Modélisation des architectures logicielles dynamiques : application à la gestion de la qualité de service des applications à base de services web

Francisco José Moo-Mena

THÈSE

préparée au Laboratoire d'Analyse et d'Architecture des Systèmes en vue de l'obtention du Doctorat de l'Institut National Polytechnique de Toulouse

École doctorale : Informatique et télécommunications

Spécialité : Sûreté de logiciel et calcul à haute performance

Par
Francisco José MOO MENA

MODÉLISATION DES ARCHITECTURES LOGICIELLES DYNAMIQUES :

Application à la gestion de la qualité de service des applications à base de services Web

JURY
M. Christian FRABOUL Professeur à l'INPT/ENSEEIHT - Président
M. Khalil DRIRA HDR Chargé de recherche au LAAS-CNRS - Directeur de thése
M. Karim DJOUANI Professeur à l'Université Paris 12/Val de Marne - Rapporteur
M. Flavio OQUENDO Professeur à l'Université de Bretagne-Sud - Rapporteur
M. Mohamed JMAEIL Professeur à l'ENIS, Sfax, Tunisie - Examinateur
M. Michel DIAZ Directeur de recherche au LAAS-CNRS - Examinateur

Soutenue le 4 Avril 2007

A la mémoire de mon père Francisco,
à ma mère Teresa,
tous les deux m'ont mis sur le bon chemin.
A mon épouse Cristina et à mon fils Emmanuel,
tous les deux sont y arrivés, par la suite, pour
m'accompagner et motiver mes pas.
Je vous aime à vous tous !

Remerciements

Ce travail a été réalisé au Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique (LAAS-CNRS). Ainsi, je tiens à remercier Messieurs Malik Ghallab et Raja Chatila, ancien et actuel Directeur du LAAS pour m'avoir accueilli dans ce laboratoire.

Je tiens aussi à remercier Messieurs J.P. Courtiat et F. Vernadat respectivement, ancien et actuel responsable du groupe de recherche *Outils Logiciels pour la Communication (OLC)*, pour m'avoir accueilli au sein de ce groupe.

Ce travail a été réalisé grâce au support financier du « Programa para el Mejoramiento del Profesorado (PROMEP) » et de la « Universidad Autónoma de Yucatán (UADY) » deux institutions publiques du Mexique.

Je tiens à exprimer ma profonde reconnaissance à M. Khalil DRIRA Chargé de Recherche au LAAS-CNRS, qui a encadré ma thèse. Que sa qualité humaine et professionnelle, sa disponibilité et ses conseils si précieux, puissent retrouver ici un geste de remerciement le plus sincère.

J'adresse mes plus sincères remerciements à M. Michel DIAZ Directeur de recherche au LAAS-CNRS, qui a bien voulu participer comme co-directeur de ma thèse.

Je tiens à exprimer ma sincère gratitude à Messieurs :

Karim DJOUANI - Professeur à l'Université Paris 12, Val de Marne,

Flavio OQUENDO - Professeur à l'Université de Bretagne-Sud, Vannes,

Christian FRABOUL - Professeur à l'INPT/ENSEEIHT, Toulouse,

Mohammed JMAIEL - Professeur à l'ENIS, Sfax, Tunisie.

pour l'honneur qu'ils m'ont fait en acceptant de participer au jury de thèse. Je remercie, en particulier, à M. DJOUANI et M. OQUENDO qui ont accepté la charge de Rapporteur de ce manuscrit, et également à M. FRABOUL qui a accepté de présider le jury.

Merci également à Olga Nabuco et Renê Pergoraro pour sa précieuse aide dans certains travaux de cette thèse et à Jean François Le Neal et Patricia Galazzo pour la relecture du manuscrit.

Je remercie l'ensemble du groupe OLC, permanents, doctorants et stagiaires pour l'accueil chaleureux. Et en particulier à mon collègue Karim Guennoun pour les discussions scientifiques et l'amitié cultivée au sein du groupe OLC. Également, pour tous mes collègues tunisiens qui sont venus à Toulouse à plusieurs reprises et qui ont toujours montré leurs qualités humaines et professionnelles. Et à ma compatriote et collègue Sara MOTA, pour toutes ses réflexions qui ont toujours rendu plus agréable mon séjour dans le bureau 28A.

Un grand merci à mes deux familles mexicaines, MOO MENA et ARZAPALO CENTENO pour son soutien permanent le long de mon séjour en France. De même, à mon cher parrain Carlos ENCALADA et à ma chère marraine et amie de toute la vie Esmeralda BARBA.

A tous les amis mexicains, brésiliens, français, polonais, libanais et tunisiens que j'ai eu la chance de rencontrer à Toulouse, en général, et au LAAS en particulier. Le nombre est vraiment long, la qualité humaine supérieure, et les blagues innombrables, et bien que je ne les liste pas ici, je suis certain que chacun s'y reconnaîtra.

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Introduction

Le développement de la technologie des services Web d'aujourd'hui, connaît de nouvelles formes d'interaction, notamment à travers sa mise en place sur des dispositifs portables pour utilisateurs nomades, pour des applications industrielles et d'entreprise. Ces nouvelles technologies offrent un potentiel important pour le développement de services complexes assistant les activités humaines (pour le travail, à la maison, et pour le divertissement) et dans des contextes mobiles, en créant des réseaux de services Web coopératifs.

Bien que la technologie des services Web offre des mécanismes qui permettent de traiter le problème de l'hétérogénéité des composants, d'autres exigences nécessitent des efforts supplémentaires pour obtenir une solution à la problématique imposée par les nouvelles applications. En effet, la technologie de service Web offre un potentiel important pour la création des applications complexes et dans des domaines divers, en composant des réseaux de service coopératifs. Néanmoins, ces réseaux sont ouverts et entraînent une dynamique dans leur composition, ce qui se traduit par la découverte et l'utilisation dynamiques des services. Ces réseaux sont aussi sensibles à la complexité liée à la gestion de la dynamique de ce type de systèmes.

Dû à cette complexité, les applications, en général, et les architectures en particulier, ont besoin de s'adapter aux changements tels que :

- les variations de la bande passante,
- les changements dans les terminaux d'accès,
- les variations du nombre d'utilisateurs.

Ainsi, les applications ont besoin de nouvelles méthodologies et techniques afin d'accomplir des exigences telles que la provision de la qualité de service (QdS).

Cette problématique inhérente aux applications distribuées de grande taille, avait été initialement identifiée par la communauté scientifique internationale au début des années 2000. A cette époque, la société IBM a annoncé une nouvelle initiative appelée « Autonomic Computing », Kephart et Chess (2003), envisageant le développement de systèmes capables de s'auto-gérer. Cette idée a été inspirée du fonctionnement du corps humain qui est pourvu d'une telle capacité. Afin d'atteindre ses objectifs, l'initiative « Autonomic Computing » a été décomposée en quatre sous-disciplines :

1. Le *self-configuring* considère la reconfiguration de composants et de systèmes en définissant des politiques de haut-niveau.
2. Le *self-healing* s'intéresse à l'auto-détection, le diagnostic et la réparation des problèmes au niveau logiciel et matériel.
3. Le *self-optimizing* implique l'auto-réglage de paramètres au niveau service.
4. Le *self-protecting* considère la protection des systèmes contre des attaques et des malveillances.

Les objectifs de notre travail partagent ceux des deux premières sous-disciplines. Concrètement, nos travaux considèrent la gestion des architectures logicielles dynamiques comme un outil pour l'adaptation des applications à base de services Web, avec l'objectif de proposer des mécanismes d'aide à la gestion de la qualité de service.

Dans la littérature, deux approches complémentaires sont considérées. La première se focalise sur le comportement de l'application, et vise à reconfigurer soit le service, soit le « workflow » impliqué lorsqu'une inconsistance, qui pourrait entraîner une dégradation de la qualité de service est observée. Une seconde approche (dite architecturale), vise à surveiller et à mesurer les interactions entre services Web au niveau architectural et à réagir par reconfiguration de l'architecture lorsqu'une une dégradation de la qualité de service est constatée. Nos travaux s'inscrivent dans cette seconde approche.

1.1 Objectifs et contributions

Les objectifs de cette thèse sont de proposer :

– Des techniques de modélisation afin d'assister les développeurs dans la conception des architectures dynamiques et adaptatives pour les applications à base de services Web.

– Des techniques pour la gestion de la qualité de service pour les applications du même type.

Dans un cadre local, cette thèse s'inscrit dans les thèmes de recherche du groupe OLC du LAAS-CNRS, en particulier ceux des axes de recherche « Composants et services de coopération » et « Architecture et protocoles de communication ». Ces deux axes visent le développement des méthodologies et des techniques pour la conception et la mise en œuvre d'architectures logicielles distribuées dynamiques et adaptatives, ainsi que la définition des protocoles de coordination associés.

Dans un contexte national, les objectifs de cette thèse sont proches de l'action spécifique du CNRS sur l'adaptabilité dynamique du GDR - Architecture, Systèmes et Réseaux.

Dans un contexte européen, les objectifs de cette thèse s'inscrivent dans le cadre du projet IST WS-DIAMOND. Ce projet étudie le développent des méthodologies et des techniques pour la conception et le développement des systèmes à base de service Web suivant une approche de « self-healing ».

Les contributions principales de cette thèse peuvent être résumées comme montré dans la figure 1.1. Notre but principal étant la reconfiguration dynamique des applications à base de services Web (*1*), nous proposons une approche guidée par la QdS afin de prévenir la dégradation de la QdS des applications à base de services Web (*1.1*). Notre proposition de reconfiguration dynamique suit une approche architecturale (*1.2*), considérée complémentaire à l'approche comportementale (*1.3*) qui est traitée par d'autres partenaires dans le cadre du projet WS-DIAMOND. Dans notre approche architecturale, les applications sont structurées par quelques types architecturaux (*1.2.1*) définis selon un méta-modèle étendant la vue de déploiement d'UML. De même, l'approche architecturale est formalisée par des mécanismes de transformation de graphes (*1.2.2*). Ces mécanismes sont utilisés pour la définition de règles de base (*1.2.2.1*) et en combinant celles-ci, nous définissons des règles de reconfiguration architecturale (*1.2.2.2*). Notre approche principale de reconfiguration dynamique a été éprouvée par deux scénarios applicatifs suivants un style orienté service, à savoir : la Revue coopérative (*1.4*) et le FoodShopping (*1.5*), tous les deux font partie du Deliverable 1.1 du projet WS-DIAMOND et ont été publiés dans le cadre de deux conférences internationales : WebIST 07 et ISCC 07.

Concernant les aspects de prévention de la dégradation de la QdS des applications à base de services Web (*1.1)*), nous proposons une approche basée sur les ontologies (*1.1.1*) permettant l'identification de différents cas de dysfonctionnement au travers de la définition des règles d'inférence. L'ontologie et les règles d'inférences ont été définies en utilisant les outils Protégé et Jess (*1.1.1.1*), respectivement. Notre approche de prévention de la dégradation de la QdS a été éprouvée sur le scénario de la revue

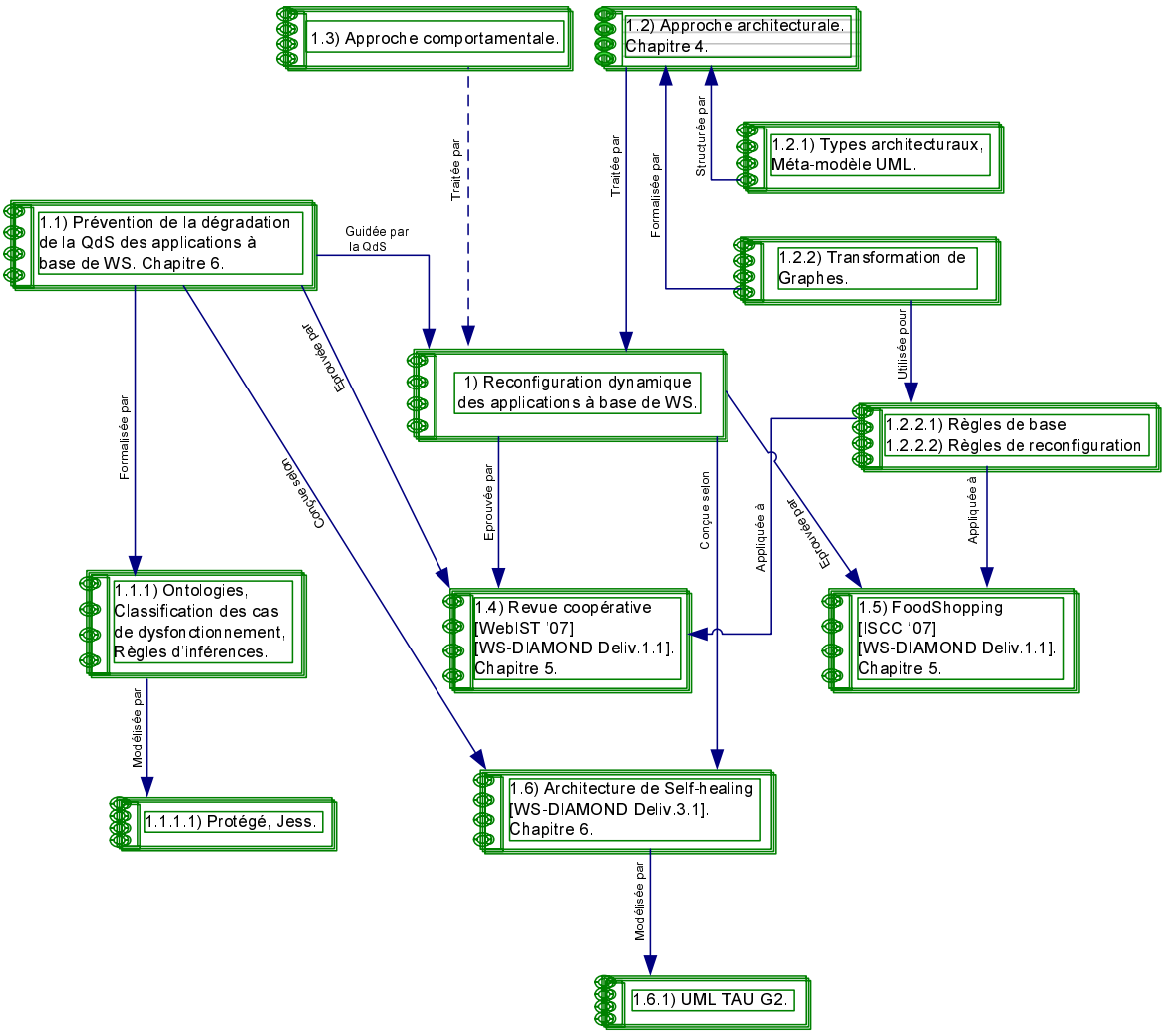


FIG. 1.1 – Illustration de l'articulation des différentes contributions de la thèse.

coopérative, instancié au cas d'un système de gestion de conférences. De même, nous proposons une architecture de « Self-healing » (*1.6*) pour la gestion de la QdS conçue dans le cadre du projet WS-DIAMOND. Nous avons modélisé et validé par simulation cette architecture dans différents scénarios en utilisant l'outil UML Tau G2.

1.2 Plan du mémoire

Nous avons structuré ce mémoire en deux parties. La première partie concernant l'état de l'art est composée de deux chapitres :

Le chapitre 2 présente une étude de synthèse sur les architectures logicielles ainsi que leurs définitions. Ensuite, une introduction sur les Langages de Description des Architectures (ADLs) est présentée. Une section particulière est consacrée à UML en tant que langage de description des architectures. Nous abordons par la suite le cas des architectures orientées services et leur application à la technologie des services Web. Une dernière section est consacrée aux architectures dynamiques.

Le chapitre 3 traite de l'adaptabilité des architectures logicielles selon deux axes : un premier qui aborde les travaux liés à la reconfiguration des architectures et un second qui traite des architectures de « self-healing ». Dans ces deux axes, nous consacrons une attention particulière aux travaux traitant des architectures à base de service Web.

La deuxième partie concernant nos contributions est divisée en trois chapitres :

Le chapitre 4 propose la modélisation des architectures logicielles dynamiques pour les applications à base de services Web. Concernant la description architecturale, de nouveaux éléments architecturaux sont introduits, notamment : le Rôle coopératif, la Catégorie de service et la Classe de service. Ces éléments sont caractérisés par le biais d'un métamodèle qui étend la vue de déploiement d'UML. Les instances de services Web et leurs interactions sont basées sur les diagrammes de composants d'UML. Ensuite, des règles de base pour la dynamique de l'architecture sont définies en se basant sur la technique de réécriture de graphes. Pour la description des architectures et des règles, des schémas XML ont été définis et validés en utilisant le logiciel XML Spy[1]. Puis, à partir des règles de base, nous avons défini des actions de reconfiguration des architectures, notamment pour les actions de duplication et de substitution.

Le chapitre 5 présente deux scénarios d'application pour valider notre approche de

[1] http ://www.altova.com/products/xmlspy/xml_editor.html

modélisation décrite dans le chapitre 4. Le premier scénario traite du processus de revue coopérative, appliqué au cas plus spécifique d'un système de gestion de publications scientifiques. Le deuxième scénario considère le processus d'une chaîne de production appliqué au cas plus spécifique d'un magasin de produits alimentaires en ligne. Dans les deux scénarios, sont définis les acteurs, le « workflow » (sous forme d'activités) et l'architecture impliquée selon notre approche. De même, les actions de reconfiguration sont décrites et appliquées lorsqu'un constat de dégradation de la QdS est identifié.

Le chapitre 6 propose une infrastructure pour la gestion de la QdS des applications à base de services Web. D'abord nous introduisons la notion d'ontologie et ses langages et ses formes de représentation. Ensuite, afin de gérer la QdS dans les applications à base de services Web, nous proposons une classification des dysfonctionnements pour la QdS. Cette classification a été appliquée au cas de la revue coopérative et a été formalisée et implémentée par le biais d'une ontologie en utilisant le logiciel Protégé. Finalement, nous définissons une architecture pour la gestion de la QdS. Cette architecture reprend et adapte quelques éléments des architectures « self-healing ». Elle a été validée par le moyen de l'outil de modélisation UML Tau G2.

Ce mémoire ce termine par un ensemble de conclusions générales et de perspectives des travaux en cours et à venir.

Première partie

Etat de l'art

Chapitre 2

Architectures logicielles

2.1 Introduction

Les architectures logicielles sont considérées comme une sous-discipline du génie logiciel. Une architecture est considérée comme l'organisation nécessaire d'un système caractérisé par ses composants, leurs relations et avec l'environnement, et les principes qui guident leur conception et évolution. Les architectures logicielles forment la colonne vertébrale pour construire des logiciels complexes et de grande taille. La description des architectures permet d'avoir l'abstraction nécessaire pour modéliser les systèmes logiciels complexes durant leur développement, déploiement et évolution.

2.1.1 Historique

Les concepts qui font naître les architectures logicielles, telles qu'elles sont connues aujourd'hui, remontent aux débuts de la discipline du génie logiciel. Les travaux pionniers de Dijkstra (1968) sur la matière ont proposé une structuration impérative d'un système avant même de se lancer à écrire des lignes de code. De même, les travaux de cet auteur mettent en évidence le besoin de la notion de « niveaux d'abstraction » dans la conception de systèmes à grande taille. Bien que Dijkstra n'utilise pas le terme « architecture » dans l'ensemble de ses travaux les notions sorties de ceux-ci représentent une base pour la définition des architectures logicielles contemporaines.

On cite souvent Sharp (1970) par ses commentaires versés en 1969 en relation avec les travaux précédents de Dijkstra : « Je pense qu'on a quelque chose en plus du génie

logiciel, quelque chose qu'on a discuté sommairement mais qu'il va falloir regarder d'avantage, c'est le sujet des architectures logicielles ». Pour illustrer ses mots, il donne l'exemple du système OS/360 : « ce système représente un beau travail d'ingénierie puisqu'il a été très bien codé. Par contre, il est constitué d'un ensemble amorphe de programmes parce qu'il n'a pas été conçu par un architecte ».

Au début des années 70 les travaux de Parnas (1972) ont introduit le concept de « modularité » qui propose d'améliorer la souplesse et le contrôle conceptuel du logiciel en réduisant le temps de développement des systèmes. Il a montré l'importance d'une bonne structuration des systèmes dans les étapes de conception dont il propose certaines idées afin d'atteindre un niveau de structuration adéquat. La notion de « modularité » considère les décisions de conception qui doivent se faire avant d'initier les travaux de mise en œuvre. Plusieurs avantages en résultent :

- le temps de développement peut se réduire puisque différents groupes pourraient travailler dans différents modules avec un faible besoin de communication,
- il est possible de faire des changements importants sur un module sans avoir besoin de retoucher les autres,
- afin de faciliter la compréhension du système il est possible d'analyser les modules séparément.

Une différence importante entre ces deux auteurs est le fait que les notions introduites par Dijkstra se sont focalisées sur des aspects de programmation (niveau de mise en œuvre) alors que les travaux de Parnas se sont centrés sur les niveaux de conception.

C'est incontestablement dans les années 90 que les architectures logicielles commencent à avoir un essor important. Les idées de base telles que donner de l'importance aux décisions prises dans les premières étapes du développement du système ainsi que de l'importance d'une définition correcte de la structure du système ont joué un rôle déterminant dans l'évolution de la discipline.

Un des premiers papiers qui traite les architectures logicielles en tant que nouvelle discipline et en termes plus proches aux notions actuelles est celui de Perry et Wolf. (1992). Dans ce papier, les auteurs font une analogie des architectures logicielles avec l'architecture des bâtiments. Ils proposent la définition des architectures logicielles par le biais de l'expression suivante :

$$Architecture\ logicielle = Eléments, Forme, Raisonnement$$

C'est-à-dire qu'une architecture est un ensemble d'éléments avec une forme particulière. Il existe trois classes d'éléments :

1. Eléments de traitement,

2. Eléments de données, et

3. Eléments de connexion.

Les éléments de traitements sont les composants qui agissent sur les éléments de données. Les éléments de données stockent l'information à traiter, et les éléments de connexion permettent de relier les différentes parties de l'architecture. La forme représente des propriétés et des relations pondérées. La pondération permet d'indiquer l'importance de la propriété ou la relation, ou la nécessité de choisir d'entre plusieurs alternatives. Le raisonnement représente la motivation pour choisir les styles architecturaux, les éléments et la forme.

Ces idées ont été à la base de la définition des premiers Langages de Description d'Architectures (ADL), tels que Wright, Rapide, Darwin,... La figure 2.1 adaptée de Kruchten *et al.* (2006) résume, par des faits marquants, l'évolution des architectures logicielles depuis les années 90. Faute d'accord sur un ADL universel, vers la fin des années 90, c'est UML (Langage de Modélisation Unifié) qui a été introduit comme standard de modélisation et avec ceci des efforts pour l'adopter en tant qu'ADL. D'autres initiatives ont été lancées visant un langage standard pour les ADLs, c'est le cas par exemple du Langage de Marquage pour la Description d'Architectures (ADML), Group (2000), qui comme d'autres, considère XML (Langage de Marquage Étendu) comme langage standard pour la description des architectures.

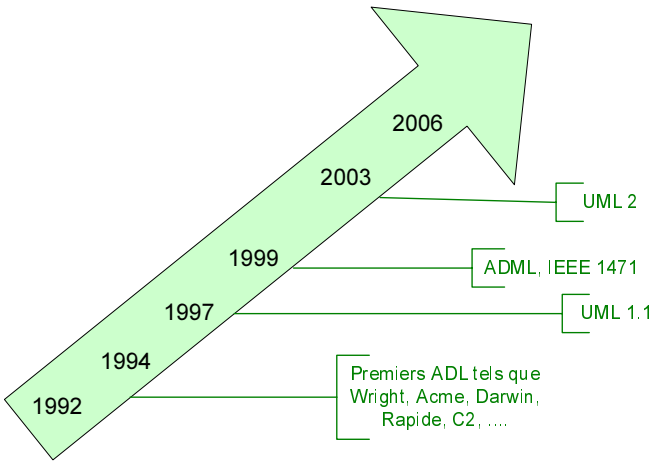


FIG. 2.1 – L'évolution des architectures logicielles.

2.1.2 Définition

A présent il n'existe pas un consensus sur une définition des architectures logicielles. Dans la littérature on pourrait retrouver quelques dizaines de définitions[1]. Devant une telle perspective nous listerons par la suite quelques définitions tirées de la littérature avec le but de donner une idée plus claire des principaux intérêts portés par cette discipline.

Selon Garlan et Shaw (1993), le niveau de conception des architectures logicielles va au delà des algorithmes et des structures de données : « la conception et la spécification de la structure globale d'un système représente un nouveau type de problème ». Les tâches comprennent :

1. l'organisation et le contrôle structural,
2. la définition de protocoles de communication, de synchronisation, et d'accès aux données,
3. l'affectation des fonctionnalités aux éléments de conception,
4. la distribution physique,
5. la composition des éléments de conception,
6. la mise à l'échelle et la performance, et
7. la sélection entre différentes alternatives de conception.

Une définition assez récente est celle de Kruchten *et al.* (2006) : « L'architecture logicielle implique la structure et l'organisation par lesquelles des composants interagissent pour créer de nouvelles applications, possédant la propriété de pouvoir être les mieux conçues et analysées au niveau système ».

Bien qu'il n'y ait pas une définition amplement acceptée, il existe des efforts de standardisation notamment à travers les travaux menés par l'IEEE sous la norme IEEE 1471-2000, IEEE (2000). Selon cet organisme : « L'architecture est définie par la pratique comme l'organisation fondamentale d'un système, intégrée par ses composants, leurs relations entre-eux et avec l'environnement, et les principes qui guident sa conception et son évolution ». Dans ce manuscrit, nous adoptons cette dernière définition.

[1] A l'heure actuelle il existe même un site Web, celui de l'Université Carnegie-Mellon aux Etats-Unis (http ://www.sei.cmu.edu/architecture/definitions.html) qui invite les visiteurs à proposer leur propre définition.

2.2 Description des Architectures

2.2.1 Approches ADL

Les premiers travaux réalisés pour décrire les architectures logicielles ont fait naître les langages de description d'architecture. En accord avec la notion d'architecture logicielle, l'idée est de fournir une structure de haut niveau de l'application plutôt que l'implantation dans un code source spécifique, Vestal (1993). Il faut préciser qu'il n'existe pas une définition unique des ADL, mais en général on accepte qu'un ADL fournit un modèle explicite de composants, de connecteurs et de leur configuration. Optionnellement, un ADL pourrait fournir des outils pour la génération de code source basés sur les architectures et pour la gestion de l'évolution des applications.

Depuis le début des années 90 bon nombre d'ADLs ont vu le jour. Parmi les plus représentatifs correspondant à une première génération nous trouvons notamment : Darwin Magee *et al.* (1995); Magee et Kramer (1996), Rapide Luckham *et al.* (1995), Wright Allen (1997); Allen et Garlan (1997). Alors que l'on peut considérer comme partie d'une deuxième génération aux ADLs : xADL Dashofy *et al.* (2001) et π-ADL Oquendo (2004). Chacun propose une architecture à sa manière, les uns privilégiant les éléments architecturaux et leur assemblage structurel et les autres s'orientant vers la configuration de l'architecture et la dynamique du système.

Face à une telle variété et au manque de consensus Medvidovic et Taylor (2000) ont fait une classification. Nous reprenons leurs définitions et les caractéristiques par la suite de la section.

Le composant

Le composant est une unité de calcul ou de stockage. Il peut être simple ou composite, et sa fonctionnalité peut aller de la simple procédure à une application complète. Le composant est considéré comme un couple spécification-code : la spécification donne les interfaces, les propriétés du composant ; le code correspond à la mise en œuvre de la spécification par le composant.

Les caractéristiques globales d'un composant sont les suivantes :

– L'*interface* d'un composant est la description de l'ensemble des services offerts

et requis par le composant sous la forme de signature de méthodes, de type d'objets envoyés et retournés, d'exceptions et de contexte d'exécution. L'interface est un moyen d'expression des liens du composant ainsi que ses contraintes avec l'extérieur.

– *Le type* d'un composant est un concept représentant l'implantation des fonctionnalités fournies par le composant. Il s'apparente à la notion de classe que l'on trouve dans le modèle orienté objet. Ainsi, un type de composant permet la réutilisation d'instances de même fonctionnalité soit dans une même architecture, soit dans des architectures différentes. En fournissant un moyen de décrire, de manière explicite, les propriétés communes à un ensemble d'instances d'un même composant, la notion de type de composant introduit un classificateur qui favorise la compréhension d'une architecture et de sa conception.

– *La sémantique* du composant est exprimée en partie par son interface. Cependant, l'interface, telle qu'elle est décrite ci-dessus, ne permet pas de préciser complètement le comportement du composant. La sémantique doit être enrichie par un modèle plus complet et plus abstrait permettant de spécifier les aspects dynamiques ainsi que les contraintes liées à l'architecture. Ce modèle doit garantir une projection cohérente de la spécification abstraite de l'architecture vers la description de son implantation avec différents niveaux de raffinements. La sémantique d'un composant s'apparente à la notion de type dans le modèle orienté objet.

– *Les contraintes* définissent les limites d'utilisation des composants. Une contrainte est une propriété devant être vérifiée pour un système ou pour une de ces parties. Si celle-ci est violée, le système est considéré comme incohérent. Elle permet ainsi de décrire de manière explicite les dépendances des parties internes d'un composant comme la spécification de la synchronisation entre composants d'une même application.

– *L'évolution* d'un composant doit être simple et s'effectuer par le biais de techniques comme le sous typage ou le raffinement. Un ADL doit favoriser la modification de ses propriétés (interface, comportement, implantation) sans perturber son intégration dans les applications déjà existantes.

– *Les propriétés non fonctionnelles* doivent être exprimées à part, permettant ainsi une séparation dans la spécification du composant des aspects fonctionnels (aspects métiers de l'application) et des aspects non fonctionnels ou techniques (aspects transactionnels, de cryptographie, de qualité de service). Cette séparation

permet la simulation du comportement d'un composant à l'exécution dès la phase de conception, et de la vérification de la validité de l'architecture logicielle par rapport à l'architecture matérielle et l'environnement d'exécution.

Le connecteur

Le connecteur modélise un ensemble d'interactions entre composants. Cette interaction peut aller du simple appel de procédure distante aux protocoles de communication. Tout comme le composant, le connecteur est un couple spécification-code : la spécification décrit les rôles des participants à une interaction ; le code correspond à l'implantation du connecteur. Cependant, la différence avec le composant est que le connecteur ne correspond pas à une unique, mais éventuellement à plusieurs unités de programmation.

Six caractéristiques importantes sont à prendre en compte pour spécifier de manière exhaustive un connecteur. Ces caractéristiques sont les suivantes :

- *L'interface* d'un connecteur définit les points d'interactions entre connecteurs et composants. L'interface ne décrit pas des services fonctionnels comme ceux du composant mais s'attache à définir des mécanismes de connexion entre composants. Certains ADLs nomment ces points d'interactions comme étant des rôles.

- *Le type* d'un connecteur correspond à sa définition abstraite qui reprend les mécanismes de communication entre composants ou les mécanismes de décision de coordination et de médiation. Il permet la description d'interactions simples ou complexes de manière générique et offre ainsi des possibilités de réutilisation de protocoles. Par exemple, la spécification d'un connecteur de type RPC qui relie deux composants définit les règles du protocole RPC.

- *La sémantique* des connecteurs est définie par un modèle de haut niveau spécifiant le comportement du connecteur. A l'opposé de la sémantique du composant, qui doit exprimer les fonctionnalités déduites des buts ou des besoins de l'application, la sémantique du connecteur doit spécifier le protocole d'interaction. De plus, le protocole d'interaction doit pouvoir être modélisé et raffiné lors du passage d'un niveau de description abstraite à un niveau d'implantation.

- *Les contraintes* permettent de définir les limites d'utilisation d'un connecteur,

c'est-à-dire les limites d'utilisation du protocole de communication associé. Une contrainte est une propriété devant être vérifiée pour un système ou pour une de ses parties. Si celle-ci est violée, le système est considéré comme un système incohérent. Par exemple, le nombre maximum de composants interconnectés à travers le connecteur correspond à une contrainte.

- *La maintenance* des propriétés (interface, comportement) d'un connecteur doit lui permettre d'évoluer sans perturber son utilisation et son intégration dans les applications existantes. Il s'agit de maximiser la réutilisation par modification ou raffinement des connecteurs existants. Un ADL donnant la possibilité de définir un connecteur doit donc permettre de le faire évoluer de manière simple et indépendante en utilisant le sous typage ou des techniques de raffinement.

- *Les propriétés non fonctionnelles* d'un connecteur concernent tout ce qui ne découle pas directement de la sémantique du connecteur. Elles spécifient des besoins qui viennent s'ajouter à ceux déjà existants et qui favorisent une implantation correcte du connecteur. La spécification de ces propriétés est importante puisqu'elle permet de simuler le comportement à l'exécution, l'analyse, la définition de contraintes et la sélection des connecteurs.

La configuration de l'architecture

La configuration de l'architecture définit les propriétés topologiques de l'application : les connections entre composants et connecteurs, mais aussi, selon les ADL, les propriétés de concurrence, de répartition, de sécurité, etc. La topologie peut être dynamique, auquel cas la configuration décrit la topologie ainsi que son évolution.

Neuf caractéristiques sont précisées pour évaluer la configuration d'un ADL. Ces caractéristiques sont les suivantes :

- *Un formalisme commun.* Une configuration doit permettre de fournir une syntaxe simple et une sémantique permettant de (a) faciliter la communication entre les différents partenaires d'un projet (concepteurs, développeurs, testeurs, architectes), (b) rendre compréhensible la structure d'une application à partir de la configuration sans entrer dans le détail de chaque composant et de chaque connecteur, (c) spécifier la dynamique d'un système, c'est-à-dire l'évolution de celui-ci au cours de son exécution.

- *La composition.* La définition de la configuration d'une application doit permettre la modélisation et la représentation de la composition à différents niveaux de détail. La notion de configuration spécifie une application par composition hiérarchique. Ainsi un composant peut être composé de composants, chaque composant étant spécifié lui-même de la même manière, jusqu'au composant dit primitif, c'est-à-dire non décomposable. L'intérêt de ce concept est qu'il permet la spécification de l'application par une approche descendante par raffinement, allant du niveau le plus général formé par les composants et les connecteurs principaux, définis eux mêmes par des groupes de composants et de connecteurs, jusqu'aux détails de chaque composant et de chaque connecteur primitifs.

- *Le raffinement et la traçabilité.* La configuration est également un moyen de permettre le raffinement de l'application d'un niveau abstrait de description général vers un niveau de description de plus en plus détaillé, et, ceci, à chaque étape du processus de développement (conception, implantation, déploiement). Ainsi il est possible, par la définition de la configuration, de garder une trace de ce qui a été fait en amont, et de créer des liens entre les différents niveaux de description de l'application. Cette caractéristique permet le rapprochement entre les modèles de haut niveau et le code.

- *L'hétérogénéité.* La configuration d'un ADL doit permettre le développement de grands systèmes avec des éléments préexistants de caractéristiques différentes. L'ADL doit être capable de spécifier une application indépendamment du langage de programmation, du système d'exploitation et du langage de modélisation.

- *Le passage à l'échelle.* Les ADLs se doivent de proposer une modélisation de systèmes qui peuvent grossir en taille. Il s'agit de prévoir le nombre d'instances et leur placement dans un environnement ainsi que la dynamique de l'application.

- *L'évolution de la configuration.* La configuration doit être capable d'évoluer pour prendre de nouvelles fonctionnalités impliquant une modification ou une évolution de l'application. Elle doit permettre de faire évoluer l'architecture d'une application de manière incrémentale, c'est-à-dire par ajout ou retrait de nouvelles classes de composants et des connecteurs.

- *L'aspect dynamique de l'application.* La configuration d'une application doit permettre la modification à l'exécution de systèmes demandant des temps d'exécution longs ou pouvant difficilement être stoppés. Elle doit spécifier le comportement dynamique de l'application, c'est-à-dire les changements de l'application qui peuvent

arriver pendant son exécution comme la création, ou la suppression d'instances de composants.

– *Les contraintes*. Les contraintes liées à la configuration viennent en complément des contraintes définies pour chaque composant et chaque connecteur. Elles décrivent les dépendances entre les composants et les connecteurs et concernent des caractéristiques liées à l'assemblage de composants que l'on qualifie de contraintes inter composants. La spécification de ces contraintes permet de définir des contraintes dites globales, s'appliquant à tous les éléments de l'application.

– *Les propriétés non-fonctionnelles*. Certaines propriétés non fonctionnelles ne concernant ni les connecteurs ni les composants doivent être exprimées au niveau de la configuration. Ces contraintes sont liées à l'environnement d'exécution. Un ADL doit donc être capable de spécifier ces contraintes au niveau de la configuration.

L'Adl Darwin

Le langage Darwin, Magee *et al.* (1995); Magee et Kramer (1996), est considéré comme un langage de description d'architecture, bien que celui-ci soit souvent appelé langage de configuration. Un langage de configuration favorise la description de la configuration d'une application, c'est-à-dire la description des interactions entre composants. La particularité de ce langage est qu'un composant est une entité instanciable. La description d'un composant au niveau du langage permet de créer de multiples instances d'un composant lors de l'exécution. Ainsi, ce type de langage se centre sur la description de la configuration et sur l'expression du comportement d'une application plutôt que sur la description structurelle de l'architecture d'un système comme le font de nombreux ADLs. La particularité de Darwin est donc de permettre la spécification d'une partie de la dynamique de l'application en terme de schéma de création de composants logiciels avant, après ou en cours d'exécution.

Le concept principal de Darwin est le composant. Un composant est décrit par une interface qui contient les services fournis et requis. Ces services s'apparentent plus aux entrées et sorties de flots de communication qu'à la notion de fonction. Deux types de composants existent :

– *les primitifs* intègrent du code logiciel, leur granularité est établie au niveau d'un processus,

– *les composites* sont des interconnexions de composants, ils représentent des unités de configuration.

La sémantique associée à un composant est celle du processus. Ainsi, une instance de composant correspond à un processus créé. Il est possible d'associer au composant un ensemble de paramètres typés d'initialisation et d'utiliser les valeurs de ceux-ci à l'intérieur de la configuration du composite de façon à décrire la configuration de manière dynamique. Par exemple, Darwin permet de fixer le nombre d'instances d'un composant lors de son initialisation. Les services requis ou fournis (ang. *require* et *provide*) correspondent à des types d'objets de communication que le composant utilise pour respectivement communiquer avec un autre composant ou recevoir une communication d'un autre composant. Ainsi, les services n'ont pas de connotation fonctionnelle. Ils décrivent les types d'objets de communication utilisés ou autorisés à appeler une fonction du composant. Ces types d'objets sont définis par le support d'exécution réparti appelé Regis, Magee *et al.* (1994) et sont limités. Ainsi, à l'exécution, un composant Darwin est un processus qui communique avec d'autres composants grâce à des objets de communication créés et gérés par le système d'exécution Regis. Parmi les types d'objet, le port est le plus courant : il s'agit d'un objet envoyant des requêtes de manière synchrone ou asynchrone entre composants répartis ou non.

La déclaration d'un composant suit la syntaxe suivante :

```
Component nom (liste de paramètres)
  Provide nomPort <port, signature>
  Require nomPort <port, signature>
```

Les composites sont des entités de configuration. Ils contiennent les descriptions des interconnexions de l'application. Une application est décrite comme un composant composite. Deux constructions syntaxiques permettent de définir des schémas d'instanciation :

- l'opérateur *inst* qui déclare une instance de composant sur un site particulier. Cet opérateur permet de décrire la phase d'initialisation,
- l'opérateur *bind* qui relie un port requis d'un composant à un port fourni d'un autre composant. Cet opérateur permet de décrire les liens entre composants au moment de l'exécution. Cet opérateur peut servir à lier un port d'un composant composite avec un port d'un composant primitif faisant partie du composite.

Darwin permet de décrire un schéma d'instanciation de composants très évolué. Par exemple, il est possible de définir des variables tels que des compteurs. Il existe également des constructions syntaxiques telles que l'itérateur *forall* et l'opérateur de test *when*. Ainsi, il est possible de spécifier le comportement de l'application au niveau comportement global en spécifiant la coopération des instances de composants.

Listing 2.1 – Architecture *pipeline* à base d'un composant composite dans Darwin.

```
component pipeline (int n) {
  provide input;
  require output;
  array F[n]:filter;
  forall k:0..n-1 {
       inst F[k];
       bind F[k].output -- output;
       when k<n-1 bind
            F[k].next -- F[k+1].prev;
      }
      bind
          input -- F[0].prev;
          F[n-1].next -- output;
     }
```

L'exemple de la figure 2.2 illustre un composant de type *pipeline* composé par une liste d'instances de composant *filter*. L'entrée *input* de chaque instance est connectée à la sortie *output* de son prédécesseur. L'itérateur *forall* permet d'instancier chacune des instances (*inst*) et de les connecter (*bind*). Lorsqu'une interface n'est pas satisfaite à l'intérieur du composant, le composant peut l'exposer comme un besoin à satisfaire par l'extérieur, c'est-à-dire par d'autres composants compatibles. Ceci est le cas, par exemple dans *F[n-1].next – output*.

L'Adl Rapide

Rapide, Luckham *et al.* (1995), est un langage de description d'architecture dont le but est de vérifier, par la simulation, la validité d'une architecture logicielle donnée. Il

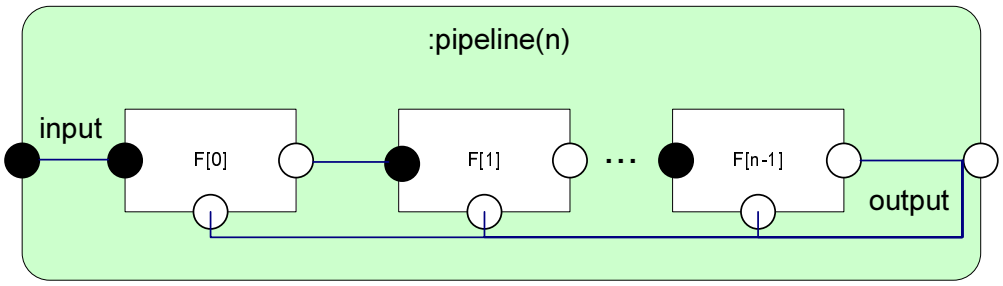


Fig. 2.2 – Architecture *pipeline* à base d'un composant composite dans Darwin.

fut proposé à l'origine au projet ARPA (ang. *Advanced Research Projects Agency*) en 1990 par l'université de Stanford aux États Unis.

Avec le langage Rapide, une application est construite à partir de modules ou de composants communiquant par échange de messages ou d'événements. Rapide fournit également un environnement composé d'un simulateur permettant de vérifier la validité de l'architecture. Les concepts de base du langage Rapide sont les suivants :

- les événements,
- les composants,
- l'architecture.

L'événement est une information transmise. Il permet de construire des expressions appelées *event patterns* qui caractérisent les événements circulant entre composants. La construction de ces expressions se fait avec l'utilisation d'opérateurs qui définissent les dépendances entre événements. Parmi ces opérateurs on trouve l'opérateur de dépendance causale ($A \rightarrow B$ si l'événement B dépend causalement de A), l'opérateur d'indépendance ($A \parallel B$), l'opérateur de différence ($A \sim B$ si A et B sont différents) et l'opérateur de simultanéité ($A \; and \; B$ si A et B sont vérifiés). Ainsi, l'événement correspond à une information permettant de spécifier le comportement d'une application.

Le composant ou le module est défini par une interface. Cette dernière est constituée d'un ensemble de services fournis et d'un ensemble de services requis. Les services sont de trois types :

1. les services *Provides* fournis par le composant appelés de manière synchrone par d'autres composants,
2. les services *Requires* demandés par le composant appelés de manière synchrone,
3. les *Actions* qui correspondent à des appels asynchrones entre composants. Deux types d'actions existent : les actions *in* et *out* qui sont des événements acceptés et envoyés par un composant.

L'interface contient également une section de description du comportement (clause *behavior*) du composant. Cette dernière correspond au fonctionnement observable du composant comme, par exemple, l'ordonnancement des événements ou des appels aux services. Ainsi, l'environnement Rapide peut simuler le fonctionnement de l'application.

De plus, Rapide permet également de spécifier des contraintes (clause *constraint*) qui sont des patrons d'événements qui doivent ou non se produire pour un composant lors de son exécution. Par exemple, une contrainte peut fixer un ordre obligatoire pour une séquence d'événements d'un composant. En général, ces contraintes permettent de spécifier des restrictions sur le comportement des composants.

L'architecture contient la déclaration des instances de composants et les règles de connexions entre ces instances. Toutes les instances sont déclarées sous forme de variables. La règle d'interconnexion est composée de deux parties. La première est la partie gauche qui contient une expression d'événements qui doit être vérifiée, la seconde est la partie droite qui contient également une expression d'événements qui doivent être déclenchés après la vérification de l'expression de la partie de gauche. Les contraintes (clause *constraint*) peuvent être utilisées pour décrire l'architecture. Elles permettent de restreindre le comportement de l'architecture en définissant des patrons d'événements à appliquer pour certaines connexions entre composants. Les parties gauches et droites peuvent être connectées par trois types d'opérateurs :

1. l'opérateur *To* connecte deux expressions d'événements simples. Il ne peut y avoir qu'un événement possible vers un composant. Si la partie gauche est vérifiée alors l'expression de la partie droite permet le déclenchement de l'événement vers l'unique composant désigné par cette expression. Cet opérateur permet de spécifier un appel de type RPC.
2. l'opérateur de diffusion ||> connecte deux expressions quelconques. Dès que la partie gauche est vérifiée, tous les événements contenus dans la partie droite sont déclenchés. Ils sont envoyés vers l'ensemble des destinataires désignés dans cette expression. L'ordre d'évaluation de cette règle de connexion est quelconque. Un déclenchement de cette règle est indépendant d'autres déclenchements antérieurs ou postérieurs.
3. l'opérateur pipeline => est identique au précédent mais l'ordre d'évaluation des règles est contrôlé. Un déclenchement de cette règle est causalement dépendant des déclenchements antérieurs de cette même règle.

   Un exemple d'architecture producteur/consommateur est décrit par la spécification suivante :

Listing 2.2 – Exemple d'architecture producteur/consommateur dans Rapide.

```
type Producer (Max : Positive) is interface
  action out Send (N: Integer);
  action in Reply(N : Integer);
  behavior
      Start => send(0);
     (?X in Integer) Reply(?X) where ?X<Max => Send(?X+1);
end Producer;
type Consumer is interface
  action in Receive(N: Integer);
  action out Ack(N : Integer);
  behavior
     (?X in Integer) Receive(?X) => Ack(?X);
end Consumer
architecture ProdCon() return SomeType is
  Prod : Producer(100); Cons : Consumer;
     connect
     (?n in Integer) Prod.Send(?n) => Cons.Receive(?n);
      Cons.Ack(?n) => Prod.Reply(?n);
end architecture ProdCon;
```

L'Adl Wright

Wright est un langage d'architecture logicielle proposé par Allen (1997); Allen et Garlan (1997) qui se centre sur la spécification de l'architecture et de ses éléments. Il n'y a pas de générateur de code, ni de plate-forme permettant de simuler l'application comme pour Rapide. Il repose sur quatre concepts qui sont le composant, le connecteur, la configuration et le style.

La notion de composant

Un composant en Wright est une unité abstraite localisée et indépendante. La description d'un composant contient deux parties importantes qui sont l'interface (interface) et la partie calcul(computation). L'interface consiste à décrire les ports, c'est-à-dire les interactions auxquelles le composant peut participer. Par exemple, un composant représentant un serveur de base de données peut avoir deux ports, un pour les requêtes du client et un autre pour l'administrateur pour ses propres tâches. Un port peut également être perçu comme une facette d'un composant. A chaque port est associée

une description formelle par le langage CSP spécifiant son comportement par rapport à l'environnement. La partie calcul, quant à elle, consiste à décrire le comportement du composant en indiquant comment celui-ci utilise les ports. Ainsi, les ports qui sont décrits indépendamment dans l'interface, sont utilisés pour décrire le comportement du composant dans le calcul.

La notion de connecteur

Un connecteur représente une interaction entre une collection de composants. Il possède un type. Il spécifie le patron d'une interaction de manière explicite et abstraite. Ce patron peut être réutilisé dans différentes architectures. Par exemple, un protocole de base de données comme le protocole de validation à deux phases (two-phase commit) peut être un connecteur. Il contient deux parties importantes qui sont un ensemble de rôles et la glue. Chaque rôle indique comment se comporte un composant qui participe à l'interaction. Le comportement du rôle est décrit par une spécification CSP. La glue décrit comment les participants (c'est-à-dire les rôles) interagissent entre eux pour former une interaction. Par exemple, la glue d'un connecteur appel de procédure indiquera que l'appelant doit initialiser l'appel et que l'appelé doit envoyer une réponse en retour.

La notion de configuration

La configuration permet de décrire l'architecture d'un système en regroupant des instances de composants et des instances de connecteurs. La description d'une configuration est composée de trois parties qui sont la déclaration des composants et des connecteurs utilisés dans l'architecture, la déclaration des instances de composants et de connecteurs, les descriptions des liens entre les instances de composants par les connecteurs. Wright supporte la composition hiérarchique. Ainsi, un composant peut être composé d'un ensemble de composants. Il en va de même pour un connecteur. Lorsqu'un composant représente un sous-ensemble de l'architecture, ce sous-ensemble est décrit sous forme de configuration dans la partie calcul du composant.

La notion de style

Le style d'une architecture permet de décrire un ensemble de propriétés communes à une famille de systèmes comme, par exemple, les systèmes temps-réel ou les systèmes de gestion de paie. Il permet de décrire un vocabulaire commun en définissant un ensemble de types de connecteurs et de composants et un ensemble de propriétés et de contraintes partagées par toutes les configurations appartenant à ce style. Ainsi, Wright permet de définir des types de connecteurs et de composants pour une famille d'architectures à la condition que ceux-ci respectent les propriétés de cette famille. Les propriétés

et les contraintes communes à une architecture peuvent être définies selon trois caractéristiques qui sont les types d'interfaces, les paramètres et les contraintes. Les types d'interface permettent de typer le rôle d'un connecteur ou le port d'un composant pour un système donné. Les paramètres comprennent les informations de style pour définir des composants ou des connecteurs avec des parties de leurs descriptions qui peuvent être en paramètre comme par exemple la partie calcul. Finalement, les contraintes sont des prédicats logiques de premier ordre qui doivent être satisfaits pour tous les éléments appartenant au style. Chaque partie d'un élément de l'architecture sous Wright (glue du connecteur, rôles du connecteur, calcul ou ports d'un composant, configuration de l'architecture du système, interface de type) a une spécification décrivant le comportement de celle-ci. Le modèle utilisé pour la spécification est le modèle CSP. Ainsi, à chaque comportement, est associé un *process* CSP. Le *process* est un patron de comportement formé d'événements observables et déclenchés par ce *process (events)*. Les événements déclenchés par le *process* possèdent une barre au dessus, les événements observables n'en possèdent pas ; de plus, les événements peuvent fournir (x !e) ou recevoir des données (x ?e). Il est possible de définir d'autres *process* tels qu'une séquence de **process** ou un ensemble de *process* exclusifs.

L'exemple suivant correspond a celui d'une architecture client/serveur présenté dans Allen *et al.* (1997). Il s'agit d'un client et un serveur interagissant via un connecteur *link* (Figure 2.3). La description d'un tel système doit considérer sa structure, en décrivant sa topologie et sa composition, ainsi que les aspects comportementaux du système en tant qu'unité globale.

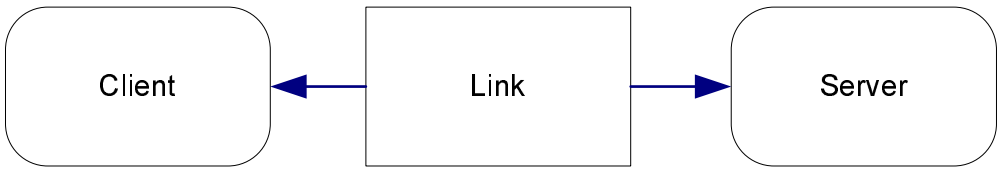


Fig. 2.3 – Architecture Client/Serveur dans Wright

Chaque description de composant fournit une spécification de haut niveau vis-à-vis des fonctionnalités et des interfaces. Tandis que la spécification du connecteur indique comment le pattern *link* combine les comportements d'un client et un serveur. Dans ce cas, un client fait une requête, laquelle est reçue par le serveur ; ensuite le serveur fournit une réponse, laquelle est communiquée au client. Cette séquence d'actions peut se répéter plusieurs fois.

La spécification Wright présentée ci-dessous, montre les patterns d'interactions et en plus fournit des détails importants sur les règles d'interactions. L'idée de base est de

traiter les composants et le connecteurs en tant que processus qui se synchronisent.

Component Client

Port p $= \overline{request} \rightarrow reply \rightarrow p \sqcap \S$

Computation $= internalCompute \rightarrow \overline{p.request} \rightarrow p.reply \rightarrow Computation \sqcap \S$

Component Server

Port p $= request \rightarrow \overline{reply} \rightarrow p \Box \S$

Computation $= p.request \rightarrow internalCompute \rightarrow \overline{p.reply} \rightarrow Computation \Box \S$

Connector Link

Role c $= \overline{request} \rightarrow reply \rightarrow p \sqcap \S$

Role s $= request \rightarrow \overline{reply} \rightarrow p \Box \S$

Glue $= c : request \rightarrow \overline{s : request} \rightarrow Glue$

$\Box s : reply \rightarrow \overline{c : reply} \rightarrow Glue$

$\Box \S$

Configuration Client-Server

Instances

C :Client ; L :Link ; S :Server

Attachments

C.p as L.c ; S.p as L.s

End Configuration

L'utilisation du choix interne ($\sqcap$) dans la spécification du client indique que c'est le client qui décide sur la génération de requêtes. L'utilisation du choix externe ($\Box$) dans

la spécification du serveur indique qu'on attend que le serveur réponde à un nombre déterminé de requêtes, et qu'il ne doit pas s'arrêter avant que ceci passe. Une terminaison correcte est indiquée par § (le processus vide).

L'Adl xADL

xADL, Dashofy *et al.* (2001), est basé sur la définition de XML Schemas W3C (2004d,e) pour la description de « familles d'architectures ». Techniquement, xADL correspond à une application d'une spécification plus abstraite et générique connue comme xArch, Dashofy et van der Hoek (2001). Dans xADL, chaque type de connecteur, de composant et d'interface est relié à une partie correspondant à son implémentation. Cette partie est remplacée par les variables définies dans le modèle de programmation et de plateforme cible.

2.2.2 Style d'architecture

Un style architectural est un moyen générique qui aide à l'expression des solutions structurelles des systèmes. Il comprend un vocabulaire d'éléments conceptuels (les composants et les connecteurs), impose des règles de configuration entre les éléments du vocabulaire (ensemble de contraintes) et véhicule une sémantique qui donne un sens (non ambiguë) à la description structurelle, Shaw *et al.* (1995); Shaw et Garlan (1996).

Chaque style véhicule des propriétés logicielles spécifiques adaptées à des critères retenus pour un système. Ainsi, dans une organisation client-serveur :

- Un serveur représente un processus qui fournit des services à d'autres processus appelés clients ;
- Le serveur ne connaît pas à l'avance l'identité et le nombre de clients ;
- Le client connaît (ou peut trouver via un autre serveur) l'existence du serveur ;
- Le client accède au serveur, par exemple via des appels de procédures distants.

Le style d'une architecture permet de décrire un ensemble de propriétés communes à une famille de systèmes comme, par exemple, les systèmes temps-réel ou les applications orientées service. Il permet de décrire un vocabulaire commun en définissant un ensemble de types de connecteurs et de composants et un ensemble de propriétés et de contraintes partagées par toutes les configurations appartenant à ce style. La conformité d'un système à un style apporte plusieurs avantages, notamment pour : l'analyse,

la réutilisation, la génération du code et l'évolution du système, Garlan *et al.* (1994); Shaw et Garlan (1996); Taylor *et al.* (1996).

Le style d'architecture C2

Un exemple concret de style d'architecture est celui défini par C2, Taylor *et al.* (1996). Ce style a été défini pour représenter les logiciels distribués. Une architecture suivant le style C2 présente les caractéristiques suivantes :

- les connecteurs permettent l'échange de messages entre composants,
- les composants ont un état associé, réalisent des opérations et échangent des messages avec d'autres composants via deux interfaces (appelées « top » et « bottom »),
- chaque interface considère un ensemble de messages qui peuvent être envoyés ou reçus. L'interface d'un composant ne peut être reliée qu'à un seul connecteur,
- un connecteur peut être relié à n'importe quel nombre de composants et de connecteurs,
- deux types de messages sont permis : *request* (appel à un composant pour réaliser une opération) et *notification* (pour informer des opérations réalisées ou des changements d'état),
- un message *request* peut seulement être dirigé vers le haut de l'architecture, alors qu'un message *notification* peut seulement être dirigé vers le bas.

La figure 2.4 illustre un exemple d'une architecture C2 pour un système de gestion de réunions. Les composants impliqués sont : (i) *MeetingInitiator* qui s'occupe des apertures de réunion, (ii) *Attendee* qui représente chacun des participants à une réunion préétablie, et (iii) *ImportantAttendee* qui représente un type spécial de participant. Trois connecteurs sont déclarés, chacun correspondant à un des composants. Certains messages sont envoyés depuis le *MeetingInitiator* vers les différents types de participants. D'autres messages sont adressés uniquement aux participants spéciaux.

C'est le composant *MeetingInitiator* qui entame les opérations en demandant les besoins de réunion aux composants *Attendees* et *ImportantAttendees*. Les composants concernés notifient au composant *MeetingInitiator* leurs besoins spécifiques, alors que celui-ci cherche à programmer des réunions en fonctions de leurs exigences. Dans la figure 2.4, une bonne partie de l'information requise par cet exemple est implicite, ainsi un ADL s'avère nécessaire pour préciser les détails.

2.3 Les architectures dynamiques

Les architectures dynamiques correspondent à des applications dont les composants sont créés, interconnectés, et supprimés pendant l'exécution. Cette dynamique répond à des contraintes liées à l'adaptabilité d'une application distribuée et à la mobilité de ses utilisateurs. Le caractère dynamique des architectures implique des difficultés supplémentaires pour la description. Pour les architectures statiques, cette description est réalisée via la déclaration des instances des composants, et de liens d'interconnexions. Cette approche est inutilisable puisque la structure même de l'architecture change.

Les modifications des architectures logicielles de manière générale peuvent se produire soit au moment de la conception, soit au moment de la pré-exécution ou encore au moment de l'exécution. Les architectures dynamiques changent de structures pendant l'exécution du système et donnent une description à propos de ce changement. Depuis quelques années, la dynamique des architectures est devenue une activité qui commence à prendre de l'ampleur puisqu'elle donne la possibilité aux administrateurs d'agir sur l'architecture de l'application.

La dynamique de l'architecture consiste donc à adapter une architecture pour prendre en compte de nouveaux besoins. Ceci permet de faire passer une architecture d'un configuration C à une configuration C'. Une opération peut être initiée soit par un administrateur, soit par l'application elle-même en réponse à des événements de surveillance.

Ces architectures dynamiques sont particulièrement utilisées dans les applications dites « sensibles au contexte » c'est-à-dire les applications qui doivent évoluer suite à des changements de besoins de l'utilisateur.

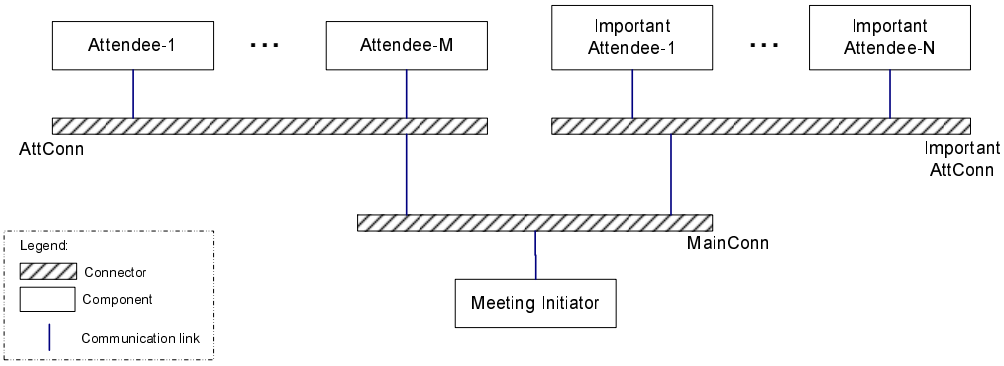


FIG. 2.4 – Style d'architecture C2 pour un système de gestion de réunions, Taylor *et al.* (1996).

2.3.1 Architectures dynamiques selon les ADL

Nombre d'ADL proposés se sont limité à une description statique des architectures logicielles. D'autres ont allés plus loin, en abordant la problématique des architectures dynamiques. Parmi les plus représentatifs on peut citer : Darwin et Wright.

Darwin

Darwin, Magee *et al.* (1995); Magee et Kramer (1996), offre deux approches pour décrire l'aspect dynamique des architectures. La première approche appelée instanciation paresseuse (ang. *lazy instantiation*) permet de retarder l'instanciation de certains composants. Ainsi, un composant offrant un service et déclaré dynamiquement en utilisant cette approche, ne sera instancié que lorsqu'un utilisateur de ce service tente d'y accéder.

La deuxième approche permettant de décrire l'évolution dynamique de l'architecture concerne ce qui est appelé dans la sémantique Darwin l'instanciation dynamique directe (ang. direct dynamic instantiation). Cette approche permet la définition de structures qui peuvent évoluer par réplication de composants. Chaque événement introduit une nouvelle instance d'un composant spécifié. Les connexions des composants créés selon cette approche sont définies de manière générique pour chaque composant répliqué.

Les deux approches définies par le langage Darwin pour la description des architectures dynamiques correspondent au traitement de plusieurs cas classiques de reconfiguration. Elles présentent, néanmoins, plusieurs limitations. Ainsi, la première approche nécessite d'énumérer et de déclarer tous les composants potentiels de l'architecture. Elle est inadéquate si, par exemple, l'architecture possède un nombre non borné de composants. Cette approche n'adresse réellement l'évolution dynamique de l'architecture que du point de vue de l'activation des composants.

La deuxième approche adresse l'évolution dynamique par la création de nouvelles instances de composants, mais le fait que les instances créées par réplication soient anonymes et soient, donc, toutes connectées de la même manière constitue une limitation très contraignante. De plus, comme le langage Darwin interdit de connecter plusieurs services offerts à un seul service requis, le raccordement des services offerts par les composants créés par réplication ne peut être décrit par Darwin. Un autre aspect non considéré par Darwin est la spécification des suppressions de composants.

Wright

L'ADL Wright considère une version qui traite la dynamique des architectures, cette version appelée dynamic Wright, Allen *et al.* (1998), a introduit des événements de contrôle permettant de spécifier les conditions sous lesquelles les transformations de l'architecture sont autorisées. Ces événements de contrôle induisent l'exécution d'une séquence d'actions élémentaires de reconfiguration comprenant l'introduction et la suppression de composants et de connecteurs (respectivement les actions *new* et *del*) et l'introduction et la suppression des liens (respectivement les actions *attach* et *detach*). Les programmes de reconfiguration (appelés *Configuror*) spécifient les politiques qui caractérisent l'évolution dynamique de l'architecture globale.

Le formalisme de description pour les architectures dynamiques défini par l'extension dynamic Wright est plus expressif que celui spécifié par Darwin. Cependant, sa puissance d'expression reste limitée par le fait qu'il implique l'énumération de toutes les configurations possibles de l'architecture. Dans la mesure où la taille des architectures dynamiques (en nombre de composants et de connecteurs) est généralement non bornée et que, par conséquent, le nombre de configurations possible est infini, cette limite réduit le champ d'application de ce formalisme aux applications de petite taille avec une faible composante dynamique.

2.3.2 UML en tant que langage pour la description des architectures

UML[2] Rumbaugh *et al.* (2005); OMG (2005) est considéré comme un successeur des langages de modélisation trouvés dans les méthodologies : *Booch* Booch (1993), *Object-Oriented Software Engineering* (OOSE) Jacobson (1992) et *Object Modeling Technique* (OMT) Rumbaugh (1997). UML est un autre mécanisme descriptif qui peut être employé pour modéliser des architectures logicielles. UML est un langage qui se base sur une représentation graphique pour modéliser un système. Il aide ses utilisateurs à produire différents diagrammes qui forment ensemble une modélisation complète du système. UML n'impose aucune méthodologie de conception, c'est-à-dire, UML n'impose pas une manière particulière pour l'utilisation des diagrammes qu'il offre, excepté l'utilisation des règles de syntaxe définies dans sa spécification.

A l'heure actuelle UML comporte 13 diagrammes. 6 diagrammes décrivent la struc-

[2]UML a évolué en plusieurs versions depuis sa création, donc dans ce manuscrit on s'adresse à la version 2.0 faute de déclarer explicitement une autre version

ture du système (figure 2.5(a)) : diagramme de classes, diagramme de structure composite, diagramme de composants, diagramme de déploiement, diagramme d'objets et diagramme de paquetage. Les 7 autres autres diagrammes permettent de décrire le comportement du système (figure 2.5(b)) : les diagrammes d'interactions (diagramme de séquence, diagramme de vue d'ensemble des interactions, diagramme de communication et diagramme de temps), diagramme d'activité, diagramme de machine à états et diagramme de cas d'utilisation.

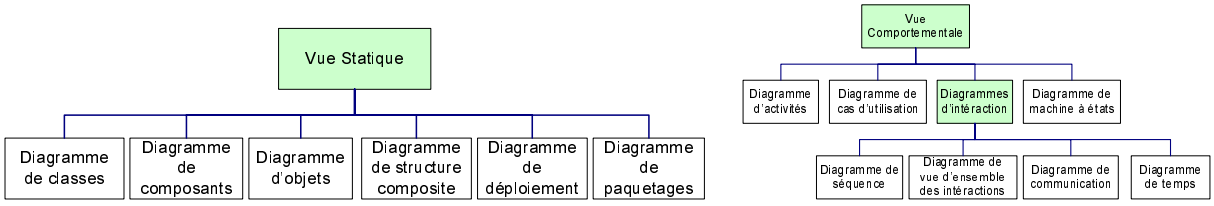
La diversité des ADLs proposés a conduit à plusieurs efforts d'analyse d'UML en tant que langage de description des architectures, Abi-Antoun et Medvidovic (1999); Egyed et Medvidovic (2001); Garlan *et al.* (2002); Gomaa et Wijesekera (2001); Hofmeister *et al.* (1999); Rausch (2001). Certains le considèrent en soit incapable de représenter les éléments des architectures logicielles, Schewe (2000), alors que d'autres considèrent la syntaxe et la sémantique d'UML comme insuffisante, mais en préconisant une extension adéquate on peut l'utiliser en tant qu'ADL. Parmi les principales exigences que doit accomplir UML pour l'adapter comme un ADL on peut citer notamment :

1. UML doit être adéquat pour modéliser les exigences structurelles d'un système, c'est-à-dire les aspects liés à la configuration ou à la topologie,

2. il doit être capable de modéliser les différents aspects comportementaux issus des divers formalismes retrouvés dans le domaine des ADLs, tels que pi-calculus, logique de premier ordre, ensembles d'événements à ordre partiel, ...,

3. il doit être capable de modéliser une large variété de paradigmes d'interaction entre composants, par exemple en utilisant la notion de connecteur,

4. la sémantique d'un connecteur en UML est différente de celle qu'on trouve dans les ADLs, donc une adaptation s'avère nécessaire. Par exemple, en UML un connecteur relie deux composants dont un offre un service que l'autre à sollicité. Tandis que dans les ADLs, souvent un connecteur peut être relié à un nombre quelconque de connecteurs et de composants.



(a) Aspect statique (b) Aspect comportementale

FIG. 2.5 – Classification de diagrammes UML.

UML et le style architectural C2

Dans les travaux qui visent une adaptation d'UML, Medvidovic *et al.* (2002) proposent une modélisation visant à représenter le style d'architecture C2. Dans un premier temps, ils proposent l'utilisation d'UML 1.4 tel qu'il est, c'est-à-dire sans considérer ses mécanismes d'extension. Pour illustrer leurs idées ils utilisent l'application de gestion de réunions (présentée dans la section 2.2.2). D'abord, ils représentent la structure du système en utilisant un diagramme de classes comme illustré dans la figure 2.6.

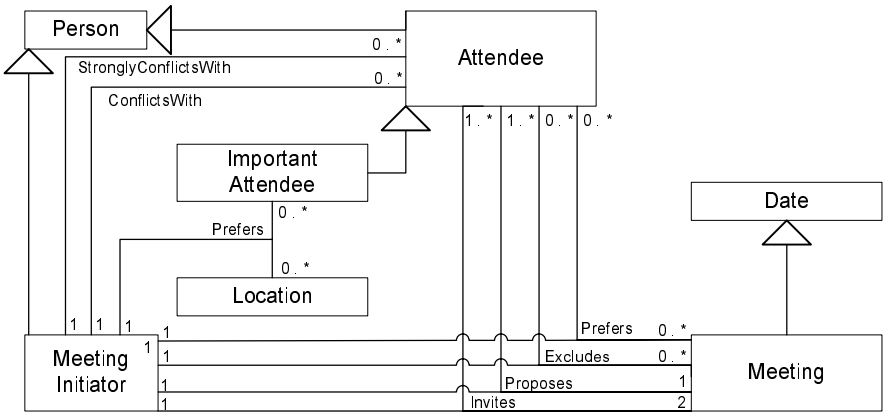
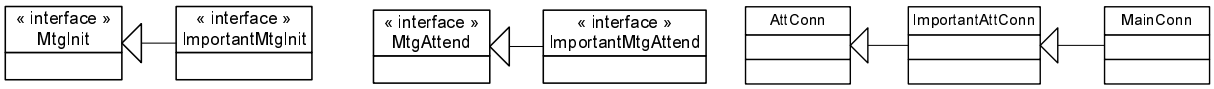


FIG. 2.6 – Modélisation d'une application de gestion de réunions par un diagramme de classe UML.

Le diagramme de classe illustre les classes issues de l'application, leurs relations d'héritage et leurs associations. Les interfaces de messages sont des éléments importants dans le style C2. Celles-ci sont décrites au travers des stéréotypes UML comme illustré dans la figure 2.7(a).



(a) Interfaces (b) Connecteurs

FIG. 2.7 – Modélisation UML des interfaces et des connecteurs d'une application de gestion de réunions.

Dans la définition du style C2 la définition de connecteurs s'avère nécessaire. Bien que ceux-ci jouent un rôle différent par rapport aux composants, ils sont aussi modélisés par le biais de classes UML (voir figure 2.7(b)). Chaque connecteur est considéré comme une classe simple qui (possiblement filtre et) renvoie les messages qu'elle reçoit vers les composants indiqués.

Le diagramme de classe de la figure 2.8 adapté au style C2 montre les relations entre classes et interfaces. Les traits avec le rond représentent la réalisation des interfaces par les classes, alors que les traits en pointillés avec des flèches orientées vers les ronds représentent des relations de dépendance d'une classe avec l'interface impliquée.

Des diagrammes de collaboration sont utilisés pour la représentation des interactions entre les instances de classes. Par exemple, la figure 2.9 illustre une collaboration entre une instance de la classe *MeetingInitiator (MI)* et les instances des classes *Attendee* et *ImportantAttendee*. De manière plus précise, *MI* lance une requête en proposant un

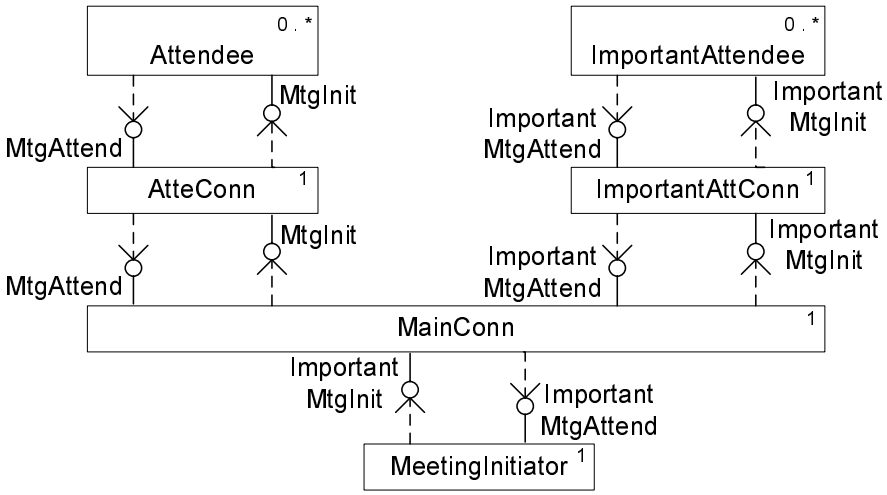


Fig. 2.8 – Utilisation des diagrammes de classes UML pour représenter une application de gestion de réunions selon le style C2.

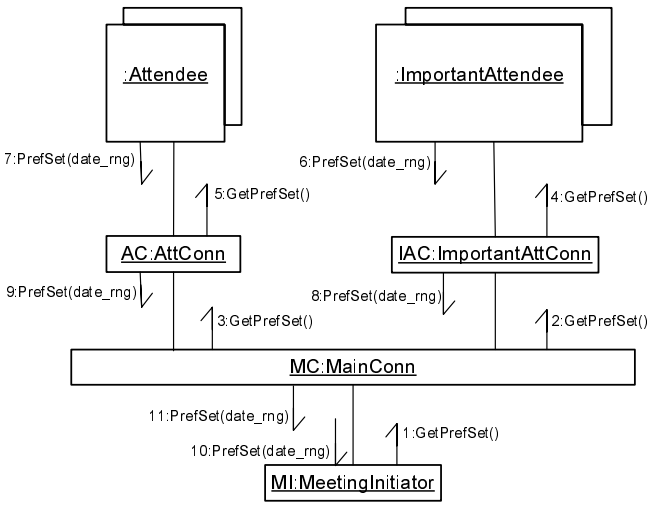


Fig. 2.9 – Utilisation des diagrammes de collaborations UML pour représenter une application de gestion de réunions selon le style C2.

ensemble de dates de réunion possibles ; l'instance de la classe *MainConn (MC)* renvoi la requête vers les instances des connecteurs *AC* et *IAC*, lesquels à leur tour, renvoient la requête vers les composants associés *Attendee* et *ImportantAttendee*. Chaque composant participant choisit une date et notifie celle-ci à *MI* via la diffusion de messages en passant par les connecteurs impliqués.

En conclusion, UML en tant que langage pour les architectures logicielles ne satisfait pas complètement les besoins structuraux nécessaires dans la description des architectures. C'est le cas, par exemple, des connecteurs que l'on doit spécifier en UML de la même façon que les composants, bien qu'il s'agit d'entités différentes. Un autre point concerne l'absence de mécanismes pour établir des règles pour la définition du style architectural.

Une façon de palier les manques d'UML est l'utilisation de ses mécanismes d'extension. Ceci consiste à adapter le meta-modèle UML afin de rajouter des nouveaux éléments nécessaires à la description des architectures. Pour cela la définition des stéréotypes s'avère nécessaire. Cette stratégie est employée, comme une deuxième option, dans l'adaptation d'UML pour représenter le style C2, Robbins *et al.* (1998).

UML et le style architectural C3

Un autre travail qui traite l'adaptation d'UML pour la description des architectures logicielles est celui de Pérez-Martínez et Sierra-Alonso (2004); Pérez-Martínez (2003). Leur travail représente un exemple des extensions lourdes (heavyweight) d'UML. Pour illustrer leur approche, ils proposent un style architectural appelé C3 lequel est basé sur le style C2. Concrètement le style C3 modifie le style C2 de la manière suivante :

- à la différence de C2, C3 ne prédétermine pas le type d'héritage des composants. Il laisse aux composants ce choix,
- les opérations des interfaces permettent seulement des paramètres d'entrée,
- le type de composants et sa structure interne ne sont pas prédéterminés,
- les connecteurs admettent uniquement des politiques de filtrage et de consommation de messages,
- préconditions et postconditions peuvent êtres définies comme des opérations des interfaces.

Pour étendre le métamodèle UML, C3 suit les consignes suivantes :

- aucun metaconstructeur (métaclasse du métamodèle UML) n'est supprimé ni modifié dans sa syntaxe ou sémantique,

– les nouveaux metaconstructeurs doivent avoir un minimum de relations avec les metaconstructeurs existants, c'est-à-dire ils doivent être auto-contenus.

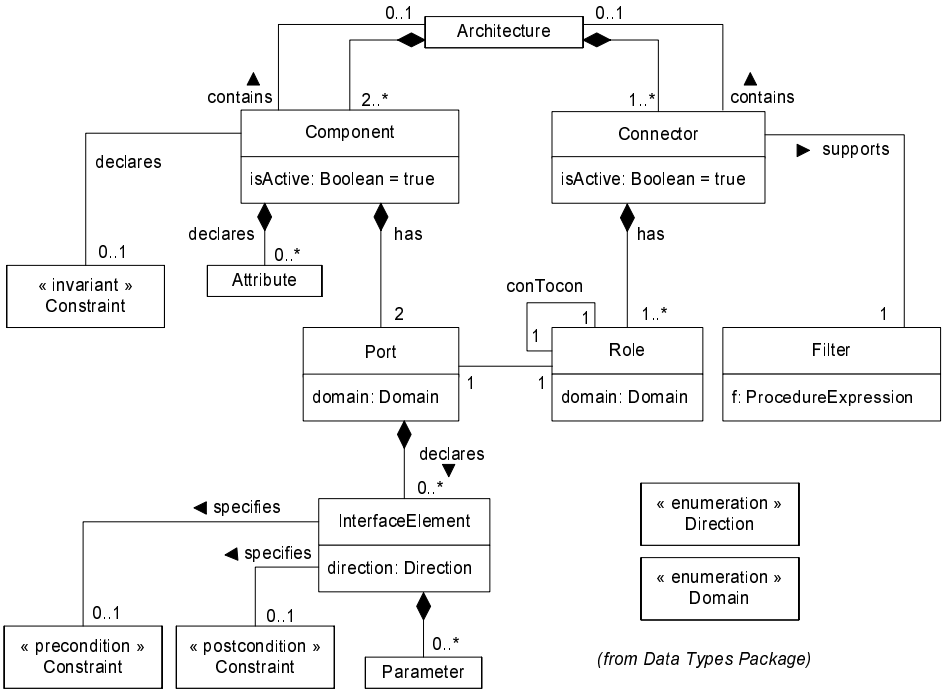


FIG. 2.10 – Syntaxe abstraite caractérisant le paquet C3.

La figure 2.10 illustre le métamodèle proposé pour la description de la syntaxe abstraite des architectures. Les nouveaux constructeurs sont rajoutés au métamodèle comme sousclasses de la métaclasse *ModelElement*, laquelle est d'ailleurs sousclasse de *Element* (la métaclasse racine). Les constructeurs *Constraint*, *Attribute* et *Parameter* définis dans le paquet *Core*, et les types *Boolean* et *ProcedureExpression* définis dans le paquet *Data Types* sont utilisés.

L'architecture

– cet élément représente un conteneur de blocs de construction qui définissent une architecture C3,

– dans le métamodèle une architecture est formée par deux composants ou plus et un connecteur ou plus.

Les composants

– sont représentés par deux ports, un avec la valeur *top* et un autre avec la valeur *bottom*,

– sont des éléments actifs dans le sens qu'ils ont leur propre flux de contrôle,

– ont un état décrit par le constructeur *Attribute*,

– peuvent déclarer un invariant, lequel est soutenu par la métaclasse *Constrait*.

Les connecteurs

– ont un ou plusieurs points d'interactions caractérisés par le constructeur *Role*,

– ils sont aussi des éléments actifs,

– ils implémentent des politiques de filtrage.

L'interface d'élément

– ce constructeur représente une opération impliquant les interactions d'un composant avec son environnement,

– elle dispose d'un nom et d'une direction (en tant qu'attributs), qui correspondent à des types énumérés ayant pour valeurs *prov* (fournisseur) et *req* (demandeur),

– une pre et/ou post condition peut lui être associée.

Le filtre

– représente la politique de filtrage associée à un connecteur,

– comme attributs il dispose d'un nom et d'une expression (f) représentant la politique de filtrage.

Le port

– représente un point d'interaction entre le composant et son environnement,

– il dispose d'un domaine représentant le *top* ou le *bottom* de son composant,

– il peut être connecté au rôle d'un connecteur.

Le rôle

– représente un point d'interaction entre un connecteur et son environnement,

– il dispose d'un domaine représentant le *top* ou le *bottom* de son connecteur,

– il peut être connecté au port d'un composant ou à un autre rôle avec un autre connecteur.

2.4 Architectures orientées service

Les architectures orientées service (SOA) sont des modèles qui définissent un système par un ensemble de services logiciels distribués, qui fonctionnent indépendamment les uns des autres afin de réaliser une fonctionnalité globale. Le choix d'une architecture SOA entre dans la perspective de transformer le Web en une énorme plate-forme de composants faiblement couplés et automatiquement intégrables. L'une des technologies la plus utilisée pour implanter ce type d'architectures est les services Web. La particularité de cette nouvelle technologie réside dans le fait qu'elle utilise la technologie Internet comme infrastructure pour la communication entre les composants logiciels et

prend le pari d'employer des standards généralistes, fortement répandus et peu coûteux tels que XML ou HTTP.

SOA est une approche architecturale permettant la création des systèmes basés sur une collection de services développés dans différents langages de programmation, hébergés sur différentes plates-formes avec divers modèles de sécurité et processus métier. Chaque service représente une unité autonome de traitements et de gestion de données, communiquant avec son environnement à l'aide de messages. Les échanges de messages sont organisés sous forme de contrats d'échange.

Comparée à la programmation orientée objet qui promeut la réutilisation au niveau micro, par le biais de classes et d'objets, l'architecture orientée service promeut la réutilisation de composants logiciels au niveau macro, par le biais de services.

L'idée de base de l'architecture orientée service est que tout élément du système d'information doit devenir un service :

- Clairement identifiable : identifier un service signifie être en mesure d'en connaître l'existence, indépendamment de sa localisation, de son mode de fonctionnement, de son mode d'appel, etc.
- Réalisant un ensemble de tâches parfaitement définies : il s'agit de comprendre le fonctionnement global du service, c'est-à-dire de connaître la liste des tâches qu'il est capable de traiter, ses conditions opérationnelles d'exécution, les exceptions qu'il peut provoquer, etc.
- Documenté : La documentation doit décrire clairement comment faire appel au service. Elle doit spécifier nettement (et si possible de manière unifiée, voire standardisée) les méthodes d'appels, les fonctions proposées et la signature du service.
- Autonome et doté d'un niveau de sécurité contrôlé.
- Fiable dans un contexte donné : La fiabilité d'un service dépend naturellement de son contexte d'utilisation. Ce contexte doit être clairement identifié et les conditions de fonctionnement du service doivent être explicitement décrites.
- Indépendant vis-à-vis d'autres services (même si pouvant faire appel à certains d'entre eux).
- Accessible sur le réseau.

2.4.1 Structure de SOA

SOA est axée autour de trois concepts fondamentaux :

1. *Le fournisseur de services* : désigne un serveur ou un système permettant l'accès aux services via un réseau tel que Internet. Cet accès s'opère via une interface de services, c'est à dire une application permettant à d'autres applications d'y accéder.
2. *Le consommateur de services* : désigne une personne, un serveur ou un système en réseau qui accède aux services et les utilise grâce à son interaction avec l'annuaire, afin de trouver un service répondant à un besoin précis. Une fois ce service localisé, le demandeur contacte ou se connecte à l'interface de services du fournisseur afin d'utiliser le service de son choix.
3. *Le registre de services* : appelé aussi annuaire de services, il représente l'entité logicielle qui joue le rôle d'intermédiaire entre les clients et les fournisseurs de services. Le concept « registre » ou « dépôt de service » est essentiel dans l'architecture orientée service. Il joue un rôle central dans le processus de localisation des besoins et dans l'interopérabilité car il est supposé fournir aux clients les informations techniques sur le fonctionnement du service et ceci dans des langages interprétables par les machines.

2.4.2 Services Web

L'une des tendances historiques qui a conduit à l'apparition des services Web est l'utilisation de l'architecture par composants comme approche d'intégration des applications, Pfister et Szyperski (1996). Les composants sont des entités logicielles fondées sur une interface et une sémantique bien définie. Ils interagissent moyennant une infrastructure qui permet de gérer la communication entre des composants au sein d'un même système ou à travers un réseau via une décomposition de la logique applicative en composants distribués, Brown (1996); Szypersky (1999). L'architecture de composants distribués a engendré un développement rapide et évolutif d'applications distribuées et complexes. Au cours de ce développement, on a assisté à la mise en place de trois architectures par composants : CCM (CORBA Component Model) OMG (2002), EJB (Enterprise Java Beans) MicroSystems (2003) et COM (Commun Object Model) Rogerson (1997). La mise en œuvre de ces trois architectures soulève des difficultés dans le cadre d'une infrastructure ouverte telle que Internet.

En effet, ces architectures, bien qu'utilisant un modèle objet distribué, proposent chacune sa propre infrastructure. Ce qui impose une forte liaison entre les services offerts par les composants et leurs clients. Ainsi on ne peut assembler que des objets CORBA (ou COM) entre eux. Le résultat est que les systèmes construits à base de ces architectures sont monolithiques.

Parallèlement après l'avènement du B2C (Business To Consumer), où les entreprises mettent en ligne leurs services pour leurs consommateurs au travers des applications Web, celles-ci souhaitent augmenter leur productivité à l'aide du paradigme B2B (Business To Business). Le B2B repose sur l'échange de produits, d'informations et de services entre entreprises. Ceci implique l'utilisation de services et la collaboration avec des systèmes proposés par d'autres concepteurs et par conséquent une maîtrise de l'hétérogénéité. L'interoperabilité est ainsi devenue une nécessité pour l'entreprise dans le monde du B2B. C'est justement ce que les services Web apportent par rapport aux solutions dites monolithiques.

D'une certaine façon, le modèle des services Web est une évolution du modèle des composants distribués rendu nécessaire par l'utilisation intensive de l'Internet. A l'instar des technologies précédentes, les services Web proposent une architecture par composants qui permet à une application de faire l'usage d'une fonctionnalité située dans une autre application. Cependant la solution des services Web repose sur l'ubiquité de l'infrastructure d'Internet alors que les autres architectures reposent chacune sur leur propre infrastructure. L'interoperabilité est donc une caractéristique intrinsèque aux services Web. Définir les services Web nécessite alors d'introduire à la fois leur architecture et leur infrastructure.

Lorsqu'on parle de Web Services, on parle aussi d'architecture orientée services. Le terme « services Web » est souvent utilisé de nos jours bien qu'il n'ait pas toujours le même sens. La définition existante s'étend du très générique vers le spécifique et le restrictif. Souvent, un service Web est vu comme une application accessible pour d'autres applications à travers le Web. C'est une définition tellement large dans ce que l'on peut dire que n'importe quel objet ayant un URL est un service Web. Par exemple, un programme accessible sur le Web avec une API.

Une définition plus précise est proposée par le consortium UDDI qui caractérise les services Web :

« Un service Web est une application métier modulaire qui possède des interfaces orientées Internet basées sur des standards ».

Cette définition est plus détaillée et elle nécessite que le service soit ouvert, pour qu'il soit invoqué à travers Internet. Malgré cette classification, la définition reste imprécise. Par exemple, on ne comprend pas bien le sens d'une application métier modulaire.

Une autre définition est proposée par le World Wide Web Consortium (W3C)

« Un service Web est un système logiciel identifié par un identificateur uniforme de ressources (URI), dont les interfaces publiques et les liens (binding) sont définis et décrits en XML. Sa définition peut être découverte dynamiquement par d'autres systèmes logiciels. Ces autres systèmes peuvent ensuite interagir avec le service Web d'une façon décrite par sa définition, en utilisant des messages XML transportés par des protocoles Internet ».

La définition du W3C est très précise mais ne spécifie pas la manière avec laquelle les services Web doivent fonctionner. La définition exige que les services Web soient « définis, décrits et découverts » ce qui clarifie le sens du mot en rendant plus concret la notion de « orienté Internet, interfaces basées sur des standards ». Elle précise que les services Web doivent être des services similaires à ceux des « middlewares » conventionnels. En plus, les services Web doivent être décrits et publiés pour qu'il soit possible d'écrire ou de définir des clients interagissant avec eux. En d'autres mots, les services Web sont des composants qui peuvent être intégrés dans des applications distribuées plus complexes. Le W3C a fixé que le XML est une partie de la solution. En effet, XML est très populaire et très utilisé aujourd'hui tout comme le HTTP et les serveurs Web. Il est considéré comme une partie de la technologie du Web.

Les services Web ne sont pas attachés à une plate-forme spécifique comme le JVM (machine virtuelle de Java) ou à une infrastructure de technologie comme CORBA parce qu'ils se concentrent sur les protocoles employés pour échanger des messages (SOAP et WSDL) et non pas l'exécution qui soutient ces protocoles. En d'autres termes, vous pouvez établir des services Web sur n'importe quelle plate-forme, en utilisant n'importe quel langage de programmation.

SOAP et WSDL sont basés sur XML dont existe déjà nombre d'analyseurs. Les analyseurs de XML sont disponibles pour la majorité des langages de programmation modernes (Java, C, C++, C#, VB, Perl, Python, etc.). Ainsi, l'infrastructure pour traiter des messages SOAP et des documents WSDL existe déjà. De même, les messages de services Web sont normalement échangés via les protocoles de TCP/IP, lesquels sont soutenus par la plupart des langages de programmation modernes et des plateformes logicielles disponibles aujourd'hui. Grâce aux constructions de services Web sur une infrastructure existante de XML et de TCP/IP, l'adoption des services Web a été rapide et répandue.

Architecture de base

Les services Web possèdent trois acteurs principaux :

1. le fournisseur de service (service provider) : Il définit le service à publier et la description du service dans l'annuaire,

2. l'annuaire (service broker) : Il reçoit et enregistre les descriptions des services publiés par les fournisseurs, d'autre part il reçoit et répond aux recherches de services lancées par les clients,

3. le demandeur (service requestor) : obtient la description du service grâce à l'annuaire utilisé par le service Services Web et invoque les services demandés.

Le scénario classique d'utilisation d'un service Web est le suivant :

1. Une première étape consiste à l'enregistrement du fournisseur du service Web auprès de l'annuaire en passant le fichier WSDL précisant la description du service Web.

2. Dans une autre étape, un client découvre le service offert par le fournisseur en consultant le fichier WSDL concerné par l'intermédiaire de l'annuaire.

3. La étape suivante consiste en l'invocation à distance du service Web a partir du fournisseur.

4. La dernière étape consiste en l'interaction entre le demandeur et le fournisseur du service Web.

L'interaction entre ces axes est représentée dans la figure 2.11.

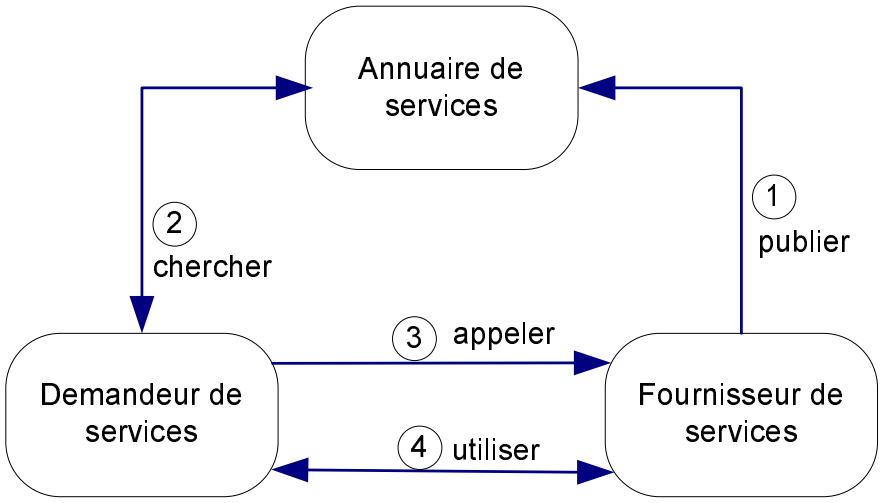


FIG. 2.11 – Architecture orientée service.

2.5 Conclusion

Bien que les ADL représentent des outils efficaces pour la description et l'analyse des architectures logicielles, son adoption reste un peu à l'écart, surtout dans les milieux industriels et d'entreprise. Ceci est dû au niveau de compétences exigées pour la compréhension des concepts et des notations. De même, la diversité des ADL proposés ayant des objectifs différents implique un choix d'adoption encore plus difficile.

Dans la même idée, l'adoption d'UML pour la description des systèmes est de plus en plus courante dans le développement du logiciel d'aujourd'hui, bien que souvent ses vues doivent être étendues pour obtenir le niveau de description requise par l'architecte.

Une solution intégrant les forteresses des deux approches s'avère intéressante, en utilisant UML en tant que mécanisme de spécification des architectures, puis en traduisant ses diagrammes (de manière automatique) vers une approche formelle, telle que les techniques de réécriture de graphes ou d'autres du même type.

Chapitre 3

Adaptabilité des architectures logicielles

3.1 Introduction

Au début des années 2000 IBM annonce une nouvelle initiative appelée « Autonomic Computing », Kephart et Chess (2003), envisageant le développement de systèmes capables de s'auto-gérer. Cette idée a été inspirée du fonctionnement du corps humain qui est pourvu d'une telle capacité. Afin d'atteindre ses objectifs, l'initiative « Autonomic Computing » est décomposée en quatre sous-axes :

1. Le *self-configuring* considère la reconfiguration de composants et de systèmes en définissant des politiques de haut-niveau.

2. Le *self-healing* s'intéresse à l'auto-détection, le diagnostic et la réparation des problèmes au niveau logiciel et matériel.

3. Le *self-optimizing* implique l'auto-réglage de paramètres au niveau service.

4. Le *self-protecting* considère la protection des systèmes contre des attaques et des malveillances.

Les objectifs de notre travail sont fortement partagés avec les deux premières sous-disciplines. Donc, dans ce chapitre nous introduisons les travaux qui proposent des approches diverses visant à apporter des contributions au défi soulevé par ces disciplines émergeantes.

3.2 Reconfiguration des architectures

3.2.1 Une approche comportementale de modélisation SOA

Baresi *et al.* (2003) proposent un modèle basé sur UML pour définir un style SOA. Ils utilisent les graphes et la transformation de graphes pour traduire et analyser les modèles. Afin d'illustrer ces idées, un exemple de « chaîne de production » est présenté. Le style architectural est décomposé en deux parties : un modèle statique et un modèle dynamique. Le modèle statique correspond à la configuration initiale caractérisée par les composants et les connecteurs. Le modèle dynamique fournit la représentation des actions de reconfiguration de l'application comme réponse aux changements imprévus de l'environnement. Des propriétés d'accessibilité et de consistance sont prouvées à l'aide d'un outil de « model checking », Dill *et al.* (1992).

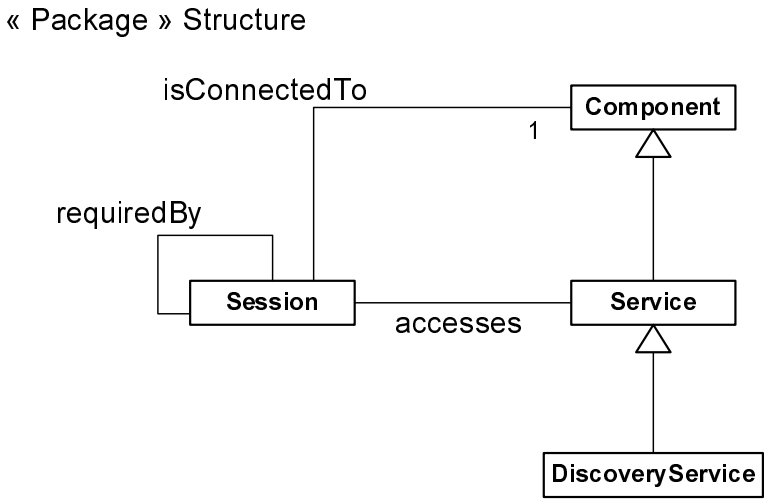


FIG. 3.1 – Module pour la partie structurelle du modèle statique, Baresi *et al.* (2003).

La partie statique du modèle, dans cette approche, est basée sur les diagrammes de classe UML. Cette modélisation considère trois types d'éléments structurés par trois « modules » :

1. le module *Structure* (figure 3.1) contient les classes principales qui définissent le style architectural.
2. le module *Specification* (figure 3.2) contient les classes représentant les documents de spécification nécessaires au modèle statique. Deux types de documents sont considérés : *Requirements* (exigences) et *ServiceSpecification* (spécification du service).

3. le module *Messages* (figure 3.3) fournit les classes nécessaires pour la communication entre composants et services.

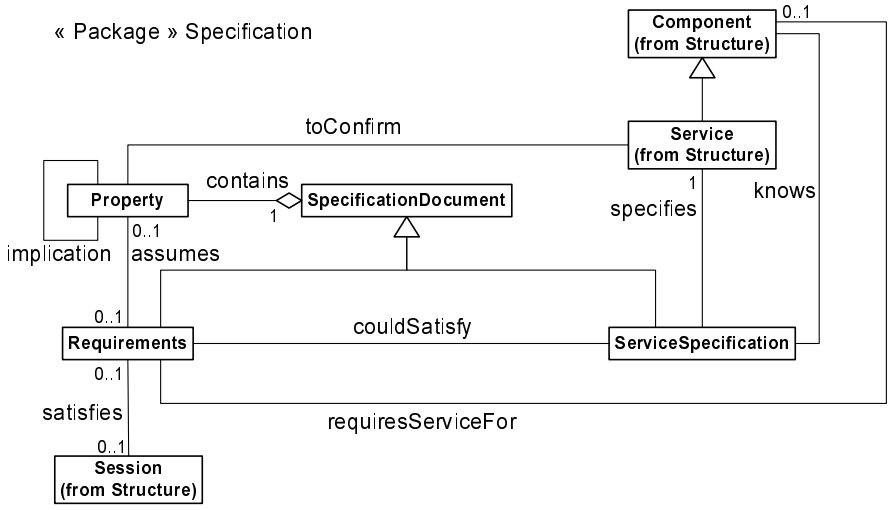


FIG. 3.2 – Module pour la partie spécification de documents du modèle statique, Baresi *et al.* (2003).

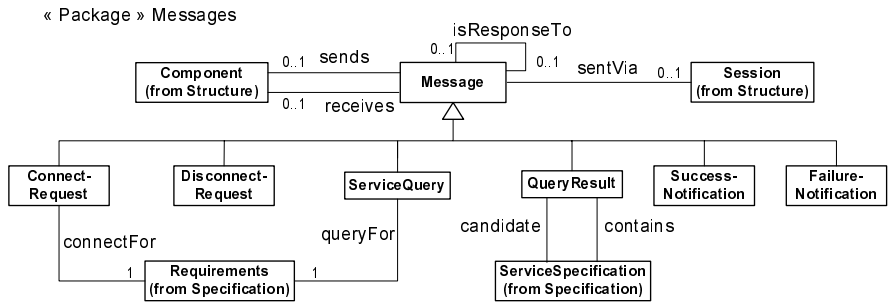


FIG. 3.3 – Module pour la partie messages du modèle statique, Baresi *et al.* (2003).

La partie dynamique du modèle considère la reconfiguration des architectures. Pour ceci, une approche basée sur règles de transformation de graphes est utilisée. Par exemple, dans la figure 3.4 une règle pour la connexion à une session est présentée. Après la réception d'un message de notification, concernant l'acceptation à la session, cette règle peut être déclenchée et un lien est établi entre le composant *serviceRequestor* et la nouvelle session.

Cette approche ne distingue pas les aspects fonctionnels des aspects non-fonctionnels définissant le niveau architectural. En effet, dans leur modèle on trouve invariablement des aspects liés aux intergiciels (middleware) ainsi que des aspects associés à la logique métier des applications. De même, cette approche considère plutôt des aspects comportementaux par opposition aux aspects architecturaux issus de notre travail.

3.2.2 Approches basées sur les graphes

Nous retrouvons dans la littérature différents travaux basés sur les graphes qui traitent des architectures logicielles. Nous pouvons citer principalement les travaux de Métayer (1998) qui constituent probablement les premiers travaux de description basée sur les graphes. Le modèle décrit par Métayer (1998) est structuré en deux niveaux, le premier décrit l'architecture sous la forme d'un graphe, et le second décrit les styles architecturaux par une grammaire de graphes. L'évolution de l'architecture est décrite par des règles de transformation de graphes. Le modèle définit une approche formelle basée sur un algorithme de vérification permettant de vérifier à priori et de manière statique la consistance des règles de l'évolution de l'architecture. Cette approche permet de prouver que les contraintes considérées par la description de l'architecture sont préservées par ces règles.

Dans Hirsch *et al.* (1999), les auteurs reprennent le même modèle pour la description de l'architecture et de son évolution dynamique. Ils décomposent la description de l'architecture en trois parties :

1. la première partie spécifie les règles de la construction de la configuration initiale,
2. la deuxième partie spécifie les règles régissant l'évolution dynamique de l'architecture, et
3. la troisième partie spécifie les règles régissant la communication.

Ces travaux abordent aussi la problématique de la consistance du point de vue de la communication et des états de composants. Le cycle de vie des composants (comprenant par exemple leur activation et leur désactivation) est simulé en affectant des labels (correspondants à l'état courant des composants) aux nœuds des graphes et des règles de réécriture. La partie décrivant la communication est spécifiée via l'introduction d'événements (en notation CSP, Hoare (1985)) et leur prise en compte en étiquetant les arcs des graphes représentant les connexions entre composants.

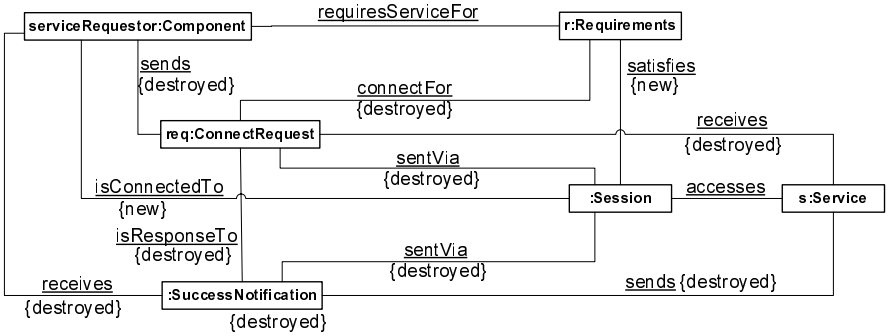


FIG. 3.4 – Règle de connexion à une session, Baresi *et al.* (2003).

3.3 Modélisation de systèmes « self-healing »

Dans cette section nous présentons les travaux liés au domaine du « self-healing ». On s'intéresse aux travaux liés à la modélisation des architectures de « self-healing » en général et plus particulièrement aux architectures de « self-healing » pour les services Web.

3.3.1 Un modèle d'adaptation pour les architectures

Dans cette approche Garlan et Schmerl (2002); Schmerl et Garlan (2002); Garlan *et al.* (2001) proposent un mécanisme basé sur une boucle fermée d'éléments pour l'adaptation des systèmes à base de composants (figure 3.5). Le comportement du système en exécution est surveillé par des composants externes. Ces composants déterminent quand le comportement du système rentre dans la plage de valeurs acceptables, et dans le cas contraire ils appliquent les mécanismes d'adaptation.

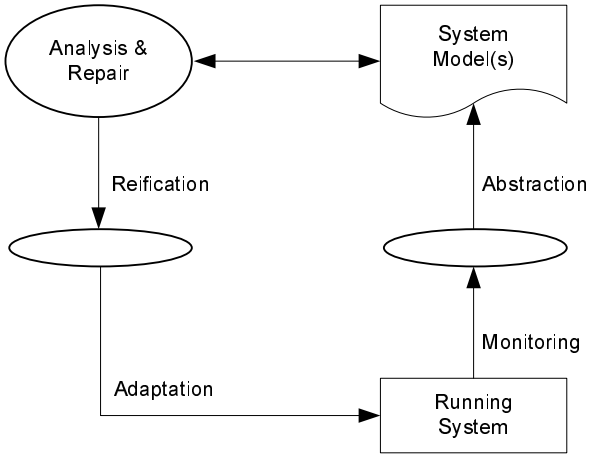


FIG. 3.5 – Modèle abstrait pour l'adaptation des architectures, Garlan et Schmerl (2002).

Plus en détail, cette approche (figure 3.6 propose une infrastructure composée par les éléments suivants :

1. Le système en exécution (Executing System). Des capteurs appelés « probes » (P) sont utilisés pour recueillir les informations pertinentes liées au fonctionnement du système,
2. les éléments de monitoring du système (Monitoring). Des adaptateurs appelés

« gauges » (G) sont proposés pour le passage entre les événements de bas niveau issus du fonctionnement du système et les propriétés architecturales,

3. le modèle architectural de l'application (Arch. Model),
4. les éléments d'analyse et évaluation du système (Analyzer),
5. les éléments de réparation à l'issue de fautes du système (Repair Handler), et
6. l'application des changements à l'architecture comme résultat de l'application des actions de réparation (Translator).

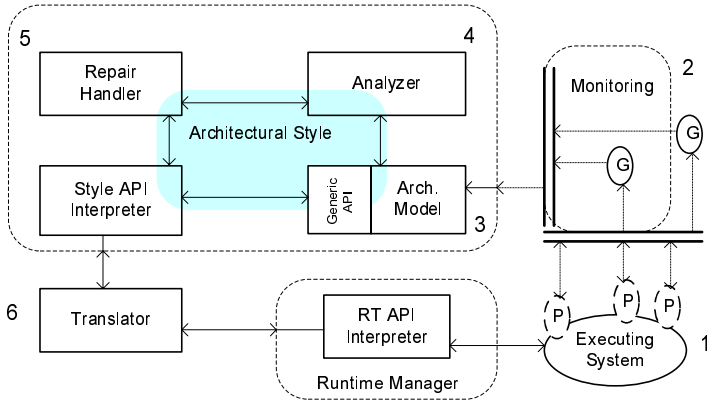


FIG. 3.6 – Infrastructure pour l'adaptation des systèmes, Garlan et Schmerl (2002).

La modélisation de l'architecture du système est faite en utilisant l'ADL ACME, Garlan *et al.* (2000). Ensuite, les propriétés à surveiller sont rajoutées en adaptant le style architectural. Finalement, les éléments de réparation sont aussi rajoutés au style architectural.

3.3.2 Une approche architecturale pour le logiciel auto-adaptatif

Oreizy *et al.* (1999, 1998) proposent une stratégie globale pour le développement de systèmes auto-adaptatifs. Cette stratégie, illustrée par la figure 3.7 ; est composée par deux parties :

1. la gestion de l'adaptabilité (*Adaptation management*) décrit le cycle de vie des logiciels adaptatifs. Le cycle de vie peut être sujet aux interventions humaines ou être complètement autonome. Différentes types d'évaluation et de surveillance (*Evaluate and monitor observations*) peuvent être mis en place durant l'exécution du système, notamment : la performance, la sûreté de fonctionnement, la vérification de contraintes,... Le plan d'adaptation (*Plan changes*) correspond à la définition d'un mécanisme d'adaptation en rapport avec les résultats de l'évaluation et la

surveillance du système. Le déploiement de changements (*Deploy change description*) implique l'application coordonnée des changements du système. Les changements du système impliquent aussi bien le remplacement d'un composant isolé que toute une reconfiguration avec des composants distribués et interdépendants.

2. la gestion de l'évolution (Evolution management) correspond aux changements effectués afin de reconfigurer et adapter l'application. Ce mécanisme suit une approche architecturale, c'est-à-dire les changements sont explicitement établis sur un modèle architectural (*Architectural model*) déployé sur une plate-forme d'implémentation (*Implementation*). Les modifications faites sur le modèle architectural sont reproduites sur l'application en respectant la consistance (*Maintain consistency*) entre le modèle et l'implémentation.

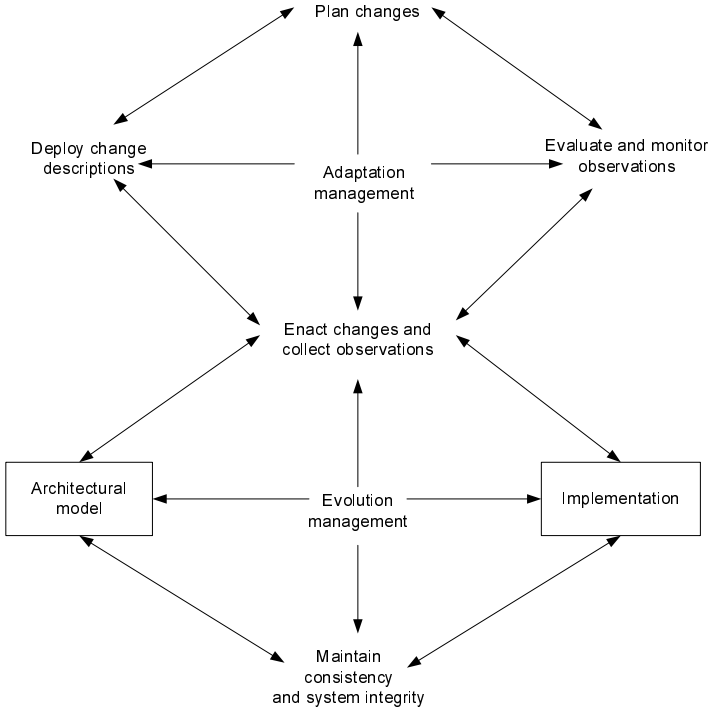


FIG. 3.7 – Une approche générique pour le logiciel auto-adaptatif, Oreizy *et al.* (1999).

3.3.3 Une approche pour le développement de composants « self-healing »

Shin (2005) propose une architecture à deux niveaux pour le développement de composants dites « self-healing ». Un composant « self-healing » en plus d'implémenter la logique métier d'une application, il est capable de détecter et de réparer situations de dysfonctionnement de façon autonome. Chaque composant est défini par deux couches (figure 3.8), la couche de service est la couche de « healing ».

La couche de service fournit la fonctionnalité nécessaire entre composants et notifie à la couche « healing » le statut de messages échangés entre composants. La communication entre couches services de différents composants est faite par le biais de connecteurs. Le mécanisme de notification de messages de la couche service est utilisé par la couche « healing », afin de surveiller et de détecter situations de dysfonctionnement des objets de la couche service. Dans le cas de situation de dysfonctionnement, le composant passe d'un état normal à un état de réparation. Dans cet état les fonctionnalités de la couche service doivent s'arrêter partiellement ou complètement.

La couche « healing » de chaque composant est responsable de la détection, reconfiguration et réparation d'objets de la couche service. Ceci est fait par le composants suivants :

- Le composant *Component Monitor* surveille la trace d'exécution de chaque tâche, connecteur et objets.
- Le composant *Component Reconfiguration Plan Generator* maintient l'information de configuration des objets de la couche de service. De même, il génère plans de reconfiguration dans les cas nécessaires.
- Le composant *Component Repair Plan Generator* maintien et génère plans de réparation pour les objets en état de dysfonctionnement.
- Le composant *Component Reconfiguration Executor* applique le plan de reconfiguration généré par le composant *Component Reconfiguration Plan Generator* afin de réorganiser les objets en état anormal.
- Le composant *Component Repair Executor* applique le plan de réparation généré par le composant *Component Repair Plan Generator* et vérifie que les objets réparés reviennent à l'état normal.
- Le composant *Component Self-Healing Controller* agit comme le coordonnateur général d'autres composants de la couche « healing ».

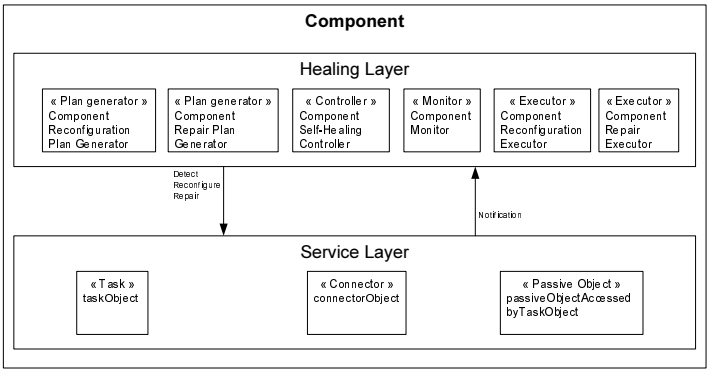


FIG. 3.8 – Architecture des composants « self-healing », Shin (2005).

3.3.4 Une approche architecturale pour la création de systèmes « self-healing »

Selon Dashofy *et al.* (2002) la capacité de réparer dynamiquement un système en temps d'exécution en se basant sur son architecture demande un certain nombre d'exigences :

1. La capacité de *décrire l'architecture du système*. La description architecturale est faite en utilisant les outils liés à l'ADL xADL 2.0.
2. La capacité d'*appliquer des changements à l'architecture* issus d'un plan de réparation. Une réparation architecturale est faite par le biais d'une stratégie appelé *diff*. Une *diff* est un document qui décrit les différences entre deux architectures logicielles spécifiées par xADL 2.0. Dans le contexte des systèmes « self-healing », la *diff* d'une architecture décrit ses différences avant et après l'application des actions de réparation. En complément de l'outil *diff* un autre outil nommé *ArchMerge* permet de fusionner deux architectures dans une architecture cible.
3. La capacité d'*analyser et valider* les résultats obtenus après l'application des actions de réparation. Ceci est fait par le biais de composant appelés *design critics*. Ces composants surveillent l'application afin de trouver possibles problèmes dans son architecture, liés à l'application des actions de réparation.
4. La capacité d'*exécuter un plan de réparation* sans réinitialiser le système. Une fois le plan de réparation choisi, celui-ci est appliqué en s'appuyant sur un outil appelé *c2.fw* . Cet outil est une librairie Java qui permet la mise en œuvre des architectures évolutives à base d'événements. En concret, lorsque le composant *AEM* (Architecture Manager Evolution) reçoit une notification de changement, celui-ci détermine si la structure de l'architecture a subi un changement, dans le cas positive il fait les appels nécessaires à l'outil *c2.fw* pour appliquer les changements au système en exécution.

Les composants est documents liés à cette stratégie, ainsi que leurs relations, sont illustrés dans la figure 3.9.

3.3.5 Une architecture pour les services Web autonomes

Birman *et al.* (2004) proposent une extension à l'architecture classique de services Web Weerawarana *et al.* (2005) afin de permettre la conception et développement de services Web autonomes. Le but est de garantir la sûreté de fonctionnement dans les systèmes critiques à base de services Web. L'extension proposée est montrée dans la figure 3.10. La définition de chaque composant de l'architecture est comme suit :

– Le composant *CHM* surveille le fonctionnement de composants de l'application de façon individuelle et rapporte des changements sensibles d'être importants pour d'autres composants. Cette information pourrait être utilisée pour détecter des éventuelles fautes et puis déclencher les mécanismes de récupération nécessaires.

– Le composant *CRM* offre mécanismes de communication fiable et synchronisation dans des groupes de processus.

– Le composant *DDS* offre mécanismes de multicast fiable pour la duplication de données et la diffusion d'information entre services Web et leurs clientes.

– Le composant *MDC* établie mécanismes de surveillance et gestion globale de l'application, par opposition à la surveillance individuelle faite par le composant *CHM*.

– Le composant *EVN* joue un rôle analogue au DDS, sauf que le premier est spécialisé sur la notification en urgence d'événements.

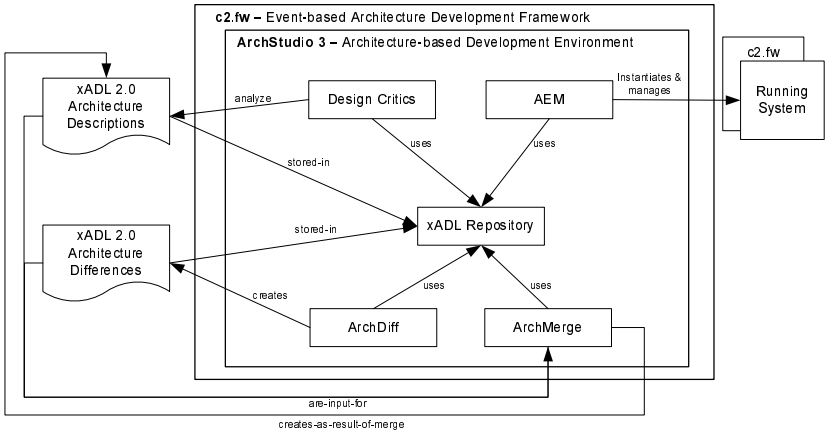


FIG. 3.9 – Outils et documents pour le développement de systèmes « self-healing », Dashofy *et al.* (2002).

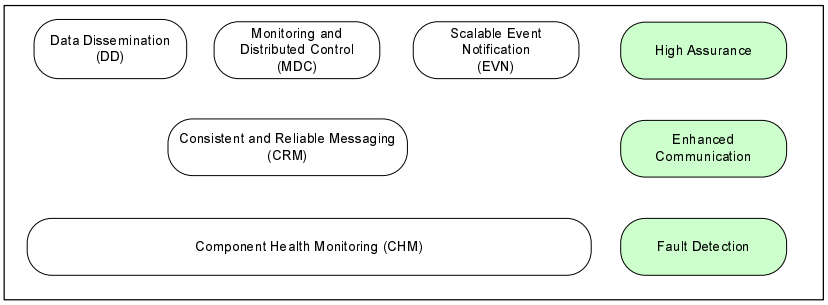


FIG. 3.10 – Composants pour les services Web autonomes, Birman *et al.* (2004).

3.3.6 Une infrastructure d'exécution pour les systèmes « self-healing »

Wile et Egyed (2004); Wile (2002) proposent une infrastructure pour surveiller, interpréter, analyser et reconfigurer les systèmes en exécution (figure 3.11). Une couche d'éléments appelés *Probes*, qui rapportent données pertinentes, sont installés avant de démarrer le système. D'autres éléments appelés *Gauges* traduisent et interprètent ces données, par rapport au modèle architectural du système, afin de repérer certains événements cibles. Les données issues des *Gauges* sont transmissent à travers le bus de *Gauges* où d'autres *Gauges* peuvent réagir ou prendre décisions de contrôle. Ensuite, les éléments *Effectors* sont appelés afin d'effectuer des changements sur le système.

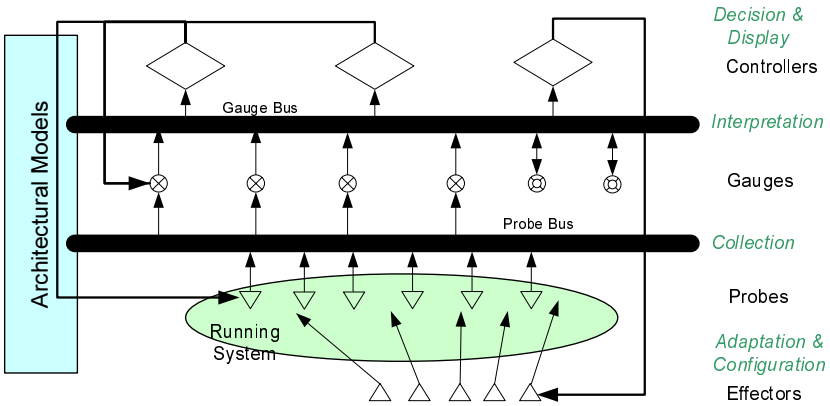


FIG. 3.11 – Architecture de l'infrastructure pour l'exécution de systèmes « self-healing », Wile et Egyed (2004).

3.3.7 Une plate-forme pour le « self-healing » dans les services Web

Gurguis et Zeid (2005) considèrent une implémentation basée sur une stratégie de « self-healing » appelée MAPE Kephart et Chess (2003) proposée par IBM. La stratégie MAPE est définie par une boucle de contrôle composée de 4 phases intégrées :

1. Dans la première phase, le composant *Monitor* collecte des données des éléments contrôlés.
2. Dans la deuxième étape, le composant *Analyzer* évalue les événements rapportés afin d'identifier la situation courante de l'élément contrôlé. Ensuite, dans le cas de dysfonctionnement il propose des actions de réparation.
3. Dans la troisième étape, le composant *Planner* spécifie les actions adéquates à prendre pour sortir d'un état de dysfonctionnement.

4. Et dans la dernière phase, le composant *Executive* exécute les actions de réparation sur les éléments contrôlés.

La proposition d'implémentation de la stratégie MAPE est illustrée par la figure 3.12. Pendant l'exécution d'un service Web, des événement correspondant au fonctionnement du service Web sont stockés et classés dans de registres *CBE* (Common Base Event). Un composant de diagnostic *Diagnosis Engine* analyse les données issues des services Web afin de reconnaître patterns présents dans les registres et avec ceci identifier possibles fautes. Pour en faire, ce composant s'appui sur une base de données de symptômes et actions de réparation *Symptoms Database*. Une fois la faute et les actions de réparation repérées, un composant *Rule Engine* détermine la faisabilité d'application des actions de réparation en accord avec les politiques du système. Ceci est fait en se basant sur une base de données de politiques (Database Policies).

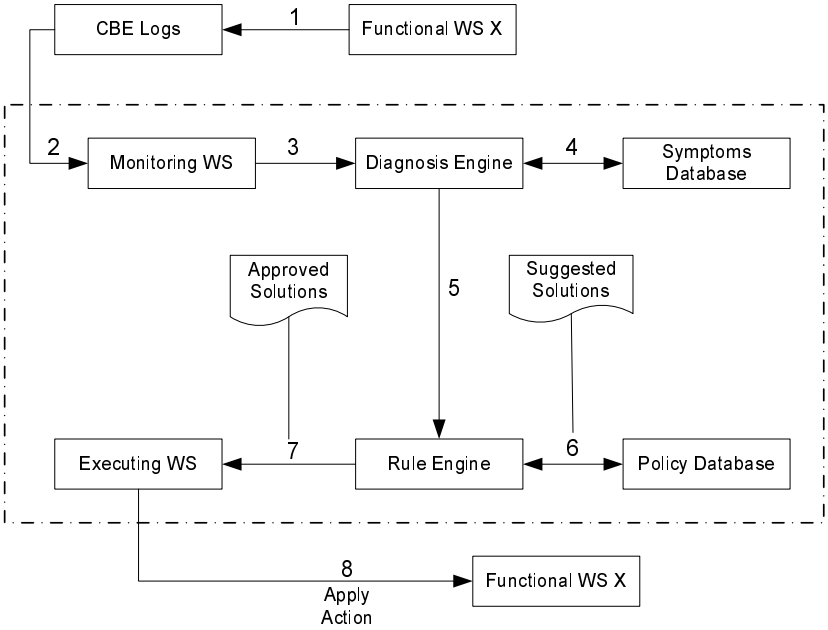


FIG. 3.12 – Implémentation de la stratégie MAPE pour le « self-healing » de services Web, Gurguis et Zeid (2005).

3.4 Conclusion

Nous avons présenté dans ce chapitre différentes approches pour l'adaptabilité des architectures. Nous reprendrons certaines idées de ces travaux pour nous inspirer et nous positionner dans le développement de nos contributions, présentées dans les prochains chapitres. En particulière, nous adoptons les éléments d'adaptabilité des architectures

pour sa description et sa reconfiguration. De même, les éléments de « self-healing » sont pris en compte, afin de proposer des mécanismes de gestion de la QdS pour les applications à base de services Web.

Deuxième partie

Contributions

Chapitre 4

Cadre logiciel pour l'adaptation des architectures

4.1 Introduction

Dans ce chapitre nous proposons une approche guidée par le modèle pour le déploiement et la reconfiguration des architectures à base de services Web. Cette approche considère un partitionnement explicite des aspects fonctionnels et non fonctionnels afin de modéliser une application définie par des services Web coopératifs. Dans cette perspective, notre première démarche consiste à représenter les aspects fonctionnels des architectures. On cherche, en première instance, à définir une spécification structurelle considérant la décomposition fonctionnelle d'une application de services Web. Comme deuxième instance, on considère la gestion des actions de réparation, au niveau architectural, au travers de la définition et de l'application de règles de transformation.

Cette proposition vise d'une part les aspects statiques tels que la spécification des architectures, et d'autre part, des aspects dynamiques par la définition de règles de transformation et leur application à la construction des actions de reconfiguration au niveau architectural. La figure 4.1 illustre les éléments composant notre stratégie. La vue conceptuelle considère les aspects liés au modèle, alors que la vue réalisation considère les aspects techniques qu'implémentent chacune des parties des aspects conceptuels. Les sections développées dans ce chapitre traitent chacun des aspects considérés par les deux vues.

4.2 Positionnement par rapport aux approches de reconfiguration

Quand on s'intéresse aux services Web « self-healing » on peut considérer deux approches pour la reconfiguration. Une première approche se focalise sur le comportement du système, en général ; et de manière plus particulière au comportement interne de ses services Web. Ainsi, lorsqu'un problème est diagnostiqué, l'infrastructure de reconfiguration mise en place est amenée à réparer le « workflow » en cours d'exécution, en appliquant les actions de réparation sur les services Web concernés. Ce type de reconfiguration peut être considéré comme une réparation fine, puisque ses actions sont censées modifier la composition interne du service Web. On peut parler aussi d'une reconfiguration directe puisque à chaque problème il y a une réaction immédiate en vue de corriger le système. Dans les dernières années un grand nombre de travaux s'intéressent à cette approche, par exemple : Modafferi *et al.* (2006); Modafferi et Conforti (2006).

La deuxième approche se focalise sur l'architecture de l'application. Ainsi, lors de l'exécution du système ses composants (services Web) sont observés. Plusieurs symptômes sont stockés avant de décider l'activation d'une action de reconfiguration. Ainsi, il n'y a pas une réaction immédiate correspondante à chaque symptôme. Ici, la réparation est faite par la reconfiguration de l'architecture, c'est-à-dire, en appliquant des actions de base telles que l'activation ou la désactivation de services et leurs connexions. Ce

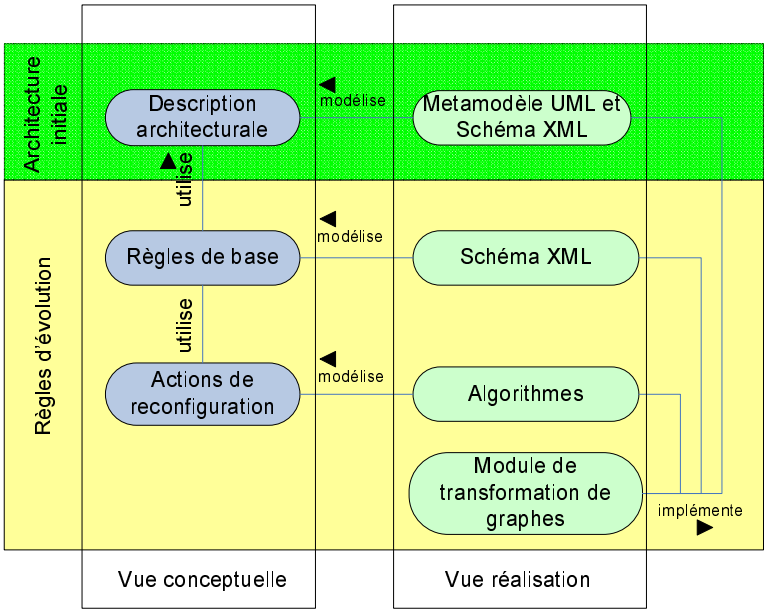


FIG. 4.1 – Stratégie générale pour le cadre logiciel.

type de reconfiguration peut être considéré comme une réparation à gros grain. Ses actions sont censées modifier l'architecture de l'application. Dans ce manuscrit nous développons une stratégie basée sur cette approche.

Le tableau 4.1 résume les différences entre ces deux approches de reconfiguration.

	Surveillance	**Diagnostic**	**Actions de Reconfiguration**
Approche			
Comportementale	Au niveau du workflow.	Une alarme entraîne éventuellement une réparation.	Appliquées à l'intérieur du WS, avec éventuellement modification du code.
Architecturale	Au niveau de l'architecture du système.	Une reconfiguration est considérée après plusieurs symptômes.	Appliquées à l'architecture, donc activation ou désactivation de WS et leurs connexions.

TAB. 4.1 – Approches de reconfiguration dans les services Web « self-healing ».

4.3 Description architecturale

Notre première démarche consiste à définir les architectures caractérisant les applications à base de services Web. Pour ceci, nous introduisons différents types visant à définir la structure des applications. Ainsi, pour nous, une application à base de service Web est caractérisée par un ensemble de *Rôles coopératifs*. Chaque *Rôle coopératif* peut contenir une ou plusieurs *Catégories de service*. Chaque *Catégorie de service* peut être composée par une ou plusieurs *Classes de service*. Et chaque *Classe de service* peut instancier un ou plusieurs services. De cette manière, chaque service peut être identifié essentiellement par ces trois types architecturaux.

Ces trois types architecturaux sont définis comme suit :

Définition 4.3.1 (Rôle coopératif)**.** *Représente une unité logique de déploiement caractérisant les sites où les acteurs participant à l'application sont connectés.*

Dans la pratique un *Rôle coopératif* peut être associé, plus particulièrement à une entité d'exécution, telle que un serveur d'application ou une machine d'orchestration. Mais il n'existe pas de corrélation entre un *Rôle coopératif* et une entité d'exécution en général, un *Rôle coopératif* peut correspondre à plusieurs entités d'exécution, ou bien une entité d'exécution peut héberger plusieurs services associés à plusieurs *Rôles coopératifs*.

La notion de *Rôle coopératif* satisfait des besoins retrouvés dans toute application basée sur les services Web, en distinguant la distribution logique des sites hébergeant des services Web. Concrètement, quelques exemples de *Rôles coopératifs* sont : « Author Site », « Supplier Site », « TrackChair Site », « Client Site »,...

Définition 4.3.2 (Catégorie de service)**.** *Représente une fonctionnalité globale communément offerte par un ensemble de services Web.*

Les services Web appartenant à la même *Catégorie de service* peuvent être déployés sur un ou plusieurs noeuds ayant différents *Rôles coopératifs*. Nous distinguons, par exemple les Catégories de service suivantes : « Travel Agency Management », « Food Shopping Management », « Cooperative Review Management », ...

Définition 4.3.3 (Classe de service)**.** *Correspond au comportement fonctionnel offert par un type particulier de service Web, en tant qu'unité logique de base associée à une Catégorie de service.*

Dans un scénario de déploiement, une *Classe de service* représente une partie spécifique de la logique métier offerte par une *Catégorie de service* étant déployée sur un noeud associé à un *Rôle coopératif*. Pour un souci de simplicité, nous considérons une seule *Classe de service* de la même *Catégorie de service*, étant déployée sur un ensemble de noeuds associés à différents *Rôles coopératifs*. Ainsi, nous distinguos par exemples, les *Classes de service* suivantes : « Booking Manager », « Customer Manager », « Reviewing Manager »,...

4.3.1 Métamodèle UML pour la description des architectures

Le standard UML propose une vue de déploiement à travers la définition de diagrammes de déploiement OMG (2005). Ces diagrammes sont essentiellement faits pour décrire une structure statique basée sur les notions de noeud, artefact, relations de communication et dépendance. Nous considérons qu'il est nécessaire d'étendre ces concepts pour représenter les exigences des applications suivant une approche orientée service,

telles que les applications à base de services Web. Dans ce genre de systèmes on doit être capable de représenter non seulement une distribution statique des composants, mais aussi les changements (dans ce cas, au niveau architectural) nécessaires, par exemple, pour palier à d'éventuels problèmes pendant l'exécution de l'application. Pour ceci, des modifications au métamodèle de déploiement UML, s'avèrent nécessaires. Parmi les différentes façons d'étendre UML, nous adoptons, pour notre solution, une stratégie de type lourd (heavyweigt). Une extension lourde implique l'ajout de nouveaux éléments et l'éventuelle modification de la sémantique des éléments UML de base.

Le métamodèle défini pour notre extension est illustré par la figure 4.2. Les restrictions liées à la définition de ce métamodèle sont les suivantes :

- chaque type architectural étend la métaclasse *Node*, alors que le Service étend la métaclasse *Component*,
- une relation d'agrégation est établie entre le Rôle coopératif et la Catégorie de service, du fait qu'une même Catégorie de service peut être déployée sur plusieurs Rôles coopératifs,
- une relation de composition est établie entre la Catégorie de service et la Classe de service, du fait qu'une même Classe de service ne peut appartenir qu'à une Catégorie de service.

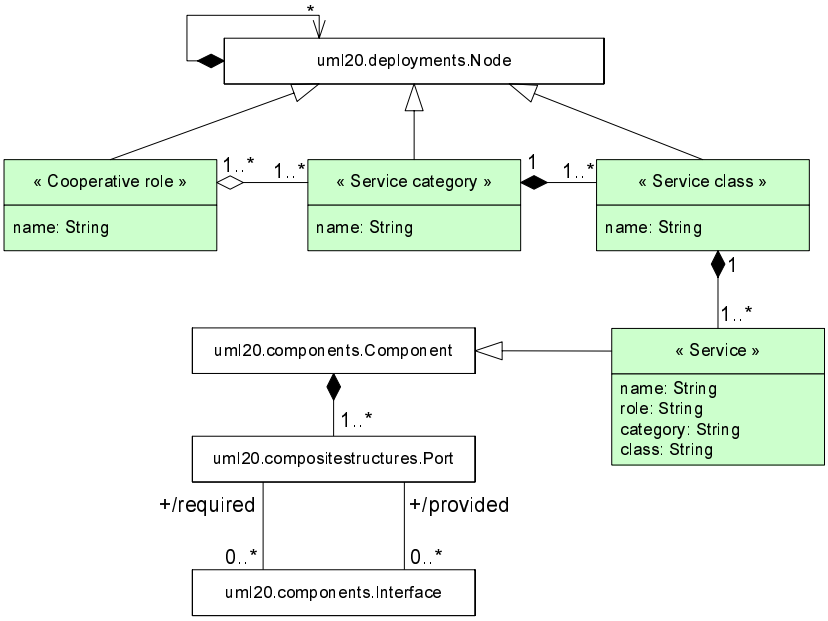


FIG. 4.2 – Un métamodèle étendant la vue de déploiement UML.

En accord avec le métamodèle, donc les notions de Rôle coopératif, Catégorie de service et Classe de service sont toutes représentées par le symbole de noeud. Une étiquette

permet de distinguer chaque élément par son nom. Les dépendances dérivées des types architecturaux sont implicitement représentés par imbrication de chaque type avec sa contre partie, en respectant l'ordre établi par le métamodèle. On utilise également des liens de communication (communication path), y compris leur multiplicité ; comme établi dans un diagramme de déploiement afin de spécifier les interactions entre Classes de services. Ceci est illustré par la figure 4.3

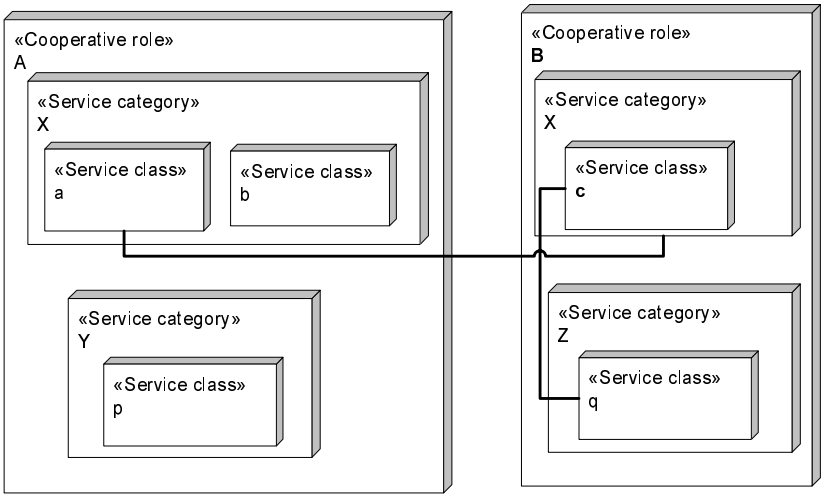


FIG. 4.3 – Description graphique d'une application de services Web modélisée par les notions de *Rôle coopératif*, *Catégorie de service* et *Classe de service*

4.3.2 Interactions entre services

Un diagramme supplémentaire est établi afin de représenter les instances de services et leurs interactions (voir figure 4.4). Du fait qu'un service étend la métaclasse *Component*, les interactions entre services utilisent les mêmes notions de ports et d'interfaces. Ainsi, chaque service est caractérisé par un dessin de composant. De même, des étiquettes rappellent les dépendances par rapport aux types architecturaux présentés précédemment. Ainsi, dans une instance de service, le *Rôle coopératif* est identifié par l'étiquette *Cr*, la *Catégorie de service* est identifiée par l'étiquette *Sct*, et la *Classe de service* est identifiée par l'étiquette *Scl*. Les services et leurs interactions doivent respecter les restrictions imposées par le métamodèle.

4.3.3 Schéma XML pour la description architecturale

Par rapport à la vue réalisation annoncée dans l'introduction de ce chapitre, nous définissons des schémas XML afin de traduire les architectures issues du métamodèle, sur un langage permettant la manipulation de leurs éléments. Par la suite nous allons décrire les principales structures du schéma XML liées à la spécifications des architectures des applications à base de services Web.

Une architecture spécifiant une application à base de services Web est définie par un type complexe *ArchType*. Ce type est composé par un élément *CoopRole* représentant les Rôles coopératifs. Un autre élément *InterRoleLink* définit les relations entre services déployés sur les différents Rôles coopératifs. De même, un attribut *archDescription* permet d'associer un nom ou description aux architectures. Le code XML consacré à cette partie est montré dans la figure 4.5. Le code complet du schéma XML pour la spécification des architectures est listé dans l'annexe A.

Les Rôles coopératifs sont caractérisés par un type complexe *RoleType*. Ce type est composé par un élément *ServiceCategory* qui définit les Catégories de service. Un autre élément *IntraRoleLink* permet d'établir les relations entre les services déployés sur les nœuds associés au même Rôle coopératif. De même, un attribut *roleName* vise à associer un nom aux Rôles coopératifs. La figure 4.6 illustre le code XML correspondant à cette partie.

Les Catégories de service sont caractérisées par un type complexe *CategoryType*. Ce type est définit par un élément *serviceClass* représentant les Classes de service associées à la Catégorie de service. Un attribut *catName* est aussi établi pour nommer les Catégories de service. Le code utilisé pour cette partie est montré dans la figure 4.7.

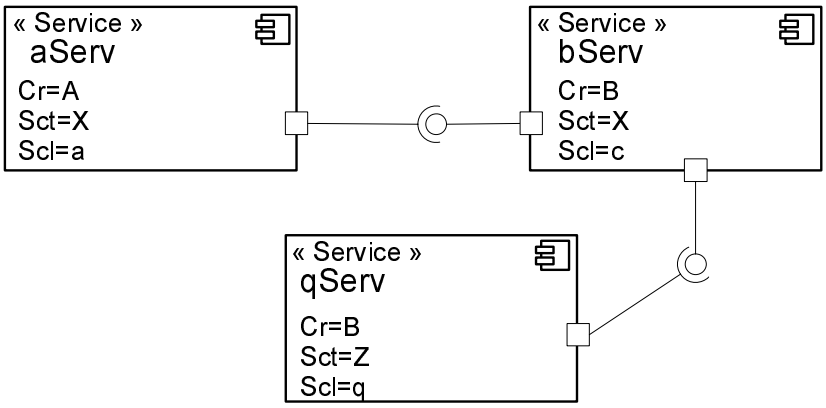


FIG. 4.4 – Description graphique des interactions entre services Web

```
<xs:complexType name="ArchType">
 <xs:sequence>
  <xs:element name="CoopRole" type="RoleType" maxOccurs="unbounded"/>
  <xs:element name="InterRoleLink" type="InterRoleLinkType" maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="archDescription" type="xs:string" use="required"/>
</xs:complexType>
```

FIG. 4.5 – Description architecturale

Les Classes de service sont caractérisées par le type complexe *ClassType*. Ce type est constitué d'un élément *Service* représentant les instances de services dérivés des Classes de service. Un attribut *className* permet de nommer les Classes de service. Ceci est montré dans le code de la figure 4.8.

```
<xs:complexType name="RoleType">
 <xs:sequence>
  <xs:element name="ServiceCategory" type="CategoryType" maxOccurs="unbounded"/>
  <xs:element name="IntraRoleLink" type="IntraRoleLinkType" maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="roleName" type="xs:string" use="required"/>
</xs:complexType>
```

FIG. 4.6 – Description des Rôles coopératifs

```
<xs:complexType name="CategoryType">
 <xs:sequence>
  <xs:element name="serviceClass" type="ClassType" maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="catName" type="xs:string" use="required"/>
</xs:complexType>
```

FIG. 4.7 – Description des Catégories de service

```
<xs:complexType name="ClassType">
 <xs:sequence>
  <xs:element name="Service" type="ServiceType"/>
 </xs:sequence>
 <xs:attribute name="className"/>
</xs:complexType>
```

FIG. 4.8 – Description des Classes de service

Un service est représenté par une séquence d'éléments représentant chacun des types architecturaux associés au service. Ceci est illustré par la figure 4.9.

Les connections entre services déployés à l'intérieur des nœuds associés au même Rôle coopératif sont caractérisées par un type complexe *IntraRoleLinkType*. Ce type permet de définir la mise en relation des deux éléments *service1* et *service2* définis chacun par le type complexe *service* (figure 4.10). De manière similaire les connections entre services déployés dans deux types Rôles coopératifs différents sont caractérisées par le type complexe *InterRoleLinkType* (figure 4.11).

```
<xs:complexType name="ServiceType">
 <xs:sequence>
  <xs:element name="role">
   <xs:complexType>
    <xs:attribute name="roleName" type="xs:string" use="required"/>
   </xs:complexType>
 </xs:element>
 <xs:element name="category">
  <xs:complexType>
   <xs:attribute name="catName" type="xs:string" use="required"/>
  </xs:complexType>
 </xs:element>
 <xs:element name="class">
  <xs:complexType>
   <xs:attribute name="className"/>
 </xs:complexType>
 </xs:element>
</xs:sequence>
 <xs:attribute name="serviceName" use="required"/>
</xs:complexType>
```

FIG. 4.9 – Description des services

```
<xs:complexType name="IntraRoleLinkType">
 <xs:sequence>
  <xs:element name="service1" type="ServiceType"/>
  <xs:element name="service2" type="ServiceType"/>
 </xs:sequence>
</xs:complexType>
```

FIG. 4.10 – Description des liens intra-rôle

4.4 Définition de règles de base

Cette démarche consiste en la définition de règles génériques afin d'introduire des actions de reconfiguration. La définition de règles est essentiellement basée sur les types architecturaux de base. C'est-à-dire chaque règle implique, dans sa spécification, l'utilisation des Rôles coopératifs, Catégories de service et Classes de service, seuls ou combinés. Le but de cette stratégie est de tirer avantage des aspects tels que :

- la définition de règles simples (impliquant un minimum d'éléments),
- la définition de règles génériques. Cette généricité est liée au fait que chaque type architectural généralement implique plusieurs instances de services Web.

Avant de définir les règles de base nous devons traduire l'architecture de l'application sur une notation basée sur les graphes. Dans cette notation, les services Web sont représentés par les nœuds du graphe et les connections entre services par les arcs du graphe. L'idée est d'appliquer des règles de transformation de graphes afin de caractériser l'adaptation dynamique de l'architecture d'une application à base de services Web. De manière plus précise, pour la représentation des règles de base nous adoptons une notation basée sur des techniques issues de la réécriture des graphes. La notation à utiliser suit l'approche ACG (ang. Abstract Component Graph) introduite dans Guennoun *et al.* (2004). Cette approche est basée sur les graphes et les règles de coordination pour décrire l'évolution dynamique des architectures. Elle est aussi employée pour simuler les différentes étapes d'instanciation des composants, le changement de comportement pendant l'exécution, la migration, et d'autres caractéristiques spécifiques aux architectures logicielles des systèmes distribués. Dans cette approche une règle est partitionnée en quatre zones permettant de spécifier les contraintes qui conditionnent l'application de la règle et les transformations qui se produisent (représentant l'adaptation dynamique de l'architecture) quand une règle est applicable. Ces zones sont les suivantes :

- La zone *Inv* représente un fragment du graphe de la règle qui doit être identifié (par homomorphisme) dans le graphe de l'architecture, si plusieurs sous graphes sont homomorphes à cette zone, un sous graphe est choisi aléatoirement ;

```
<xs:complexType name="InterRoleLinkType">
 <xs:sequence>
  <xs:element name="service1" type="ServiceType"/>
  <xs:element name="service2" type="ServiceType"/>
 </xs:sequence>
</xs:complexType>
```

FIG. 4.11 – Description des liens inter-rôle

- La zone *Abs ou Rest* représente un fragment du graphe de la règle qui ne doit pas être identifié (par homomorphisme) dans le graphe de l'architecture pour que la règle soit applicable ;
- La zone *Del* est un fragment du graphe de la règle qui sera supprimé à l'application de la règle ;
- La zone *Add* est un fragment du graphe de la règle dont une copie isomorphe sera ajoutée après l'application de la règle.

Afin de définir les règles de base nous considérons un service Web[1] comme un élément d'un graphe de l'architecture défini par :

$WS_{(i,j,k,l)}$ où :
i représente un Rôle coopératif,
j représente une Catégorie de service,
k représente une Classe de service, et
l représente l'identifiant d'un service Web.

Dans la plupart de cas, les variables *i,j,* et *k* sont suffisantes pour distinguer les services Web impliqués dans la définition de règles. Le cas échéant, l'identifiant *l* devrait être utilisé afin de distinguer les services Web comportant des valeurs similaires dans *i,j,* et *k*.

Par la suite on présente trois règles de base définies à partir d'une séquence d'actions élémentaires. Dans ces règles, les variables X, Y et Z correspondent à des motifs représentent plusieurs instances de services Web. Les règles de base sont définies comme suit.

Règle 1

Implique l'ajout des éléments. Dans la pratique, ceci pourrait entraîner l'activation ou le déploiement d'instances de services Web. La figure 4.12 illustre une description textuelle de cette règle, alors que la figure 4.13 illustre sa représentation graphique.

Règle 2

Implique l'ajout des connexions. Une restriction est nécessaire au niveau des connexions entre une paire d'éléments puisque leurs connexions doivent rester uniques. La fi-

[1]Ici, service Web représente un type à partir duquel seront générées les instances de services Web.

gure 4.14 illustre une description textuelle de cette règle, alors que la figure 4.15 illustre sa représentation graphique.

Règle 3

Implique la suppression des éléments et connexions respectives. On doit remarquer que, du fait d'associer les paramètres aux types architecturaux, chaque variable pourrait affecter plusieurs éléments. La figure 4.16 illustre une description textuelle de cette règle,

```
Rule [name=R1](i,j,k,l,i',j',k',l')
 Match Vertices:
   X with parameters i,j,k,l;
 Add Vertices:
   Y with parameters i',j',k',l';
```

FIG. 4.12 – Règle 1 (représentation textuelle).

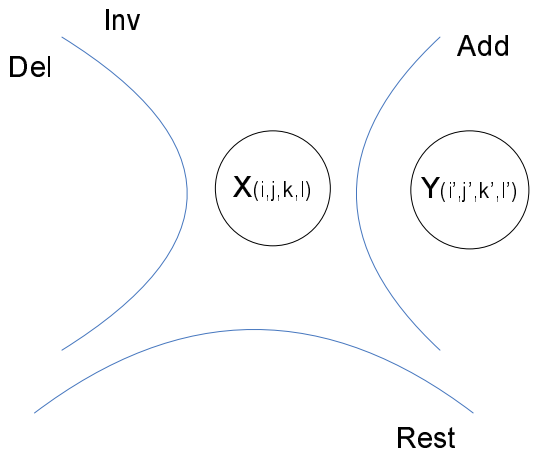


FIG. 4.13 – Règle 1 (représentation graphique).

```
Rule [name=R2](i,j,k,l,i',j',k',l',i'',j'',k'',l'')
 Match Vertices:
   X with parameters i,j,k,l;
   Y with parameters i',j',k',l';
   Z with parameters i'',j'',k'',l'';
 Match Edges:
   X->Z, Z->X
 Add Edges:
   Y->Z, Z->Y
 Restriction Edges:
   Y->Z, Z->Y
```

FIG. 4.14 – Règle 2 (représentation textuelle).

alors que la figure 4.17 illustre sa représentation graphique.

En considérant ces principes, une règle de base décrivant, par exemple : « l'ajout de nouvelles instances de services Web ayant pour Catégorie de service *AuthorManagement* », serait réalisée en appliquant la Règle R1 avec le motif $Y_{(j=AuthorManagement)}$. Cette règle concerne tous les éléments associés à la Catégorie de service impliquée, c'est-à-dire toutes les Classes de services plus les instances de service Web définis sous

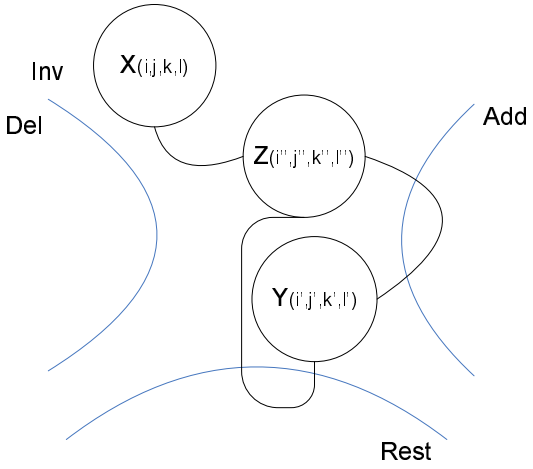


FIG. 4.15 – Règle 2 (représentation graphique).

```
Rule [name=R3](i,j,k,l)
 Delete Vertices:
   X with parameters i,j,k,l;
```

FIG. 4.16 – Règle 3 (représentation textuelle).

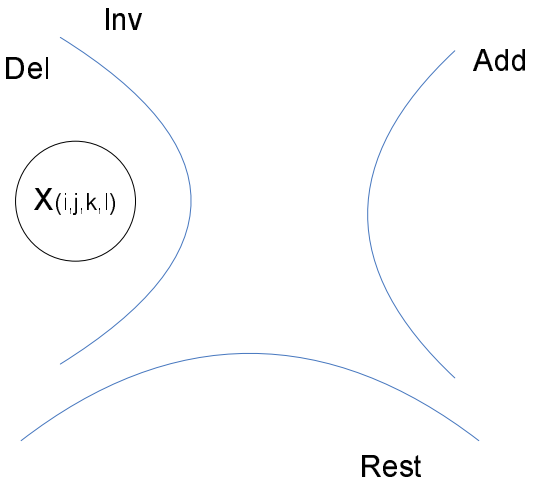


FIG. 4.17 – Règle 3 (représentation graphique).

la Catégorie de service *AuthorManagament*.

Un autre exemple serait : « la désactivation des services Web associés au Rôle coopératif *TrackChair* et à la Catégorie de service *ConferenceManagement* ». Pour ceci, nous appliquons la Règle R3 avec le motif $X_{(i=TrackChair,j=ConferenceManagement)}$. De nouveau, cela implique tous les éléments associés à ce Rôle coopératif et à cette Catégorie de service.

4.4.1 Schéma XML pour les règles de base

Nous avons défini un schéma XML afin de représenter les règles de base. Cette schéma hérite certains types définis dans le schéma représentant les architectures. La figure 4.18 montre la définition du type complexe *ruleType*, il permet de décrire les quatre zones impliquées dans une règle de base. Chacune des zones est définie par un type complexe nommé *Node*. La spécification du type *Node* est montrée dans la figure 4.19. Le code complet du schéma XML pour la spécification des règles de base est listé dans l'annexe A.

```
<xs:complexType name="ruleType">
 <xs:sequence>
  <xs:element name="Add" type="Node"/>
  <xs:element name="Del" type="Node"/>
  <xs:element name="Inv" type="Node"/>
  <xs:element name="Abs" type="Node"/>
  <xs:element name="IntersectionLinks" minOccurs="0" maxOccurs="unbounded">
 <xs:complexType>
  <xs:complexContent>
   <xs:extension base="archCoop:InterRoleLinkType">
    <xs:attribute name="InvolvedSections" use="required">
     <xs:simpleType>
      <xs:restriction base="xs:string"/>
     </xs:simpleType>
    </xs:attribute>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
 </xs:element>
</xs:sequence>
</xs:complexType>
```

FIG. 4.18 – Description des règles de base

Par rapport aux connexions entre nœuds, nous trouvons la définition des connexions intra-zone par le biais de l'élément *Interaction* défini dans le type Node. Alors que les

interactions inter-zone sont définies par le biais de l'élément *IntersectionLinks* défini dans le type *ruleType*. Dans la définition de ce même type, un attribut *InvolvedSections* permet d'identifier les zones impliquées par la connexion.

```
<xs:complexType name="Node">
 <xs:sequence>
 <xs:element name="service" type="archCoop:ServiceType"/>
 <xs:element name="Interaction" type="archCoop:IntraRoleLinkType" minOccurs="0"
                maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>
```

FIG. 4.19 – Description d'un nœud et ses connexions à l'intérieur d'une zone de la règle

4.5 Définition de règles de reconfiguration

Afin de définir la dynamique de la reconfiguration au niveau architectural nous proposons l'utilisation de règles de transformation. Cette dynamique considère les actions classiques nécessaires pour le diagnostic et la reconfiguration des applications.

Pour définir les actions de reconfiguration requises par un système de services Web, on se sert des règles de base et on définit les politiques pour sa correcte application. C'est-à-dire, on définit une séquence adéquate de règles de base afin d'établir des actions telles que la duplication ou la substitution de services Web.

La règle de *duplication*

La duplication considère l'addition de services ayant des fonctionnalités équivalentes. Le but est de bien repartir la charge entre les services afin de prévenir la dégradation de la qualité de service.

L'action de reconfiguration nommée *Duplication* est définie par la séquence de règles de base suivantes :

1. R1$(X_{(i,j,k,l)}, Y_{(i',j',k',l')})$. // appliquer la règle R1 pour toutes les instances de services Web identifiées (par homomorphisme) à partir des motifs X et Y.

2. R2($X_{(i,j,k,l)}, Y_{(i',j',k',l')}, Z_{(i'',j'',k'',l'')}$). // appliquer la règle R2 pour toutes les instances de services Web identifiées (par homomorphisme) à partir des motifs X,Y et Z.

La règle de *substitution*

La substitution considère la redirection du flux du processus entre deux services Web. L'application de cette action est traduite par la désactivation du premier service et son remplacement par le second.

L'action de reconfiguration nommée *Substitution* est définie par la séquence de règles de base suivantes :

1. R1($X_{(i,j,k,l)}, Y_{(i',j',k',l')}$). // appliquer la règle R1 pour toutes les instances de services Web identifiées (par homomorphisme) à partir des motifs X et Y.
2. R2($X_{(i,j,k,l)}, Y_{(i',j',k',l')}, Z_{(i'',j'',k'',l'')}$). // appliquer la règle R2 pour toutes les instances de services Web identifiées (par homomorphisme) à partir des motifs X,Y et Z.
3. R3($X_{(i,j,k,l)}$). // appliquer la règle R3 pour toutes les instances de services Web identifiées (par homomorphisme) à partir du motif X.

4.5.1 Outils pour la transformation des graphes

Tous les éléments décrits dans notre stratégie peuvent être traduites dans un outil logiciel implémentant des techniques de transformation de graphes. Dans le groupe OLC du LAAS-CNRS et dans le cadre de la thèse de Guennoun (2006), un tel outil a été développé. Les résultats en performance, pour la première version, sont encourageants. Néanmoins, dans son état actuel, on se trouve avec une contrainte majeure, du fait que les règles sont appliquées sur une seule instance (choisie arbitrairement) des homomorphismes trouvés dans le graphe de l'architecture. Dans notre approche, ceci nous empêche d'actualiser le graphe de l'architecture correctement, du fait que pour nous, la règle doit s'appliquer sur toutes les instances associées aux types architecturaux définis dans la règle.

Il existe d'autres outils du même genre et développés dans d'autres institutions,

tels que : AGG[2] ou Fujaba[3], néanmoins ils restent inadéquats du fait de son faible performance pour traiter des architectures à grande échelle.

4.6 Conclusion

Dans ce chapitre nous avons présenté une approche pour la description des architectures des applications à base de services Web. La proposition représente une approche originale visant l'automatisation du processus de gestion des architectures par des actions de reconfiguration.

De même, notre approche associe de façon originale, des techniques semi-formelles (UML), mais amplement utilisées ; avec des techniques formelles (technique de réécriture de graphes) qui sont moins connues dans les milieux industriels et d'entreprise. La description des architectures par les types structuraux, a permis un niveau d'abstraction adéquate, autorisant la construction des règles de base simples, et une manipulation qui s'avère performante pour l'application des techniques de réécriture de graphes.

Le processus d'automatisation de notre approche est partiellement achevé. Une traduction des diagrammes UML par le biais des plugins de l'outil Fujaba vers le langage XML a été développée dans le cadre d'une collaboration avec l'Université de Sfax en Tunisie. De même, le moteur de transformation de graphes en cours de développement dans le LAAS-CNRS est en phase de consolidation.

[2]http ://tfs.cs.tu-berlin.de/agg/
[3]http ://wwwcs.uni-paderborn.de/cs/fujaba/

Chapitre 5

Scénarios applicatifs

5.1 Introduction

Dans ce chapitre on introduit deux scénarios applicatifs illustrant l'application de l'approche introduite dans le chapitre précédant. Un premier exemple traite le cas d'une application de revue coopérative, affecté au cas particulier d'un système de gestion de conférences scientifiques. Le deuxième exemple considère une application de magasin en ligne. Dans les deux exemples les architectures, les acteurs, les activités et les actions de reconfiguration sont présentées.

5.2 La revue coopérative

Cet exemple illustre le fonctionnement d'une activité coopérative commune adressant le problème du « processus de revue ». Ce problème connaît plusieurs adaptations dans différents domaines d'activité[1]. L'application de la revue coopérative que nous étudions dans le contexte du projet DIAMOND IST (2006) a pour but soutenir ce genre d'activités.

L'ensemble des services, des paramètres, des acteurs et des processus sont définis de façon générique et peuvent être appliqués à différents domaines applicatifs ayant besoin

[1]Dans les activités industrielles, telles que l'ingénierie de systèmes embarqués complexes, le « processus de revue » est connu comme l'un des plus complexes de l'industrie aérospatiale (voir le projet IST-DSE Baurens *et al.* (2001))

du processus de revue. Afin de faciliter la compréhension de cet exemple nous nous focalisons sur le classique scénario du processus de revue des publications scientifiques. Nous abordons le cas des conférences scientifiques, en décrivant le fonctionnement automatique de ses différentes activités, par le moyen de la conception d'un système de gestion de conférences qui suit une approche orientée service. Plusieurs travaux dans la littérature ont abordé cette problématique, notamment : Nierstrasz (2000); Papagelis *et al.* (2005); Mathews et Jacobs (1996).

Cette description prend en compte les différentes étapes caractérisant les activités notamment :

1. la recherche par les auteurs des conférences satisfaisant certains critères (par exemple : thème de recherche, réputation de la conférence, date limite pour la soumission et la publication des articles, maison d'édition,...),
2. la recherche par le Président du comité de programme des relecteurs satisfaisant certains critères (par exemple : domaine de recherche, niveau d'expertise, laboratoire de recherche,...),
3. le processus de soumission d'articles par les auteurs,
4. le processus d'évaluation des articles par les relecteurs,
5. la récupération des rapports d'évaluation,
6. la diffusion des résultats issus des rapports envoyés par les relecteurs,
7. l'édition et publication des actes.

5.2.1 Acteurs et Workflow

Plusieurs acteurs coopèrent afin d'accomplir les différentes tâches de gestion du processus de revue coopérative. Ces acteurs sont définis comme suit :

- Le Président du comité de programme (PCP) représente l'acteur principal dans l'organisation d'une conférence. Ses tâches sont requises dans diverses étapes du système, que ce soit dans le planning des conférences ou dans la publication des actes. Dans la pratique, cette responsabilité peut être partagée entre plusieurs personnes.
- Les responsables de thèmes (RT) sont en charge de gérer chacune des sessions qui composent une conférence.
- Les auteurs représentent chacun des participants potentiels aux sessions d'une conférence.

– Les relecteurs représentent les experts dans chacun des domaines de recherche issus d'une conférence. Ils sont choisis en fonction de leur compétence pour produire un rapport objectif sur les articles qui leurs sont affectés.

Pour la représentation du workflow du système de gestion de conférences, nous adoptons une approche que nous appelons coopérative. Dans cette approche nous nous intéressons aux interactions entre services Web issues des activités caractérisant les fonctionnalités offertes par le système. Ces activités correspondent aux différentes phases du processus de revue coopérative. Ainsi, le workflow du système correspond au comportement externe de ses composants, par opposition à leur comportement interne.

5.2.2 Description des phases

Plusieurs interactions, organisées sous forme d'activités entre services Web, pourraient se mettre en place pour définir la dynamique de cette application. Par la suite nous décrivons les plus importantes. Du fait de l'intérêt dans les interactions entre les services Web du système, nous proposons une description par le biais de diagrammes de séquence.

Phase de recherche de conférences

Cette phase décrit le processus où les auteurs cherchent des « appels à communication » proches de leur domaine de recherche.

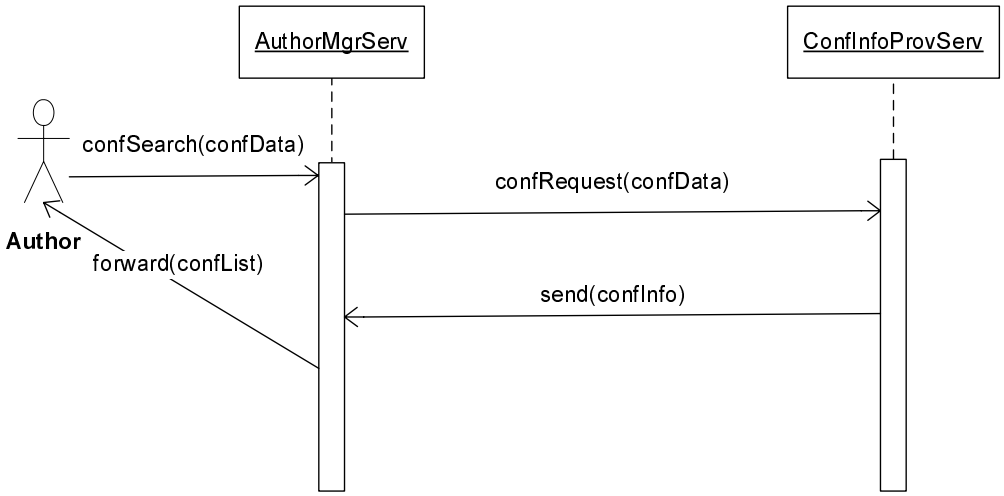


FIG. 5.1 – Recherche de conférences.

La figure 5.1 illustre le diagramme de séquence concernant cette phase. Initialement, un auteur (Author) génère une requête *confSearch(confData* visant la recherche de conférences. La requête est envoyée par le service Web *AuthorMgrSrv* vers le service Web *ConfInfoProvServ*. A son tour le service Web *ConfInfoProvServ* répond au service Web *AuthorMgrSrv* en envoyant l'information des conférences *send(confInfo)* satisfaisant les critères de choix établis par l'auteur.

Phase de recherche de relecteurs

Cette phase concerne le processus de recherche et de sélection des relecteurs. Les relecteurs potentiels sont choisis par leurs qualifications et leur niveau d'expertise dans un domaine donné. Ils sont sollicités pour manifester leur intérêt de participation aux évaluations des articles soumis.

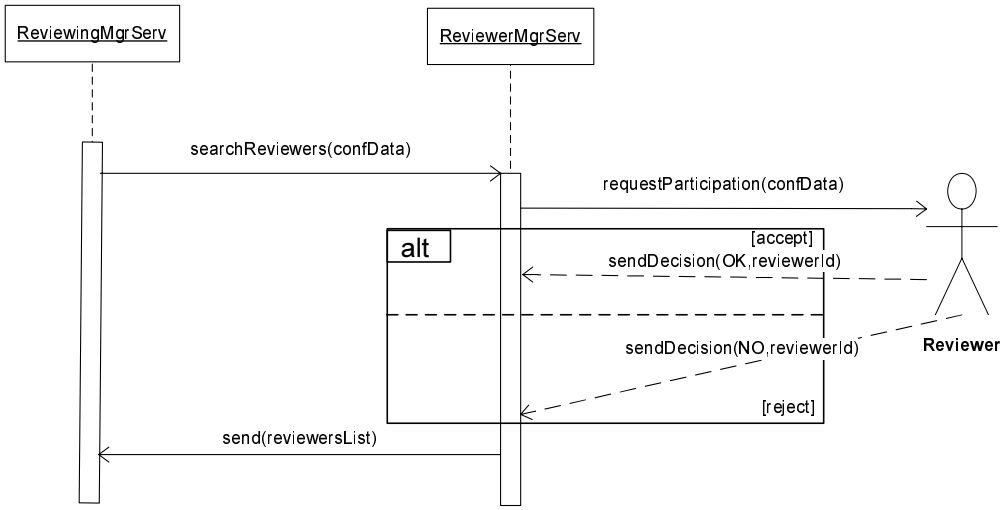


FIG. 5.2 – Recherche de relecteurs.

La figure 5.2 illustre le diagramme de séquence concernant cette phase. Le service Web *ReviewingMgrServ* envoie une requête *searchReviewersData(confData)* contenant les données de la conférence, et visant à inviter aux relecteurs à participer dans une conférence. Cette requête est destinée au service Web *ReviewerMgrServ* qui â son tour l'adresse aux relecteurs. Chaque relecteur doit accepter ou refuser l'invitation. Les relecteurs qui ont accepté de participer à la conférence sont rassemblés par le service Web *ReviewerMgrServ* qui adresse cette information au service Web *ReviewingMgrServ*.

Phase d'inscription des auteurs

Cette phase considère le processus d'inscription des auteurs, nécessaire pour la soumission d'articles à la conférence.

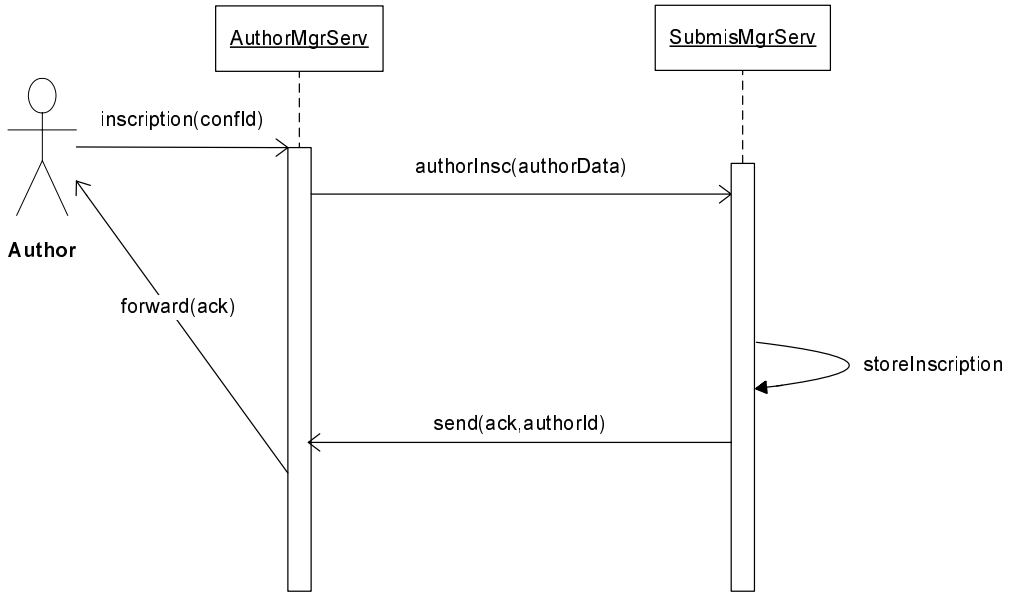


FIG. 5.3 – Inscription à une conférence.

La figure 5.3 illustre le diagramme de séquence concernant cette phase. Initialement, un auteur (Author) demande son inscription à une conférence *inscription(confId)*. Cette demande est prise par le service Web *AuthorMgrServ* qui génère une requête *authorIns(authorData,confId)* contenant : les données de l'auteur et de la conférence. Cette requête est destinée au service Web *SubmisMgrServ* qui enregistre à l'auteur et envoie une confirmation *send(ack,authorId)* au service Web *AuthorMgrServ*. Enfin, l'auteur est informé de la prise en compte de son inscription.

Phase de soumission d'articles

Cette phase implique la soumission d'articles par les auteurs intéressés à une conférence.

La figure 5.4 illustre le diagramme de séquence concernant cette phase. Initialement, un auteur (Author) demande la soumission d'un article à une conférence *submit(paper)*. Cette demande est prise par le service Web *AuthorMgrServ* qui génère une requête *submit(authorId,paper)* contenant : l'identifiant de l'auteur et le papier à déposer. Cette

requête est destinée au service Web *SubmisMgrServ* qui enregistre la soumission et envoie une confirmation *send(ack,authorId,paperId)* au service Web *AuthorMgrServ*. Enfin, l'auteur est informé de la prise en compte de sa soumission.

Phase d'affectation d'articles

Cette phase implique le processus d'affectation des articles aux relecteurs par les membres du comité de programme.

La figure 5.5 illustre le diagramme de séquence concernant cette phase. Après d'avoir crée la liste d'affectation des articles, le service Web *ReviewingMgrServ* envoie une requête *send(paper, paperId, reviewerId)* contenant : l'article à qualifier, l'identifiant du papier et l'identifiant du relecteur. Cette requête est prise par le service Web *ReviewerMgrServ*, qui adresse le papier aux relecteurs concernés. Le relecteur doit accuser de réception, ainsi au travers du service Web *ReviewerMgrServ*, une confirmation *send(ack,paperId,reviewerId)* est adressée au service Web *ReviewingMgrServ*.

Phase de transmission de rapports

Cette phase implique l'envoie des rapports de revue par les relecteurs.

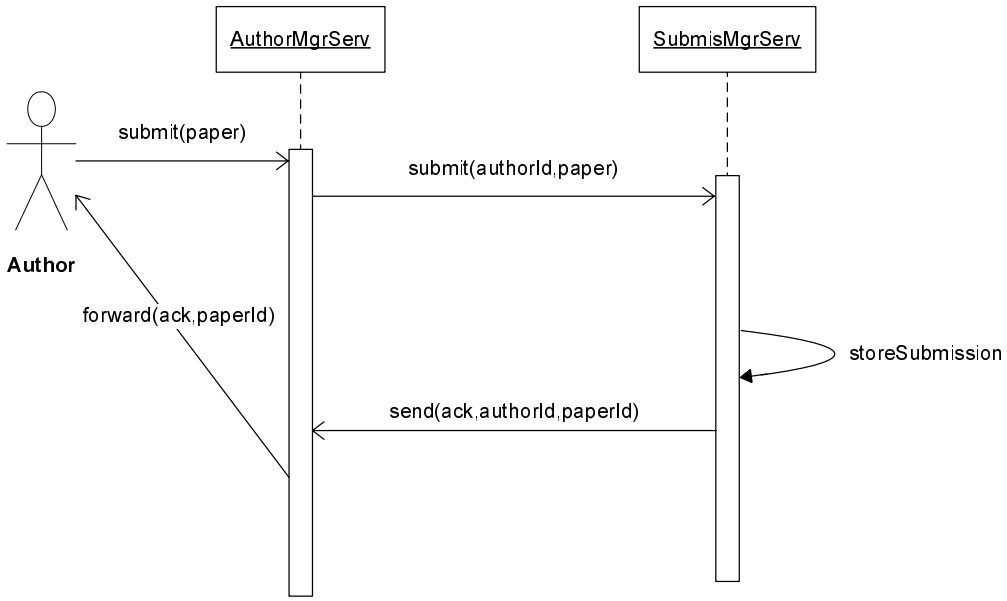


FIG. 5.4 – Soumission d'articles.

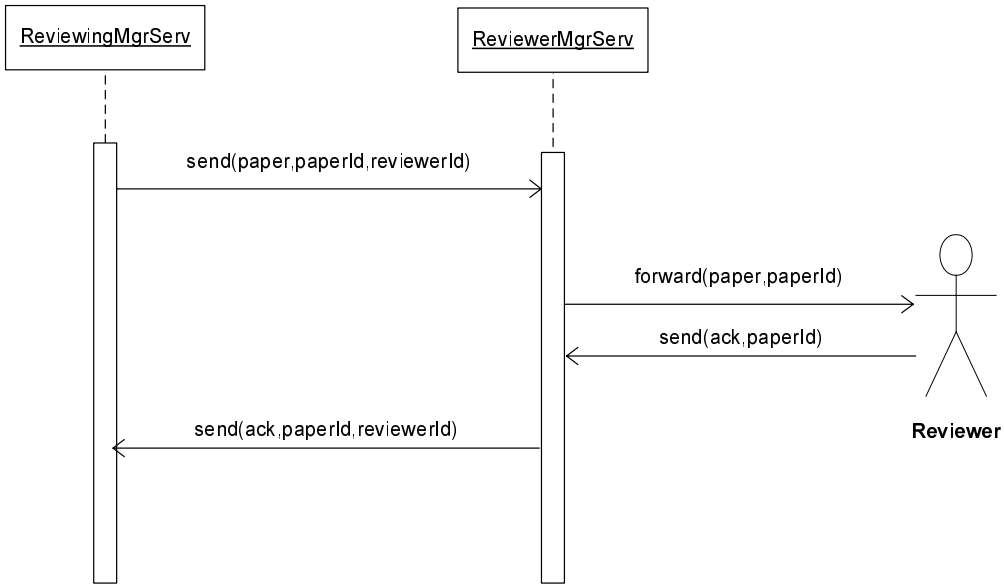


FIG. 5.5 – Affectation des articles.

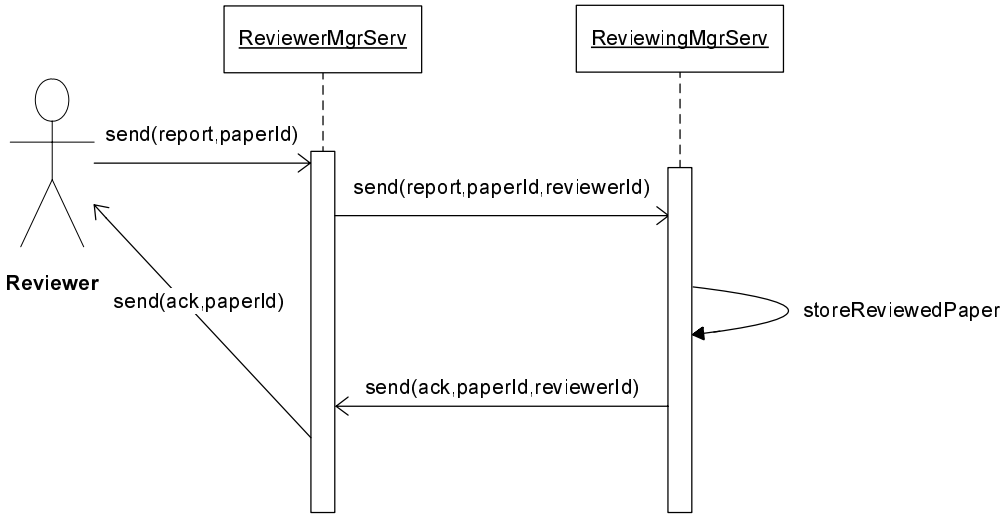


FIG. 5.6 – Transmission de rapports.

La figure 5.6 illustre le diagramme de séquence concernant cette phase. Initialement, un relecteur (Reviewer) demande la transmission de son rapport de revue d'un article *send(report,paperId)*. Cette demande est prise par le service Web *ReviewerMgrServ* qui génère une requête *send(report,paperId,reviewerId)* contenant : le rapport, l'identifiant de l'article et l'identifiant du relecteur. Cette requête est destinée au service Web *ReviewingMgrServ* qui enregistre le rapport et envoie une confirmation *send(ack,paperId,reviewerId)* au service Web *ReviewerMgrServ*. Enfin, le relecteur est informé de la prise en compte de son rapport.

Phase de notification

Cette phase traite le processus de notification des rapports de revue aux auteurs. La décision sur l'acceptation ou le refus est prise par le PCP en se basant sur les rapports de revue envoyés par les relecteurs.

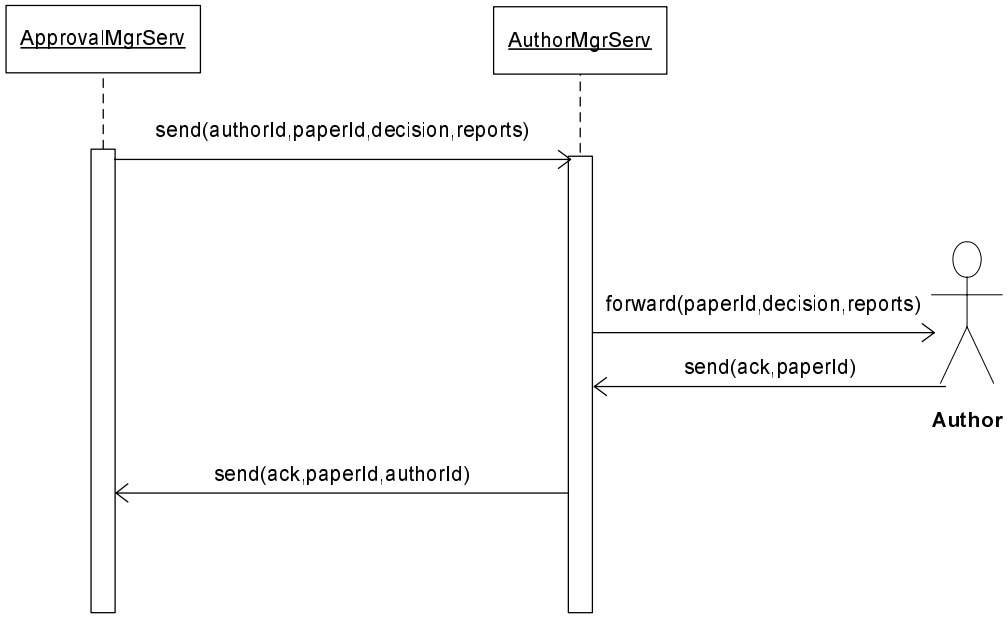


FIG. 5.7 – Notification d'évaluation.

La figure 5.7 illustre le diagramme de séquence concernant cette phase. Après d'avoir crée la liste des articles acceptés et la liste des articles refusés, le service Web *ApprovalMgrServ* envoie une requête *send(authorId,paperId,decision,rapports)* contenant : l'identifiant de l'author, l'identifiant du papier, la décision finale sur le papier (accepté ou refusé) et les rapports de revue. Cette requête est prise par le service Web *AuthorMgrServ*, qui adresse son contenu à l'auteur concerné. L'auteur doit accuser de réception, ainsi au travers du service Web *AuthorMgrServ*, une confirmation *send(ack,paperId,authorId)* est adressée au service Web *ApprovalMgrServ*.

Phase de soumission des versions finales des articles

Cette phase traite la soumission, par les auteurs des versions finales des articles sélectionnés afin d'éditer les actes.

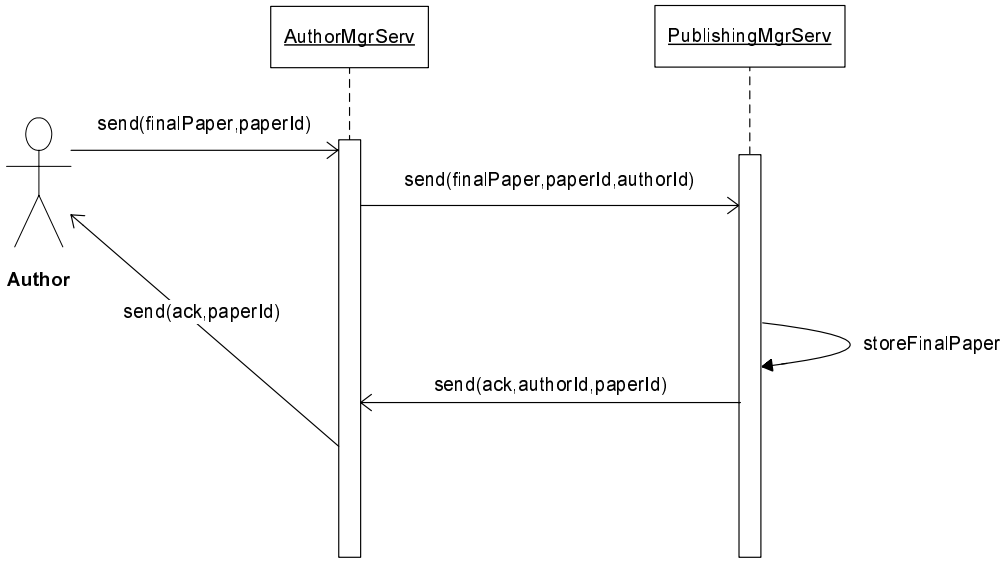


FIG. 5.8 – Soumission des articles pour édition.

La figure 5.8 illustre le diagramme de séquence concernant cette phase. Initialement, un auteur (Author) dont son papier a été accepté pour publication, demande la soumission de la version finale *submit(finalPaper,paperId)*. Cette demande est prise par le service Web *AuthorMgrServ* qui génère une requête *send(finalPaper,paperId,authorId)* contenant : la version finale du papier, l'identifiant du papier et l'identifiant de l'auteur. Cette requête est destinée au service Web *PublishingMgrServ* qui enregistre le papier et envoie une confirmation *send(ack,authorId,paperId)* au service Web *AuthorMgrServ*. Enfin, l'auteur est informé de la prise en compte de la version finale de son papier.

Dépendances entre phases

Les relations de dépendances entre les différentes phases du système de gestion de conférences sont illustrées dans la figure 5.9. Les deux premières phases, recherche de conférences et recherche de relecteurs peuvent être démarrées en parallèle. Après ces deux phases, l'inscription des auteurs et la soumission des articles doit être terminées avant de la phase d'affectation des articles. Ensuite, les phases de transmission de rapports et notification aux auteurs doivent se succéder. Enfin, lorsqu'un article est accepté la phase de soumission de version finale de l'article s'avère nécessaire.

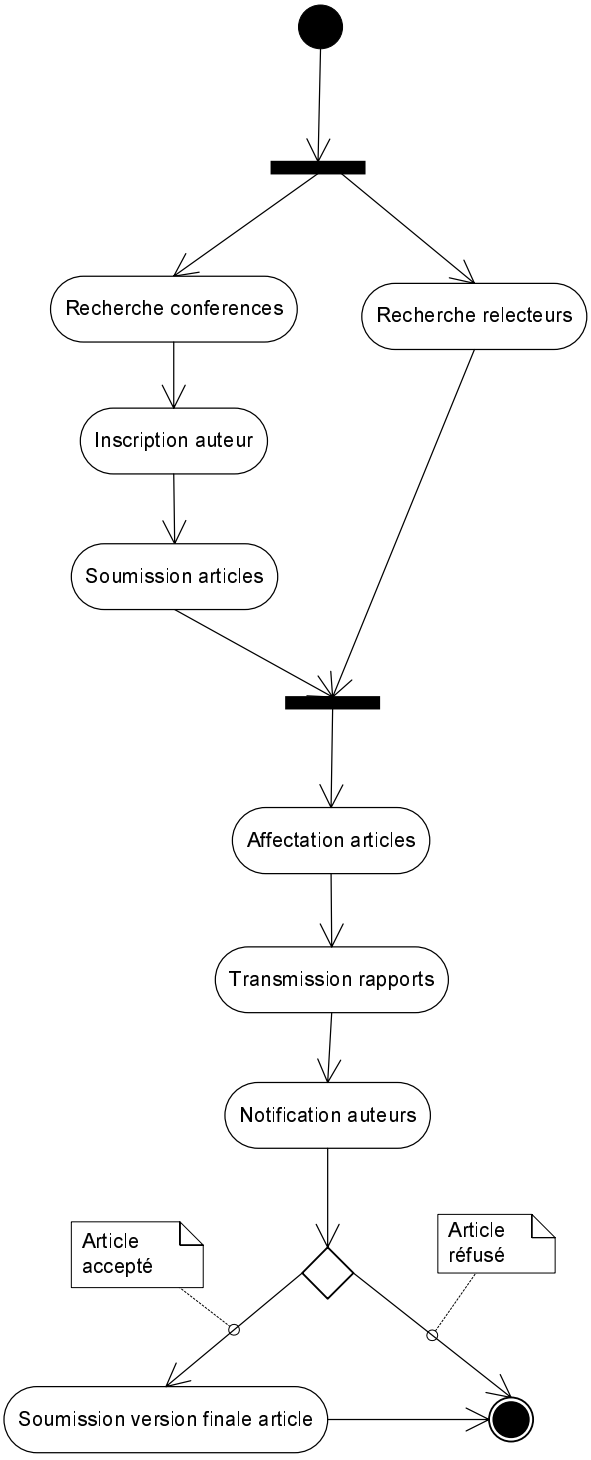


FIG. 5.9 – Relations de dépendances entre phases.

5.2.3 Description architecturale

La description architecturale concernant cet exemple est présentée dans la figure 5.10. Cette description implique :

- Quatre Rôles coopératifs :
  - *AuthorSite* implique les sites participants pour les auteurs,
  - *ConfChairSite* implique les sites participants pour les Présidents des conférences,
  - *TrackChairSite* implique les sites participants pour les track Chairs,
  - *ReviewerSite* implique les sites participants pour les relecteurs.
- Trois Catégories de services :
  - *AuthorMgmt* considère la gestion des fonctionnalités génériques vis-à-vis des auteurs,
  - *ConfMgmt* considère la gestion des fonctionnalités génériques vis-à-vis des conférences,
  - *ReviewerMgmt* considère la gestion des fonctionnalités génériques vis-à-vis de relecteurs.
- Sept Classes de Services :
  - *AuthorMgr* définit les fonctionnalités particulières concernant les auteurs,
  - *ConfInfoProv* définit les fonctionnalités concernant les renseignements liés aux conférences,
  - *ApprovalMgr* définit les fonctionnalités du processus d'approbation des publications,
  - *PublishingMgr* définit les fonctionnalités de gestion de publication des actes,
  - *SubmisMgr* définit les fonctionnalités de gestion liée au processus de soumission des papiers,
  - *ReviewingMgr* définit les fonctionnalités concernant la gestion du processus de révision des papiers,
  - *ReviewerMgr* définit les fonctionnalités particulières aux relecteurs.

5.2.4 Application des règles de reconfiguration

Afin de faciliter la lecture des actions de reconfiguration, nous utilisons un style left-right (gauche-droit). C'est-à-dire chaque action de reconfiguration est composée par un

côté gauche et un côté droit. Du côté gauche, on décrit les éléments de l'architecture (services Web et connections) nécessaires à l'application des règles. Et du côté droit, on décrit les éléments de l'architecture souhaitée par l'application des règles. Dans la pratique, les modifications subies par l'architecture comme résultat de l'application de l'action de reconfiguration sont obtenues en réalisant plusieurs actions élémentaires, notamment : l'activation et la désactivation de services et de connections.

Règle de duplication

Un premier exemple concernant les actions de reconfiguration appliquées au système de gestion de conférences considère la duplication de services Web. On pourrait imaginer plusieurs cas où cette action s'avère nécessaire, par exemple, des symptômes tels que la surcharge ou le débit faible. Comme illustré dans la figure 5.11 à un moment donné plusieurs services *AuthorMgrServ* pourraient essayer d'accéder à la même instance de service *ConfInfoProvServ*, donc il est pertinent d'améliorer la performance, afin de satisfaire ce besoin, en déployant de nouvelles instances de ce dernière service. Dans ce cas, la règle de duplication doit être appliquée. En suivant la description introduite dans la section 4.5, l'application de la règle de duplication ($R_1 R_2$) concernant cet exemple est définie comme suit :

Pour toutes les instances de service Web définies par les motifs X, Y et Z, appliquer les règles R1 (fig. 5.12) et R2 (fig. 5.13) définies ci-dessous.

Le tableau 5.1 résume la séquence des actions élémentaires nécessaires pour la règle

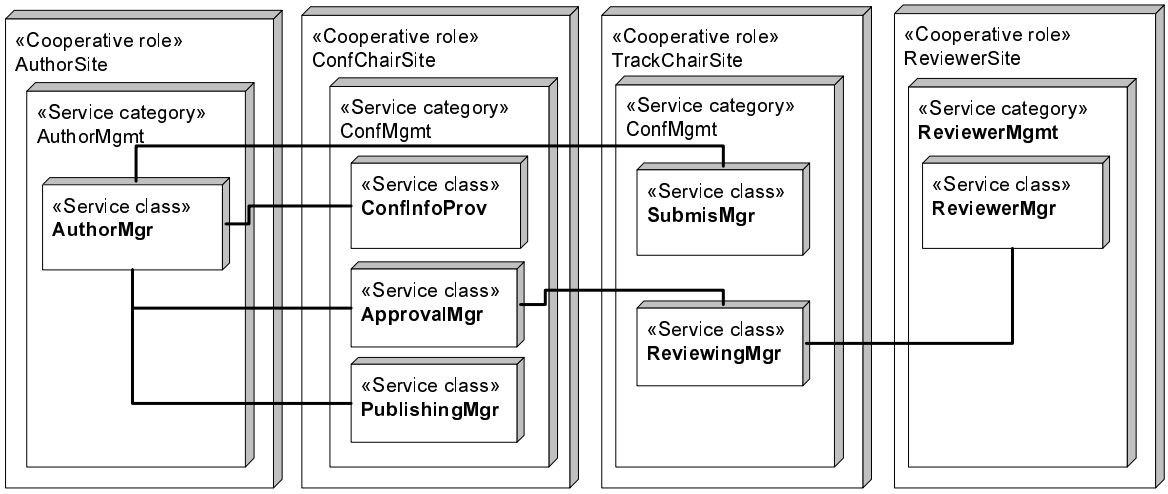


FIG. 5.10 – Architecture de déploiement du système de gestion de conférences.

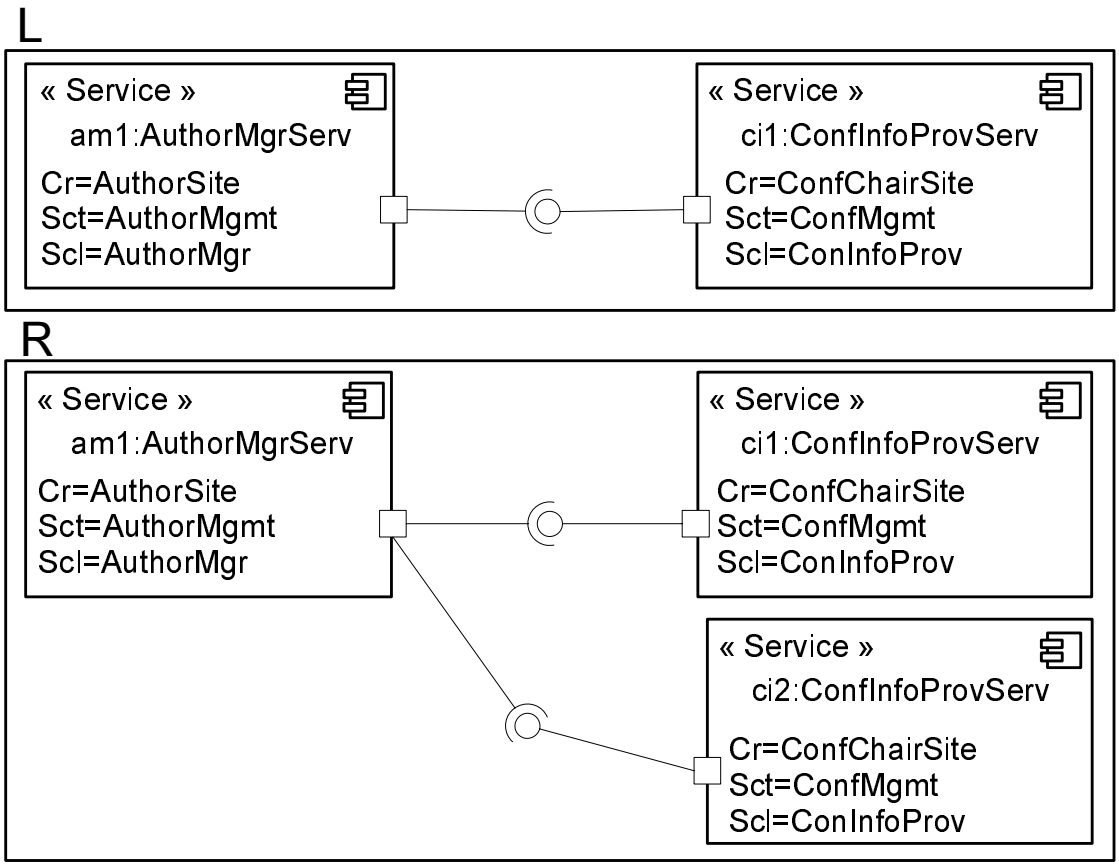


FIG. 5.11 – Application de la règle de duplication de services Web.

```
Rule [name=R1](i,j,k,l,i',j',k',l')
 Match Vertices:
   X with parameters i=ConfChairSite,j=ConfMgmt,k=ConfInfoProv,l=*;
 Add Vertices:
   Y with parameters i'=ConfChairSite,j'=ConfMgmt,k'=ConfInfoProv,l'=new;
```

FIG. 5.12 – Règle R1 appliquée à l'action de duplication de services Web.

```
Rule [name=R2](i,j,k,l,i',l',i'',j'',k'')
 Match Vertices:
   X with parameters i=ConfChairSite,j=ConfMgmt,k=ConfInfoProv,l!=new;
   Y with parameters i'=ConfChairSite,j'=ConfMgmt,k'=ConfInfoProv,l'=new;
   Z with parameters i''=*,j''=*,k''=*;
 Match Edges:
   X->Z, Z->X
 Add Edges:
   Y->Z, Z->Y
 Restriction Edges:
   Y->Z, Z->Y
```

FIG. 5.13 – Règle R2 appliquée à l'action de duplication de services Web.

de duplication.

Action de Reconfiguration	Actions élémentaires
Duplication	add_Service(ci2) bind_Service(am1,ci2)

TAB. 5.1 – Actions élémentaires exécutées pendant l'application de la règle de duplication.

Règle de substitution

Un deuxième exemple considère un cas d'application de la règle de reconfiguration nommée substitution. Des cas de dysfonctionnement récurrents telles que : un service de mauvaise qualité, un débit faible ou le déni de service ; pourraient entraîner la décision d'appliquer ce genre de reconfiguration. Comme illustré dans la figure 5.14, à un moment donné il est possible qu'un groupe de services Web reliés par la même *Catégorie de service* tombe dans un état de dysfonctionnement, donc à ce moment il est pertinent de lancer une action de substitution sur tous les services Web impliqués par cette catégorie, visant à ramener le système dans un état normal d'exécution. Pour cela, il faut également prendre en compte les connections préétablies avant d'appliquer l'action de reconfiguration, afin de les préserver avec les nouvelles instances des services créés par cette action. Ceci est le cas, par exemple, en appliquant la règle de substitution sur les services *AuthorMgrServ* et *ConfInfoProvServ*. De la même manière, plusieurs autres cas pourraient être illustrés. En suivant la description introduite dans la section 4.5, l'application de la règle de substitution ($R_1R_2R_3$) concernant cet exemple est définie comme suit :

Pour toutes les instances de services Web définies par X, Y et Z, appliquer les règles R1 (fig. 5.15),R2 (fig. 5.16) et R3 (fig. 5.17) définies ci-dessous.

Le tableau 5.2 résume la séquence des actions élémentaires associées à la règle de substitution.

5.2.5 Mise en œuvre du système de gestion de conférences

Le système de gestion de conférences a été implémenté en utilisant la technologie des services Web. Par la suite, nous présentons les éléments principaux de sa mise en œuvre.

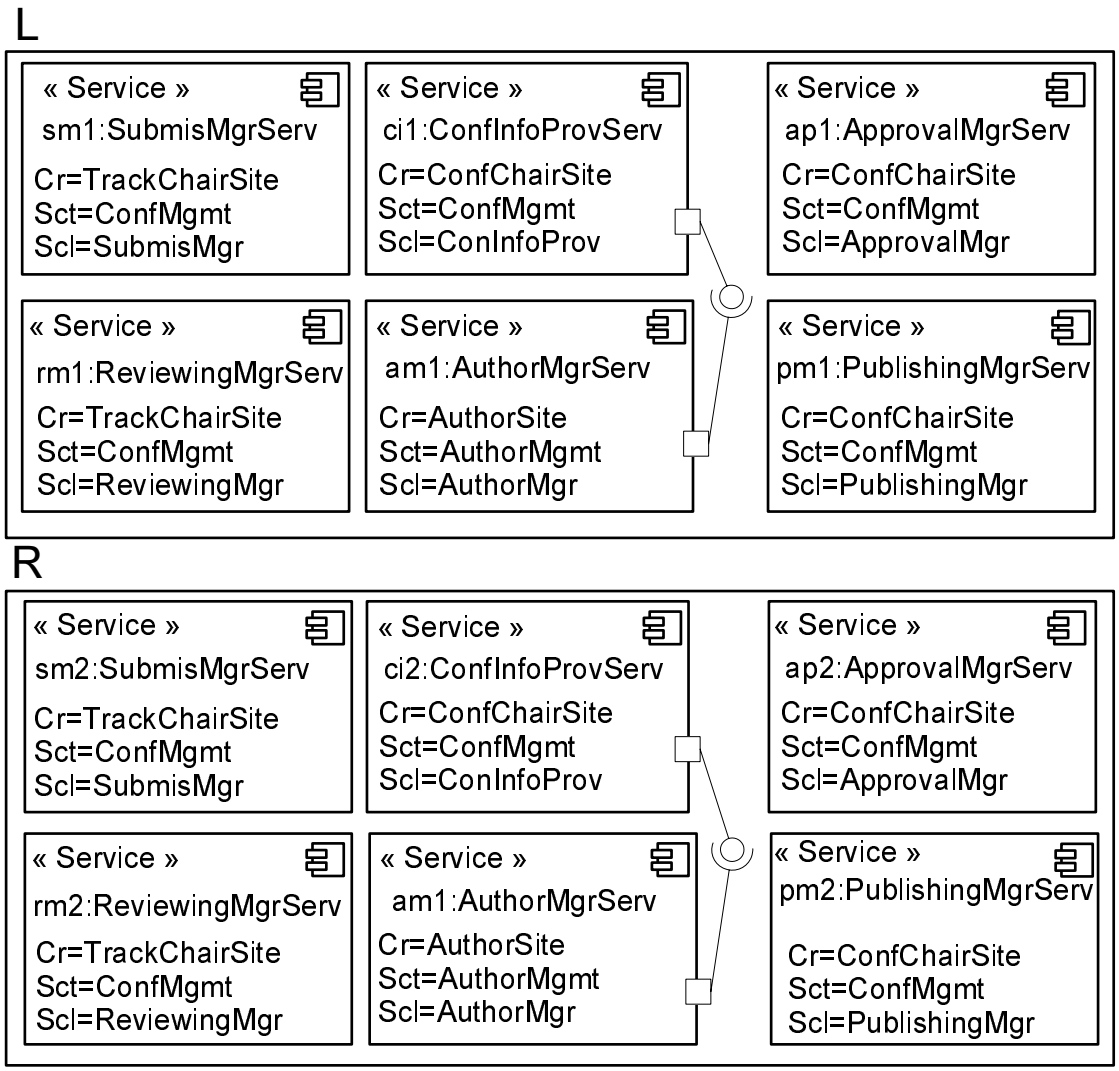


FIG. 5.14 – Application de la règle de substitution de services Web par le biais de la *Catégorie de service*.

```
Rule [name=R1](j,j',l')
 Match Vertices:
   X with parameters j=ConfMgmt;
 Add Vertices:
   Y with parameters j'=ConfMgmt,l'=new;
```

FIG. 5.15 – Règle R1 appliquée à l'action de substitution de services Web.

```
Rule [name=R2](j,l,j',l',i'',j'',k'')
 Match Vertices:
   X with parameters j=ConfMgmt,l!=new;
   Y with parameters j'=ConfMgmt,l'=new;
   Z with parameters i''=*,j''=*,k=''=*;
  Match Edges:
   X->Z, Z->X
 Add Edges:
   Y->Z, Z->Y
 Restriction Edges:
   Y->Z, Z->Y
```

FIG. 5.16 – Règle R2 appliquée à l'action de substitution de services Web.

```
Rule [name=R3](j,l)
 Delete Vertices:
   X with parameters j=ConfMgmt,l!=new;
```

FIG. 5.17 – Règle R3 appliquée à l'action de substitution de services Web.

Action de reconfiguration	**Actions élémentaires**
Substitution	add_Service(sm2)
	add_Service(ci2)
	add_Service(ap2)
	add_Service(rm2)
	add_Service(pm2)
	bind_Service(ci2,am1)
	unbind_Service(ci1,am1)
	deactivate_Service(sm1)
	deactivate_Service(ci1)
	deactivate_Service(ap1)
	deactivate_Service(rm1)
	deactivate_Service(pm1)

TAB. 5.2 – Actions élémentaires exécutées pendant l'application de la règle de substitution.

Environnement Logiciel

Pour la réalisation de ce système, sous la plate-forme J2EE (Java 2 Entreprise Edition) comme environnement de développement, les outils suivants ont été utilisés.

Le langage de programmation JAVA a été choisi pour décrire les différents services Web formant cette application. Outre ses avantages de portabilité, de simplicité, d'être orienté objet et dynamique, Java nous permet de développer des applications Web grâce à la technologie JSP et Servlet.

Eclipse a été utilisé tant qu'environnement de développement. Eclipse est un environnement de développement intégré (ang. IDE : Integrated Development Environment) dont le but est de fournir une plate-forme modulaire pour permettre de réaliser toute sorte de développements informatiques. Nous avons choisi Eclipse, d'une part parce qu'il utilise Java comme langage de programmation, d'autre part vu la pertinence de cet outil pour le développement des applications utilisant la technologie des services Web et enfin pour l'extensibilité qu'offre cet outil grâce aux plug-ins.

Apache Tomcat a été utilisé en tant que serveur d'applications. Tomcat est un serveur d'applications Java Open source qui fonctionne sur plusieurs plates-formes (Linux, Windows, MAcOs...). Ce serveur nous permet d'héberger le site Web de l'application. Il nous offre un environnement pour l'exécution des services Web (grâce à Axis).

On a travaillé avec Apache Axis en tant que conteneur de déploiement. Axis est un ensemble d'APIs permettant le déploiement de services Web. Axis nous permet de déployer des services Web à travers Eclipse et permet de générer automatiquement leurs fichiers de description.

Concernant la gestion de la base de données, le système de gestion de bases de données Open Source MySQL a été choisi pour stocker toutes les données relatives au système de gestion de conférences à savoir les informations personnelles de tous les utilisateurs du système (PCP, PT, auteur) et les informations relatives à chaque conférence. Ce choix se justifie par son adaptabilité avec les applications Web et sa capacité de supporter une masse importante d'enregistrements.

Enfin, les protocoles de transfert de fichiers HTTP et le protocole de messagerie SMTP sont utilisés pour le routage des interactions.

Architecture de mise en œuvre

L'architecture globale de mise en œuvre est illustrée par la figure 5.18. Cette architecture qui repose sur l'utilisation des services Web a pour objectif d'assurer une flexibilité d'échange de services entre les différents composants du système de gestion de conférences. Le choix des services Web participants respecte la spécification du système de gestion de conférences, décrit précédemment, qui donne l'ensemble des services à considérer conforme à une architecture orientée service. Ces services Web servent à exécuter des tâches variées pour supporter l'ensemble des fonctionnalités de l'application et ils peuvent être classés en trois parties :

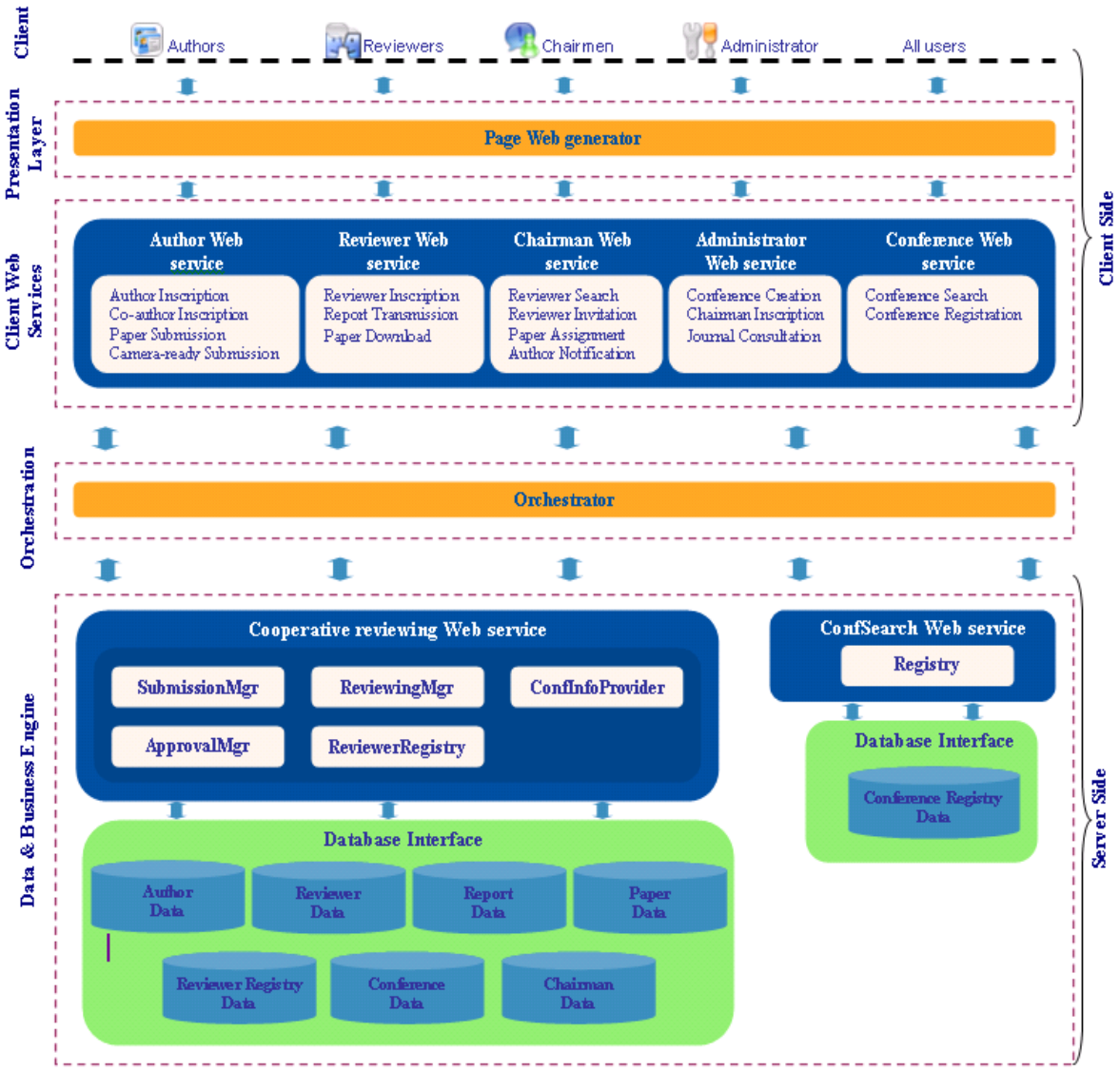


FIG. 5.18 – Architecture de mise en œuvre du système de gestion de conférences.

1. Partie cliente (Client Side)
   - Couche de présentation : C'est l'interface homme machine qui assure l'incorporation entre les clients. Elle contient les services Web clients suivants qui se distinguent par l'acteur qui les utilise :
     - Service Web *Author* : Il est interrogé sur chaque traitement spécifique à l'auteur à savoir son inscription, la soumission de ses papiers, la soumission finale, etc.
     - Service Web *Reviewer* : Il est invoqué par le relecteur pour répondre à toutes ses demandes à savoir son inscription, la transmission de ses rapports et le téléchargement des papiers qui lui sont assignés.
     - Service Web *Chairman* : Il est interrogé si les services demandés concerne le président de la conférence. Ces services sont la recherche et l'invitation des relecteurs, l'affectation des papiers soumis et la notification des auteurs par la décision prise concernant leurs papiers.
     - Service Web *Administrator* : Il est spécifique à l'administrateur et lui fournit tous les services nécessaires pour recouvrir la tâche d'administration à savoir la création de nouvelles conférences, l'attribution des rôles et le suivi des actions des utilisateurs pour chacune des conférences.
     - Service Web *Conference* : Il peut être invoqué par tout utilisateur de l'application voulant procéder à l'enregistrement pour assister à une conférence choisie ou la recherche d'une conférence.
2. Partie orchestration. Pour faciliter la communication entre les services Web de l'application, nous avons choisi de mettre en place un service Web centralisé qui joue le rôle d'un orchestrateur (ang. Orchestrator). Sa tâche se restreint à la redirection des requêtes envoyées par les différentes entités vers leurs destinations ce qui permet de bien organiser le trafic d'échange de services entre eux. Il agit donc comme un chef d'orchestre pour les services Web sur lesquels se base notre système de gestion de conférences en assurant une flexibilité extrême.
3. Partie serveur (Server Side). Cette partie contient deux principaux services Web : *Cooprerative reviewing* et *ConfSearch*.
   - Service Web *Cooperative reviewing* : Il englobe les services Web suivants :
     - *ConfInfoProvider* : Il assure la création de nouvelles conférences ainsi que l'affectation des présidents pour chacune d'elles.
     - *SubmissionMgr* : Il permet la sauvegarde des données personnelles de l'auteur ainsi que ses données d'authentification, de gérer le processus de soumission des papiers et celle de la soumission finale en cas d'acceptation d'un papier.

- *AprovalMgr* : Il assure la prise de décision finale ainsi que la notification des auteurs par cette décision.
- *ReviewingMgr* : Il permet la sauvegarde des données personnelles du relecteur ainsi que ses données d'authentification. De plus, il gère l'affectation des papiers aux relecteurs adéquats. Il assure aussi la revue des papiers assignés et la transmission des rapports d'analyse.
- *ReviewerRegistry* : Il se charge de stocker les données de base des relecteurs ainsi que leur recherche suivant certains critères.

Pour assurer toutes ces fonctionnalités, ces services Web prennent la charge d'accéder à la base de données (Database interface) soit pour enregistrer toutes les données concernant les conférences, les auteurs, les relecteurs, les présidents, les rapports et les papiers lors de leurs créations soit pour répondre à une requête portant sur les informations relatives à un de ces éléments.

- service Web *ConfSearch*. Nous y recensons le service Web *Registry* permettant aux différents acteurs de lancer une recherche sur des conférences existantes selon un certain nombre de critères.

Description des interfaces utilisateur

Par la suite, nous décrivons les interfaces utilisateur les plus importantes du système de gestion de conférences.

La page d'accueil

La page d'accueil du site présente au début un aperçu général sur les différents services offerts par le système. Elle affiche aussi une liste des conférences hébergées par le site (voir figure 5.19).

A travers des liens hypertextes, l'utilisateur a le choix entre :

- La création d'une nouvelle conférence (c'est principalement la tâche de l'administrateur),
- La participation à une conférence pour chacun des utilisateurs (PCP, PT, auteur),
- La recherche d'une conférence respectant certains critères,
- L'inscription à une conférence pour payer les frais d'enregistrement.

En suivant le lien « Create your own conference », une interface d'identification s'affiche demandant le login et le mot de passe de l'administrateur. Une fois l'identification

FIG. 5.19 – Page d'accueil du système de gestion de conférences.

de l'administrateur est validée, la page d'accueil concernant l'administrateur s'affiche présentant dans un menu horizontal toutes les fonctionnalités possibles. En cliquant sur le bouton « Configuration », l'administrateur a la possibilité de configurer les différents paramètres d'installation du site sur la machine hébergente (adresse du serveur SMTP, le chemin du répertoire d'installation du serveur d'application, l'adresse du serveur, etc.).

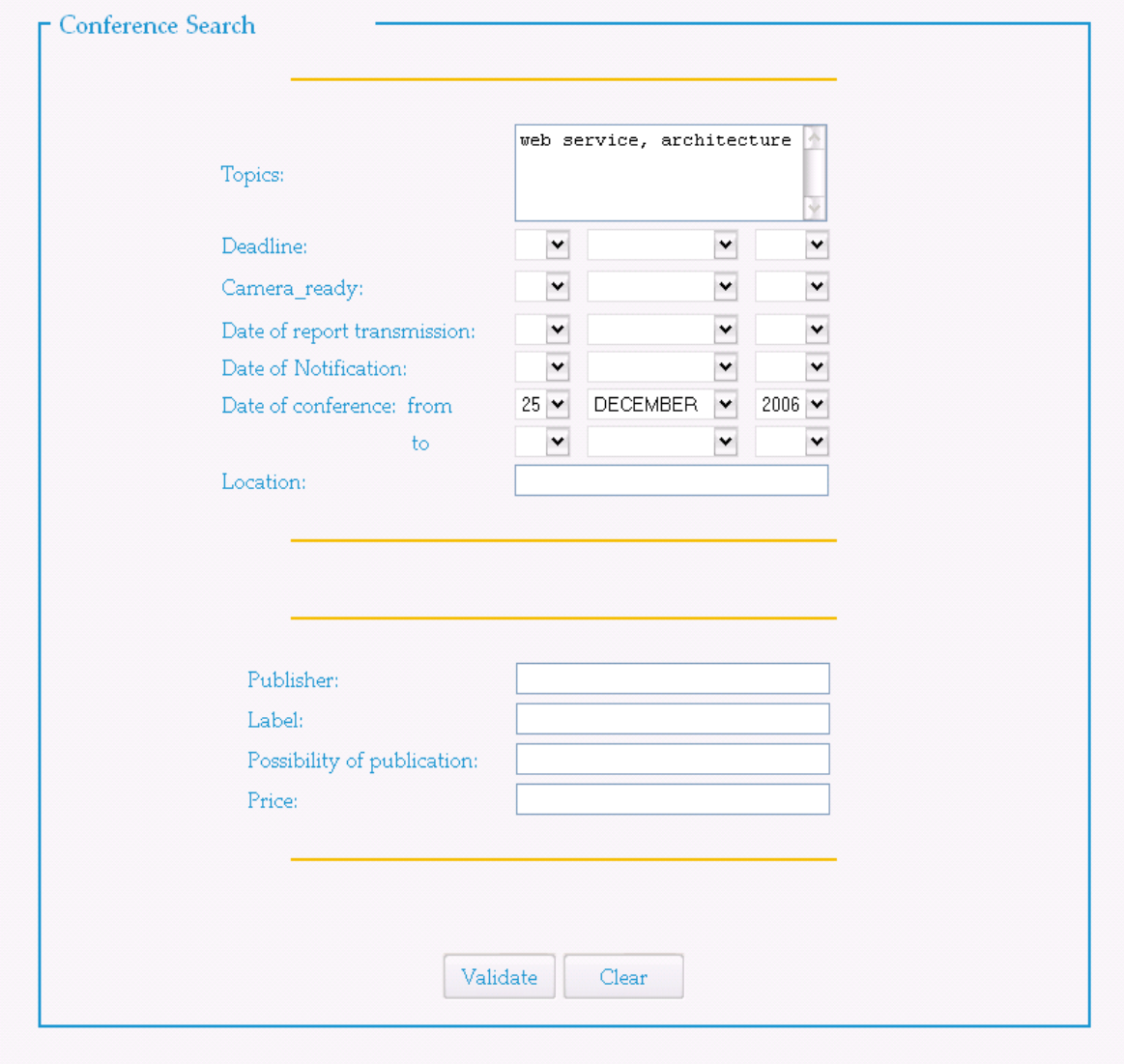


FIG. 5.20 – Page de recherche de conférences.

Recherche de conférences

En cliquant sur le lien « Search for a conference », l'interface de recherche apparaît. L'utilisateur saisit les différents critères sur lesquels porte sa recherche. Le résultat s'affiche sous forme d'une liste contenant les données relatives à chaque conférence

répondant à ces critères de recherche (Voir figure 5.20). En validant les critères de recherche, une interface contenant les titres des conférences ouvertes et répondants aux critères déjà saisis s'affiche.

Soumission d'un papier

Un auteur ne peut soumettre son papier que s'il est déjà inscrit dans la conférence. Pour cela deux choix sont possibles : Si l'auteur n'est pas inscrit, un lien lui permet de faire son inscription et s'il est déjà inscrit, l'interface d'identification apparaît demandant à l'auteur de saisir son nom d'utilisateur et son mot de passe. Après son identification, un auteur peut soumettre la version de son papier plusieurs fois avant de dépasser la date limite de soumission (deadline). Pour la première soumission, l'auteur doit sélectionner le bouton radio « New Paper » et remplir les informations relatives à son papier à savoir : le titre, les différents sujets couverts par le papier, le résumé, les mots clés et ensuite, il doit soumettre la version complète de son article (Voir figure 5.21). Seuls les formats pdf, doc, ps, txt sont acceptés. Autrement, un message d'erreur apparaît.

L'auteur qui s'est identifié et qui a rempli toutes les informations précédemment mentionnées, est considéré comme l'auteur principal de l'article. Si le papier est rédigé par plusieurs auteurs (co-auteurs), l'auteur principal est invité à les inscrire en cliquant sur le bouton « Other authors »et à saisir leurs différentes informations. Ces co-auteurs sont ajoutés à la base de données et le système attribue à chacun un nom d'utilisateur et un mot de passe qu'il envoie par courrier électronique. Si l'une des données relatives à l'article n'est pas fournie, un message d'erreur apparaît demandant à l'auteur de remplir le champ manquant et de refaire la soumission. Si la soumission du papier est validée, un code est attribué au papier soumis et automatiquement expédié à l'adresse électronique de l'auteur principal.

Affectation des papiers

L'opération d'affectation des papiers aux relecteurs se fait en cliquant sur le bouton du menu. L'affectation n'est possible que si la date de « deadline » est dépassée. Autrement, un message d'erreur sera affiché.

Une fois la date finale de soumission est dépassée, le PCP aura une interface contenant le nombre total de papiers soumis ainsi que le nombre de relecteurs qui participent à la conférence. A travers cette interface, le PCP précise les différentes contraintes d'affectation : le nombre minimum de relecteurs pour la revue d'un papier et le nombre maximum de papiers affectés à un relecteur (Voir figure 5.22).

FIG. 5.21 – Page de soumission d'articles.

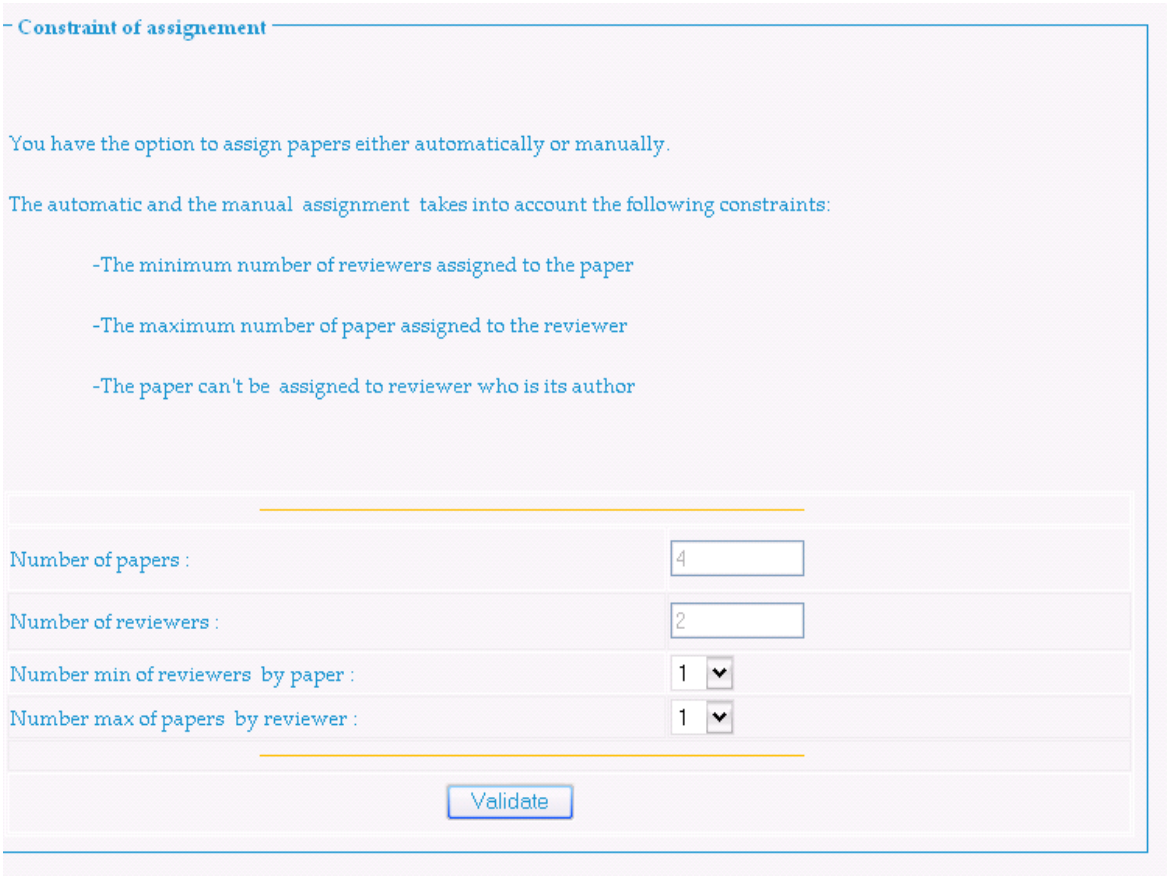


FIG. 5.22 – Phase d'affectation des articles aux relecteurs.

En validant les contraintes d'affectation le PCP aura le choix entre l'affectation automatique ou l'affectation manuelle des papiers. Pour une affectation automatique, le PCP suit le lien « Assign paper ». Ainsi, l'affectation des papiers aux relecteurs se fait automatiquement en respectant les contraintes déjà définies et des courriers électroniques seront envoyés aux relecteurs pour leur informer des papiers à télécharger. L'affectation d'un papier à un relecteur n'est valide que si ce dernier n'est ni un auteur ni un co-auteur du papier. Le PCP aura la possibilité de modifier la liste des relecteurs qui ont été affectés à un papier.

Transmission d'un rapport

Pour chaque papier, un formulaire d'évaluation apparaît permettant au relecteur de saisir ses commentaires et de donner son jugement concernant ce papier (Voir figure 5.23).

A la suite de cette démarche, deux types de rapports d'analyse seront envoyés au site de la conférence. Le premier est destiné au président et le deuxième est destiné à l'auteur (Voir figure 5.24). Le relecteur n'a pas le droit de modifier un rapport déjà envoyé.

5.3 L'exemple « Foodshopping »

L'exemple du « Foodshopping » considère une société qui vend et délivre des produits d'alimentation. Pour placer ses produits, cette société dispose d'un magasin (ang. Shop) en ligne et de plusieurs entrepôts (ang. Warehouses) localisés dans des régions différentes. Les entrepôts stockent de produits non-périssables et sont en charge de la délivrance aux clients en fonction de la proximité aux domiciles des clients.

Les clients (ang. Customers) interagissent avec le magasin afin de placer des commandes de produits, acquitter la facture et recevoir les produits. Dans le cas de produits périssables, ou de produits indisponibles en stock, la société doit contacter plusieurs fournisseurs (ang. Suppliers).

Bien que la plupart des interactions citées par ce cas d'étude sont électroniques, et effectuées par de services Web, dans certains cas il peut avoir des interactions impliquant des acteurs humains (par exemple, la livraison des produits). Néanmoins, afin de garder un esprit d'automatisme nous nous limitons à traiter uniquement les interactions électroniques.

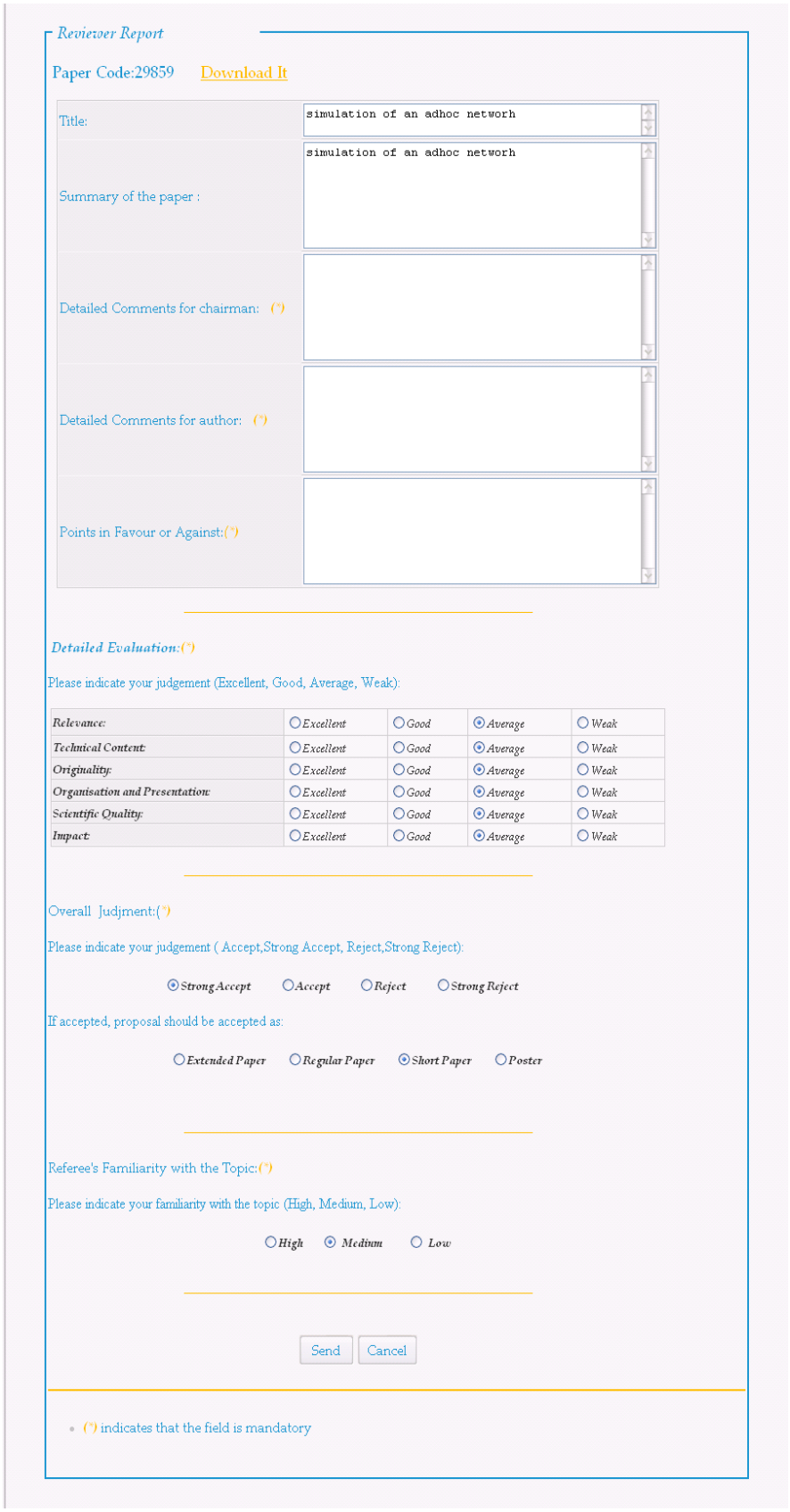


FIG. 5.23 – Rapport d'évaluation des articles.

FIG. 5.24 – Transmission du rapport d'évaluation.

5.3.1 Acteurs et workflow

Dans chaque conversation les acteurs suivants sont impliqués :

– un client,

– un magasin,

– un entrepôt,

– un nombre variable de fournisseurs, qui pourrait être aussi 0.

Lorsqu'un client fait une commande, le magasin choisi d'abord l'entrepôt plus proche à l'adresse du client qui sera impliqué dans la conversation. Les produits commandés sont considérés de deux types :

1. Produit périssable, c'est-à-dire il ne peut pas être stocké, donc il n'est disponible qu'auprès des fournisseurs. Ces produits sont gérés directement par le magasin.
2. Produit non-périssable, ils sont disponibles auprès des entrepôts et sont gérés par eux-mêmes.

La premier démarche à faire est de vérifier la disponibilité des produits commandés, auprès d'entrepôts pour les produits non-périssables et de fournisseurs pour les produits périssables. Par un souci de simplicité, on ne considère pas les échanges de produits entre différents entrepôts. Dans le cas de disponibilité, les produits sont temporairement réservés afin d'éviter les conflits entre plusieurs ordres.

Une fois que le magasin reçoit toutes les réponses sur la disponibilité des produits, il peut décider de continuer avec la commande, ou bien l'annuler. Une commande est annulée chaque fois qu'au moins un produit est indisponible. Dans le dernier cas, toutes les réservations des produits sont annulées et la conversation est terminée.

Si la commande continue, le magasin calcule le coût total (produits + livraison) en ensemble avec l'entrepôt qui détermine le coût de livraison. Ensuite, le magasin envoie la facture au client, qui peut décider de payer ou pas. Si le client ne paye pas, toutes le réservations de produits sont annulées et la conversation est terminée. Si le client paye, tout les produits réservés sont confirmés et les fournisseurs sont sollicités à envoyer les produits à l'entrepôt. L'entrepôt ressemblera tout les produits et les enverra au client.

Par la suite, pour chaque acteur nous décrivons son workflow et ses relations avec les autres acteurs dans l'ensemble du workflow. Chaque workflow est représenté par un diagramme d'activités. Dans la notation un trait continu indique une interaction électronique alors qu'un trait en pointillé indique une interaction physique.

Workflow côté Client

Nous représentons le workflow du client (figure 5.25) de manière abstraite, c'est-à-dire nous ne représentons pas son comportement interne mais seulement son interface avec les autres acteurs. La raison est que le client est considéré comme un composant externe, et donc nous n'avons pas les détails de son comportement interne.

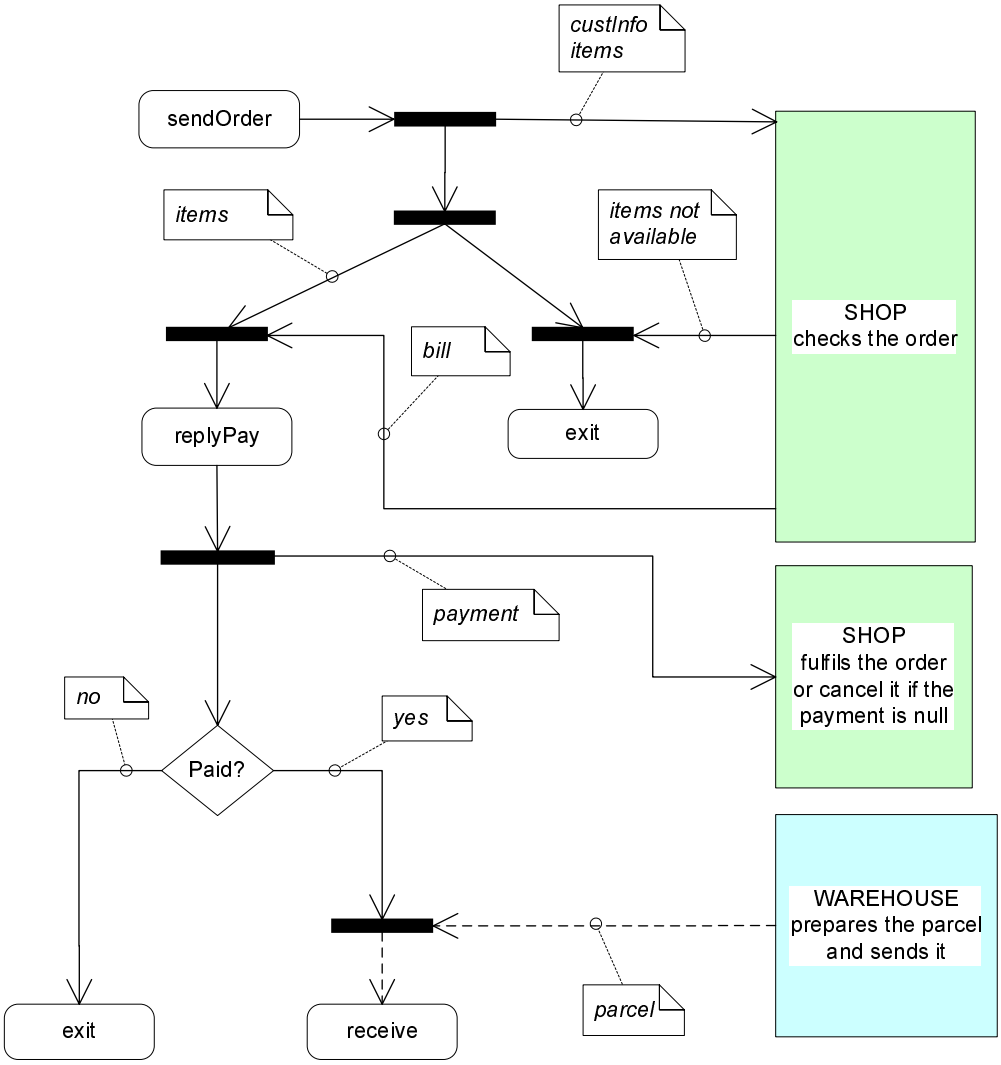


FIG. 5.25 – Workflow du client dans l'application « Foodshopping ».

Le client commence son workflow en faisant une commande (sendOrder). La commande décrit les produits (items) qu'intéressent au client et ses coordonnées personnels (custInfo). Ensuite, le client attend une réponse du magasin ; si un ou plusieurs produits ne sont pas disponibles la conversation termine (exit). Dans le cas contraire, le client reçoit la facture (bill) et décide payer (replyPay) en envoyant le paie (payment) au magasin. Si le client décide de ne pas payer, la conversation termine (exit). Si le client a

payé, il attend le colis (parcel) avec les produits commandés depuis l'un des entrepôts. Dans ce workflow tous les interactions sont électroniques sauf la livraison des produits par l'entrepôt.

Workflow côté Magasin

Le workflow du magasin (figure 5.26) implique plusieurs activités internes. Lorsque le magasin reçoit une commande (receiveOrder) décrivant les produits (items) commandés et les coordonnées du client (custInfo), il choisit l'entrepôt le plus proche au client (selectWH) et divise en deux groupes la commande (splitOrder) : un pour les produits périssables (ns_items) et un autre pour les produits non-périssables (s_items). Ensuite, le magasin vérifie la disponibilité des produits périssables (checkAvail&reserve) auprès des fournisseurs, en demandant une réservation provisoire des produits disponibles. Le magasin reçoit en retour la liste de produits réservés (ns_resitems), les codes de réservation correspondants (ns_rescodes) et la réponse des disponibilités(ns_answers).

La liste de produits non-périssables (s_items) est envoyée à l'entrepôt choisi (checkAvail), qui envoi en retour une réponse collective (s_answers) sur la disponibilités des produits.

Si l'un des produits commandés n'est pas disponible, la commande est annulée. Le magasin communique cette décision au client et annule les réservations des produits (unreserved) auprès des fournisseurs et de l'entrepôt.

Par contre, si tous les produits de la commande sont disponibles, le magasin demande à l'entrepôt de calculer le coût de livraison (shipCost), qui dépend de la distance entre l'entrepôt et l'adresse du client, ainsi que du poids total des produits commandés (c'est pourquoi le magasin envoi à l'entrepôt la liste de produits (items) ainsi que les coordonnées du client (custInfo).

Le magasin calcule le coût total (totalCost) et envoi la facture au client, qui devrais l'acquitter (payment). Si le client décide de ne pas payer, le magasin annule toutes les réservations (unreserved) auprès des fournisseurs et de l'entrepôt. Si le client paie, le magasin renvoi la commande à l'entrepôt (fwOrder), qui à ce moment, deviens le responsable de la livraison. L'entrepôt envoi aux fournisseurs ses coordonnées (whInfo) et les codes de réservation des produits (ns_items) afin de lui renvoyer les produits réservés (requestSupply).

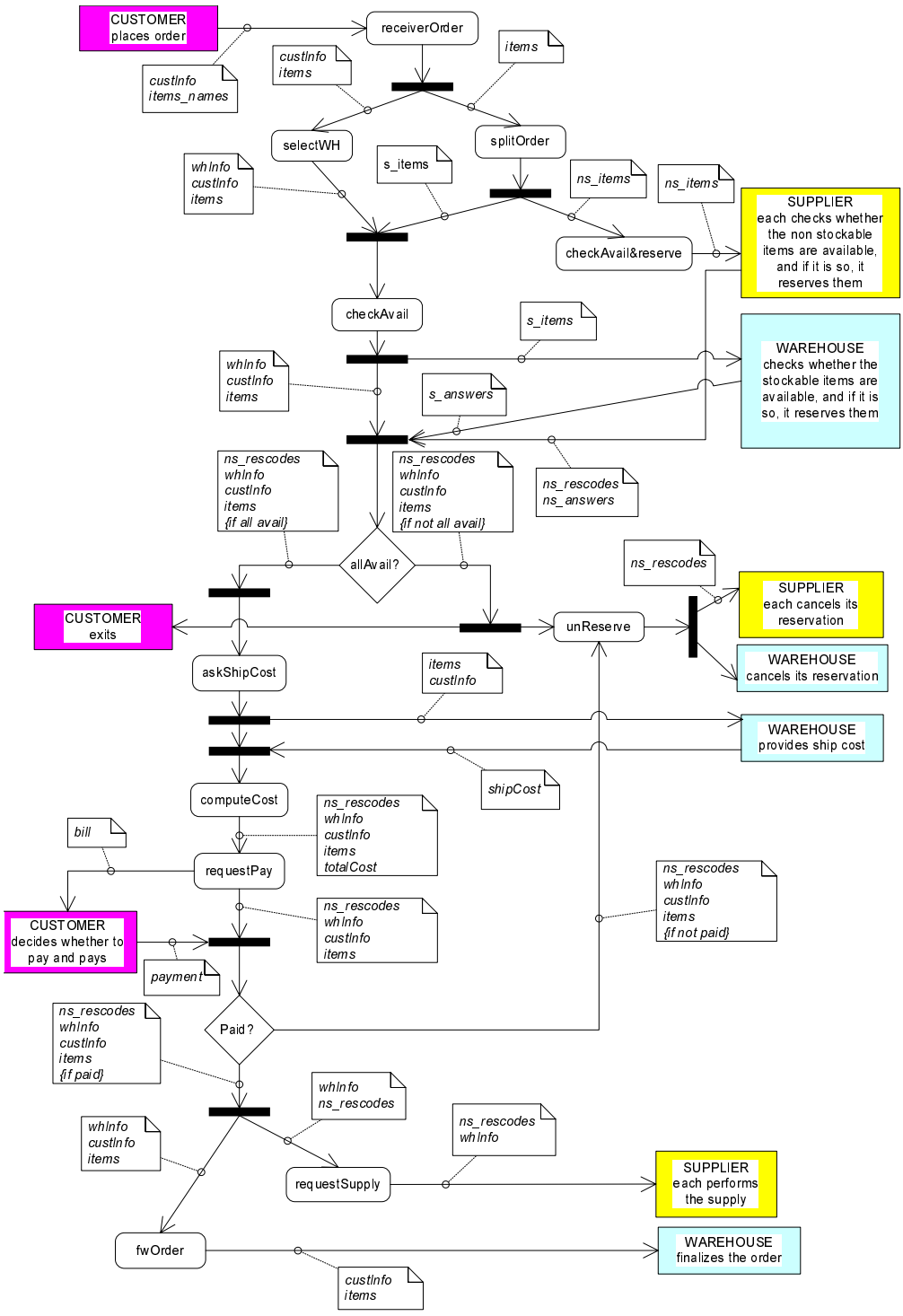


FIG. 5.26 – Workflow du magasin dans l'application « Foodshopping ».

Workflow côté Entrepôt

Le workflow de l'entrepôt commence en recevant une commande depuis le magasin, afin de vérifier la disponibilité de produits non-périssables (s_items) et de les réserver (reserveAvail). Si quelques produits ne sont pas disponibles, l'entrepôt contacte les fournisseurs afin de vérifier leur disponibilité et de les réserver (findSuppliers). En retour, l'entrepôt reçoit la liste de produits réservés (s_items), les codes de réservations correspondants (s_rescodes) et les réponses de disponibilité (s_answers).

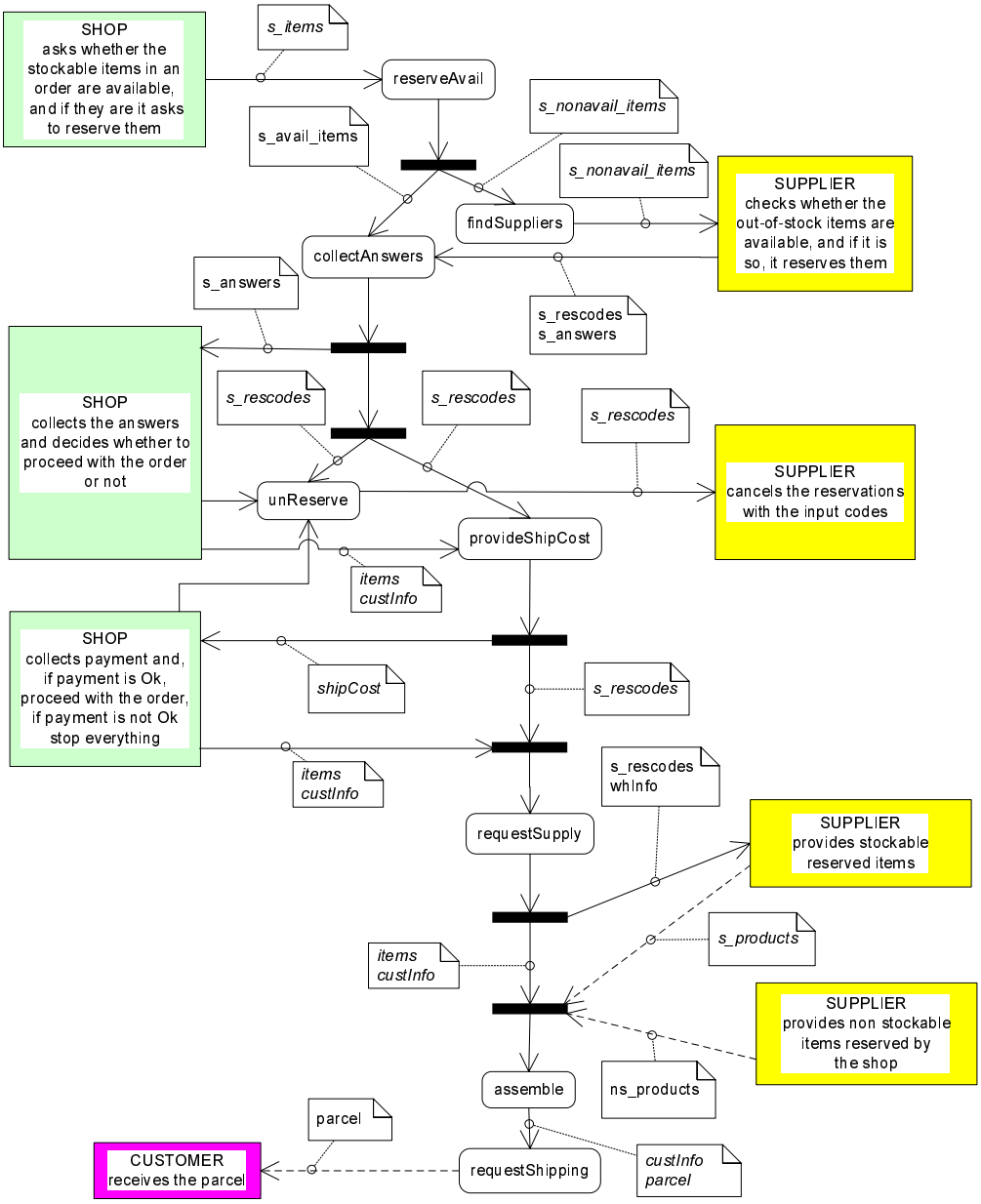


FIG. 5.27 – Workflow de l'entrepôt dans l'application « Foodshopping ».

L'entrepôt prépare une réponse collective vis-à-vis de la disponibilité des produits et l'envoi au magasin (collectAnswers). Ensuite, deux cas sont possibles : le magasin décide d'annuler la commande ou de procéder. Dans le premier cas, le magasin doit annuler toutes les réservations auprès des fournisseurs (unreserved). Dans le second cas, le magasin demande à l'entrepôt de calculer le coût de livraison (provideShipCost).

Ensuite le magasin demande à l'entrepôt de continuer avec la commande. Dans le cas de produits non-disponibles, l'entrepôt demande aux fournisseurs de lui envoyer les produits réservés (requestSupply), en lui donnant les codes de réservation (s_rescodes) et ses coordonnées (whInfo).

A ce moment, l'entrepôt doit préparer le colis avec les produits. Afin de pouvoir réaliser ceci, l'entrepôt doit attendre les items en provenance des fournisseurs, que ce soit les produits non-périssables réservés par lui même ou les produits périssables réservés par le magasin.

Une fois que le colis est prêt, l'entrepôt demande au transporteur (requestShipping) de le livrer au client.

Workflow côté Fournisseur

A l'instar du client, le workflow du fournisseur (figure 5.28) est abstrait du fait que chaque fournisseur peut avoir des comportements internes différents.

Ici on décrit un workflow générique et indépendant du service Web qui contactes le fournisseur. C'est pourquoi, le service Web qui achète des produits est nommé acheteur (buyer), alors que celui qui reçoit les produits est nommé récepteur (receiver). En ce qui concerne cet exemple de « Foodshopping », l'acheteur peut être le magasin ou l'entrepôt, tandis que le récepteur est toujours l'entrepôt.

Le workflow du fournisseur commence quand il est sollicité par l'acheteur de vérifier la disponibilité de quelques produits et de les réserver (verify&reserve). Le fournisseur envoi en retour la liste de produits réservés (resitems), les codes de réservation correspondants (rescodes) et les réponses de disponibilités (answers).

Ensuite l'acheteur peut demander l'annulation de la réservation (unReserve) ou demander au fournisseur d'envoyer les produits (supply) à l'adresse de livraison du récepteur (sendAddress).

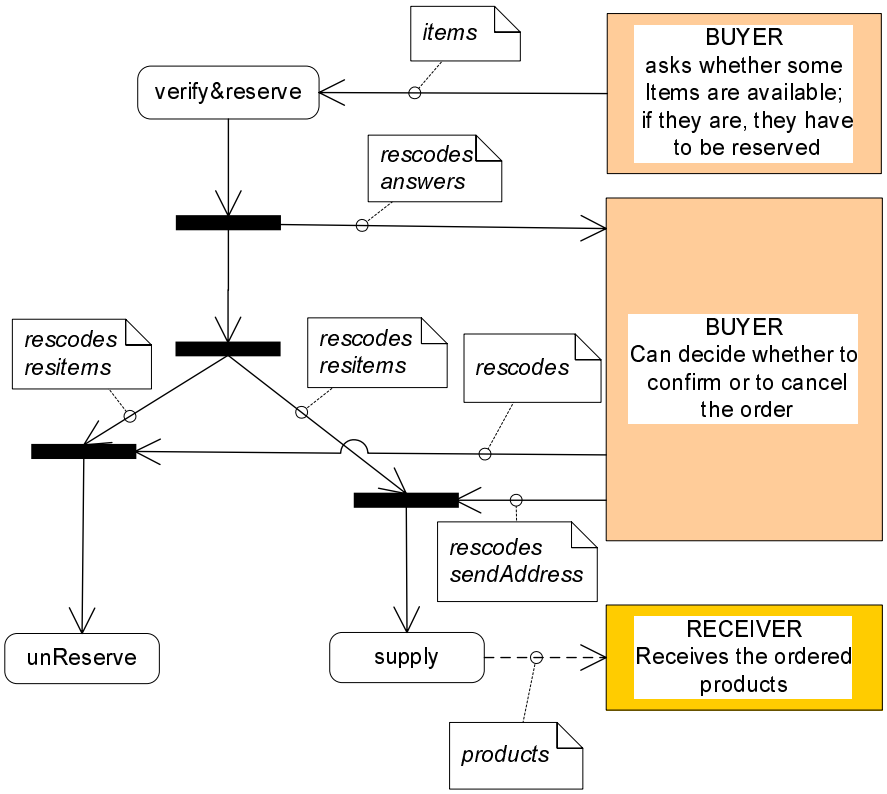


FIG. 5.28 – Workflow du fournisseur dans l'application « Foodshopping ».

5.3.2 Description architecturale

La description architecturale caractérisant cet exemple est illustrée par la figure 5.29. Cette description implique :

- Quatre Rôle coopératifs
  - *CustSite* considère les sites participant pour les clients,
  - *ShopSite* représente les sites de déploiement pour les éléments caractérisant le magasin,
  - *WHSite* représente les sites participant pour l'entrepôt,
  - *SupSite* considère les sites participant pour les fournisseurs.
- Quatre Catégories de service
  - *CustMgmt* gère les fonctionnalités générales vis-à-vis des clients,
  - *ShopMgmt* gère les fonctionnalités générales vis-à-vis du magasin,
  - *WHMgmt* considère la gestion des fonctionnalités générales vis-à-vis des entrepôts,
  - *SupMgmt* considère la gestion des fonctionnalités générales vis-à-vis des fournisseurs.
- Six Classes de service
  - *CustMgr* caractérise les fonctionnalités spécifiques aux clients,
  - *OrderMgr* caractérise les fonctionnalités spécifiques au processus de commandes,
  - *StockMgr* caractérise les fonctionnalités spécifiques à la gestion de stocks,
  - *SalesMgr* caractérise les fonctionnalités spécifiques au processus de ventes,
  - *WHMgr* caractérise les fonctionnalités spécifiques aux entrepôts,
  - *SupMgr* caractérise les fonctionnalités spécifiques aux fournisseurs.

Une configuration initiale illustrant des services Web et leurs interactions, est présentée par la figure 5.30. La fonctionnalité caractérisant le magasin est décomposée en trois services Web :

1. Le service Web *OrderMgrServ* est responsable de la gestion des commandes d'articles, pour ceci il doit établir une conversation avec le service Web *WHMgrServ* (en ce qui concerne le transport et les produits non-périssables) et avec le service Web *SupMgrServ* (en ce qui concerne les produits périssables).
2. Le service Web *StockMgrServ* est responsable de la gestion de l'indisponibilité des produits, en interaction avec le service Web *SupMgrServ*.

3. Le service Web *SalesMgrServ* gère toutes les démarches concernant les paiements et l'émission des factures. Pour ceci, ce service doit établir une conversation avec le service Web *OrderMgrServ* afin d'autoriser ou d'annuler une commande.

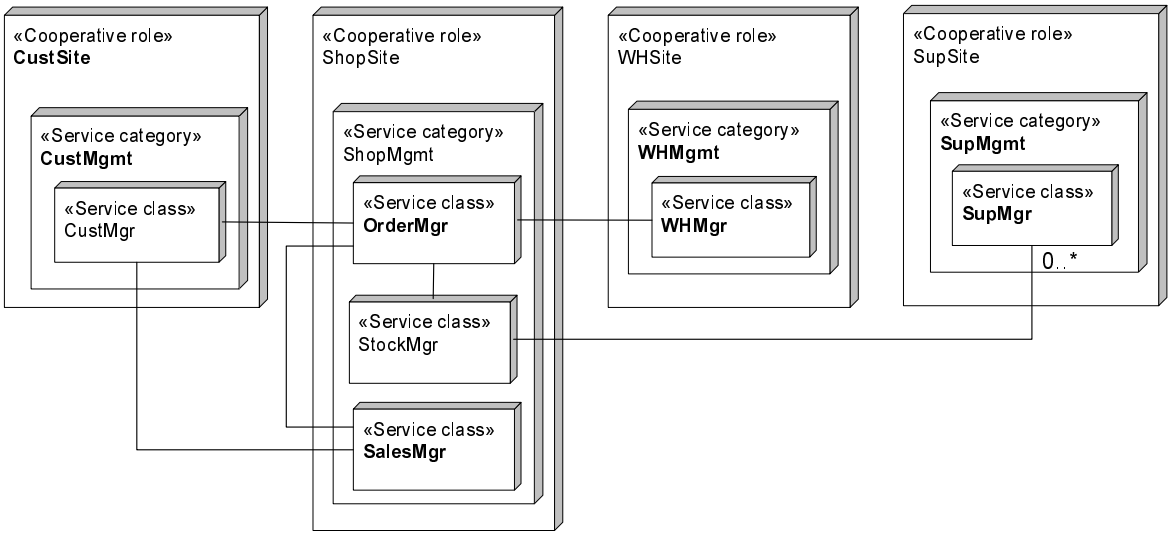


FIG. 5.29 – Architecture de déploiement de l'application « Foodshopping ».

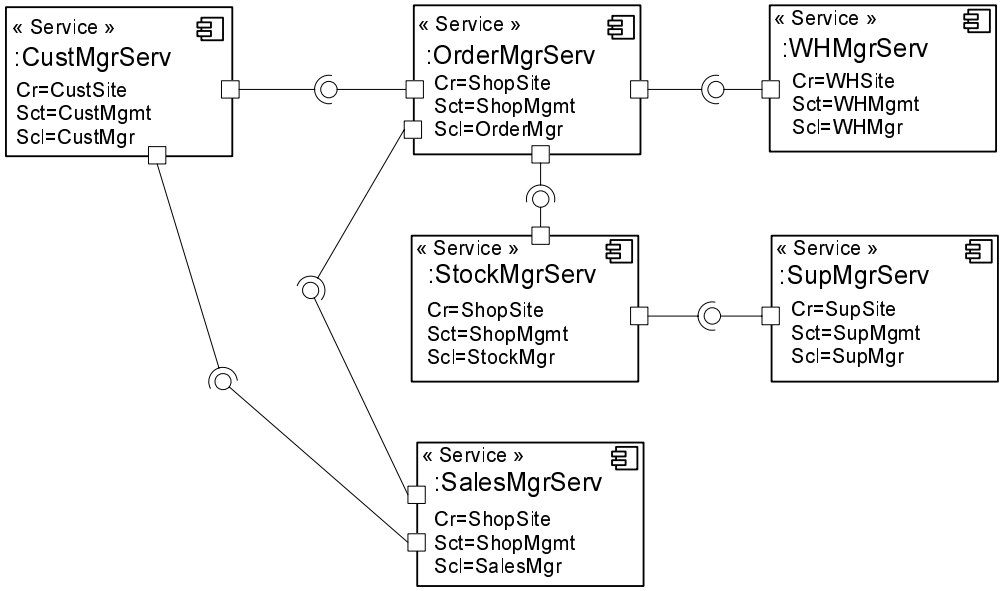


FIG. 5.30 – Interconnexion des services Web interagissant dans l'application « Foodshopping ».

5.3.3 Application des règles de reconfiguration

Règle de duplication

Un premier exemple illustrant les actions de reconfiguration pour l'application « Food-shopping » considère la duplication de services Web. En effet, plusieurs raisons pourraient être à l'origine de la décision d'appliquer cette action, notamment des symptômes associés : à la surcharge et à un faible taux de réponse. Et ceci afin d'anticiper et de prévenir la dégradation de la QdS offerte par les services Web.

Comme montré dans la figure 5.31, il s'avère que à un moment donné, suite à une sur-demande de produits, un service Web *OrderMgrServ* essayant d'adresser plusieurs instances du service Web *WHMgrServ* entraîne le déploiement de nouvelles instances de ce dernier service. En effet, c'est le maintien de la QdS qui pousse à la création de ces nouvelles instances. Dans ce cas, la règle de duplication est appliquée en adressant les instances de services Web associés au Rôle coopératif *WHSite*.

En suivant la description introduite dans la section 4.5, l'application de la règle de duplication ($R_1 R_2$) concernant cet exemple est définie comme suit :

Pour toutes les instances de service Web définies par les motifs X, Y et Z, appliquer les règles R1 (fig. 5.32) et R2 (fig. 5.33) définies ci-dessous.

Le tableau 5.3 résume la séquence des actions élémentaires nécessaires pour la règle de duplication.

Action de Reconfiguration	**Actions élémentaires**
Duplication	add_Service(wh3) add_Service(wh4) bind_Service(om1,wh3) bind_Service(om1,wh4)

TAB. 5.3 – Actions élémentaires exécutées pendant l'application de la règle de duplication.

Règle de substitution

Un second exemple décrivant des actions de reconfiguration pour l'application « Food-shopping » considère l'action de substitution. Des cas de dysfonctionnement récurrents

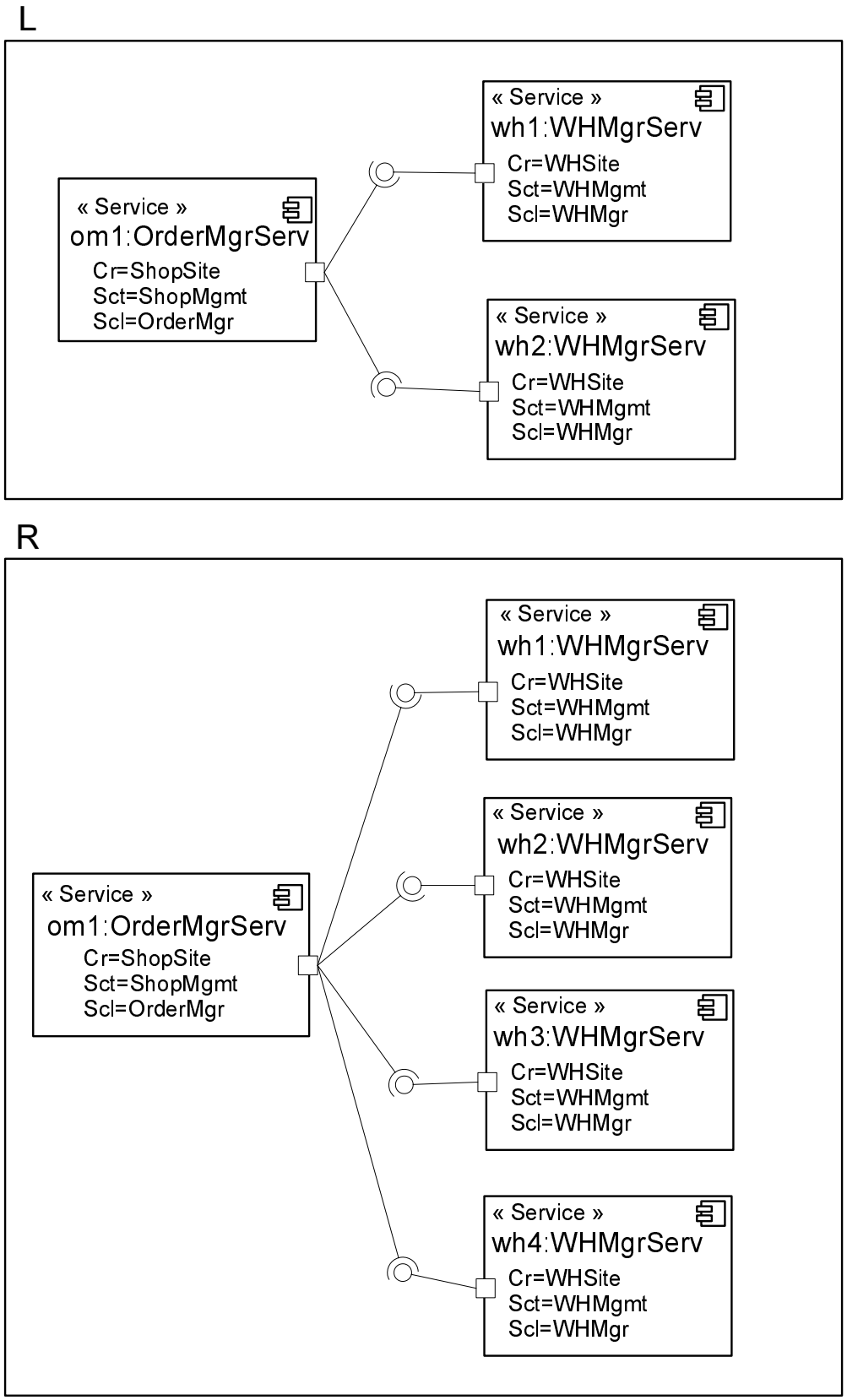


FIG. 5.31 – Application de la règle de duplication.

telles que : des services Web retournant des valeurs erronées, temps de réponse tardive ou encore le déni de service pourraient déclencher l'application de ce type d'action.

Comme illustré dans la figure 5.14 il se peut que, à un moment donné, un groupe de services Web associés à la même Catégorie de service rentre dans un état de dysfonctionnement. Ainsi, une manière efficace de palier à cette situation considère la substitution de tous le services Web associés à cette Catégorie de service (*ShopMgmt*).

On doit également tenir compte des interactions en cours menés par les services Web à remplacer, afin de les rétablir avec les nouvelles instances de services Web déployées. Dans cet exemple, c'est le cas entre les couples de services Web suivants :

- *CustMgrServ* et *OrderMgrServ*,
- *OrderMgrServ* et *WHMgrServ*, et
- *OrderMgrServ* et *StockMgrServ*.

Dans ce scénario, la règle de substitution est appliquée en adressant le Rôle coopératif *ShopSite* et la Catégorie de service *ShopMgmt*. Plusieurs autres scénarios pourraient être illustrés en affectant des valeurs différentes aux variables associées à chaque type architectural.

En suivant la description introduite dans la section 4.5, l'application de la règle de substitution ($R_1 R_2 R_3$) concernant ce scénario particulier est définie comme suit.

```
Rule [name=R1](i,i',l')
 Match Vertices:
   X with parameters i=WHSite;
 Add Vertices:
   Y with parameters i'=WHSite,l'=new;
```

FIG. 5.32 – Règle R1 appliquée à l'action de duplication de services Web.

```
Rule [name=R2](i,l,i',l',i'',j'',k'')
 Match Vertices:
   X with parameters i=WHSite,l!=new;
   Y with parameters i'=WHSite,l'=new;
   Z with parameters i''=*,j''=*,k''=*;
 Match Edges:
   X->Z, Z->X
 Add Edges:
   Y->Z, Z->Y
 Restriction Edges:
   Y->Z, Z->Y
```

FIG. 5.33 – Règle R2 appliquée à l'action de duplication de services Web.

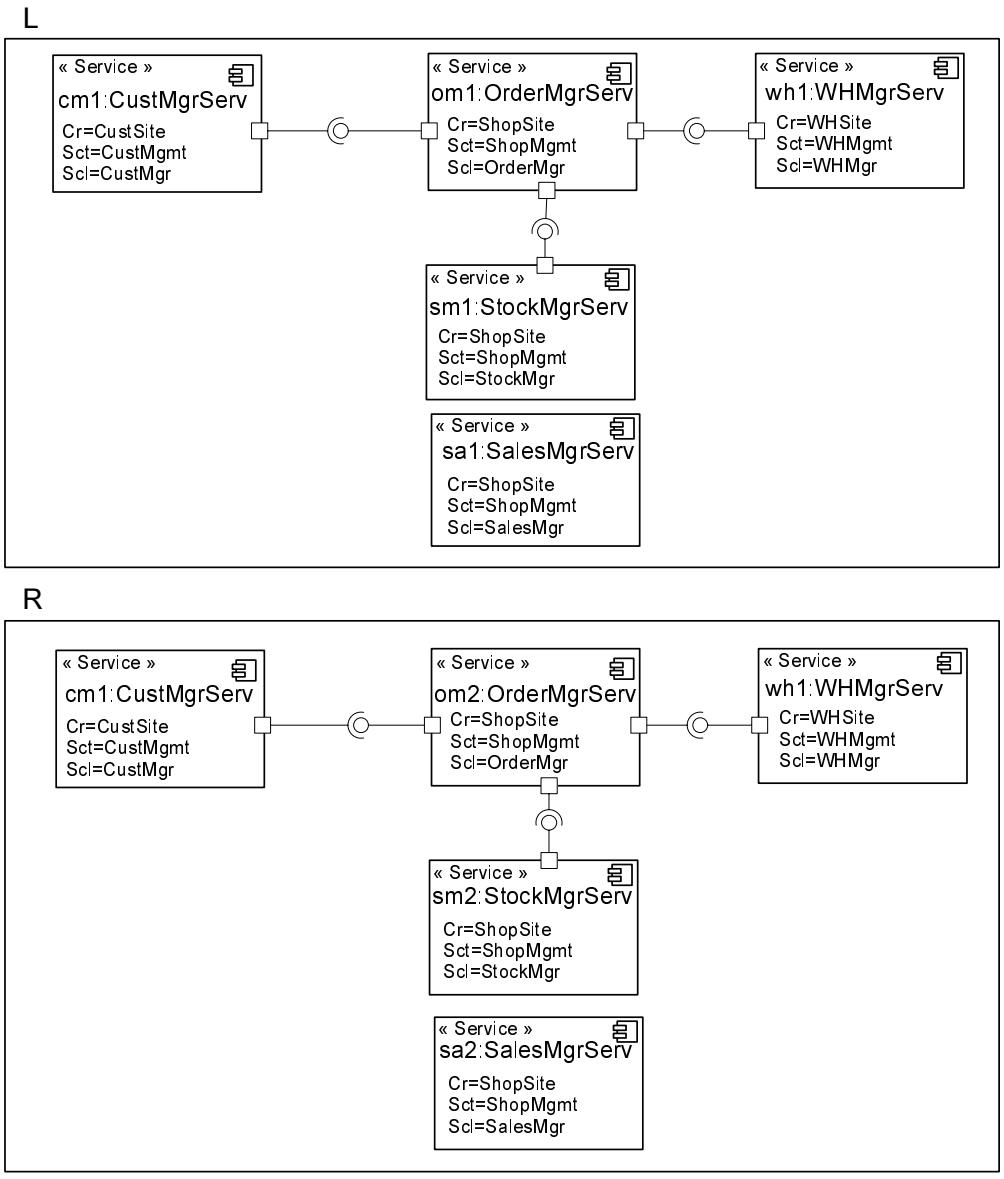


FIG. 5.34 – Application de la règle de substitution.

Pour toutes les instances de services Web définies par X, Y et Z, appliquer les règles R1 (fig. 5.35), R2 (fig. 5.36) et R3 (fig. 5.37) définies ci-dessous.

```
Rule [name=R1](i,j,i',j',l')
 Match Vertices:
   X with parameters i=ShopSite,j=ShopMgmt;
 Add Vertices:
   Y with parameters i'=ShopSite,j'=ShopMgmt,l'= new;
```

FIG. 5.35 – Règle R1 appliquée à l'action de substitution de services Web.

```
Rule [name=R2](i,j,i',j',l',i'',j'',k'')
 Match Vertices:
   X with parameters i=ShopSite,j=ShopMgmt;
   Y with parameters i'=ShopSite,j'=ShopMgmt,l'= new;
   Z with parameters i''=*,j''=*,k=''=*;
  Match Edges:
   X->Z, Z->X
 Add Edges:
   Y->Z, Z->Y
 Restriction Edges:
   Y->Z, Z->Y
```

FIG. 5.36 – Règle R2 appliquée à l'action de substitution de services Web.

```
Rule [name=R3](i,j,l)
 Delete Vertices:
   X with parameters i=ShopSite,j=ShopMgmt,l!=new;
```

FIG. 5.37 – Règle R3 appliquée à l'action de substitution de services Web.

Le tableau 5.4 résume la séquence des actions élémentaires associées à la règle de substitution.

5.4 Conclusion

Les deux scénarios présentés dans ce chapitre éprouvent l'approche présentée dans le chapitre précédent. Un point à souligner et le fait qu'ils traitent deux niveaux différents de spécification mais que s'avèrent complémentaires. Tandis que le scénario du système de gestion de conférences adopte une approche coopérative, en se focalisant sur les interactions externes des services Web, l'exemple du « Foodshopping » adopte une approche comportementale, en se focalisant sur la description du comportement interne des services Web. Les deux scénarios ont été développés dans le cadre du projet européen

WS-DIAMOND et ils font partie du Deliverable 1.1. L'exemple du « Foodshopping », en particulier la partie correspondante au workflow, a été développé en collaboration avec le Politecnico di Milano. Les sections concernant les règles de reconfiguration font partie du Deliverable 3.1 du même projet.

La mise en œuvre du système de gestion de conférence a été faite dans le cadre d'une collaboration avec l'Université de Sfax, en Tunisie.

Action de reconfiguration	**Actions élémentaires**
Substitution	add_Service(om2) add_Service(sm2) add_Service(sa2) bind_Service(cm1,om2) bind_Service(om2,sm2) bind_Service(om2,wh1) unbind_Service(cm1,om1) unbind_Service(om1,sm1) unbind_Service(om1,wh1) deactivate_Service(om1) deactivate_Service(sm1) deactivate_Service(sa1)

TAB. 5.4 – Actions élémentaires exécutées pendant l'application de la règle de substitution.

Chapitre 6

Vers une infrastructure de gestion de la QdS

6.1 Introduction

Dans ce chapitre nous présentons une ontologie pour le traitement des dysfonctionnements de la QdS. Cette ontologie est définie à partir d'une classification des dysfonctionnements qui peuvent être subis pour une application à base de services Web. En tant que politiques de gestion de la QdS nous proposons des règles qui traitent d'une part la détection des dysfonctionnements de la QdS et d'autre part la prévention (sous forme de recommandations) de ces dysfonctionnements. L'ontologie a été appliquée au système de gestion de conférences décrit dans le chapitre précédent.

De même, une architecture pour la gestion de la QdS est proposée. Cette architecture reprend des éléments propres des architectures issues du domaine du « self-healing ». L'architecture proposée a été modélisée et validée avec un outil UML.

6.2 Les ontologies et leur spécification

En informatique, le terme ontologie désigne un ensemble structuré de savoirs dans un domaine de connaissance particulier. Une ontologie décrit une sémantique associée qui fournit une signification aux données par le biais de modèles conceptuels. De même, elle permet une catégorisation au style « pages jaunes » en associant taxonomies à des

relations, des restrictions et des règles. On distingue généralement deux entités globales au sein d'une ontologie. La première, à objectif terminologique, définit la nature des éléments qui composent le domaine de l'ontologie en question, un peu comme la définition d'une classe en programmation orientée objet définit la nature des objets que l'on va manipuler par la suite. La seconde partie d'une ontologie explicite les relations entre plusieurs instances de ces classes définies dans la partie terminologique. Ainsi, au sein d'une ontologie, les concepts sont définis les uns par rapport aux autres (modèle en graphe de l'organisation des connaissances), ce qui autorise un raisonnement et une manipulation de ces connaissances.

Les premières définitions ont été données par les chercheurs du domaine de l'intelligence artificielle, « l'ontologie définit le vocabulaire et un ensemble de restrictions que combinés peuvent modéliser un domaine », Neches *et al.* (1991). Cette idée servait (et encore sert) à un objectif visant la réutilisation et le partage des sources de connaissance. La définition la plus citée dans le Web, celle de Gruber (1995) « une spécification explicite d'une conceptualisation » a été expliquée par Studer *et al.* (1998) comme : « la *conceptualisation* représente un modèle abstrait d'un phénomène en identifiant les concepts importants relatifs à ce phénomène ; alors que, *explicite* signifie que le type des concepts usés et les restrictions dérivées de son utilisation sont définis explicitement ».

Les ontologies forment la base du Web sémantique. Ce que veut dire que le métamodèle défini par l'ontologie est utilisé par tous les langages qui décrivent le Web sémantique et encore par d'autres applications qui décrivent des bases de connaissances et assistent dans le développement de ces bases. Le Web sémantique est une extension du Web actuel dans lequel les informations sont accompagnées par une signification bien définie en assistant les personnes et les ordinateurs à travailler de forme coopérative, Berners-Lee *et al.* (2001).

6.2.1 Langages pour la représentation des ontologies

Depuis son apparition dans l'informatique, plusieurs langages pour la représentation des ontologies ont étés proposés, parmi les plus importants on pourrait citer notamment :

- *OKBC* (ang. Open Knowledge Base Connectivity), Chaudhri *et al.* (1998), est une API permettant d'accéder à des bases de connaissance (Knowledge Representation System).
- *KIF* (ang. Knowledge Interchange Format), Ginsberg (1991), est un langage pro-

jeté pour être utilisé pour l'échange de connaissances entre systèmes informatiques dissemblables (développés par différents programmeurs, dans différents moments, dans langages différentes etc.). Il a été conçu pour être un format d'échange pour les ontologies et constitue une extension du langage de prédicats de premier ordre. Il fournit des moyens pour représenter connaissance sur connaissance. Ceci permet à l'utilisateur d'expliciter des décisions liées à la représentation de la connaissance et d'introduire une nouvelle construction de la représentation de la connaissance, sans modifier le langage.

– *RDF*(ang. Resource Description Framework),W3C (2004b), est un langage XML permettant de décrire des métadonnées et facilitant leur traitement. Il est censé faciliter le traitement automatique de l'information du Web par des agents logiciels, transformant ainsi le web d'un regroupement d'informations uniquement destinées aux humains, en un état de réseau de processus en coopération. Dans ce réseau, le rôle de RDF est de fournir une langage commun compréhensible par tous les agents. DF procède par une description de savoirs (données tout comme métadonnées) à l'aide d'expressions de structure fixée. La structure fondamentale de toute expression en RDF est une collection de triplets, chacun composé d'un sujet, un prédicat et un objet. Un ensemble de tels triplets est appelé un graphe RDF. Dans un graphe, chaque triplet représente l'existence d'une relation entre les choses symbolisées par les noeuds qui sont joints.

– *DAML+OIL* (ang. Ontology Inference Layer + Ontology Inference Layer), W3C (2001), est un ensemble de déclarations RDF et XML. RDF a été conçue pour permettre aux utilisateurs de construire leurs propres définitions de métadonnées. Néanmoins, à moins que le producteur et le consommateur des informations aient le même accord commun, les informations ne peuvent pas être partagées. RDF n'est pas suffisante parce que permet seulement une construction limitée des restrictions, seulement applicable dans des termes de la portée et/ou les propriétés du domaine, donc il n'admet pas la représentation des propriétés des propriétés, l'équivalence ou la disjonction et il ne possède pas sémantique définie. Le langage DAML+OIL étend RDF en facilitant la construction de modèles d'inférence. OIL est un langage basé sur des frames qui étend RDFS (Projet RDF) avec un ensemble de primitives qui augmente les possibilités de description. DAML est une extension de RDF et une partie de leurs idées sont inspirées de OIL. En novembre 2001, la W3C a débuté le Web Ontology Working Group, ayant pour objectif la définition d'un langage pour le Web sémantique. Ce groupe a utilisé DAML+OIL comme son point de départ et visait la définition d'un nouveau langage appelée *OWL* (ang. Web Ontology Language)

Le langage OWL

OWL, W3C (2004a), est tout comme RDF, un langage XML profitant de l'universalité syntaxique de XML. Fondé sur la syntaxe de RDF/XML, OWL offre un moyen d'écrire des ontologies Web. OWL se différencie du couple RDF/RDFS en ceci que, contrairement à RDF, il est justement un langage d'ontologies. Si RDF et RDFS apportent à l'utilisateur la capacité de décrire des classes et des propriétés, OWL intègre, en plus, des outils de comparaison des propriétés et des classes : identité, équivalence, contraire, cardinalité, symétrie, transitivité, disjonction, etc. OWL offre aux machines une plus grande capacité d'interprétation du contenu Web que RDF et RDFS, grâce à un vocabulaire plus large et à une vraie sémantique formelle.

OWL W3C (2004a), est formé de trois sous-langages offrant des capacités d'expression croissantes et destinés à des communautés différentes d'utilisateurs :

- *OWL Lite* est le sous langage de OWL le plus simple. Il est destiné aux utilisateurs qui ont besoin d'une hiérarchie de concepts simple. OWL Lite est adapté, par exemple, aux migrations rapides depuis d'anciens thésaurus.
- *OWL DL* est plus complexe que OWL Lite, permettant une expressivité bien plus importante. OWL DL est fondé sur la logique descriptive (d'où son nom, OWL Description Logics), un domaine de recherche étudiant la logique, et conférant donc à OWL DL son adaptation au raisonnement automatisé. Malgré sa complexité relative face à OWL Lite, OWL-DL garantit la complétude des raisonnements (toutes les inférences sont calculables) et leur décidabilité (leur calcul se fait en une durée finie).
- *OWL Full* est la version la plus complexe d'OWL, mais également celle qui permet le plus haut niveau d'expressivité. OWL Full est destiné aux situations où il est plus important d'avoir un haut niveau de capacité de description, quitte à ne pas pouvoir garantir la complétude et la décidabilité des calculs liés à l'ontologie. OWL Full offre cependant des mécanismes intéressants, comme par exemple la possibilité d'étende le vocabulaire par défaut de OWL. Il existe entre ces trois sous langage une dépendance de nature hiérarchique : toute ontologie OWL Lite valide est également une ontologie OWL DL valide, et toute ontologie OWL DL valide est également une ontologie OWL Full valide.

Eléments du langage

Dans cette partie, nous introduisons les éléments du langage OWL le plus importants. Pour des explications exhaustives du rôle de chacun des éléments composant le

vocabulaire d'OWL, il est recommandé de se reporter à W3C (2004a).

Les classes

Une classe définit un groupe d'individus qui sont réunis parce qu'ils ont des caractéristiques similaires. L'ensemble des individus d'une classe est désigné par le terme « extension de classe », chacun de ces individus étant alors une « instance » de la classe. Les trois versions d'OWL comportent les mêmes mécanismes de classe, à ceci près que OWL FULL est la seule version à permettre qu'une classe soit l'instance d'une autre classe (d'une métaclasse). A l'inverse, OWL Lite et OWL DL n'autorisent pas qu'une instance de classe soit elle-même une classe.

La déclaration d'une classe se fait par le biais du mécanisme de « description de classe », qui se présente sous diverses formes. Une classe peut se déclarer des manières suivantes :

1. l'indicateur de classe. La description de la classe se fait, dans ce cas, directement par le nommage de cette classe.
2. l'énumération des individus composant la classe. Ce type de description se fait en énumérant les instances de la classe, à l'aide de la propriété *owl :oneOf*. Ce mécanisme ne fait pas partie de OWL Lite.
3. La restriction de propriétés. La description par restriction de propriété permet de définir une classe anonyme composée de toutes les instances de *owl :Thing* qui satisfont une ou plusieurs propriétés. Ces contraintes peuvent être de deux types : contrainte de valeur ou contrainte de cardinalité. Une contrainte de valeur s'exerce sur la valeur d'une certaine propriété de l'individu (par exemple, pour un individu de la classe *Humain*, *sexe = Homme)*, tandis qu'une contrainte de cardinalité porte sur le nombre de valeurs que peut prendre une propriété (par exemple, pour un individu de la classe *Humain*, *aPourFrere* est une propriété qui peut ne pas avoir de valeur, ou avoir plusieurs valeurs, suivant le nombre de frères de l'individu. La contrainte de cardinalité portant sur *aPourFrere* restreindra donc la classe décrite aux individus pour lesquels la propriété *aPourFrere* apparaît un certain nombre de fois). Il existe naturellement différents opérateurs de comparaison pour établir les contraintes.
4. Enfin, les descriptions par intersection, union ou complémentaire permettent de décrire une classe par, comme leur nom l'indique, l'intersection, l'union ou le complémentaire d'autres classes déjà définies, ou dont la définition se fait au sein même de la définition de la classe courante.

Il existe dans toute ontologie OWL une superclasse, nommée *Thing*, dont toutes les autres classes sont des sous-classes. Ceci nous amène directement au concept d'héritage,

disponible à l'aide de la propriété *subClassOf*. Il existe également une classe nommée *noThing*, qui est sous-classe de toutes les classes OWL. Cette classe ne peut avoir aucune instance.

Les instances de classes

La définition d'un individu consiste à énoncer un « fait », encore appelé « axiome d'individu ». On peut distinguer deux types de faits :

- les faits concernant l'appartenance à une classe. La plupart des faits concerne généralement la déclaration de l'appartenance à une classe d'un individu et les valeurs de propriété de cet individu.
- les faits concernant l'identité des individus. Une difficulté qui peut éventuellement apparaître dans le nommage des individus concerne la non-unicité éventuelle des noms attribués aux individus. Par exemple, un même individu pourrait être désigné de plusieurs façons différentes. C'est la raison pour laquelle OWL propose un mécanisme permettant de lever cette ambiguïté, à l'aide des propriétés *owl :sameAs*, *owl :diffrentFrom* et *owl :allDifferent*.

Les propriétés

Les propriétés OWL permettent d'exprimer des faits au sujet des classes et de leurs instances. OWL fait la distinction entre deux types de propriétés :

1. les propriétés d'objet, *owl :ObjectProperty*, permettent de relier des instances à d'autres instances,
2. les propriétés de type de donnée, *owl :DatatypeProperty*, permettent de relier des individus à des valeurs de données.

Ces deux classes sont elles-même sous-classes de la classe RDF *rdf :Property*. La définition des caractéristiques d'une propriété se fait à l'aide d'un axiome de propriété qui, dans sa forme minimale, ne fait qu'affirmer l'existence de la propriété. En plus du mécanisme d'héritage et de restriction du domaine et de l'image d'une propriété, il existe divers moyens d'attacher des caractéristiques aux propriétés, ce qui permet d'affiner grandement la qualité des raisonnements liés à cette propriété. Parmi les caractéristiques de propriétés principales, on trouve la transitivité, la symétrie, la fonctionnalité et l'inverse.

Le Langage SWRL

SWRL (ang. Semantic Web Rule Language), W3C (2004c), est une proposition de standard pour la spécification de règles. Il combine les sous langages OWL DL et OWL

Lite de OWL avec certains sous langages de RuleML. En particulier, SWRL définit plusieurs opérateurs mathématiques.

SWRL est un langage qui enrichit la sémantique d'une ontologie définie en OWL. SWRL permet contrairement à OWL, de manipuler des instances par des variables (?x, ?y, ?z,...). SWRL ne permet pas de créer des concepts ni des relations, il permet simplement d'ajouter des relations suivant les valeurs des variables et la satisfaction de la règle. Les règles SWRL sont construites suivant ce schéma : *antécédent* → *conséquent*.

L'antécédent et le conséquent repésentent des conjonctions d'atomes. Un atome est une instance de concept, une relation OWL ou une des deux relations SWRL *same-as(?x, ?y)* ou *different-from (?x, ?y)*. Le fonctionnement d'une règle est basée sur le principe de satisfiabilité de l'antécédent ou du conséquent. Pour une règle, il existe trois possibilités :

1. l'antécédent et le conséquent sont définis. Si l'antécédent est satisfait alors le conséquent doit l'être ;

2. l'antécédent est vide, cela équivaut à un antécédent satisfait ce qui permet de définir des faits ;

3. le conséquent est vide, cela équivaut à un conséquent insatisfait, l'antécédent ne doit pas être satisfiable.

Exemple de règle

Dans l'exemple suivant, la relation *estOncleDe* a été construite en OWL, SWRL apporte la description de cette relation et relie les instances concernées. Ainsi, OWL permet de définir le concept d'Oncle de la manière suivante :

intersectionOf(SubClassOf(Homme), estfrereDe(Pere))

Néanmoins, OWL ne permet pas de définir une relation qui représente le fait d'être oncle d'une personne. Avec SWRL nous définissons une telle relation de la manière suivante :

$aEnfant(?x, ?y) \wedge estfrereDe(?z, ?x) \rightarrow estOncleDe(?z, ?y)$

6.3 La qualité de service

Il n'existe pas de consensus sur la définition de la qualité de service (QdS). La recommandation ITU-X.902[1] définit la QdS comme « un ensemble d'exigences dans le comportement collectif d'un ou plusieurs objets ». Dans le contexte des technologies de l'information et multimédia, la QdS a été définie par Vogel *et al.* (1995), comme « l'ensemble des caractéristiques quantitatives et qualitatives d'un système multimédia, nécessaires pour atteindre la fonctionnalité requise par l'application ». Nous pouvons aussi dire que la qualité de service représente l'aptitude d'un service à répondre de manière adéquate à des exigences qui visent à satisfaire ses usagers. Ces exigences peuvent être liées à plusieurs aspects d'un service, par exemple : son accessibilité, sa disponibilité, sa fiabilité, etc.

Afin de pouvoir gérer la QdS dans une application à base de services Web, nous tenons à identifier les différentes catégories ou types de dysfonctionnement que l'application pourrait confronter. Ainsi, dans cette section nous proposons une classification des dysfonctionnements organisée par catégories.

La figure 6.1 présente la classification des dysfonctionnements proposée. Cette classification considère trois catégories majeures :

1. La catégorie *QoS Mismatch* considère les dysfonctionnements liés au non respect du contrat de la QdS entre le service demandeur et le service fournisseur.
2. La catégorie *Semantical Mismatch* considère les dysfonctionnements liés aux mauvaises interprétations des méthodes ou des paramètres entre le service demandeur et le service fournisseur (par exemple, deux interfaces qui offrent des fonctionnalités similaires mais qui sont adressées avec des noms différents).
3. La catégorie *Functional Mismatch* considère toutes les autres catégories qui ne sont classées dans les deux premières catégories. Cette catégorie considère les problèmes d'exécution d'un service dûs, par exemple, à une implémentation incorrecte du service.

La catégorie destinée à la gestion de la QdS est décomposée dans les sous-catégories suivantes :

- La catégorie *Generic QoS* considère des valeurs tels que la disponibilité, la sécurité, le temps de réponse, et le « throughput ».
- La catégorie *Application-specific QoS* est décomposée en :

[1] The International Telecommunication Union (ITU) standard X.902, Information technology - Open distributed processing - Reference Model.

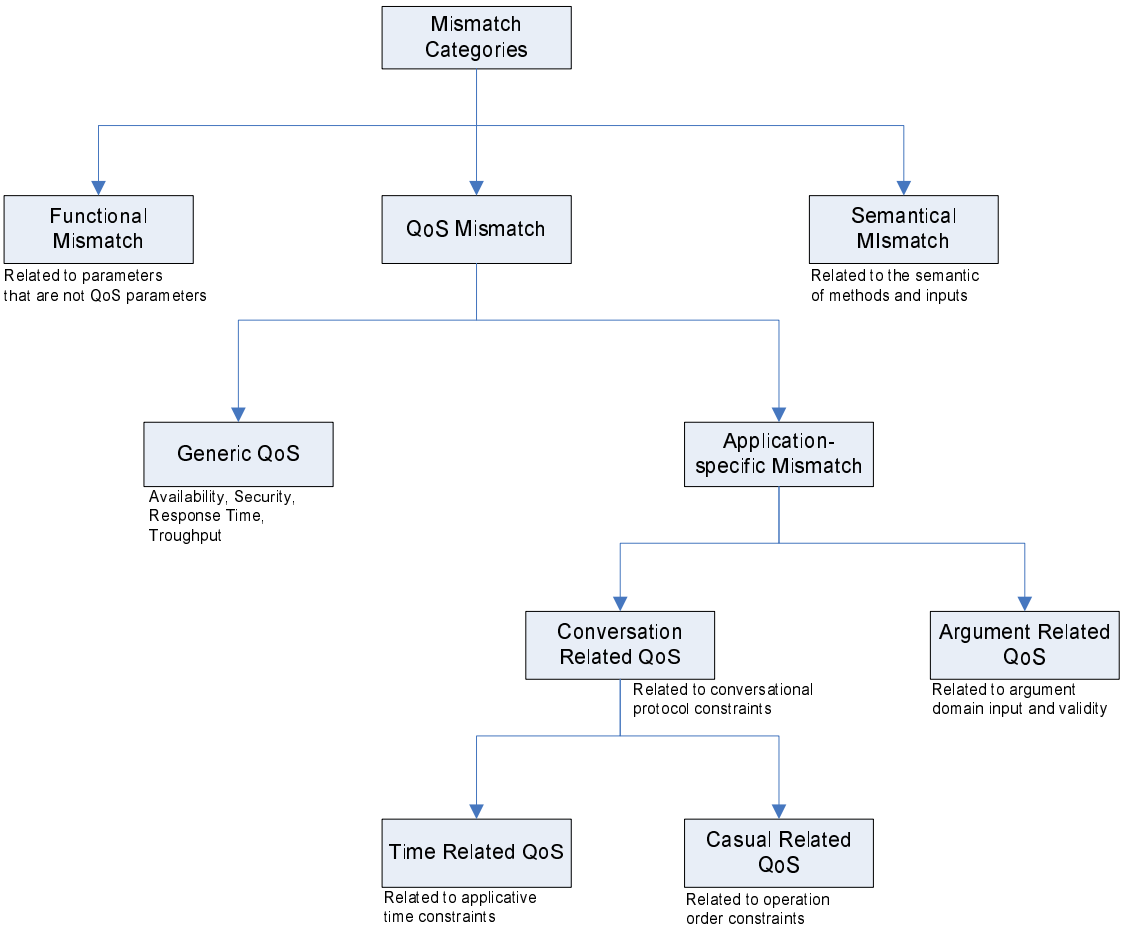


FIG. 6.1 – Classification générale des dysfonctionnements.

- La catégorie *Argument Related QoS* traite la validité dans les valeurs des paramètres (ou leurs combinaisons).
- La catégorie *Conversational Related QoS* considère la validité des protocoles gérant les interactions entre les services demandeurs et fournisseurs. Elle est décomposée en :
  - La catégorie *Time Related QoS* considère les problèmes liés aux contraintes de temps.
  - La catégorie *Causal Related QoS* considère les problèmes liés à l'ordre correcte dans l'exécution des opérations entre services.

6.3.1 La QdS dans la revue coopérative

Concernant le processus de revue coopérative, nous avons identifié une liste des dysfonctionnements qui pourraient dégrader la QdS. En effet, il s'agit des événements à mesurer afin de diagnostiquer des éventuels problèmes du système. La partie dite détection est menée, dans une architecture de gestion de la QdS, par les composants de surveillance, en première instance, et puis par les composants de diagnostic en deuxième instance. Par la suite nous présentons la liste des dysfonctionnements identifiés et classés par rapport à la classification des dysfonctionnements donnée dans la section précédente.

I. Activité de recherche de conférence

1. Renvoie de conférences dont le « deadline » est dépassé : *Argument Related QoS*.
   - Causes possibles :
     (a) inconsistance dans la sémantique de la date,
     (b) dysfonctionnement du service Web ConfInfoProv,
     (c) message en retard.
   - Détection possible :
     (a) par l'utilisateur,
     (b) par le service Web AuthorMgr.
2. Renvoie de conférences dont le thème n'est pas pertinent : *Argument Related QoS*.
   - Causes possibles :
     (a) inconsistance dans la sémantique du thème,
     (b) dysfonctionnement du service Web ConfInfoProv,

(c) message en retard.

– Détection possible :

(a) par l'utilisateur,

(b) par le service Web AuthorMgr.

II. Activités d'inscription des auteurs et de soumission d'articles

1. Inscription d'un auteur et non réception d'une confirmation : *Time Related QoS*.
   – Causes possibles :
   (a) perte de connexion,
   (b) panne du service Web SubmisMgr,
   (c) panne du service Web AuthorMgr.
   – Détection possible :
   (a) par l'utilisateur (si panne du service Web AuthorMgr),
   (b) par le service Web AuthorMgr (si perte de connexion entre le service Web AuthorMgr et le service Web SubmisMgr, ou si panne du service Web SubmisMgr).
2. Soumission d'un papier et non réception d'une confirmation : *Time Related QoS*.
   – Causes possibles :
   (a) perte de connexion,
   (b) panne du service Web SubmisMgr,
   (c) panne du service Web AuthorMgr.
   – Détection possible :
   (a) par l'utilisateur (si panne du service Web AuthorMgr),
   (b) par le service Web AuthorMgr (si perte de connexion entre le service Web AuthorMgr et le service Web SubmisMgr, ou si panne du service Web SubmisMgr).
3. Inscription correcte mais puis impossibilité de connexion : *Functional Mismatch*.
   – Causes possibles :
   (a) perte de connexion,
   (b) panne du service Web SubmisMgr,
   (c) panne du service Web AuthorMgr.
   – Détection possible :

(a) par l'utilisateur (si panne du service Web AuthorMgr),

(b) par le service Web AuthorMgr (si perte de message entre le service Web AuthorMgr et le service Web SubmisMgr, ou si panne du service Web SubmisMgr),

4. Pas de soumission de papier après l'inscription d'un auteur : *Time Related QoS*.
   - Causes possibles :
     (a) l'utilisateur n'a pas soumis le papier,
     (b) perte de connexion,
     (c) panne du service Web AuthorMgr.
   - Détection possible :
     (a) par l'utilisateur (si perte de connexion entre le service Web AuthorMgr et le service Web SubmisMgr, ou si panne du service Web AuthorMgr),
     (b) par le service Web SubmisMgr (si perte de connexion entre le service Web AuthorMgr et le service Web SubmisManager, ou si panne du service Web AuthorMgr).

III. Activité d'affectation de papiers aux relecteurs

1. Non affectation d'un relecteur : *Functional Mismatch*.
   - Causes possibles :
     (a) perte de connexion,
     (b) dysfonctionnement du service Web ReviewingMgr,
     (c) dysfonctionnement du service Web ReviewerMgr,
   - Détection possible :
     (a) par l'utilisateur.
2. Relecteurs non qualifiés (sujet de recherche ou compétences non concordants) : *Argument Related QoS*.
   - Causes possibles :
     (a) dysfonctionnement du service Web ReviewingMgr.
   - Détection possible :
     (a) par le service Web ReviewerMgr,
     (b) par l'utilisateur relecteur.
3. Relecteur qui est auteur du papier (ou qui travaille dans la même institution) : *Argument Related QoS*.

- Causes possibles :
  (a) dysfonctionnement du service Web ReviewingMgr (e.g. prise en compte d'un nombre insuffisant de relecteurs).
- Détection possible :
  (a) par le service Web ReviewerMgr,
  (b) par l'utilisateur relecteur.

IV. Activité de qualification des papiers

1. Papier non envoyé à temps (« deadline » dépassé) aux relecteurs : *Time Related QoS*.
   - Causes possibles :
     (a) propagation du dysfonctionnement « Non affectation d'un relecteur »,
     (b) perte de connexion,
     (c) dysfonctionnement ou panne du service Web ReviewingMgr.
   - Détection possible :
     (a) par l'utilisateur relecteur.
     (b) par le service Web ReviewerMgr (si non envoie ou perte de message entre le service Web ReviewingMgr et le service Web ReviewerMgr).
2. Rapport non envoyé à temps (« deadline » dépassé) : *Time Related QoS*.
   - Causes possibles :
     (a) le relecteur ne l'a pas envoyé,
     (b) perte de connexion,
     (c) dysfonctionnement ou panne du service Web ReviewerMgr.
   - Détection possible :
     (a) par le service Web ReviewingMgr (si non envoie ou perte de message entre le service Web ReviewerMgr et le service Web ReviewingMgr).
3. Rapport ne correspondant pas au papier (paperId) : *Argument Related QoS*.
   - Causes possibles :
     (a) le relecteur confond les ids de deux papiers différents,
     (b) perte de connexion,
     (c) dysfonctionnement des services Web ReviewerMgr ou ReviewingMgr, ou ApprovalMgr.
   - Détection possible :

(a) par le service Web ReviewingMgr (si dysfonctionnement ou perte de connexion avec le service Web ReviewerMgr),

(b) par l'auteur (si dysfonctionnement du service Web ApprovalMgr ou perte de connexion entre le service Web ApprovalMgr et le service Web ReviewingMgr).

V. Activité de sélection de papiers et notification

1. Décision ne respectant pas le délai de temps : *Time Related QoS*.
   - Causes possibles :
     (a) propagation du dysfonctionnement « Non affectation d'un papier »,
     (b) perte de connexion.
   - Détection possible :
     (a) par l'auteur.

VI. Activité de soumission des versions finales de papiers

1. Non envoie de version final du papier dans la limite du temps (« deadline ») : *Time Related QoS*.
   - Causes possibles :
     (a) l'auteur ne le renvoie pas,
     (b) perte de connexion,
     (c) dysfonctionnement du service Web AuthorMgr.
   - Détection possible :
     (a) par le service Web PublishingMgr,
     (b) par le PCC (dans tous les cas).

6.4 Modélisation de la QdS par les ontologies

Dans cette section nous abordons la modélisation de la QdS par le biais des ontologies et nous utilisons OWL en tant que langage pour leur définition. Pour cette

modélisation nous avons choisi l'outil Protégé[2]. Protégé est un éditeur d'ontologies distribué en open source par le groupe d'informatique médicale de l'université de Stanford. Protégé n'est pas un outil spécialement dédié à OWL, mais un éditeur hautement extensible, capable de manipuler des formats très divers. Le support d'OWL, comme de nombreux autres formats, est possible dans Protégé grâce à un plugin dédié.

Concernant les règles d'inférence, nous avons utilisé pour l'édition de règles le plugin SWRL[3] de Protégé, et pour la validation et l'exécution des règles le moteur d'inférence Jess[4].

6.4.1 Définition de la structure de l'ontologie

La première étape dans la définition d'une ontologie consiste à définir sa structure, c'est-à-dire, les classes qui la caractérisent. Du fait que nous sommes intéressés à appliquer l'ontologie à l'exemple de la revue coopérative, représenté par le système de gestion de conférences, nous pouvons définir l'ontologie en deux parties. Une première partie qui caractérise un système de gestion de conférences et une deuxième partie caractérisant les différents types de dysfonctionnements affectant la QdS.

Pour la première partie nous proposons la réutilisation et l'adaptation de l'ontologie KA[5]. Cette ontologie définit des concepts liés au domaine de la recherche académique, et a été proposée par Ian Horrocks.

Pour la deuxième partie nous reprenons la classification introduite précédemment.

La figure 6.2 présente la structure des classes de l'ontologie proposée. La classe racine de toute ontologie est *owl :Thing*. L'ontologie KA, que nous avons importée, est visualisée à partir de la classe *Object*. Cette ontologie a été simplifiée et adaptée, en fonction de nos besoins, au niveau des classes et des propriétés. L'ontologie caractérisant la classification des dysfonctionnements est visualisée à partir de la classe *Mismatch-Categories*. Ce premier aperçu de l'ontologie ne montre pas les relations entre classes, du fait que ceci est établi au niveau des propriétés.

[2]http ://protege.stanford.edu/
[3]http ://protege.cim3.net/cgi-bin/wiki.pl ?SWRLJessTab
[4]http ://herzberg.ca.sandia.gov/jess/
[5]http ://protege.cim3.net/cgi-bin/wiki.pl ?ProtegeOntologiesLibrary

6.4.2 Définition des propriétés de l'ontologie

Les propriétés d'une classe permettent de la caractériser et d'établir des relations d'interdépendance. Un premier cas est illustré par la figure 6.3. La classe *ArgumentRelatedQoS* contient deux propriétés : *nonQualifiedreviewer* et *reviewerMismatch*. La première correspond au cas où un relecteur a été affecté avec un papier qui ne correspond pas à son domaine de recherche. Et la deuxième correspond au cas où un relecteur a été affecté avec un papier dont lui est un des auteurs.

La définition de ces deux propriétés, elle même, ne permet pas de déduire les valeurs qui leurs seront affectées lors de la création des instances de la classe *ArgumentedRelatedQoS*. A ce moment, ces deux propriétés pourraient prendre des valeurs à partir de l'application des règles d'inférence. Donc, on pourrait dire pour ce type de propriétés, qu'il s'agit de propriétés inférées.

Les autres classes qui définissent la classification des dysfonctionnements de la QdS, contiennent des propriétés du même type.

Un autre cas montre la définition des propriétés de la classe *Publication*, illustrée par la figure 6.4. La propriété *hasReviewer* montre une restriction indiquant un nombre minimum de relecteurs associés à une publication. Ce type de restriction est établi par le biais des fonctionnalités offertes par le langage OWL. De même, les valeurs associées lors de la création des instances sont validées par l'outil Protégé. Une autre propriété,

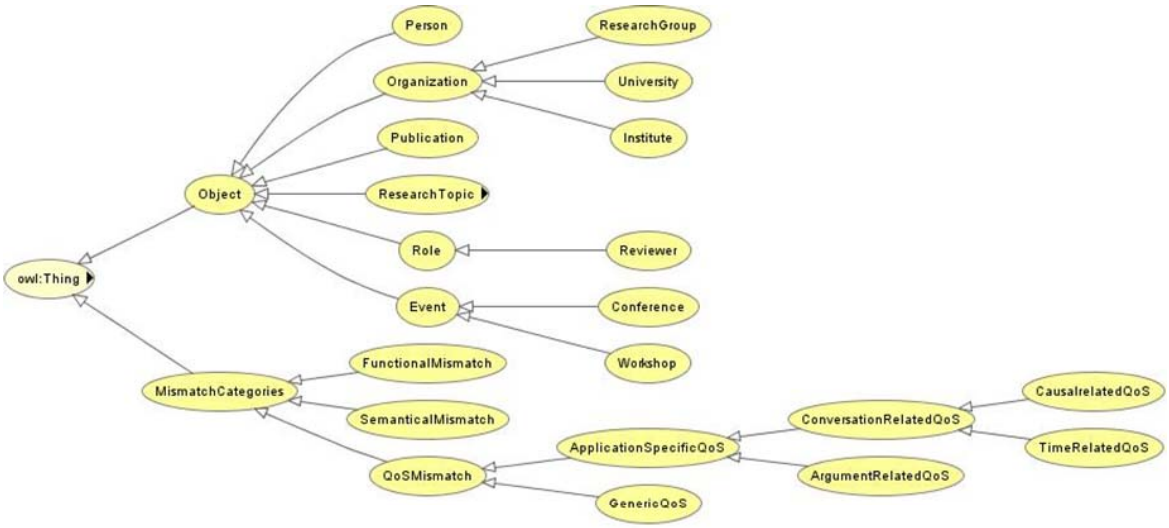


FIG. 6.2 – Ontologie caractérisant les cas de dysfonctionnements dans un système de gestion de conférences.

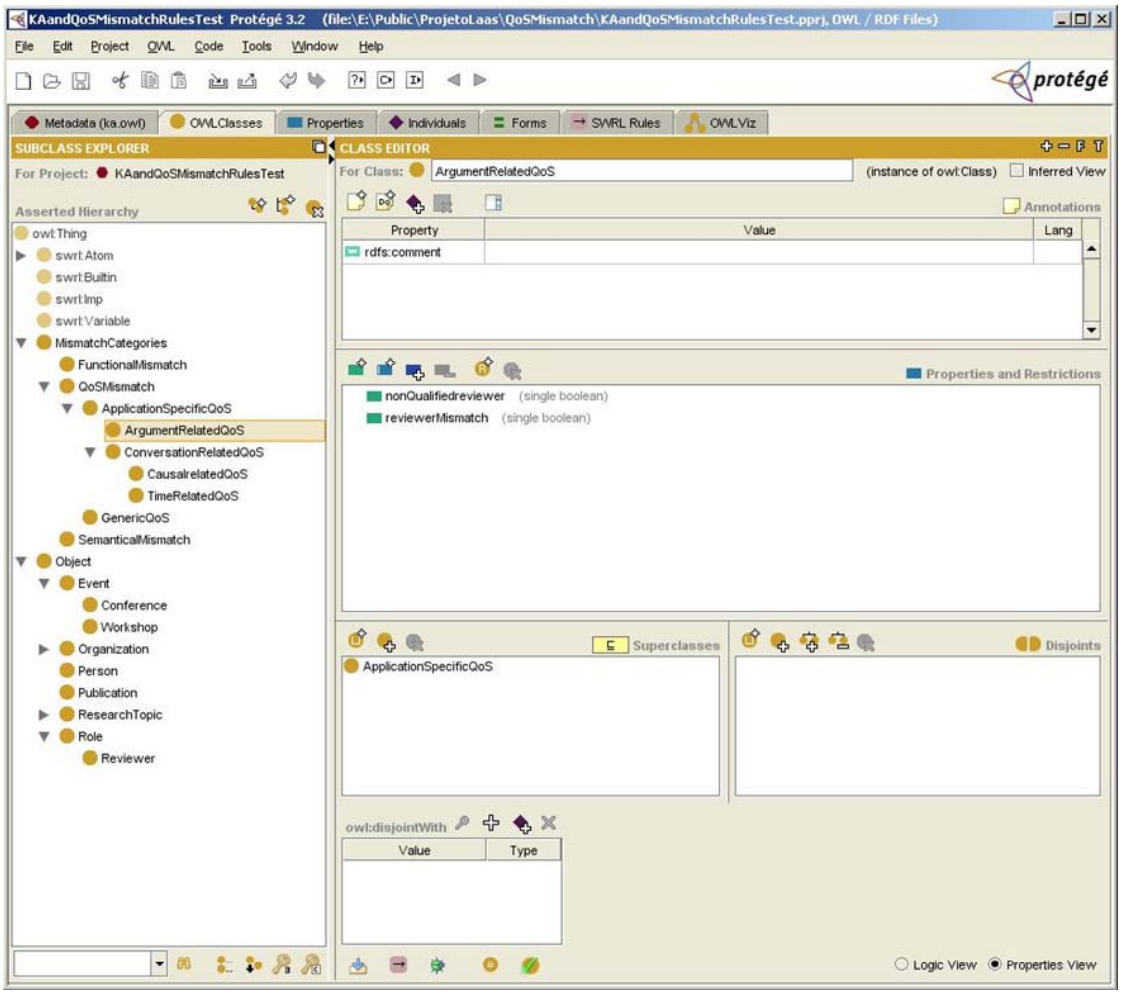


FIG. 6.3 – Propriétés de la classe *ArgumentRelatedQoS*.

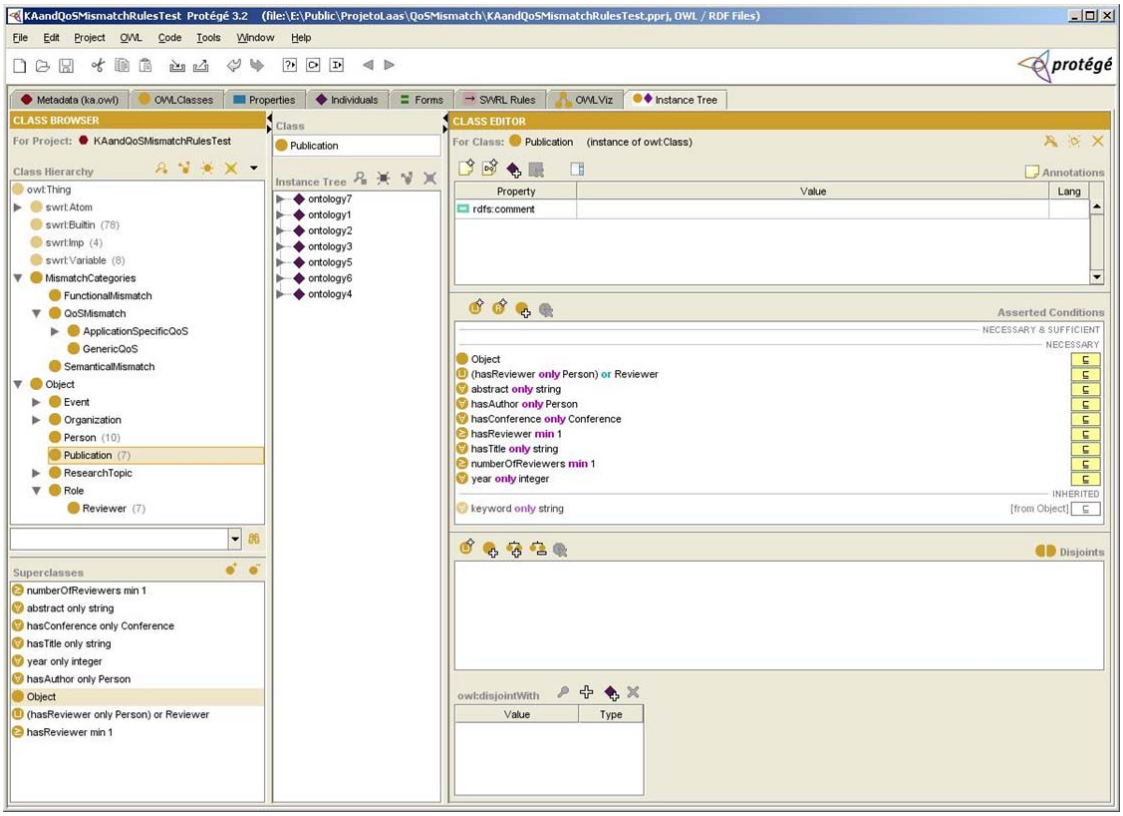


FIG. 6.4 – Propriétés de la classe *Publication*.

telle que *hasConference* met en rapport la classe *Publication* avec la classe *Conference* en indiquant que les valeurs valides pour cette propriété doivent correspondre à la définition de la classe *Conference*. Une propriété du même genre est *hasAuthor* qui relie la classe *Publication* avec la classe *Person*.

A la fin de la liste des propriétés de la classe *Publication*, on peut trouver une propriété nommée *keyword*. Cette propriété dénote les mots clés associés à la classe *Publication*. De même, elle correspond à une propriété qui a été héritée d'une super-classe, en l'occurence la classe *Object*.

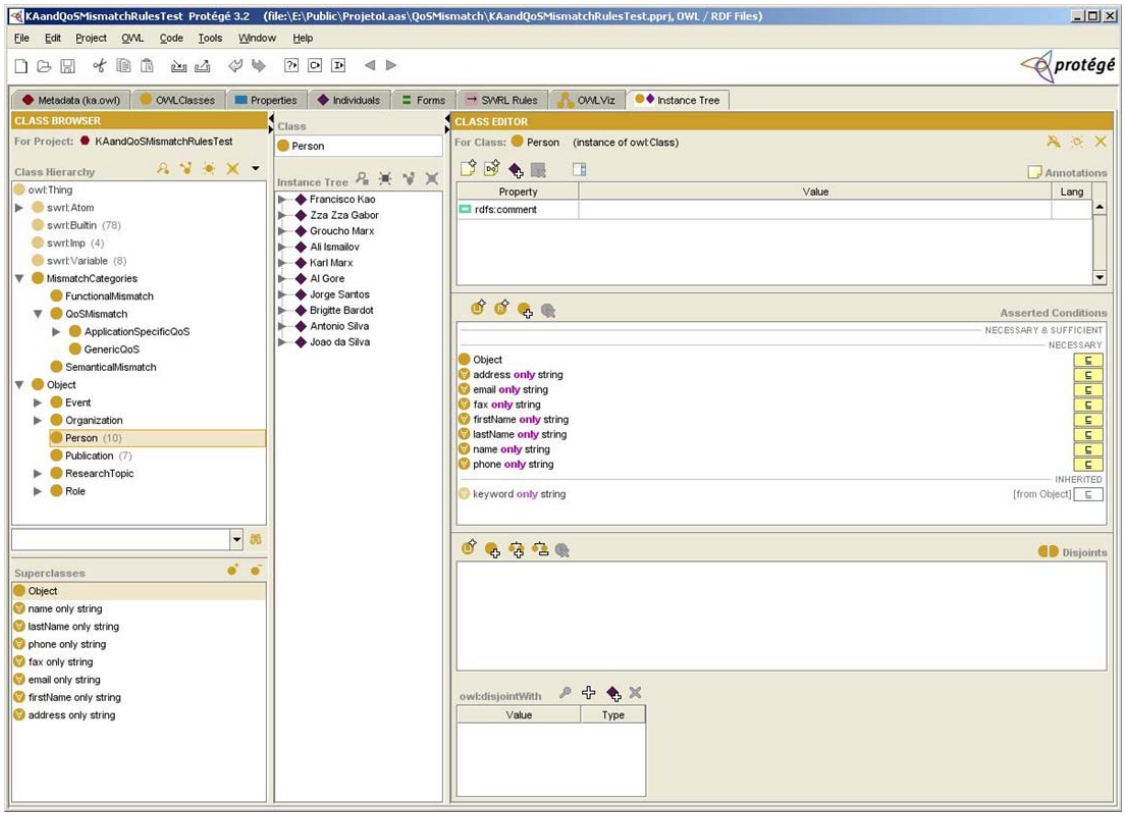


FIG. 6.5 – Propriétés de la classe *Person*.

Les propriétés de la classe *Person* sont illustrées par la figure 6.5. Ces propriétés correspondent à des données particulières caractérisant une personne. Les valeurs valides pour ces propriétés sont associées à un type, en l'occurence le type *string*. Dans la même figure, on peut apercevoir quelques individus (instances) associés à cette classe.

Un dernière définition des propriétés correspond au cas de la classe *Reviewer*, illustrée par la figure 6.6. Un relecteur doit être une personne, donc ceci est définie par la propriété *describedBy* qui met en rapport la classe *Reviewer* avec la classe *Person*. Un relecteur doit avoir au moins une publication affféctée, ceci est défini par la propriété

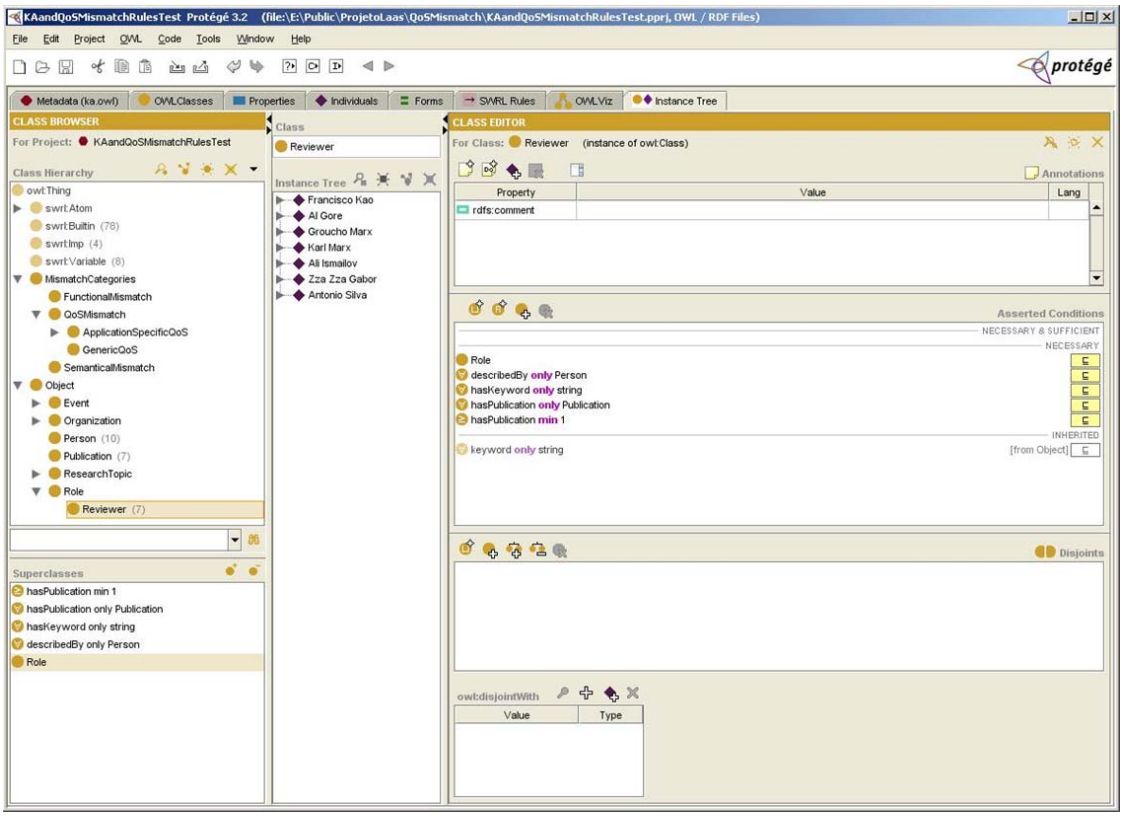


FIG. 6.6 – Propriétés de la classe *Reviewer*.

hasPublication.

6.4.3 Définition des individus

Une fois que l'on a défini les classes et leurs propriétés, on doit créer des individus afin d'alimenter la base de connaissance issue de l'ontologie. La création des individus est validée par les restrictions imposées pendant la définition des propriétés.

Les figures 6.7, 6.8, 6.9 et 6.10, illustrent successivement la création des individus pour les classes *Conference*, *Publication*, *Person* et *Reviewer*. Dans tous le cas, le volet *INSTANCE BROWSER* liste les individus déjà créés pour la classe en question.

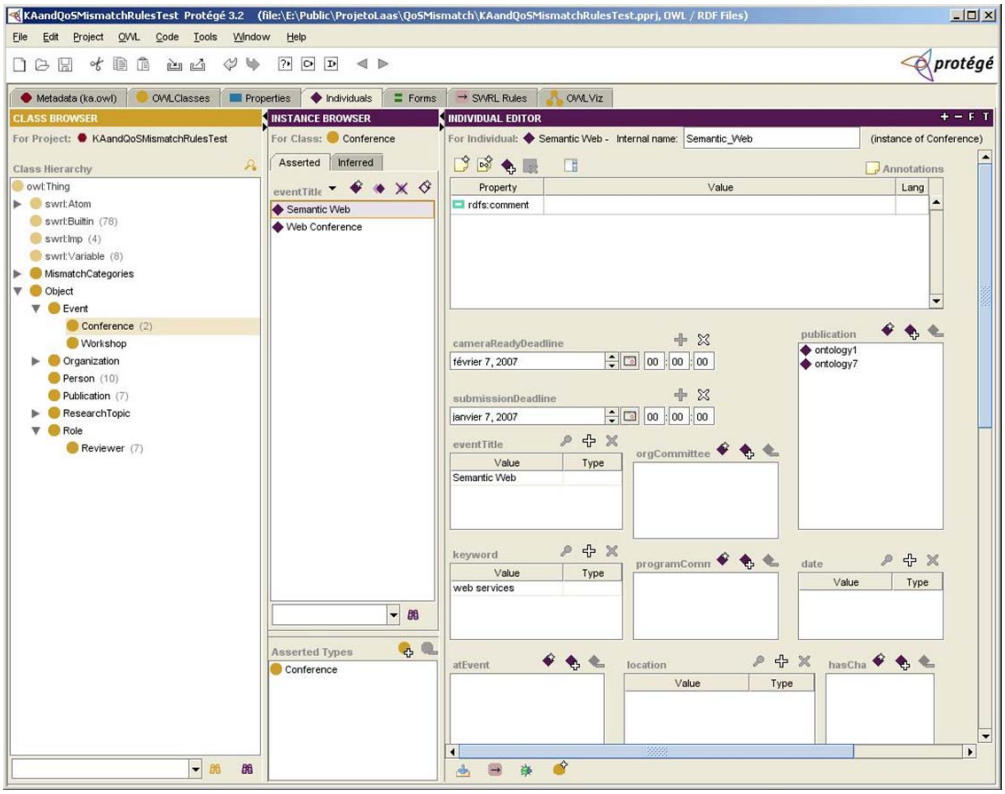


FIG. 6.7 – Instances de la classe *Conference*.

Un exemple de restriction, introduit précédemment, est celui du nombre minimum de publications affectées à un relecteur. Ainsi, dans la figure 6.10, lors de la création des individus de la classe *Reviewer*, la propriété *hasPublication* a été encadrée en rouge (ou en gris foncé), indiquant que la valeur courante (case vide) n'est correspond pas à la restriction imposée par la propriété (minimum 1).

Ce type de restriction permet de prévenir certain types de dysfonctionnements identifiés dans l'exemple du système de gestion de conférences. A l'occurrence, lorsqu'un papier n'a pas été affecté à un relecteur (dysfonctionnement *III.1*).

6.4.4 Définition et application des règles

Du fait que le langage OWL est limité au niveau des inférences que l'on peut établir lors de la définition des propriétés des classes. Nous devons construire des règles d'inférence qui vont nous permettre de déduire des dysfonctionnements qui pourraient entraîner une dégradation de la QdS.

Comme annoncé préalablement, la définition des règles est faite en utilisant le langage SWRL supporté par le plugin ayant le même nom dans Protégé. Ensuite, la compilation et l'exécution des règles sont faites par le biais du moteur d'inférence Jess.

Les dysfonctionnements à repérer sont ceux identifiés et exposés dans la classification des dysfonctionnements de la QdS introduite précédemment.

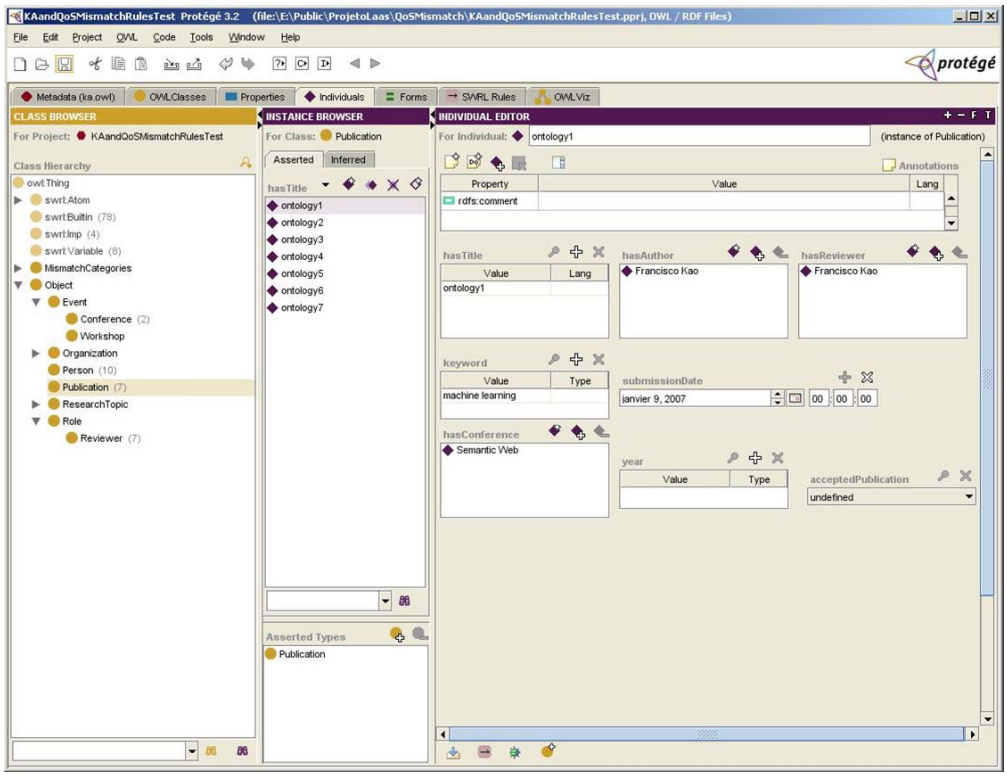


FIG. 6.8 – Instances de la classe *Publication*.

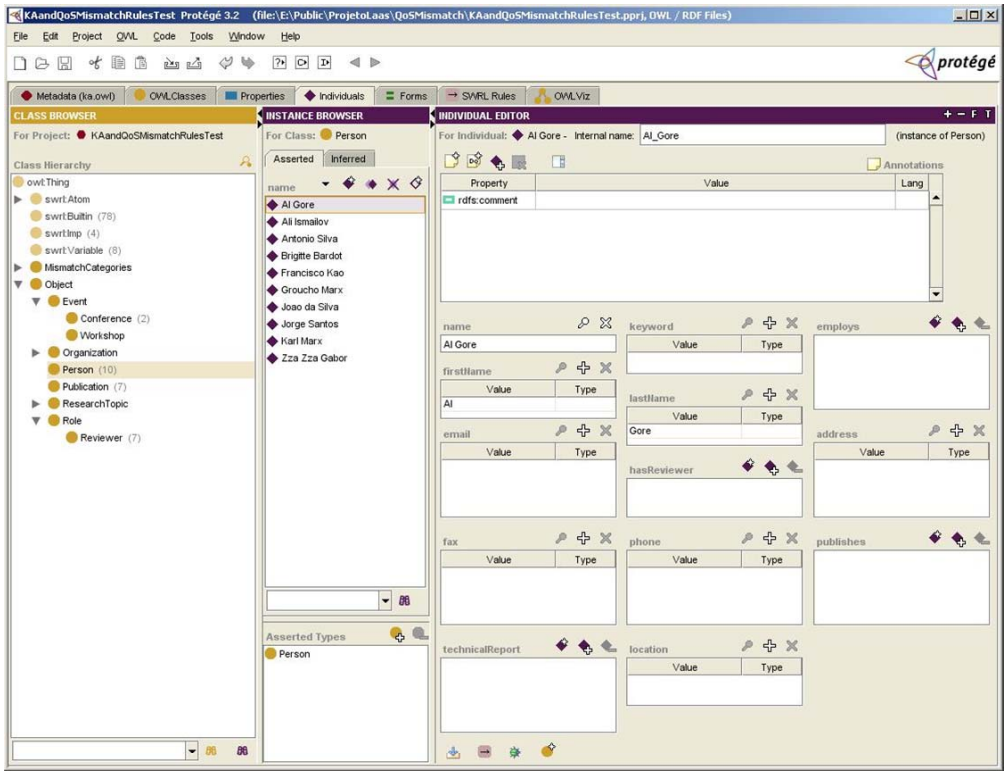


FIG. 6.9 – Instances de la classe *Person*.

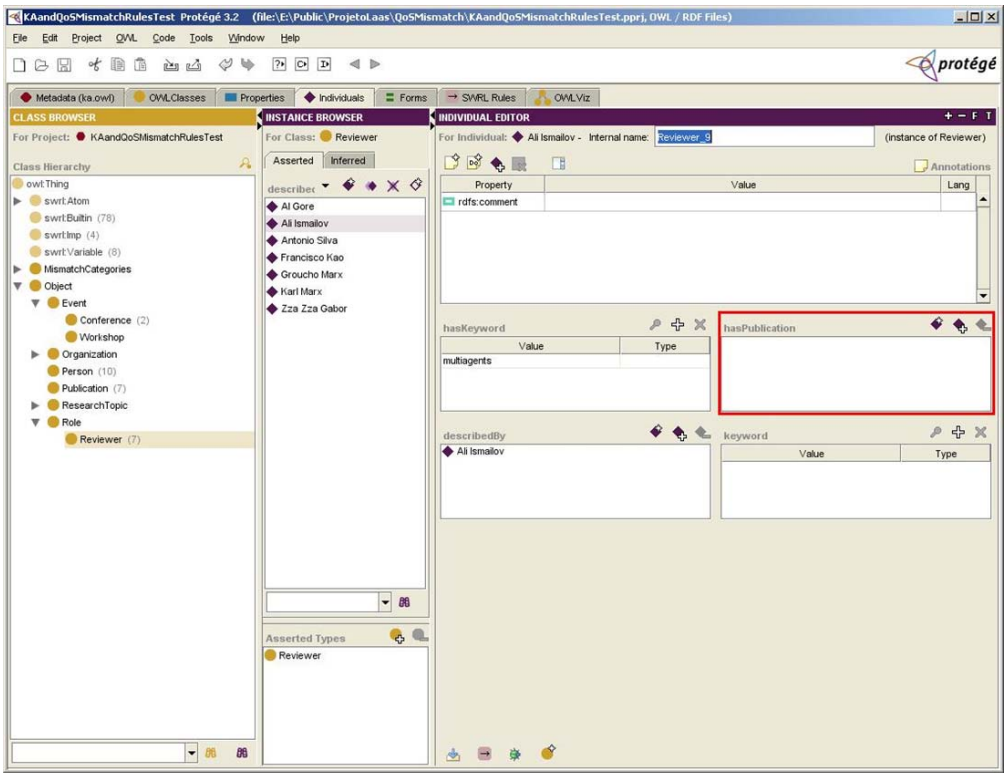


FIG. 6.10 – Instances de la classe *Reviewer*.

Une première règle nommée *keywordMismatch* est illustrée par la figure 6.11. Cette règle permet d'identifier les relecteurs dont le papier affecté ne correspond pas à son domaine de recherche. Ceci a été présenté comme le dysfonctionnement *III.2* dans la classification des dysfonctionnements de la QdS pour le système de gestion de conférences. D'autres types de dysfonctionnements pourrait être identifiés par des règles du même type, notamment le dysfonctionnement *I.2* Renvoie de conférences dont le thème n'est pas pertinent et le dysfonctionnement *IV.3* Rapport ne correspondant pas au papier (paperId).

La logique derrière la définition de cette règle est comme suit :

– Du côté de l'antécédent :

1. pour chaque publication cherche ses mots clés et les noms des relecteurs qui lui ont été affectés,
2. pour chaque relecteur récupère son nom et cherche les mots clés qui définissent son domaine de recherche,
3. du fait que les noms des relecteurs sont obtenus à partir de deux classes différentes (*Publication* et *Reviewer*), on doit vérifier qu'il s'agit de la même personne, en comparant leurs noms.
4. pour inférer qu'il s'agit d'un dysfonctionnement, les mots clés associés au relecteur et à la publication doivent être différents.

– Du côté du conséquent :

1. si l'antécédent étais vrai, on pourrait affirmer qu'un dysfonctionnement caractérisé par la propriété *nonQualifiedreviewer* est présent. Cette propriété a été définie dans la classe *ArgumentRelatedQoS*. Lors de son exécution, la règle montrera les noms des relecteurs qui sont dans une telle situation.

La règle nommée *ReviewerNoAuthor* est illustrée par la figure 6.12. Cette règle permet d'identifier les papiers dont le relecteur affecté est aussi un des auteurs. Ceci a été présenté comme le dysfonctionnement *III.3* dans la classification des dysfonctionnements de la QdS pour le système de gestion de conférences.

La logique derrière la définition de cette règle est comme suit :

– Du côté de l'antécédent :

1. pour chaque publication récupère le nom de ses auteurs et le nom des relecteurs qui lui ont été affectés,
2. pour inférer qu'il s'agit d'un dysfonctionnement, le nom d'un des auteurs doit être égal au nom d'un des relecteurs.

– Du côté du conséquent :

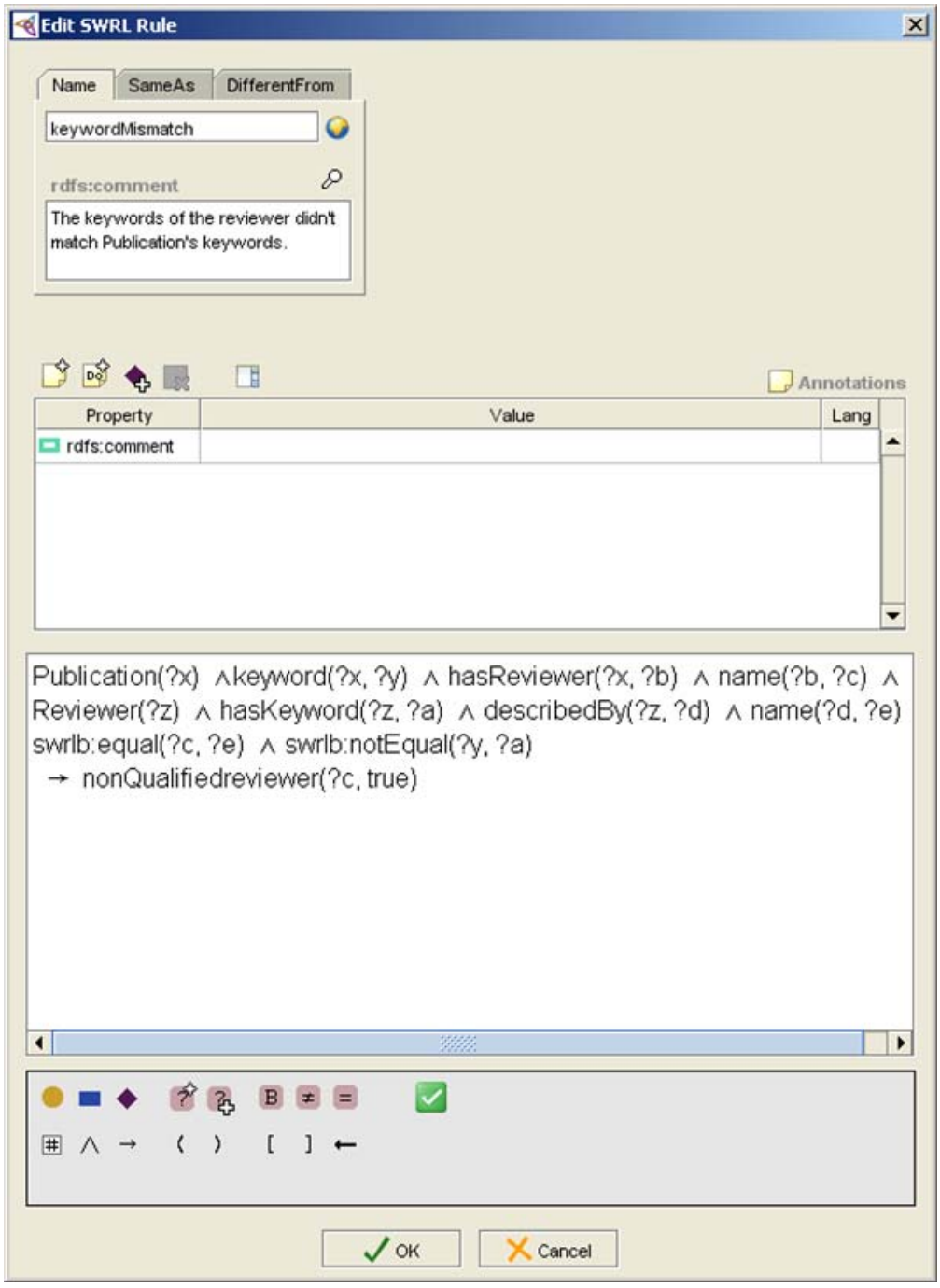


FIG. 6.11 – Dysfonctionnement dans l'affectation des papiers : relecteur dont le domaine de recherche et étranger au sujet du papier.

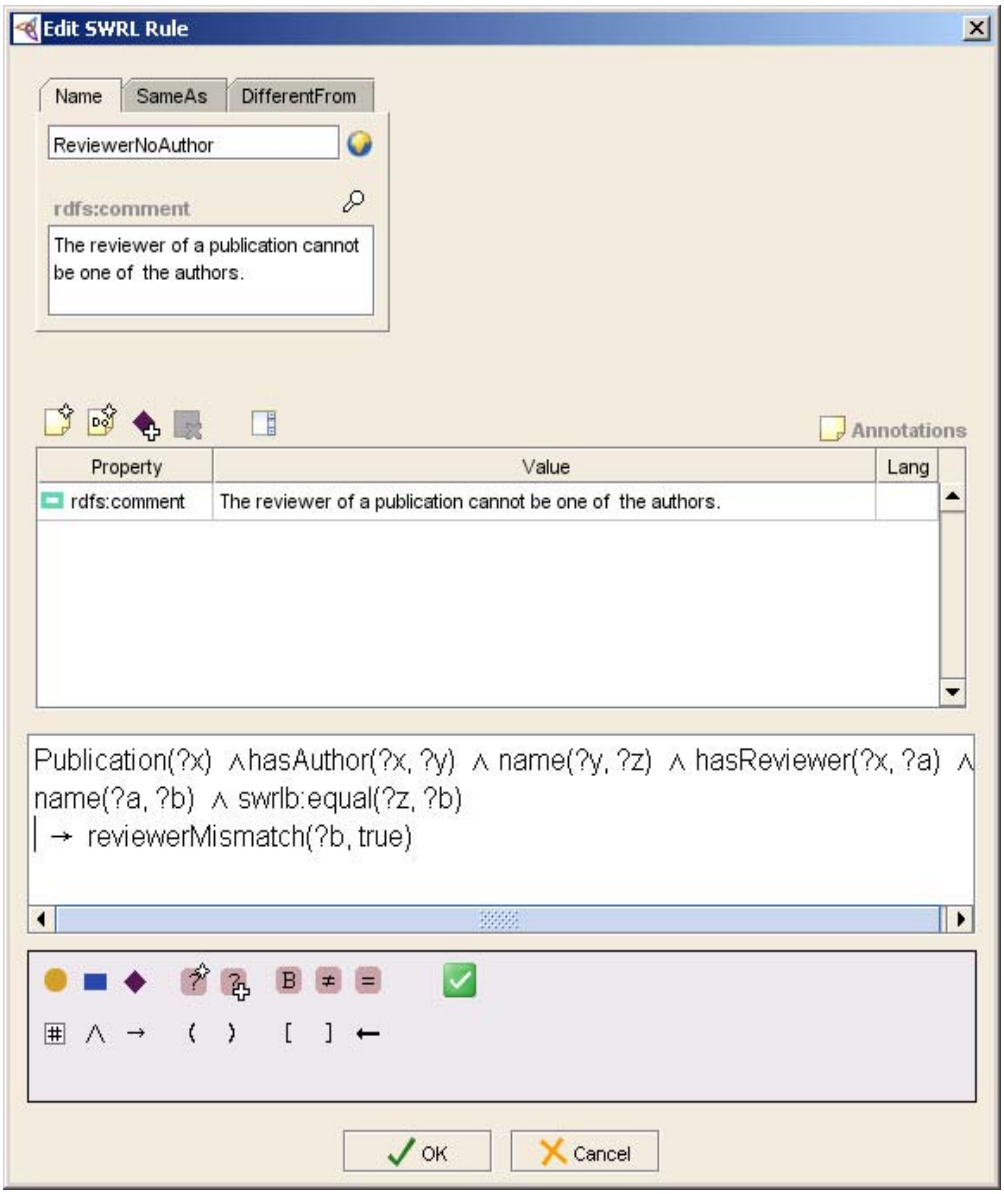


FIG. 6.12 – Dysfonctionnement dans l'affection des papiers : Papier dont le relecteur et aussi auteur.

1. si l'antécédent étais vrai, on pourrait affirmer qu'un dysfonctionnement caractérisé par la propriété *reviewerMismatch* est présent. Cette propriété a été définie dans la classe *ArgumentRelatedQoS*. Lors de son exécution, la règle montrera les noms des relecteurs qui sont dans une telle situation.

La règle nommée *SubmissionOut_Of_Date* est illustrée par la figure 6.13. Cette règle permet d'identifier les papiers qui n'ont pas respecté la date limite établie pour la soumission dans une conférence. Cette règle représente un cas générique qui pourrait se décliner dans plusieurs des cas listés dans la classification des dysfonctionnements de la QdS pour le système de gestion de conférences. En particulier nous pouvons l'associer aux cas suivants :

- *I.1* Renvoie de conférences dont le « deadline » est dépassé.
- *II.4* Pas de soumission de papier après l'inscription d'un auteur.
- *IV.1* Papier non envoyé à temps (« deadline » dépassé) aux relecteurs.
- *IV.2* Rapport non envoyé à temps (« deadline » dépassé).
- *V.1* Décision ne respectant pas le délai de temps.
- *VI.1* Non envoie de version final du papier dans la limite du temps.

La logique derrière la définition de cette règle est comme suit :

- Du côté de l'antécédent :
  1. pour chaque conférence cherche son nom et le « deadline » établi pour la soumission des papiers,
  2. pour chaque publication cherche le nom de la conférence où elle a été soumisse, et la date de soumission,
  3. du fait que les noms des conférences sont obtenus à partir de deux classes différentes (*Conference* et *Publication*), on doit vérifier qu'il s'agit de la même conférence, en comparant leurs noms.
  4. pour inférer qu'il s'agit d'un dysfonctionnement, la date où la publication a été soumisse doit être postérieure à la date limite pour la soumission des publications établie par la conférence.
- Du côté du conséquent :
  1. si l'antécédent étais vrai, on pourrait affirmer qu'un dysfonctionnement caractérisé par la propriété *outOfDayMismatch* est présent. Cette propriété a été définie dans la classe *TimeRelatedQoS*. Lors de son exécution, la règle montrera les noms des publications qui sont dans une telle situation.

Mise à part les règles pour la détection des dysfonctionnements, nous pouvons aussi définir des règles qui agissent de façon préventive. Dans ce cas, il s'agit d'actions qui pourrait être vérifiées afin d'éviter les dysfonctionnement qui amènent le système vers

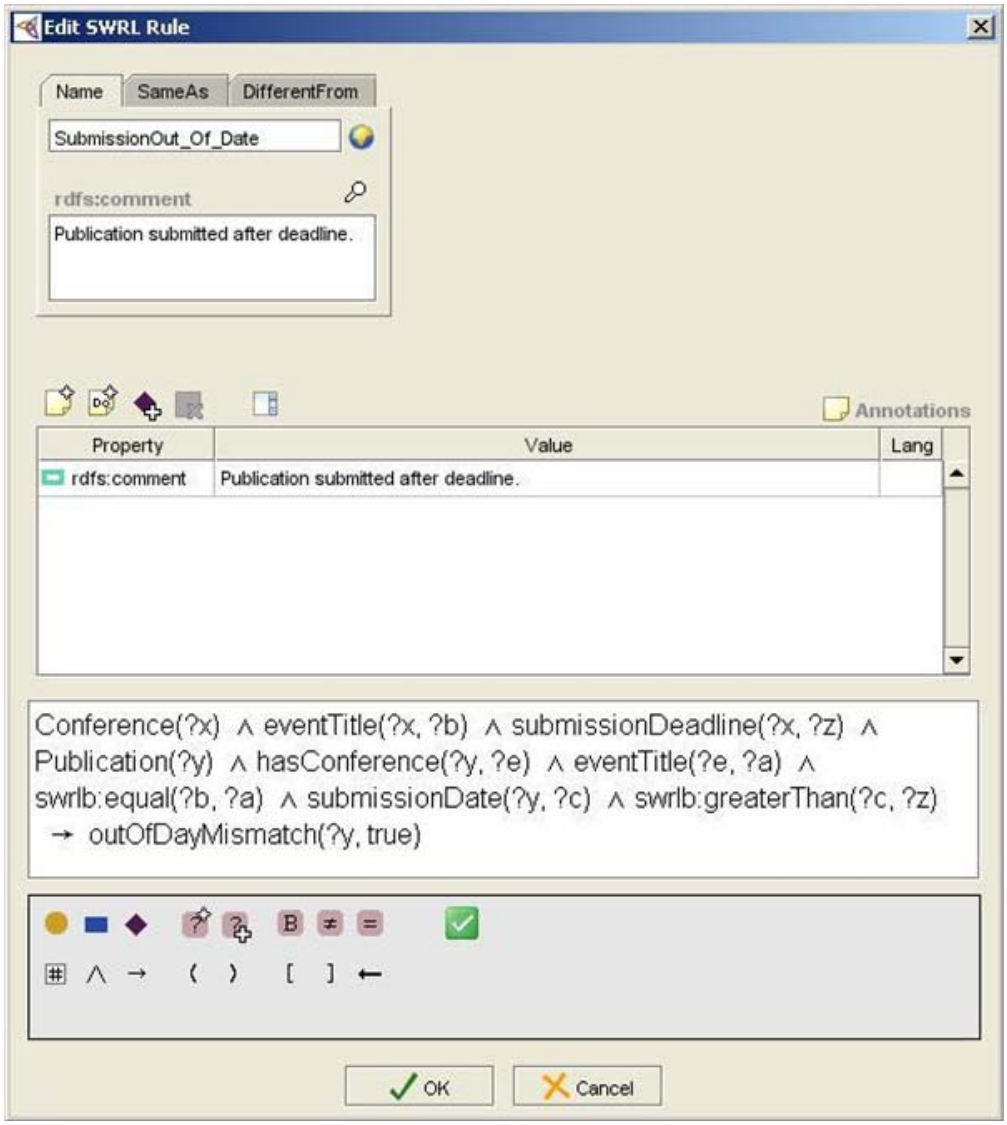


FIG. 6.13 – Dysfonctionnement dans la soumission des papiers : La date de soumission ne respect pas le « deadline ».

un état anormal où la QdS est dégradée. Ce type de règle nous l'appelons une *recommandation*.

La figure 6.14 illustre un exemple de recommandation. Afin de prévenir le dysfonctionnement où un relecteur reçoit un papier dont il est auteur, la recommandation nommée *RecommendedReviewers_conference* pourrait être utilisée. Ainsi, cette règle cherche des relecteurs ayant des intérêts de recherche proches des thèmes de la conférence.

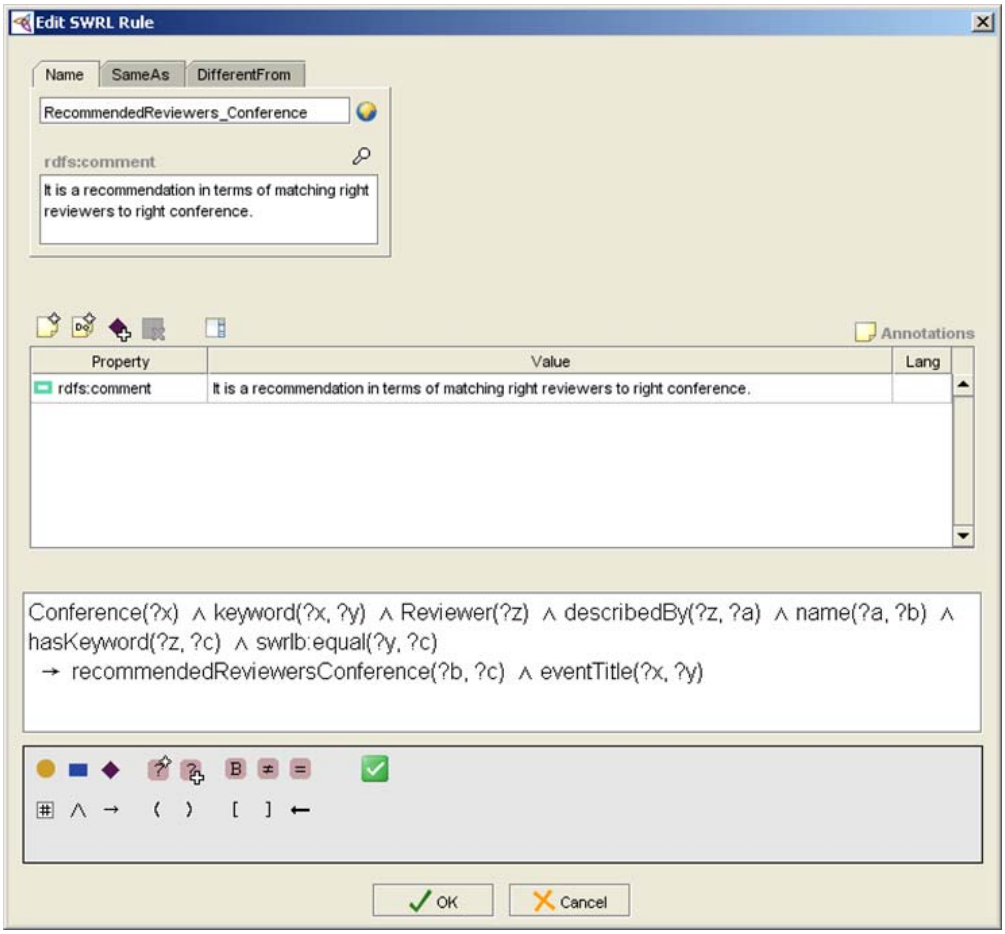


FIG. 6.14 – Recommandation de relecteurs en fonction des thèmes de la conférence.

La logique derrière la définition de cette règle est comme suit :

– Du côté de l'antécédent :

1. pour chaque conférence cherche ses mots clés identifiant ses thèmes de recherche,
2. pour chaque relecteur récupère son nom et ses mots clés identifiant ses intérêts de recherche,

3. pour inférer qu'il s'agit d'un relecteur habilité pour la conférence, les mots clés de la conférence et du relecteur doivent coïncider.

– Du côté du conséquent :

1. si l'antécédent étais vrai, on pourrait affirmer qu'un relecteur est habilité, ceci est caractérisé par la propriété *recommendedReviewers*. Cette propriété a été définie dans la classe *Conference*. Lors de son exécution, la règle montrera les noms des relecteurs, les mots clés coïncidents et les conférences où les relecteurs sont habilités.

Application des règles

Nous avons appliqué les règles introduites précédemment, un exemple des résultats est illustré dans la figure 6.15. Les résultats obtenus sont en fonction des instances définies dans la basse de connaissance de l'ontologie. Durant l'exécution l'outil Jess applique toutes les règles définies dans l'éditeur de règles, est les résultats (caractérisés par le conséquent de la règle) son affichés sous l'onglet *Asserted Properties*.

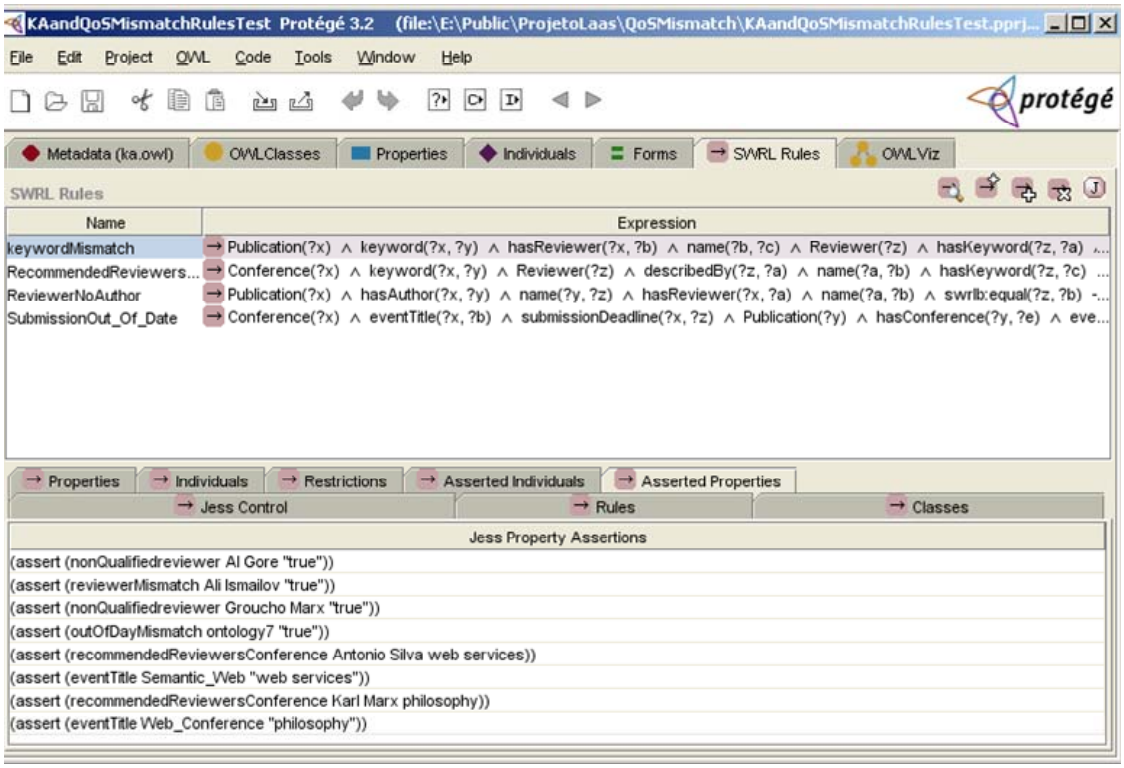


FIG. 6.15 – Exécution des règles avec Jess.

6.5 Reconfiguration au niveau classe de service

Dans ce travail, du fait que l'on traite la reconfiguration architecturale, nous nous focalisons sur la reconfiguration au niveau classe de service, par opposition à la reconfiguration au niveau instance de service (liée à la reconfiguration comportementale).

Le but étant d'offrir une infrastructure de gestion de la QdS pour les applications à base de services Web.

Les décisions de reconfiguration au niveau classe de service dépendent d'une analyse statistique des valeurs recueillies par la mesure et la surveillance des paramètres de QdS. Ces valeurs sont obtenues à partir de plusieurs interactions entre instances de services Web.

Plusieurs cas d'interaction pourrait se présenter afin d'établir une reconfiguration au niveau classe de service. Par la suite nous introduisons plusieurs scénarios possibles.

6.5.1 Cas 1. Interaction simple

Ce cas considère des conversations entre plusieurs instances d'une même paire de services Web. Plusieurs dérivations peuvent être possibles.

Conversation acyclique

Par acyclique nous voulons dire une conversation sur un même sens, comme présenté dans la figure 6.16. Normalement, les interactions dans cette schéma sont indépendantes, ainsi une surveillance locale s'avère nécessaire pour chaque paire demandeur (ang. requester) / fournisseur (ang. provider). Dans tous le cas, la surveillance est réalisée en mesurant les paramètres de QdS aussi bien du côté demandeur que du côté fournisseur.

Conversation cyclique

Par cyclique nous voulons dire une conversation dans les deux sens comme illustré par la figure 6.17. Normalement, les interactions dans ce schéma sont dépendantes,

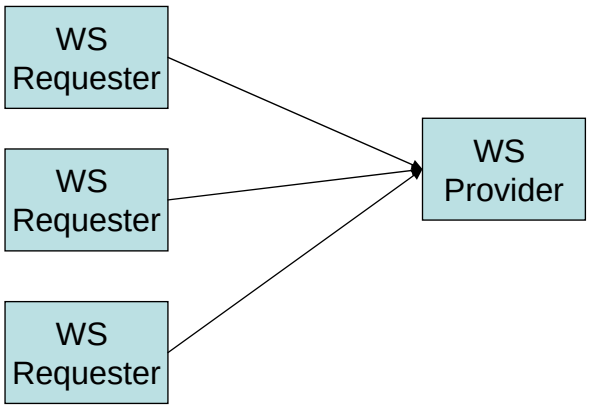


FIG. 6.16 – Interaction entre Web services simple et acyclique.

ainsi une surveillance globale s'avère nécessaire (éventuellement par rassemblement de plusieurs surveillances locales) pour l'ensemble des instances impliquées.

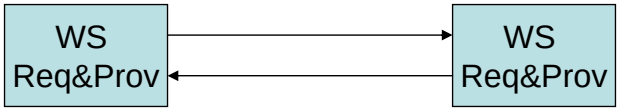


FIG. 6.17 – Interaction entre Web services simple et cyclique.

6.5.2 Cas 2. Interactions multiples

Ce cas considère des conversations entre plusieurs paires de services Web. A l'instar du cas 1, plusieurs dérivations peuvent être possibles.

Conversation acyclique

Normalement, ce type de conversation considère des interactions dépendantes, ainsi une surveillance globale s'avère nécessaire (fig. 6.18).

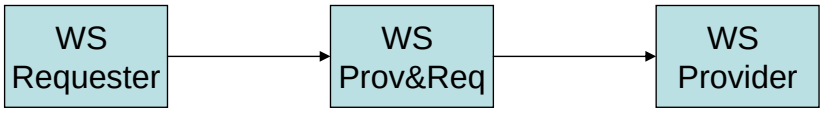


FIG. 6.18 – Interaction entre Web services multiple et acyclique.

Conversation cyclique

Normalement, les interactions dans ce type de conversation sont dépendantes, ainsi une surveillance globale s'avère nécessaire pour l'ensemble des instances de services Web impliqués (6.19).

FIG. 6.19 – Interaction entre Web services multiple et cyclique.

6.6 Architecture de gestion de la QdS

Toujours dans le but d'offrir des mécanismes pour la gestion de la QdS, nous nous sommes inspirés des travaux sur les architectures de « self-healing », pour proposer une architecture de gestion de la QdS. Cette architecture, illustrée par la figure 6.20, est décomposée en quatre modules :

1. Le module *Measurement* (MeMo),
2. Le module *Monitoring* (MoMo),
3. Le module *Diagnosis & Repair* (D&R), et
4. Le module *Reconfiguration* (ReMo).

Dans le module MeMo un intercepteur *Requester Side Intercepteur* (RSI) est défini du côté du service Web demandeur (ang. WS Requester) et un autre intercepteur *Provider Side Intercepteur* (PSI) est placé du côté du service Web fournisseur (ang. WS Provider). Ces intercepteurs ont le but d'attraper les requêtes, en provenance et à destination des services Web, de calculer et de loger les valeurs correspondant aux paramètres de QdS, obtenues à partir des interactions entre instances de services Web.

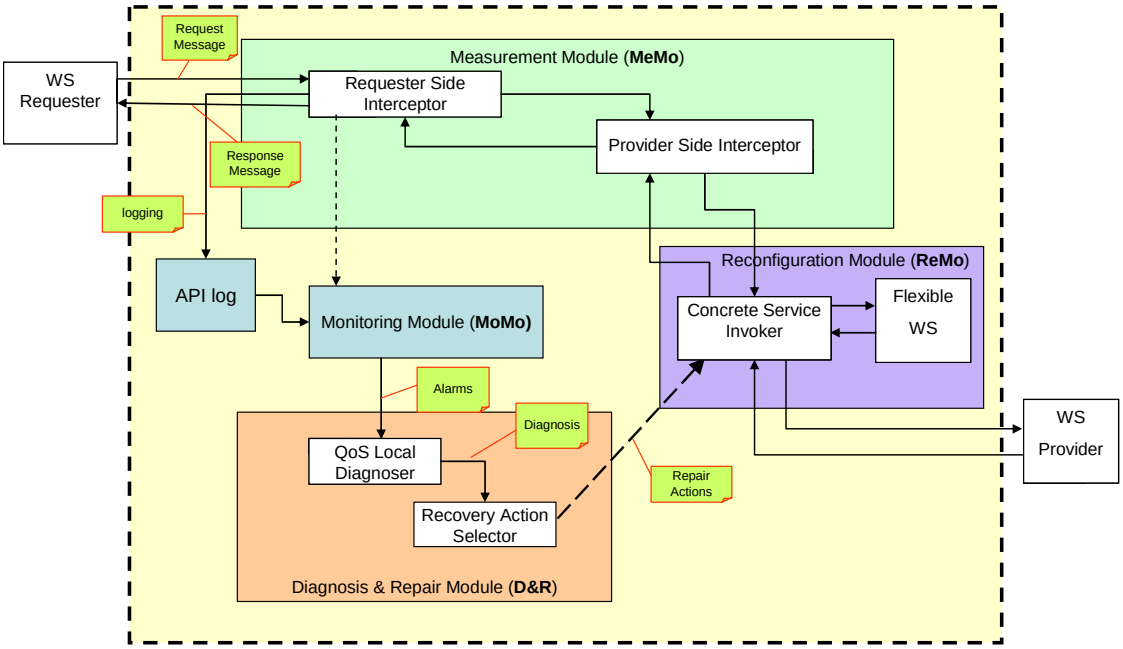


FIG. 6.20 – Architecture de gestion de la QdS.

Le module *MoMo* récupère et analyse depuis le log les valeurs de QdS calculées, afin de repérer des symptômes impliquant éventuellement une dégradation de la QdS. Ces symptômes sont notifiés au module D&R sous forme d'événements d'alarme.

Dans le module D&R les événements d'alarme sont analysés par le composant *QoS Local Diagnoser* (QLD). Dans le cas d'identification d'une dégradation de la QdS, le composant QLD notifie le type de problème au composant *Recovery Action Selector* (RAS) qui détermine les actions de reconfiguration à mettre en place par le module *ReMo*.

Le service Web *Flexible WS* correspond à un service qui offre la même signature du service Web fournisseur réel. Dans le but d'assurer une connexion avec le service Web fournisseur. Le service Web *Flexible Service* (FS) opère de la façon suivante :

Quand le PSI reçoit une requête en provenance du service Web demandeur, il lit la requête (pour des raisons de surveillance sur les paramètres de QdS), et puis il laisse passer le message vers l'intercepteur *Concrete Service Invoquer* (CSI). Le CSI prend une copie des données contenue dans la requête (input du service Web), et ensuite :

1. Il laisse passer la requête vers le FS,
2. Il construit une nouvelle requête vers le service Web fournisseur, avec les paramètres interceptés du message se dirigeant vers le FS.

Ensuite, le CSI attend la réception du message de retour du FS, et le résultat d'invocation du service Web fournisseur. Lorsqu'il les reçoit tous les deux, il met le résultat du service Web fournisseur dans le message provenant du FS, et il laisse passer le message de retour à destination finale du service Web demandeur. Et comme ça, le service Web demandeur envoie une requête vers le FS et reçoit une réponse du service Web fournisseur.

6.6.1 Modélisation de l'architecture de gestion de la QdS

Dans le but de valider l'architecture de gestion de la QdS proposée, nous l'avons modélisée avec le langage UML. De manière plus précise, nous avons utilisé le profil UML/SDL de l'outil Tau G2 de la société Telelogic[6]. Cet outil permet de générer des simulations qui permettent d'observer si le comportement déclaré dans la spécification du système correspond au comportement affiché par la simulation. Pour générer des simulations avec TAU nous devons suivre la démarche suivante :

1. Créer des diagrammes de classe pour les composants du système,
2. Créer des diagrammes de classe caractérisant les messages qui seront échangés par les composants du système,
3. Créer des diagrammes de structure composite caractérisant les modules composant le système.
4. Créer des diagrammes de machine à état caractérisant le comportement des composants du système.

Une fois tous ces éléments réunis, la simulation peut se déclencher. Le résultat de la simulation peut s'afficher de formes diverses, pour en faire, nous avons privilégié la vue de diagrammes de séquence.

Diagrammes de classe

La figure 6.21 illustre les classes caractérisant les composants qui participent dans l'architecture. La classe *QoSMgmt* représente l'architecture générale, elle a des relations de compositions avec chacun des modules définissant l'architecture. Pour la définition de la simulation, présentée plus tard, nous avons des classes représentant les services

[6]http ://www.telelogic.com/

Web demandeur (WSReq) et fournisseur (WSProv, WSProv2). Les interfaces *FromUser, ToUser* correspondent aux points de communication du système avec l'environnement (i.e. l'utilisateur externe du système). Le commentaire placé en bas de la même figure, représente une initialisation de variable qui prend du sens lors de l'exécution de la simulation.

Un autre diagramme de classe a été défini pour la représentation des messages qui seront échangés par les composants du système. L'application de ces messages sera illustrée pendant la phase de simulation de l'architecture. Afin de ne pas encombrer d'avantage cette section on a placé la figure B.1 correspondante à ce diagramme dans l'annexe B.

Diagrammes de structure composite

Les diagrammes de structure composite ont pour but de définir les relations entre les composants définissant un module. D'abord, nous avons intégré tous les modules composant l'architecture de gestion de la QdS, et les composants interagissant avec ces modules, dans un seul diagramme de structure composite global, ceci est illustré par la figure 6.22.

Les interactions entre modules et entre composants et modules, sont faites par le biais des ports et des connecteurs. Dans chaque port on définit les messages qui sont attendus en entrée et en sortie, ces messages doivent correspondre à des messages spécifiés dans le diagramme de classe de la figure B.1.

Pour chaque module composé de plusieurs composants, et défini dans le diagramme de structure composite global, nous avons détaillé sa composition interne, donc on a défini de nouveaux diagrammes spécifiant les composants et leurs relations (par des ports et des connecteurs). Les diagrammes de structure composite concernant les modules *Measurement*, *Diagnosis&Repair* et *Reconfiguration* sont illustrés successivement par les figures 6.23, 6.24 et 6.25.

Les diagrammes de machine à état correspondant au comportement des composants sont placés dans l'annexe B.

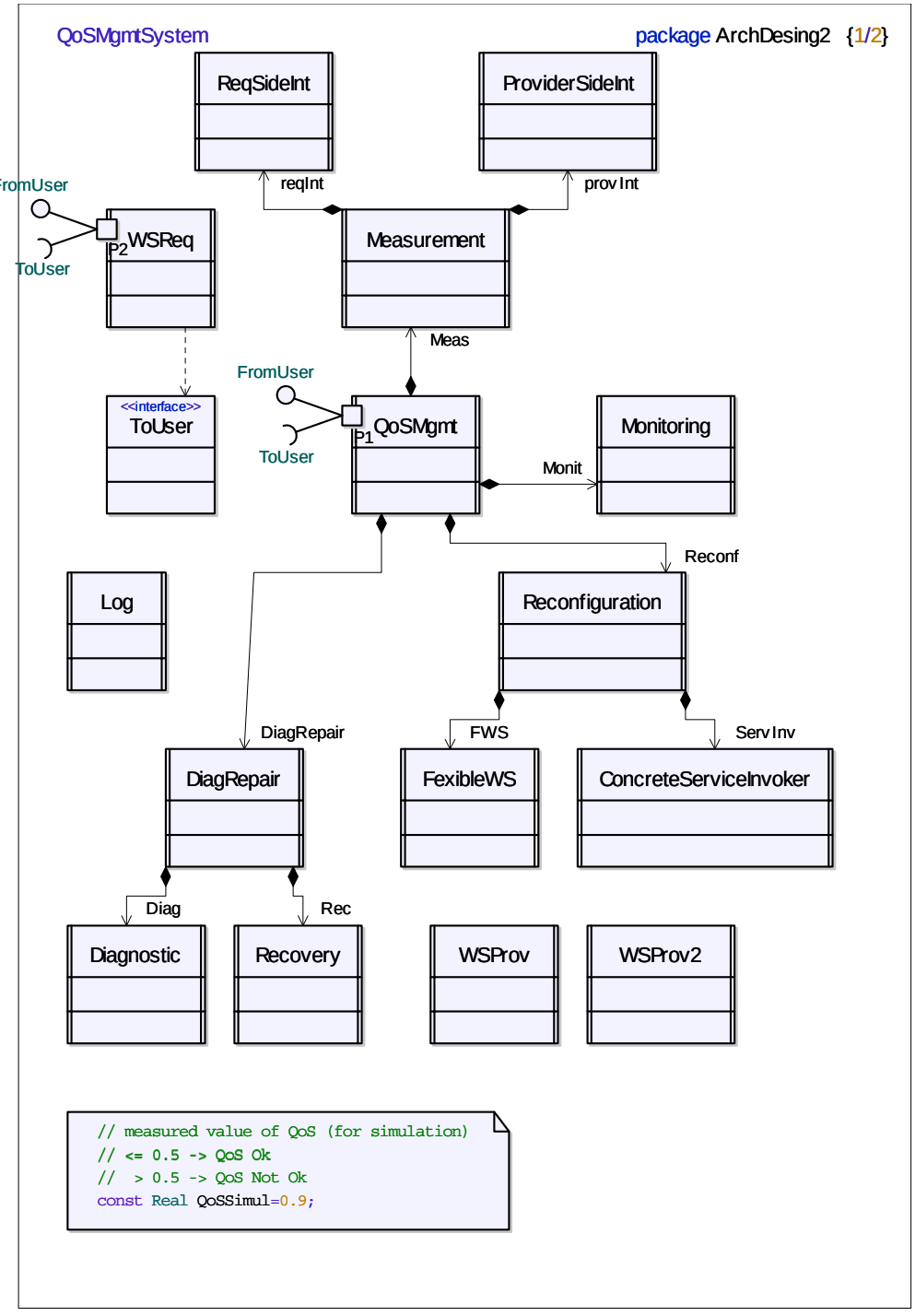


FIG. 6.21 – Diagramme de classes.

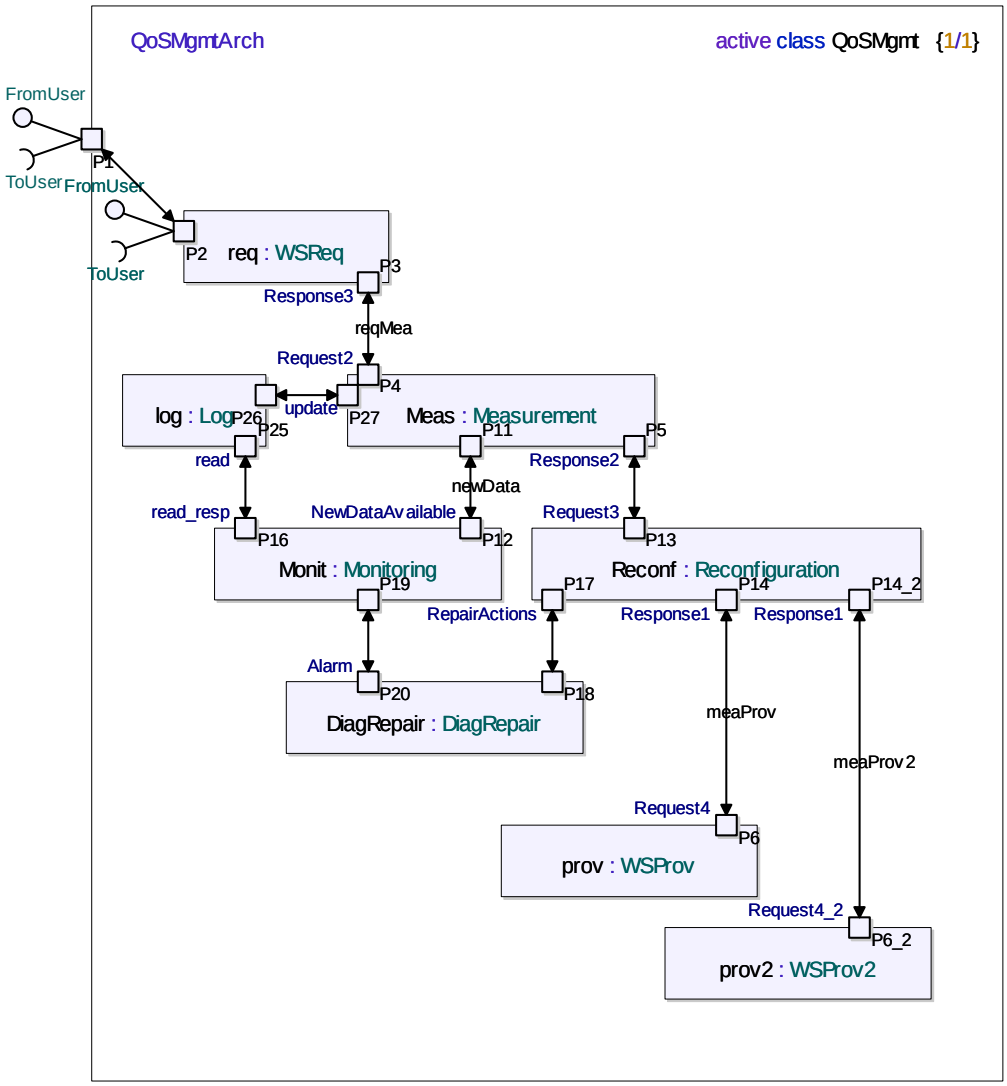


FIG. 6.22 – Diagramme de structure composite global.

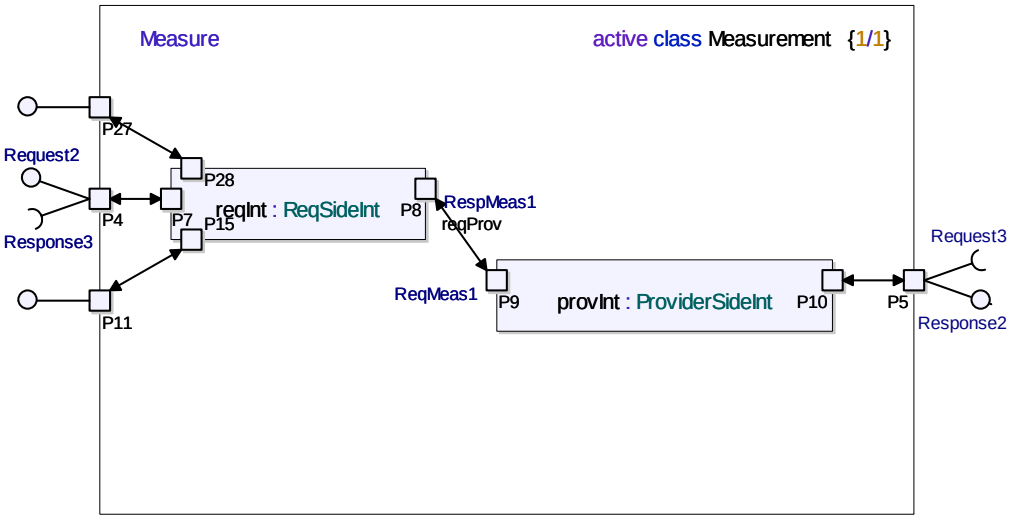


FIG. 6.23 – Diagramme de structures composites pour le module *Measurement*.

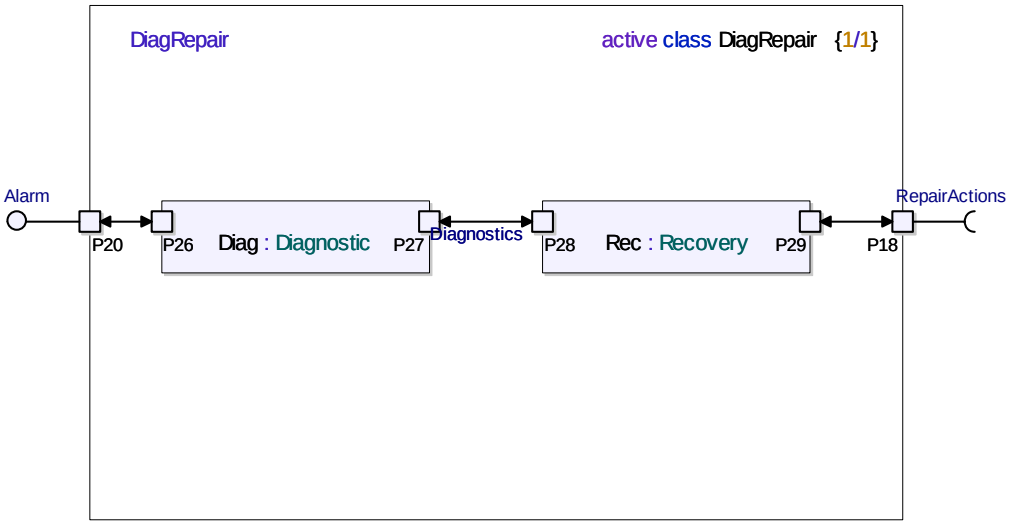


FIG. 6.24 – Diagramme de structure composite pour le module *Diagnosis&Repair*.

6.6.2 Validation de l'architecture de gestion de la QdS

Comme annoncé précédemment, nous avons fait des simulations avec l'outil TAU afin de valider l'architecture de gestion de la QdS proposée. Par la suite, on expliquera les résultats de cette simulation par le biais d'un scénario d'utilisation. Le cas d'utilisation illustre le comportement des composants du système, lorsque la QdS est dégradée et donc une action de substitution d'une instance de service Web par une autre s'avère nécessaire. Pour des soucis de clarté on décomposera la simulation en trois phases.

Première phase de la simulation

Dans la première phase (figure 6.26) un utilisateur (actor) fait une requête au travers d'un service Web (req). La requête se dirigeant vers un service Web fournisseur, est interceptée, d'abord pour l'intercepteur côté demandeur *reqInt*, puis par l'intercepteur côté fournisseur *provInt* et enfin pour l'intercepteur *Concrete Service Invoker* (ServInv). Ensuite, la requête est adressée au *Flexible Service* (FWS) et au service Web fournisseur (prov). L'intercepteur CSI attend la réponse des deux services invoqués et puis laisse passer la réponse vers l'intercepteur côté fournisseur *provInt*, qui ensuite la renvoie vers l'intercepteur côté demandeur *reqInt*. A ce moment, l'intercepteur côté demandeur *reqInt* laisse passer la réponse vers le service Web demandeur (req), ensuite il renvoie les résultats de ses calculs au log, et puis envoie un message vers le composant *Monitoring*

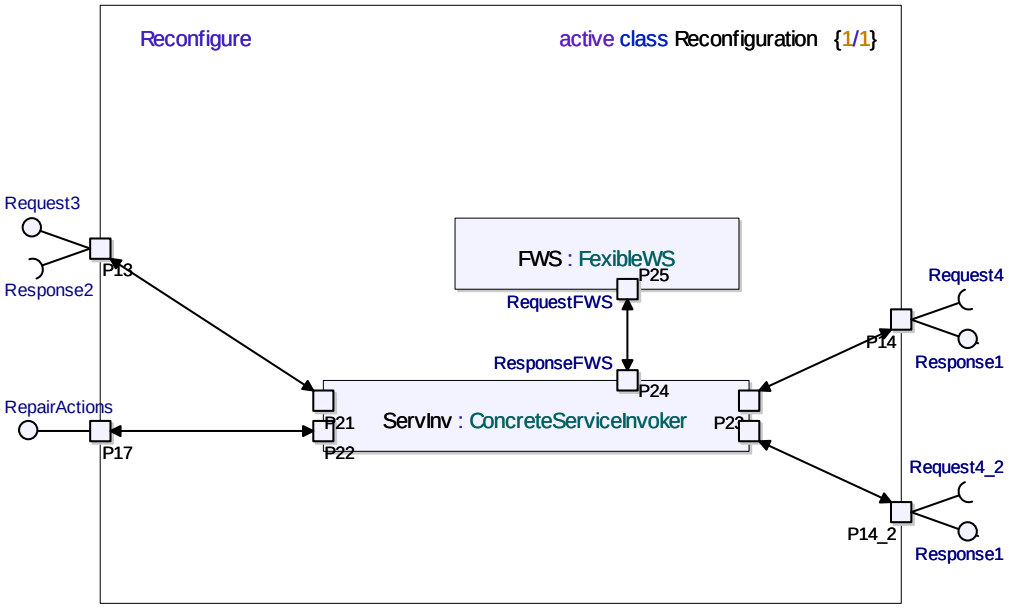


FIG. 6.25 – Diagramme de structure composite pour le module *Reconfiguration*.

(Monit) en indiquant qu'une nouvelle valeur a rentrée dans le log. Le message qui arrive au service Web demandeur et enfin délivré à l'utilisateur (actor).

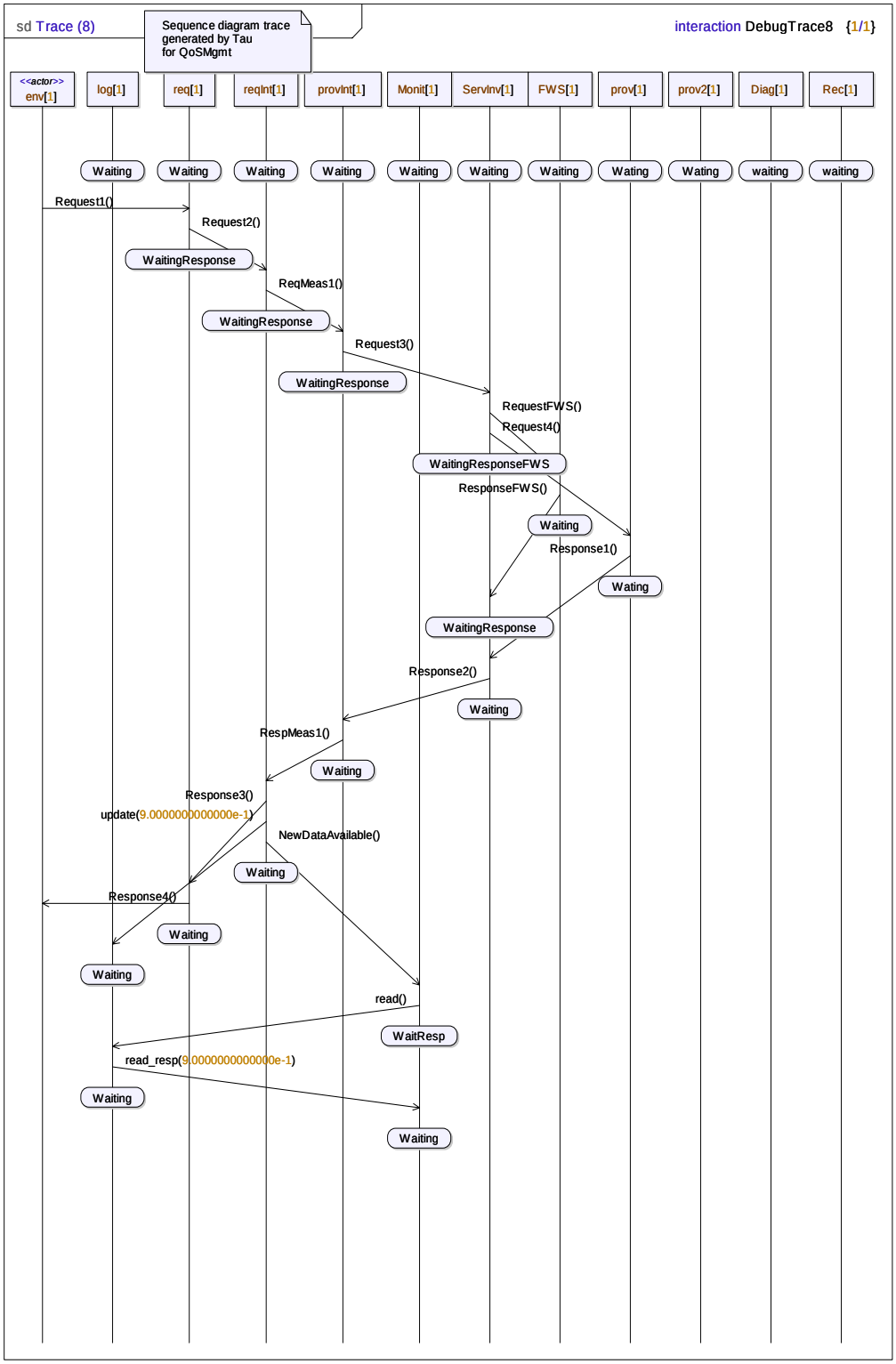


FIG. 6.26 – Simulation de l'architecture de gestion de la QdS (1/3).

Par rapport aux mesures des paramètres de QdS, chaque passage d'une requête vers les intercepteurs du module *Measurement* représente une copie des données et puis des calculs pour mesurer l'état de la QdS. Les résultats de ses calculs seront stockés dans le log. Ainsi, le composant Monitoring (Monit) récupère les données du log qui seront ensuite analysés.

Deuxième phase de la simulation

La deuxième phase de la simulation (6.27) représente des nouvelles requêtes entre services Web de façon similaire à celle décrite dans la première phase. Par contre, cette fois lorsque le composant *Monitoring* analyse les données stockées dans le log, il aperçoit des symptômes qui pourrait entraîner une dégradation de la QdS. Donc, le *Monitoring* adresse une alarme vers le composant de diagnostic (Diag). Ensuite, ce composant détermine qu'une dégradation de la QdS a eu lieu, et puis communique la détection de ce problème au composant *Recovery* (Rec). Ensuite, le composant *Recovery* décide les actions de reconfiguration à prendre, en l'occurence, la substitution de l'instance du service Web fournisseur *prov*, pour l'instance *prov2*. A ce moment, l'intercepteur *Concrete Service Invoker* (ServInv) reçoit l'ordre d'appliquer cette action de reconfiguration.

Troisième phase de la simulation

Dans la troisième phase de simulation, quand de nouvelles requêtes de l'utilisateur sont générées, celles-ci seront adressées vers l'instance du service fournisseur *prov2*, et non plus vers l'instance *prov*. Cette dernière phase est illustrée par la figure 6.28.

6.7 Conclusion

Dans ce chapitre nous avons proposée une approche pour la gestion de la QdS pour les applications à base de services Web. Cette approche a été basée sur la définition d'une classification des dysfonctionnements qui ont été caractérisés, pour son traitement, sous forme d'ontologies. Nous avons validé l'ontologie en l'appliquant au système de gestion de conférences. Les règles définies pour la détection et la prévention des dysfonctionnements, traitent des cas génériques, ainsi elles couvrent un nombre important des dysfonctionnements affectant la QdS pour le système de gestion de conférences.

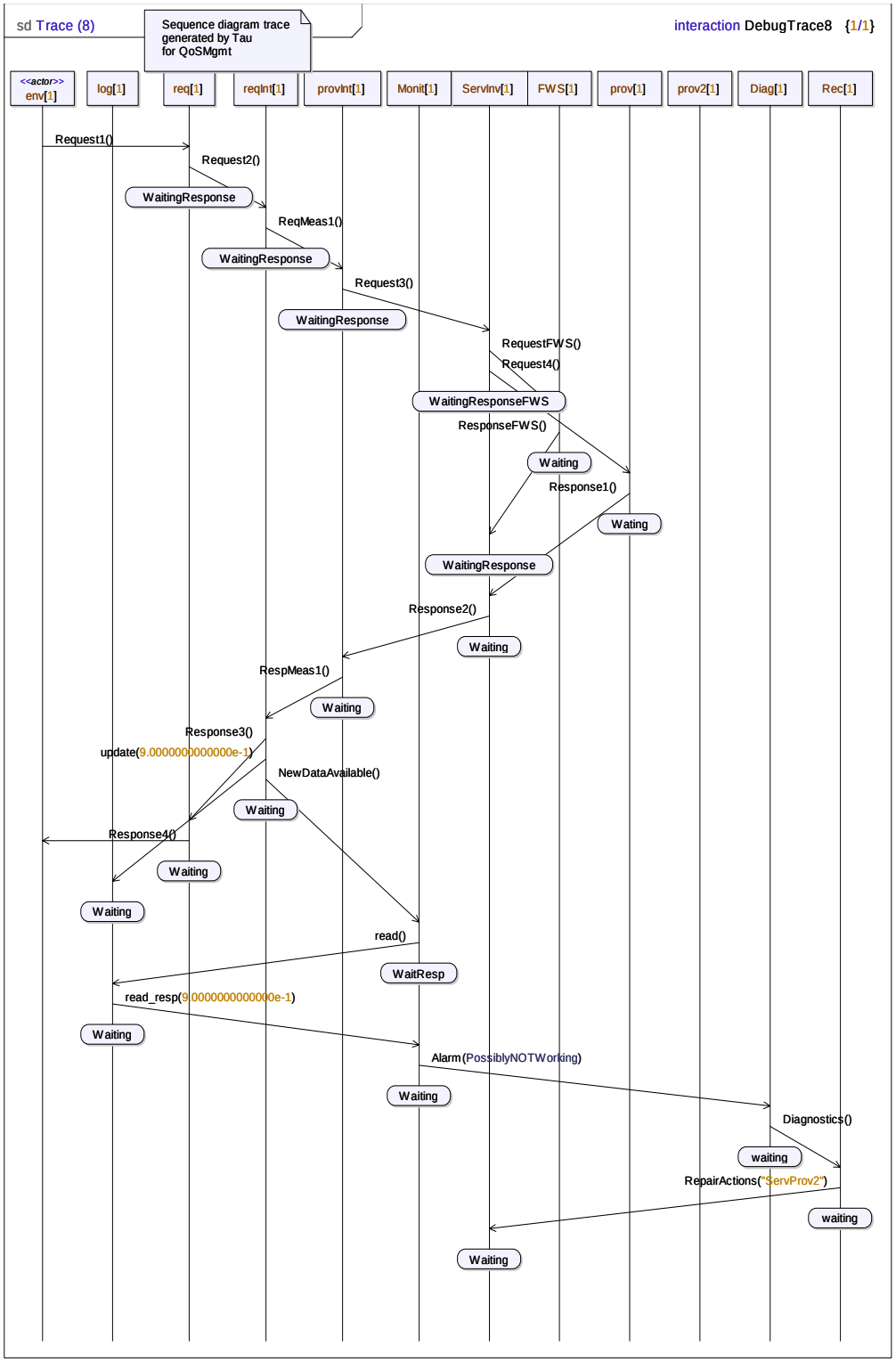


FIG. 6.27 – Simulation de l'architecture de gestion de la QdS (2/3).

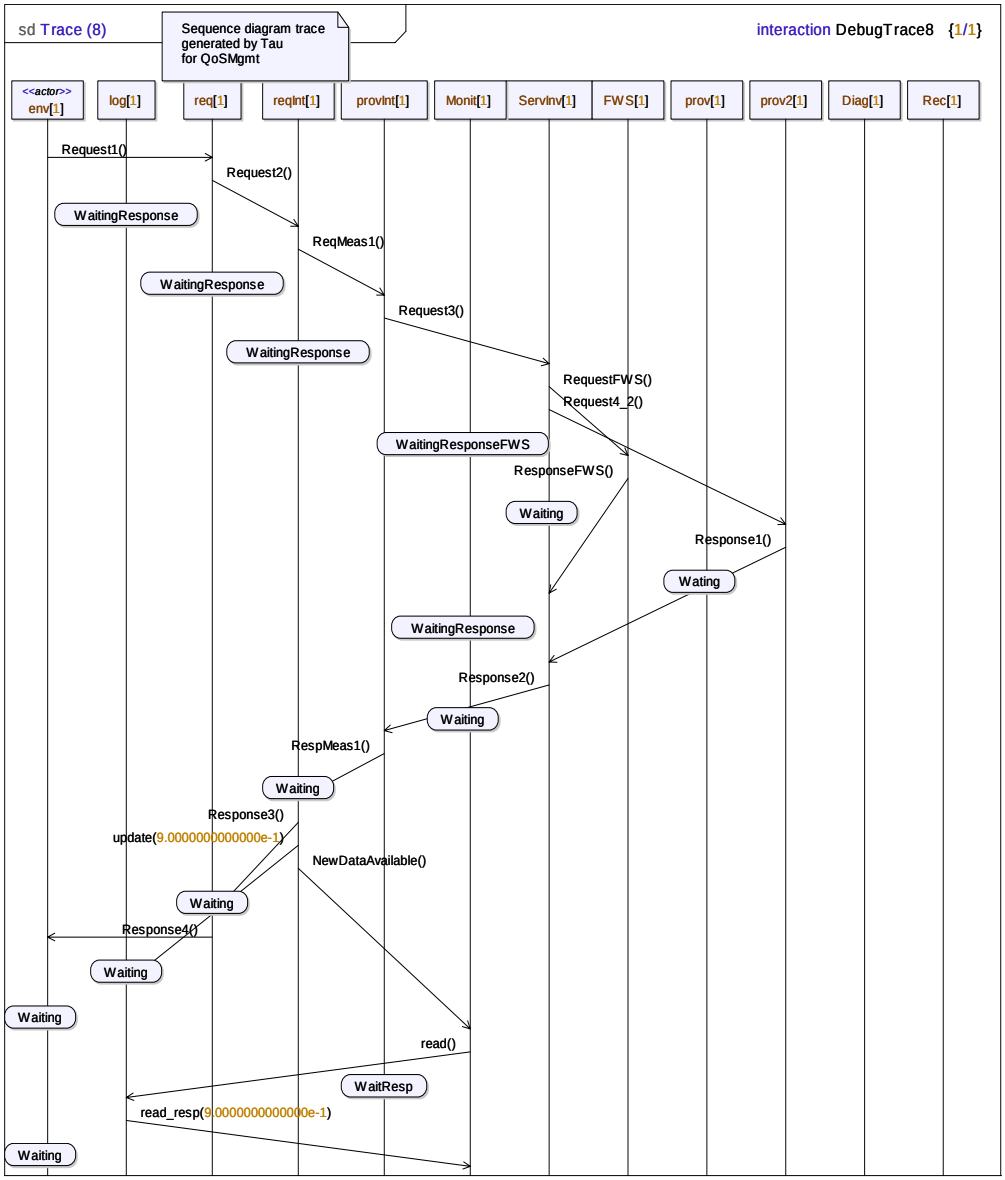


FIG. 6.28 – Simulation de l'architecture de gestion de la QdS (3/3).

L'architecture proposée pour la gestion de la QdS, reprend des éléments identifiés dans les systèmes qui suivent une approche de « self-healing ». Cette architecture représente un intérêt particulier pour le projet WS-DIAMOND, et elle est en cours de développement par le LAAS-CNRS et l'Université de Sfax, en Tunisie.

Chapitre 7

Conclusion

7.1 Bilan des contributions

Dans ce mémoire nous avons présenté l'ensemble des travaux réalisés dans le cadre de cette thèse. Le manuscrit a été organisé en deux parties.

La première partie a été composée de deux chapitres qui décrivent l'état de l'art sur les thèmes de recherche abordés par cette thèse. Le chapitre 2 a introduit les architectures logicielles et ses principales techniques de modélisation, par les ADL et par UML, et ses adaptations. Nous avons aussi abordé le cas des architectures orientées services et leur application particulière aux services Web. Nous avons terminé ce chapitre par la présentation des architectures dynamiques et par leur modélisation avec les ADL. Le chapitre 3 traite de l'adaptabilité des architectures logicielles. Nous avons présenté les travaux sur la reconfiguration des architectures, au moyen d'UML et des techniques de réécriture de graphes. Enfin, nous avons présenté un bilan des travaux qui proposent des approches diverses pour les systèmes de « self-healing ».

La deuxième partie a été composée de trois chapitres qui présentent les contributions apportées par nos travaux de thèse. Le chapitre 4 propose un cadre logiciel pour l'adaptation des architectures des applications à base de services Web. La proposition représente une approche originale visant l'automatisation du processus de gestion des architectures par des actions de reconfiguration. De même, notre approche associe de façon originale, des techniques semi-formelles (UML), mais amplement utilisées ; avec des techniques formelles (technique de réécriture de graphes) qui sont moins connues dans ce domaine d'application. La description des architectures par les types structuraux, a permis un niveau d'abstraction adéquat, qui autorise la construction des règles

de base simples, et qui applique les techniques de réécriture de graphes de façon performante. Le processus d'automatisation de notre approche est partiellement achevé. Une traduction des diagrammes UML par le biais des plugins de l'outil Fujaba vers le langage XML a été développée dans le cadre d'une collaboration avec l'Université de Sfax en Tunisie. Le chapitre 5 a présenté deux scénarios applicatifs validant l'approche de description et de reconfiguration des architectures. Ces scénarios traitent deux niveaux de spécification différents et complémentaires. Le scénario du système de gestion de conférences adopte une approche coopérative, en se focalisant sur les interactions externes des services Web. L'exemple du « Foodshopping » adopte une approche comportementale, en se focalisant sur la description du comportement interne des services Web. Les deux scénarios ont été éprouvés dans le cadre du projet européen WS-DIAMOND et font partie du Deliverable 1.1. Les sections concernant les règles de reconfiguration font partie du Deliverable 3.1 du même projet. La mise en œuvre du système de gestion de conférences a été réalisée dans le cadre d'une collaboration avec l'Université de Sfax, en Tunisie. Le chapitre 6 a proposé une approche pour la gestion de la QdS pour les applications à base de services Web. Cette approche a défini une classification des dysfonctionnements affectant la QdS des applications. Cette classification a été caractérisée, pour son traitement, par des ontologies. L'ontologie a été validée et appliquée au système de gestion de conférences. Les règles définies pour la détection et la prévention des dysfonctionnements, traitent des cas génériques, ainsi elles couvrent un nombre important des dysfonctionnements affectant la QdS pour le système de gestion de conférences. L'architecture proposée pour la gestion de la QdS, reprend des éléments identifiés dans les systèmes qui suivent une approche de « self-healing ».

7.2 Travaux en cours et perspectives

Plusieurs axes dérivent par la suite de nos travaux, tant de niveau théorique que de niveau pratique.

Nous projetons de poursuivre le développement du processus d'automatisation de l'approche d'adaptation des architectures par UML et XML, et du moteur de transformation de graphes. Par cette intégration, les résultats générés par l'outil de graphes, à l'issue de l'application des règles de reconfiguration, sont redirigés vers une API de gestion d'applications à base de services Web. L'architecture envisagée pour l'API suit une approche à base de canaux à événements. Le but est d'avoir un outil complet pour l'analyse des architectures dynamiques et adaptables par intégration des diverses techniques. A moyen terme nous comptons enrichir les types architecturaux (le Rôle coopératif, la Catégorie de service et la Classe de service) qui structurent les architectures des appli-

cations à base de service Web. L'idée est d'associer des propriétés sémantiques à chacun des types. Ces propriétés peuvent servir par exemple, pour affiner le choix entre services Web, lorsque l'on applique des actions de reconfiguration. Ce dernier point pourrait être réalisé par le biais des ontologies et en utilisant les techniques d'annotation issues du Web sémantique.

Il s'avère nécessaire d'étendre la gestion de la QdS, définie par les ontologies, en suivant l'approche de QdS par objectif, où les services Web demandeur et fournisseur se mettent d'accord à partir de contrats de QdS indiquant le niveau de QdS à maintenir. Par rapport au système de gestion de conférences, d'autres paramètres pourraient être pris en compte. Par exemple, le niveau de qualité des actes lors de la phase de recherche de conférences, ou le nombre de publications, dans revues de prestige, lors de la phase de recherche de relecteurs. De même, nous envisageons de définir des politiques de détection et de prévention de la dégradation de la QdS pour leur application au système de « Foodshopping ».

Dans le cadre du projet WS-DIAMOND, d'autres équipes développent des mécanismes pour le diagnostic et la reconfiguration de services Web au niveau comportemental. Dans l'approche comportementale des actions de reconfiguration sont déclenchées lorsqu'un dysfonctionnement est détecté sur une instance d'un service Web. Ces mécanismes sont considérés complémentaires à notre approche de reconfiguration architecturale. Un exemple de ces mécanismes considère la compensation (retour en arrière vers un état sûr du système et reprise par des routes alternes) dans le « workflow » de l'application. Des efforts de recherche sont en cours pour la composition des deux approches aux niveaux du diagnostic et de la reconfiguration. Une première idée consiste à mettre les mécanismes comportementaux plus rapidement en application, et lorsque cela ne suffit pas, basculer vers les mécanismes architecturaux. Dans la même idée, l'architecture de gestion de la QdS représente un intérêt particulier pour le même projet. Elle est en cours de développement par le LAAS-CNRS et l'Université de Sfax, en Tunisie. A l'heure actuelle, une première version de la mise en œuvre des modules *Meaurement* et *Reconfiguration* a été réalisée. Cette implémentation a été testée sur la grille G5000.

Bibliographie

Abi-Antoun, M. et Medvidovic, N. (1999). Enabling the refinement of a software architecture into a design. *Dans Proceeding of the Second International Conference on The Unified Modeling Language*. Springer-Verlag.

Allen, R., Douence, R. et Garlan, D. (1998). Specifying and analyzing dynamic software architectures. *Dans Proceedings of the Conference on Fundamental Approaches to Software Engineering*, Lisbon, Portugal.

Allen, R. et Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249.

Allen, R., Garlan, D. et Douence, R. (1997). Specifying dynamism in software architectures. *Dans Proceedings of the Workshop on Foundations of Component-Based Software Engineering*, Zurich, Switzerland.

Allen, R. J. (1997). *A Formal Approach to Software Architecture*. Ph.d. thesis, Carnegie Mellon University, School of Computer Science.

Baresi, L., Heckel, R., Thöne, S. et Varró, D. (2003). Modeling and validation of service-oriented architectures : application vs. style. *Dans ESEC / SIGSOFT FSE*, pages 68–77.

Baurens, B., Cambou, B., Drira, K., Molina-Espinosa, J. et Nabuco, O. (2001). Dse v1 integrated implementation report. Rapport LAAS 01280, LAAS-CNRS. Project IST-1999-10302.

Berners-Lee, T., Hendler, J. et Lassil, O. (2001). The semantic web. *Scientific American*.

Birman, K., van Renesse, R. et Vogels, W. (2004). Adding high availability and autonomic behavior to web services. *Dans ICSE '04 : Proceedings of the 26th International Conference on Software Engineering*, pages 17–26, Washington, DC, USA. IEEE Computer Society.

Booch, G. (1993). *Object-oriented Analysis and Design with Applications, 2nd edition.* Benjamin Cummings, Redwood City.

Brown, A. W. (1996). *Component-based software engineering : selected papers from the Software Engineering Institute.* Wiley-IEEE Computer Society Press, Los Alamitos, USA.

Chaudhri, V. K., Farquhar, A., Fikes, R., Karp, P. D. et and, J. P. R. (1998). Open knowledge base connectivity 2.0.3. Rapport technique, Stanford University and SRI International.

Dashofy, E., Hoek, A. et Taylor, R. (2001). A highly-extensible, xml-based architecture description language. *Dans Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA)*, Amsterdam,NL.

Dashofy, E. et van der Hoek, A. (2001). Representing product family architectures in an extensible architecture description language. *Dans Proceedings of the International Workshop on Product Family Engineering*, Bilbao, Spain.

Dashofy, E. M., van der Hoek, A. et Taylor, R. N. (2002). Towards architecture-based self-healing systems. *Dans WOSS '02 : Proceedings of the first workshop on Self-healing systems*, pages 21–26, New York, NY, USA. ACM Press.

Dijkstra, E. (1968). The structure of the the multiprogramming system. *Communications of the ACM*, 11(5):341–346.

Dill, D. L., Drexler, A. J., Hu, A. J. et Yang, C. H. (1992). Protocol verification as a hardware design aid. *Dans IEEE International Conference on Computer Design : VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society.

Egyed, A. et Medvidovic, N. (2001). Consistent architectural refinement and evolution using the unified modeling language. *Dans Proceeding of the 1st Workshop on Describing Software Architecture with UML*, pages 83–87, Toronto, Canada.

Garlan, D., Allen, R. et Ockerbloom, J. (1994). Exploiting style in architectural design. *Dans Proceedings SIGSOFT '94 Symposium on the Foundations of Software Engineering.*

Garlan, D., Cheng, S.-W. et Kompanek, A. J. (2002). Reconciling the needs of architectural description with object-modeling notations. *Sci. Comput. Program.*, 44(1):23–49.

Garlan, D., Monroe, R. T. et Wile, D. (2000). Acme : Architectural description of component-based systems. *Dans* Leavens, G. T. et Sitaraman, M., éditeurs : *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press.

Garlan, D. et Schmerl, B. (2002). Model-based adaptation for self-healing systems. *Dans WOSS '02 : Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA. ACM Press.

Garlan, D., Schmerl, B. et Chang, J. (2001). Using gauges for architecture-based monitoring and adaptation. *Dans Proceedings of a Working Conference on Complex and Dynamic Systems Architecture*.

Garlan, D. et Shaw, M. (1993). An introduction to software architecture. *Dans* V.Ambriola et G.Tortora, éditeurs : *Advances in Software Engineering and Knowledge Engineering*, volume I, New Jersey, USA. World Scientific Publishing Company.

Ginsberg, M. L. (1991). Knowledge interchange format : the KIF of death. *AI Magazine*, 12(3):57–63.

Gomaa, H. et Wijesekera (2001). The role of uml, ocl and adls in software architecture. *Dans Proceeding of the Workshop on Describing Software Architecture with UML*, Toronto, Canada.

Group, T. O. (2000). Architecture description markup language (adml). Rapport technique 1.

Gruber, T. (1995). Toward principles for the design of ontologies used for knowledge sharing. *International Journal Human-Computer Studies*, 43(5-6):907–928.

Guennoun, K. (2006). *Architectures Dynamiques dans le Contexte des Applications à Base de Composants et Orientées Services*. Thèse de doctorat, Université TOULOUSE III (PAUL SABATIER).

Guennoun, K., Drira, K. et Diaz, M. (2004). A proved component-oriented approach for managing dynamic software architectures. *Dans 7th iasted international conference on software engineering and application*.

Gurguis, S. A. et Zeid, A. (2005). Towards autonomic web services : achieving self-healing using web services. *Dans DEAS '05 : Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–5, New York, NY, USA. ACM Press.

Hirsch, D., Inverardi, P. et Montanari., U. (1999). Modeling software architectures and styles with graph grammars and constraint solving. *Dans The 1st Working IFIP Conference on Software Architecture*, pages 127–142. Kluwer.

Hoare, C. (1985). *Communicating Sequential Processes*. Prentice Hall.

Hofmeister, C., Nord, R. et Soni, D. (1999). Describing software architecture with uml. *Dans Proceeding of the First Working IFIP Conf. on Software Architecture*, San Antonio,TX. IEEE.

IEEE (2000). *Recommended Practice for Architectural Description of Software-Intensive Systems*. IEEE Press.

IST (2006). The web services diagnosability, monitoring and diagnostic project. http ://wsdiamond.di.unito.it.

Jacobson, I. (1992). *Object-Oriented Software Engineering : A Use Case Driven Approach, 1st edition*. Addison-Wesley.

Kephart, J. O. et Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

Kruchten, P., Obbink, H. et Stafford, J. (2006). The past, present, and future for software architecture. *Software, IEEE*, 23(2):22–30.

Luckham, D. C., Kenney, J. L., Augustin, L. M., Vera, J., Bryan, D. et Mann, W. (1995). Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355.

Magee, J., Dulay, N., Eisenbach, S. et Kramer, J. (1995). Specifying distributed software architectures. *Dans Proceeding of the 5th European Software Engineering Conference, ESEC '95*.

Magee, J., Dulay, N. et Kramer, J. (1994). A constructive development environment for parallel and distributed programs. *Dans Proc. of the IEEE International Workshop on Configurable Distributed Systems*, Pittsburgh PA, USA.

Magee, J. et Kramer, J. (1996). Dynamic structure in software architectures. *Dans Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 3–14.

Mathews, G. J. et Jacobs, B. E. (1996). Electronic management of the peer review process. *Dans Proceedings of the fifth international World Wide Web conference on Computer networks and ISDN systems*, pages 1523–1538, Amsterdam, The Netherlands, The Netherlands. Elsevier Science Publishers B. V.

Medvidovic, N., Rosenblum, D. S., Redmiles, D. F. et Robbins, J. E. (2002). Modeling software architectures in the unified modeling language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):2–57.

MEDVIDOVIC, N. et TAYLOR, R. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 28(1).

MICROSYSTEMS, S. (2003). Enterprise java beans specification version 2.1. Technical report, Honeywell Technology Center.

MODAFFERI, S. et CONFORTI, E. (2006). Methods for enabling recovery actions in ws-bpel. *Dans OTM Conferences (1)*, pages 219–236.

MODAFFERI, S., MUSSI, E. et PERNICI, B. (2006). Sh-bpel : a self-healing plug-in for ws-bpel engines. *Dans MW4SOC '06 : Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, pages 48–53, New York, NY, USA. ACM Press.

MÉTAYER, D. L. (1998). Describing software architecture styles using graph grammars. *IEEE Transactions On Software Engineering*, 24(7):521–533.

NECHES, R., FIKES, R., FININ, T., GRUBER, T., PATIL, R., SENATOR, T. et SWARTOUT, W. R. (1991). Enabling technologies for knowledge sharing. *AI Magazine*, 12(3).

NIERSTRASZ, O. (2000). Identify the champion. *Dans* HARRISON, N., FOOTE, B. et ROHNERT, H., éditeurs : *Pattern Languages of Program Design*, volume 4, pages 539–556. Addison Wesley.

OMG (2002). Corba component model specification formal/2002-06-65. Rapport technique 3, Object Management Group.

OMG (2005). Unified modeling language specification. infrastructure. formal/05-07-04 Version 2.0, Object Management Group.

OQUENDO, F. (2004). pi-adl : An architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures. *ACM Software Engineering Notes*, 29(4).

OREIZY, P., GORLICK, M. M., TAYLOR, R. N., HEIMBIGNER, D., JOHNSON, G., MEDVIDOVIC, N., QUILICI, A., ROSENBLUM, D. S. et WOLF, A. L. (1999). An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62.

OREIZY, P., MEDVIDOVIC, N. et TAYLOR, R. N. (1998). Architecture-based runtime software evolution. *Dans ICSE '98 : Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA. IEEE Computer Society.

Papagelis, M., Plexousakis, D. et Nikolaou, P. (2005). Confious : Managing the electronic submission and reviewing process of scientific conferences. *Dans WISE*, pages 711–720.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commununications of the ACM*, 15(12):1053–1058.

Pérez-Martínez, J. E. (2003). Heavyweight extensions to the uml metamodel to describe the c3 architectural style. *SIGSOFT Softw. Eng. Notes*, 28(3).

Pérez-Martínez, J. E. et Sierra-Alonso, A. (2004). Uml 1.4 versus uml 2.0 as languages to describe software architectures. *Dans First European Workshop on Software Architecture, EWSA*, pages 88–102.

Perry, D. E. et Wolf., A. L. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52.

Pfister, C. et Szyperski, C. (1996). Why objects are not enough. *Dans Proceedings, International Component Users Conference*, Munich, Germany.

Rausch, A. (2001). Towards a software architecture specification language based on uml and ocl. *Dans Proceeding of the Workshop on Describing Software Architecture with UML*, Toronto, Canada.

Robbins, J. E., Medvidovic, N., Redmiles, D. F. et Rosenblum, D. S. (1998). Integrating architecture description languages with a standard design method. *Dans ICSE '98 : Proceedings of the 20th international conference on Software engineering*, pages 209–218, Washington, DC, USA. IEEE Computer Society.

Rogerson, D. (1997). *Inside COM*. Microsoft Press, Redmond VA, USA.

Rumbaugh, J. (1997). *OMT Insights : Perspectives on Modeling from the Journal of Object-Oriented Programming*. Cambridge University Press.

Rumbaugh, J., Jacobson, I. et Booch, G. (2005). *The Unified Modeling Language Reference Manual. Second Edition*. Pearson Education, Inc.

Schewe, K.-D. (2000). Uml : A modern dinosaur ? a critical analysis of the unified modelling language. *Dans* Kangassalo, H., Jaakkola, H. et Kawaguchi, E., éditeurs : *Proc. 10th European-Japanese Conference on Information Modelling and Knowledge Bases, Saariselkä (Finland), 2000*. IOS Press, Amsterdam.

Schmerl, B. et Garlan, D. (2002). Exploiting architectural design knowledge to support self-repairing systems. *Dans SEKE '02 : Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 241–248, New York, NY, USA. ACM Press.

Sharp, I. P. (1970). Software engineering techniques : Report of a conference sponsored by the nato science committee. page 12.

Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D. et Zelesnik, G. (1995). Abstraction for software architecture and tools to support them. *Dans IEEE Transactions on Software Engineering*, volume 21, pages 314–335.

Shaw, M. et Garlan, D. (1996). *Software Architecture : Perspective on an Emerging Discipline*. Prentice Hall Eds.

Shin, M. E. (2005). Self-healing components in robust software architecture for concurrent and distributed systems. *Sci. Comput. Program.*, 57(1):27–44.

Studer, R., Benjamins, V. R. et Fensel, D. (1998). Knowledge engineering : Principles and methods. *Data Knowledge Engineering*, 25(1-2):161–197.

Szypersky, C. (1999). *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley.

Taylor, R. N., Medvidovic, N., Anderson, K. M., Jr., E. J. W., Robbins, J. E., Nies, K. A., Oreizy, P. et Dubrow, D. L. (1996). A component and message-based architectural style for GUI software. *Software Engineering*, 22(6):390–406.

Vestal, S. (1993). A cursory overview and comparison of four architecture description languages. Technical report, Honeywell Technology Center.

Vogel, A., Kerherve, B., von Bochmann, G. et Gecsei, J. (1995). Distributed multimedia and qos : A survey. *IEEE MultiMedia*, 02(2):10–19.

W3C (2001). Daml+oil. Reference description. En ligne, http ://www.w3.org/TR/daml+oil-reference,.

W3C (2004a). Owl web ontology language overview. W3c recommendation. En ligne, http ://www.w3.org/TR/owl-features/,.

W3C (2004b). Resource description framework. W3c recommendation. En ligne, http ://www.w3.org/RDF/,.

W3C (2004c). Swrl : A semantic web rule language combining owl and ruleml. W3c recommendation. En ligne, http ://www.w3.org/Submission/SWRL/.

W3C (2004d). Xml schema part 1 : Structures second edition. W3c recommendation. En ligne, http ://xmlfr.org/w3c/TR/xmlschema-1/,.

W3C (2004e). Xml schema part 2 : Datatypes second edition. W3c recommendation. En ligne, http ://www.w3.org/TR/xmlschema-2/,.

Weerawarana, S., Curbera, F., Leymann, F., Storey, T. et Ferguson, D. (2005). *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and more*. Prentice Hall.

Wile, D. S. (2002). Towards a synthesis of dynamic architecture event languages. *Dans WOSS '02 : Proceedings of the first workshop on Self-healing systems*, pages 79–84, New York, NY, USA. ACM Press.

Wile, D. S. et Egyed, A. (2004). An externalized infrastructure for self-healing systems. *Dans WICSA*, pages 285–290. IEEE Computer Society.

Annexe A

Schémas XML

Listing A.1 – Schéma XML pour la spécification des architectures à base de services Web.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns="http://www.laas.fr/~fjmoomen/WSCoopArch"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.laas.fr/~fjmoomen/WSCoopArch"
 elementFormDefault="qualified"
 attributeFormDefault="qualified">
 <xs:element name="WSCoopArch" type="ArchType" />
 <xs:complexType name="ArchType">
        <xs:sequence>
                <xs:element name="CoopRole" type="RoleType"
                    maxOccurs="unbounded" />
                <xs:element name="InterRoleLink" type="
                    InterRoleLinkType" maxOccurs="unbounded" />
        </xs:sequence>
  <xs:attribute name="archDescription" type="xs:string" use="
      required" />
 </xs:complexType>
 <xs:complexType name="RoleType">
        <xs:sequence>
                <xs:element name="ServiceCategory" type="
                    CategoryType" maxOccurs="unbounded" />
                <xs:element name="IntraRoleLink" type="
                    IntraRoleLinkType" maxOccurs="unbounded" />
        </xs:sequence>
```

```
        <xs:attribute name="roleName" type="xs:string" use="
           required"/>
</xs:complexType>
<xs:complexType name="CategoryType">
        <xs:sequence>
                <xs:element name="serviceClass" type="
                   ClassType" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="catName" type="xs:string" use="
           required"/>
</xs:complexType>
<xs:complexType name="ClassType">
        <xs:sequence>
                <xs:element name="Service" type="ServiceType"/
                   >
        </xs:sequence>
        <xs:attribute name="className"/>
</xs:complexType>
<xs:complexType name="ServiceType">
        <xs:sequence>
        <xs:element name="role">
                <xs:complexType>
                        <xs:attribute name="roleName" type="
                           xs:string" use="required"/>
        </xs:complexType>
                </xs:element>
                <xs:element name="category">
                        <xs:complexType>
          <xs:attribute name="catName" type="xs:string" use="
             required"/>
                        </xs:complexType>
                </xs:element>
                        <xs:element name="class">
                                <xs:complexType>
                                        <xs:attribute name="
                                           className"/>
                                </xs:complexType>
                        </xs:element>
                </xs:sequence>
                <xs:attribute name="serviceName" use="required
```

```
                    "/>
        </xs:complexType>
        <xs:complexType name="InterRoleLinkType">
                <xs:sequence>
                        <xs:element name="service1" type="
                            ServiceType"/>
                        <xs:element name="service2" type="
                            ServiceType"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="IntraRoleLinkType">
                <xs:sequence>
                        <xs:element name="service1" type="
                            ServiceType"/>
                        <xs:element name="service2" type="
                            ServiceType"/>
                </xs:sequence>
        </xs:complexType>
</xs:schema>
```

Listing A.2 – Schéma XML pour la spécification des règles de base

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns="http://www.laas.fr/~fjmoomen/ArchCoopRules
    -0.2"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:archCoop="http://www.laas.fr/~fjmoomen/WSCoopArch"
  xmlns:ns1="http://www.laas.fr/~fjmoomen/ArchCoopRules-0.2"
  targetNamespace="http://www.laas.fr/~fjmoomen/ArchCoopRules
      -0.2"
   elementFormDefault="qualified"
   attributeFormDefault="qualified">
        <xs:import namespace="http://www.laas.fr/~fjmoomen/
            WSCoopArch" schemaLocation="Z:\public_html\
            WSCoopArch.xsd" />
        <xs:element name="basicRule" type="ruleType" />
        <xs:complexType name="ruleType">
              <xs:sequence>
                    <xs:element name="Add" type="Node" />
                    <xs:element name="Del" type="Node" />
                    <xs:element name="Inv" type="Node" />
                    <xs:element name="Abs" type="Node" />
                    <xs:element name="IntersectionLinks"
                        minOccurs="0" maxOccurs="unbounded">
                          <xs:complexType>
                          <xs:complexContent>
                          <xs:extension base="
                              archCoop:InterRoleLinkType">
                                <xs:attribute name="
                                    InvolvedSections"
                                    use="required">
                                      <xs:simpleType
                                          >
                                      <
                                          xs:restriction
                                           base="
                                          xs:string" />
                                      </
                                          xs:simpleType
                                          >
                          </xs:attribute>
```

```
                        </xs:extension>
                            </xs:complexContent>
                            </xs:complexType>
                    </xs:element>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="Node">
                <xs:sequence>
                    <xs:element name="service" type="
                        archCoop:ServiceType"/>
                    <xs:element name="Interaction" type="
                        archCoop:IntraRoleLinkType"
                        minOccurs="0" maxOccurs="unbounded"/
                        >
                </xs:sequence>
        </xs:complexType>
</xs:schema>
```

Annexe B

Diagrammes UML

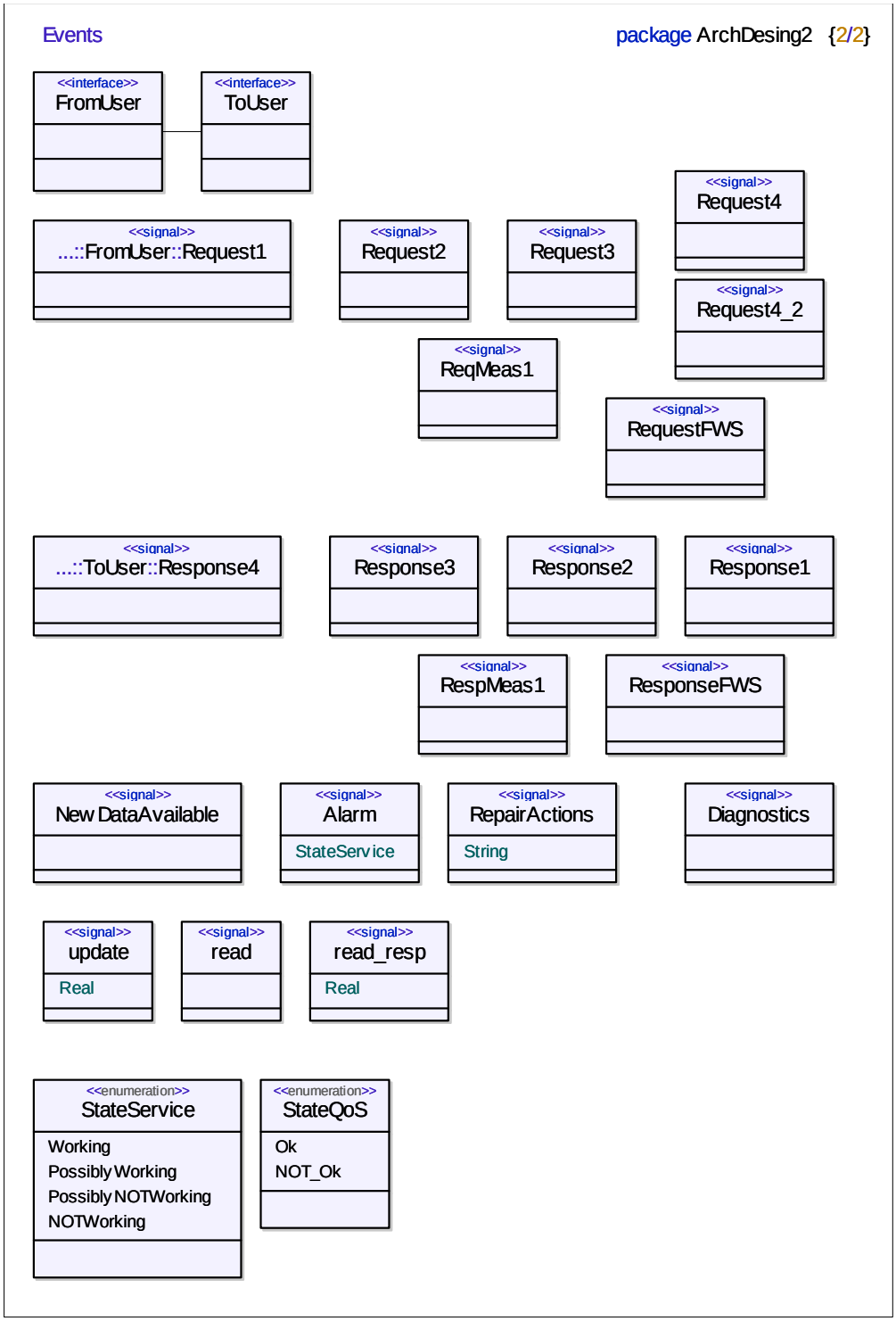


FIG. B.1 – Messages échangés pour les composants de l'architecture de gestion de la QdS.

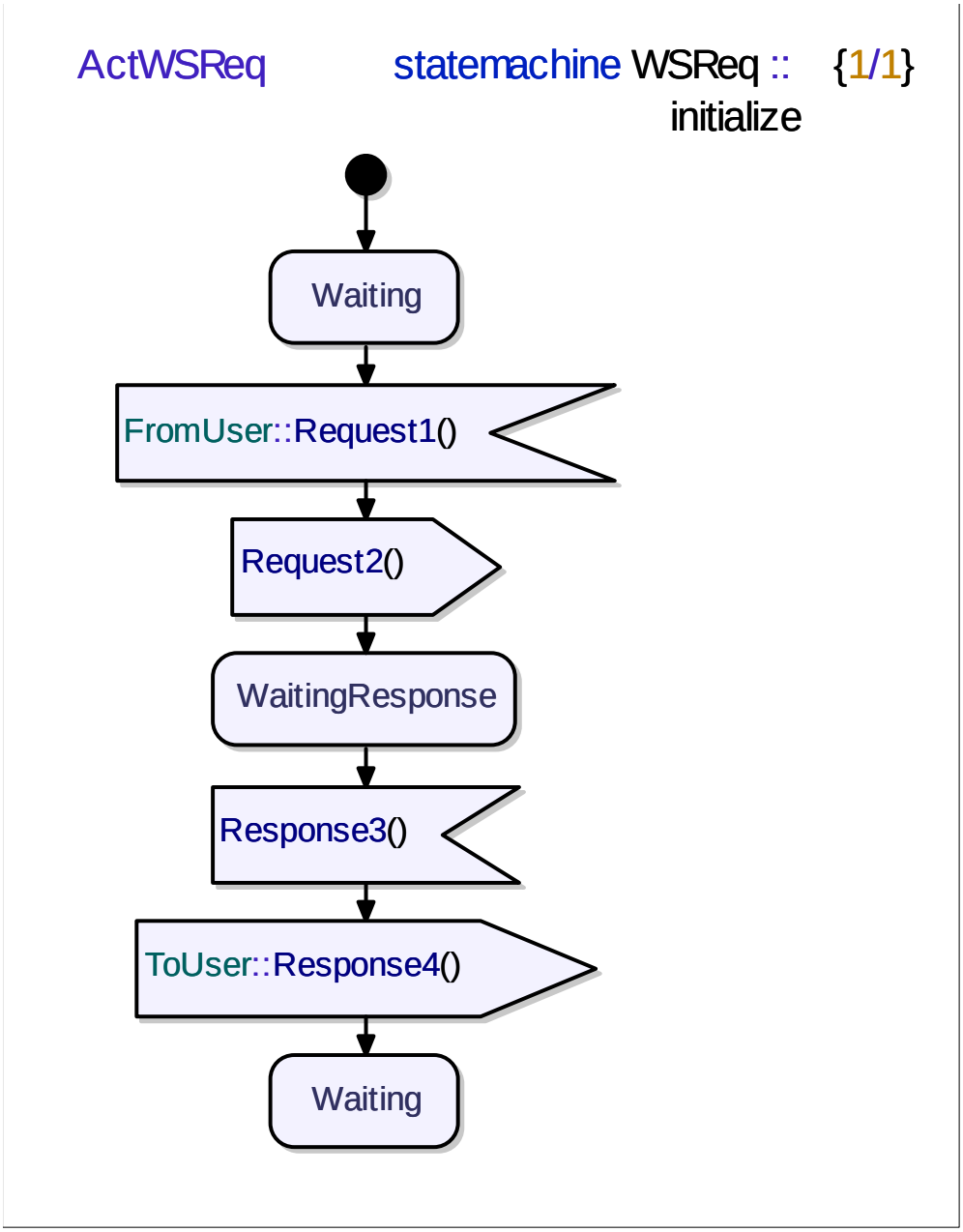


FIG. B.2 – Diagramme de machine à état du service Web demandeur.

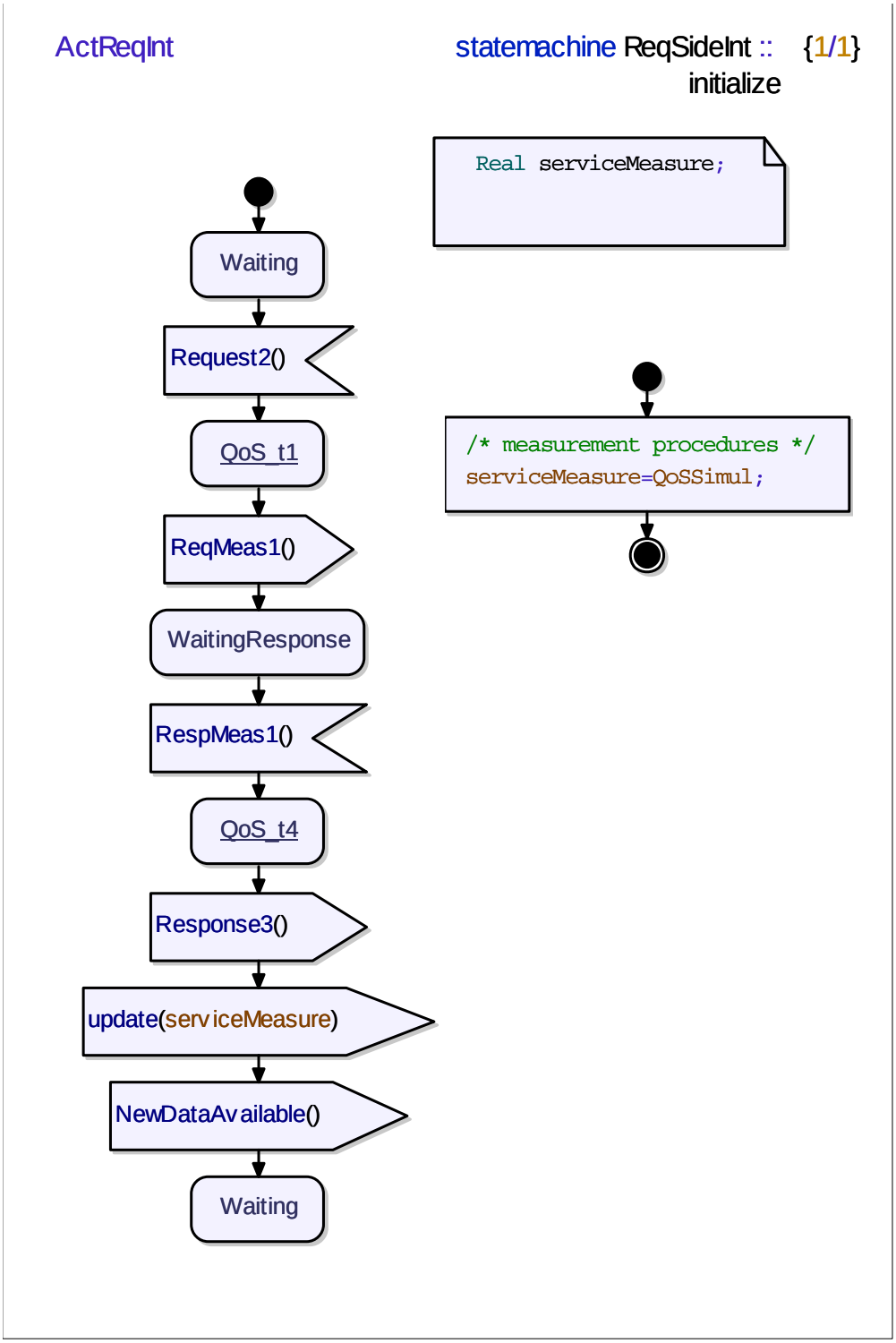


FIG. B.3 – Diagramme de machine à état de l'intercepteur du côté du WS demandeur.

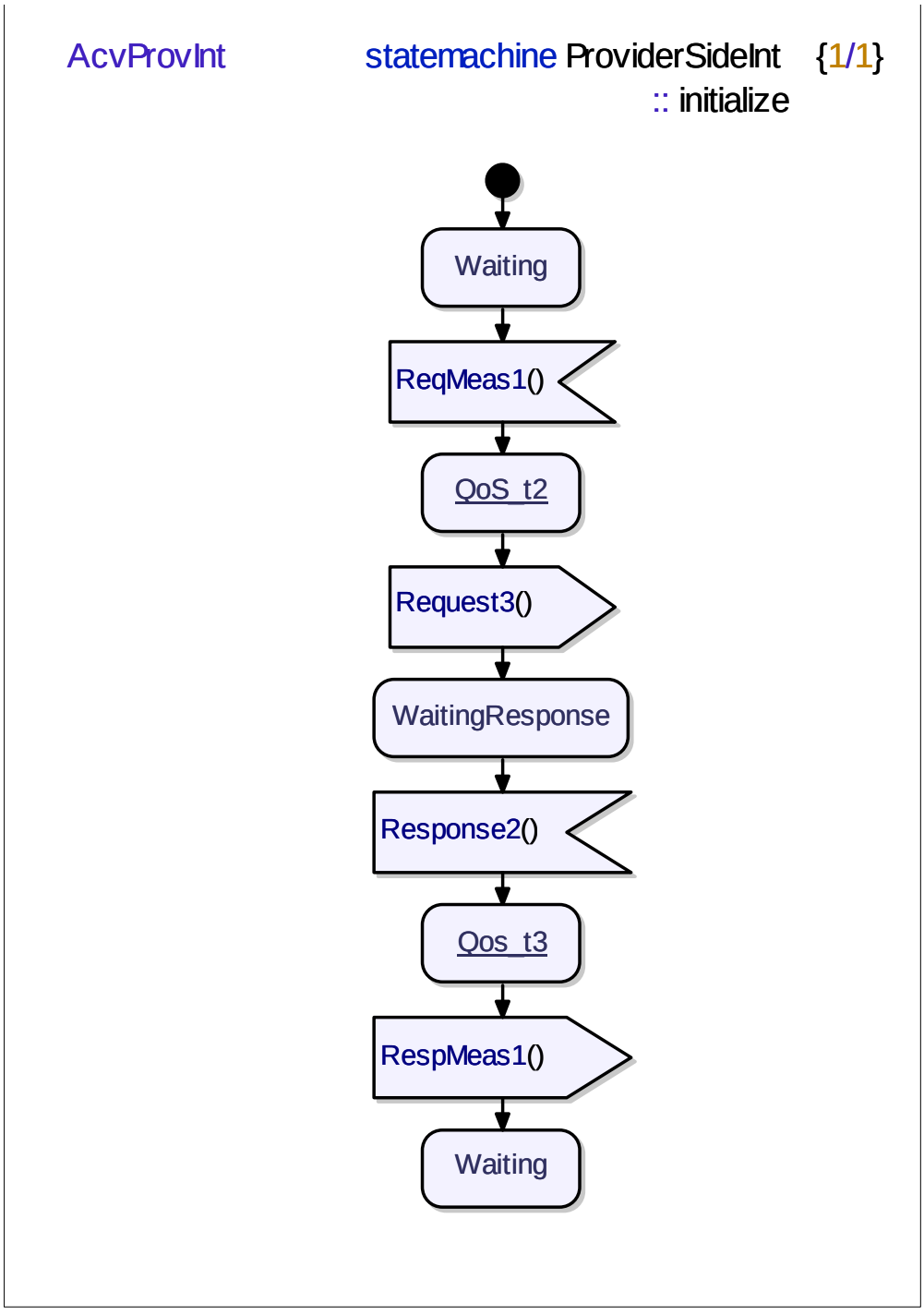


FIG. B.4 – Diagramme de machine à état de l'intercepteur du côté du WS fournisseur.

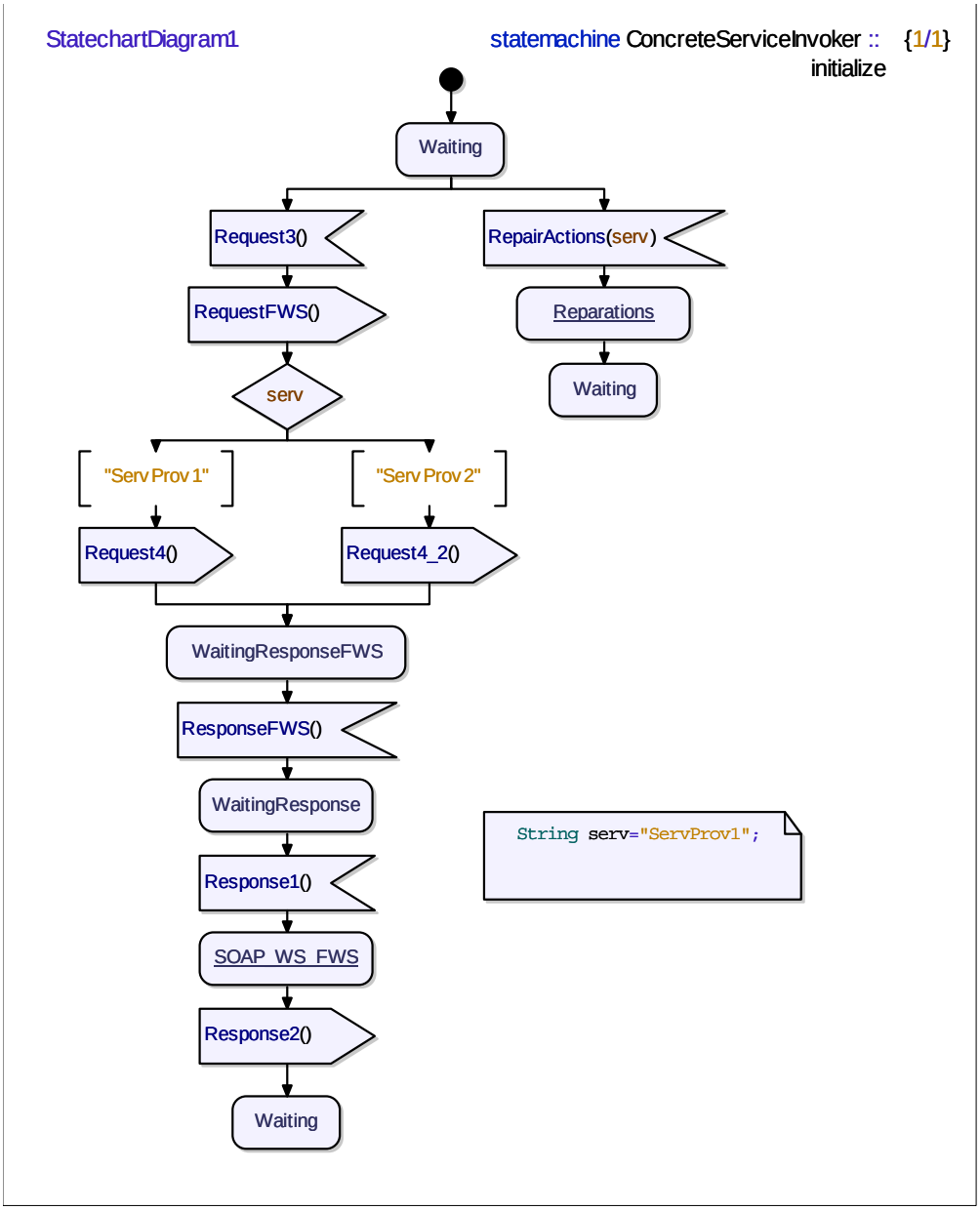


FIG. B.5 – Diagramme de machine à état de l'intercepteur *Concrete Service Invoquer* (1/2).

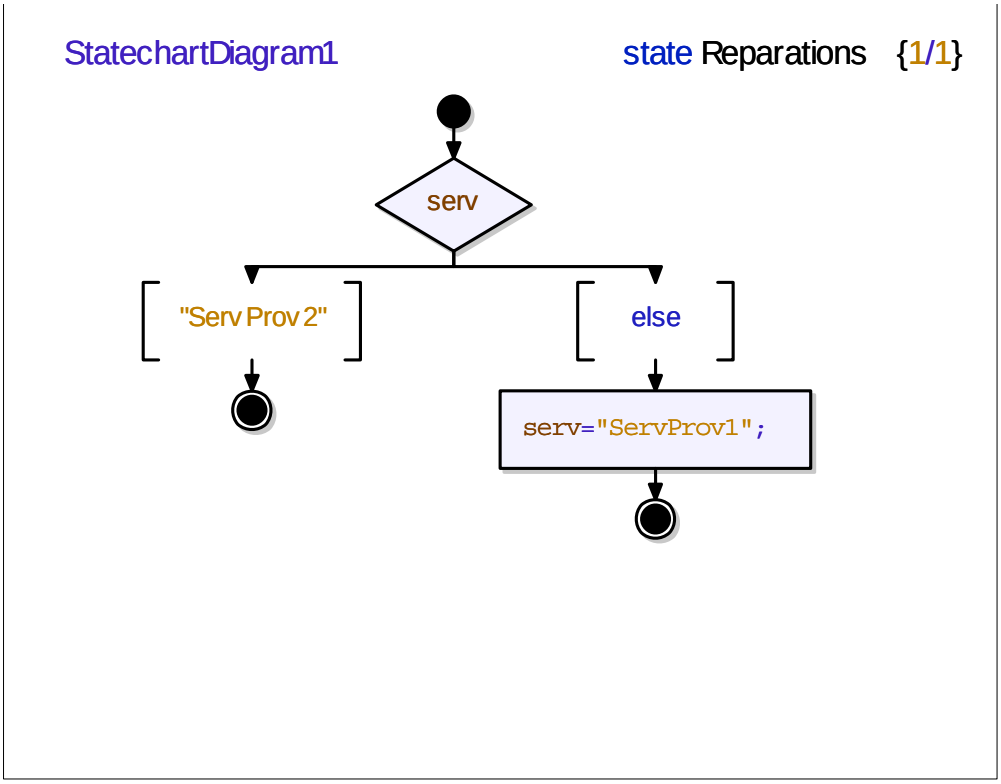


FIG. B.6 – Diagramme de machine à état de l'intercepteur *Concrete Service Invoquer* (2/2).

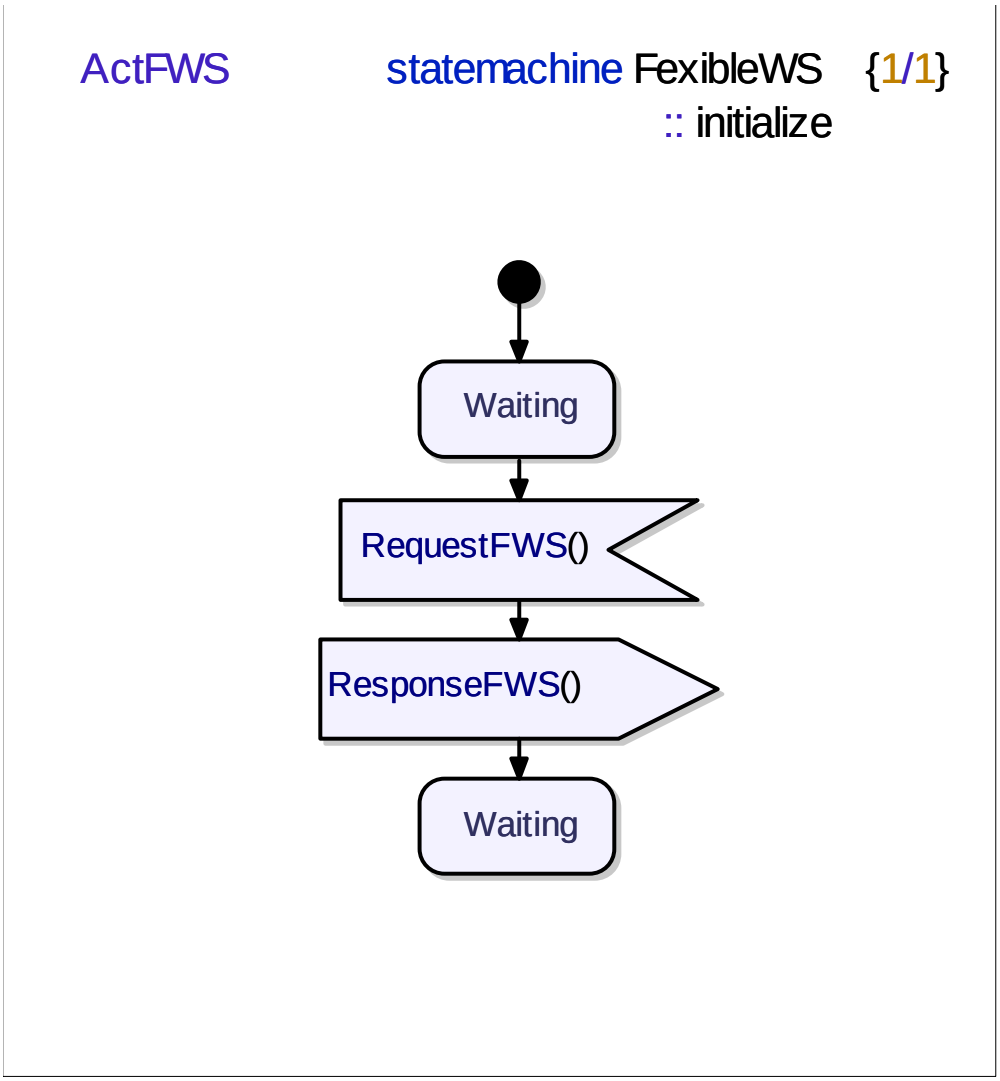


FIG. B.7 – Diagramme de machine à état du service Web *FS*.

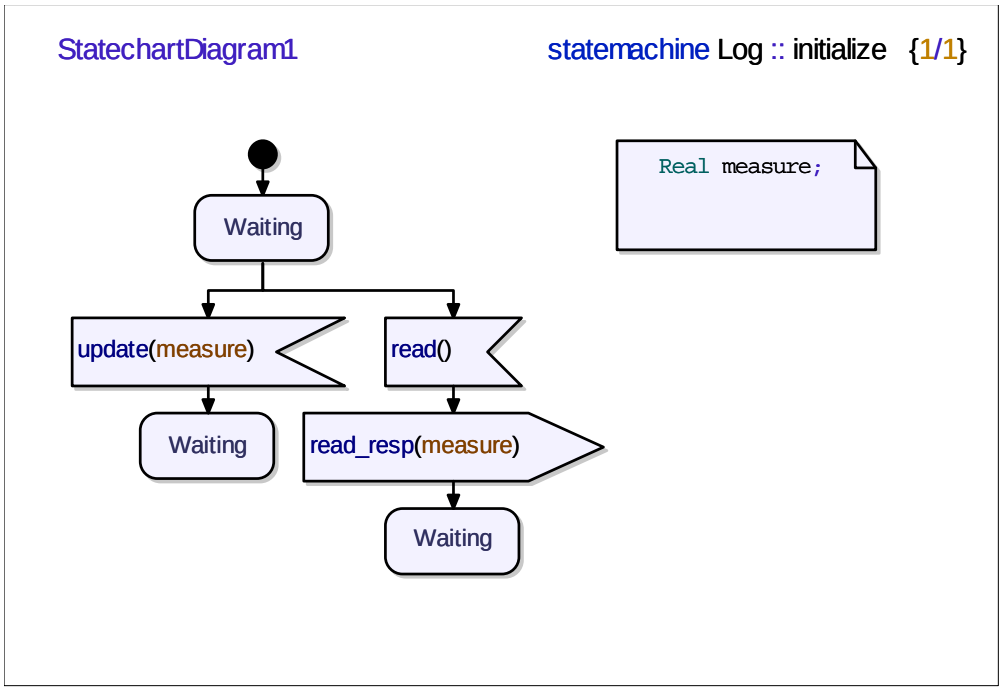


FIG. B.8 – Diagramme de machine à état du composant *Log*.

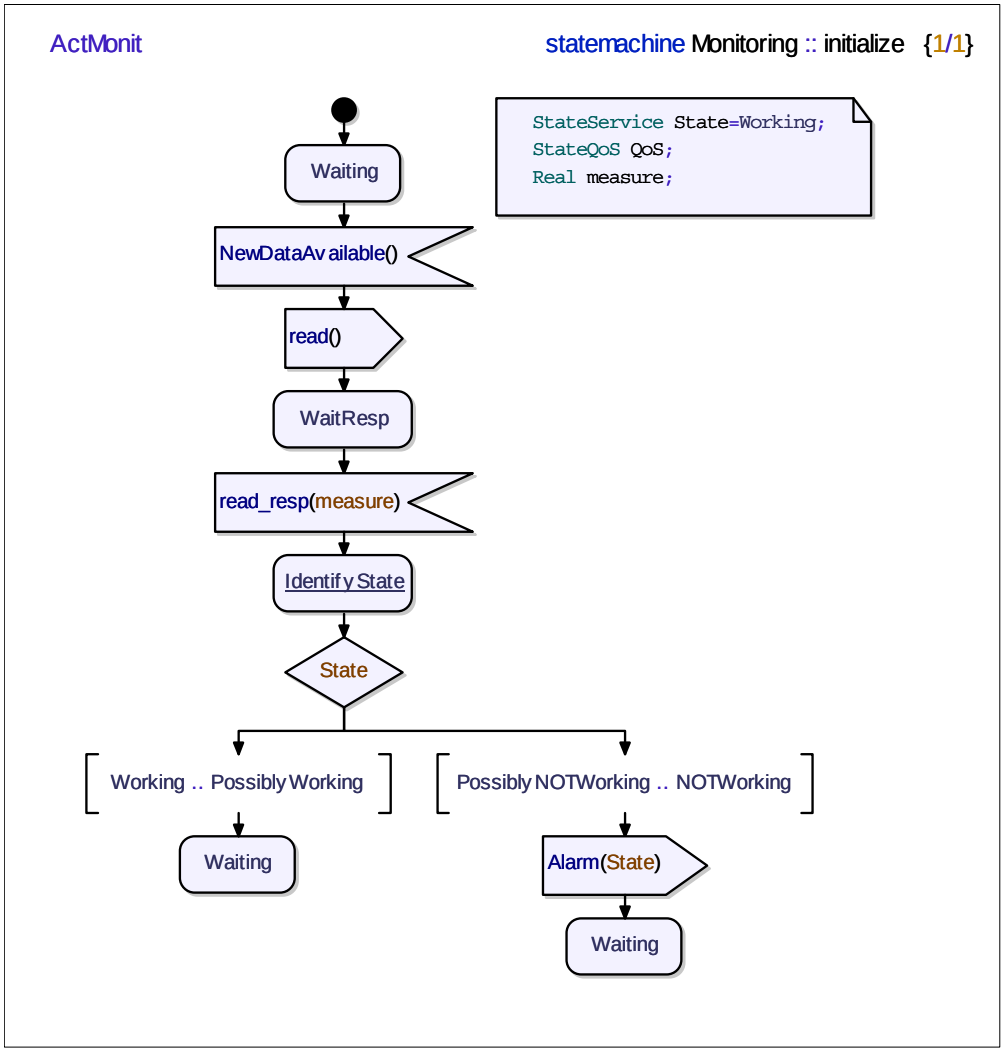


FIG. B.9 – Diagramme de machine à état du composant *Monitoring* (1/2).

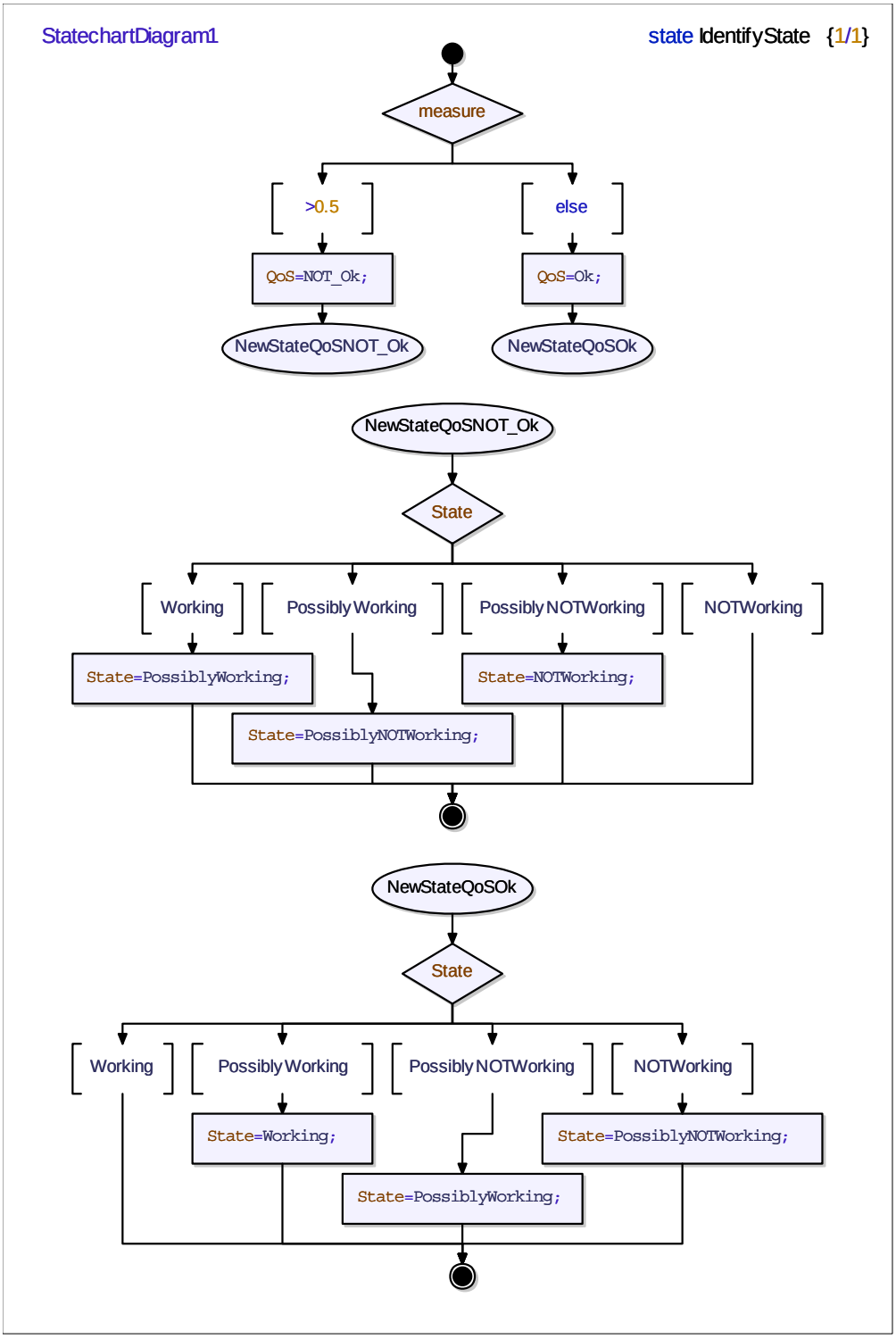


FIG. B.10 – Diagramme de machine à état du composant *Monitoring* (2/2).

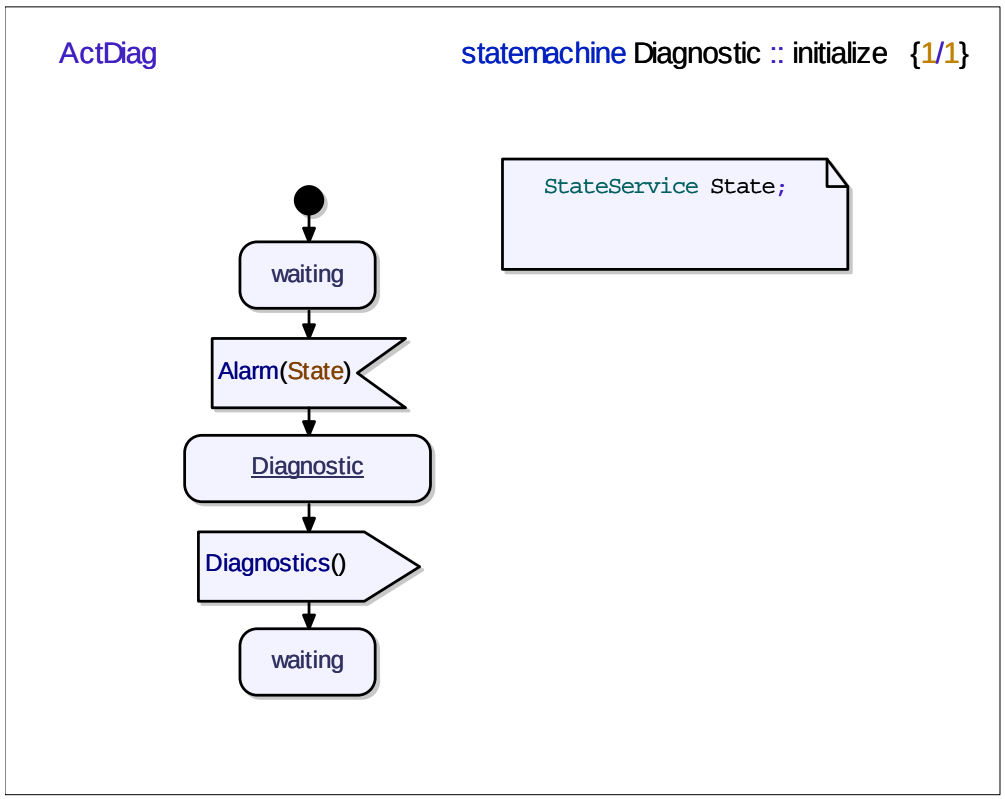


FIG. B.11 – Diagramme de machine à état du composant *Diagnostic*.

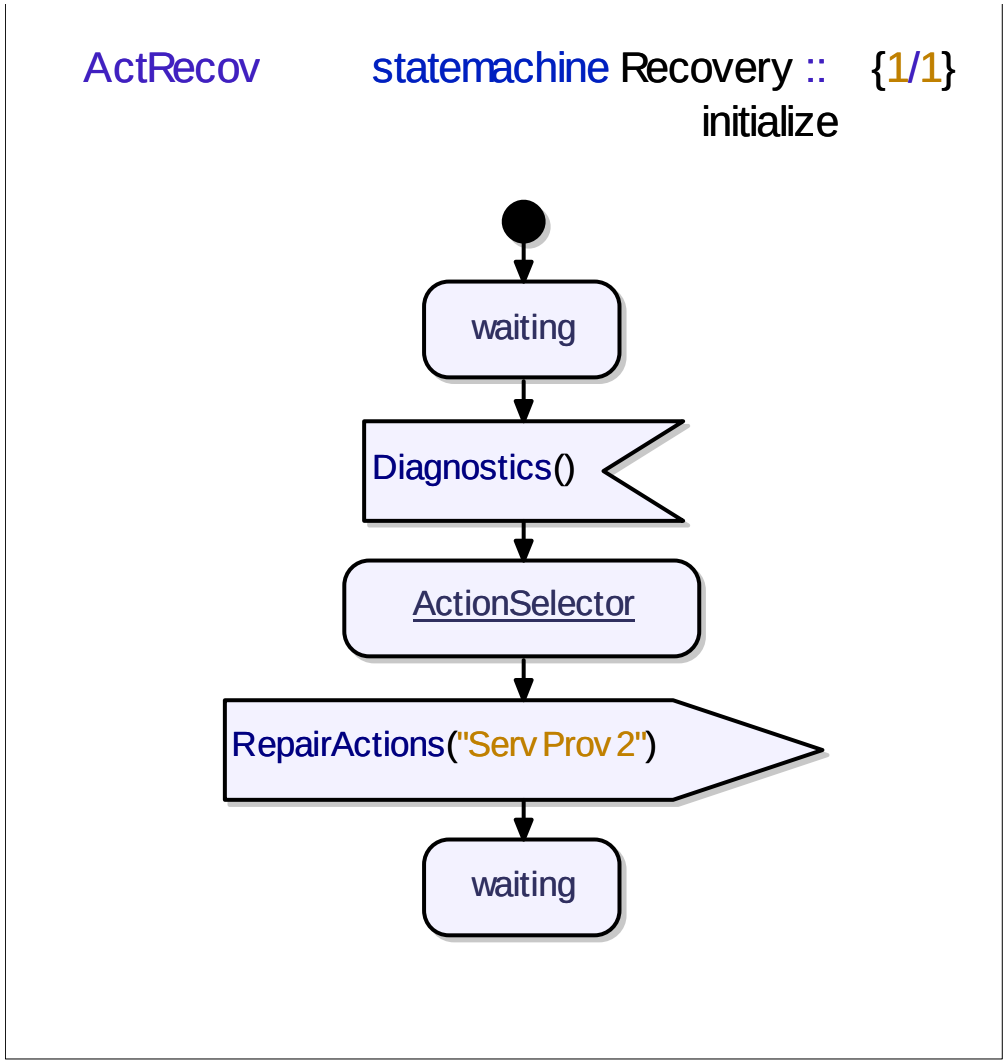


FIG. B.12 – Diagramme de machine à état du composant *Recovery*.

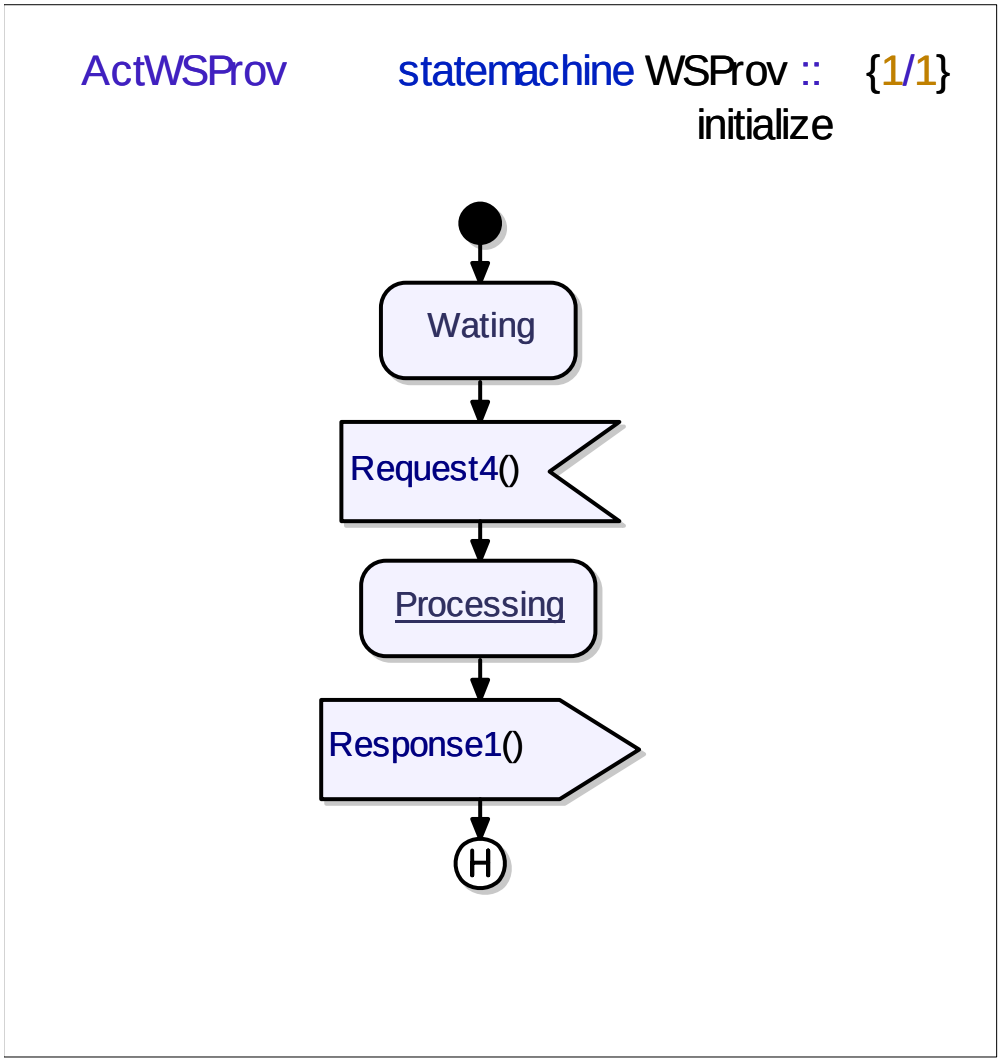


FIG. B.13 – Diagramme de machine à état du premier service Web fournisseur.

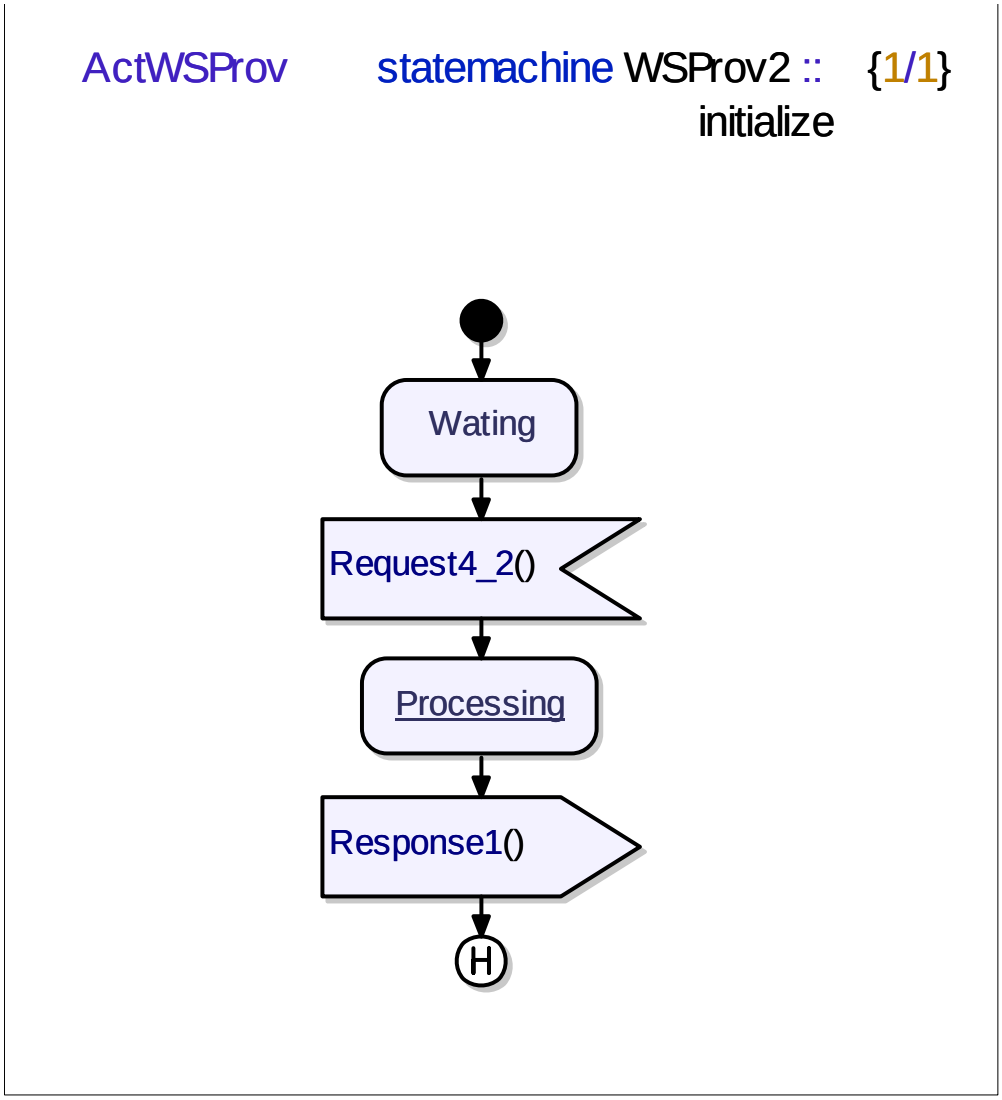


FIG. B.14 – Diagramme de machine à état du second service Web fournisseur.

Annexe C

Publications de l'auteur

C.1 Conférences internationales avec actes et comité de lecture

- F.J.MOO-MENA et K.DRIRA. Reconfiguration of web services architectures : a model-based approach. The Twelfth IEEE Symposium on Computers and Communications (ISCC'07), Aveiro (Portugal), 1-4 Juillet 2007.

- F.J.MOO-MENA et K.DRIRA. Modeling architectural level repair in web services. 3rd International Conference on Web Information Systems and Technologies (WEBIST'2007), Barcelone (Espagne), 3-6 Mars 2007, pp.240-245.

- F.J. MOO-MENA et K.DRIRA. A component-based design approach for collaborative distributed systems. 3rd IEEE International Symposium and School on Advance Distributed (ISSADS'2004), Guadalajara (Mexique), Janvier 2004 Advanced Distributed Systems. Eds. V.Larios, FF.Ramon, H.Unger, Lecture Notes in Computer Science 3061, Springer, 2004, ISBN 3-540-22172-7, pp.197-206.

C.2 Manifestations d'audience nationale

- F.J.MOO-MENA. Conception d'architectures orientées composants pour les logiciels coopératifs distribués.Actes du Colloque de l'Ecole Doctorale Informatique et Telecommunications (EDIT'2005), Toulouse (France), 11-12 Avril 2005, 5p.

C.3 Délivrables du projet WS-DIAMOND

- R.BEN HALIMA , K.DRIRA , K.GUENNOUN et F.J.MOO-MENA. Specification of execution mechanisms and composition strategies for self-healing Web services. Phase 1. Projet IST WS-DIAMOND N°516933, Février 2007, 102p.
- K.DRIRA , K.GUENNOUN , F.J.MOO-MENA , Y.PENCOLE , X.PUCEL , A.SUBIAS et L.TRAVE-MASSUYES. Requirements, application scenarios, overall architecture, and test/validation specification, common working environment and standards at Milestone M1. Projet IST WS-DIAMOND N°516933, Mars 2006, 166p.

C.4 Rapports de recherche

- F.J.MOO-MENA et K.DRIRA. Toward architectural-level repair in self-healing web services. Rapport LAAS N°06636, Septembre 2006, 12p.
- F.J.MOO-MENA. Toward effective modeling of cooperative service-oriented architecture. Rapport LAAS N°06607, Septembre 2006, 11p.
- F.J.MOO-MENA et K.DRIRA. An approach for cooperative service-oriented architecture deployment. Rapport LAAS N°05506, Septembre 2005, 8p.
- F.J.MOO-MENA , K.DRIRA et M.DIAZ. Distributed cooperative architectures for collaborative software design. Rapport LAAS N°05278, Juin 2005, 18p.