M_2. En règle générale, M_2 est obtenu à partir de M_1 au cours du processus d'analyse et de conception.
Toute transformation est composée d'une série de transformations élémentaires. Les types de transformation élémentaire sont au nombre de quatre :
– le raffinement d'objet ;
– le raffinement de modèle ;
– le raffinement d'opération ;
– le raffinement d'action.
La méthode Catalysis [DW99] ne propose pas simplement une classification de transformations élémentaires. Elle propose également des patrons de méthode, fondés sur l'application de motifs récurrents de transformations (raffinements) lors des phases d'analyse et de conception du logiciel.

Relation avec les composants logiciels Étant donné que la transformation de modèle recouvre un ensemble de techniques ayant pour point commun l'exécution d'opérations lisant et modifiant des modèles, nous excluons dans ce chapitre les catégories de composants logiciels pour lesquelles la structure n'exige pas de transformation de modèles (par exemple DotNet). On peut noter au passage que certains modèles de composants (par exemple au sens de Catalysis) s'appuient sur la transformation de modèles pour la réalisation de l'architecture.
Plus précisément, un composant au sens de Catalysis est un fragment de modèle qui contient des éléments typiques des interactions et de la structuration sous forme de composants : ports, connecteurs, etc. Cependant, un tel composant n'est pas implanté tel quel dans une architecture technologique à composants. Les différents éléments caractéristiques du composant doivent être transformés lors d'étapes de raffinement, jusqu'à obtenir un architecture dépendante de la plate-forme (avec ou sans notion de composants). Le métamodèle de composant de Catalysis est donc du genre *platform-independant* (voir [DW99], chapitre 10).

Transformation de paquetages. La transformation de paquetages a pour but d'adapter des éléments de modèle dans un but de réutilisation. À cette fin, ces éléments sont organisés en paquetage, structures dotées de moyens de paramétrage. Une fois les paramètres liés à des valeurs, un élément de modèle est construit sur la base du paquetage.

Extraction de modèles formels Les outils d'analyse de propriétés formelles d'un modèle n'acceptent en général que des formalismes et des notations bien spécifiques. Pour pouvoir utiliser ces outils, il est nécessaire de faire des projections du modèle à analyser. L'intégration de tels outils dans un environnement de développement fondé sur les modèles est souvent réalisée de la manière suivante :

1. définition d'un métamodèle F par étude du langage d'entrée et du formalisme de l'outil ;
2. définition d'une transformation τ construisant un modèle conforme à F à partir du modèle à analyser M_1 ;
3. exécution de la transformation τ et production d'un modèle conforme au métamodèle de l'outil cible, dans une syntaxe adaptée ;
4. exécution de l'outil d'analyse externe et production des résultats ;

5. transformation inverse des résultats pour interprétation dans le contexte du modèle initial à analyser.

5.5 DSL et outils

L'objectif de cette section est de faire un tour d'horizon de l'utilisation de l'ingénierie des modèles pour mettre en oeuvre un métamodèle particulier et les applications qui y font référence. Cela revient à offrir un DSL permettant de décrire les modèles conformes à un métamodèle et le comportement associé à ce métamodèle. Ce métamodèle peut décrire une transformation ou être manipulé par une transformation.

5.5.1 DSL et Software factories

Les usines logicielles (*software factories* [GSC+04]) sont la vision de Microsoft de l'ingénierie des modèles. Le terme d'usine logicielle fait référence à une industrialisation de développement logiciel et s'explique par l'analogie entre le processus de développement proposé et une chaîne de montage industrielle classique. En effet, dans l'industrie classique, les usines ont les propriétés suivantes :

– Une usine fabrique une seule famille de produits. Si l'on considère par exemple l'industrie automobile, une chaîne de montage ne fabrique généralement qu'un seul type de voiture avec différentes combinaisons d'options.
– Les ouvriers sont relativement spécialisés. Il n'est pas rare de trouver des ouvriers ayant des compétences sur plusieurs postes de la chaîne de montage mais il est très rare qu'un ouvrier ait toutes les compétences depuis l'assemblage à la peinture des véhicules.
– Les outils utilisés sont très spécialisés et fortement automatisés. Les outils utilisés sur une chaîne de montage sont conçus uniquement pour cette chaîne de montage ce qui permet d'atteindre des degrés d'automatisation importants.
– Toutes les pièces assemblées ne sont pas fabriquées sur place. Une chaîne de montage automobile ne fait généralement qu'un assemblage de pièces standard ou préfabriquées à un autre endroit.

L'idée des usines logicielles est d'adapter ces caractéristiques au développement de logiciels, qui ont fait leur preuve pour la production de masse de familles de produits matériels. Les deux premiers points correspondent à la spécialisation des fournisseur de logiciels et des développeurs à l'intérieur des équipes de développement. Le troisième point correspond à l'utilisation d'outils de génie logiciel spécifiques du domaine d'application, c'est-à-dire de langages, d'assistants et transformations spécifiques. Le dernier point correspond à la réutilisation de composants logiciels sur étagères. Par exemple, l'environnement de développement Visual Studio .NET 2005 de Microsoft a été conçu autour de ces idées. Cet outil propose un environnement générique extensible pouvant initialement être configuré pour un ensemble de domaines d'application prédéfinis.

5.5.2 DSL pour les métamodèles et les transformations

L'objectif de cette section est faire un tour d'horizon des DSLs de l'état de l'art qui permettent de décrire les métamodèles à manipuler et les métamodèles qui décrivent des transformations.

Les techniques de transformation sont souvent utilisées comme mécanisme de base pour décrire la sémantique de la composition (voir section 4.3). Dans ce sens, différentes approches ont travaillé sur la définition de langages de transformation dédiés à un métamodèle pour la définition de sémantiques de composition dans le cadre de ce métamodèle. Considérons trois approches qui ont travaillé sur la construction de tels langages :

1. TranSAT [Bar05b] pour le modèle de composant SafArchie ;
2. SmartModels [CL06], pour composer les programmes Java ;
3. Nano *et al.* [Nan04] pour une approche à base d'annotations et de transformations de modèles pour l'intégration de services.

Les spécificités de ces différentes approches ont été présentées dans la section 4.3. Nous ne détaillerons donc pas d'avantage ces approches ici.

5.5.3 Outils de transformation

L'objectif est ici de faire un tour d'horizon des outils qui permettent d'implanter les DSLs mentionnés ci-dessus (langages qui décrivent une transformation).
De nombreux outils, tant commerciaux que dans le monde de l'open source sont aujourd'hui disponibles pour faire la transformation de modèles. On peut grossièrement distinguer quatre catégories d'outils :

1. les outils de transformation génériques qui peuvent être utilisés pour faire de la transformation de modèles ;
2. les facilités de type *langages de scripts* intégrés à la plupart des ateliers de génie logiciel ;
3. les outils conçus spécifiquement pour faire de la transformation de modèles et prévus pour être plus ou moins intégrables dans les environnements de développement standards ;
4. les outils de métamodélisation au sens propre, dans lesquels la transformation de modèles revient à l'exécution d'un métaprogramme.

Dans la première catégorie on trouve notamment
– d'une part les outils de la famille XML, comme XSLT [W3C05b] ou Xquery [W3C05a] ;
– d'autre part les outils de transformation de graphes (la plupart du temps issus du monde académique) [EGdL+05].
Les outils de la première catégorie ont l'avantage d'être déjà largement utilisés dans le monde XML, ce qui leur a permis d'atteindre un certain niveau de maturité. En revanche, l'expérience montre que ce type de langage est assez mal adapté à des transformations de modèles complexes (c'est-à-dire allant au-delà des tâches de transcodage syntaxique), car ils ne permettent pas de travailler au niveau de la sémantique des modèles manipulés mais simplement à celui d'un arbre couvrant le graphe de la syntaxe abstraite du modèle. Ceci impose de nombreuses contorsions qui rendent rapidement ce type de transformation de modèles complexes à élaborer, à valider et surtout à maintenir sur de longues périodes.
Dans la seconde catégorie figure une famille d'outils de transformation de modèles proposés par des vendeurs d'ateliers de génie logiciel. Par exemple, l'outil *Arcstyler* [Hub] de Interactive Objects propose la notion de *MDA-Cartridge* qui encapsule une transformation de modèles écrite en *JPython* (langage de script construit à l'aide de Python et de Java). Ces *MDA-Cartridges* peuvent être configurées et combinées avant d'être exécutées par un interpréteur dédié. L'outil propose aussi une interface graphique pour définir de nouvelles *MDA-Cartridges*. Dans cette catégorie on trouve aussi l'outil *Objecteering* [Sof] de Objecteering Software (filiale de Softeam), qui propose un langage de script pour la transformation de modèles appelé J, ou encore OptimalJ [Com] de Compuware qui utilise le langage TPL, et bien d'autres encore, y compris dans le monde de l'open source avec des outils comme Fujaba [BGN+04] (From UML to Java and Back Again).
L'intérêt de cette catégorie d'outils de transformation de modèles est d'une part leur relative maturité (car certains d'entre eux comme le langage *J* d'Objecteering sont développés depuis plus d'une décennie) et d'autre part leur excellente intégration dans l'atelier de génie logiciel

qui les héberge. Leur principal inconvénient est le revers de la médaille de cette intégration poussée : il s'agit la plupart du temps de langages de transformation de modèles propriétaires sur lesquels il peut être risqué de miser sur le long terme. De plus, historiquement ces langages de transformation de modèles ont été conçus comme des ajouts au départ marginaux à l'atelier de génie logiciel qui les héberge. Même s'ils ont aujourd'hui pris de l'importance, ils ne sont toujours pas vus comme les outils centraux qu'ils devraient être dans une véritable ingénierie dirigée par les modèles. Ces langages montrent à nouveau leurs limites lorsque les transformations de modèles deviennent complexes et qu'il faut les gérer, les faire évoluer et les maintenir sur de longues périodes.

Dans la troisième catégorie, c'est-à-dire les outils conçus spécifiquement pour faire de la transformation de modèles et prévus pour être plus ou moins intégrables dans les environnements de développement standards, on va trouver par exemple *Mia-Transformation* [MS] de Mia-Software. Cet un outil exécute des transformations de modèles prenant en charge différents formats d'entrée et de sortie (XMI, tout autre format de fichiers, API, dépôt). Les transformations sont exprimées comme des règles d'inférence semi-déclaratives (dans le sens où il n'y a pas de mécanisme de type de programmation logique), qui peuvent être enrichies en utilisant des scripts Java pour des services additionnels tels que la manipulation de chaînes.

PathMATE [Sol] de Pathfinder Solutions est un autre environnement configurable de transformation de modèles qui s'intègre particulièrement bien avec les ateliers de modélisation UML dominant le marché.

On trouvera dans le monde académique de nombreux projets open source s'inscrivant dans cette approche : les outils ATL [JK05] et MTL [VJ04] de l'INRIA, AndroMDA [And], BOTL [Bra03] (Bidirectional Object oriented Transformation Language), Coral [AP04] (*Toolkit to create/-edit/transform new models/modeling languages at run-time*), ModTransf [Dum] (*XML and ruled based transformation language*), QVTEclipse [Tra] (une implantation préliminaire de quelques unes des idées du standard QVT dans Eclipse) ou encore UMT-QVT [UQ] (*UML Model Transformation Tool*).

La dernière catégorie d'outils de transformation de modèles est celle des outils de métamodélisation dans lesquels la transformation de modèles revient à l'exécution d'un métaprogramme. Le plus ancien et le plus connu de ces outils est certainement MetaEdit+ [SLTM91, TR03] de MetaCase. Celui-ci permet de définir explicitement un métamodèle, et de programmer au même niveau tous les outils nécessaires, allant des éditeurs graphiques aux générateurs de code en passant par des transformations de modèles. Dans une veine similaire, XMF-Mosaic [Xac] de Xactium est un environnement complet de définition de langage. Au cœur de XMF-Mosaic on trouve un noyau exécutable de définition de langage (qui est d'ailleurs auto-définissant). Tout est modélisé sur cette base, que ce soit les langages (par exemple UML), les outils, les interfaces graphiques, parseurs et autres analyseurs XML, mais à la différence de l'environnement MetaEdit+ obtenu par génération plutôt que par instanciation. Le langage de métamodélisation de base est un langage à objets qui a toute la puissance d'un langage de programmation complet. Cette propriété en fait un outil particulièrement puissant pour la transformation de modèles. Dans le monde de l'open source on trouvera dans cette catégorie l'environnement Kermeta [MFJ05b] développé à l'INRIA. Kermeta peut être décrit comme l'ajout d'un aspect d'exécutabilité dans le MOF par un mécanisme astucieux inspiré du tissage d'aspects ; ceci lui permet de garder une compatibilité totale avec les environnements MOF standards (par exemple EMF). On peut enfin noter qu'il y a aussi d'autres travaux dédiés à la manipulation de modèles EMF mais à partir de *JavaScript* [TV05].

5.6 Validation des transformations de modèles

L'objectif est de montrer qu'il est nécessaire de réaliser des contrôles avant de pouvoir mettre en oeuvre une transformation et d'autre part qu'il faut après la transformation garantir que celle-ci n'a pas fait n'importe quoi. Une transformation de modèle est l'exécution d'un programme transformateur, que celui-ci soit spécifié directement (sous une forme opérationnelle) ou par application de règles de transformation exécutées par un moteur de règles (spécification déclarative). En conséquence, la définition d'une transformation est une implantation d'une spécification. Comme toute implantation, une définition de transformation doit être validée par rapport à sa spécification.

5.6.1 Validation avant transformation

Avant de réaliser un ensemble de transformations que ce soit pour obtenir un modèle dérivé ou un modèle qui résulte d'une composition, il est intéressant de vérifier que les transformations sont composables. Cela revient en particulier à détecter les possibles dépendances entre les différentes transformations et les conflits qui peuvent apparaitre à l'issue de ces transformations.
Parmi les approches qui s'attaquent à cette problématique on trouve en particulier *AGG*[2] [Tae04] et *Condor* [KB03]. Une étude de ces deux approches est réalisée dans [MKR06] sur le thème du *refactoring*. La séquence de transformations correspond à l'ensemble des changements à réaliser sur un modèle représentant une réification de la hiérachie des classes à faire évoluer. Les transformations représentent une succession d'opérations élémentaires comme par exemple l'ajout, la suppression ou le renommage d'un noeud qui pourra notamment représenter un attribut, une méthode ou une classe.
La première approche est basée sur la théorie des graphes tandis que la seconde s'appuie sur la logique et plus précisément sur le concept de transformation conditionnelle (transformation gardée par une précondition). L'approche *AGG* permet de détecter d'une part, les conflits qui résultent de l'impossibilité de sérialiser deux transformations quelque soit l'ordre d'application choisi et d'autre part, l'existence de dépendances séquentielles (deux transformations doivent être réalisées dans un certain ordre).
De son coté, *Condor* permet la détection de dépendances de déclenchement (*triggering dependencies*) ou d'inhibition (*inhibition dependencies*). En d'autres termes, si la postcondition (dérivée automatiquement de la précondition et de la transformation) permet de satisfaire la précondition d'une seconde transformation alors il y a une dépendance de déclenchement entre les deux transformations. Si au contraire elle rend fausse cette précondition alors cela correspond à une dépendance d'inhibition.

5.6.2 Validation après transformation

Les transformations de modèles sont au cœur des processus de développement utilisant l'ingénierie des modèles. De ce fait, pour assurer la qualité du logiciel produit, il est primordial que les transformations de modèles utilisées soit fiables et robustes. En effet, il suffit d'une seule transformation erronée dans le processus qui conduit de la modélisation des besoins au code pour que le processus de développement dans son ensemble soit fragilisé. Le besoin de qualité pour les transformations est d'autant plus important qu'elles sont généralement faites pour être réutilisées. Si l'on souhaite que l'ingénierie des modèles tienne ses promesses en terme de qualité du logiciel produit, il est donc nécessaire de mettre en œuvre des techniques de validation adaptées pour les transformations de modèles.

[2] AGG pour Attributed Graph Grammar.

D'un point de vue pratique, les transformations de modèles sont des programmes dont les données d'entrée et de sortie sont des modèles. La complexité des transformations est variable mais peut dans certain cas atteindre celle d'un compilateur. Comme pour tout programme, la validation des programmes de transformation de modèles doit donc passer par une phase de test rigoureuse. Dans la littérature, bien qu'un grand nombre de travaux mettent la transformation de modèle au cœur de l'ingénierie de modèles, très peu de travaux s'intéressent au test de ces programmes. Il existe pourtant deux raisons de ne pas se contenter des techniques de test de programmes existantes :

1. Les techniques de test de programme ne sont pas adaptées à la grande complexité des données manipulées par les transformations de modèles. Les modèles manipulés par les transformations sont des graphes d'objets qui peuvent atteindre une grande complexité.
2. Les transformations de modèles ont des caractéristiques spécifiques qui ne sont pas exploitées par les techniques de test classiques. Par exemple, le fait que les modèles soient décrits par des métamodèles fournit un élément de spécification originale qui peut être utilisé pour le test [FSB04].

Dans [Küs01], l'auteur identifie clairement le besoin de techniques des validations et de tests spécifiques aux transformations de modèles. L'article insiste sur le fait que les techniques éprouvées de génie logiciel telles que la modularité, la séparation des préoccupations ou l'anticipation des changements doivent être réutilisés pour le développement des programmes de transformation de modèles. Dans le cadre d'une approche de transformation de modèle déclarative, l'auteur propose une technique permettant de vérifier des propriétés syntaxiques et des propriétés de vivacité sur un ensemble de règles de transformation.
Dans [SL04], les auteurs présentent les problèmes de validation qu'ils ont rencontrés au cours du développement du moteur déclaratif de transformation de modèles Tefkat [DGL+03]. L'article montre le parallèle qui existe entre le test de ce moteur de transformation et le test des transformations de modèles elles-mêmes. Le principal problème est dans les deux cas d'utiliser des modèles comme données de test : il faut les éditer, les stocker, les comparer et les faire évoluer. L'idée d'utiliser des critères de test systématiques est abordée comme une possibilité mais dans la pratique les auteurs se sont contentés de créer manuellement des modèles de test.
Dans [LZG05], Lin *et al.* identifient les différentes activités liées au test de transformation de modèles et propose un *framework* pour les organiser. Le problème de la sélection et de la génération des modèles de test n'est pas abordé. Les auteurs concentrent leur effort sur les problèmes liés à la comparaison de modèles nécessaire pour vérifier qu'un modèle obtenu est conforme à un modèle attendu (fonction d'*oracle*). L'article propose un premier algorithme de comparaison de modèle inspiré d'algorithmes de correspondance de graphes.

5.7 Conclusion

Les différents concepts et verrous scientifiques ne seront pas tous traités ni même abordés dans le projet FAROS. L'ingénierie des modèles est dans FAROS un moyen et non un but. Sans préjuger des résultats de l'étude des métamodèles et des transformations qui sera réalisée dans le lot 2, il est possible de relier certains concepts de ce chapitre aux concepts et aux problèmes des chapitres précédents (Web services, contrats et composition).
En particulier, un objectif que nous nous sommes fixé ici est de montrer comment nous pourrions utiliser l'IDM dans le cadre de FAROS. Les choix mentionnés ci-après ne sont là qu'à titre d'exemple pour donner une vision plus concrète. Dans la suite le terme "(méta)modèle exécutable" est utilisé pour exprimer l'idée que l'on pourra appliquer des traitements et ceci par opposition à un (méta)modèle uniquement descriptif.

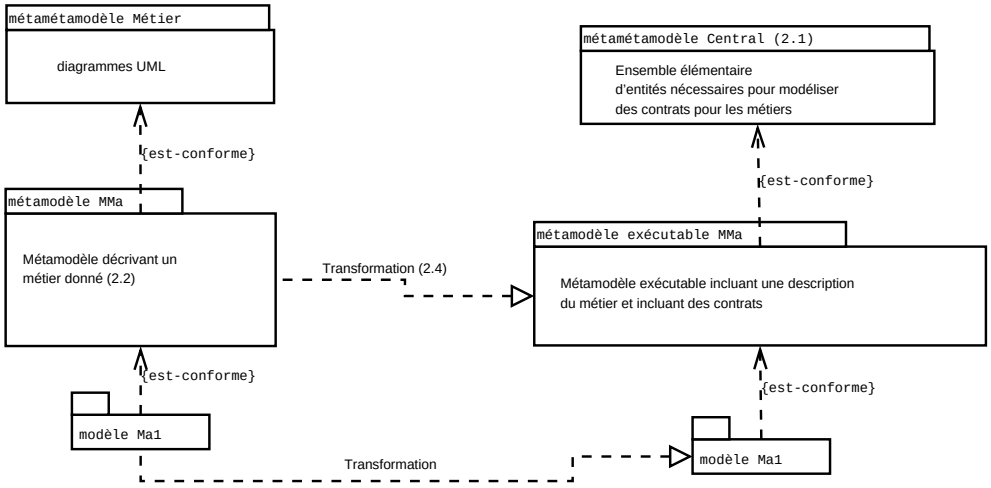

FIG. 5.3 – Métamodèles et transformations de modèles

5.7.1 Relation services/ingénierie des modèles

Intéressons nous à un métamodèle métier *MMa* (voir figure 5.3). Il pourra être est défini à l'aide de diagrammes UML incluant des contraintes OCL. Ces diagrammes représenteraient le langage de surface ou dédié à la description des métamodèles métiers.
Ainsi, les diagrammes de séquences, *use-cases* d'UML pourront définir les interactions entre l'utilisateur et les services et entre les services eux-mêmes. Le diagramme des classes pourra permettre la définition des propriétés et des opérations associées à chaque entité intervenant dans la description du métier. Tout ceci pourrait correspondre à une représentation possible des métamodèles que l'on attend comme résultat dans le lot 2.2.
A ce stade, il semble préférable que les concepts manipulés dans le métamodèle ne concernent pas la notion de service, de composant ou d'autres paradigmes. Nous avons uniquement besoin de disposer de l'expressivité nécessaire pour décrire le métier et les contraintes associées au métier. A partir d'un métamodèle *MMa* il est possible de définir des modèles conformes (*Ma1* dans la figure 5.3). Ils pourront être décrits en utilisant un langage de surface généré à partir de la description du métamodèle métier et éventuellement complété et adapté. L'objectif est de rendre plus facile la saisie des modèles par un utilisateur non informaticien. Typiquement cela pourrait se rapprocher de l'éditeur qui est généré par Eclipse à partir d'un modèle EMF [BEG+03].
Pour illustrer notre propos prenons l'exemple d'un métamodèle pour le métier "mise en œuvre d'un central téléphonique" (voir figure 5.4). Pour le spécifier il est nécessaire de pouvoir définir des types de contraintes dédiés à la mise en oeuvre des centraux téléphonique comme par exemple *garantir un débit entre deux serveurs frontaux*. Ce métamodèle correspond à un PIM et ne prend donc pas en compte les aspects particuliers à une plate-forme d'exécution. Pour diminuer la complexité de ce métamodèle il sera possible de le découper en plusieurs facettes qui seront assemblées par composition [Bar05b].
Un métamodèle exécutable dédié à un métamodèle métier donné est construit par transformation (lot 2.4) à partir du métamodèle (c'est un métamodèle du Lot 2.2.)[3]. A priori cette transformation devrait être extra-modèle et le métamodèle résultat est conforme au métamétamodèle central défini dans le lot 2.1. Il contiendra à la fois des informations sur le métier et une représentation des contrats en vue d'une projection vers les plate-formes d'exécutions (voir figure 5.5). Rappelons que le métamodèle métier est orienté "expert métier" alors

[3]Se rapprocher de l'exemple dont un extrait est défini dans la figure 5.4.

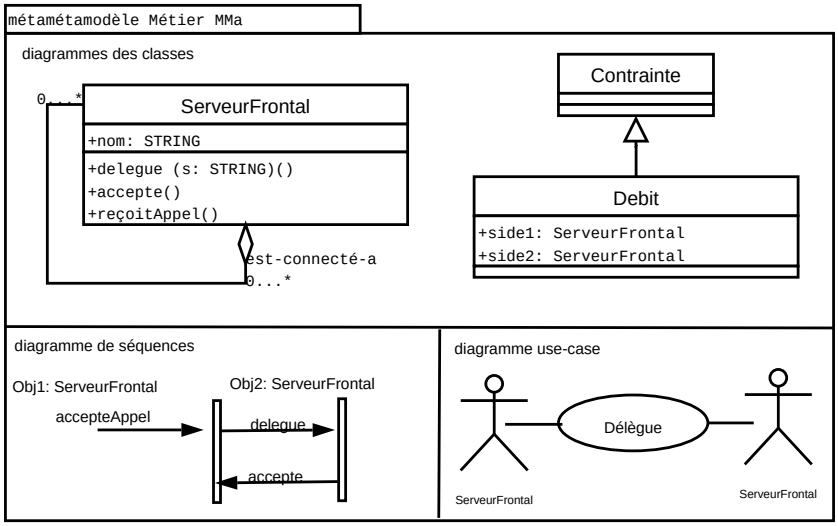

FIG. 5.4 – Extrait d'un métamodèle pour des centraux téléphoniques

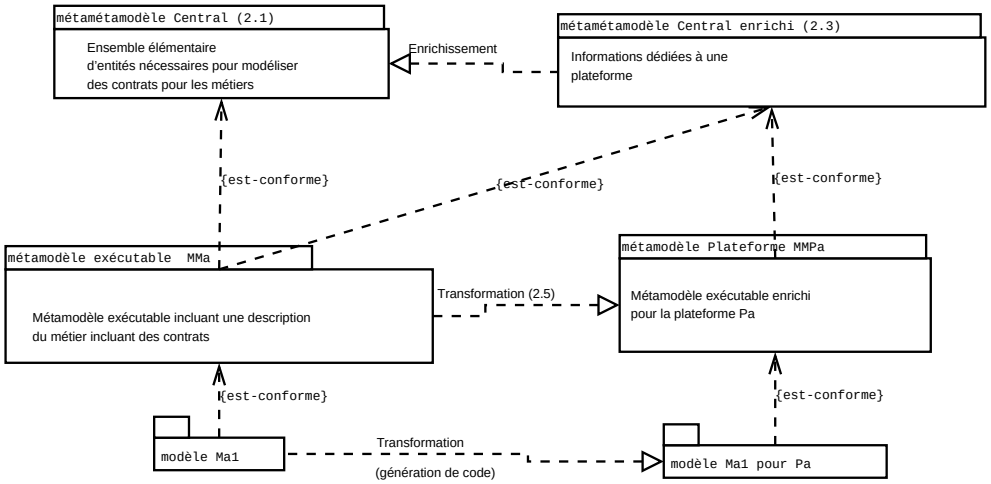

FIG. 5.5 – Projection vers la plate-forme

que le modèle exécutable est orienté vers une exploitation automatisée de ce modèle. On peut imaginer que ce métamodèle puisse être mis en oeuvre sous la forme d'un modèle kermeta (modèle EMF équipé de contraintes OCL et enrichi d'une description du comportement) [MFJ05b, MFJ05a].

5.7.2 Relation contrats/ingénierie des modèles

La modélisation des contrats est au centre de nos préoccupations, il est donc important de pouvoir caractériser les différents types de contrats et d'avoir l'expressivité suffisante pour définir les contraintes qui leur sont associées. Ainsi, le métamodèle central pourra être enrichi afin de contenir les informations additionnelles (en particulier concernant les contrats) qui pourront être exploitées au niveau d'une plate-forme d'exécution spécifique. Pour cela on pourra par exemple composer le métamodèle central avec les informations contenues dans un méta-métamodèle dédié à une plate-forme (voir figure 5.5). Le métamodèle *MMa* pourra donc être transformé (transformation intramodèle) afin d'inclure des aspects spécifiques à la

plate-forme d'exécution afin d'obtenir le métamodèle *MMpa*. Cela permettra pour un modèle *Ma1* de générer le code correspondant sur la plate-forme.

Bibliographie

[And] AndroMDA. Andromda. http ://www.andromda.org/.

[AP04] Marcus Alanen and Ivan Porres. Coral : A metamodel kernel for transformation engines. In D. H. Akerhurst, editor, *Proceedings of the Second Europena Workshop on Model Driven Architecture (MDA)*, number 17, pages 165–170, Canterbury, Kent CT2 7NF, United Kingdom, Sep 2004. University of Kent.

[Bar05a] Olivier Barais. *Construire et Maîtriser l'évolution d'une architecture logicielle à base de composants*. Thèse de doctorat, Laboratoire d'Informatique Fondamentale de Lille, Universié de Lille, Lille, France, novembre 2005.

[Bar05b] Olivier Barais. *Construire et Maîtriser l'évolution d'une architecture logicielle à base de composants*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Lille, France, nov 2005.

[BC04] E. Baniassad and S. Clarke. Theme : an approach for aspect-oriented analysis and design. *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 158–167, 2004.

[BEG+03] Frank Budinsky, Ray Ellersick, Timothy J. Grose, Ed Merks, and David Steinberg. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003. *ISBN : 0131425420*.

[BG04] Salim Bouzitouna and Marie-Pierre Gervais. Composition Rules for PIM Reuse. In *Proceedings of the 2nd European Workshop on MDA with Emphasis on Methodologies and Transformations (EWMDA'04)*, Canterbury, UK, September 2004. CSREA Press. 8 pages.

[BGb05] Salim Bouzitouna, Marie-Pierre Gervais, and Xavier blanc. Models Reuse in MDA. In *Proceedings of the 2005 International Conference on Software Engineering Research and Practice - SERP'05*, Las Vegas, Nevada, USA, June 2005. CSREA Press. 8 pages.

[BGN+04] Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool integration at the metamodel level within the fujaba tool suite. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3) :203–218, August 2004.

[Bra03] Peter Braun. Metamodel-based Integration of Tools. In *Proceeding of ESEC/FSE 2003, TIS 2003 Workshop on Tool Integration in System Development*, 2003.

[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.

[CLT06] P Crescenzo, Ph. Lahire, and Emanuel Tundrea. La généricité paramétrée au service des modèles métiers. In R. Rousseau and C. Urtado, editors, *Actes de LMO'2006, conférence nationale sur les Langages et Modèles à Objets.*, pages 151–166, Nîmes, France, mars 2006. Editions Hermès Lavoisier.

[CM98] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming*, volume 1490 of *LNCS*, pages 170–194, Pisa, Italy, septembre 1998.

[Com] Compuware. Optimalj. http ://www.compuware.com/products/optimalj/.

[DGL+03] Keith Duddy, Anna Gerber, Michael Lawley, Kerry Raymond, and Jim Steel. Model transformation : A declarative, reusable patterns approach. In *EDOC*, pages 174–185, 2003.

[Dum] Cedric Dumoulin. Modtransf. http ://www.lifl.fr/west/modtransf/.

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

[EFB+05] J. Estublier, J-M. Favre, J. Bézivin, L. Duchien, R. Marvie, S. Gérard, B. Baudry M. Bouzhegoub, J-M. Jézéquel, M. Blay, and M. Riveil. Action Spécifique CNRS sur l'Ingénierie Dirigée par les Modèles. Rapport de synthèse 1.1.2, CNRS, janvier 2005.

[EGdL+05] Karsten Ehrig, Esther Guerra, Juan de Lara, Laszló Lengyel, Tihamér Levendovszky, Ulrike Prange, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Model transformation by graph transformation : A comparative study. In *MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*, 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.

[FSB04] Franck Fleurey, Jim Steel, and Benoit Baudry. MDE and validation : Testing model transformations. In *Proc. of the SIVOES-Modeva workshop, SIVOES (Specification Implementation and Validation Of Embedded Systems)-MoDeVa (Model Design and Validation)*, Rennes, novembre 2004.

[GFD] T. Girba, J.M. Favre, and S. Ducasse. Using meta-model transformation to model software evolution.

[GHJV99] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Catalogue de modèles de conception réutilisables*. Addison-Wesley Publishing Co., 1999.

[Gom04] Hassan Gomaa. *Designing Software Product Lines with UML : From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[gre03] *Graph rewriting and transformation (GReAT) : a solution for the model integrated computing (MIC) bottleneck*, 2003.

[GSC+04] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi. *Software Factories : Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley ; 1st edition, 2004. *ISBN : 0471202843*.

[Hub] Richard Hubert. Arcstyler – the architectural ide for mda. http ://www.io-software.com/.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica, 2005.

[JM02] I. John and D. Muthig. Tailoring Use Cases for Product Line Modeling. In *Proceedings of the International Workshop on Requirements Engineering for Product Lines, REPL'02*, pages 26–32, Essen, Germany, September 2002.

[KB03] Gunter Kniesel and Uwe Bardey. Static Dependency Analysis for Conditional Program Transformations. Technical Report ISSN 0944-8535, CS Dept. III, University of Bonn, Bonn, Germany, July 2003.

[Küs01] J. M. Küster. Systematic validation of model transformations. 2004-10-01.

[LZG05] Y. Lin, J. Zhang, and J. Gray. A testing framework for model transformations. *Model-driven Software Development - Research and Practice in Software Engineering, Springer*, 2005.

[MFJ05a] P.A. Muller, F. Fleurey, and J.M. Jézéquel. Weaving executability into object-oriented meta-languages. *Proceedings of MODELS/UML*, 2005.

[MFJ05b] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume to be published of *LNCS*, pages –, Montego Bay, Jamaica, octobre 2005. Springer.

[MKR06] Tom Mens, Gunter Kniesel, and Olga Runge. Transformation dependency analysis : a comparison of two approaches. In R. Rousseau and C. Urtado, editors, *Actes de LMO'2006, conférence nationale sur les Langages et Modèles à Objets.*, pages 167–182, Nîmes, France, mars 2006. Editions Hermès Lavoisier.

[MS] Mia-Software. Mia-transformation. http ://www.mia-software.com/.

[MSFB05] Pierre-Alain Muller, Philippe Studer, Frédéric Fondement, and Jean Bezivin. Platform independent web application modeling and development with netsilon. *Software and Systems Modeling*, 4(4) :424–442, 2005.

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

[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.

[OMG01] The Action Semantics Consortium OMG. Action semantics for the UML, mars 2001. ad/2001-03-01.

[OMG03] Object Management Group OMG. *UML 1.5 Object Constraint Language Specification*, mars 2003. Version 1.5.

[OMG05] Object Management Group OMG. MOF QVT Final Adopted Specification, novembre 2005. Version 2.0.

[OMG06] Object Management Group OMG. Meta-Object Facility (MOF) Core Specification, janvier 2006. Version 2.0.

[Pro05] Projet Triskell, INRIA. MTL : Model Transformation Language. http ://modelware.inria.fr/rubrique8.html, 2005.

[SCF05] Jen-Sébastien Sottet, Gaelle Calvary, and Jean-Marie Favre. Ingénierie de l'Interaction Homme-Machine. In *Actes des premières journées sur l'ingénierie dirigée par les modèles*, page 16, Paris, juillet 2005.

[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.

[SL04] Jim Steel and Michael Lawley. Model-based test driven development of the tefkat model-transformation engine. In *Proceedings of ISSRE04 (International Conference on Software Reliability Engineering)*, St Malo, France, novembre 2004.

[SLTM91] Kari Smolander, Kalle Lyytinen, Veli-Pekka Tahvanainen, and Pentti Marttiin. Metaedit : a flexible graphical environment for methodology modelling. In *CAiSE '91 : Proceedings of the third international conference on Advanced information systems engineering*, pages 168–193, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[Sof] Objecteering Software. Objecteering. http ://www.objecteering.com/.

[Sol] Pathfinder Solutions. Pathmate : Transformation engine. http ://www.pathfindermda.com/.

[StOs00a] R. Soley and the OMG staff. MDA Model-Driven Architecture, novembre 2000. Online presentation http ://www.omg.org/mda/presentations.htm.

[StOS00b] R. Soley and the OMG Staff. Model-Driven Architecture. OMG Document - White paper, Draft 3.2, document available at www.omg.org, novembre 2000.

[Tae04] Gabriele Taentzer. AGG : A Graph Transformation Environment for Modeling and Validation of Software. In J. Pfaltz, M. Nagl, and B. Boehlen, editors, *proceedings of Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2004.

[TR03] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+ : defining and using domain-specific modeling languages and code generators. In *OOPSLA '03 : Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 92–93, New York, NY, USA, 2003. ACM Press.

[Tra] Laurence Tratt. Qvteclipse. http ://qvtp.org/downloads/qvtp-eclipse/.

[TV05] Christophe Tombelle and Gilles Vanwormhoudt. Models scripting with emf & javascript. http ://www.enic.fr/people/Vanwormhoudt/modelscripting/index.html, décembre 2005.

[UQ] UMT-QVT. http ://umt-qvt.sourceforge.net/.

[vDKV00] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages : An annotated bibliography. *ACM SIGPLAN Notices*, 35(6) :26–36, juin 2000.

[VJ04] D. Vojtisek and J.-M. Jézéquel. MTL and Umlaut NG - engine and framework for model transformation. *ERCIM News 58*, 58, juillet 2004.

[W3C04] W3C. XML Schema Part 0 : Primer Second Edition W3C Recommendation, octobre 2004. Version 1.1.

[W3C05a] W3C. XQuery 1.0 : An XML Query Language Version 2.0, novembre 2005. Candidate Recommendation.

[W3C05b] W3C. XSL transformations (XSLT) version 2.0, avril 2005. Working Draft 4.

[Xac] Xactium. Xmf-mosaic. http ://www.xactium.com.

[Zia04] Tewfik Ziadi. *Manipulation de lignes de produits en UML*. Thèse de doctorat, Université de Rennes I, Rennes, France, décembre 2004.

CHAPITRE 6

Conclusions

Rédacteurs : Philippe Collet, Thierry Coupaye.

Ce document avait un double objectif. Il devait d'une part permettre d'établir un vocabulaire et des éléments communs pour tous les partenaires du projet FAROS. D'autre part, il devait déterminer plus précisément le périmètre scientifique du projet, les tenants et les aboutissants des problématiques abordées (services, contractualisation, composition, ingénierie des modèles). Nous établissons dans cette dernière section une brève synthèse des études effectuées. Celles-ci nous permettent déjà de positionner les approches orientées services et orientées composants de manière complémentaire (cf. 6.1.2). Enfin, nous terminons ces conclusions en déterminant des premiers éléments directeurs pour la suite du projet.

6.1 Synthèse

6.1.1 Problématiques abordées

Nous avons d'abord étudié les approches orientées services et les standards industriels qui leur sont actuellement associées. Hormis la caractéristique principale du couplage faible entre les services, nous constatons une rapide évolution dans les standards et dans les approches elles-mêmes. Des besoins de structuration apparaissent notamment de l'approche SOA (à travers des concepts architecturaux
Nous constatons ainsi que des variations comme le *Service Oriented Computing* et le *Service Component Architecture* démontrent les besoins grandissants de structuration au sein de l'approche SOA ((*Service Oriented Architecture*) dès que sont notamment abordés les problèmes de *management* et d'orchestration dynamique. Ces besoins s'expriment principalement par l'incorporation de notions architecturales, largement récupérées des concepts de l'approche par composants. La section 6.1.2 effectue plus bas une comparaison entre SOA et *Component-Based Software Engineering* (CBSE).
Concernant les contrats, nous avons étudié un grand nombre de formalismes et de systèmes spécifiques aux différents paradigmes. Nous avons ainsi analysé différentes formes de contrats comportementaux — principalement à base d'assertions, d'automates, d'algèbres de processus et de diagrammes de séquence —, de contrats relatifs à la qualité de services — en s'intéressant à la fois aux langages de spécificaiton et aux systèmes de surveillance à l'exécution—, ainsi qu'à ceux relatifs à l'architecture. Nous avons aussi étudié les approches contractuelles spécifiques aux SOAs. Les formes de contrat se caractérisent par leur nature — comportementale, liée à la qualité de services, spécifique à l'architecture — et par les capacités des formalismes utilisés. Les capacités d'expressivité sont variables, comme le sont les techniques de vérification et la forme des vérifications possibles. Nous retenons que l'on peut, suivant les cas, vérifier qu'un contrat est lui-même cohérent, compatible — ou substituable — avec un autre, ou enfin conforme à une réalisation (une implémentation). Cette diversité dans les formes, les capacités, les propriétés vérifiées et les manières de les vérifier justifie pleinement la conception d'un modèle abstrait de contrats, tel qu'il doit être conçu dans le lot 2 du projet FAROS.
L'étude des formes de composition nous montre que la composition en tant que mécanisme

est primordial dans le développement logiciel, et bien plus encore dans les SOAs. L'utilisation de différents mécanismes de composition nécessite néanmoins un certain nombre de garanties, que des contrats appropriés pourraient justement fournir. A l'inverse, les contrats eux-mêmes ont besoin d'être composés selon différents points de vue.
La forte variabilité dans la forme des éléments que nous manipulons dans le projet FAROS (services, architectures, contrats, compositions) nécessite ainsi d'effectuer un grand nombre d'abstractions et de formaliser ces dernières dans des modèles appropriés. Un tour d'horizon des emplois de l'ingénierie des modèles a ainsi été effectué. Il confirme l'utilité de cette approche pour, d'une part, réaliser des modèles centrés sur les services, les contrats et les plates-formes d'exécution, et d'autre part, effectuer des transformations valides entre ces modèles.

6.1.2 Discussion sur les approches orientées services et composants

Les approches des architectures orientées services et des architectures à base de composants apparaissent essentiellement à ce jour comme concurrentes. Plusieurs raisons peuvent sans doute être évoquées dont une grande variabilité au sein même de chacune des approches (spécialement dans les composants), un probable manque de maturité (spécialement dans les services) et globalement une médiatisation organisé par des communautés différentes.
Cette section compare de manière synthétique les caractéristiques des deux approches SOA et CBSE en mettant en relief deux points qui à notre avis participent de beaucoup à la confusion dans la comparaison des deux approches et à la difficulté à les positionner : la question de l'intégration et la question des points de vue fonctionnels et techniques.

Caractéristiques de SOA Les caractéristiques proéminentes de l'approche SOA sont les suivantes :

Couplage faible Les services sont des *fonctions* accessibles uniquement au travers d'interfaces qui présentent les "services rendus" d'un service donné aux consommateurs de services. Les services ne présentent aucune dépendance dans leur réalisation (*i.e.* de dépendances entre services, les services pouvant avoir des dépendances vers d'autres composants entrant dans leur implantation).

Boîtes noires Un service n'ayant aucune connaissance sur l'architecture interne et l'implantation d'autres services (ni sur lui-même d'ailleurs), les services apparaissent réellement comme des boîtes noires dont seul "l'extérieur de la boîte" est visible. Cet extérieur est constitué des interfaces offertes par les services : SOA reprend et promeut le principe de séparation entre interfaces et implantations que l'on trouve dans la plupart des langages de programmation récents.

Sans état Un corollaire et même une condition nécessaire du couplage faible est que les services n'aient pas d'état. En tout état de cause — un service étant ultimement implanté par un (ou plusieurs) programme, il a bel et bien un état —, l'état d'un service n'a aucune incidence sur ses interactions avec d'autres services et cet état n'est ni accessible, ni manipulable, par les consommateurs de services (ou d'autres entités *e.g.* consoles d'administration).

Découverte Les services ne présentant aucune interdépendance intrinsèque *i.e.* n'ayant que très peu de moyen de connaître les services avec lesquels ils sont susceptibles d'interagir, les mécanismes de découverte de services (services de nommage, courtage, découverte) présentent une importance particulière.

Orchestration Les dépendances et a fortiori les interactions entre services n'apparaissant pas dans les services, il est nécessaire de spécifier "séparément des services aux-mêmes"

quand et comment ils interagissent. C'est le rôle dévolue à l'orchestration (NB : les concepts antérieurs de *workflows* et *process* étaient à peu près strictement équivalents).

Gros grain Cette caractéristique n'est pas strictement technique et intrinsèque à l'approche comme peuvent l'être les précédentes, néanmoins, le couplage faible et les contextes d'utilisation principaux de SOA (orchestration de web services sur Internet, orchestration d'objets/services en environnement domestique) induisant une granularité relativement figée et assez grosse des services (un service dans l'approche SOA sera rarement une table de hachage au fin fond d'un SGBD !).

Les bénéfices majeurs de l'approche SOA concernent la dynamicité, le passage à l'échelle et l'amélioration de la disponibilité. En effet, un service n'étant finalement qu'une fonction accessible à un point bien connu (typiquement un noeud dans un réseau), il est très naturel d'imaginer que de nombreux services rendant la même fonction puissent être instanciés et donc en exécution simultanément sur de multiples noeuds offrant par là même d'intéressantes possibilités pour la disponibilité de ces services et le passage à l'échelle de systèmes à base de services.
Les inconvénients majeurs actuels de l'approche SOA concernent essentiellement la sûreté de fonctionnement (en particulier la sécurité et la garantie de disponibilité) et la maintenabilité (en particulier l'administrabilité et la garantie de qualités de service).
Il est intéressant de remarquer que le coupage faible a un rôle central dans SOA. La plupart des autres caractéristiques de l'approche en découlent (pas d'états, découverte, orchestration), ainsi que ses avantages et inconvénients. La disponibilité est un cas intéressant dans ce contexte puisque, dans une vision optimiste, l'on peut penser qu'un service nécessaire au sein d'une orchestration sera toujours disponible (car instancié potentiellement de manière multiple) ; néanmoins, du fait même du couplage faible, rien de permet de le garantir (un service peut très bien être supprimé unilatéralement sur un noeud donné).

Caractéristiques de CBSE Les caractéristiques proéminentes de l'approche CBSE sont les suivantes :

Composition Un composant est une entité logicielle se conformant à un modèle de composants qui spécifie des règles de composition et d'interaction entre les composants. Il n'y a pas ici opposition frontale entre SOA et CBSE mais là ou SOA insiste sur l'orchestration, CBSE insiste sur la composition.

Composition structurelle Bien que les travaux du domaine s'intéressent à différents types d'interaction (synchrone, asynchrone, par signaux, etc.) et de composition (dont la composition comportementale qui n'est pas sans rapport avec la notion de contrat) ; les modèles de composant proposent fondamentalement des concepts et mécanismes pour contrôler les *liaisons* entre composants et en particulier les liaisons entre les composants et leurs sous-composants, *i.e.* la *structure hiérarchique* d'un système. Le couplage est donc bien plus fort entre composants qu'entre services mais plus faible que dans les approches précédentes, en particulier l'objet, en ce sens que les liaisons entre composants ne sont pas statiques ou "enfouies dans le code" mais sont au contraire externalisées et manipulables de manière programmatique et souvent dynamique.

Boîtes noires, blanches, grises, etc. Comme SOA, CBSE reprend et promeut le principe de séparation entre interfaces et implantations. Cependant, les composants sont également libres d'exposer ou non leur structure interne en termes de sous-composants - d'ou la caractérisation parfois employée de boîtes noires, blanches, voire grises, etc. On peut à ce sujet noter que ce débat (dû pour partie à la définition très connue du concept de composant par C. Szyperski comme "une boîte noire sans état...") au perdu beaucoup de son intensité initiale au sein de la communauté CBSE.

Réflection la plupart des composants, en tout cas dans les modèles les plus récents et les "plus avancés", offrent des mécanismes pour l'introspection et plus généralement la réflection. Ces mécanismes étant nécessaires à la composition structurelle, *i.e.* au contrôle des liaisons et donc généralement également au contrôle du cycle de vie (arrêt et redémarrage des composants), au contrôle des états, etc. Plus généralement, on retrouve ce besoin pour le contrôle des aspects techniques des composants qui motivent les modèles de composants industriels à la EJB.

Grain arbitraire L'approche CBSE est agnostique sur la granularité des composants qui peuvent être de taille complètement arbitraire, de la taille d'un driver dans un OS ou d'un pool jusqu'à un moniteur transactionnel ou un SGBD complet. En réalité, l'une des plus grande force de l'approche à composants et précisément de permettre un contrôle uniforme de toutes les ressources logicielles à tous les niveaux d'abstraction souhaités.

Etats L'approche CBSE est également globalement agnostique sur le fait que les composants n'aient ou pas un état - bien que dans ses domaines d'utilisation préférentielle (infrastructures logicielles techniques : OS, middleware, serveurs d'applications, etc.), les composants aient souvent un état. On retombe sans doute ici sur la discussion à propos des couplages plus ou moins faibles de SOA et CBSE.

Les bénéfices majeurs de l'approche à composants sont la flexibilité, la dynamicité (plus grande administrabilité et maintenabilité) et la prédictabilité des systèmes construits par assemblage : les architectures à composants fournissent des *abstractions structurelles* qui permettent de raisonner sur le *comportement* des systèmes logiciels (vis-à-vis de l'intégrité, sécurité, isolation, performances, QoS, temps-réel, etc.).
Les inconvénients majeurs actuels de l'approche CBSE concernent le manque de techniques garantissant formellement des propriétés prévisibles (*e.g.* temps-réel), le verrou principal concernant les compromis entre flexibilité et confiance attendues des systèmes construits par assemblage de composants.

Point de vue de l'intégration et de la réutilisation Un premier point qui complexifie le positionnement et plus globalement le débat entre SOA et CBSE, et qui souvent conduit à les opposer frontalement, concerne le problème de l'intégration qui constitue un important verrou industriel - en particulier dans le contexte de l'émergence d'Internet, et plus globalement par la convergence de l'informatique et des télécommunications, dans lequel les systèmes logiciels deviennent omniprésents, souvent critiques (coûts de développement, déploiement et maintenance, image de marque, time-to-market) et complexes. Cette complexité recouvre plusieurs aspects dont la complexité algorithme intrinsèque des systèmes (dûs par exemple à la distribution, aux fautes, à l'asynchronisme) mais également, et c'est ce qui nous intéresse le plus ici, une complexité en termes d'ingénierie dû à la grande hétérogénéité à tous les niveaux : équipements (réseaux et terminaux), protocoles réseaux, systèmes d'exploitation, middleware, services ; à la variabilité en termes d'environnement d'exécution (mobilité, nomadisme), à l'ouverture des systèmes (topologie dynamique).
Dès lors, pourquoi les approches SOA et CBSE apparaissent-elles si différentes et concurrentes ? Un élément de réponse que l'on peut proposer est que, si SOA et CBSE sont globalement des technologies pour l'intégration, elles ne concernent pas, in fine, le même type d'intégration :

– l'approche SOA concerne une intégration que l'on peut qualifier de *a posteriori* dans laquelle on ne s'intéresse guère à la façon dont les "composants" sont construits et déployés. les "composants" existent et doivent être intégrés quasiment tels quels - les seule contraintes portant sur le respect de protocoles de découverte et de communication (*e.g.* SOAP, UPnP). Les "composants" intégrés sont globalement hétérogènes, à granularité fixe et plutôt grosse. Le couplage entre et le contrôle sur les "composants" est faible à nul. Ce type d'intégration

correspond à un développement opportuniste dans lequel des services existants et gérés par des entités administratives multiples (*e.g.* services sur le web) sont intégrés au sein d'orchestrations. Il est également intéressant de noter que l'intégration au sein d'orchestration correspond bien à une réutilisation mais une "réutilisation de code en exécution". Un service est souvent (dans les approche de type UPnP, un peu moins dans les approches de type web services) un singleton. La réutilisation concerne le "partage d'instance" : un même service (singleton) peut être intégré dans plusieurs orchestrations simultanément.

– l'approche CBSE concerne une intégration que l'on peut qualifier de *a priori* dans laquelle les composants sont explicitement construits pour être conformes à un modèle architectural. Les approches à composants ont également souvent vocation à couvrir le cycle de vie complet du logiciel (développement, déploiement, administration, maintenance). Les composants intégrés sont homogènes mais à granularité arbitraire. Le couplage entre et le contrôle sur les composants est fort mais dynamique. Ce type d'intégration correspond à une démarche systématique de développement et de déploiement. La réutilisation dans l'approche CBSE concerne essentiellement la réutilisation de code en tant que tel, *i.e.* plutôt des "types" (ou "librairies") de composants qui peuvent être instanciés de manière multiple. Bien sûr le partage d'instances comme dans l'approche SOA est également possible. Enfin, certains modèles à composants comme Fractal, permettant également le partage d'instances au sein d'un même système mais possiblement à différentes localisations dans l'assemblage du système.

Points de vue fonctionnel et technique Un second point qui brouille le débat entre SOA et CBSE touche au fondement même des deux approches puisqu'elle concerne l'architecture des systèmes logiciels. En effet, cela va sans dire mais cela va mieux en le disant : l'approche SOA ne traite que du point de vue fonctionnel de l'architecture logicielle ; tandis que l'approche CBSE couvre à la fois les points de vue fonctionnel (quels fonctions/services constituent le système cible), structurel (comment les fonctions/services sont organisés) et technique (intégration d'aspects techniques/non fonctionnels voire projection sur infrastructure technique pour le déploiement) avec souvent, il est vrai, une focalisation sur les points de vue structurel et technique.

6.2 Éléments directeurs

Nous esquissons ici les premiers besoins pour la suite du projet FAROS.

Établir et maintenir un ensemble de définitions pour les éléments manipulés. Une première étape est bien évidemment d'établir les définitions de service et d'architecture orientée services. C'est en partie l'objectif du prochain livrable, F-1.2, qui établira les spécifications d'une architecture pour la contractualisation de services. Celles-ci détailleront notamment les particularités des services, l'expression de leur composition et les contrats adaptés à la composition fiable de services et de composants logiciels. Cette spécification des besoins permettra de concevoir par la suite le modèle abstrait de contractualisation qui va constituer la contribution majeure du 2ème lot du projet.

Fonder les définitions et les premiers travaux sur des exemples significatifs extraits des applications de validation. La diversité de métiers et d'architectures envisagés dans les trois applications de validation du 4ème lot du projet FAROS vont nous permettre de couvrir de manière significative notre problématique. Différentes réalisations de SOAs, de contrats et de composition sont effectivement nécessaires pour poser les définitions, effectuer les abstractions correspondantes, puis établir les modèles que nous voulons réaliser. L'étude de l'état

de l'art nous montre qu'un nombre conséquent de caractéristiques peuvent être déterminées pour les formes de contrat (granularité, portée métier ou technique, etc.) et de composition.

Confronter le modèle abstrait et les exemples aux capacités des plates-formes cibles. C'est l'objectif du 3ème lot du projet FAROS, dans lequel seront étudiés, dans les plates-formes cibles, les types de contrats et de compositions disponibles. Ces études seront réalisées en tenant compte de la complémentarité possible entre SOA et CBSE. Ces deux approches peuvent être combinées, mais dans différentes parties du système considéré. L'approche CBSE est utilisée typiquement pour l'implantation de composants métiers dans des serveurs d'applications de type "back office", tandis que l'approche SOA est utilisée pour intégrer ces composants métiers en donnant une vue agglomérée et pour interagir avec d'autres services dans l'organisation considérée et a fortiori avec des services extérieurs et disponibles sur Internet. Il faudra aussi envisager d'intégrer SOA et CBSE. Cette proposition consiste à pousser plus loin l'intégration des approches SOA et CBSE pour tirer pleinement partie de leur complémentarité. La proposition s'inscrirait globalement dans des approches comme SCA ou OSGi, mais là ou ces approches ont tendance à "mélanger" les concepts et mécanismes provenant de SOA et CBSE avec comme résultats des modèles pas toujours homogènes et ni très faciles à manipuler ; il serait possible de conserver intactes les deux vues complémentaires grâce :

– à une ingénierie des services en termes de composants, *i.e.* une implantation sous forme de composants des services,
– à un mécanisme d'*exposition* de certains composants sous forme de services permettant une gestion du cycle de vie (en particulier déploiement, administration, reconfiguration) à la CBSE tout en offrant une couche de communication faiblement couplée à la SOA entre ces services/composants.

Cette dernière proposition serait intéressante, d'une part pour réutiliser assez directement les travaux sur la contractualisation de composants, et d'autre part pour pouvoir se focaliser sur d'autres aspects tels que les contrats de type QoS ou la composition de contrats entre services et composants. La mise en œuvre de cette approche complémentaire devra aussi être conçue de façon à exploiter au mieux les techniques et mécanismes de programmation par aspects fournies dans certaines des plates-formes cibles.

Confronter le modèle abstrait et les exemples à l'approche IDM proposée dans le projet. Ceci se fera dans les parties en aval du 2ème lot. L'approche et l'outillage envisagés pour suivre une démarche orientée modèles devront avoir la capacité à transformer des contrats exprimés dans un modèle de services métiers en des contrats adaptés à la supervision par des services et des composants logiciels basés sur des technologies précises.

ANNEXE
A

Acronymes

ADL : Architecture Description Language
AOP : Aspect Oriented Programming
AOSD : Aspect Oriented Software Development
BPEL : Business Process Execution Language
BPEL4WS : cf. BPEL (for Web Service)
BPELWS : cf. BPEL (for Web Service)
BPML : Business Process Modeling Language
BPMN : Business Process Modeling Notation
CBSE : Component-Based Software Engineering
CCL-J : Component Constraint Language for Java
CCM : CORBA Component Model
CCS : Calculus of Communicating Systems
CORBA : Common Object Request Broker Architecture
CQML : Component Quality Modeling Language
CSP : Communicating Sequential Processes
DSL : Domain Specific Language
ESB : Entreprise Service Bus
FSP : Finite State Processes
IDM : Ingénierie Dirigée par les Modèles
ISL : Interaction Specification Language
ISL4Wcomp : Interaction Specification Language For WComp
JAC : Java Aspect Components
JML : Java Modeling Language
JMX : Java Management Extensions
LTS : Labelled Transition System
MDA : Model Driven Architecture
MDE : Model Driven Engineering
MOF : Meta-Object Facility
MSC : Message Sequence Charts
OCL : Object Constraint Language
ODP : Open Distributed Processing
OSGi : Open Services Gateway Initiative
OWL : Ontology Web Language
OWL-S : Ontology Web Language for Service

QML : QoS Modeling Language
QoS : Quality of Service
QoSCL : Quality of Service Constraint Language
RM-ODP : Reference Model of Open Distributed Processing
SCA : Service Component Architecture
SLA : Service Level Agreement
SLAng : SLA Language
SOAP : Simple Object Access Protocol
SOA : Service Oriented Architecture
SOC : Separation Of Concerns
TLA : Temporal Logic of Actions
UDDI : Universal Description, Discovery and Integration
UML : Unified Modeling Language
Wcomp : Behavioral Composition for Wearable and Situated Computing
Ws-Agreement : Web Services Agreement Specification
WSCA : Web Services Conceptual Architecture
WS-CDL : Web Services Choregraphy Description Language
WSCI : Web Services Choregraphy Interface
Ws-Context : Web Services Context Specification
WSCL : Web Services Conversation Language
WSDL : Web Services Description Language
WSFL : Web Services Flow Language
WSLA : Web Service Level Agreement
Ws-Negotiation : Web Services Negotiation Specification
WSOL : Web Service Offerings Language
Ws-Policy : Web Services Policy Framework