R-*, Réflexion au Service de l'Évolution des Systèmes de Systèmes

Jonathan Labéjof

Ecole Doctorale Sciences Pour l'Ingénieur Université Lille Nord-de France – EDSPI – 072

– R-* –
Réflexivité au service de l'Évolution des Systèmes de Systèmes

THÈSE

présentée et soutenue publiquement le 20 décembre 2012

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille

(spécialité informatique)

par

Labéjof Jonathan

Composition du jury

Président :	Antoine Beugnard	ENST de Bretagne
Rapporteur :	Jean-Michel Bruel	Université de Toulouse
Examinateurs :	Pierre Chatel	THALES COMMUNICATIONS & SECURITY
	Philippe Roose	LIUPPA / Université de Pau et des Pays de l'Adour
Directeur:	Lionel Seinturier	Université Lille 1 & IUF
Encadrant:	Philippe Merle	Inria Lille - Nord Europe
Invité :	Antoine Léger	THALES COMMUNICATIONS & SECURITY

UMR Lille1/CNRS 8022

Mis en page avec la classe thloria.

Résumé

Dans un monde de plus en plus connecté, le besoin d'interconnecter des systèmes hétérogènes apparait de plus en plus présent. Les Systèmes de Systèmes (SoS) sont une approche de supervision et de contrôle global où les systèmes constituants sont caractérisés comme des sous-systèmes du SoS.

Certains de ces sous-systèmes peuvent être sujets à un environnement dynamique leur demandant d'évoluer pour répondre à de nouvelles exigences, voire de s'adapter s'ils doivent répondre à un besoin de disponibilité. La principale difficulté dans la gestion des évolutions possibles est qu'elles peuvent impacter plusieurs sous-systèmes connectés, et par extension, une vision globale comme celle proposée par un système de systèmes.

Ainsi, les problèmes d'évolution et d'adaptation d'un SoS se posent.

Dans un cadre d'ingénierie logicielle, cette thèse propose l'approche R-* qui soutient l'hypothèse que plus un système est Réflexif, et plus il devient capable de s'adapter, et donc d'évoluer.

Trois contributions majeures et un cas d'utilisation évalué sont proposés pour justifier l'intérêt de R-*.

R-DDS et R-MOM ajoutent la capacité de réflexivité dans des sous-systèmes de communication asynchrones, et R-EMS ajoute la capacité de réflexivité sur la vision globale d'un SoS, incluant ses sous-systèmes et son environnement.

R-DDS ajoute la réflexivité à l'intergiciel de Service de Distribution de Données dédié aux domaines du temps-réel et de l'embarqué. R-MOM remonte en abstraction pour proposer la réflexivité au niveau des fonctionalités d'un intergiciel asynchrone. R-EMS est un système de gestion d'environnement réflexif en support de l'utilisation d'un SoS.

Finalement, le cas d'utilisation est une implémentation d'un sous-modèle du système de systèmes TACTICOS de THALES, qui servira également d'environnement d'évaluation.

Mots-clés: Système de Systèmes (SoS), Système Distribué, Qualité de Services (QoS), Service de Distribution de Données (DDS), Architecture Orientée Services (SOA), Modèle à composant Réflexif, Intergiciel Orienté Message (MOM), Adaptation, Reconfiguration, Interopérabilité.

Abstract

In a connected world, interconnection of heterogeneous systems becomes a need. Systems of systems (SoS) answer to this need by providing global supervision and control over such systems, named sub-systems in the context of SoS.

Some sub-systems face to a dynamic environment, therefore, they have to evolve in order to meet new requirements, and they have to perform adaptations whenever availability is a requirement. Main difficulty about evolution is it can concern a set of sub-systems, or a global vision such as one provided by system of systems.

Therefore, the problems of evolution and adaptation are important.

In the domain of software engineering, this thesis provides the R-* approach that defends the hypothesis that the more a system is Reflective, and the more it is able to adapt, and so, to evolve.

Three major contributions and one evaluated use case justify R-*. R-DDS and R-MOM add reflective capabilities in asynchronous communication sub-systems, and R-EMS adds reflectivity on a global vision of a SoS, its sub-systems and its environment.

R-DDS adds reflectivity to Data Distribution Service dedicated to real-time and embedded domains. R-MOM goes up in abstraction compared to R-DDS, in adding reflective capabilities at level of asynchronous middleware functionalities. R-EMS is a Reflective Environment Management System helping SoS use.

Finally, use case and evaluation are done over a sub-model implementation of THALES' SoS TACTICOS.

Keywords: System of Systems (SoS), Distributed System, Quality of Services (QoS), Data Distribution Service (DDS), Services Oriented Architecture (SOA), Reflective Component Model, Message-Oriented Middleware (MOM), Adaptation, Reconfiguration, Interoperability.

Remerciements

Je tiens à remercier toutes les personnes qui ont contribué dans l'aboutissement de cette thèse, aussi bien par le biais de leurs connaissances techniques, que par leur sympathie.

Je tiens tout d'abord à remercier tout particulièrement mes encadrants de thèse, Lionel Seinturier, Philippe Merle, Hugues Vincent et Antoine Léger, pour leur patience et leur soutien donnés tout au long de ce travail de thèse.

Je tiens également à remercier, Laurence Duchien (INRIA) et Pascal Llorens (THALES), qui se sont montrés très accueillants et disponibles pour m'intégrer dans leurs équipes respectives. Et un grand merci aux collègues avec qui j'ai passé le plus clair de mon temps et qui ont contribué dans les idées et dans l'apport de connaissances techniques dans mes travaux.

Je remercie également monsieur Chartier, propriétaire du logement que j'ai occupé durant ces trois années dans la sympatique ville qu'est Champlan, à une heure de Palaiseau. La tranquilité et le calme de l'endroit furent parfaits pour prendre du recul sur mes idées une fois la journée active terminée.

Finalement, je remercie tout particulièrement Amandine pour m'avoir aidé à démarrer ce travail.

Et surtout un grand merci à mes parents, à Gaëlle et à ma famille qui ont accepté de ne pas trop s'intéresser à mon travail pour que je puisse me changer les idées en leur présence, et qui sont restés disponibles lorsque j'avais besoin d'eux pour conclure ce long, riche et intense chapitre de trois ans de vie.

À toi, lecteur curieux ou intéressé, puisses-tu trouver les réponses qui t'aideront à résoudre le but de ta venue.

Table des matières

Chapitre 2
État de l'art

Table des figures

Liste des tableaux

Acronymes

Acronyme	Définition	URL (http ://)	section
ALF	Action Language For Foundational UML	www.omg.org/spec/ALF/	2.4.3
AOP	Aspect Oriented Programming		2.3.4
API	Application Programming Interface		
AMQP	Advanced Message Queuing Protocol	www.amqp.org/	2.5.7
AST	Abstract Syntax Tree		
BPEL	Business Process Engine Language		
BNF	Backus Naur Form		
BPMN	Business Process Modeling Notation	www.bpmn.org/	
CBSE	Component Based Software Engineering		
CCM	CORBA Component Model	www.omg.org/spec/CCM/	2.6.1
CMS	Combat Management System		
COM	Component Object Model	www.microsoft.com/com/	2.6.3
CORBA	Common Object Request Broker Architecture	www.omg.org/corba/	2.5.2
CQML	Component QML		2.7.3
DARPA	Defense Advanced Research Projects Agency	www.darpa.mil/	
DI	Dependency Injection		2.3.2
DDS	Data Distribution Service	www.omg.org/spec/DDS/	2.5.6
DDSI	DDS Interoperability Wire Protocol	www.omg.org/spec/DDSI/	2.5.6
DNS	Domain Name System		
DSL	Domain Specific Language		2.4.4
DSM	Domain Specific Modeling		2.4.4
ESB	Enterprise Service Bus		2.5.10
EDP	Event Driven Programming		
FMP	Formal Method Programming		
FOP	Feature Oriented Programming		2.3.1
FP	Functional Property		
IoC	Injection of Control		2.3.2
IP	Internet Protocol		
IT	Information Technology		
ITEmIS	IT, Embedded and Integrated Systems	research.petalslink.org/display/itemis/	
IDL	Interface Description Language	www.omg.org/gettingstarted/omg_idl/	
JMS	Java Message Service	www.oracle.com/technetwork/java/jms/	2.5.5
LwCCM	Lightweight CCM		2.6.1
MAPE	Monitor, Analyze, Plan, Execute		2.2.2
MAPE-K	MAPE-Knowledge		2.2.2
MARTE	Modeling and Analysis of RT-E systems		2.7.5
MDA	Model Driven Architecture	www.omg.org/mda/	2.3.5
MDE	Model Driven Engineering		2.3.5
MOM	Message-Oriented Middleware		2.5.4
NFP	Non-Functional Property		
NPC	Non Polynomial-Complete		
nRT	non-RT		
OCL	Object Constraint Language	www.omg.org/ocl/	2.7.1
OMG	Object Management Group	www.omg.org/	
OOP	Object Oriented Programming		
ORB	Object Request Broker		2.5.2
OS	Operating System		
OWL	Web Ontology Language	www.w3.org/TR/owl-features/	5.3.1
PIM	Platform Independent Model		2.3.5

Acronyme	Définition	URL (http ://)	section
PSM	Platform Specific Model		2.3.5
QML	QoS Modeling Language		2.7.3
QoS	Quality of Services		2.7
REST	REpresentational State Transfer		
RPC	Remote Procedure Call		2.5.1
RT	Real-Time		2.2.1
RT-E	Real-Time and Embedded		
SCA	Services Component Architecture	www.oasis-opencsa.org/sca/	2.6.5
SLA	Service Level Agreement	www.service-level-agreement.net/	2.7.4
SOA	Services Oriented Architecture		2.2.4
SOAP	Simple Object Access Protocol	www.w3.org/TR/soap/	
SoC	Separation of Concerns		2.3.3
SoS	System of Systems		2.2.1
SPF	Single-Point Failure		
SPL	Software Product Line		2.3.1
SysML	Systems Modeling Language	www.sysml.org/	2.4.3
TCP	Transmission Control Protocol		
UDP	User Datagram Protocol		
UML	Unified Modeling Language	www.uml.org/	2.4.3
WSLA	Web Service Level Agreement	www.research.ibm.com/wsla/	2.7.4
URI	Uniform Resource Identifier		
URL	Uniform Resource Locator		
W3C	World Wide Web Consortium	www.w3.org/	
WAN	Wide Area Network		
WCF	Windows Communication Foundation		
WS	Web Services	www.w3.org/TR/ws-arch/	2.5.3
WSDL	Web Services Description Language	www.w3.org/TR/wsdl/	
XML	eXtensible Markup Language	www.w3.org/XML/	
XP	eXtreme Programming		

Première partie

Contexte et travaux autour de la réflexivité pour l'évolution des Systèmes de Systèmes

Chapitre 1

Introduction

Sommaire

1.1 Systèmes de systèmes, solution à l'interconnection des technologies d'information

Dans les sociétés actuelles, les systèmes utilisant des technologies d'information, ou IT, sont devenus le principal moyen de diffuser et de contrôler l'information. Ils sont présents dans la plupart de nos structures, dans les administrations, dans l'économie, etc. mais présents également chez l'individu, via les ordinateurs personnels, ordinateurs portables, smartphones.

Faute de solution alternative aussi rapide et efficace pour traiter l'information, ils sont véritablement devenus indispensables dans le fonctionnement et l'expansion des sociétés. Ils doivent rester disponibles et cohérents avec les besoins de la société, sous peine de subir des pertes économiques considérables [Sommerville et al., 2012].

Toujours dans un objectif d'expansion des activités de la société, les Systèmes de Systèmes (sous-section 2.2.1, page 12), ou SoS, s'intéressent à l'échange d'informations entre des systèmes IT, tout en apportant des capacités globales de supervision, de contrôle, d'interopérabilité et d'évolution.

Un exemple de Système de Systèmes est le système TACTICOS présenté dans l'évaluation de cette thèse (section 6.1, page 109). Ce système de Systèmes est utilisé pour superviser et contrôler des infrastructures et du matériel naval afin d'effectuer des missions militaires. Ainsi, il devient possible de connaître l'état d'un bateau et du matériel qu'il embarque à tout moment, depuis un poste de contrôle extérieur à ce bateau, et de lui donner rapidement des ordres pour ravitailler un autre navire en difficulté.

1.2 Problématique, et base de la solution

1.2.1 Problématique des environnements dynamiques

Les systèmes de systèmes offrent des capacités globales sur des systèmes indépendants (appelés sous-systèmes du SoS) les uns des autres, sujets à des domaines d'utilisation divers et variés, et pouvant évoluer pour répondre à de nouvelles formes de besoins d'utilisation.

L'environnement d'un système de systèmes, constitué de tous ces sous-systèmes, est donc hétérogène et dynamique.

Techniquement parlant, les sous-systèmes doivent opérer ensemble, ou interopérer, dans la réalisation d'un SoS, et la spécification globale doit pouvoir s'adapter aux changements réalisés par chacun des sous-systèmes. La principale difficulté est de préserver l'indépendance des sous-systèmes vis-à-vis de la vision globale, afin de limiter la propagation des erreurs de fonctionnement si le système global ou l'un des sous-systèmes devenait incohérent, suite à une attaque, ou à une faute d'exécution.

Dans ce sens, le sujet de cette thèse peut être reformulé par la question suivante :

Comment assurer des capacités globales sur des systèmes indépendants, hétérogènes et dynamiques ?

Un grand nombre de questions se posent par la suite :

- Quelle est la dualité entre les sous-systèmes et le système global ?
- Où s'applique l'interopérabilité ?
- Comment modéliser les sous-systèmes, leurs interactions et les domaines d'activité (informatique, administration, finance, médical, etc.) ?
- Qu'est-il nécessaire de globaliser ? (La tolérance aux fautes ? Le déploiement ? La sécurité ? La certification ? ...)
- Quels sont les éléments à rendre re-configurables à l'exécution ?
- etc.

1.2.2 Un réponse basée sur la réflexivité

Même s'il faut considérer qu'un système est l'union de couches matérielle et logicielle, la spécialité de cette thèse étant l'ingénierie des systèmes, seule la couche logicielle sera étudiée [Cheng et al., 2009] pour répondre au sujet, tout en justifiant des besoins de modélisation des couches matérielles.

Ingénierie système :

Le domaine de l'ingénierie système [Cheng et al., 2009] vise à améliorer les fonctionalités d'un système tout au long de son cycle de vie, c'est-à-dire, pendant les phases de développement, de déploiement, de gestion, d'utilisation et d'évolution. Mais aussi d'une manière plus spécifique, elle sert à améliorer les capacités d'usage multiple, de flexibilité, de fiabilité, d'économie d'énergie, de recouvrement, de personnalisation, de configuration et d'auto-optimisation.

La première réponse de cette thèse aux aspects de supervision et de contrôle est d'offrir un moyen d'observer l'exécution de tous les éléments d'un système qui sont sujets à être supervisés. Concernant les aspects de contrôle, il faut que les éléments visés puissent être modifiés. Ces deux éléments de réponse appartiennent à la capacité de réflexivité, qui sera le fer de lance de cette thèse.

Réflexivité :

Capacité d'un programme à examiner et à modifier ses structures internes (réflexion structurelle) ou son exécution (réflexion comportementale) durant son exécution [Smith, 1982].

Ainsi, cette thèse propose l'approche R-* [1], qui regroupe un ensemble de méthodologies, d'architectures et de solutions techniques sur la base de la Réflexivité, pour mettre en place ou consolider des systèmes de systèmes.

1. prononcez $\epsilon R\ staR$

1.3 Défis

L'approche R-* se posant sur la base de la réflexivité comme nécessaire pour un système de systèmes, il convient de poser les défis suivants :

1. Identifier quels sont les éléments du système auxquels il faut appliquer la réflexivitié,
2. Trouver un moyen non-intrusif pour le métier des éléments et respectueux des exigences systèmes pour y appliquer la réflexivité de la manière la plus transparente possible sans nuire au système.

Soit où et comment appliquer la réflexivité dans un système de systèmes, tout en respectant ses exigences.

1.3.1 Défi 1 : Où appliquer la réflexivité ?

L'identification des éléments à rendre réflexif dépend du degré de variabilité que l'on souhaite offrir au système. La variabilité ne pourra être appliquée partout puisqu'elle implique l'ajout d'une dimension dynamique qui peut être incompatible avec des exigences systèmes liées au matériel ou à la vitesse d'exécution par exemple.

Cette thèse se focalisant sur les exigences d'un système, la variabilité sera identifiée par les fonctions attendues par un système. Ainsi, les domaines de recherche utilisés pour parvenir à cette identification seront la programmation orientée fonctionnalité (sous-section 2.3.1, page 21), et la séparation des préoccupations (sous-section 2.3.3, page 22), qui seront appliquées dans les modèles de distribution RPC (sous-section 2.5.1, page 30) et MOM (sous-section 2.5.4, page 31).

Une fois les éléments de variabilité identifiés, (soient dans notre cas, les fonctions des modèles de distribution RPC et MOM) il convient de leur ajouter le support de réflexivité.

1.3.2 Défi 2 : Comment appliquer la réflexivité ?

La réflexivité sera appliquée à l'aide de modèles réflexifs qui seront utilisés en tant que sur-couche des éléments du système. Afin de rester non-intrusif avec le métier des sous-systèmes auxquels on applique les modèles, les éléments à rendre réflexif sont modélisés dans ces modèles. Ainsi, le système réflexif final proposera un graphe de dépendance où les sous-systèmes embarqués dépendront de la couche réflexive, sans nécessairement en avoir connaissance (voir l'injection de dépendance dans le cas contraire, sous-section 2.3.2, page 21).

Deux types de modèles sont utilisés, afin de supporter la réflexivité au niveau de l'exécution des composants logiciels, ou au niveau de la description des exigences du système.

Le premier est un modèle à composant réflexif, provenant des approches à composant (sous-section 2.6, page 39). Celui-ci aura à charge d'embarquer les composants logiciels d'un système, et de les exécuter.

Le second utilisera l'ingénierie dirigée par les modèles (sous-section 2.3.5, page 23) pour définir et gérer les exigences d'un système.

La programmation orientée aspects (sous-section 2.3.4, page 23) permettra de lier fortement et de manière dynamique les deux modèles aux composants d'exécution du système.

1.4 Description de la Contribution

La contribution R-* apporte des réponses théoriques aux défis identifiés en utilisant différents types de modèles réflexifs pour enrichir les parties opérationnelles ou d'exigences du système. Et une réponse pratique à l'aide d'une implémentation d'un SoS (chapitre 6) pour justifier la faisabilité de R-*.

La figure 1.1 illustre l'implication de R-* dans la vie d'un sous-système (au bas de la figure), lié à d'autres éléments qui contribuent dans sa réalisation. Ces éléments sont son environnement, sa spécification et les autres sous-systèmes qui appartiennent à l'environnement, mais qui interagissent avec le sous-système clef. Ainsi, R-* intervient aussi bien autour de l'environnement et de la spécification, qu'au niveau des composants du sous-système et de sa communication avec d'autres sous-systèmes.

Concernant la partie communication, deux principaux paradigmes de communication utilisés par les systèmes distribués actuels sont identifiés par cette thèse, soient le paradigme d'Appel de Procédure

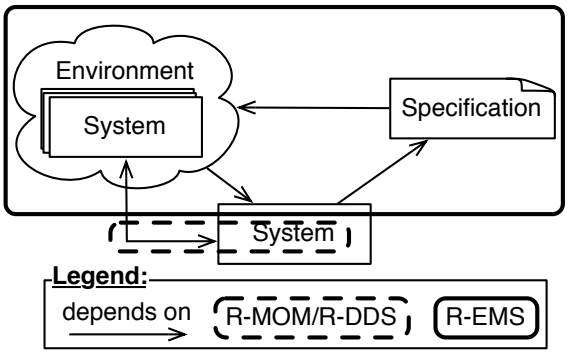


FIGURE 1.1 – **Vision complémentaire des contributions R-DDS, R-MOM et R-EMS dans un système de systèmes** – Représentation de l'environnement, de la spécification, des systèmes communiquant et des contributions autour d'un sous-système.

Distante, ou RPC (sous-section 2.5.1, page 30), et le paradigme des Intergiciels Orientés Messages, ou MOM (sous-section 2.5.4, page 31). Ils sont respectivement utilisés pour réaliser des communications synchrone et asynchrone.

Même si seules les réponses aux paradigmes de communication RPC et MOM sont proposées, les méthodes sont voulues suffisamment génériques pour pouvoir être appliquées dans des paradigmes existants, et idéalement futurs.

Pour y arriver, R-* doit s'enrichir de l'état de l'art, pour ne pas renouveler des erreurs passées, et surtout pouvoir apporter un réel incrément scientifique au sujet de cette thèse.

Dans l'état de l'art, plusieurs méthodes et langages de programmation utiles à l'adaptation ont été identifiés. De ce fait, ils ont été discutés et confrontés individuellement à la vision R-*. Or, il s'avère qu'ils répondent tous à des problématiques différentes, et souvent de manière complémentaire. Par exemple, Sven Apel et Don Batory ont montré à travers un cas d'étude d'implémentation d'une ligne de produits logiciel distribuée sur un réseau de pair-à-pair, que AOP (sous-section 2.3.4, page 23) et FOP (sous-section 2.3.1, page 21) sont complémentaires [Apel and Batory, 2006].

Le rapprochement avec les approches SOAs va dans un premier temps consister à réaliser un découpage fonctionnel couplé au modèle à composant réflexif FraSCAti (sous-section 2.6.6, page 45). Ce dernier s'appuie sur les possibilités de programmation sus-citées, et permet de personnaliser à grain très fin les fonctionalités d'un système.

Quand au support des exigences système, il va demander d'étendre le travail réalisé autour de la gestion des qualités de services (section 2.7, page 48) et de la formalisation de la spécification avec des notions de réflexivité pour s'assurer de la supervision et du contrôle sur les sous-systèmes.

Les modèles réflexifs doivent également être synchronisés avec les éléments d'exécution du système pour assurer une cohérence entre les informations modélisées et leur réalisation.

1.4.1 Utilisation du paradigme RPC

Le paradigme RPC connaît un grand nombre d'implantations. Seul FraSCAti (sous-section 2.6.6, page 45) sera retenu après une comparaison avec d'autres technologies (sous-section 2.5.11, page 38), parce qu'il implémente l'approche SCA (sous-section 2.6.5, page 45) afin de concevoir au mieux le paradigme RPC dans un environnement hétérogène. La partie maintenance du système s'appuie sur le modèle à composant réflexif Fractal (sous-section 2.6.4, page 43) qui a fait ses preuves en terme de solution reconfigurable à chaud, et de performance d'exécution.

De ce fait, si l'objectif est d'utiliser le paradigme RPC, R-* sollicite l'utilisation de FraSCAti, ou d'un modèle à composant possédant au moins les mêmes propriétés de réflexivité, provenant du monde Fractal, et de flexibilité de configuration, issue du monde SCA.

1.4.2 Utilisation du paradigme MOM

Le paradigme MOM connaît lui aussi une pléthore de travaux qui ont cherché à affronter un environnement dynamique, mais comme expliqué dans l'état de l'art (sous-section 2.5.11, page 38), aucun travail n'est suffisant pour répondre au mieux à l'approche globale d'un système de systèmes.

De ce fait, la thèse propose deux approches pour résoudre ce manque de prise en charge, soit l'ajout de réflexivité et d'interopérabilité au niveau des fonctionnalités du paradigme MOM et au niveau des fonctionnalités d'une implantation du paradigme MOM.

L'objectif étant de voir à quel niveau de granularité il faut descendre pour garantir le meilleur support de la réflexivité lorsque le but final est d'échanger des données de manière asynchrone, tout en satisfaisant des exigences spécifiques à un sous-système, ou globales.

Ainsi, la première contribution R-DDS (chapitre 3, page 55) s'attaque à la spécification DDS pour les systèmes RT-E, et la seconde contribution R-MOM (chapitre 4, page 67) remonte en abstraction au niveau du paradigme MOM pour proposer les mêmes ajouts de capacités, avec l'interopérabilité entre technologies de type MOM.

La conception des architectures considère dans un premier temps le même processus qui comprend l'identification des fonctions, puis leurs transformation dans un modèle à composant réflexif, qui offre ainsi de nombreuses autres fonctionnalités propres à une dimension dynamique. Puis un travail d'extension des solutions est réalisé pour poursuivre vers des préoccupations d'adaptation.

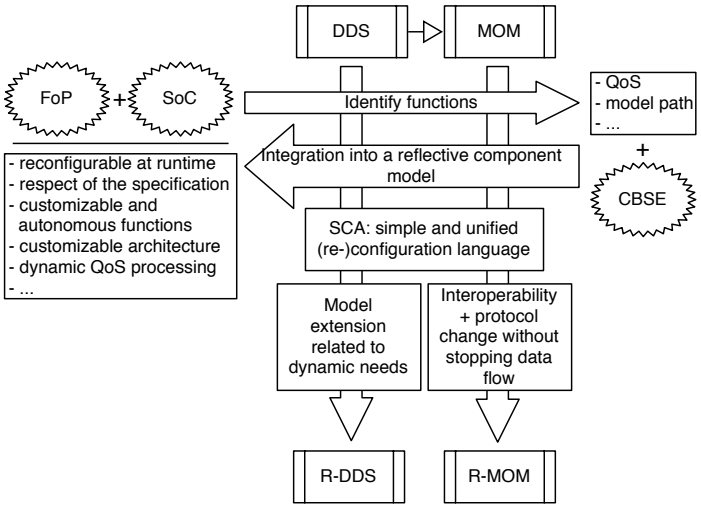


FIGURE 1.2 – **Transformation des paradigmes DDS et MOM vers les architectures R-DDS et R-MOM** – Découpage fonctionnel, intégration dans un modèle à composant réflexif et extension des paradigmes.

La figure 1.2 présente les méthodes utilisées pour enrichir les solutions existantes avec une dimension d'adaptation. En haut de la figure sont présentées les deux solutions à enrichir, soient la spécification DDS et le paradigme MOM. En dessous de chaque solution se trouve une flèche verticale descendante qui montre le résultat d'extension des solutions sous la forme des contributions R-DDS et R-MOM. Les autres formes graphiques sont utilisées pour décrire les procédés qui amènent aux différentes contributions. Soient dans un premier temps, l'utilisation des méthodes de Programmation orientée Fonctionalité (FoP) et de Séparation des Préoccupations (SoC) (en haut à gauche) pour identifier les fonctions (en haut à droite) des deux solutions d'entrée. Puis, de droite à gauche, on intègre les fonctions identifiées dans des composants réflexifs à l'aide de l'approche composant (CBSE). La conséquence de l'ajout de la réflexivité dans les solutions ajoute de nouvelles fonctionalités, telles que la possibilité de reconfigurer à l'exécution des fonctions devenues autonomes, ou encore de réaliser du traitement dynamique de qualité de services. Ensuite, on ajoute l'approche SCA pour unifier et simplifier la configuration des différentes solutions.

Finalement, en fonction de l'état des deux travaux d'enrichissement effectués, on ajoute des fonctionalités plus proche des préoccupations dynamiques. Par exemple, le modèle DDS est étendu pour que le modèle R-DDS puisse être plus simple à modifier à l'exécution (il devient possible de changer le type de donnée associé à des objets de publication/souscription). Et la solution R-MOM supporte l'interopérabilité entre technologies de communication asynchrone, ou encore, permet de changer de protocole de communication à l'exécution sans arrêter le flot de données.

1.4.3 Modélisation et gestion de l'environnement

Toute adaptation d'un système implique la modification de sa configuration, et éventuellement de sa spécification (sous-section 2.2.2, page 15), qui dépendent tous deux de l'environnement naturel dans lequel est exécuté le système.

Cette remarque introduit la troisième contribution, qui s'intéresse à la gestion d'un environnement, défini dans un modèle réflexif.

La difficulté de la définition d'un tel environnement est qu'il doit pouvoir contenir les éléments structurés (composants, propriétés non-fonctionnelles, choix politiques et économiques) et sémantiques (relations conceptuelles entre objets structurés), tout en permettant une dualité entre le modèle et tout élément du système.

De nombreux travaux connexes existent, soit lié à la modélisation du contexte ou des propriétés non-fonctionnelles, soit lié à la configuration des éléments structurés du système, ou encore lié à la définition des ontologies. Mais il reste encore un fossé à franchir pour pouvoir modéliser un environnement comprenant un mélange de domaines d'activité (politique, économique, etc.). Une analyse formelle de ces domaines permettrait de construire ou d'adapter la spécification ou la configuration d'un système pour qu'il réponde au mieux aux besoins exprimés par l'environnement.

Pour répondre au mieux à cette problématique, cette étude propose le canevas R-EMS (chapitre 5, page 83), qui définit un modèle réflexif, exécutable et extensible pour pouvoir modéliser un environnement, et réaliser du traitement sur son contenu. Le modèle est couplé à des moyens de communication réflexifs et interopérables avec tout élément d'un système utilisant les moyens de communication RPC ou MOM, afin d'être utilisable et accessible dans un milieu hétérogène et dynamique. Le modèle est également suffisamment générique et indépendant de toute représentation visuelle (texte, diagramme, etc.) afin de rester formalisé quelque soient les domaines d'activité ou les systèmes qui l'enrichissent et ainsi éviter un maximum les erreurs d'interprétation de son contenu. Finalement, à titre d'exemple d'utilisation, le langage R-EML, orienté programmation informatique, est proposé pour faciliter son utilisation par tout élément du système, depuis sa conception jusqu'à son exécution.

1.5 Plan général

Le manuscrit de cette thèse se découpe en trois grandes parties en plus de cette partie d'introduction qui regroupe la présentation du sujet (sections 1.1 et 1.2), l'identification des défis (section 1.3), un rapide aperçu des contributions (section 1.4) proposés précedemment, et qui se poursuit avec une discussion autour des travaux existants dans l'état de l'art (chapitre 2).

L'état de l'art comprend quatre sections qui apportent des éléments de réponse pour comprendre quels sont les constituants possibles d'un système, et des moyens de conception d'architectures réflexives. Ainsi sont regroupés, une base de connaissance (section 2.2), des méthodes de programmation pour l'application de la réflexivité (section 2.3), des descriptions de langages de programmation (section 2.4), des paradigmes de communication et des intergiciels (section 2.5), des modèles à composants (section 2.6) et finalement, des solutions de gestion de propriétés non-fonctionnelles (section 2.7).

La partie II comprend les architectures et les détails d'implantation des contributions, soient dans l'ordre de présentation, l'extension réflexive de la spécification DDS (R-DDS, chapitre 3), le MOM réflexif et intéropérable (R-MOM, chapitre 4) et le système de gestion d'environnement (R-EMS, chapitre 5). Cette partie se termine par une évaluation des contributions dans l'implantation d'un système de systèmes (chapitre 6).

La partie III conclut sur l'incrément scientifique apporté par la thèse à travers les contributions (section 7.1), et sur de possibles perspectives (section 7.2).

Finalement, la partie IV apporte des détails techniques sur certaines parties discutées dans les contributions. Soient des détails de conception d'une liaison DDS pour une architecture SCA (annexe A), et d'implantation du système R-EMS (annexe B).

Chapitre 2

État de l'art

Sommaire

Ce chapitre présente une base de connaissance nécessaire pour comprendre l'orientation scientifique de la réponse au sujet. Les aspects théoriques, scientifiques et pratiques sont discutés et confrontés à la vision de cette thèse.

2.1 Avant propos

Ce chapitre propose une base de connaissance dans le but de mieux comprendre l'axe de recherche utilisé pour résoudre les défis identifiés dans l'introduction (sous-section 1.3, page 5).

Chaque partie de la base de connaissance est présentée brièvement par un aspect théorique qui se veut général, et qui sera ensuite confronté par les points de vue issus de travaux scientifiques ou techniques et par celui de l'approche R-*.

La base de connaissance est organisée en deux étapes. Premièrement, des notions relatives aux points clefs du sujet seront présentées (sous-section 2.2), puis des axes de recherche relatifs aux défis de la thèse seront discutés.

La première phase s'intéresse aux points clefs du sujet, c'est-à-dire aux systèmes de systèmes (sous-section 2.2.1) et à l'évolution (sous-section 2.2.2). Mais elle s'intéresse également à des prérequis des systèmes hétérogènes actuels, soient l'interopérabilité (sous-section 2.2.3) et les architectures orientées services (sous-section 2.2.4). Ces deux derniers domaines de recherche sont adressés dans cette thèse car ils sont nécessaires pour parvenir à concevoir et utiliser un système de système évolutif dans la logique R-*.

La seconde phase se concentre sur où et comment appliquer la réflexivité dans les systèmes de systèmes, qui vont nécessiter de maîtriser des méthodes (sous-section 2.3) et des langages de programmation (sous-section 2.4), des paradigmes de communication (sous-section 2.5), l'approche des modèles à composants (sous-section 2.6) et finalement les moyens de gérer les exigences non-fonctionnelles (sous-section 2.7).

2.2 Base de connaissance nécessaire

La base de connaissance proposée ici tient compte du point de vue de cette thèse en terme de critères de nécessité, via les définitions des systèmes de systèmes (sous-section 2.2.1), de l'évolution/adaptation (sous-section 2.2.2), de l'hétérogénéité et interopérabilité (sous-section 2.2.3), et des architectures orientées services (sous-section 2.2.4).

2.2.1 Systèmes distribués et Systèmes de systèmes – SoS

Cette thèse considère un système distribué, ou réparti, comme un système déployé sur différentes plateformes. Chaque plateforme héberge un ou plusieurs acteurs qui utilisent des paradigmes de communication pour s'échanger des données, et satisfaire des exigences.

D'un point de vue général, une exigence est l'expression d'un besoin documenté sur ce qu'un produit ou un service particuliers devraient être ou faire [Wikipedia].

Dans le contexte des systèmes informatiques, la définition suivante est proposée ;

Exigence système :

Déclaration qui identifie un attribut, une capacité, une caractéristique ou une qualité nécessaires d'un système pour qu'il y ait une valeur et une utilité pour un utilisateur [INCOSE].

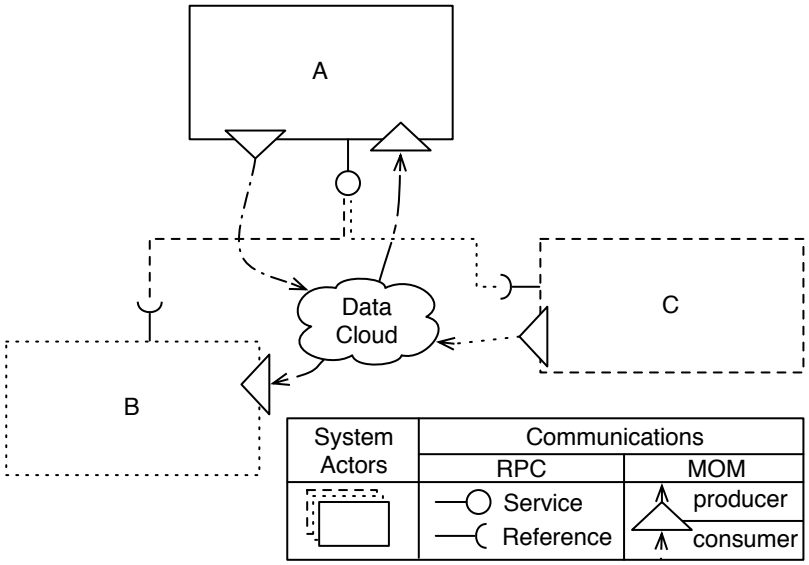


FIGURE 2.1 – **Vue très haut niveau d'un système réparti** – trois acteurs communiquant par RPC et MOM

La figure 2.1 montre un exemple de vue logicielle d'un système réparti hétérogène en représentant des éléments pouvant être concernés par des besoins d'évolution du système. Soient, les boîtes A, B et C qui sont des acteurs pour des adaptations de type fonctionnelles, et les liaisons de communications de type RPC et MOM pour des adaptations de type fonctionnelles et non-fonctionnelles (chiffrement des requêtes, disponibilité d'un site, etc.).

Les différents types de liaison (plus ou moins segmentés dans la figure) par paradigme de communication mettent en évidence qu'il existe une pléthore de technologies pouvant supporter les paradigmes de communication recherchées. Et leur contribution fait appel à la capacité d'interopérabilité qui devient nécessaire dans de tels systèmes, mais rarement mise en œuvre pour des soucis de choix politiques, économiques ou techniques (sous-section 2.2.3).

Derrière chaque type de liaison, il faut voir les aspects fonctionnels et non-fonctionnels, qui ne sont pas totalement interopérables, et pourtant ils répondent tous à une exigence de contrôle et de supervision de haut niveau attendue par le système réparti.

Au-delà des problématiques des systèmes répartis, le cadre de la thèse concerne principalement les systèmes de systèmes (SoS).

La définition prise en compte par cette thèse est la suivante :

Système de systèmes :

Un système de systèmes, ou SoS, est une collection de systèmes autonomes, dédiés ou orientés tâche qui concentrent ensemble leurs ressources et leurs capacités afin de créer un nouveau système qui offre plus de fonctionalités et de performances que la somme des fonctionnalités issues des systèmes constituants [Jamshidi, 2005]

Pour des facilités de lecture, un sous-système est l'équivalent d'un système constituant de système de systèmes.

L'une des principales difficultés des SoS provient de leur nature hétérogène. En effet, les sous-systèmes peuvent appartenir à des domaines d'exécution divers et variés, parmi les domaines des technologies d'information (IT), ou ceux de l'embarqué, ou encore ceux du temps-réel (RT). Voire même de plusieurs à la fois comme les domaines du temps-réel et de l'embarqué (RT-E) qui équipent les avions de ligne par exemple.

Un SoS doit donc assurer une cohérence dans un environnement dynamique et hétérogène, où la pluralité des exigences et des technologies utilisées nécessite de parfaire une interopérabilité entre sous-systèmes pour l'intégrité du système global.

Ainsi, l'interopérabilité devient un prérequis pour un système de systèmes.

Pour répondre efficacement à la problématique de l'évolution dans les systèmes de systèmes, il faut donc considérer des besoins à très haut niveau tels que ceux liés aux capacités d'un SoS. Mais aussi prendre en compte celles pouvant être sollicitées par les divers sous-systèmes, c'est à dire, celles liées aux domaines de l'embarqué, du temps-réel ou des technologies d'information.

Il existe au moins trois niveaux de famille d'architecture logicielle d'un SoS, soit du plus abstrait au plus concret, le niveau opérationel où l'on définit les exigences globales, les acteurs et les rôles. Le niveau système où sont spécifiés les logiciels et les intergiciels à utiliser. Finalement, le niveau composant représente la réalisation des exigences et des composants logiciels. Chacun de ces trois niveaux peut se décomposer en niveaux de la même famille, par exemple, au niveau composant, on peut retrouver par relation de composition, plusieurs niveaux de composants.

Le tableau 2.1 récapitule ces trois familles.

Table 2.1 – Niveaux d'architecture logicielle d'un Système de systèmes – du plus abstrait au plus concret

Niveau	Nom	Description
Haut	Opérationel	Exigences globales du système, définition des acteurs et des rôles
Moyen	Système	Définition des logiciels et des intergiciels
Bas	Composant	Réalisation des exigences et des composants logiciels

Système de Technologie d'Information – IT

Les systèmes d'information servent à manipuler un grand nombre de données pour un très grand nombre de clients. Le plus souvent, ils respectent le mode clients-serveur. Les données sont idéalement stockées dans une base de données pour faciliter les accès et les enregistrements concurrents.

Seules les infrastructures côté serveur sont considérées dans ce type de système, et donc les exigences sont définies au niveau des services offerts par le serveur. L'humain est très présent dans la boucle de ces systèmes, ce qui signifie que son échelle de temps permet de ne pas viser les performances maximales, seul le temps de disponibilité du système pour un client compte.

Système Temps Réel – RT

Un système temps-réel [Liu, 2000] est un système dont l'exigence principale dépend de la logique des résultats qu'il satisfait et d'un facteur temporel [Stankovic and Ramamritham, 1998]. Les exigences

temps-réel peuvent être facilement caractérisées par la phrase : "Une réponse délivrée trop tard devient une mauvaise réponse" (définition des systèmes temps-réel, partie 1, sous-section 2.1.1 [PLSEK, 2009]). Plus précisemment, les performances temps-réel requièrent un grand nombre de capacités, telles que la prédictibilité, la performance, le débit, la montée en charge [En-Nouaary, 2007], la sécurité, etc.

En dépit de cette liste non exhaustive des capacités imposées, le compromis entre performance et prédictibilité est le plus difficile à obtenir.

Pour faire ce compromis, il existe deux types de systèmes temps-réel, les systèmes temps-réel dur et les systèmes temps-réel mou.

Les systèmes temps-réel dur considèrent la délivrance d'un résultat de traitement non délivré dans les temps prévus comme un résultat nulle ou négatif (un bug). La caractérisation d'une telle exigence requiert de faire la preuve forte du métier d'un tel système, à l'aide de techniques d'analyses statiques et théoriques réalisées avant le déploiement qui font foi d'une probabilité d'échec très faible, souvent inférieur à 10^{-9}.

Les systèmes temps-réel mou considèrent un non respect de délivrance de résultat comme une information à identifier, en plus du résultat délivré.

Ainsi, les systèmes temps-réel dur sont utilisés dans des systèmes critiques qui engagent directement la vie d'êtres humains comme l'électronique des avions, alors que les systèmes temps-réel mou sont utilisés là où il y a un besoin d'évaluer le temps de traitement du système, comme les systèmes de contrôle de traffic aérien.

Les systèmes temps-réels doivent être déterministes dans toute opération qu'ils doivent effectuer, pour pouvoir contrôler si tout va bien d'une manière très rigoureuse.

Systèmes Embarqués

Les systèmes embarqués sont des systèmes qui sont utilisés par des matériels fortement contraints physiquement, et donc contraints en composants matériels, comme la mémoire ou le processeur qui doit respecter un certain refroidissement ou une faible alimentation.

Concrètement, il n'y a pas de définition précise avec des limites de contraintes matérielles données pour distinguer les systèmes embarqués des autres systèmes. C'est pourquoi cette étude va chercher à répondre au mieux aux problématiques de consommation mémoire et processeur en sollicitant un minimum de ressources possible.

2.2.2 Évolution et adaptation

L'évolution d'un système est une étape de son cycle de vie qui marque une modification importante de son métier, suite à une volonté de répondre aux exigences systèmes. Cette étude considère une évolution comme résultante de tâches d'adaptation dynamique ou statique, c'est-à-dire, effectuées durant son exécution, ou nécessitant l'arrêt du système.

En accord avec [Cheng et al., 2009], l'étude considère des tâches d'auto-adaptation du système comme des tâches d'adaptation dynamique et statique du système. Par ailleurs, toute adaptation du système est considérée comme une évolution.

La solution préconisée pour réaliser une adaptation dans les systèmes actuels est l'utilisation d'une boucle de contrôle [Shaw., 1995, Dobson et al., 2006, Kephart and Chess, 2003].

Cette boucle de contrôle est stimulée par le processus MAPE [Seborg et al., 1989]. IBM a ensuite proposé d'enrichir ce processus avec une brique de connaissance partagée, et de renommer le tout en MAPE-K [Computing et al., 2006, Huebscher and McCann, 2008] (voir la figure 2.2).

Ce processus est toujours utilisé par IBM pour promouvoir l'Architecture de Référence pour le Calcul Autonome (ACRA), qui est une solution de boucle de contrôle pour les systèmes autonomes (figure 2.3).

Ce dernier identifie les niveaux d'orchestration et de gestion des resources du système à l'aide de processus MAPE-K élémentaires, et de bases de données.

Voici en détail les différentes phases du processus MAPE-K.

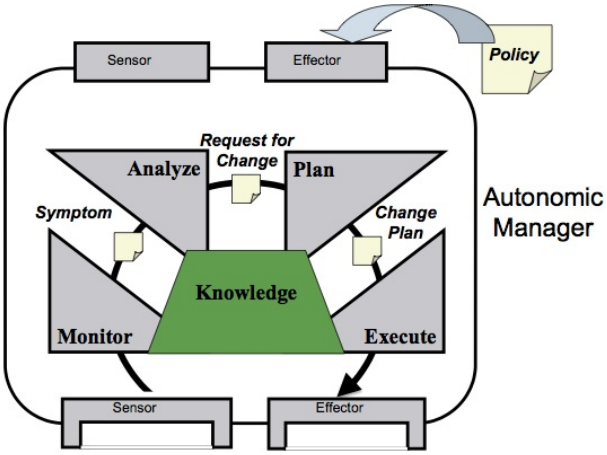


FIGURE 2.2 – **Processus d'évolution cyclique MAPE-K d'IBM pour gérer un élément** – Observation, Analyse, Planification, Exécution et Connaissance partagée

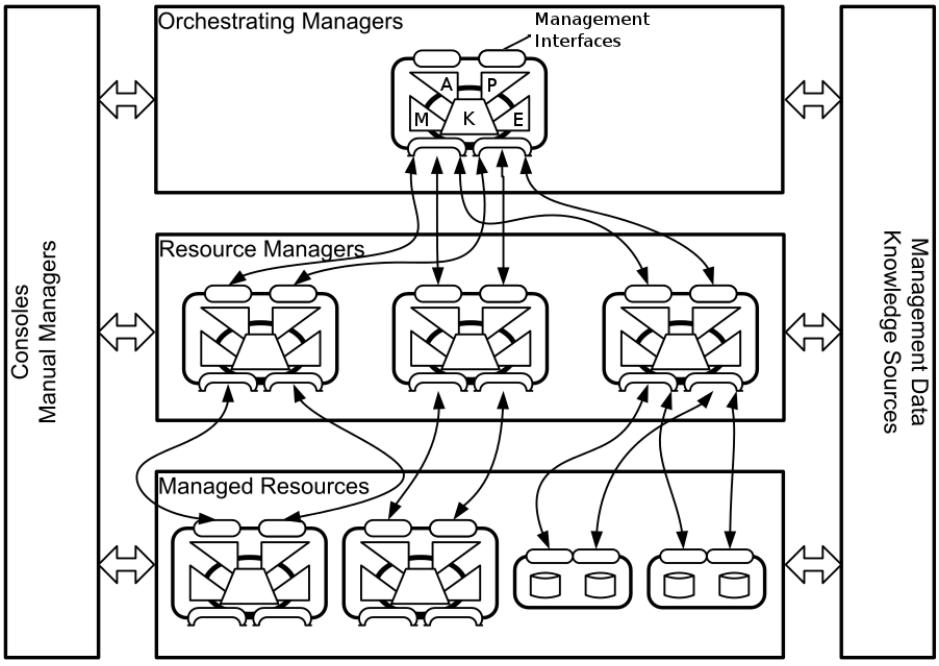


FIGURE 2.3 – **ACRA : Architecture de Référence pour le Calcul Autonome** – Gestion d'un système autonome par niveaux d'orchestrations et de resources (source : [Tamura, 2012])

Observation de l'adaptation

L'étape d'observation consiste à observer l'environnement d'exécution du système.

Puisqu'une adaptation peut provenir d'une exigence insatisfaite, il suffirait de modéliser toutes les exigences pour permettre au système de réagir de manière automatique à la non satisfaction d'une exigence. Or, cette modélisation est imparfaite aujourd'hui, soit incomplète (les travaux sur les propriétés non-fonctionnelles sont toujours d'actualité), soit trop difficile d'accès lorsqu'elle est partagée par plusieurs éléments d'un système (contrôle centralisé ou décentralisé).

Une fois qu'une exigence insatisfaite est identifiée, il faut passer à l'étape d'analyse de l'adaptation.

Analyse de l'adaptation

L'étape d'analyse consiste à évaluer tous les éléments concernés capables de satisfaire l'exigence et les éléments impactés par la réalisation de l'exigence.

Idéalement, cette étape recquiert de pouvoir parcourir le système pour déduire toutes les relations entre éléments. Cependant, outre la difficulté de jouer avec un modèle global dans un système qui mélange différentes technologies et modèles de représentation, la satisfaction d'une exigence est rarement isolée, et d'autres exigences peuvent être impactées. De ce fait, cette étape peut ne pas être exhaustive. La solution est donc de maximiser la satisfaction des exigences, en leur affectant un poids par exemple.

Des approches "multi-critère" où excèle THALES avec l'outil MYRIAD [Labreuche and Le Huédé, 2005] sont utiles à cette étape, pour peu que l'on ait la possibilité de modéliser les exigences et la possibilité de les lier à l'activité des éléments du système.

Une fois l'analyse réalisée, il convient de planifier l'adaptation.

Planification de l'adaptation

La planification de l'adaptation consiste à développer les opérations à appliquer au système pour satisfaire à nouveau ses exigences.

De manière à tester ces opérations, il convient de réaliser un modèle analytique du système (et si le temps et les ressources le permettent, réaliser une simulation du système), de manière à pouvoir identifier les analyses non perçues durant l'étape précédente car issues d'un modèle de représentation globale du système incomplet, ou à des effets de bord, ou encore à des défauts matériels.

De telles simulations sont possibles à l'aide des environnement virtuels, et dans ce domaine, le CLOUD [Grossman, 2009, Hoffert et al., 2010] a un grand rôle à jouer, puisqu'il permet de fournir et de copier une infrastructure physique (IaaS), une plateforme logique (PaaS) et même un logiciel (SaaS).

Mais là encore, cette étape dépend d'un environnement imprédictible dans l'absolue (à l'heure actuelle), dont l'environnement de simulation n'est qu'une représentation plus ou moins fidèle.

Lorsque les opérations d'adaptation ont été déterminées, il ne reste plus qu'à les exécuter dans le système.

Exécution de l'adaptation

L'étape d'exécution de l'adaptation consiste à exécuter des tâches d'adaptation.

Par mesure de sécurité, il convient de garantir une méthode de recouvrement [Avizienis et al., 2004] dans le cas d'une régression de satisfaction des exigences.

Quelque soit le résultat de cette étape, le système revient dans un état d'observation.

Connaissance partagée

Le composant de connaissance partagée entretient un modèle qui correspond à la configuration du système sur laquelle est rattaché la boucle de contrôle. Toutes les informations qu'il contient doivent être persistantes, et indépendantes des autres phases de traitement de MAPE-K.

2.2.3 Hétérogénéité et Interopérabilité

Hétérogénéité

Par définition un système hétérogène est un système comprenant tout type d'acteur, et soumis à différentes contraintes matérielles.

La problématique de l'hétérogénéité est liée à des systèmes qui durant leur exécution, doivent interagir avec des acteurs de plus en plus nombreux et divers. Confronté à un environnement dynamique, un tel système doit continuellement assurer une certaine disponibilité, et un contrôle sur tous ses éléments. La communication entre le système et un élément doit être assurée par l'élément ou par le système, c'est à dire que si le système ou l'élément ne parlent pas le même langage, c'est soit au système, soit à l'élément de s'adapter.

Idéalement, le moyen de communication entre tous les éléments d'un système pourrait être le même, mais dans les systèmes de système (SoS), cette vision des choses est rarement possible. Par exemple, le domaine de l'embarqué qui peut participer dans un SoS, introduit des contraintes fortes en terme de puissance de calcul et de mémoire disponible, et il apparait que l'élément ne soit pas capable d'utiliser n'importe quel moyen de communication. Dans ce cas, c'est au système de s'adapter au moyen de communication de l'élément. Le système devient donc interopérable avec tous ses éléments.

Ainsi, dans le cadre des systèmes de systèmes, un système hétérogène doit être interopérable.

Interopérabilité

L'interopérabilité est la capacité d'une ressource à interagir avec d'autres ressources, sans perte de données ou d'information non fonctionnelle. Elle est essentielle pour des besoins d'adaptation puisqu'elle sous-entend qu'une ressource doit rester accessible quelques soient le contexte d'exécution, et les types d'interaction possibles.

Cependant, cette capacité est très difficile à mettre en œuvre dans les systèmes actuels. Elle est souvent contrainte à des choix politiques divergeants pour peu que plusieurs acteurs d'un même système souhaitent avoir un maximum de responsabilité, et donc promouvoir un moyen de communication qu'ils maîtrisent, ou respectant un certain standard. Les moyens de communication respectant un même paradigme sont divers et variés. Par exemple, la section 2.5.4, qui s'intéresse aux MOMs, met en évidence qu'il existe plus d'une dizaine d'intergiciels orientés message, offrant plus ou moins de qualités de services, ou une API plus ou moins complexe en fonction des domaines adressés.

Les technologies existantes poursuivent une course effreinée dans le but d'offrir toujours plus de fonctionalités bas-niveau comme de la rapidité de traitement, l'utilisation faible d'empreinte mémoire, ou la prise en charge de qualités de services spécifiques, sans se concentrer sur des besoins haut niveau comme l'interopérabilité. Bien entendu, la dernière technologie développée est censée couvrir des nouveaux besoins identifiés dans un système. Mais si ce système souhaite profiter de la dernière technologie, il doit procéder au changement avec l'ancienne, avec le risque de perdre au passage une partie du contexte d'exécution mis en place lors de l'utilisation de l'ancien système, qui est incompatible avec la nouvelle technologie.

De plus, la plupart de ces solutions ne s'appuie pas sur des standards qui ont un impact beaucoup plus large sur les connaissances des différents architectes et intégrateurs systèmes. Le changement de technologie implique une maîtrise de la nouvelle technologie, une migration du contexte d'exécution et un risque d'intégration élevé dans un contexte de modèle de distribution de donnée. Le but de l'intéropérabilité est idéalement de s'abstraire de la maîtrise et de la migration de la nouvelle technologie, et de se concentrer uniquement sur l'intégration qui serait facilitée car rattachée à un contexte maîtrisé.

Les bus de service d'entreprise, ou ESB (sous-section 2.5.10), fournissent une plateforme qui vise l'interopérabilité entre intergiciels, en utilisant un bus de messages pour le transport de données. Même si dans un monde parfait, s'appuyer sur un bus de messages est une bonne idée, cela sous-entend également qu'il faut pleinement faire confiance au fonctionnement du bus. La qualité des échanges d'information dépend en grande partie de ce bus. Les ESBs utilisent des MOMs dédiés aux systèmes d'information, donc les ESBs ne peuvent être utilisés dans des domaines déterministes comme le sont ceux du temps-réel, ou fortement contraints comme dans le domaine de l'embarqué. Finalement, utiliser un ESB, c'est devenir

complètement dépendant du MOM utilisé, profitant de ses forces, et soumis à ses faiblesses. Si il tombe en panne, c'est toute la couche de communication du système qui tombe en panne.

Des travaux de recherche comme PolyORB (sous-section 2.5.9) ou comme [Na and Lee, 2002] résolvent l'interopérabilité avec des adaptateurs de communication locaux. Ainsi, si un acteur souhaite communiquer avec un autre acteur, il doit parler le même langage. La force de PolyORB est le mélange de paradigmes de communication, c'est à dire la possibilité de confondre les modes RPC et MOM. Par ailleurs, dans les travaux de recherche existants, l'interopérabilité est préservée autour du niveau fonctionnel de la requête, laissant de côté les aspects non-fonctionnels.

Même si des travaux de recherche proposent des résultats plus riches que certains produits de l'industrie, leur utilisation dépend surtout de leur réputation qui n'engage que celle d'un laboratoire de recherche confrontée à des groupes d'industriels comme IBM, Google, Microsoft, Oracle, etc. et qui ont eux aussi leur propre laboratoire de recherche. Au final certaines idées peuvent être récupérées de tels travaux, mais rarement l'intégralité, et des besoins haut niveau ne sont au final que faiblement couverts.

C'est une des raisons qui pousse cette thèse à considérer que soutenir une démarche d'interopérabilité pour des solutions utilisées par différents acteurs, nécessite de partir de bases communes à ces acteurs, c'est-à-dire, sur des standards qu'ils ont l'habitude d'utiliser, et qui sont accompagnés d'un panel d'outils annexes. Ces standards sont utilisés par un grand nombre d'architectes qui ne sont pas les seuls décideurs dans la conception de systèmes distribués. D'autres intervenants, travaillant sur d'autres domaines que l'architecture, ont aussi leur mot à dire. Un grand nombre de paramètres rentrent en ligne de compte pour terminer sur des décisions à la fois politiques, financières et techniques.

Démontrer que la nouvelle solution est dépassée ne suffit pas à imposer la nouvelle solution. Il faut anticiper sur les décisions stratégiques de conception, décisions qui privilégient la réutilisation de briques de composants existants, et non la formation des développeurs et architectes pour comprendre la nouvelle solution. Parier sur le long terme dans l'ingénierie logicielle est une idée difficile à considérer car le domaine de l'ingénierie logicielle lui-même évolue très vite. Ce qui équivaut intuitivement à dire qu'une entreprise pourrait passer son temps à former ses architectes, ou espérer que tous ses architectes passent leur temps à s'auto-former, sans ne jamais rien produire si elle voulait être toujours à la pointe de la technologie.

Une solution est la répartition des tâches, mais seuls les grandes infrastructures peuvent se permettre de telles formules. Dans la réalité, ce sont aussi ces entreprises qui ne souhaitent pas adopter de tels modèles de fonctionnement, et finissent par développer leur standard qui sont brevetés pour contrôler le marché.

Les centres de recherche (intra/extra-entreprise) continuent de fonctionner de cette manière, mais le défi de l'interopérabilité en informatique reste avant tout un défi scientifique, politique et économique car il s'adresse à tout le monde du milieu de l'informatique. Ne pas comprendre cela, c'est selon cette thèse, un très mauvais angle d'attaque pour résoudre l'interopérabilité. La progression doit être lente et proposer de nouvelles solutions sur un standard. Rendre d'autres solutions obsolètes obligera les utilisateurs de l'ancienne solution à trouver un intérêt plutôt qu'une crainte de devoir tout réapprendre. Puis une fois que l'ancienne solution est devenue inutile, un nouveau standard sera né, par la suite d'une évolution d'un ancien standard. Ce processus implique que le standard évolué devienne incompatible avec l'ancienne version.

2.2.4 Architectures Orientées Services – SOA

Les Architectures Orientées Services (SOA) sont un principe de conception qui se veut flexible et qui entre en jeu durant les phases de développement et d'intégration des systèmes d'information. Idéalement, on y retrouve les concepts suivants : réutilisation, modularité, composition, componentisation, interopérabilité, respect de standard, identification, catégorisation, surveillance et suivi des services.

Dans l'approche de cette thèse, les SOAs sont déterminants pour jouir de systèmes de systèmes hétérogènes et évolutifs. Voici en détail ces différents concepts.

Réutilisation

Une architecture doit pouvoir être réutilisable. Ceci favorisant le temps de développement d'un système à partir de parties déjà implémentées. Cette caractéristique permet de réunir différentes méthodes de

conception et de besoins ciblés pour offrir des modèles d'abstraction, capables de répondre à des besoins plus génériques.

Modularité

Modulariser une architecture est un principe de conception qui vise à décomposer les fonctions d'une entité. Permettant ainsi de se concentrer sur un traitement en particulier sans avoir à se soucier d'autres comportements de cette même entité qui ne sont pas concernés par le traitement. Dans le monde des services, il s'agit de découper un maximum le métier d'un service en "sous-service" autonomes. Ces sous-services sont ainsi plus facile utiliser, configurer et modifier.

Composition

Dans l'aspect modulaire des SOAs, la dépendance entre services se fait de deux manières, soit par relation de communication client-serveur, soit par composition de service. La composition de service permet de compléter de manière hiérarchique l'implémentation d'un service. Même si la relation de communication client-serveur et la composition remplissent techniquement et fonctionnellement le même rôle (idée de besoin de ressources), la relation de composition permet au niveau de la conception de l'architecture de ne spécifier que le strict nécessaire des besoins visibles de l'extérieur (principe d'encapsulation), les besoins internes sont compris dans la définition et n'ont pas besoin d'être connus en dehors de l'implémentation du service.

Componentisation

La componentisation est un principe qui amène une structure UML (sous-section 2.4.3) particulière dédiée à la notion de service client. Conceptuellement, il trouve sa place dans la description des architectures orientées service et techniquement, il existe de nombreuses solution visant à s'appuyer sur les description d'architecture orientées composant, telles que CCM (sous-section 2.6.1), OpenCOM (sous-section 2.6.3), OSGi (sous-section 2.6.2), et plus récemment SCA (sous-section 2.6.5).

Conceptuellement, le composant est une entité fonctionnelle qui offre des ports entrant et sortant, permettant de relier l'extérieur et l'intérieur du composant. En UML2.0, la spécialisation d'un composant de communication est connue sous le nom de connecteur, où les ports sont devenus des rôles. Un connecteur permet donc de gérer une forme de communication (implantée dans le connecteur) alors que le composant fonctionnel permet de spécifier un besoin fonctionnel.

Les différentes solutions d'architecture orientée composant ont étendu ces deux concepts, mais aucun modèle à composant ne respecte parfaitement le formalisme de l'UML2.0. Chacun est plus ou moins riche, et plus ou moins complexe, en fonction des besoins visés (voir sous-section 2.6 pour plus de détails sur les différentes solutions).

Interopérabilité

Voir la sous-section 2.2.3.

Respect de standard

Un standard est un modèle technique pivot pour faciliter les communications entre deux entités hétérogènes.

Chaque entité doit respecter le standard pour satisfaire la communication de données entre elles.

Identification et catégorisation des services

L'identification des services est un point clef dans la réalisation de SOAs, car il permet au client d'avoir un premier contact avec le serveur qui l'intéresse, les étapes suivantes seront de catégoriser le service, puis d'en tirer une première description pour se mettre d'accord sur le dialogue à utiliser pour communiquer avec le serveur.

La dernière approche innovante dans ce domaine reste celui des ontologies [Keskes et al., 2011]. Les ontologies réflèchissent au niveau conceptuel ou comportemental du système, contrairement aux approches objets ou UML qui ont une approche plus métier et structurelle des systèmes.

Surveillance des services

Pour garder un système orienté service cohérent, il apparait nécessaire de contrôler les activités définies par les architectes/développeurs et induites des requêtes de client extérieur au système, ou du système lui-même. La principale règle de cohérence imposée par ces systèmes sont la disponibilité, qui dépend bien évidemment du nombre d'appels des clients. Bien-entendu, il est possible d'instaurer une politique de nombre d'appels à des services, mais cela entraine bien souvent des listes d'attentes, et contraint fortement à redéfinir l'architecture pour répondre aux nouveaux besoins extérieurs.

Suivi des services

Comme conséquence de la surveillance des services, le suivi permet de réunir des informations sur le service lui-même et des services dont il a besoin pour promouvoir ses opérations à l'exécution. C'est une vue abstraite des dépendances d'un service à l'exécution.

Ce dernier point conclut la définition des SOAs de cette étude, à présent, il est temps de réduire le champ des possibilités de recherche en se focalisant sur le contexte des systèmes visés.

2.3 Méthodes de programmation pour l'application de la réflexivité

2.3.1 Programmation orientée fonctionalité et lignes de produit logiciel – FOP et SPL

La programmation orientée fonctionalité [Prehofer, 1997] propose de construire un programme en se focalisant sur les fonctionalités à fournir. Ainsi, elle nécessite d'identifier un programme comme un découpage fonctionnel.

FOP a donné naissance aux lignes de produit logiciel [Clements and Northrop, 2001]. Les SPL fournissent un ensemble de méthodes et d'outils pour la création d'un système logiciel depuis un ensemble de fonctionalités partagées.

Ainsi, les SPL sont déterminantes dans la satisfaction des exigences d'un système en terme de fonctionalité, puisqu'elle permet de déterminer un programme à l'aide d'une décomposition fonctionnelle. Russel Nzekwa a étudié et développé CORONA [Nzekwa, 2010] comme solution SPL réflexive, afin d'avoir une plateforme de composition de fonctionalités auto-adaptative pour les systèmes.

Ce travail est tout à fait pertinent dans l'approche R-* car elle respecte l'hypothèse du tout réflexif (sous-section 1.3, page 5).

2.3.2 Injection de dépendance et inversion de contrôle – DI et IoC

L'injection de dépendance et l'inversion de contrôle sont deux patrons d'architecture [Fowler, 2004] qui sont dédiés aux systèmes auto-adaptatifs. Utilisés d'une manière complémentaire, ils garantissent un couplage fort et dynamique entre les couches fonctionnelles et non-fonctionnelles d'un système.

L'injection de dépendance permet de configurer, durant l'exécution du système, une couche logicielle en fonction de dépendances requises. Ces dépendances peuvent ainsi être initialisées en fonction du contexte d'exécution, de manière totalement indépendantes des couches logicielles en ayant besoin. Ceci permet de faciliter une re-configuration de la couche visée durant l'exécution qui ne dépendrait que de ces dépendances.

L'inversion de contrôle introduit dynamiquement une opération dans un séquencement d'opérations de plus haut niveau. Ainsi, l'opération, de plus bas-niveau et non-prévue par la chaîne d'exécution définie statiquement, prend le contrôle sur la bonne exécution de cette chaîne. Le contrôle appartient ainsi à un

élément qui dépend pourtant du niveau d'exécution où est déroulée la chaîne d'opérations. Techniquement parlant, il s'agit d'une forme d'interception.

Pour réaliser ces deux techniques, il faut pouvoir parcourir le code ciblé, pour y identifier les dépendances ou les opérations recherchées. Soit, l'identification est faite depuis l'extérieur du code, soit avec l'utilisation du patron de conception "décorateur" [Gamma, 1995] pour marquer les endroits ciblés. L'utilisation des décorateurs semble plus simple car elle permet de s'abstraire d'une grande partie de la sémantique et de la logique de programmation du code ciblé, en déléguant au développeur le soin d'indiquer les endroits qui pourraient l'intéresser pour injecter du code ou intercepter des appels [Chiba and Ishikawa, 2005]. Même s'il s'agit de la méthode la plus utilisée, il faut reconnaitre qu'elle reste intrusive, dans le sens où le code visé doit s'adapter à la logique des IoC et DI pour programmer ses opérations. Mais elle est moins intrusive dans le cas d'une réutilisation de code, puisque la personne qui ré-introduit le code doit être au courant des couches logicielles de plus haut niveau. Dans tous les cas, agir directement sur le code n'est pas idéal dans une logique d'ingénierie logicielle pour un système adaptatif, car cela revient à contraindre le code à contenir des liens statiques avec les couches de plus haut niveau. Que se passe-t-il si l'on souhaite réutiliser ce code dans une autre couche de plus haut niveau ? Rien de plus que de devoir re-compiler le code, ou compter sur du code exécutable à la volée comme les langages de script, avec les qualités et les défauts que cette étude leur reconnait (voir le point de vue sur les langages de haut-niveau en sous-section 2.4.2).

2.3.3 Séparation des Préoccupations – SoC

La séparation des préoccupations [Dijkstra, 1974, Hursch and Lopes, 1995] justifie le fait qu'il est nécessaire de considérer un maximum de points de vue lors de la conception d'un système. Puis les confrontations de ces points de vue doivent être faites uniquement après avoir réussi à les satisfaire indépendemment les uns des autres.

L'objectif de cette méthode est de séparer un maximum ce qui peut l'être, et supposer ainsi que l'on peut mieux réfléchir sur des problématiques différentes lorsqu'on les attaque séparément, de manière à mélanger différents points de vue. Ainsi, toute relation devient découplée de leur logique de liaison. Par exemple, les relations de dépendances sont vues au second plan, alors qu'elles sont évidemment très importante durant l'exécution d'un programme.

La séparation des préoccupations vise à promouvoir le travail collaboratif, sans pour autant travailler avec du code spaghetti que pourrait être le traitement d'un modèle complet. La séparation des préoccupations est utile dans un premier temps pour travailler sur des éléments élémentaires. Ainsi, un logiciel peut être testé à grand fin, avec des tests unitaires par exemple. Bien entendu, il ne faut pas se limiter à ces tests unitaires. Une fois les éléments élémentaires testés en dehors de tout contexte d'exécution complexe et comprenant d'autres éléments d'exécution, il convient de reproduire ces tests avec des relations jouant avec d'autres éléments, etc. Jusqu'à reproduire petit à petit un maillage de plus en plus important de relations entre éléments élémentaires.

Dans un contexte d'adaptation, la SoC est utile pour constituer une architecture avec des points de variabilité suffisant pour pouvoir re-configurer une fonctionalité avec un minimum d'impact sur d'autres fonctionalités. Par exemple, c'est ce que font très bien les approches à composant (section 2.6). Ils découplent le métier d'un élément élémentaire avec la manière dont les relations qu'il entretient avec les autres éléments sont réalisées. Ainsi, la personne qui code un composant définit les types d'éléments avec lesquels il souhaite intéragir, mais d'autres personnes peuvent configurer le composant pour utiliser un moyen de communication particulier pour offrir les resources nécessaires au code métier. Par exemple, via un accès direct, ou bien en utilisant un proxy de manière transparente pour le développeur du composant.

La séparation des préoccupations est une vrai force dans la conception de l'architecture d'un système adaptatif, puisque dépendant du découpage fonctionnel de ce dernier, il ajoute en variabilité et facilite le travail collaboratif.

La faiblesse de cette approche est qu'il y a un risque à vouloir appliquer un découpage trop fin d'une architecture, avec des points de vue conflictuels pour pouvoir fonctionner à nouveau ensemble lors d'un assemblage de plus haut niveau.

2.3.4 Programmation Orientée Aspect – AOP

La programmation orientée aspect, ou AOP, est issue de l'approche SoC (sous-section 2.3.3) qui s'est intéressée à la composition des points de vue d'un logiciel.

AOP [Kiczales et al., 1997] insiste sur la possibilité de séparer les préoccupations en offrant des concepts capables d'enrichir du code existant avec des fonctionalités supplémentaires.

Ainsi, l'application des fonctionalités supplémentaires sur le code existant, ou tissage d'aspects, se fait en deux temps :

1. Définition de greffons (*advice* en anglais) qui sont les fonctionalités qui restent indépendantes du code visé,
2. ajout de points de coupe dans le code métier pour indiquer dans quelle partie du code sont censés agir les greffons.

L'exécution des aspects se produit à l'aide d'un tisseur qui associe les greffons aux points de coupe dans le code existant.

Il existe deux manières d'appliquer des points de coupe dans du code cible. La manière la plus intrusive est d'appliquer le patron de conception "décorateur" [Gamma, 1995] dans le code cible pour associer les points de coupe à des éléments du code, dans ce cas, elle devient spécifique au code visé. La manière la moins intrusive est d'utiliser un moyen d'identification de la structure cible agnostique du langage cible, comme les AST (arbres syntaxique abstraits, [Pfenning and Elliot, 1988]) pour y appliquer des points de coupe. Cette méthode est la moins utilisée car elle préconnise une traduction entre le moyen d'identification et la structuration du code cible. Pourtant, c'est aussi la méthode qui selon cette étude respecte AOP dans un contexte d'environnement dynamique. En effet, avoir le moins d'intrusion possible permet de pérenniser des propriétés non-fonctionnelles qui ne sont pas nécessairement spécifiques à un langage donné.

Il est important de bien comprendre qu'un aspect va agir autour d'une instruction, et non à l'intérieur de son exécution. Il s'agit d'une technique pour capturer l'accès à un programme existant, et puis de pouvoir agir avant et après. Il devient possible alors d'ajouter de manière transparente à l'utilisateur un accès par authentification, ou sécurisé avec une clef de chiffrement, ou tout autre traitement non-fonctionnel.

La force des aspects est de toujours pouvoir ajouter du pré/post-traitement sur tous les langages existants. Dans une approche d'adaptation, il convient de voir qu'utiliser un tisseur avec des greffons dynamique permet de modifier à l'exécution ces pré/post-traitements [Popovici et al., 2002, anter et al., 2010].

Par contre, le paradigme AOP joue en faveur des effets de bord, en agissant en sur-couche d'un code supposé valide. Une erreur des greffons peut empêcher le code visé de s'exécuter correctement, voir introduire un non-respect d'autres traitements qui dépendent d'une logique temporelle dans les systèmes temps-réels par exemple (sous-section 2.2.1).

Il existe de nombreux travaux de recherche qui ont implémenté AOP pour ajouter de la dynamicité dans les approches SOA. Par exemple, AO4BPEL [Charfi and Mezini, 2007] propose une modification d'un moteur d'exécution de BPEL pour supporter le tissage dynamique d'aspects. D'autres permettent de modifier le comportement interne de services exposés au niveau opérationnel [Baligand and Monfort, 2004]. Fractal (sous-section 2.6.4), SCA (sous-section 2.6.5) et FraSCAti (sous-section 2.6.6) utilisent AOP pour permettre à du code métier d'intercepter dynamiquement les appels faits sur le composant qui en est responsable. D'autres travaux facilitent l'intégration de qualités de services dans des systèmes [Tambe et al., 2009, Ortiz and Bordbar, 2008].

2.3.5 Ingénierie et architecture dirigée par les modèles – MDE

L'ingénierie dirigée par les modèles, ou MDE [Schmidt, 2006], est une approche qui vise à modéliser un système informatique dont le méta-modèle est défini spécifiquement pour ce type de système. Le méta-modèle et les modèles sont écrits dans le même langage, ce qui permet de raisonner sur un modèle de description commun à haut et bas niveau d'abstraction. Cette approche permet de procéder à des études de comparaison et de transformation entre modèles.

De nombreux outils existent, dont les plus connus et étudiés sont EMF [2] [Steinberg et al., 2008], et les DSL Tools [3] [Cook et al., 2007]. Les méta-méta-modèles proposés par ces outils pour modéliser des méta-modèles diffèrent mais s'appuient sur une représentaiton XML, ce qui garantit une certaine compatibilité et des transformations possibles entre méta-modèles inter-outils [Bézivin et al., 2005].

Une autre particularité de ces outils est aussi la possibilité d'utiliser un modèle exécutable, en complément des modèles statiques disponibles au format XML. Un modèle exécutable est à la base le résultat d'une génération de code depuis un méta-modèle. Le code généré est plus ou moins enrichi par d'autres opérations qui visent à se rapprocher un maximum de la description du méta-modèle. Puis une librairie est compilée et exécutée pour manipuler des modèles dynamiques crées ou chargés depuis un modèle statique. Ainsi, ces modèles exécutables préservent l'ensemble des caractéristiques fournies par les modèles statiques, mais dans une dimension dynamique. Ainsi, ils deviennent plus ou moins réflexifs, comprenant une API commune à tous les modèles issus du même outil DSL, mais aussi une API spécifique au méta-modèle. Pour faciliter les modifications à l'exécution, des paradigmes de programmation sont proposés. Le principal est le paradigme de la programmation par évènement, permettant ainsi à un utilisateur du modèle d'être notifié par la modification d'un élément du modèle. D'autres comme le paradigme transactionnel, ou le paradigme de vérification permettent de consolider la cohérence du modèle.

Dans la vision auto-adaptative de R-*, des travaux offrent des solutions pour l'utilisation de modèles pour accompagner ou péreniser l'utilisation d'un contexte d'exécution. Par exemple, [Cassou, 2011] invite à utiliser un modèle pour développer une spécification qui deviendrait ainsi couplée à toute autre activité intéressée par le modèle comme le développement d'un système ou son utilisation. C'est une idée tout à fait séduisante pour réduire les écarts d'interprétation de tout un chacun, et qui donne lieu à des implantations d'une même technologie qui finissent par être incapable d'interopérer ensemble.

L'approche MDE a ouvert la voie à d'autres axes de recherche, tels que les Architectures Dirigées par les Modèles, ou MDA [Kleppe et al., 2003]. Ces deux axes sont considérés par l'approche R-* comme primordiales dans le contexte de l'étude des systèmes auto-adaptatifs, avec notamment l'approche par modèle indépendant/spécifique de la plateforme (PIM/PSM) qui utilise une vue abstraite d'un modèle fortement lié à des modèles spécifiques à des plateformes. L'approche PIM/PSM permet par exemple de définir des besoins à haut niveau d'abstraction dans un PIM, puis de générer des PSMs qui faciliteront la transformation et l'adaptation d'éléments de réalisation sur une plateforme identifiée par les PSMs. La dernière étape est de pouvoir générer du code depuis un PSM vers les composants logiciels.

2.4 Langages de programmation

Les langages de programmation sont un moyen de définir les instructions à exécuter dans un programme, à partir des paradigmes ou méthodes de programmation.

Apparaissant comme la vision la plus basse du comportement d'un élément de système, il devient primordial de les connaitre pour identifier des points de variabilité possibles (sous-section 1.3.1, page 5), ou des moyens d'ajout de réflexivité à grain très fin dans un système (sous-section 1.3.2, page 5).

Un très grand nombre existe actuellement, et chacun est apparu avec les besoins de leur époque. Toutefois, ce manuscrit identifie cinq grandes familles, qui sont la programmation bas-niveau (sous-section 2.4.1), la programmation haut-niveau (sous-section 2.4.2), le langage de modélisation unifié (sous-section 2.4.3), les langages dédiés (sous-section 2.4.4) et les langages réflexifs avec l'exemple de Smalltalk (sous-section 2.4.5). Le programmation bas-niveau est aussi appelée programmation matérielle, car elle considère la manière dont fonctionne le matériel sur lequel il doit être exécuté. La programmation haut-niveau offre un niveau d'abstraction sur la programmation matérielle, dans le but de faciliter l'écriture des programme. Certains langages appartiennent aux deux familles, ou d'autres servent à compléter la première famille (voir la programmation par aspect dans la sous-section 2.3.4). Les différences entre niveaux sont également caractérisées par des caractéristiques techniques qui sont liées à la grammaire du langage et aux compilateurs utilisés pour traduire le langage en instructions machines. Parmi ces caractéristiques techniques, on retrouve notamment la taille des fichiers de code, l'empreinte mémoire, la vitesse de compilation et la vitesse d'exécution.

2. Canevas de Modélisation d'Eclipse `http://wwww.eclipse.org/emf/`
3. Outils de langage dédié à un domaine `http://www.domainspecificdevelopment.com/`

Toutefois, il est important de comprendre qu'il n'y a pas à l'heure actuelle de langage qui soit optimal quelque soit le système où il est utilisé. Certains langages sont plus complets que d'autres, mais il ne suffit pas d'un seul langage pour répondre aux problématiques de tous les systèmes.

2.4.1 Langages de programmation bas-niveau

Ces langages répondent à des besoins fortement spécifiques aux plateformes sur lesquelles ils sont exécutées. Ils limitent grandement les accès logiciels entre le matériel et les intructions qui y sont définies. L'empreinte mémoire est généralement donc réduite par rapport à l'autre famille des langages de programmation.

Le langage "Assembleur" est le langage le plus bas niveau car il demande de manipuler directement des instructions et des registres du processeur. C'est le moins couteux mais aussi le plus difficile à maintenir. De plus, il n'est pas portable sur toutes les machines, même si la plupart des instructions sont les mêmes, leur utilisation est proche de l'architecture physique du processeur, et dépend donc des spécificités du processeur, comme la taille des registres mémoire par exemple (8 bits, 16 bits, 32 bits ou dernièrement 64 bits), ou l'architecture (Intel, PowerPC, etc.).

La spécificité de ces langages vis-à-vis des plateformes sur lesquelles ils sont destinés à être exécutés impliquent qu'ils privilégient l'empreinte mémoire et la vitesse d'exécution, au détriment de la maintenance, et des règles de cohérence du langage. De ce fait, ils sont principalement utilisés dans des systèmes fortement contraints de type embarqués (voir la sous-section 2.2.1), même si concernant les algorithmes les plus complexes, rien n'assure que le développeur sera capable de développer un programme moins lourd et plus rapide avec un langage bas-niveau, plutôt que haut-niveau.

2.4.2 Langages de programmation haut-niveau

Les langages haut-niveau sont une abstraction du langage bas-niveau, ajoutant par exemple des règles de cohérence et des patrons de conception [Wolfgang, 1994, Gamma, 1995] pour faciliter l'écriture de certaines instructions redondantes et complexes, comme l'inclusion de librairies (ensemble d'instructions de langage), ou la gestion d'erreurs d'allocations mémoires, ou encore l'utilisation d'instructions complexes et utiles ("tant que", "exécution au moins une fois", condition à choix multiple, typage de structure, etc.). Les langages "C" et "C++" font parti des plus connus, et restent très proches des langages bas-niveau. Ils nécessitent à son utilisateur de gérer la mémoire allouée par chaque structure de donnée, et permet toujours d'écrire du langage "Assembleur".

Concernant l'exécution, on trouve les langages interprétés qui sont exécutés à la volée depuis les fichiers source, et les langages compilables. Cette étude caractérise deux familles de langage exécutable, ceux dont la compilation est spécifique à une machine, et ceux dont la compilation est portable, c'est-à-dire que le compilateur reste générique, mais l'exécution est assurée par une machine virtuelle. C'est le cas notamment du langage Java, ou des langages du .NET.

Là encore, ils ont chacun leurs points forts et points faibles en fonction de l'environnement dans lequel on souhaite les utiliser, ou les réutiliser. Mais l'important par rapport à cette étude est qu'ils permettent, pour des préoccupations de haut-niveau, d'écrire les mêmes algorithmes de programmation, avec plus ou moins de facilité, tout en respectant différentes philosophies de développement.

2.4.3 Langage de modélisation unifié UML et profiles SysML et ALF

L'OMG a contribué dans l'essor de l'OOP en concevant le langage de modélisation unifié UML [Group, 2007]. Apparu en 1995 et standardisé en 1997, il est aujourd'hui proposé dans une version 2 [OMG, 2001] comprenant entre autres un méta-modèle orienté composants (sous-section 2.6), ou encore le Langage de Contraintes Objet (sous-section 2.7.1). L'UML souhaite faciliter la conception d'un programme informatique, de manière structurelle et visuelle, via un formalisme proposé à travers un méta-modèle et treize types de diagramme, permettant de définir acteurs, rôles et opérations du programme.

Un profil UML est une spécialisation du méta-modèle pour un système donné, c'est-à-dire qu'un profil contient des éléments UML pré-définis et utiles pour modéliser un système.

La grande force de l'UML est également considérée comme sa plus grande faiblesse par cette thèse : UML est le bilan d'un grand nombre d'années d'études sur la conception de programmes informatiques, qui regroupe tous les contextes identifiés depuis sa création. Cette grande quantité de connaissance et de mélange de contexte d'exécution historique fait qu'il est jugé trop compliqué pour en comprendre toutes les subtilités, et trop complet pour être utilisé dans des environnements hétérogènes. Le maîtriser demande beaucoup de temps et d'efforts pour un architecte. De ce fait, et par manque de temps, chaque architecte en aura nécessairement une interprétation qui sera différente des personnes ayant pensé ou écrit la spécification. Pour preuve, aucun environnement de développement UML ne respecte dans son intégralité les annotations, et encore moins les sémantiques du formalisme UML (voir la mise en garde de la page d'accueil des tutoriels du profil SysML sur `http://www.uml-sysml.org/` qui évalue plus d'une vingtaine d'outils existants). Sachant que les architectes, issus de parcours professionnels différents, doivent travailler ensemble dans la conception d'un système, la conséquence d'un tel mélange d'interprétations faiblement formalisées est une difficulté de conception du système qui augmente de manière exponentielle avec les nombres de concepts et d'exigences souhaités.

La seconde critique faite par cette thèse est que UML a toujours été trop basé sur la manière structurelle et visuelle de considérer un programme informatique, alors qu'aujourd'hui, les systèmes sont de plus en plus hétérogènes [4], et requièrent la participation d'acteurs venant avec leurs propres solutions et de issus cultures différentes. Il faut donc réfléchir de manière sémantique pour mettre d'accord ces acteurs sur les types d'éléments employés. L'approche par ontologie (sous-section 5.3.1) est une solution qui devient nécessaire dans la conception de systèmes hétérogènes qui doivent communiquer avec d'autres systèmes hétérogènes puisqu'il est plus facile de se mettre d'accord sur la sémantique des éléments, plutôt que sur leur structure.

Ces critiques sont faibles par rapport à l'étendu des possibilités de l'UML dans la conception de programmes en général. Mais elles restent non négligeables dans notre vision de conception de systèmes hétérogènes et évolutifs.

Cette thèse approuvant dans sa globalité nombre de travaux proposés par la pensée UML, elle réutilise des concepts pour avancer vers une solution de conception plus souple et plus simple à utiliser. Aussi, le profil SysML et le langage ALF de l'OMG ont été étudiés.

Langage de Modélisation de Systèmes – SysML

Le langage de modélisation de systèmes [OMG, 2010, Weilkiens, 2007, Friedenthal et al., 2011] est un profil UML dédié à la modélisation de tout système. Il prend en compte un nombre considérable de paquets UML où chacun d'eux aura été identifié depuis les premières études menées sur la conception de systèmes. Ce qui signifie que historiquement, toutes les réflexions s'y retrouvent, et donc que les problèmes d'hier doivent se faire une place avec les problèmes d'aujourd'hui.

La figure 2.4 montre les paquets réutilisés ou étendu par le paquet UML4SysML.

La complexité apportée par tant de paquets UML implique de passer un temps considérable pour les apprendre et les maîtriser pour une seule personne, ou de solliciter un grand nombre d'architectes pour diviser les connaissances requises, et espérer une bonne entente entre tous pour les lier dans la conception d'un système. Un tel tissage de concepts est à lui seul une raison de se poser la question sur le nombre d'architectes à employer pour concevoir un système qui respecterait une telle approche. L'approche se veut exhaustive et couvrir tous les cas possibles et imaginables, mais dans un monde qui va très vite, est-ce vraiment la solution, ou ne vaut-il pas mieux proposer un méta-outil de conception de système qui favorise l'extensibilité des concepts plutôt que le nombre pour pouvoir répondre à des besoins en particulier.

Cette décision appartient aux concepteurs du système, mais cette thèse n'a pas pris le temps suffisant d'examiner scrupuleusement chacun des paquets utilisés par UML4SysML, et encore moins d'analyser le résultat d'un tel maillage de concepts. Il n'y a donc pas de prise de position définitive quand à l'intérêt de l'UML dans l'approche R-*, mais il y a un risque quand à sa compréhension et surtout sa réalisation dans les outils existants qui ne respectent pas complètement la spécification (d'autres exemples sur d'autres spécifications OMG ou non sont à considérer comme un problème général, comme le démontre

4. Les points communs entre des éléments hétérogènes sont davantages sémantiques que structurels.

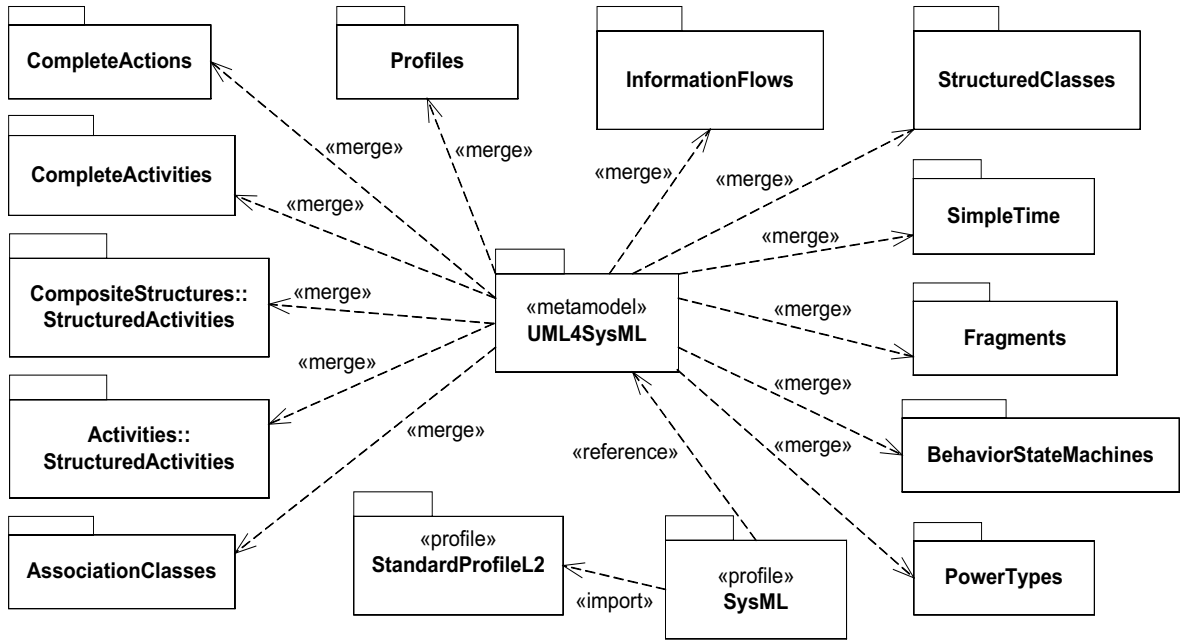


FIGURE 2.4 – **Diagramme de classe du méta-modèle UML4SysML** – Réutilisation et extension de paquets UML

les technologies DDS de la sous-section 2.5.6, ou AMQP de la sous-section 2.5.7).

Langage d'Action UML – ALF

Le langage d'action de fondation d'UML, ou ALF [Alf, 2010], sert de complément à SysML, sous la forme d'une représentation textuelle d'un modèle UML, avec des structures de contrôle ou d'exécution provenant des langages Java (description des classes, méthodes, et annotations) et OCL (gestion des séquences de valeurs). La philosophie UML du tout visuel est respectée ici avec environ deux cent cinquante neuf éléments du méta-modèle qui sont purement textuels (voir le diagramme de classe de la figure 2.5 où tous les éléments héritent du *SyntaxElement*).

L'objectif principal de la spécification est de compléter les diagrammes UML avec du code exécutable qui respecte le formalisme UML, mais qui couvre les aspects bas-niveau trop faiblement perçus par UML. Le respect du formalisme UML va même jusqu'à permettre de représenter entièrement un modèle UML. ALF propose une syntaxe proche du Java et profite également de l'expressivité du langage OCL (sous-section 2.7.1) pour la gestion des listes de valeurs.

La raison pour laquelle il est étudié ici est dû au fait qu'il est perçu comme une solution de définition de langage de programmation pivot entre les choix de conception de systèmes s'appuyant sur UML. En effet, l'utilisation de l'UML dans le développement d'un système soutient que l'on s'appuit sur des modèles de haut niveau de compréhension des exigences systèmes, à travers des diagrammes de représentation comportementale ou structurelle, qui seront déclinées en composants logiciels, avec l'aide de divers langages de programmation. Par ailleurs, il n'est pas jugé idéal pour les autres formes de conception, pour les raisons qu'UML est lui-même jugé insuffisant pour les systèmes auto-adaptatifs.

De plus, la philosophie visuelle de l'approche UML est appliquée sur ALF en le contraignant à une représentation syntaxique de son objectif, et respectant l'expressivité de Java et d'OCL. Pourquoi contraindre un besoin fort de couverture comportementale bas-niveau à une seule représentation possible ? Une solution plus conceptuelle aurait pu ouvrir les perspectives d'utilisation de ALF au delà des commentaires des diagrammes UML qui attendent d'être complétés par des instructions ALF. Et pourquoi ne pas proposer une autre représentation visuelle sous la forme d'un diagramme ALF en plus des diagrammes

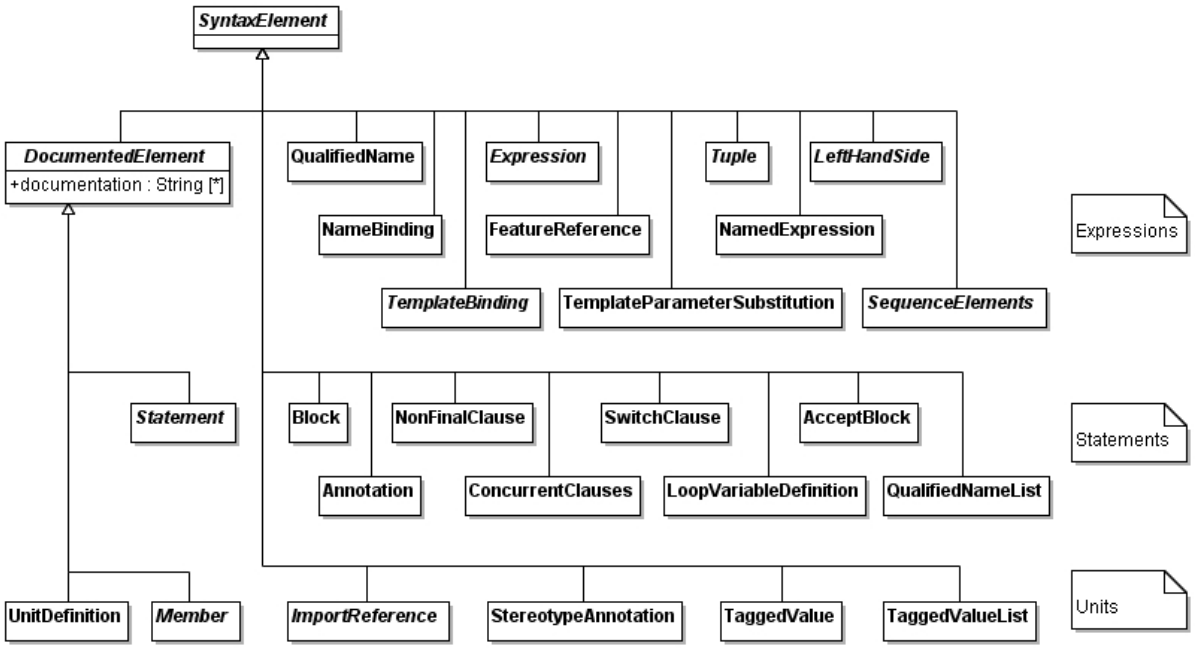


FIGURE 2.5 – **Vue haut niveau du diagramme de classe des éléments ALF** – expressions, éléments déclaratifs et unités (source : [Alf, 2010])

UML existants ? La réponse de l'OMG n'est pas identifiée, c'est pourquoi cette remarque restera dans une position de critique vis-à-vis de la philosophie R-*. Cette représentation qui est pourtant accompagnée de nombreux commentaires et de descriptions sur le fonctionnement de ses éléments démontre sa réelle vocation à être surtout utilisée dans des diagrammes UML, et non à interopérer avec d'autres langages de programmation ou processus de développement non issus de la philosophie UML. Il faudra attendre une seconde itération de l'écriture de la spécification pour que l'approche ALF, soit davantage ouverte à d'autres philosophies non orientées Java et OCL.

En résumé, ALF répond à un besoin fort de description structurelle et comportementale bas-niveau et à une couverture haut-niveau de UML, mais sa seule représentation textuelle disponible le contraint à rester utilisable dans des approches de type UML, avec pour limites, celles issues du Java et de l'OCL.

2.4.4 Langages dédiés à un domaine – DSL

Les langages dédiés à un domaine, ou DSL, proposent un langage spécifique à un domaine qui va pouvoir être transformé en modèles et méta-modèles exploités par d'autres personnes qui ont des préoccupations plus génériques ou abstraites. Ces mêmes préoccupations qui pourront plus tard exploiter les modèles avec une API générique inconnue des utilisateurs du langage, mais utilisée par les méta-modèles du même méta-méta-modèle.

À la manière de l'IoC (sous-section 2.3.2), cette approche permet de faciliter le travail de manière très efficace entre les utilisateurs du langage et du méta-modèle, tout en limitant les interactions entre eux. Les utilisateurs du méta-modèle conçoivent un langage en fonction des besoins de ses utilisateurs. Ces derniers utilisent le langage pour construire des modèles conformes au méta-modèle, qui seront exploités par les utilisateurs du méta-modèle. Les utilisateurs du langage ne connaissent que les langages qui sont spécifiques à leur domaine, et en conséquence, seuls les utilisateurs du méta-modèle ont connaissance des méta-modèles et des modèles.

Dans le cadre des systèmes auto-adaptatifs, [Haugen et al., 2010] propose une méthode pour ajouter de la variabilité dans un DSL, malgré la forte dépendance à un méta-modèle statique. Ce travail sera

réutilisé par l'approche R-* pour offrir un langage réflexif de modéliation de systèmes et d'environnements.

2.4.5 Langage réflexif – Smalltalk

Smalltalk [Goldberg and Robson, 1983] est un langage objet où tous les éléments sont réflexifs. C'est à dire que les instructions, les types, l'analyseur de texte et le moteur d'exécution font parti d'un même modèle réflexif, exécuté par une machine virtuelle, afin d'assurer une portabilité des codes compilés.

La communauté des programmeurs Smalltalk contribue dans un grand nombre d'applications commerciales [5], mais surtout dans des travaux de recherche scientifique qui ont donné lieu à des innovations d'ingénierie logicielle. Par exemple, on trouve les patrons de conception [Gamma, 1995], l'*Extreme Programming* (XP) [Beck, 2000], et le réusinage de code ("refactoring" en anglais).

Un exemple pertinent d'apport en ingénieurie logicielle est le concept de trait [Ducasse et al., 2006] qui peut se résumer en la composition fine de comportement par réutilisation des données internes d'une classe. Là où la relation d'héritage permet d'enrichir la définition d'une classe depuis une autre classe, les traits descendent en granularité pour la définition d'une classe à l'aide du contenu d'une autre classe.

Au final, Smalltalk est un langage totalement dynamique, où la modification du modèle n'a de limite que la cohérence résultante des modifications.

Dans le cadre de cette thèse, Smalltalk répond idéalement à la réponse du tout réflexif visée par l'approche R-* en s'intéressant à la granularité la plus fine possible à l'aide d'instructions réflexives. Hélas, Smalltalk recquiert une machine virtuelle pour être exécuté, et ne considère pas les contraintes de plateformes incapables d'exécuter ces machines virtuelles.

2.5 Paradigmes de communication et intergiciels

Une fois les paradigmes, méthodes et langages de programmation utiles à l'adaptation des systèmes définis, on retrouve logiquement la présentation des paradigmes de communication.

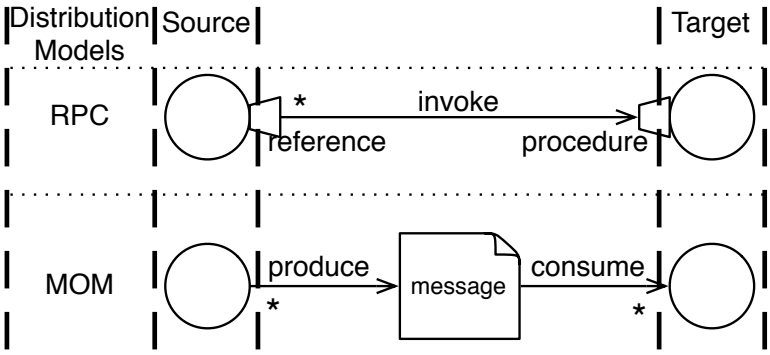


FIGURE 2.6 – **Paradigmes de communication étudiés** – Modèles de distribution de données RPC et MOM avec acteurs source et cible

Les paradigmes de communication sont apparus avec le besoin d'assurer des échanges d'information entre éléments du système, et entre processus d'exécution. Parmi un grand nombre de paradigmes de communication existants [Mehta et al., 2000], deux ont été étudiés dans cette thèse car principalement utilisés dans les systèmes distribués. Ils sont schématisés dans la figure 2.6.

Cette section présente une étude réalisée sur les paradigmes RPC (sous-section 2.5.1) et MOM (sous-section 2.5.4), avec les travaux relatifs, puis termine en présentant les approches multi-paradigmes que sont PolyORB (sous-section 2.5.9) et les ESBs (sous-section 2.5.10).

5. `http://www.pharo-project.org/about/success-stories`

2.5.1 Appel de procédure distante – RPC

L'appel de procédure distante est un moyen couplé pour réaliser une communication entre acteurs d'un système.

Dans la figure 2.6, le lien *invoke* montre cette connection directe entre acteurs cible et source.

Ce paradigme permet d'invoquer une opération définie dans un autre processus que celui réalisant l'appel. Pour ce faire, l'acteur source doit se mettre d'accord avec l'acteur cible à l'aide d'un contrat pour réaliser la communication. Une fois que la source et la cible se sont mises d'accord sur le contrat, la source construit une requête qui sera transmise à la cible de manière synchrone (la source attent la réponse de prise en charge de la requête). Puis le résultat de la requête est émis par la cible de manière synchrone (la source reste en attente du résultat) ou asynchrone (la source n'attend pas de résultat). La requête est constitué du nom de la procédure à appeler, de propriétés fonctionnelles (paramètres de la procédure, resource d'appel de retour dans le cas d'un appel asynchrone) et non-fonctionnelles (politique de sécurité, temps maximal autorisé pour l'invocation, etc.).

Le langage à l'origine du contrat doit être commun aux source et cible de la communication. D'où l'intérêt de disposer d'une architecture interopérable pour pouvoir respecter le contrat.

Deux implantations exclusives de RPC ont été étudié dans ce document, soit l'architecture CORBA (sous-section 2.5.2) et les Services Web (sous-section 2.5.3).

2.5.2 CORBA

CORBA [6] [Steve, 1997] est une architecture logicielle proposée par l'OMG, qui allie la souplesse des composants avec la communication des ORBs. Cette architecture, dont la première spécification est née en 1991, est aujourd'hui riche en spécifications et travaux divers.

L'objectif de CORBA fut de proposer un moyen de communication universel quelque soit les environnements à utiliser. Le langage de programmation en entrée est le langage de description d'interface (IDL). Ce langage est suffisamment générique pour être utilisé dans la génération de code spécifique à l'environnement visé, avec des types et opérations sans corps, qui doivent être remplies par les intégrateurs et développeurs système. Malgrés le fait que CORBA puisse fournir des opérations adaptées à un contexte particulier, la couche de communication est imposée par l'approche ORB. Ce manque cruel de flexibilité pour la communication devient un souci pour des raisons politiques et techniques car elle reste difficilement extensible, ou insuffisamment outillée. D'autres travaux plus à même de répondre à des besoins des systèmes dynamiques ont été proposés, mais hélas avec un faible succès. En effet, rivaliser avec une architecture connue d'un très grand nombre d'architectes dans les entreprises implique que l'on retrouve encore cette architecture dans les systèmes complexes actuels, faisant foi d'une certaine robustesse par la longévité de la solution et du nombre de personnes se portant garant de son efficacité. De ce fait, son utilisation devient aujourd'hui davantage la conséquence d'un choix politique et d'une stratégie financière à court terme, qu'une vraie décision technique sur le long terme. Même si cela fait bien longtemps que les ORBs ont été abandonnés par les chercheurs car des solutions plus efficaces pour le développement ou la maintenance de système ont été proposées (voir les approches WS ou SCA en sous-sections 2.5.3 et 2.6.5).

2.5.3 Services Web - WS*

Cette technologie est, à l'instar de CORBA, massivement utilisée dans le domaine de l'industrie depuis 2002, comme une plateforme d'intégration de couches logicielles existantes et hétérogènes [Linthicum, 2000]. De la même manière que pour CORBA, le protocole de communication est imposé, soit par la technologie REST, soit par l'utilisation du XML dans la description des données, service et transport de messages.

La réelle plus value des WS sur CORBA est certainement sa vision plus proche de l'approche SOA. Chris Peltz a introduit le concept d'Architecture Orientée Services Web (WSOA) [Peltz, 2003] pour démontrer que l'on peut utiliser les WS pour implémenter l'approche SOA. Ce qui n'est pas le cas de l'approche CORBA qui a nécessité l'ajout de l'approche composant pour se rapprocher des mêmes problématiques (sous-section 2.6.1).

6. `http://www.omg.org/corba/`

Les WS comprennent neuf spécifications regroupées sur les thèmes de l'interopérabilité, de la fiabilité, des transactions, des évènements, des projets, de gestion et d'administration, de sécurité, de description des services et des processus métiers. Finalement, de nombreuses implantations existent dans différents langages, parmi lesquelles on trouve Axis [7] et Axis2 [8], .NET Framework Web Services [9] et gSOAP [10] utilisées au cours de cette étude.

Les WS proposent le Langage de Définition des Services Web (WSDL) pour définir un contrat RPC, et le Simple Protocole d'Accès à des Objets (SOAP) pour le transport de messages. Tous deux sont structurés avec le langage de balise étendu (XML), ce qui rend les WS très verbeux, donc non idéal pour des systèmes fortement contraints, et a fortiori, non idéal pour les systèmes RT-E, voir certains systèmes de systèmes.

2.5.4 Intergiciel orienté messages – MOM

Les intergiciels orientés messages sont un paradigme de communication à couplage lâche, c'est-à-dire que la durée de vie d'un message est par défaut indépendante de celle de son producteur. En conséquence, le mode asynchrone entre entités productrices et consommatrices est privilégié.

La figure 2.6 montre qu'un acteur source du modèle de distribution de type MOM, produit un message. Ce message devient autonome, et indépendant de sa source. Il peut être ensuite récupéré par une ou plusieurs cibles. Les sources et les cibles de messages sont appelés respectivement producteurs et consommateurs de messages.

Il est possible de définir un contrat logique (appelé "sujet") pour le mode publication/souscription ou un contrat physique (appelé "queue") en mode point à point entre sources et cibles de message. Dans les deux cas, le consommateur peut récupérer la donnée suivant deux modes, soit par "tirage" (*pull*, synchrone) où la cible demande explicitement de consommer un message, ou par "poussée" (*push*, asynchrone) où la cible est notifiée de la disponibilité d'un message. Cette fois, le contrat peut être totalement indépendant du métier des sources et des cibles.

Il existe un très grand nombre d'architectures et d'implantations basées sur le paradigme MOM.

Parmi les architectures on retrouve le Service de Messages Java (JMS) ou encore le service de distribution de données (DDS) du groupe de gestion des objets (OMG). Malheureusement, il n'existe pas à l'heure actuelle d'architecture MOM, et la conséquence est qu'en l'absence de standard, il devient difficile d'interopérer entre architecture et implantation du paradigme MOM.

Parmi les implantations connues, on retrouve un grand nombre implantant JMS comme ActiveMQ [11], OpenJMS [12], JORAM [13] et JBossMQ [14]. D'autres implantant le Service de Distribution de Données (DDS) comme OpenSplice [15], RTI, OpenDDS. D'autres MOM existent comme 0MQ [16], JGroups.

Le couplage lâche garanti par cette architecture est propice dans des environnements dynamiques, et sont donc essentiels dans l'approche de ce document.

Les trois implantations du paradigme MOM les plus utilisées dans les systèmes distribués ont été étudiées, soit le service de message Java (sous-section 2.5.5), le service de distribution de données (sous-section 2.5.6) et le protocole de queue de message avancé (sous-section 2.5.7).

2.5.5 Service de Message Java – JMS

Le service de message Java est la spécification [Hapner et al., 2002] du paradigme MOM pour le langage Java. La spécification se concentre sur l'API Java de manipulation des messages, à l'aide d'un ensemble d'interfaces, mais ne s'intéresse pas à l'aspect protocolaire.

7. http://axis.apache.org/axis/
8. http://axis.apache.org/axis2/
9. http://msdn.microsoft.com/en-us/library/ms950421.aspx
10. http://gsoap2.sourceforge.net/
11. http://activemq.apache.org/
12. http://openjms.sourceforge.net/
13. http://joram.ow2.org/
14. https://community.jboss.org/wiki/JBossMQ
15. http://www.prismtech.com/opensplice
16. http://www.zeromq.org/

Ainsi, JMS réutilise des mécanismes propres à Java. Certains messages à échanger sont prévus par la spécification pour embarquer des données de type primitif ou de collection (données de type nombre, texte, tableau d'octets, liste, dictionnaire), alors que les objets de type non pré-définis doivent hériter de l'interface *java.util.Serializable*.

L'API du receveur étant générique, c'est à lui de vérifier les types de message qu'il reçoit.

Des valeurs de qualités de service accompagnent ces messages, mais là encore, leur réalisation dépendra du moteur d'exécution.

Cette spécification propose les deux paradigmes de distribution de messages, c'est à dire, soit en point à point, soit en publication et souscription.

L'utilisation d'une API facilite la portabilité du code, et donc, permet de s'abstraire du moteur d'exécution. Par ailleurs, ce MOM dépend trop fortement du code Java, et la conséquence est que les messages ne pourront pas parvenir à des plateformes d'exécution ne supportant pas Java. La spécification indique tout de même que dans ce cas, il faut regarder du côté des messages pouvant transporter des tableaux de données, mais là encore, il faudra s'entendre davantage avec le moteur d'exécution qui définit la couche de transport, plutôt qu'avec l'approche JMS.

2.5.6 Service de distribution de données – DDS

DDS est une spécification OMG d'une API de service de distribution de données en mode publication et souscription pour les systèmes temps-réel et embarqués [OMG, 2007] (voir la section 2.2.1). Contrairement à JMS, l'API proposée est donnée à l'aide du langage de description d'interface (IDL) qui était définit à l'origine pour CORBA (sous-section 2.5.2). Ainsi, l'API est dite générique pour tout type de langage (section 2.4), mais la réalisation dépendra toujours de l'interprétation du moteur d'exécution.

L'API est divisée en deux parties, soit DCPS pour la couche de communication, et DLRL pour la couche d'exploitation des données reçues. En conséquence de quoi, la couche DLRL est optionnelle.

Cette étude a pu tester trois implémentations de DDS : OpenSplice [17], RTI [18], OpenDDS [19]. L'étude de DDS au cours de la thèse a été encouragée par l'entreprise THALES dans le cadre d'une solution d'intergiciel asynchrone pour le temps réel et l'embarqué dans le projet ANR ITEmIS [20]. Mais d'après la spécification de l'extension d'interopérabilité pour DDS (DDS-I), il existerait une dizaine d'implémentations connues.

L'intégration de DDS dans le projet ITEmIS fut ma première contribution. La solution OpenSplice a été choisie pour deux raisons : c'est l'implémentation qui respecte le plus la spécification (mais pas complètement), et c'est aussi le produit où THALES a le plus d'expertise, en terme de contributions, implantations de cas d'utilisation et réalisations dans des grands systèmes.

Ce travail d'intégration qui comprend de la communication flexible dans un système ITEmIS se présenta comme idéal pour démarrer l'étude de la thèse.

La maîtrise de cette technologie fut apportée en théorie par une pléthore de spécifications et de papiers de recherche OMG [OMG, 2007, OMG, 2009c, OMG, 2009b, Xiong et al., 2007, Hoffert et al., 2009] mais aussi par des tests concrets offerts par OpenSplice.

Modèle de distribution de données

DDS est un modèle de publication et de souscription orienté données, où des entités (*Entity*) s'accordent sur un sujet (*Topic*) et un domaine physique (*Domain*) pour réaliser des échanges de données.

Le diagramme de classe de la figure 2.7 montre les différents types d'entités disponibles pour publier et souscrire à des données. Ainsi, DDS requiert de s'inscrire à un domaine grâce à un participant de domaine (*DomainParticipant*). Ce participant de domaine est capable de créer des participants, tels que les entités de publication (*Publisher*), de souscription (*Subscriber*) et des sujets (*Topic*) relatifs à un objet *TypeSupport* créé à partir d'un fichier IDL décrivant la structure des données à échanger. Finalement, l'écriture et la lecture de données sont assurées respectivement par des *DataWriter*s et des *DataReader*s,

17. `http://www.prismtech.com/opensplice`
18. `http://www.rti.com`
19. `http://www.opendds.org`
20. `http://research.petalslink.org/display/itemis/ITEmIS+Overview`

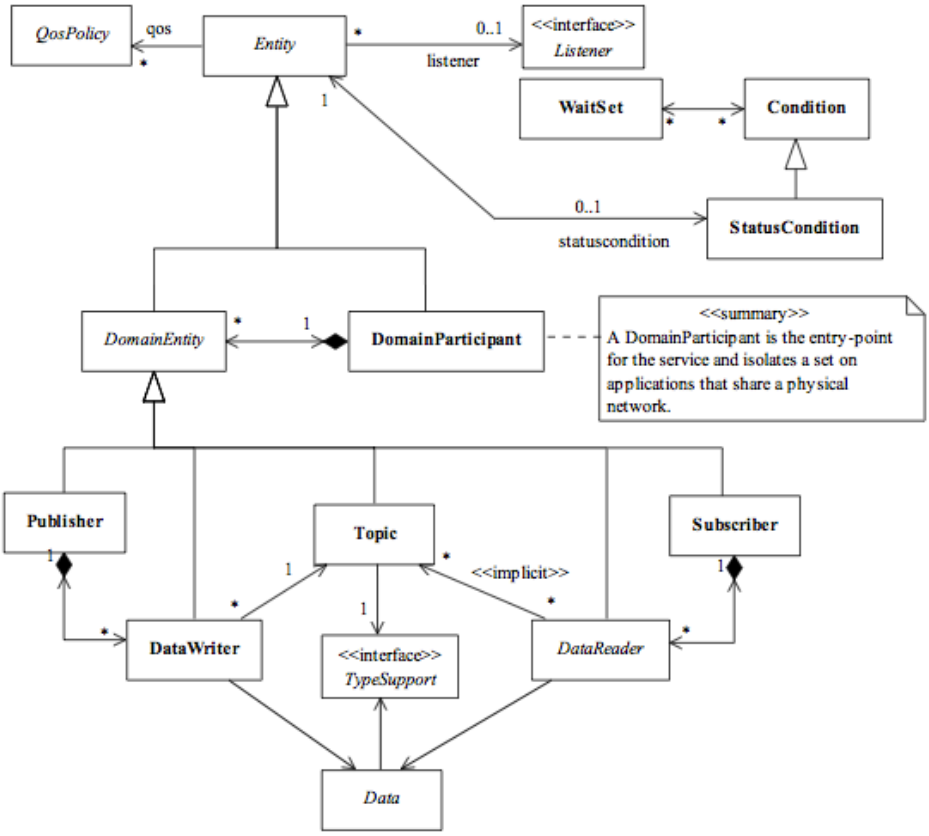


FIGURE 2.7 – **Diagramme de classes des entités DDS** – Entités couplées à une condition de status et à un ensemble de qualités de service (QoS). Source : [OMG, 2007]

qui sont spécifiquent à un sujet. Le *DataWriter* est créé par un *Publisher*, et le *DataReader* est créé par un *Subscriber*.

Une entité produit des évènements qui peuvent être récupérés en mode "tirage" et "poussée" (*pull* et *push*, sous-section 2.5.4) respectivement à l'aide d'une référence vers une condition de status (*StatusCondition*) et de l'implantation du patron de conception Observateur [Wolfgang, 1994] (*Listener*).

Finalement, toutes les qualités de service du modèles sont à configurer sur les entités (*QoSPolicy*).

Traitement d'évènements du modèle en mode "tirage" DDS propose le concept de *Condition*, qui couplé à un object *WaitSet*, permet de traiter des évènements émis par le modèle DDS en mode "tirage". Ainsi, ce concept se décline en 4 types. La condition de status (*StatusCondition*) permet de capturer des évènements issus d'une entité (mise à disposition, incohérence de configuration, etc.). La condition de Garde (*GuardCondition*) émet des évènements de l'application. La condition de lecture (*ReadCondition*) permet d'être mis au courant de la disponibilité d'une donnée auprès d'une entité de lecture de données. Finalement, la condition de requête (*QueryCondition*) hérite de la condition de lecture, et propose un filtre appliqué aux données disponibles depuis une entité de lecture de données.

Traitement d'évènements du modèle en mode "poussée" Le mode "poussée" est possible à l'aide du patron de conception Observateur [Wolfgang, 1994] implanté par les entités. Toute entité est observable par un observateur.

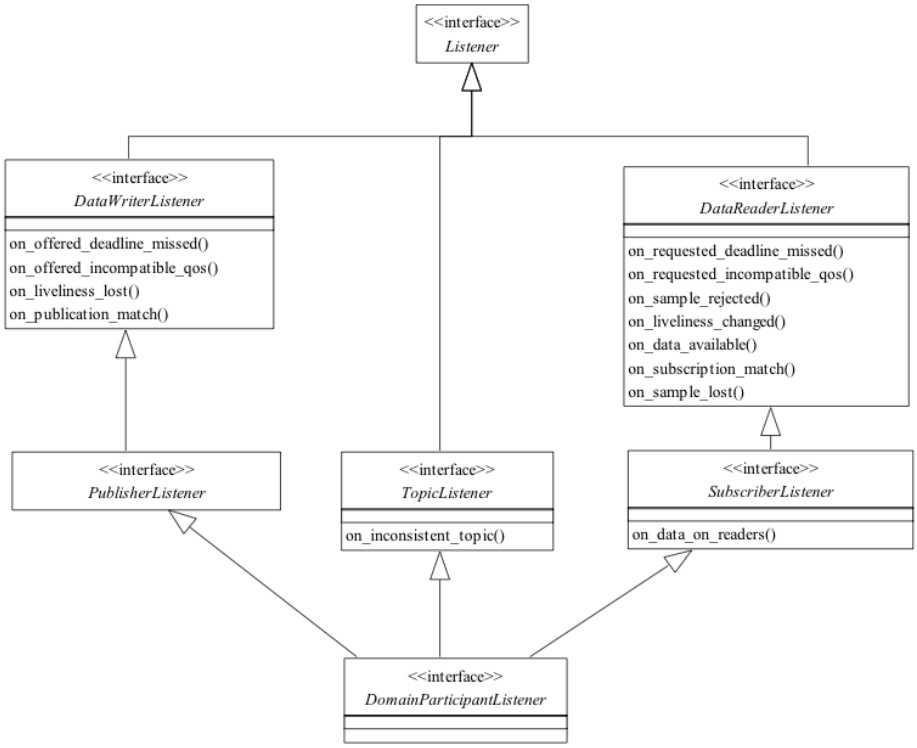


FIGURE 2.8 – **Diagramme de classes des *Listener*s DDS** – Héritage par niveau de responsabilité

La figure 2.8 représente l'ensemble des *Listener*s existants par type d'entité, avec un héritage par niveau de responsabilité permettant ainsi la propagation d'évènements vers les entités parentes. Par exemple, la disponibilité d'une donnée au niveau d'un *DataReader* pourra être perçue par son *Subscriber* parent, et son *DomainParticipant*.

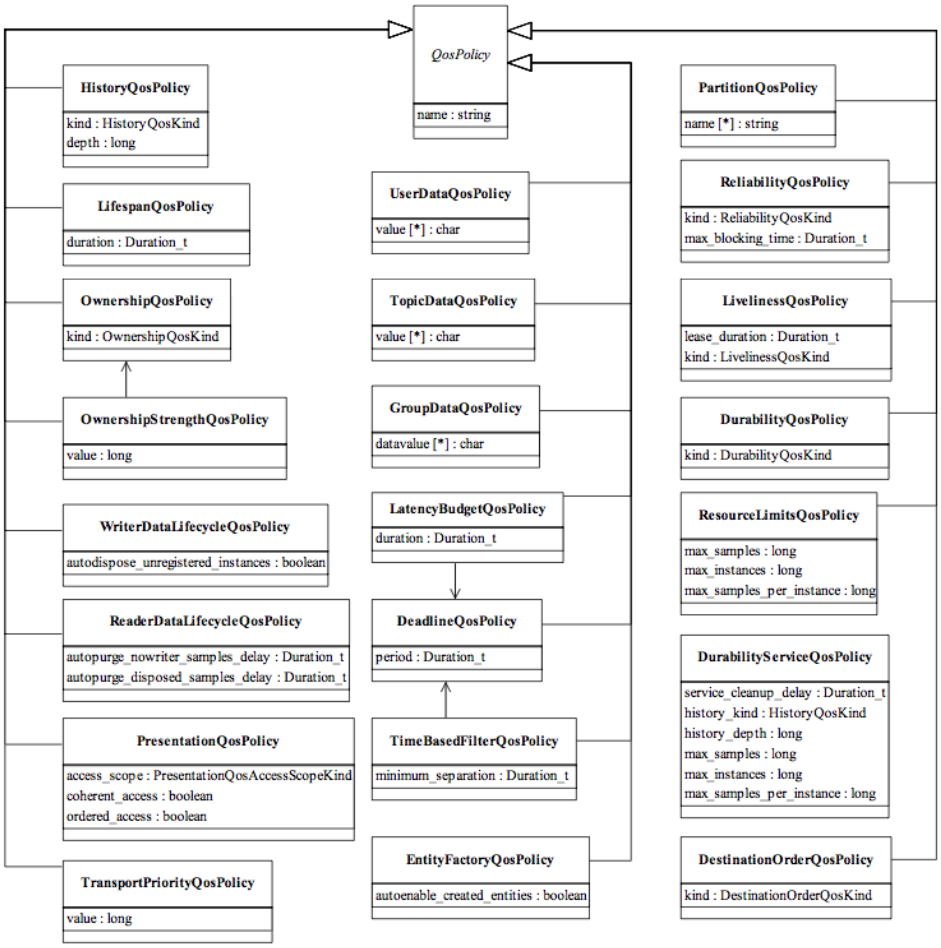


FIGURE 2.9 – **Diagramme de classes des politiques de qualités de service de DDS** – Héritage depuis le concept de politique de qualité de Services

Politiques de qualités de service L'orientation temps-réel de DDS lui confère un panel de 21 qualités de service représentées dans le diagramme de classe de la figure 2.9. C'est un nombre très important comparé à d'autres intergiciels orientés message, comme JMS (voir la sous-section 2.5.5) qui ne supporte qu'une dizaine de qualités de service plus proches des systèmes de technologies d'information (IT). Ainsi, DDS offre toutes les QoS supportées par JMS qui accompagnent l'envoi des données parmi les types de durée de vie, d'ordonnancement, ou de persistance. Les autres QoS sont propres au modèle DDS comme le partitionnement d'un domaine, ou en rapport avec l'activité des entités de publication et souscription de donnés (auto-disponibilité dans le système, temps maximum d'inactivité avant suppression).

Difficultés d'utilisation

L'orientation de DDS pour les domaines de l'embarqué lui confère une grande performance mais une flexibilité très faible.

Le nombre conséquent de qualités de service couvertes par DDS est sa plus grande force comparé aux autres solutions orientées MOM, mais la configuration de ces qualités de service est très compliquée. Les causes sont les dépendances entre QoS difficiles à résoudre car non spécifiées, et surtout résultantes d'un nombre considérable de combinaisons de QoS possibles. Pour preuve, la spécification OMG des connecteurs DDS pour LwCCM (voir la sous-section 2.6.1) [OMG, 2009b] définit deux profiles DDS - ou ensembles de valeur de QoS - pour que les données se comportent comme des informations d'état ou d'évènement.

Une conséquence directe de ce grand nombre de QoS est la difficulté à configurer et faire évoluer le système. Toutefois, la spécification propose un moyen d'intercepter une configuration incohérente avec le modèle (voir le paragraphe sur les conditions de status). Mais là encore, les implémentations sont capables d'intercepter de petites incohérences du modèle, mais les plus importantes peuvent faire tomber le moteur d'exécution.

2.5.7 Protocole de queue de message avancé – AMQP

AMQP [21] est une étude menée sur l'interopérabilité entre intergiciels de type MOM par un grand nombre d'acteurs et spécialistes des MOMs. La spécification [AMQ, 2010] définit une architecture aux niveaux transport et réseau. Ainsi, il est possible d'utiliser ce protocole sous l'API JMS (travail réalisé par la technologie JORAM [22]). La force de cette approche est qu'elle est agnostique des langages de programmation connus, en dehors du format XML pour définir la structure des données. Elle considère aussi les fonctionalités identifiées par les MOMs utilisés dans les domaines des systèmes d'information, mais non d'autres requises par les systèmes RT-E, comme la gestion des instances de données de DDS.

Même si la spécification est complète, cela ne suffit pas à ce que l'interopérabilité soit possible, car elle nécessite que les autres participants doivent la respecter, c'est à dire, développer les connecteurs nécessaires. Or, pour des choix politiques, économiques et techniques (sous-section 2.2.3), le développement de ces connecteurs est rarement fait. Quand les connecteurs sont développés, rien n'assure qu'ils soient complets. Par exemple la technologie RabbitMQ [23] qui est de plus en plus utilisée pour sa simplicité de configuration, et ses nombreuses implantations [24], et systèmes d'exploitation supportés [25], permet de n'envoyer ou de recevoir que des tableaux de données. Cette technologie ne gagne pas en réputation pour son respect du standard, mais pour son ouverture aux différents langages et environnements d'exécution.

Il semble donc que même un groupe d'industriels proposant une spécification ne soit pas suffisant pour offrir de l'interopérabilité entre MOMs. Ce qui justifie la remarque faite sur le point de vue de l'étude pour fournir de l'interopérabilité (sous-section 2.2.3).

21. `http://www.amqp.org/`
22. `http://joram.ow2.org/`
23. `http://www.rabbitmq.com/`
24. C/C++, Java/JVM, Ruby, Python, .NET, PHP, Perl, Ada, Erlang, Lisp, Haskell et Ocaml.
25. Windows, Linux, MacOSX, OpenVMS, Amazon EC2, Web Messaging et Android

2.5.8 DREAM

DREAM [Leclercq et al., 2004] est un travail de recherche Inria qui spécialise l'approche Fractal (section 2.6.4) pour l'implantation de technologies MOMs. L'idée est de proposer une API générique et dynamique pour gérer les messages (création, suivi d'instance, suppression, découverte par type), puis ensuite, les adapter à la technologie MOM visée avec des interfaces de composants qui proposent des fonctions similaires à celles attendues par les MOMs testés. L'implantation de JMS (sous-section 2.5.5) avec DREAM a montré des résultats plus efficaces en terme de performance des temps de calcul que l'intergiciel JORAM/JMS.

Même si le travail est très intéressant d'un point de vue couverture des fonctionalités du paradigme MOM, avec de nombreux points de flexibilité, il ne couvre pas d'interopérabilité entre MOM, ni de spécialisation de méthode de sérialisation (la sérialisation Java est utilisée), et l'API de création de messages nécessite la réflexivité Java et reste lourde pour les domaines du temps-réel.

La dépendance forte de DREAM avec le langage Java est un pari risqué en tant que proposition d'intergiciel adaptatif, puisqu'il pré-suppose que Java sera toujours utilisé à terme, rayant par la même occasion toute possibilité de communication avec les solutions Microsoft. Par ailleurs, DREAM ne semble pas suffisamment dédié à tout type de système, puisqu'il impose une description des messages via l'utilisation de l'interface "java.lang.Serializable", et l'utilisation de la réflexivité Java.

2.5.9 PolyORB

En réponse à l'approche M2M [Baker, 2001], PolyORB [Quinot, 2003, Pautet, 2001] est un travail de recherche d'un intergicel qui vise l'interopérabilité entre intergiciels, et non uniquement aux ORB comme pourrait en faire croire le nom. PolyORB est aussi appelé intergiciel schizophrène, parce qu'il permet de jouer avec plusieurs personnalités applicatives.

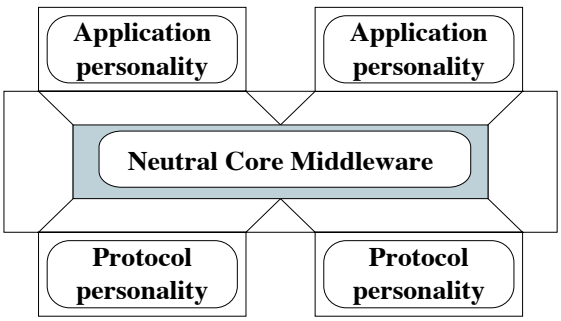


FIGURE 2.10 – **Architecture en couche de PolyORB** – Couches applicative, neutre et protocolaire (source : [Vergnaud et al., 2004])

Pour ce faire, PolyORB est constitué de trois types de couche logicielle, soit une couche neutre qui sert d'intermédiaire à des couches applicatives et protocolaires (voir la figure 2.10). Les couches applicatives et protocolaires sont personnalisables par l'utilisateur, c'est à dire que l'approche s'adapte à tout domaine applicatif ou protocolaire, quelque soit le paradigme RPC ou MOM visé.

Ainsi, les couches applicatives construisent des requêtes qui sont traitées par la couche neutre, puis re-distribuées aux couches protocolaires. La couche protocolaire transmet la requête à d'autres couches protocolaires qui si elles sont implantées par PolyORB pourront reconstruire une requête pour la transmettre à la couche neutre disponible, qui la traitera à son tour pour finir par la remettre aux couches applicatives en attente de pouvoir répondre à des requêtes.

La figure 2.11 montre un exemple d'un tel processus entre un client DSA (Ada 95 Distributed System Annex) et un serveur CORBA.

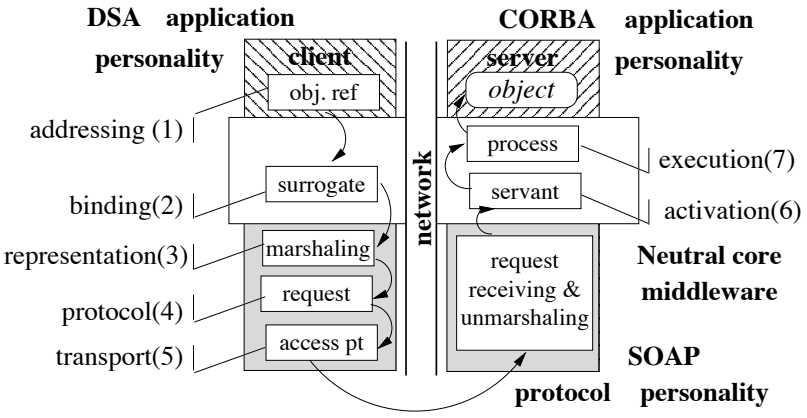


FIGURE 2.11 – **Parcours de requête d'invocation** – Depuis un client DSA vers un serveur CORBA (source : [Vergnaud et al., 2004])

Dans un contexte de système auto-adaptatif, l'intérêt de PolyORB est très intéressant car il est ouvert à toute forme de communication RPC ou MOM.

Malheureusement, l'interopérabilité de PolyORB ne s'intéresse qu'à l'aspect fonctionnel des requêtes. Les aspects non-fonctionnels sont différents par paradigme de communication (la durée de vie d'un message traité dans un système MOM n'a pas de signification pour une requête dans un système RPC). Le mélange des paradigmes proposés rendrait le mélange des propriétés non-fonctionnelles impossible, puisque d'un point de vue opérationnel, ils ne sont pas destinés à appliquer les mêmes fonctions.

2.5.10 Bus de service d'entreprise – ESB

L'architecture ESB [Papazoglou and van den Heuvel, 2007] propose une solution d'interopérabilité entre intergiciels, quelque soit leur nature (RPC ou MOM). L'idée est, comme PolyORB (sous-section 2.5.9), de se servir d'un modèle de communication intermédiaire entre les différents intergiciels, où les commandes seront transportées par un intergiciel.

Cette solution répond à des besoins d'adaptation du système pour pouvoir communiquer avec tout élément du système.

Cependant, certains points de flexibilité devraient être apportés. La forte dépendance à un bus de message le contraint à faire confiance à ce bus, malgrè les fonctionalités de sécurité ou de tolérance aux fautes proposées par les intergiciels connectés à l'ESB.

L'approche "modèle pivot" proposée par l'ESB est intéressante pour faciliter l'interopérabilité entre intergiciels, mais la forte dépendance dans un seul intergiciel en particulier peut faciliter la faute d'un point d'accès (SPF) et réduire l'efficacité de la communication globale à l'étendu des fonctionalités de ce même intergiciel. S'il s'avère insuffisant, le remplacer sera une tâche lourde et nuisible vis à vis du temps de disponibilité du système.

2.5.11 Bilan

L'étude des paradigmes de communication est à présent terminé. Il est temps de faire un bilan des analyses de ces paradigmes dans le contexte des systèmes de systèmes.

Toutes les solutions étudiées sont reportées dans le tableau 2.2 et soumises à des critères élémentaires et nécessaires aux systèmes de systèmes. Ainsi, l'évaluation simplifiée comprend les critères de configuration (Conf.), d'indépendance avec la plateforme d'exécution (PIM), de re-configuration ou de montée en charge

TABLE 2.2 – Efficacité des paradigmes de communication par rapport à des exigences de système auto-adaptatifs et hétérogènes

Solution	Conf.	PIM	Re-Conf./MC	NFP	Interoperability		Domain	
					Data	Transport	RT-E	IT
RPC	+/-	+++	+++	+++	+++	-	+++	+++
CORBA	++	+++	+	+	+++	+	+++	++
WS	+/-	+++	+	++	+++	++	+	+++
MOM	+/-	+/-	+	+	+++	+/-	+++	+++
JMS	-	++	++	+	++	-	-	+++
DDS	-	+++	++	++	+++	++	+++	++
AMQP	-	+++	-	+	+++	+++	-	+++
DREAM	+++	++	+++	+	-	++	-	+++
PolyORB	-	++	-	-	+++	++	+++	+++
ESB	+/-	+	+	++	+++	+++	-	+++

(Re-Conf/MC), de traitements non-fonctionnels (NFP), d'interopérabilité des données et du transport, mais aussi des domaines visés (RT-E ou IT).

Chaque critère comprend une valeur de couverture ou de support par solution, proposée dans l'ordre croissant des valeurs de l'ensemble suivant : $\{-, +, ++, + + +\}$. La valeur "+/−" correspond à une évaluation indéterminée. Ces valeurs sont déduites des remarques et des discussions posées au cours des précédentes sous-sections.

Le tableau met en évidence qu'il n'existe pas de moyen de communication qui réponde de manière optimale à toutes les exigences attendues par les systèmes auto-adaptatifs et hétérogènes.

Certains sont plus aptes à fournir des solutions pour les domaines RT-E, d'autres aux domaines IT. La plupart adresse l'interopérablité des données, alors que d'autres s'intéressent davantage à la couche transport.

Cependant, une remarque globale met en avant le fait que des solutions multi-paradigmes sont un non-sens de par le non-respect de la dimension non-fonctionnelle qui devient incohérente si on cherche à l'appliquer (voir l'incohérence de l'aspect multi-paradigme PolyORB, en sous-section 2.5.9).

Les approches multi-paradigmes sont à proscrire des architectures, mais qu'elles restent toutefois complémentaires au niveau système.

2.6 Approches composants

La notion de composant choisie par cette étude respecte la définition donnée par Szypersky :

"Un composant logiciel est une unité de composition avec seulement des interfaces contractualisées et des dépendances de contexte. Un composant logiciel peut être déployé indépendemment et est sujet à la composition par une troisième partie." [Szyperski et al., 2002]

Cette définition met en avant trois propriétés. La première est la relation qu'il entretient avec l'extérieur via des interfaces et des dépendances. Ensuite, un composant peut être déployé sur une plateforme d'exécution, indépendamment des autres composants logiciels de la plateforme. Finalement, un composant apporte la propriété de composition dans le développement logiciel.

Ces propriétés se retrouvent dans [Crnkovic et al., 2009], à travers les concepts génériques que sont le composant, l'interface (offerte ou attendue) et la liaison d'interface. S'ajoutent les propriétés extra-fonctionnelles, qui sont appelées ici propriétés non-fonctionnelles, et qui ne font pas partie de la définition de Szypersky, mais qui accompagnent la partie fonctionnelle/métier du composant.

Même si l'UML2 (sous-section 2.4.3) a proposé un méta-modèle comme standard de modélisation de composants, les modèles à composant existants ne respectent pas stricto-sensus ce standard et réutilisent ces concepts. Ils prennent la liberté de les renommer, ou de compléter leurs comportements en fonction des

besoins logiciels qu'ils adressent (par exemple, un modèle à composant comme AUTOSAR [26] qui adresse des besoins logiciels pour l'embarqué dans les voitures ne va pas proposer des composants de même constitution que des composants génériques comme OpenCOM présenté dans la sous-section 2.6.3).

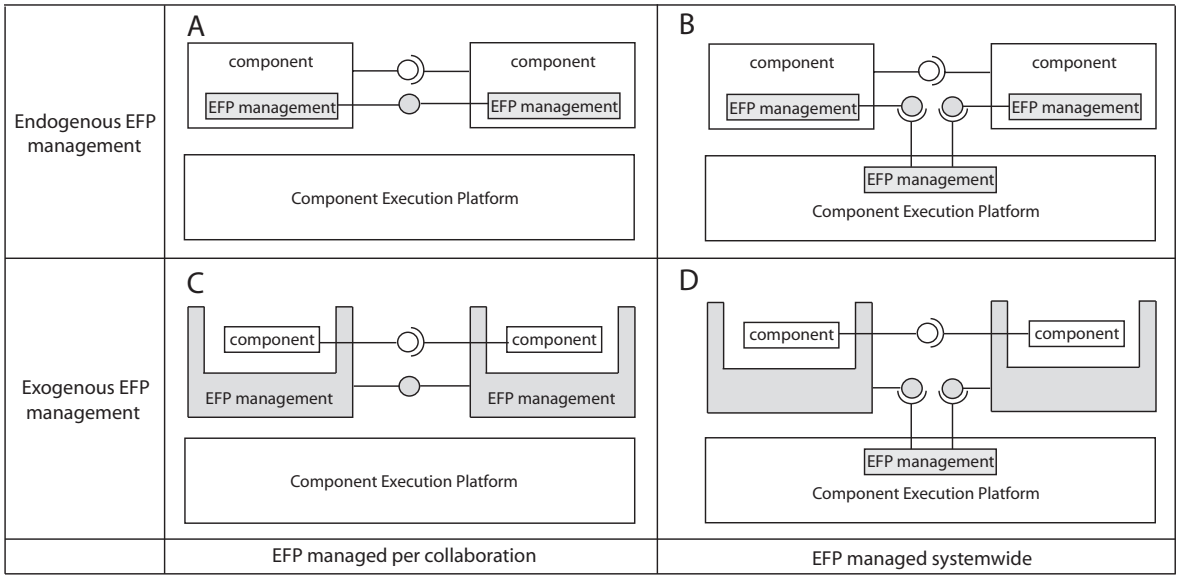


FIGURE 2.12 – **Gestion des propriétés non-fonctionnelles d'un composant** – approches Endogène ou Exogène, et autonome ou dépendantes du système (source : [Crnkovic et al., 2009])

Par exemple, et comme le montre la figure 2.12, la gestion des propriétés non-fonctionnelles d'un composant peut respecter une certaine loi de conception. Soit en étant dépendant du cycle de vie du composant (partie endogène située en haut de la figure), et détruite une fois le composant détruit. Soit en étant indépendant du cycle de vie du composant (partie exogène située au bas de la figure). De plus une certaine autonomie peut être proposée (partie gauche de la figure) ou au contraire une dépendance avec la plateforme peut être sollicitée (à droite de la figure).

Finalement, la propriété de composition d'un modèle à composant peut être réalisée avec le principe du composite. Pour cette étude, un composant est un composite dont la partie métier est réalisée par d'autres composants.

Concernant la configuration, les architectures à composant peuvent être décrites soit dans des Langages de Description d'Architecture (ADL) [Medvidovic and Taylor, 2000], soit graphiquement, avec idéalement la méthode DI (sous-section 2.3.2) pour faciliter la liaison entre le code métier à embarquer dans un composant, et le modèle à composant.

Voyons à présent six modèles à composants étudiés ici : soit CCM (sous-section 2.6.1), OpenCom (sous-section 2.6.3), OSGi (sous-section 2.6.2), Fractal (sous-section 2.6.4), SCA (sous-section 2.6.5) et FraSCAti (sous-section 2.6.6).

2.6.1 Modèle à Composant CORBA – CCM et LwCCM

CCM [OMG, 2001, Jung and Hatcliff, 2007, OMG, 2004, Dearle, Alan, 2007] est une réponse à l'approche CORBA (sous-section 2.5.2) dans le monde des composants. Le but fut de rajouter les caractéristiques du SOA, au monde CORBA, à l'aide de l'approche composant.

Toujours dans la même optique que CORBA, l'idée est dans un premier temps d'utiliser l'IDL pour générer une couche de code composant en plus du code CORBA. Les opérations sont ainsi décrites en tant que facette de composant, et les relations avec d'autres composants sont résolues à l'aide de

26. `http://www.autosar.org/`

réceptacles. Puis, la programmation par évènement est rendue possible à l'aide des sources et puits d'évènements qui sont respectivement producteurs et consommateurs d'évènements. Il est également possible de paramétrer l'initialisation du code métier à l'aide des attributs du composant. Finalement, des opérations non-fonctionnelles sont possible à l'aide de l'interface promue par le composant.

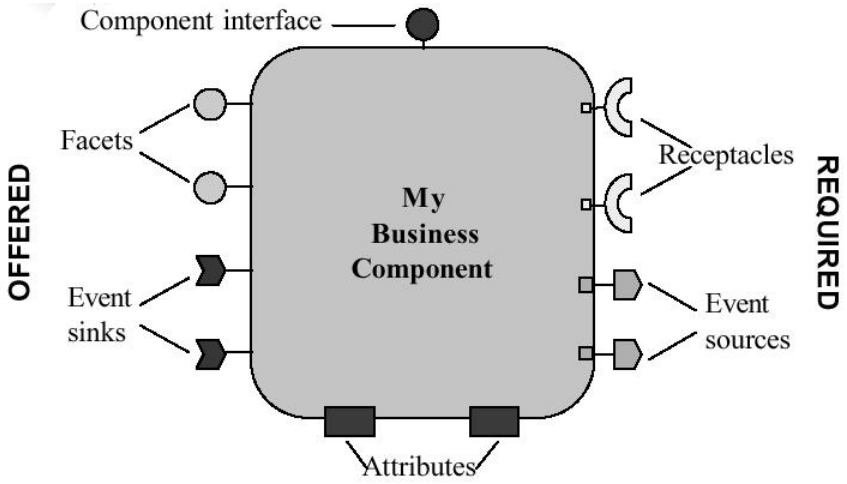


FIGURE 2.13 – **Modèle abstrait du composant CCM** – source : `http://igm.univ-mlv.fr/institut/`

La figure 2.13 représente une vue abstraite d'un composant CCM décrit ci-dessus.

Malheureusement, CCM dépend toujours d'un ORB pour assurer la liaison entre composants. Ne laissant pas le choix du protocole à utiliser, il y a un risque que cette sur-couche composant ne soit pas propice à l'environnement d'exécution cible.

Pour garantir à nouveau CCM dans le domaine de l'embarqué, le modèle CCM pour l'embarqué (LwCCM) est né depuis la spécification de CCM, en retirant simplement le surplus inutile pour les systèmes RT-E.

La spécialisation d'une couche de communication fut possible grâce à la notion de connecteur, introduite avec le besoin de faire converser le monde CORBA avec le monde du DDS [OMG, 2009b].

Ainsi, CCM est, sur le papier, un modèle à composant dédié aux systèmes RT-E avec une sous-partie appelée LwCCM, indépendant de la plateforme d'exécution, du code métier, et du protocole de communication avec d'autres intergiciels.

Autrement, la caractéristique de composition des SOAs n'est pas supportée par CCM car la relation de composition entre composant n'existe pas, et de plus, la communication entre composants est imposée par un ORB, spécifique au moteur d'exécution CCM.

Finalement, et par expérience personnelle pour avoir développé des connecteurs WS pour LwCCM dans le projet ITEmIS [Labéjof, 2010], l'approche composant n'utilise pas de méthodes propres à la séparation des préoccupations (sous-section 2.3.3). Ce qui implique que l'intégrateur doit travailler sur le développement de classes intermédiaires entre la couche composant et les interfaces métier qui sont heureusement générées. Ainsi, il y a beaucoup d'informations redondantes à redéfinir au niveau métier, au niveau composant (IDL, interfaces générées et fichiers de configuration), plutôt que de les définir une seule fois depuis le code métier, ou les fichiers IDL à l'aide de décorations [Gamma, 1995] par exemple. Finalement, les connecteurs obligent les composants métier à implanter toutes les facettes et les réceptacles utilisés par le connecteur, et pas seulement ceux qui sont utilisés par le métier. Sans quoi, la configuration est invalide.

CCM est une solution incomplète pour des perspectives de développement de systèmes avec une approche SOA, et manque cruellement de spécialisation de communication pour s'ouvrir aux autres intergiciels.

2.6.2 OSGi

OSGi est un modèle à composant ouvert et dynamique, basé sur Java, pour le développement, le déploiement et la gestion de services. À l'origine, il se voulait capable de répondre aux contraintes du domaine de l'embarqué (sous-section 2.2.1). Dans OSGi, les applications sont appelées des bundles. Ces bundles peuvent être dynamiquement déployés et mis à jour, à l'aide de librairies Java maintenues dans des fichiers de type JAR.

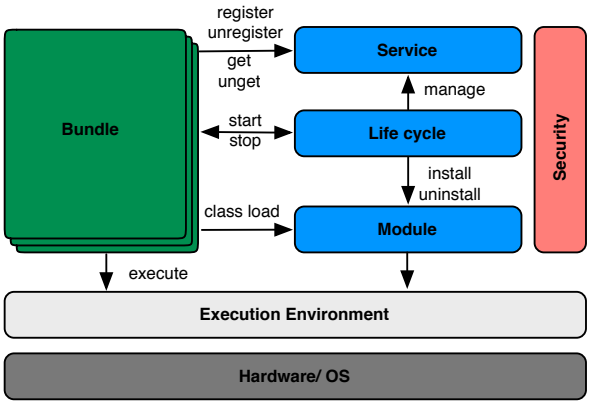


FIGURE 2.14 – **Couches OSGi** – Securité, Module, Cycle de vie et Service (source : [OSGi, 2009])

Les fonctionalités de OSGi sont organisées en différentes couches, représentées dans la figure 2.14, et définies dans la liste suivante :

1. Sécurité : étend la spécification d'architecture de sécurité Java 2, afin de limiter les fonctionalités offerts par les bundles avec des capacités pré-définies.
2. Module : établit les règles pour la déclaration de dépendance des bundles. Détermine comment les fonctionalités sont importées ou exportées.
3. Cycle de vie : gère le démarrage, l'arrêt, l'installation, la désinstallation et la mise à jour des bundles.
4. Service : définit un modèle de programmation dynamique qui simplifie le développement des bundles, avec un découplage entre interfaces Java, et l'implémentation qui peut être choisie à l'exécution.

Le modèle à composant OSGi est une excellente alternative à l'approche CCM, avec l'ajout d'une couche réflexive nécessaire dans les systèmes actuels. Cependant, sa forte réciprocité avec le langage Java le contraint à un isolement avec les autres langages de programmation, et limite ses possibilités à celles de Java.

2.6.3 OpenCOM

OpenCOM [Coulson et al., 2008] est un modèle à composant réflexif académique basé sur COM de Microsoft [27], et qui se veut indépendant des plateformes et langages d'exécution visés. Ce modèle à composant est implémenté en C, C++ et Java.

Son modèle de programmation réutilise l'IDL 3 qui est une version extensible de l'IDL (sous-section 2.5.2), comprenant la définition de type de donnée, d'opération, de composant, et d'interfaces de composants utilisées ou offertes.

Le modèle à composant comprend une composition hiérarchique à l'aide des composants de nature composite appelés *capsules*.

La spécificité de ce modèle à composant est qu'il utilise un noyau de conception permettant d'y rattacher un canevas de composant personnalisable en fonction des plateformes et exigences de systèmes visées (voir figure 2.15).

27. `https://www.microsoft.com/com/`

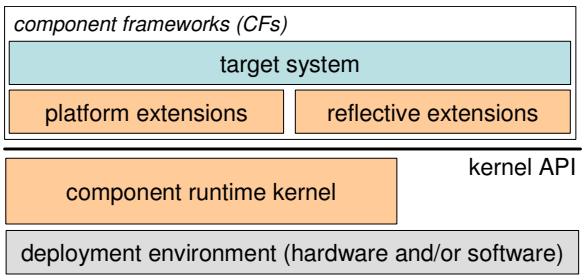


FIGURE 2.15 – **Architecture du modèle à composant OpenCOM** – Environnement de déploiement, noyau, et canevas de composants

Par exemple, le noyau de base permet de déployer des composants statiques. Étendu avec des propriétés dynamiques, le noyau pourra être embarqué dans le modèle à composant pour pouvoir le modifier à l'exécution (chargement des composants et modification dynamique des liaisons inter-composants).

Les extensions possibles sont de trois types, soit des *caplets*, des *chargeurs* et des *lieurs*. Les caplets permettent de gérer la composition des capsules. Les chargeurs garantissent le déploiement et la destruction des composants. Finalement, les lieurs rendent possible les liaisons inter-composants.

Même si OpenCOM est faiblement outillé, sa force reconnue par cette thèse est son utilisation dans un grand nombre de projets et de domaines d'exécution, qui démontrent son utilité. Par exemple [Duran-Limon et al., 2003] est un cas d'utilisation d'OpenCOM dans un système temps-réel et embarqué.

2.6.4 Fractal

Le modèle à composant Fractal [Bruneton et al., 2002, Bruneton et al., 2004, Seinturier et al., 2007] s'est proposé comme un travail de recherche pour développer et maintenir facilement un système logiciel, sans être dépendant d'un langage de programmation, ou d'un environnement d'exécution.

D'une manière générale, Fractal utilise toutes les méthodes de programmation que nous avons analysé dans ce manuscrit (section 2.3).

Comparé aux autres modèles à composant, sa spécificité est que tout composant héberge une membrane avec des contrôleurs, qui sont eux-mêmes des composants, et qui gèrent de manière personalisable le composant associé avec des propriétés non-fonctionnelles.

Ainsi, il existe une membrane par défaut, représenté dans la figure 2.16 contenant les contrôleurs suivants :

Binding gère les liaisons avec les autre composants,

Lifecycle gère le cycle de vie extensible du composant via un état défini par un champ texte ("STARTED" ou "STOPPED" par défaut),

Name gère le nom du composant,

Super gère la relation de composition du point de vue d'un composant fils,

Component gère l'implémentation du composant, déterminée par des composants fils (la nature du composant est d'être un composite dans ce cas) ou du code métier (la nature du composant est d'être un composant primitif/élémentaire).

La séparation de la membrane et de la partie métier du composant permet de rendre disponible les contrôleurs quelque soit la disponibilité du métier, permettant d'introspecter ou de reconfigurer le composant.

Ainsi, chaque composant a sa propre membrane, et la liberté de choix du code d'implantation du métier est aussi libre que celle concernant les traitements non-fonctionnels assurés par les contrôleurs.

La liaison entre composants est réalisée en connectant des ports et des interfaces de composant. Le modèle Fractal ne contraignant pas la réalisation des ports, il est possible de la personnaliser. Ainsi, on

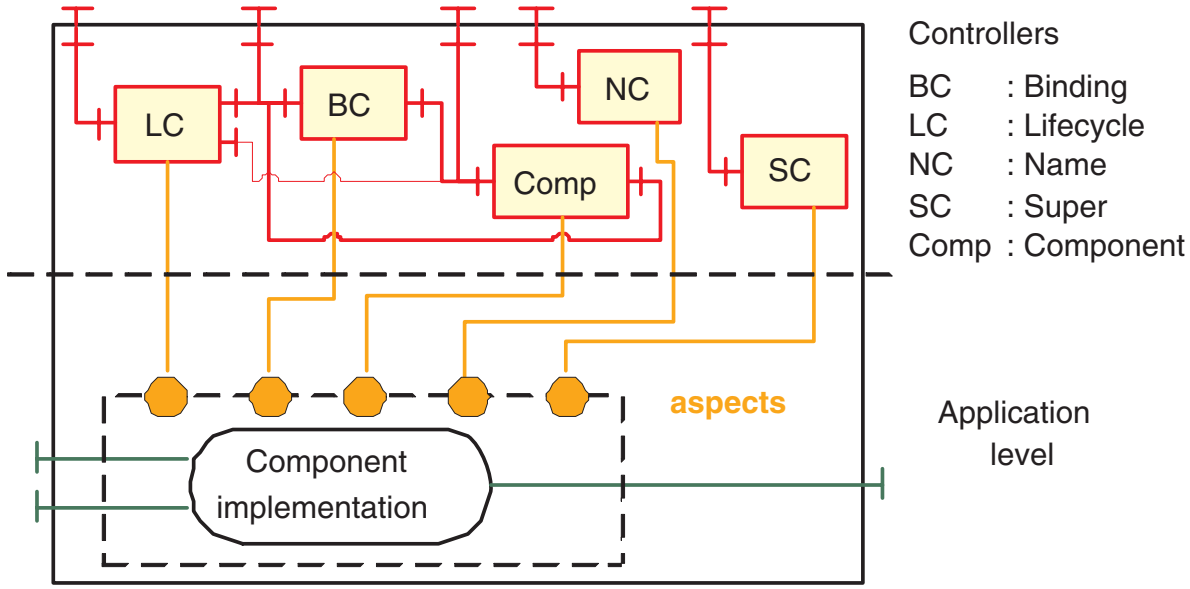


FIGURE 2.16 – **Structure par défaut d'un composant Fractal** – source : [Seinturier et al., 2007]

a un réel découplage entre le métier des composants Fractal, et le moyen d'exposer leur fonctionnement, et ce, même à l'exécution.

De plus, ce modèle propose le partage de composant. Ce dernier permet d'appliquer une relation de composition multi-parentée, où un composant peut dépendre fortement de plusieurs parents.

En complément de l'architecture, le module Fraclet introduit les méthodes de programmation de SoC (sous-section 2.3.3), IoC et DI (sous-section 2.3.2) pour ajouter des fonctions d'adaptation nécessaire dans les systèmes à développer. Fraclet permet de faciliter le développement et la maintenance du code, en liant le code métier au modèle à composant, à l'aide de décorations (patrons de conception [Gamma, 1995]). Le code métier peut ainsi définir les services à exposer, ou les références et les propriétés à utiliser. Il peut même intercepter les appels faits sur ces contrôleurs (par exemple, l'initialisation, le démarrage, l'arrêt ou la destruction du composant peuvent être capturés par le code métier) ou sur lui-même [Pessemier, 2007]. Dans le sens inverse, les fichiers de configuration d'extension "fractal" sont proposés au format XML, pour assurer une portabilité basée sur ce langage.

D'autres modules viennent aider à supporter les adaptations à l'exécution :

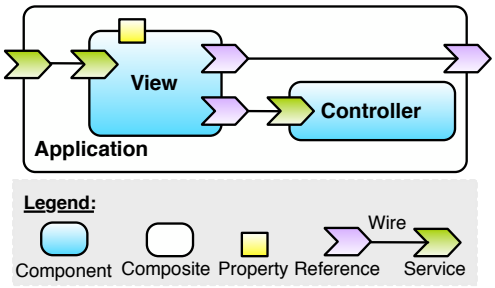
- Fractal explorer [David et al., 2006] : permet d'introspecter et de reconfigurer un modèle à composant durant son exécution, via une interface utilisateur graphique (GUI),
- FScript [David et al., 2008] : langage pour dérouler des opérations de re-configuration.

Fractal a eu beaucoup de difficulté pour se populariser à une époque où le modèle à composant CCM (sous-section 2.6.1) était utilisé dans la plupart des systèmes RT-E (sous-section 2.2.1), standardisé par l'OMG et outillé. Toutefois, certaines bonnes idées ont été reprise alors que CORBA, et son moyen de communication ORB imposé, commençait à s'essoufler. C'est ainsi que l'on retrouve les points forts de l'architecture de Fractal dans le modèle à composant SCA qui souhaite se débarasser définitivement des lacunes imposées par l'approche CORBA.

Comparé à OSGi, Fractal est indépendant d'un langage de programmation et propose un niveau de personnalisation des aspects non-fonctionnels bien au delà de ceux d'OSGi.

2.6.5 Architecture à Composants de Services – SCA

SCA [OASIS, 2010b, Jim Marino, 2009] est un ensemble de spécifications pour la conception de systèmes distribués basé sur l'approche SOA et sur les principes de l'ingénierie logicielle orientée composant (CBSE) [Project, 2007]. Dans SCA, les bases de construction sont les composants logiciels, qui peuvent offrir des services, ou solliciter d'autres composants via des références, à l'aide de descriptions d'interfaces, et de propriétés. Ces services et références sont liées à l'aide de fils ("wire" en anglais).



Application.composite

```
<composite name="application.composite">
  <service name="run" promote="View/run"/>
  <component name="View">
    <implementation.java class="app.gui.SwingGui"/>
    <service name="run">
      <interface.java interface="java.lang.Runnable"/>
    </service>
    <reference name="control" target="Controller/control">
      <interface.java interface="app.ControllerService"/>
    </reference>
    <property name="orientation">landscape</property>
  </component>
  <component name="Controller">
    <implementation.java class="app.ctrl.Controller"/>
    <service name="control">
      <interface.java interface="app.ControllerService"/>
    </service>
  </component>
</composite>
```

FIGURE 2.17 – **Configuration graphique et textuelle d'un composite SCA** – Simple application vue/contrôleur (source : [Romero, 2011])

La figure 2.17 expose les deux moyens de configurer un composite SCA, soit de manière graphique, soit de manière textuelle à l'aide d'un fichier d'extension "composite" respectant une structure au format XML.

À l'image de Fractal, SCA est conçu pour être indépendant des langages de programmation, des protocoles de communication, des langages de description d'interface et des propriétés non-fonctionnelles. À la manière de Fractal, tout est personnalisable, seul les concepts de composant, interface, propriété et liaisons sont imposés pour séparer les préoccupations (SoC voir sous-section 2.3.3) des implantations des couches métier et non-fonctionnelles du système.

Toujours en comparaison avec Fractal, des spécifications SCA proposent de faciliter la liaison entre le modèle à composant et le code métier avec la méthode d'injection de dépendance (DI voir sous-section 2.3.2) et des points de coupe AOP (voir sous-section 2.3.4). Mais l'architecture ne spécifie pas solutions pour le contrôle des composants à l'exécution.

Aujourd'hui, le consortium OASIS standardise SCA de manière à la démocratiser chez la plupart des architectes d'entreprise ou académique, et peut-être même réussir à remplacer les solutions de type CORBA.

Tout est entre les mains des moteurs d'exécution SCA.

2.6.6 FraSCAti

FraSCAti [Seinturier et al., 2009, Seinturier et al., 2012] est le seul moteur d'exécution SCA (sous-section 2.6.5) utilisant Fractal (sous-section 2.6.4).

Les autres moteurs d'exécution ne supportent pas autant de réflexivité et de fonctionalités à l'exécution que Fractal. C'est la raison pour laquelle FraSCAti est le modèle à composant utilisé dans les implémentations de cette thèse.

FraSCAti respecte les concepts des spécifications SCA, et supporte les langages de programmation Java, C, BPEL et Scala [Rouvoy and Merle, 2012]. Plus toutes les fonctionalités d'adaptation que propose déjà l'approche Fractal (tels que le tissage dynamique d'aspects par exemple [Seinturier et al., 2007]).

Les composants FraSCAti sont donc une spécialisation des composants Fractal pour l'architecture SCA. De ce fait, ils ont leur propre membrane qui diffère de celle de Fractal.

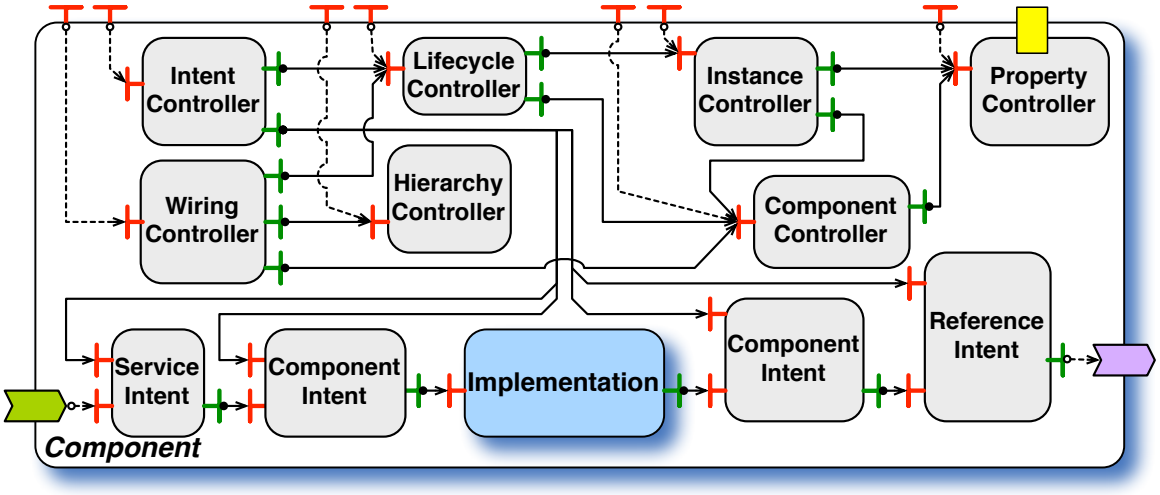


FIGURE 2.18 – **Personnalité d'un composant FraSCAti** – Spécialisation d'un composant Fractal pour un composant SCA (source : [Seinturier et al., 2012])

La figure 2.18 montre la structure Fractal d'un composant FraSCAti/SCA.
Les contrôleurs supplémentaires sont :

Intent gère les interceptions des appels aux contrôleurs, aux services (**IntentController**), aux références (**ReferenceIntent**) ou à l'implantation (**ComponentIntent**),

Instance gère l'instanciation de l'implémentation, utile pour redémarrer le métier du composant,

Property gère les propriétés SCA du composant, c'est à dire la (re-)configuration du métier du composant,

Hierarchy gère la localisation physique du composant dans le modèle à composant,

Wiring spécialisation du **BindingController** Fractal.

La particularité de FraSCAti est qu'il fournit la preuve de son auto-adaptation en offrant un moteur d'exécution écrit en FraSCAti/SCA. Cette approche permet un nombre incroyable de possibilités quand à la re-configuration du moteur. Par exemple, il devient possible d'ajouter durant son exécution un nouveau type de communication.

La figure 2.19 montre l'architecture du moteur d'exécution, et les points de variabilité. Ainsi, il devient possible de spécialiser tous les concepts SCA, et ce, même durant l'exécution.

2.6.7 Bilan

Les approches composants étudiées dans cette section ont permis d'identifier un grand nombre de travaux et d'outils, avec des fonctionalités diverses et variées en fonction des domaines visés.

Le tableau 2.3 récapitule les caractéristiques utiles pour s'approcher de la conception et de la maintenance d'un système hétérogène et auto-adaptatif. Y sont comparés la modélisation (Model.), la configuration (Conf.), le déploiement dynamique (Dyn. depl.), la re-configuration (Re-conf.), l'orientation systèmes RT, embarqué (E.) ou IT. Finalement, le support des propriétés de réflexivité (Reflec.) et de partage de composant (Shar.).

À l'image du développement logiciel, les modèles à composant ont été conçus dans un premier temps pour répondre à un besoin particulier. Puis ils ont été enrichis par de plus en plus de caractéristiques avec

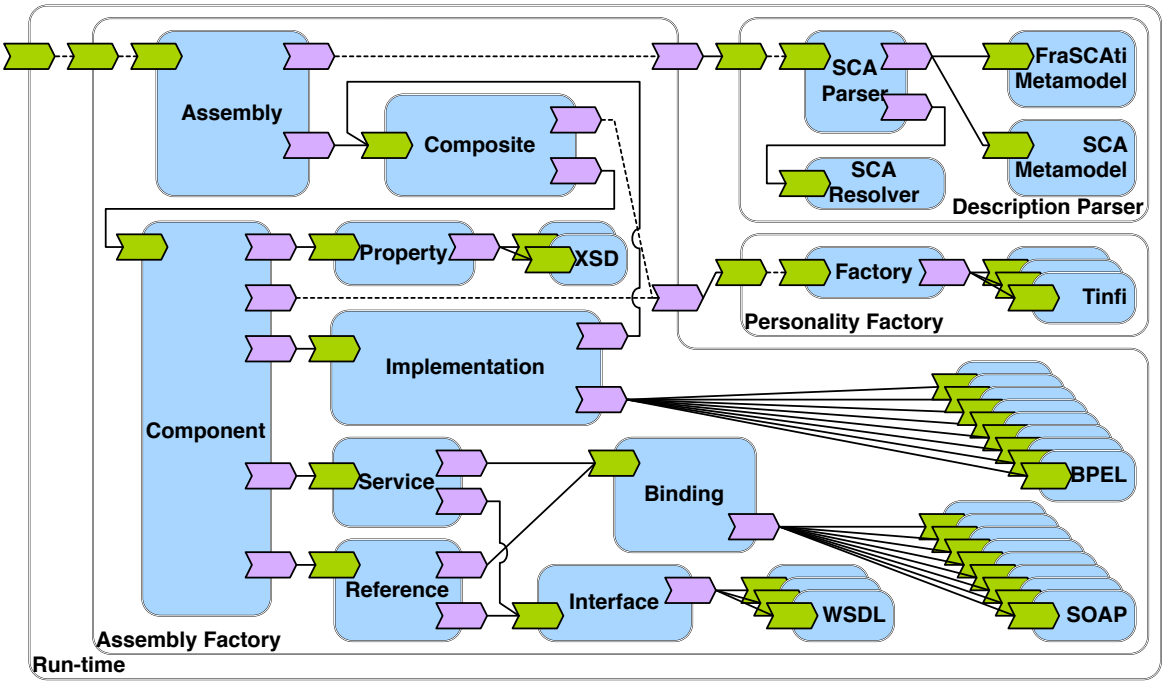


FIGURE 2.19 – **Architecture du moteur d'exécution FraSCAti** – Variabilité des concepts SCA (source : [Seinturier et al., 2012])

TABLE 2.3 – Tableau caractéristique des modèles à composant étudiés

Solution	Life cycle				Domain			Property	
	Model.	Conf.	Dyn. depl.	Re-conf.	RT	E	IT	Reflec.	Shar.
(Lw)CCM	-	++	+	+	+++	+++	++	-	-
OSGi	-	+	+/-	++	-	+++	++	X	-
OpenCOM	-	+	+	++	+++	+++	+++	X	-
Fractal	ADL	++	+++	+++	-	+	+++	X	X
SCA	ADL	+++	-	-	-	-	+++	-	-
FraSCAti	ADL	+++	+++	+++	-	+	+++	X	X

d'autres besoins arrivant. C'est la raison pour laquelle aucun d'eux ne répond à toutes les caractéristiques, mais certains sont plus ou moins complets.

Dans cette thèse qui s'intéresse avant tout à la réflexivité, il faut comprendre que les trois modèles à composant majeurs sont OSGi, OpenCOM, Fractal (FraSCAti étant considéré comme une implémentation de Fractal). OpenCOM et Fractal sont agnostiques du langage, contrairement à OSGi qui est orienté Java, et qui restreint fortement son utilisation dans un système hétérogène. OpenCOM est agnostique des langages de programmation et des plateformes d'exécution, et il a fait ses preuves dans un grand nombre de systèmes RT-E, mais ne propose pas de moyen de configuration de modèle, ni de relation de partage de composant, et impose un noyau d'exécution avec trois types de traitement non-fonctionnel (extensions). Fractal est également agnostique des langages de programmation et des plateformes d'exécution, gère la relation de partage de composant, propose un ADL pour la configuration, et accepte tout type de traitement non-fonctionnel (contrôleurs), mais pêche dans la preuve de son exécution dans des systèmes RT-E.

Fractal est plus riche en terme de flexibilité et de facilité de conception qu'OpenCOM, mais il n'a pas fait ses preuves dans le domaine RT-E. De par ses capacités, il sera choisi par rapport à OpenCOM et OSGi.

Le modèle à composant FraSCAti (sous-section 2.6.6) se retrouve comme la somme des solutions de réflexivité de Fractal (sous-section 2.6.4) et de liberté de configuration SCA (sous-section 2.6.5), plus d'autres qui répondent à des problématiques de systèmes dynamiques et hétérogènes ([Romero, 2011]). Le seul point négatif apparaissant au tableau vis à vis de cette étude est, tout comme Fractal, la non preuve, à notre connaissance, que FraSCAti peut fonctionner dans un environnement fortement contraint, même si sa capacité modulaire permet en théorie d'utiliser un minimum de composants.

Le modèle à composant FraSCAti apparait comme étant le plus intéressant pour l'approche R-*.

2.7 Satisfaction de spécification et de propriétés non-fonctionnelles

Cette section s'intéresse aux travaux et outils considérant la définition et la gestion de la spécification et des propriétés non-fonctionnelles dans un système distribué qui sont comprises comme des Qualités de Services (QoS) dans une approche SOA.

Même si les premières discussions ne vont pas concerner directement les QoS, il est important de comprendre que des idées vont être mises en avant pour pouvoir les réutiliser dans le domaine des SOAs.

Les qualités de service sont communément définies au niveau opérationnel du système, et doivent être respectées au cours de l'exécution du système [Beugnard et al., 1999]. Pour ce faire, différents travaux ont été proposés (identifiés surtout par Gabriel Tamura [Tamura, 2012]).

Des spécifications [Frølund and Koistinen, 1998b, Frølund and Koistinen, 1998a, Beisiegel et al., 2007b] qui proviennent de différents organismes qui ne garantissent pas l'interopérabilité entre spécifications.

Mais aussi des langages [Aagedal, 2001, Röttger and Zschaler, 2003, Becker, 2008, Baligand et al., 2008], des modèles [Collet et al., 2005, Chang and Collet, 2007, Lee et al., 2009, Comuzzi and Pernici, 2009], des modèles de sémantiques formelles [Braga et al., 2009, Cansado et al., 2010] et des techniques de gestion de QoS à l'exécution [Ortiz and Bordbar, 2008].

Les techniques et les solutions étudiées sont proposées dans l'ordre croissant de niveau d'architecture d'un système (sous-section 2.2.1). Soit en sous-section 2.7.1, l'OCL qui s'intéresse au code métier. En sous-section 2.7.4, les SLAs et les WSLAs qui s'intéressent aux contrats de QoS dans le domaine des services. En sous-section 2.7.5, MARTE propose un support des qualités de service dans les différents niveaux d'architecture d'un système. Finalement, la dernière étude en sous-section 2.7.6 concerne l'ontologie de QoS OWL-Q pour les WS.

2.7.1 Langage de Contraintes d'Objets – OCL

Le langage OCL enrichit le langage UML (sous-section 2.4.3), en permettant de définir des assertions structurelles qui sont vérifiées lors de changement d'état du métier, ou lors de pré/post traitements d'opérations, tout en profitant de techniques d'analyses formelles.

Cette thèse considère l'approche OCL comme proche du paradigme AOP (sous-section 2.3.4) dans le sens où elle s'applique autour de changements d'état d'un code impératif de manière non-intrusive en découplant parfaitement la définition et la portée des contraintes des éléments métier écrits (paramètres, attributs, méthodes, classe, etc...).

Dans des systèmes qui nécessitent une conformité du code (sûreté d'algorithme de chiffrement par exemple, ou exécution de systèmes critiques), une approche comme OCL devient nécessaire.

Malheureusement, la spécification ne suffit pas à assurer la compatibilité entre outils existants [Gogolla et al., 2008], et donc devient impossible à appliquer dans des environnements hétérogènes.

2.7.2 Méthodes formelles

Le support des propriétés non-fonctionnelles à l'aide des méthodes formelles ([Braga et al., 2009] et [Cansado et al., 2010]) permet de s'abstraire des langages de programmation, tout en s'appuyant sur des fondements mathématiques. Ainsi, il devient possible de réaliser des preuves de cohérence de modèles nécessaires dans des systèmes à caractère critiques, où une erreur peut entrainer la mort de personnes.

2.7.3 Langage de Modélisation de qualité et ses variations dans le paradigme composant – QML, CQML et CQML+

La plupart des définitions de QoS considèrent les caractéristiques identifiées par Frølund et Koistinen [Frølund and Koistinen, 1998b] comme une base de réponse solide.

Ce langage spécifie les propriétés et les conditions entre QoS, et permet de définir des types de contrats qui sont similaires à des types structurés.

CQML [Tambe et al., 2009] et CQML+ [Röttger and Zschaler, 2003] sont des variations de QML qui comprennent des spécifications de contrat avec des caractéristiques supplémentaires à utiliser durant la phase de conception, mais aussi quelques types d'éléments ajoutés parmi ceux qu'il est possible de modéliser, comme les composants par exemple. L'une de ces caractéristiques est la relation de prestation des QoS [Becker, 2008] (satisfaction des contrats de QoS pour la dépendance avec les resources matérielles).

Néanmoins, ces langages manquent cruellement d'extensibilité. Premièrement, ils n'ont pas de moyen de spécifier différent niveaux de QoS pour être monitorés ou satisfaits pour une propriété de QoS donnée. Ensuite, même si quelques une de ces approches s'intéressent à une représentation utile pour l'exécution, leur sémantique n'est pas formellement spécifiée (sous-section 2.7.2). Un autre défaut est qu'ils offrent des possibilités d'évaluation de critère qualitatifs et quantitatifs qui ne sont pas suffisamment puissants, et du coup, difficiles à satisfaire (par exemple, il est possible d'indiquer un objectif croissant de quantité comme une température que l'on souhaiterait maximiser, mais il n'est pas possible de spécifier qu'un objectif est attendu dans un intervalle de valeur, comme un potentiel d'hydrogène, ou "pH", qu'un système souhaiterait avoir neutre). Finalement, ils laissent le problème d'une satisfaction dynamique des contrats de QoS pour les systèmes auto-adaptatifs.

2.7.4 Contrat de niveau de Services et extension aux services – SLA et WSLA

Le Contrat de Niveau de Service, ou SLA [Tripathy and Patra, 2011] est une négotiation entre un fournisseur de service et un client, applicable sur différents niveaux (client, service, entreprise, multi-niveau). Ces négotations définissent clairement les concepts de services, priorités, responsabilités, guaranties, et alertes. S'ajoutent à cela les caractéristiques propres aux services telles que la disponibilité, la serviabilité, la performance, l'exécution, etc...

Avec l'avènement des WS (sous-section 2.5.3), IBM a proposé de porter les SLAs dans le monde des WS avec les Contrats de Niveau de Services Webs, ou WSLAs, sous la forme d'une spécification [Keller and Ludwig, 2003b].

Ce portage a introduit l'uniformisation des SLAs dans un format électronique en proposant des structures syntaxiques des conditions exprimant les SLAs. Il devient alors possible de transformer, d'évaluer et de produire des évènements de contexte qui pourront déclencher des actions dans le cas de violation de SLAs [Ludwig et al., 2003].

Malheureusement, les sémantiques de ces actions ne sont pas formellement vérifiées (sous-section 2.7.2), ce qui pose des soucis de confiance dans leur exécution, et donc dans la satisfaction des contrats.

2.7.5 Profile UML MARTE

MARTE [MAR, 2009] est une spécification d'un profile UML pour la modélisation et l'analyse des systèmes RT-E. L'objectif est de supporter la spécification, la conception et la vérification/validation de ces systèmes, à travers des analyses de performance ou de planification.

MARTE couvre la modélisation des aspects logiciels et matériels des systèmes RT-E, dans le but de faciliter la communication entre architectes, intégrateurs et développeurs. Il souhaite également l'interopérabilité entre les outils de développement utilisés pour la spécification, la conception, la vérification, la génération de code, et tout ce qui est rattaché aux approches MDE (sous-section 2.3.5).

MARTE est ainsi découpé en quatre piliers fondamentaux. Soit la modélisation de QoS, la modélisation d'architecture, la modélisation basée sur les plateformes et l'analyse des modèles basé sur les QoS.

Comme toutes les approches UML, MARTE se veut structurel et graphique. Il n'y a pas de volonté à s'ouvrir à d'autres environnements non-issus des méta-modèles UML.

2.7.6 Ontologie OWL-Q

[Kritikos and Plexousakis, 2006] définit une ontologie pour les QoS dans le contexte d'utilisation des WS (sous-section 2.5.3). Cette ontologie est supposée complète et extensible.

Dans le cadre des SoS (sous-section 2.2.1), il convient de dynamiquement pouvoir étendre cette ontologie aux systèmes RT-E, et de ne pas la restreindre au paradigme WS.

2.7.7 Techniques de gestion dynamiques de QoS

[Ortiz and Bordbar, 2008] utilise astucieusement les approches MDEs (sous-section 2.3.5) pour assurer la réalisation de qualités de services dans un système distribué utilisant différentes technologies WS (sous-section 2.5.3). La méthodologie est en trois étapes. Premièrement, utiliser un profile UML en tant que PIM pour modéliser les qualités de services compatibles avec WSLA. Puis il faut introduire un ensemble de règles de transformation PIM vers PSM. Et finalement, décrire la génération de code dans les PSMs. Le paradigme AOP (sous-section 2.3.4) est utilisé pour tisser les règles de transformation et de génération de code sur les PSMs, après définition des points de coupe dans le PIM.

Cette approche répond en partie aux objectifs de R-*, puisqu'elle est capable d'automatiser la configuration et le déploiement de QoS pour assurer une cohérence entre les contrats de QoS et leur réalisation. Par contre, l'accent est mis sur la modélisation d'un système basé sur les WS, ce qui réduit considérablement les exigences d'un SoS qui nécessite d'autres technologies de communication, et l'aspect re-configuration à chaud n'est pas visé.

[Baligand et al., 2008] propose un moyen de définir des QoS dans des processus BPEL, et une automatisation des tâches. Un seul langage est proposé, non extensible mais proposant un grand nombre d'évènements spécifiques aux préoccupations BPEL et de possibilités de réactions (dont certaines sont l'exécution d'activités BPEL) dans le cas d'une violation de règle.

Le couplage fort entre les deux est intéressant. Mais la contrainte d'un seul et unique langage non-extensible pour modéliser les QoS, et le fait que l'environnement soit spécifique aux orchestrations BPEL, ne sont pas appropriés dans les problématiques d'environnement hétérogènes et dynamiques.

2.7.8 Bilan

Le domaine de recherche autour du support des propriétés non-fonctionnelles et du respect des spécifications dans les systèmes distribués est très riche. Le tableau 2.4 énumère les travaux discutés pour satisfaire soit la définition, soit la maintenance des valeurs avec effet d'exécution en réaction à une violation de contrat.

Par exemple, il devient important de pouvoir définir la spécification et les QoS de manière indépendante des systèmes en ayant besoin, puisque dans le cadre d'un SoS, il est impossible de prévoir à l'avance

TABLE 2.4 – Travaux autour du respect des spécifications et support des qualités de services dans les systèmes distribués

Work	Domain	Advantages	Weaknesses
OCL	Code specification	Not intrusive	Textual
QML/CQML/CQML+	QoS	Modelization	Not extensible
SLA/WSLA	QoS	Agreements	No formalization
SysML/ALF/MARTE	Specification/QoS	Community	Too complex
OWL-Q	QoS for WS	Semantic	WS specific
[Ortiz and Bordbar, 2008]	QoS	Unification & not intrusive	No formalization
[Baligand et al., 2008]	QoS for BPEL	Dynamic processing	Not extensible

comment et quels sont les systèmes qui devront fonctionner de concert pour réaliser un système global. Ainsi, la solution [Ortiz and Bordbar, 2008] (sous-section 2.7.7) offre un moyen de s'abstraire des systèmes et plateformes en utilisant un système tierce dédié à la définition des QoS, et qui profite des approches PIM/PSM et AOP pour déléguer leur exécution à des plateformes d'exécution.

Cette méthode découple, et maintient une tracabilité entre exécution des QoS et définition globale. Ainsi, on peut espérer pouvoir déléguer l'analyse des modèles PIM/PSM à des outils de formalisation (sous-section 2.7.2) afin de fournir une preuve des contrats attendus, ou de leur violation.

Deuxième partie

Contributions

Cette partie décrit R-*, sous la forme d'un ensemble de méthodologies, d'architectures et de solutions techniques pour permettre à un système distribué de s'adapter tout en respectant au mieux ses exigences.

La réponse de ce manuscrit se découpe en trois contributions majeures :

- une architecture de communication asynchrone, et adaptative à grain fin pour les systèmes RT-E (chapitre 3),
- une architecture de communication asynchrone, adaptative et interopérable (chapitre 4),
- un canevas de gestion d'environnement pour un système distribué (chapitre 5).

Ainsi qu'une implémentation d'un sous-modèle du Système de Systèmes TACTICOS, qui servira de cas d'utilisation et de démonstration des avantages et inconvénients de l'approche R-* (chapitre 6).

Chapitre 3

R-DDS - Un service de distribution de données reconfigurable à l'exécution pour les systèmes RT-E

Sommaire

Ce chapitre présente R-DDS, une extension adaptative de l'intergiciel DDS (sub-section 2.5.6, page 32) pour les systèmes RT-E.

3.1 Pourquoi un service de distribution de données reconfigurable pour les systèmes RT-E ?

Comme expliqué dans la partie de l'état de l'art (chapitre 2, page 11), le canevas R-* veut apporter des éléments de réponse pour les systèmes de systèmes qui doivent faire face à des environnements dynamiques. Parmi les paradigmes de communication asynchrones et re-configurables à l'exécution, de nombreuses solutions existent déjà, comme par exemple DREAM (sous-section 2.5.8, page 37), mais aucune n'adresse les systèmes RT-E, alors qu'ils sont eux aussi confrontés à des environnement dynamiques (voir la description du système TACTICOS de la section 6.1, page 109).

DDS (sub-section 2.5.6, page 32) est un intergiciel utilisé dans des systèmes dédiés aux domaines du temps réel et de l'embarqué. Depuis la publication de sa première spécification, cet intergiciel a connu des extensions comme DDSI [OMG, 2009c] pour permettre l'interopérabilité entre technologies DDS, et la mise en place de nouvelles qualités de services proposées par des implantations. Par exemple, l'implantation OpenSplice de la société Prismtech [28] qui, faute de couvrir la spécification depuis la première

28. www.prismtech.com

version de son logiciel, est aujourd'hui à la version 6 qui propose de nouvelles qualités de service et un support de la spécification DDSI pour assurer l'interopérabilité avec les autres technologies DDS.

Alors que la technologie DDS évolue en fonction des besoins des systèmes existants, il est très difficile de faire évoluer les modèles DDS utilisés dans ces mêmes systèmes car DDS n'est pas évolutif durant son exécution.

Or, ni la spécification, ni les implantations ne proposent de possibilités de mises à jour à chaud du modèle de distribution. C'est pourquoi nous nous sommes intéressés à une extension du modèle DDS pour couvrir la capacité du modèle de distribution DDS.

L'extension proposée se nomme R-DDS.

L'idée fait suite à une première contribution dans le projet ITEmIS qui visait à développer une liaison DDS pour la spécification du modèle d'évènement SCA [Beisiegel et al., 2009] (annexe A, page 137), dans la plateforme FraSCAti (sous-section 2.6.6, page 45). La conception de la liaison a mis en évidence des capacités de re-configuration propres à la plateforme FraSCAti qui invitèrent Philippe Merle à proposer la conception non pas d'une liaison DDS pour SCA, mais d'une version DDS orientée composant réflexif.

L'architecture visée s'appuie sur l'API de la spécification DDS, mais l'implémentation est résolue par un découpage fonctionnel, pour gérer le plus finement possible des tâches d'adaptation. Une conséquence d'un tel découpage est la possibilité de gérer les propriétés indépendemment des autres fonctionnalités, comme le choix du moteur d'exécution DDS.

3.2 Architecture

La contribution R-DDS est l'extension de la spécification DDS de ce document, où l'implémentation est réalisée par un découpage fonctionnel fin dans des composants réflexifs. Ce découpage satisfait des besoins d'adaptation de son propre modèle et du système l'utilisant comme modèle de distribution. La transformation du modèle DDS en composants assure une portabilité des systèmes DDS (interfaces des composants respectant la spécification), et un enrichissement du métier en ajoutant plus de contrôle et de sécurité sur les entités et les qualités de services DDS.

La description de l'architecture se fait en trois temps. Dans un premier temps, il s'agît d'identifier les fonctionnalités des éléments DDS (sous-section 3.2.1), puis de proposer un assemblage (sous-section 3.2.2), et finalement, de les projeter dans un modèle à composant réflexif (sous-section 3.2.3).

3.2.1 Découpage fonctionnel de DDS

Le découpage fonctionnel dépend des fonctions du modèle de distribution de DDS, et où une grande partie de la logique opérationnelle se retrouve au niveau du concept de l'entité DDS.

Les relations de l'*Entity* de la figure 2.7 exposent deux fonctionnalités. Une entité entretient des propriétés non-fonctionnelles (**G**) à l'aide d'une liste de *QoSPolicy*, et informe des changements de son état de manière asynchrone avec son *Listener*, ou de manière synchrone avec une condition de status (**E**). Lorsque l'on s'intéresse aux types *DomainParticipant* et *DomainEntity*, on retrouve une relation de composition entre entités. Cette relation permet le parcours du modèle de distribution autour d'un domaine DDS (**P**). Une entité respecte un cycle de vie, à l'aide des statuts de disponibilité (indisponible et disponible, **L**). Finalement, certaines propriétés fonctionnelles sont prises en charge par des entités, comme c'est le cas pour un *Topic* (**G**).

Table 3.1 – Acronymes des fonctionnalités d'une entité DDS

Acronyme	Fonctionnalité
G	Gestion des Propriétés
P	Parcours de modèle
E	Suivi d'évènements
L	Cycle de vie

Le tableau 3.1 montre les acronymes des fonctionnalités d'une entité DDS citées précédemment. Ces acronymes seront réutilisés dans la suite de ce chapitre.

3.2.2 Assemblage fonctionnel de DDS

L'assemblage fonctionnel nécessite de réaliser un schéma de dépendance entre fonctions. Or, ce dernier est très compliqué à concevoir et à mettre en œuvre, car il doit pouvoir évoluer dans notre approche puisque nous considérons un modèle adaptable à de nouvelles fonctions, comme de nouvelles qualités de services. Cette tâche est très rude, et dépasse de loin le travail proposée par ce manuscrit, même si des éléments de réponse doivent pouvoir se trouver dans le chapitre 5, page 83 relatif à la gestion des qualités de services dans un système distribué.

Dans un premier temps, l'assemblage sera résolu par la fonctionalité **P**. Toutes les intéractions inter-fonctionalités nécessiteront **P**. De ce fait, le degré de dynamicité d'adaptation recherché sera possible si et seulement si **P** est suffisamment générique pour ne pas avoir besoin d'évoluer dans le temps.

L'assemblage correspond ainsi à un arbre où **G**, **E** et **L** sont les fils de **P**. Les fonctions **G**, **E**, **L** et **P** sont encapsulées dans un composant, et seule **P** est promue à l'extérieur du composant. Un tel schéma permet de s'abstraire d'une relation de dépendance directe entre fonctionalités, mais nécessite une concentration sur **P**. Cependant, dans le contexte d'utilisation de DDS, cette concentration n'a que peu de répercussions sur l'utilisation primaire du modèle de distribution qui consiste à écrire et lire des données.

3.2.3 Projection de l'assemblage fonctionnel de DDS dans un modèle à composant réflexif

La sous-section précédente a mis en évidence un assemblage respectant une structure arborescente. Dans un modèle à composant, une telle architecture est possible à l'aide de deux types de relations entre composants, c'est à dire, la liaison d'interface, et la relation de composition. La liaison d'interface est une relation qui garantit une communication entre deux composants de même niveau de profondeur par composition dans le modèle. Alors que la relation de composition permet d'introduire une propriété de dépendance physique entre un composant parent, et les composants fils. Tous les composants représentant des fonctions d'une entité DDS sur un même plan logique, la liaison d'interface devient préférable aux relations de composition pour parfaire l'assemblage. Et tous se retrouveront dans un composant parent qui sera vu de l'extérieur comme l'entité DDS, à l'aide d'une interface promue par le composite, garantissant le respect de la spécification DDS, tout en masquant dans le composite l'aspect adaptatif de la solution via l'assemblage fonctionnel particulier.

Cette projection de l'objet entité dans le modèle à composant réflexif est similaire à l'approche Fractal [Bruneton et al., 2004] (sous-section 2.6.4, page 43), où les composants de fonctionalités d'entité sont des contrôleurs du composite, cachés dans une membrane Fractal. Cependant, il manque la partie métier. Cette partie sera représentée dans un composant qui implémentera l'interface d'une entité DDS offerte par ce même composant. Son implantation respecte le patron de conception "adaptateur" [Wolfgang, 1994] pour contenir un entité DDS provenant d'un moteur d'exécution, et adapter les appels provenant de l'interface offerte par le composite, vers les contrôleurs (changement de qualités de services vers le contrôleur couvrant la fonction **G**, etc...), et/ou solliciter l'entité embarquée pour finaliser les appels. Ainsi, les capacités d'adaptation à l'exécution sont réalisées de manière transparente pour l'utilisateur intéressé par l'interface de l'entité DDS.

La figure 3.1 illustre la projection d'une entité DDS dans le modèle à composant réflexifs, avec une représentation du modèle DDS à gauche, et le modèle R-DDS à droite. La figure montre un héritage du composite *Publisher* vers le composite *Element*, mais cet héritage est indirect, car entre les deux, il y a un composite *Entity*. Le composite *Element* permet d'introduire d'autres concepts DDS jugés suffisamment utiles pour ne pas les réduire à de simples propriétés comme c'est déjà le cas pour le domaine DDS qui est une propriété d'un *DomainParticipant*, ou des *Partition*s qui sont des qualités de services DDS. Il sera présenté plus en détails dans la sous-section 3.2.4.

Avant de rentrer dans le détail du modèle complet de R-DDS, voici une description des différents contrôleurs et du composant fonctionnel embarquant le moteur d'exécution. Leur implantation vise la

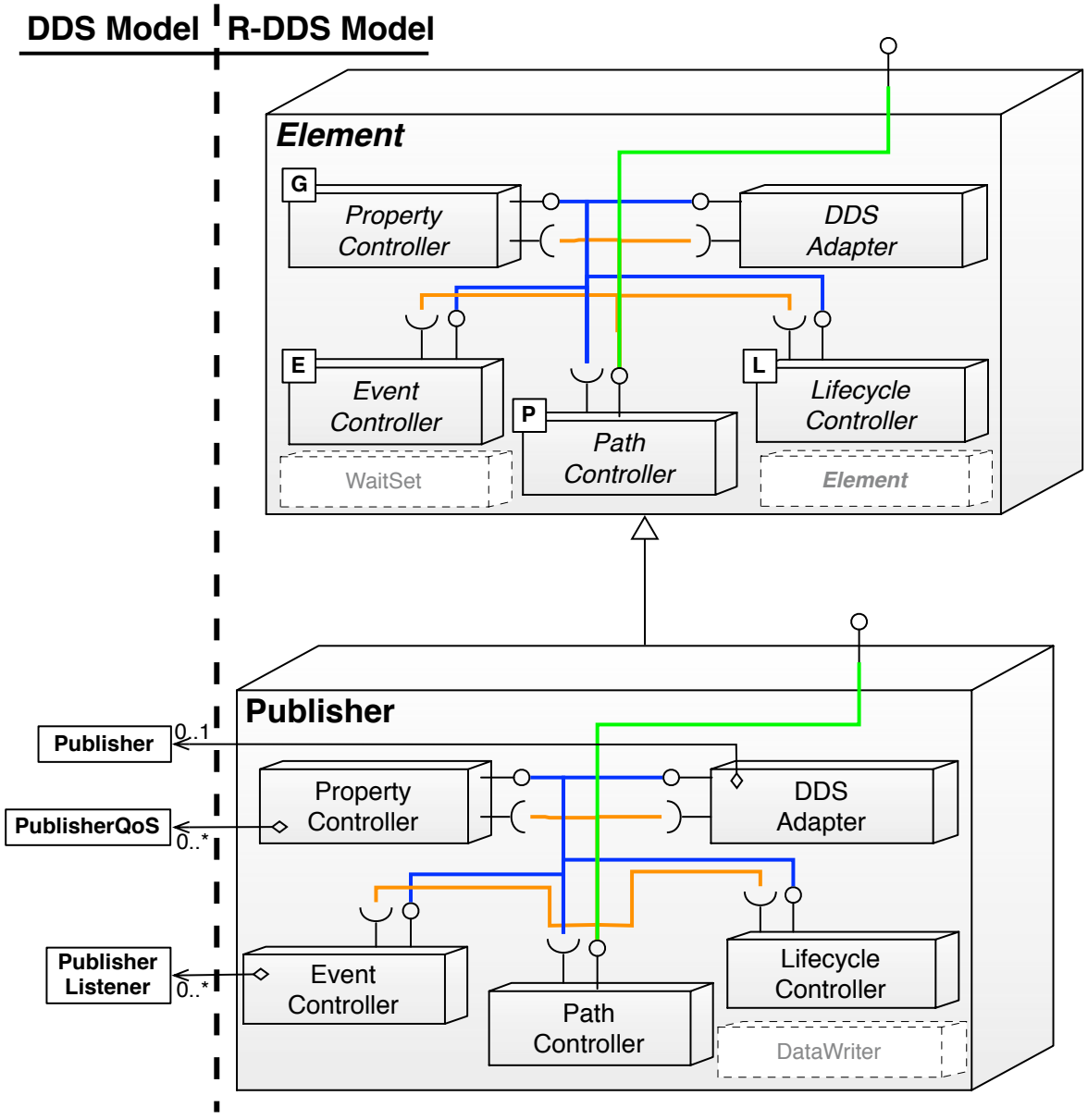


FIGURE 3.1 – **Structure et spécialisation d'un composite *ElementComposite*** – Avec vue des concepts DDS embarqués dans R-DDS

couverture des fonctions DDS de manière indépendante, mais également des fonctions d'adaptation ou d'accompagnement dans la configuration et l'exécution du système.

Contrôleur de parcours du modèle – PathController

Le contrôleur de parcours du modèle a à charge de résoudre le parcours d'un modèle DDS mais aussi sa modification dans un contexte d'adaptation.

Composant pivot du composite parent et des composants fils, son interface est suffisamment générique pour ne pas avoir à évoluer dans le temps. C'est pourquoi, c'est la seule interface qui est offerte au niveau du composite parent. Ses fonctions sont l'accès et la modification des composants du modèle (composants fils et parents du composite). Il a donc une visibilité sur tous les composants, via des relations de liaison d'interface avec les autres composants de contrôle, ou par relation de composition avec les composites fils et parents.

Dans les modèles à composants réflexifs les liaisons d'interface sont spécifiques à la communication entre composants. De ce fait, elles sont plus difficiles à modifier que les relations de compositions qui elles sont génériques et non-fonctionnelles. Par expérience, il est beaucoup plus rare de faire des opérations de parcours de modèle lorsque l'on utilise un système DDS que d'autres opérations comme l'arrêt ou l'envoi/la réception de messages. Les relations de connections sont privilégiées entre composants entre le *PathController* et les autres. Et là encore, le fait que le *PathController* est le seul à être connecté directement avec tous les autres composants est dû au fait qu'il est le seul qui doit gérer quels sont les composants à utiliser pour réaliser une adaptation ou parfaire une tâche fonctionnelle.

Composant fonctionnel – *DDSAdapter*

Le but du composant fonctionnel est d'enrichir de manière transparente la spécification DDS avec des capacités d'adaptation, c'est à dire une meilleure gestion des différentes fonctions d'une entité DDS dans un environnement dynamique.

Un adaptateur DDS respecte la logique du patron de conception du même nom [Wolfgang, 1994]. Il redirige des méthodes d'interface DDS vers les composants de contrôle avant de déléguer les appels vers un moteur d'exécution DDS.

Il offre l'interface DDS représentée par le concept R-DDS. Sa fonction principale est d'assurer le bon fonctionnement d'un objet DDS embarqué (représenté par la relation de composition entre un *Publisher* DDS et un composant *DDSAdapter*). C'est donc auprès de lui que l'utilisateur devra spécifier quel moteur d'exécution DDS utiliser, et invoquer les opérations DDS offertes par l'interface de la spécification.

Tout invocation d'opération demandée auprès de ce composant sera redirigée vers le composant de contrôle intéressé. Par exemple, s'il s'agît de connaître le *DomainParticipant* parent d'un composite *Publisher*. L'appel de méthode auprès du *DDSAdapter* demandera au *PathController* de retrouver le *DomainParticipant* parmi les composites parents. Le changement de *Listener* DDS sera délégué à l'*EventController*. Dans le cas d'une opération qui risque de modifier l'objet DDS embarqué, le *DDSAdapter* vérifie que l'appel s'est bien passé, puis invoque la dite méthode auprès de l'objet DDS. Toute erreur est transformée en un évènement auprès de l'*EventController*.

Contrôleur de propriété – PropertyController

Le contrôleur de propriété sert de composant de contrôle pour toutes les propriétés (fonctionnelles et non-fonctionnelles) d'une entité DDS.

C'est à dire toutes les propriétés de l'objet DDS embarqué (nom, qualités de service, etc...), ou d'autres plus spéficiques au système comme les URIs des librairies utiles pour le moteur d'exécution. Dans un contexte d'adaptation, il facilite la configuration et le traitement de propriété de manière dynamique, afin de répondre au problème bien connu de gestion d'une vingtaine de qualités de services DDS inter-dépendantes entre elles. Cette inter-dépendance ouvre la voie à un très grand nombre de combinaisons difficile à maitriser, et entraine des comportements de données non attendus. Comme preuve de cette difficulté à surmonter, la spécification DDS pour LwCCM [OMG, 2009b] a proposé deux combinaisons pour que la donnée se comporte comme un évènement ou comme une information d'état.

Dans ce contrôleur, les qualités de services ne sont pas regroupées par politique, comme c'est le cas dans la spécification DDS, mais modifiables une à une, limitant ainsi le maintien d'une combinaison complète. Une fois la gestion des propriétés d'un objet DDS déléguée à un composant tiers, il devient possible de persister leurs valeurs en cas de panne du moteur d'exécution, de modification majeur du composite (changement de composite parent), voire même de réplication du composite.

De plus, le *PropertyController* résoud dynamiquement le traitement de propriétés dynamiques ou statiques [29]. Si une propriété est statique, il devient facile de la modifier en demandant au contrôleur de cycle de vie de redémarrer le composite (voir l'exemple de modification de *TypeSupport* de la sous-section 3.2.5. Le traitement de la propriété est dynamique et présent comme valeur d'une entrée d'un dictionnaire où la clef correspond au nom de la propriété.

Contrôleur d'évènement – EventController

Le but du contrôleur d'évènement est de gérer l'émission d'information de changement d'état d'une entité. Le contexte d'adaptation de R-DDS enrichit ce contrôleur en permettant l'émission d'évènements propres au contexte applicatif, ou d'empêcher la diffusion d'évènements.

Les évènements émis par ce contrôleur sont génériques afin de pouvoir exister quelque soit les évolutions du modèle R-DDS effectuées. Et sont identifiables via un identifiant textuel unique par type d'évènement.

Chaque transmission d'évènement se fait de manière asynchrone. La transmission s'effectue vers les contrôleurs d'évènement des composites parents, ou vers un objet implémentant l'interface DDS *Listener* attendu par le composant fonctionnel du même composite.

La structure des évènements comprend le contexte d'émission avec des informations propres à la création de l'évènement (par exemple, l'évènement "changement de propriété" est accompagné du nom de la propriété, et des ancienne et nouvelle valeurs), mais aussi un dictionnaire qui pourra être utilisé tout au long de l'existence de l'émission de l'évènement.

Conjointement aux évènements émis par les entités DDS, l'*EventController* d'un composite héritant du composite *Entity* implémente aussi l'interface *Listener* attendue par l'entité DDS embarquée dans le *DDSAdapter*, et transforme les évènements issus du moteur en évènements R-DDS.

Contrôleur de cycle de vie – LifecycleController

Le *LifecycleController* gére le cycle de vie du composite. Par défaut, le composite connait deux états, arrêté et démarré. Un composite démarré correspond dans la logique DDS à une entité dans l'état disponible, ou *enabled*. Contrairement à DDS, R-DDS permet d'alterner entre les états de disponibilité d'une entité DDS, afin d'isoler une entité du modèle, le temps d'effectuer une tâche d'adaptation critique.

Toute action sur le cycle de vie est répétée de manière synchrone aux composites fils, afin de faciliter la reconstruction d'une arborescence de composants R-DDS, puisque par hypothèse, un composite est valide si et seulement si tous ses composites parents (par relation de composition) sont valides.

3.2.4 Extension du modèle DDS

DDS offre les concepts de *Domain* et de *Partition* en tant que zones physiques où les données peuvent circuler dans un réseau. Or, ces concepts sont réduits par leur nom, et oblige les implémentations à proposer une manière plus rîche de les gérer, et indépendante de la spécification. Les implémentations OpenSplice [30] et RTI [31] utilisent un fichier de configuration par *Domain* et par *Partition* qui s'éloigne de la spécification, et en conséquence, s'opposent à l'approche de la spécification qui vise à supporter la portabilité du paradigme DDS.

Cette sous-section s'intéresse à l'ajout de fonctionnalités dans ces concepts pour faciliter la configuration, à la maintenance de zones physiques de distribution de modèles et aussi à la portabilité des systèmes DDS quelque soit les moteurs d'exécution utilisés.

29. Contrairement à la propriété statique, une propriété dynamique est modifiable durant l'exécution du système.
30. `www.prismtech.com/opensplice`
31. `www.rti.com`

L'idée de base est de profiter du travail réalisé dans la sous-section précédente sur le composite *Entity*, pour ajouter le concept d'*Element*, composant de base de tout composite R-DDS, non-héritant du composite *Entity*. Ainsi, les composites de type *Domain*, *Partition* et *Entity* héritent du type de composant *Element*. La figure 3.1 illustre la structure de cet élément avec une relation d'héritage indirecte entre les composites de type *Element* et *Publisher*.

La spécification DDS détermine une relation de composition entre concepts clefs (par exemple, un participant à un domaine peut être composé d'entités de domaine), et certains concepts sont reliés par une relation de référence, alors que l'existence de l'un deux dépend fortement de l'autre. C'est le cas pour le lecteur de données dont l'existence dépend fortement d'une entité de souscription et d'un sujet. Notre modèle étendu propose de profiter de la relation de composition à parents multiples, ou propriété de composant partagé [Bruneton et al., 2002].

Cette relation de composition va nous permettre de propager les évènements du modèle de manière asynchrone vers les parents, mais également d'appliquer les opérations du cycle de vie vers les fils de manière synchrone.

La relation de participation/contribution entre éléments est propre au modèle R-DDS. Cette relation hérite de la relation de composition, mais ne propage pas l'action d'arrêt du composant parent. L'objectif est d'assurer une dépendance conceptuelle entre deux composants, sans que la condition de disponibilité ne soit validée (condition qui valide un composant fils si et seulement si tous les composants parents sont disponibles).

En dehors des relations, un travail d'extension est également proposé pour certains concepts clefs. DDS introduit les participants et les partitions comme éléments spécifiques à un domaine. Dans le modèle R-DDS, le concept de domaine est étendu en tant qu'élément de contrôle sur des partitions et des participants. Et toujours dans le respect de la spéficication DDS, un composite *Partition* pourra participer dans la réalisation des *Subscriber*s et des *Publisher*s puisque le concept de *Partition* est à l'origine une qualité de services de ces concepts.

Ensuite, DDS supporte les types de données de manière programmée avec une classe *TypeSupport*. Ce concept n'est pas représentatif à sa juste valeur dans un système distribué où la structure des données peuvent évoluer. Il est introduit dans le modèle R-DDS en tant que composite de type *Element*. Ce composite doit avoir un contrôle sur les composites de type *Topic*. Ce contrôle est résolu par relation de composition dans le modèle R-DDS.

Finalement, le composite racine de nom *System* offre une vue arborescente sur un modèle DDS. Le modèle devient ainsi plus facile à contrôler avec la remontée d'information d'état des éléments DDS. Il sera le parent de composites de type *TypeSupport* et *Domain*.

La figure 3.2 illustre le modèle DDS étendu, sous la forme de modules, avec les cinq nouveaux composites de nom *Element*, *TypeSupport*, *Domaine*, *Partition* et *System*, et les différents types de relation.

Le modèle est ainsi découpé en sept modules :

- le module système sert de racine à une arborescence de composants R-DDS,
- le module de flot de données définit les espaces physiques dans lesquels transitent les données,
- le module de participation définit les entités de participation à un domaine,
- les modules de publication et de souscription contiennent respectivement les éléments propres à l'envoi et à la réception de données,
- le module de condition justifie la prise en charge d'évènements du modèle,
- et le module de donnée définit les éléments à charge de définir les types de données pouvant transiter dans un système DDS.

À présent, voyons quelles sont les tâche d'adaptation possibles avec ce nouveau modèle.

3.2.5 Adaptation du modèle R-DDS

Le modèle à composant réflexif introduit la capacité de reconfiguration à l'exécution. Mais quelles sont les adaptations possibles dans le contexte du modèle R-DDS qui étend celui de DDS ?

Tout d'abord, le métier principal de DDS est la publication et la souscription de données. Le modèle R-DDS comprend des composants qui peuvent exister indépendement des autres, mais pour être disponibles, c'est à dire être capable de réaliser les fonctions DDS, ils ont besoin d'être liés à un modèle. Par exemple, un *DataWriter* sans *Topic* parent ne pourra pas écrire de données, mais acceptera qu'on y modifie des

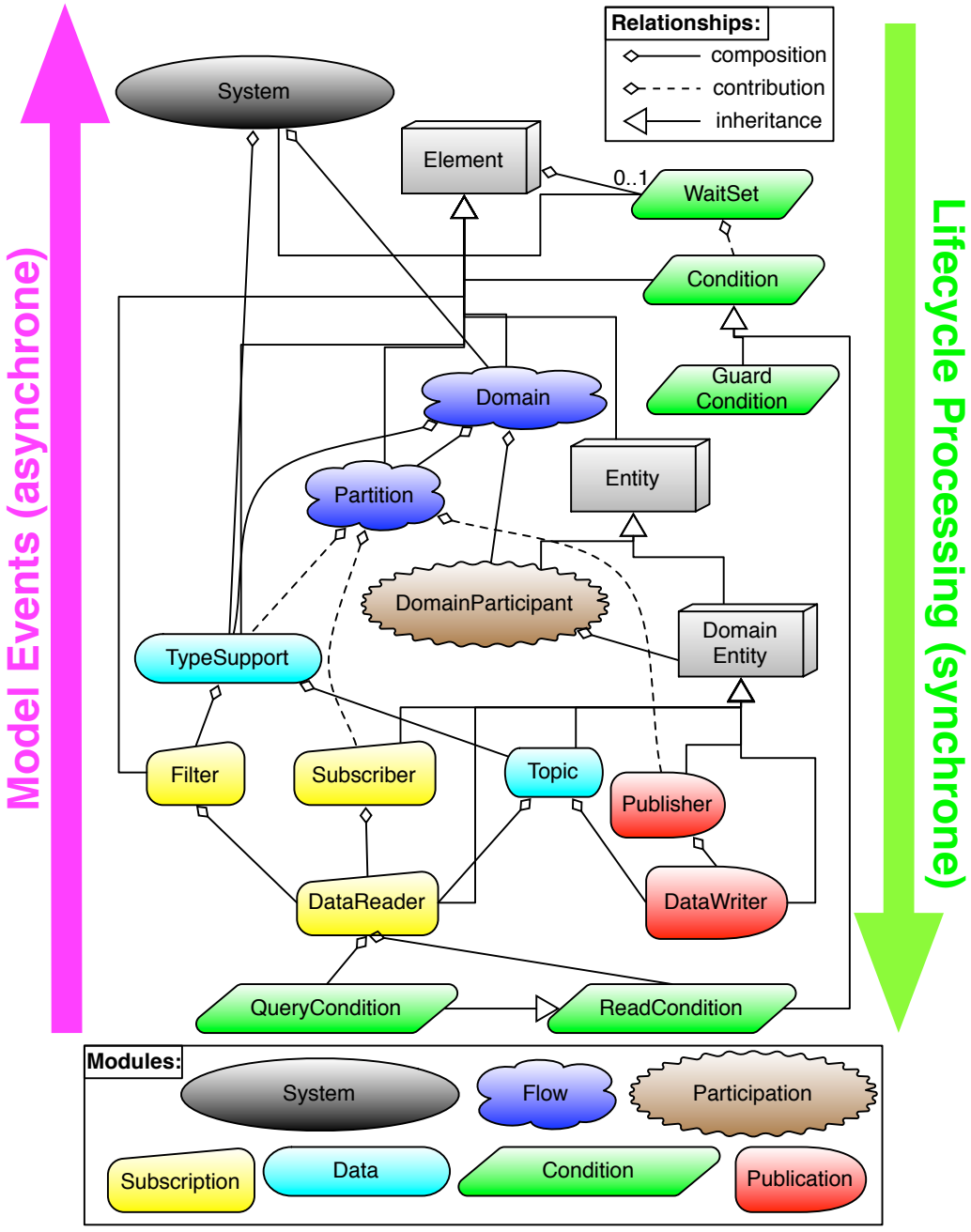


FIGURE 3.2 – **Diagramme des modules du modèle à composant R-DDS** – avec relations inter-composants, et sens de propagration d'évènements du modèle et de traitement de leur cycle de vie

propriétés comme des qualités de services, tout en restant indisponible. Ainsi, la relation de composition entre composants facilite grandement la visibilité de ce qui est exécuté, le contrôle et l'adaptation du modèle à tout instant.

Un autre exemple est la prise en charge de l'évolution de type de données. Le modèle DDS ne permet pas de copier une configuration et de la dupliquer, c'est à dire dans notre exemple, qu'une fois qu'un type de données doit être modifié, l'ancien objet *TypeSupport* associé devient inutile, ainsi que tous les objets l'utilisant, soit tous les *Topic*s, les *WaitSet*s et *Condition*s, les *DataReader*s, les *DataWriter*s, les *QoS* et les *Listener*s. Tout doit être recréé manuellement à l'aide de l'API DDS.

Le modèle R-DDS facilite grandement les évolutions de type de données, en proposant soit de modifier des propriétés du composant implémentant le *TypeSupport* à évoluer, soit déplacer tous les fils de l'ancien composant *TypeSupport* vers le nouveau, si le composant du nouveau *TypeSupport* à utiliser existe déjà. Chacune des opérations nécessitera des pré/post-traitements qui correspondent respectivement à l'arrêt et au démarrage du composant *TypeSupport* parent des composants fils. Ces traitements seront répétés sur tous les composants fils, de ce fait, il n'y a même pas à s'inquiéter des re-configurations à appliquer dans les composants fils, puisque celle-ci sera automatisée, c'est à dire que les objets seront reconstruits contextuellement au nouveau type de donnée.

D'autres exemples d'adaptation sont grandement facilités par exemple avec le partitionnement de domaine DDS. Dans ce cas, il suffit de vérifier qu'un *Publisher* ou un *Subscriber* ait bien le parent de type *Partition* attendu. Ou mieux, si il y a besoin de changer de domaine qui sert de cloisonnement physique dans les échanges de données DDS, il suffit d'effectuer des reconfigurations au niveau d'un composite de type *Domain* pour voir l'adaptation au nouveau domaine DDS de tous les composants fils à différents niveau.

Le tableau 3.2 enumère les différentes adaptations facilitées par R-DDS, par niveau de difficulté de réalisation dans le modèle DDS.

Table 3.2 – Adaptations facilitées par R-DDS triées par niveau de difficulté d'application avec le modèle DDS (voir 3.2.5 pour plus de détails)

Adaptation tasks	**Difficulty level application in DDS, from + to +++**
Domain Change	+++
Data Type Change	+++
Static QoS (persistency for example)	++
Copy of configuration	+
Dynamic Property Change	+

Ainsi, une reconfiguration complexe mais nécessaire dans un environnement dynamique est plus simple à réaliser avec R-DDS qu'avec DDS. Il suffit d'une opération de re-configuration avec un composite parent (modification d'une propriété ou modification d'un parent) pour que tout le modèle sous-jacent se reconstruise.

3.3 Mise en œuvre de R-DDS

L'implantation réutilise le modèle à composants FraSCAti (voir la sous-section 2.6.6, page 45). Ainsi, le langage de configuration dépend d'un fichier ".composite" SCA, et d'autres outils sont mis à disposition pour accompagner des tâches de re-configuration à l'exécution.

Elle est disponible dans un projet svn à l'adresse suivante :

`http://websvn.ow2.org/listing.php?repname=frascati&path=/sandbox/jlabejof/RDDS/`.

Les implantations OpenSplice et RTI de DDS ont été embarquées dans l'architecture R-DDS pour vérifier l'interopérabilité. Cette interopérabilité est possible si et seulement si les implantations couvrent les fonctionnalités proposées par la spécification, et pas nécessairement la signature des classes, puisque le but est d'embarquer un moteur d'exécution DDS dans des composants d'adaptation pour rediriger les

appels d'une interface DDS vers les composants de contrôle ou vers le même moteur d'exécution DDS. Fort heureusement, les deux implantations couvrent tous les aspects attendus, et la portabilité a pû être vérifiée.

La figure 3.3 représente trois vues d'un même système DDS. Y sont représentées de haut en bas, la configuration avec le contenu d'un fichier ".composite" SCA, la vue composant puis la vue opérationnelle du système R-DDS. Dans ce système, il y a un domaine composé de deux partitions. Les deux parties du bas montrent de manière plus compréhensible comment et de quoi est composé le système. La vue composant affiche trois domaines, soit le *DomainComposite* fils, le *DomainA* intermédiaire et le *DomainB* englobant. L'idée est de se servir des composites englobants de même nature pour associer le principe de spécialisation des composites internes. Ainsi, la troisième vue ne laisse apercevoir que le *DomainB*, tout en conservant les partitions *PartitionA* et *PartitionB*. Par ailleurs, la définition de la propriété "Status" au niveau de *DomainB* n'est pas visible, car elle n'a pas été promue par *DomainA*, et est donc interprétée comme une propriété privée de *DomainA*.

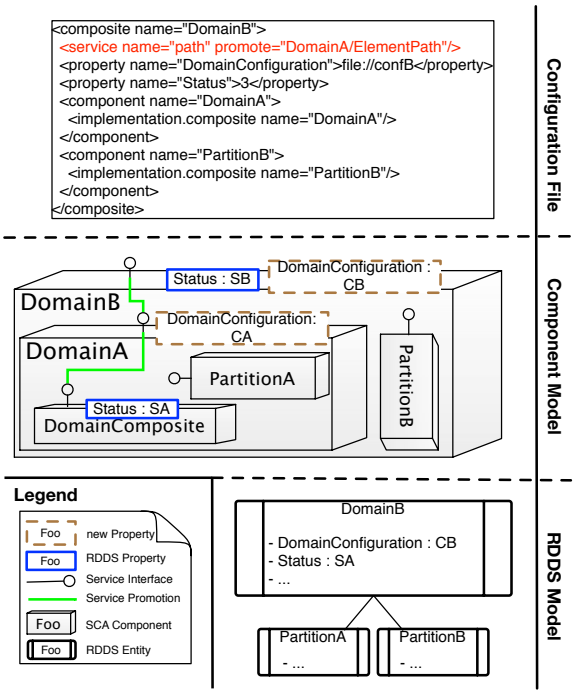


FIGURE 3.3 – **Configuration d'un domaine et de partitions R-DDS** – Vue composite SCA, composant et opérationnelle

Finalement, la propriété *DomainConfiguration* est une propriété propre au moteur d'exécution utilisée par la configuration, et permet d'indiquer la configuration du domaine associé.

Un dernier point important avant de terminer ce chapitre concerne la configuration du partage de composant. Comme expliqué dans la sous-section 2.6.5, page 45, contrairement à Fractal, le modèle SCA ne considère pas la propriété de composant partagé. De ce fait, la mise en œuvre propose une propriété de nom "kind", de type String, et portée par le concept d'*Element* qui justifie le genre d'un composite définit dans un fichier de configuration SCA.

Ce genre peut prendre les trois valeurs suivantes :

new_instance valeur par défaut si le genre n'est pas indiqué. Le comportement sera le même que celui spécifié par la spécification SCA,

shared indique que le composant est partagé, dans ce cas, le nom du composite sera l'identifiant du composite partagé dans le modèle R-DDS,

inherits_from relation d'héritage entre composants.

Cette mise en œuvre réalise tout ce qui a été décrit dans ce chapitre, et elle est évaluée dans le chapitre 6, page 109.

3.4 Conclusion et Résultats

L'extension R-DDS de l'architecture DDS a permi d'introduire la capacité d'adaptation pour des besoins très fins dans les systèmes RT-E (section 3.1) de manière transparente pour la spécification.

La réponse proposée se concentre sur un découpage fonctionnel de DDS (sous-section 3.2.1) et sur l'ajout de la capacité de re-configuration via un modèle à composants réflexifs couplé à l'approche Fractal (sous-section 3.2.2) pour enrichir la spécification de manière à faciliter sa configuration et sa maintenance (section 3.2).

De manière à compléter l'approche en s'intéressant à l'exécution de R-DDS, la section 6, page 109 évalue la mise en œuvre dans un SoS inspiré d'un système existant, et fait foi d'une utilisation possible de l'architecture DDS étendu.

Un seul résultat a vu le jour sous la forme d'un brevet THALES en court de dépôt en Europe et aux État-Unis en 2011.

Chapitre 4

R-MOM – Un intergiciel asynchrone, adaptatif et interopérable

Sommaire

R-MOM est une architecture qui vise à accompagner un système dans l'utilisation du paradigme de communication MOM. R-MOM facilite la configuration et la re-configuration de besoins fonctionnels (description et valeur de l'information à échanger) et non-fonctionnels (support de qualités de services, protocol d'échange, règle d'encodage, etc...) de ce paradigme dans un système.

Ainsi, R-MOM peut-être vu à haut niveau comme un MOM adaptable et interopérable avec d'autres MOM.

4.1 Pourquoi avoir un MOM adaptable et interopérable ?

Dans le monde des SoS (sous-section 2.2.1, page 12), de nombreux domaines d'application doivent intéragir ensemble à l'aide de différents paradigmes de communication. Chaque domaine d'application a des contraintes qui lui sont propres, l'embarqué est sujet à des contraintes en quantité de mémoire, le temps-réel dur doit utiliser des infrastructures physiques sûres pour profiter de couches logicielles déterministes, etc... Les intergiciels de type MOM doivent répondre à ces contraintes, et c'est pourquoi ils sont techniquement plus ou moins aptes à répondre à un domaine en particulier. DDS par exemple, est un intergiciel conçu pour les systèmes temps-réel et embarqué, et c'est pourquoi l'API d'envoi et de réception de donné est spécifiques aux types de données. JMS est plus proche des problématique haut niveau, et son API est plus flexible et générique. L'envoi et la réception de données n'est pas spécifique à un type particulier, mais JMS propose un nombre de QoS plus restreint que DDS. Il n'y a pas de solution idéale tout domaine confondu. Et même si une technologie est plus apte à répondre à un domaine d'application,

il s'avère que le choix technique est devancé par un choix politique, qui joue en faveur d'accord de licence entre distributeurs de technologies avec les relations des commerciaux et architectes du système. Malgré tous ces choix, il convient à un moment donné de s'accorder sur la nature des informations à transmettre, et donc de s'abstraire des moyens de communication à utiliser.

Dans ce cas, l'interopérabilité entre données et protocole de communication est une solution, qui garantit la transmission des données sans les problématiques liées aux choix politiques et techniques de conception du système.

Quand à l'adaptation, il s'agît d'une étape nécessaire dans l'exécution d'un système soumis à un environnement changeant. C'est-à-dire, où les exigences évoluent comme dans tous les systèmes actuels. Un MOM est capable d'assurer des propriétés non-fonctionnelles (sécurité, filtrage, etc...) comme fonctionnelles (nom logique de distribution, type de message, etc...). Le rendre adaptatif, c'est lui permettre de se mettre à jour en considérant de nouvelles exigences, comme par exemple de nouvelles politique de sécurité, d'encodage (propriétés non-fonctionnelles) ou de structure de données à envoyer ou recevoir (propriétés fonctionnelles). Les MOMs actuels ne sont pas extensibles à ce point, et sollicitent à l'application source et cible de la communication, de prendre en charge ces aspects d'adaptation.

De ce fait, l'adaptation est nécessaire dans les MOM.

Le couple adaptation et interopérabilité est nécessaire pour garantir la transmission d'une information avec le respect d'exigences systèmes qui évoluent.

4.2 Architecture de R-MOM

L'architecture de R-MOM est basée sur un modèle à composant réflexif. R-MOM propose un découpage et un assemblage fonctionnel de l'application en composants réflexifs. Le modèle résultant comprend une partie statique propre à la portabilité de l'application, et une partie dynamique spécialisable pour les besoins systèmes.

Ainsi, dans un premier temps, nous établissons la liste des fonctionalités du paradigme MOM. Puis, nous établissons un schéma de dépendance statique entre ces fonctions qui permette de réaliser toutes les opérations d'adaptation possibles.

4.2.1 Découpage fonctionnel d'un MOM

Comme présenté dans la section 2.5.4, page 31, une application MOM comprend des entités pour envoyer et recevoir des messages. Un message est une donnée applicative sérialisable et soumise à un ensemble de qualités de services. Le message est transmis de producteur à consommateur de message en respectant une distribution physique (vers une queue), ou une distribution logique (vers un sujet).

L'étude décompose le paradigme MOM en la liste des six fonctionnalités suivantes :

Le but recherché d'un MOM est l'envoi et la réception de données de manière asynchrone entre applications. Ainsi, on se préoccupe un minimum de comment, de quand et par qui vont être consommées les données, seules les opérations de production et de consommation sont importantes. Dans ce contexte, on appellera de telles données des messages. Ainsi, un message est produit ou consommé selon un certain protocole **P**. Pour transmettre les messages sur le réseau, il est important de pouvoir sérialiser les messages suivant un encodage particulier **C**, et une description précise de la structure de la donnée **D** permettant de reconstruire ou de vérifier l'intégrité d'un message. La transmission des messages répond à une politique de distribution de messages **B**, afin de garantir une certaine logique de transmission des messages vers les consommateurs. En considérant ce paradigme comme une SOA, un message est accompagné de qualités de services **Q** influant sur son comportement dans le système. Finalement, les éléments de consommation de messages dans une chaîne de distribution peuvent vouloir filtrer **F** les valeurs de messages reçus.

Les acronymes du tableau 4.1 résument l'ensemblent des fonctionalités décritent dans le chapitre précédent, et seront réutilisés dans toute la suite de cette section.

Table 4.1 – Fonctions et capacités attendues par l'approche MOM

Acronymes	Fonction/Capacité
P	Protocole d'envoi et de réception du message
B	Logique de Distribution du message
C	Méthode de (dé)sérialisation du message
Q	Application des qualités de services sur des messages
D	Description du message
F	Filtrage du message ou de sa structure

4.2.2 Assemblage fonctionnel d'un MOM

L'assemblage des six fonctions du paradigme MOM identifiées dans la sous-section précédente, doit pouvoir couvrir ces fonctions le plus simplement et exhaustivement possible, mais également, permettre de faciliter toute tâche d'adaptation de tâche.

Voici une liste des dépendance entre fonctions :

1. **P** a besoin de **Q**, **C**, **F** et **B** respectivement pour traiter les informations non-fonctionnelles du message, pour échanger un message avec la couche de transport réseau, pour filtrer un message reçu, et pour établir le routage du message,
2. **C** a besoin de **D** pour résoudre dynamiquement la sérialisation d'un message,
3. **Q** a besoin de **P** pour traiter les informations non-fonctionnelles relatives au message.

La logique de dépendance entre fonctions fait intervenir certaines informations relatives à un message, comme sa description, ou ses propriétés non-fonctionnelles. De ce fait, nous allons nous servir d'une donnée générique pour échanger ces informations entre fonctionnalités. AMQP (voir section 2.5.7, page 36) propose le concept d'*Envelope* en tant que structure pour transporter des informations liées à un message. La structure de l'*Envelope* n'a pas vocation à être modifiée dans le temps. Notre reprenons ce concept pour assurer l'interopérabilité au niveau de la couche de transport, et aussi au niveau de l'interface d'échange d'information entre composants fonctionnels.

Ainsi, tout composant souhaitant traiter une *Envelope* doit implémenter l'interface *IEnvelopeProcessor* (voir la figure 4.2), qui propose la méthode de nom *process*, prenant en paramètre une *Envelope* et retournant un entier comme résultat du traitement. Plus de détail sur l'*Envelope* sont donnés dans la sous-section 4.2.6.

Cette interface est suffisamment complète pour que l'architecture n'ait pas besoin d'en utiliser une autre, et surtout simple pour facilement comprendre sa portée conceptuelle sur l'approche R-MOM.

4.2.3 Projection du paradigme MOM dans un modèle à composants

La projection du paradigme MOM dans un modèle à composants vise à proposer un assemblage de composants où chacun est à charge d'exécuter une fonction.

À l'image du paradigme MOM, l'architecture de R-MOM se focalise autour de la donnée à échanger, c'est-à-dire l'*Envelope*. Ainsi, l'assemblage proposé est une spécialisation de deux types de composants. Le type *EnvelopeProcessor* traite les *Envelope*s, et le type *BoundEnvelopeProcessor* nécessite un composant de type *EnvelopeProcessor*. L'assemblage final vise à réutiliser ces deux composants, pour concevoir une production, un traitement et une consommation des *Envelope*, à travers les six fonctions identifiées dans la sous-section précédente.

Le diagramme de composants UML de la figure 4.1 illustre les explications relatives aux spécialisations de ces deux familles de composant. La légende associe les composants R-MOM avec les fonctionalités/capacités de l'approche MOM que nous avons listé dans ce manuscrit.

Ainsi, on associe **P** aux types de composants *EnvelopeConsumer* et *EnvelopeProducer*. Ces deux types de composants intéragissent directement soit avec une application (spécialisation en composants de type *MessageProducer* et *MessageConsumer*), soit avec le réseau ou une liaison de communication avec un autre

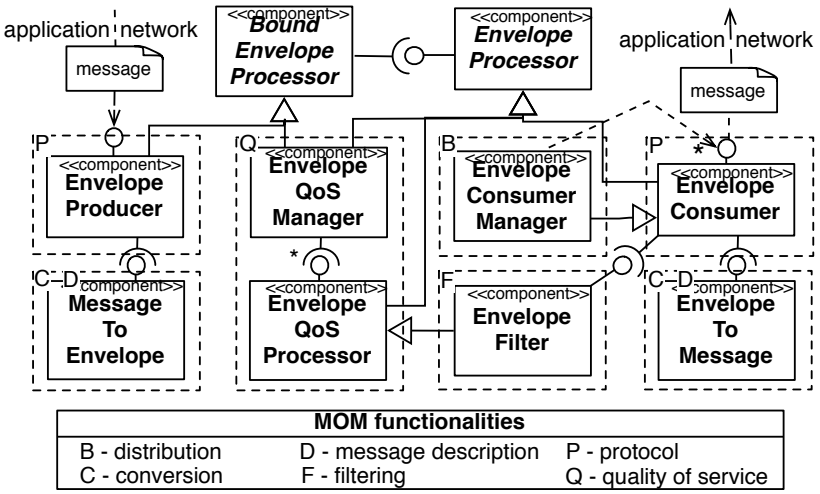


FIGURE 4.1 – **Vue de l'architecture R-MOM** – Diagramme de composants UML

intergiciel (spécialisation en composants de type *EnvelopeReceiver* et *EnvelopeSender*). **Q** est implémenté par l'*EnvelopeQoSManager* qui est à charge de gérer des exécuteurs de qualités de service joués par des composants de type *EnvelopeQoSProcessor*s. **B** est géré localement par l'*EnvelopeConsumerManager* qui est responsable d'une liste de composants de type *EnvelopeConsumer*s dont il hérite. **C** et **D** vont être assurés en tête et bout de l'assemblage, respectivement par les composants de type *EnvelopeToMessage* et *MessageToEnvelope*. Et **F** va enrichir les données consommées par un *EnvelopeConsumer* à l'aide d'un unique *EnvelopeFilter* qui hérite de la famille des *EnvelopeQoSProcessor*.

Voyons à présent en détail les interfaces des types de composants.

4.2.4 API de R-MOM

Il existe une interface par type de composants, où le nom est précédé par le caractère *I* pour "Interface".

La diagramme de classe de la figure 4.2 représente l'ensemble des interfaces décrites ci-dessous, soit une quinzaine d'interfaces pour quinze types de composants, devant couvrir l'ensemble des besoins attendus par le paradigme MOM. mais aussi assurer des adaptations propres à différents domaines d'application. L'interopérabilité est offerte par des composants qui respectent le patron de conception de l'adaptateur [Wolfgang, 1994], situés en bout de chaîne de l'assemblage qui comprend la production d'une *Envelope*, l'exécution des qualités de services et la consommation de l'*Envelope*.

Voici en détail l'utilisation de chaque interface dans les différentes étapes de traitement des *Envelope*s.

Production d'une *Envelope*

La production d'une *Envelope* est la première étape dans la chaîne d'assemblage de composants R-MOM. Sa réalisation est possible à l'aide d'une spécialisation du composant *EnvelopeProducer*. Deux spécialisations sont possibles, en fonction d'où provient le message à traiter, c'est à dire depuis une application, ou depuis un protocole de communication.

Si le message vient d'une application, il est préconisé d'implémenter l'interface *IMessageProducer*, qui offre soit la méthode propre à une production générique de messages, avec un dictionnaire de qualités de services, et des données applicatives, soit la production directe d'*Envelope*. Dans le cadre d'une portabilité de technologies avec R-MOM, il est possible de lier un composant de liaison d'interface (voir la sous-section 4.2.5) qui implémente une interface propre à la technologie liée, mais qui va rediriger l'appel vers le composant implémentant l'interface *IMessageProducer*.

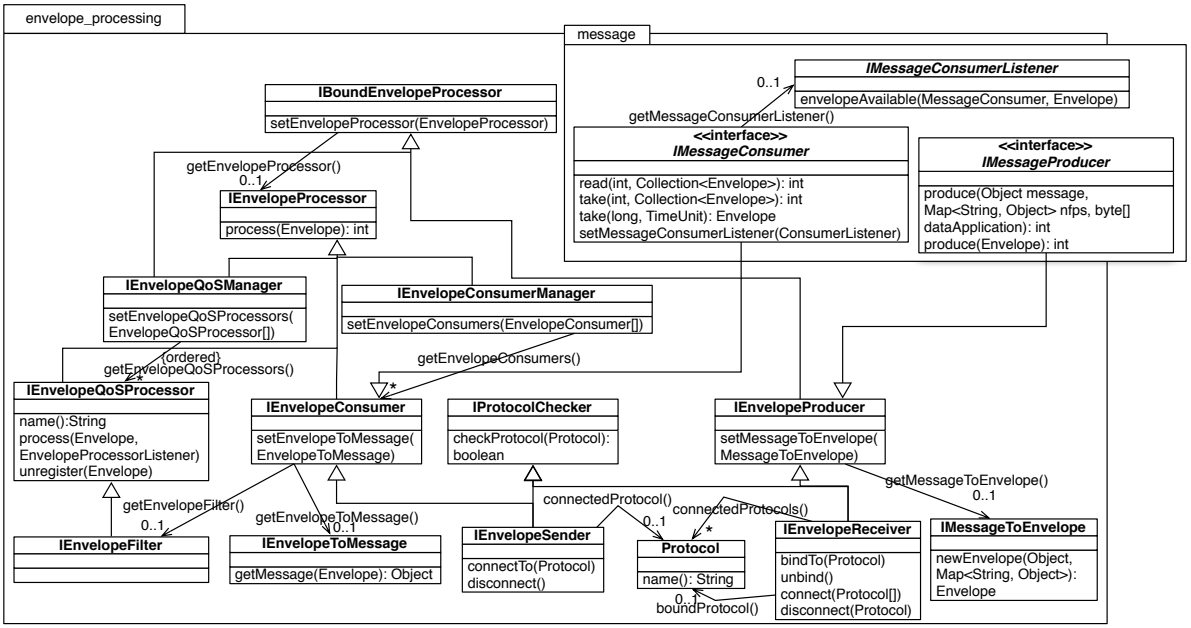


FIGURE 4.2 – **Interfaces des composants** – Diagramme de classes UML

Si le message vient d'une technologie de communication, il est faut utiliser un composant dit de liaison protocolaire (voir la sous-section 4.2.5) qui implémente l'interface *IEnvelopeReceiver* de la figure 4.2.

Au final, chaque *Envelope* produite pourra être traitée par un composant implémentant l'interface *IEnvelopeProcessor*, c'est à dire un composant de type *EnvelopeQoSManager* ou de type *EnvelopeConsumer*, en prenant garde de lui ajouter des verrous d'exclusivité d'exécution dans le cas d'accès concurrents si plusieurs *EnvelopeProducer* sont liés au même *EnvelopeProcessor*.

Exécution des qualités de services d'une *Envelope*

Cette partie est optionnelle dans le processus de traitement d'une *Envelope*, au cas où le système ne souhaite pas exécuter de qualités de services.

L'idée est de proposer à l'utilisateur une chaîne de responsabilité [Gamma, 1995] pour traiter un ensemble de qualités de services à la carte, identifiables à l'aide d'un nom, et ce de manière dynamique. Ainsi, le composant *EnvelopeQoSManager* implémente l'interface *IEnvelopeQoSManager*. Ce composant entretient une liste de composants de type *EnvelopeQoSProcessor* triés par nom de la qualité de services à exécuter. Lorsque l'*EnvelopeQoSManager* traite une *Envelope*, il construit un tableau d'*EnvelopeQoSProcessor*s relatifs aux noms de qualités de services identifiées dans l'*Envelope*. Ensuite, il délègue le traitement de l'*Envelope* aux différents *EnvelopeQoSProcessor*s sélectionnés, de manière synchrone ou asynchrone en fonction du choix de l'utilisateur.

Un *EnvelopeQoSProcessor* offre la méthode $process(Envelope, EnvelopeProcessorListener) : integer$ pour traiter une *Envelope*. Le tableau 4.2 liste les valeurs de retour possibles. En fonction des différents résultats de traitement, l'*Envelope* est passée au prochain *EnvelopeProcessor* lié, ou supprimé.

La logique de traitement est la suivante : Soit un *EnvelopeQoSManager* (*EQM*) qui contient une liste (*EQPs*) d'*EnvelopeQoSProcessor*s (*EQP*) aptes à traiter une *Envelope* (*E*). Le traitement de *E* par *EQM* commence par sélectionner un *EQP* parmi *EQPs* pour traiter *E*. La suite dépend du résultat du traitement :

0 Si le traitement est un succès, on retire *EQP* de *EQPs*. Si la liste devient vide, alors le traitement des qualités de services de *E* est terminé, et l'*Envelope* est donnée à l'*EnvelopeProcessor* lié à l'*EnvelopeQoSManager*,

TABLE 4.2 – Valeur et signification du traitement de qualités de services d'une *Envelope*

Résultats du traitement	**Description**
0	Succès du traitement
1	Succès temporaire du traitement
2	Pause du traitement
4	Échec du traitement

1 si le traitement est temporairement un succès, comme par exemple dans la vérification de la durée de vie d'une donnée, on se retrouve dans le cas **0**,

2 si le traitement est en pause, comme par exemple dans le cas d'une réception ordonnée de données suivant l'ordre d'envoi, on ne fait rien de plus car on doit attendre la fin du traitement,

4 si le traitement est un échec, on appel la méthode $unregister(E)$ de tous les *EQP* de *EQPs* afin de les avertir que l'*Envelope* ne doit pas être consommée.

Les cas **1** et **2** font intervenir un besoin de traitement asynchrone puisqu'il s'agît de tâches temporelles non bloquantes. De ce fait, chaque *EQP* notifie l'interface *EnvelopeProcessorListener* associée à l'*Envelope* à traiter pour avertir du résultat, en se passant en paramètre la notification. La méthode de notification respecte la signature $processed(EnvelopeProcessor, Envelope, int)$. Les paramètres sont respectivement l'*EQP* source du traitement, l'*Envelope* traitée et le résultat du traitement.

Une fois que les qualités de services de l'*Envelope* sont traitées avec succès, l'*Envelope* devient consommable, et transmise à l'*EnvelopeProcessor* lié à l'*EnvelopeQoSManager*.

Consommation d'une *Envelope*

Une *Envelope* qui est consommable est récupérée par un composant de type *EnvelopeConsumer*, en bout de chaîne d'assemblage. Ce type dépend de ce que l'utilisateur souhaite faire, c'est à dire consommer les *Envelope*s via une application, les transmettre via un protocole particulier, ou finalement gérer la distribution locale.

S'il s'agît de transmettre l'*Envelope* vers une application, il suffit d'utiliser un composant de type *EnvelopeConsumer* implémentant l'interface *IMessageConsumer*. Cette interface s'inspire des différentes interfaces existantes dans les différents intergiciels respectant le paradigme MOM, et de DREAM (sous-section 2.5.8, page 37) pour proposer tous les moyens connus de consommer une *Envelope*, en mode dit de "poussée" ou de "tirage". Une fois un tel composant mis à disposition, il est possible d'y lier un autre composant appelé composant de liaison d'interface (voir la sous-section 4.2.5) afin de garantir une portabilité de R-MOM dans des applications utilisant une autre technologie de communication.

S'il s'agît de transmettre l'*Envelope* vers une technologie de communication tierce, il suffit d'utiliser un composant de type *EnvelopeConsumer* qui utilise le protocole de la technologie ciblée et implémente l'interface *IEnvelopeSender* de la figure 4.2. Ce type de composant s'appel un composant de liaison protocolaire (voir la sous-section 4.2.5).

Finalement, s'il s'agît de gérer la distribution locale de l'*Envelope*, le composant *EnvelopeConsumerManager* (qui hérite du type *EnvelopeConsumer*) propose de gérer une liste d'*EnvelopeConsumer*s à charge de consommer une *Envelope* reçue. Ainsi, il devient possible de spécifier localement le mode de distribution, permettant ainsi de respecter des politiques de distribution courantes, comme "un vers plusieurs", ou "un vers un". À l'exécution, ce composant permet de gérer plusieurs protocoles de consommation d'*Envelope*s en même temps, et de parfaire des modifications de certains protocoles sans gêner la disponibilité des autres composants de la liste de distribution. Cette dernière caractéristique met en évidence la possibilité de changer de liaison de protocole en limitant les pertes de données.

Ainsi, nous terminons la description de la chaîne d'assemblage de composants R-MOM pour répondre efficacement à tous les patrons de conception d'une telle architecture, en incluant la prise en compte d'adaptation et d'interopérabilité avec des préoccupations de disponibilité du système.

4.2.5 Interopérabilité et portabilité via des liaisons d'interface et protocolaire

L'interopérabilité est supportée par le type de liaison protocolaire et la structure des transporteurs de données définit dans la sous-section 4.2.6. La portabilité du système est assurée par le type de liaison d'interface. Les deux types de liaison visent à assurer la valeur et les qualités de services d'un message entre différents technologies MOM d'un système.

Liaison d'interface

La liaison d'interface est un type de liaison qui propose l'API d'un MOM pour produire ou consommer des *Envelope*s dans un assemblage de composants R-MOM. L'idée est de proposer pour une technologie T ciblée, un composant qui implémente une interface provenant de l'API de T, et qui redirige les appels provenant de l'interface vers des composants de type *EnvelopeConsumer* et *EnvelopeProducer*. Le message et ses qualités de service sont transformées pour être transportées dans une *Envelope* R-MOM.

Afin de respecter le modèle d'une technologie MOM liée à R-MOM, chaque composant de liaison d'interface peut être lié à un composant de liaison protocolaire. Ainsi, les appels de parcours de modèle sont résolus par liaison interface-protocole. Et les appels de modification du modèle (modification de qualité de services, envoi/réception de messages) sont redirigés vers des composants R-MOM intermédiaires avant d'atteindre les différents composants protocolaires.

Liaison protocolaire

La liaison protocolaire est un type de liaison qui propose d'utiliser le protocole d'envoi et de réception de messages d'une technologie MOM. L'idée est de spécialiser les composants R-MOM de types *EnvelopeSender* et *EnvelopeReceiver*. Ces derniers auront à charge de transformer les qualités de services d'une *Envelope* en celles promues par la technologie de communication utilisée, et inversement.

Configuration et application de l'interopérabilité

Les possibilités d'interopérabilité sont présentes en bout de chaîne d'assemblage des composants R-MOM, c'est à dire au niveau des composants de production et de consommation d'*Envelope*s.

Les interfaces *Protocole* et *IProtocolChecker* de la figure 4.2 contribuent dans la configuration et l'utilisation des composants de liaison protocolaire.

La configuration d'une liaison de protocole s'inspire de 0MQ à l'aide d'une simple URI pour configurer le mode de réception ou d'envoi de données. Le format général de l'URI répond à l'expression régulière suivante :

$\$protocol' :' [\$user['{:}' \$password]'@'][\$high\_destination['{:}' \$low\_destination]][['.']'/'\$conf\_file]$

où les crochets indiquent une entrée optionnelle, et guillements sont des champs nécessaires, et les valeur préfixées par $ sont des variables correspondent à :

protocol Le protocol à utiliser. UDP ou JMS par exemple,

user & password le nom et le mot de passe de l'utilisateur pour une connexion authentifiée,

high_destination & low_destination informations haute et basse de la destination. "queue :default" pour JMS dans le cas de l'utilisation d'une queue de nom "default' par exemple, ou "192.0.0.1 :2525" pour l'utilisation d'UDP avec l'adresse "192.0.0.1" et le port "2525",

conf_file chemin d'accès vers un fichier de configuration du moteur d'exécution de la liaison.

4.2.6 Envelope : Concept de transport de messages et de propriétés

Inspiré du travail réalisé par AMQP, l'*Envelope* est dédiée au transport de messages et d'informations non-fonctionnelles du message (description, qualités de services et méthode d'encodage). Cet objet est transmis entre composants R-MOM et sur le réseau pour assurer de l'intéropérabilité entre MOMs, mais aussi permettre l'adaptation des données.

L'*Envelope* AMQP est actuellement la solution étudiée par la plupart des grands acteurs autour de la conception des MOMs, et de mise en place de solutions dynamiques et interopérables. Nous le prendrons

donc comme référence absolue dans le domaine de recherche d'interopérabilité entre MOM dans toute cette sous-section, et en conséquence, chaque amélioration sera annoncée comme majeure car meilleure que celles proposées par ce groupe d'acteur.

L'objectif de cette sous-section est d'améliorer les points d'interopérabilité proposés par AMQP en ajoutant plus de flexibilité sur la méthode d'encodage du message et des propriétés non-fonctionnelles, tout en garantissant une taille de sérialisation plus petite que celle d'origine.

Structure de l'*Envelope*

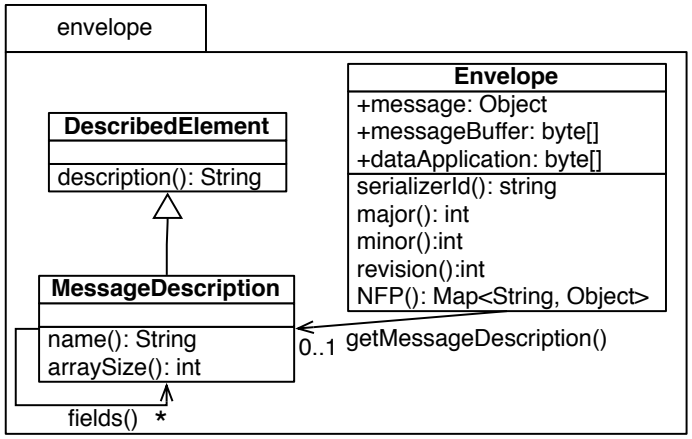


FIGURE 4.3 – **API de l'*Envelope* de R-MOM** – Diagramme de classes UML

La figure 4.3 présente l'API mise à disposition pour manipuler une *Envelope* durant le processus de traitement des composants R-MOM. Ainsi, elle contient un certain nombre d'informations relatives au message, correspondants aux champs :

message la valeur,

getMessageDescription() la description,

NFP() les propriétés non-fonctionnelles qui contient des couples de nom et valeurs de qualités de services,

messageBuffer la valeur sérialisée couplée à la méthode de sérialisation décrite par :

- **serializerId()** un identifiant textuel,
- **major(), minor() et revision()** une version logicielle décomposée en trois valeurs entières,

dataApplication des données d'application non traitées par R-MOM.

L'interopérabilité est couverte avec la description de la donnée transmise qui est agnostique de la technologie utilisée au dessus de la couche réseau. Ainsi, les entités d'envoi et de réception de données peuvent s'adapter à la description de la donnée.

L'adaptation est couverte avec la version logicielle jointe à la description de la donnée qui permet de mieux gérer les évolutions de la description.

Si l'*Envelope* doit être sérialisée, elle respectera la structure décrite dans le tableau 4.3.

La réduction de la taille des messages est aidée des méthodes *varint* et *varbytes*.

TABLE 4.3 – Contenu de l'*Envelope* sérialisée

PARTIE	PROPRIETE	TAILLE (Octets)
HEADER	body position	≥ 1
	serializer Id	≥ 0
	QoS	≥ 0
BODY	foot position	≥ 1
	serializer Id	≥ 1
	Message	≥ 0
FOOTER	Application Data	≥ 0

varint :

Tout entier de l'*Envelope* est appelé *varint*, et utilise la méthode d'encodage du même nom du canevas de sérialisation "ProtoBuf" développé et utilisé par Google [a]. Cette méthode permet d'utiliser autant d'octets nécessaires qu'il y en a contenant des bits positifs dans la valeur binaire de l'entier. Par exemple, l'encodage de la valeur 3 valant 11 en binaire est réalisé sur un octet. 512 qui vaut 10 00000000 en binaire est codé sur deux octets, etc...

a. `http://code.google.com/p/protobuf/`

Le but du *varint* est d'éviter la consommation d'octets inutile dans la sérialisation d'une *Envelope*.

varbytes :

Tout tableau ou simple champ texte (où chaque caractère est encodé sur un octet) de l'*Envelope* est appelé *varbytes*. L'encodage est précédé par un *varint* donnant la longueur du tableau d'octets. Si le tableau est vide, la longueur est de 0, et donc la taille du *varbytes* est de 1.

La version sérialisée comprend trois parties, nommées respectivement l'en-tête, le corps et le pied.

L'en-tête L'en-tête de l'*Envelope* s'intéresse aux qualités de services car ce sont les informations qui seront traitées en priorité par les composants R-MOM. Dans le cas contraire, le premier entier renvoit à l'offset correspondant au premier octet du corps de l'*Envelope*. Le second entier contient la méthode d'encodage des qualités de services. Le troisième champ correspond aux valeurs des qualités de services.

La règle d'encodage permet d'adapter R-MOM à des besoins système, soit fortement contraints en mémoire, soit au contraire aptes à jouir des principes dynamiques liés aux opérations d'évolution les plus complexes possibles.

Trois valeurs par défaut sont proposées. L'idée est de couvrir aussi bien des besoins dynamiques, ouvrant la voix vers de la découverte de QoS, comme des besoins fortement contraints visant une empreinte mémoire minimale. Ainsi, chaque valeur va influer dans la manière d'encoder les qualités de service.

Voici les types d'encodage par défaut, où les valeurs sont encodées dans des *varbytes*, chacunes précédées par un identifiant présent uniquement dans les solutions dynamiques :

0 : Solution dynamique et générique. L'identifiant enregistré est le nom du type de QoS enregistrées dans un *varbytes* via un simple champ texte, permettant de réaliser de la découverte de QoS.

1 : Solution entre dynamique et spécifique. L'identifiant enregistré est un entier codé dans un *varint*. Cet entier doit être résolu par l'entité qui reçoit l'*Envelope*, et donc répond à un besoin de qualités de services optionnelles.

≥ 2 : Solution la plus spécifique à des besoins systèmes précis et non évolutifs au niveau de l'*Envelope* car aucun identifiant n'est enregistré. L'ordre et l'utilisation des qualités de services doivent être connus à l'avance.

L'en-tête propose d'accompagner un message avec un nombre illimité de qualités de service, contrairement à AMQP qui est limité à $255 * 4 - 8 = 1012$ octets. De plus, R-MOM permet de choisir le mode d'encodage des qualités de service en fonction des besoins, tout en réduisant le nombre de bits des identifiants par rapport à AMQP qui impose un mode d'encodage statique.

Finalement, le nombre minimal d'octets alloués pour l'en-tête de l'*Envelope* R-MOM est de 1 (le premier champ vaut 1 si l'offset suivant correspond au corps de l'*Envelope*), contrairement aux 8 imposés par AMQP.

R-MOM est plus propice à l'adaptation et à l'interopérabilité des qualités de services, tout en garantissant une empreinte mémoire plus petite que celle d'AMQP.

Le corps Le corps de l'*Envelope* contient toutes les informations relatives à la valeur du message. Il contient trois champs. Le premier est la valeur de l'offset correspondant à la dernière partie de l'*Envelope*. Le deuxième champ correspond à la méthode de sérialisation. Et le troisième champ comprend la valeur sérialisée du message.

Le champ de la méthode de sérialisation comprend le nom de la méthode. Libre à l'utilisateur de compléter le nom de la méthode avec une information de version qui est nécessaire dans un environnement évolutif.

Si le nom de la méthode n'est pas donné dans le cas du simple transport d'un tableau d'octets, un seul octet est consommé.

L'*Envelope* AMQP propose huit octets fixes pour marquer la méthode de sérialisation, soit 4 octets pour le nom, un octet de transition, et un octet par propriété de version, soit un pour l'identifiant majeur, un pour l'identifiant mineur, et un dernier pour l'identifiant de révision. L'information de version proposée par AMQP est trop contraignante dans des systèmes qui n'en ont pas nécessairement besoin, et de plus, il n'est pas certain que toutes les cultures se suffisent à un tel format si le besoin se révelait.

De ce fait, AMQP impose 8 octets fixes pour gérer une méthode de sérialisation, contre au moins 1 octet pour R-MOM.

Le corps de l'*Envelope* de R-MOM est beaucoup plus flexible et d'empreinte mémoire beaucoup plus petite que dans le cas d'AMQP.

Le pied Cette partie est similaire à celle d'une *Envelope* AMQP. L'idée reste de positionner en dernière partie un tableau d'octets qui sera exclusivement traité par l'application. Cependant, contrairement à AMQP, R-MOM déduit la taille de ce dernier segment non pas d'une taille d'*Envelope* globale, mais à l'aide du *varbytes* utilisé.

Ainsi, dans le cas d'AMQP, la taille totale de l'*Envelope* est calculée à partir des 8 premiers octets, donc de taille maximale fixe. Dans le cas de R-MOM, la taille totale de l'*Envelope* est déduite de la taille des trois parties, qui sont chacune de taille variable.

4.2.7 Conclusion

Le découpage fonctionnel appliqué sur le paradigme des intergiciels orientés message, a permis d'introduire R-MOM, une solution proche des besoins de différents types de systèmes distribués en terme d'utilisation du modèle de distribution MOM. R-MOM adresse à la fois un assemblage complet et flexible des fonctionalités attendues d'un MOM, pour faciliter la configuration, et la maintenance avec une gestion de l'adaptation poussée et une interopérabilité des données et des protocoles complète.

De plus, le transport des données a été mûrement réfléchi pour étendre l'interopérabilité et l'adaptation des données sur le réseau, tout en réduisant un maximum la taille des méta-données à 2 bits dans le meilleur des cas (le troisième étant la taille du message qui apparait nécessaire), mais sans contraindre le choix du contenu du message. Ainsi, l'utilisateur peut utiliser des messages aussi verbeux que le format XML ou REST qui sont des standards d'utilisation dans les projets actuels, ou bien beaucoup plus spécifiques.

Comme expliqué précédemment dans la sous-section 2.5.7, page 36, l'interopérabilité doit être facile à implanter pour être assurée, de ce fait, R-MOM met l'accent sur la liberté de pouvoir réutiliser des standards de transport de donnée comme REST ou XML, mais aussi la configuration qui utilise le format

des URI. Sans cela, de nombreuses solutions d'interopérabilité peuvent être proposées, mais sans garantir leur mise en œuvre dans les technologies existantes ou à venir.

4.3 Mise en œuvre

Dans le but de concrétiser l'architecture R-MOM, une implémentation est proposée dans cette section, et précède une évaluation de l'architecture dans la section 6, page 109.

La sous-section 4.3.1 présente l'ensemble des technologies intégrées à R-MOM et des remarques propres aux phases de configuration et de maintenance des systèmes sous-jacents. Et la sous-section 4.3.2 analyse trois exemples d'intégration de technologies de communication.

Le tout est disponible à l'adresse suivante sous la forme d'un projet svn :

`http://websvn.ow2.org/listing.php?repname=frascati&path=/sandbox/jlabejof/R-MOM/`

4.3.1 Exécution et couverture des fonctionnalités

L'implémentation de l'architecture R-MOM se base sur le modèle à composant FraSCAti (sous-section 2.6.6, page 45). Ainsi, nous profitons de l'approche SCA pour faciliter la configuration des assemblages de composants R-MOM et de capacités réflexives provenant des travaux de Fractal (sous-section 2.6.4, page 43) pour faciliter la maintenance de tels assemblages durant leur exécution.

Actuellement, l'implémentation comprend dix types de composants de liaison, soient JGroups [32], JBossMQ [33], JORAM [34], ActiveMQ [35], OpenJMS [36], RabbitMQ [37], OpenSplice [38], 0MQ [39], KryoNet [40] et Esper [41]. Les protocoles de communication UDP et TCP sont également supportés dans le cas de simples et rapides échanges de données. Même si Esper et KryoNet ne font pas partie de la famille des MOM, leur API d'envoi et de réception de données a été adaptée aux liaisons d'interfaces de R-MOM.

Trois librairies de sérialisation sont utilisées pour implémenter des composants de type *EnvelopeToMessage* et *MessageToEnvelope* pour (dé)sérialiser les *Envelope*s : la sérialisation Java qui nécessite des objets implémentant l'interface Java *Serializable*, ProtoBuf [42] qui favorise l'évolution de la structure des données, la vitesse et la taille de l'encodage, et Kryo [43] qui fournit une taille inférieur et une vitesse supérieur d'encodage par rapport à ProtoBuf, mais qui reste spécifique au langage Java.

Finalement, cinq qualités de services DDS (quelques unes sont également utilisées dans JMS et AMQP) sont implantées et proposées via des composants de type *EnvelopeQoSProcessor*. Soit la durée de vie, l'ordonnancement, la persistance, la latence entre deux réceptions, et l'assurance de réception de données.

Configuration

La configuration se base sur les fichiers composites SCA pour injecter le code métier dans des composants, et réciproquement, injecter dans le code métier des références vers les composants liés. Une telle utilisation de DI joue en la faveur de l'approche SoC, en permettant de développer au niveau du code métier le code des composants sans se soucier du bon fonctionnement des autres composants. Et au niveau des composants d'assurer les liaisons entre composants, sans se soucier de la nature des traitements des codes métier.

32. `http://www.jgroups.org/`
33. `http://www.jboss.org/`
34. `http://joram.ow2.org/`
35. `http://activemq.apache.org/`
36. `http://openjms.sourceforge.net/`
37. `http://www.rabbitmq.com/`
38. `http://www.opensplice.com/`
39. `http://www.zeromq.org/`
40. `http://code.google.com/p/kryonet/`
41. `http://esper.codehaus.org/`
42. `http://code.google.com/p/protobuf/`
43. `http://code.google.com/p/kryo/`

Ainsi, il devient facile de composer des assemblages à la carte, en s'abstrayant en grande partie de la complexité des traitements réalisés par les composants. On peut se satisfaire des interfaces, et facilement localiser un composant source d'une erreur de traitement d'une *Envelope*.

Maintenance

La maintenance d'un assemblage de composants R-MOM tire sa force du découpage fonctionnel. Dans le cas d'une modification du modèle, il est possible de modifier une fonction sans stopper les flux de données qui ne dépendent pas d'elles, et surtout en préservant le contexte d'exécution.

Par exemple, la modification d'un composant de liaison de production d'*Envelope* va stopper le flux de données émis par ce composant, mais ne vas pas stopper l'activité de tous les autres composants qui ne dépendent pas de ce flux. Ainsi, les autres producteurs d'*Envelope* vont continuer d'envoyer des données vers un composant de type *EnvelopeProcessor* qui pourrait être commun à ces composants. Réciproquement, le changement de composants de consommation d'*Envelope* est garanti sans nécessité l'arrêt des autres composants du même type, à condition de les faire précéder dans l'assemblage par un composant de type *EnvelopeConsumerManager*.

La modification du modèle sans stopper les flux de données non-concernés par la modification, et la préservation du contexte d'exécution de l'assemblage garantit par les autres composants, sont essentielles lors d'opérations de maintenance d'un système distribué.

4.3.2 Intégration de JMS, AMQP et DDS dans R-MOM

À titre d'exemples, cette sous-section présente l'intégration de JMS, AMQP et DDS, à travers des liaisons d'interface et de protocole spécifiques aux différentes technologies.

Chacune des descriptions d'intégration décrit l'architecture des composants de liaison et la transformation de l'*Envelope* appliquée pour être utilisée par les différentes implémentations du paradigme MOM.

Intégration de JMS

Comme expliqué dans la sous-section 2.5.5, page 31, l'approche JMS facilite la portabilité des applications JMS via une API commune. Seul la création d'un objet de type *DestinationFactory* est spécifique à l'application utilisée. L'intégration de JMS réalisée dans R-MOM suit cette logique. Des composants de liaison spécifiques à l'API JMS sont proposés, et seules les liaisons de protocole requièrent un objet de type *DestinationFactory* pour satisfaire le transport de messages et la création d'objets JMS.

La figure 4.4 présente une vue architecturale de notre solution, découpée en trois colonnes, qui représentent respectivement à gauche et à droite les composants de liaison JMS pour la production et la consommation d'*Envelope*s. Et dans la colonne centrale, des composants génériques de production, de consommation et de traitement de qualités de services. Les liaisons d'interface sont préfixées par le champ "JMSMessage", et les liaisons protocolaires sont préfixées par le champ "JMSEnvelope".

L'implémentation de R-MOM a introduit des composants optionnels de type *DestinationFactory* pour les solutions JMS de nom ActiveMQ [44], JORAM [45], JBossMQ [46] et OpenJMS [47]. Le but étant que le moteur d'écécution JMS à utiliser dépende de ce composant. Les autres composants sont des composants spécifiques au modèle JMS.

Une *Envelope* sérialisée est transportée à l'aide du type de message JMS *BytesMessage* servant à transporter des tableaux d'octets. Un message JMS transporte des qualitées de service qui seront traitées ou non. Les qualités de services de l'*Envelope* qui peuvent être traitées par le moteur JMS sont copiées dans le message, dans le but de laisser la responsabilité de leur exécution au moteur de distribution de messages.

La figure 4.5 montre l'utilisation d'un fichier composite SCA pour la configuration minimale d'un assemblage de composants R-MOM pour la production de messages JMS.

44. `http://activemq.apache.org/`
45. `http://joram.ow2.org/`
46. `http://www.jboss.org/`
47. `http://openjms.sourceforge.net/`

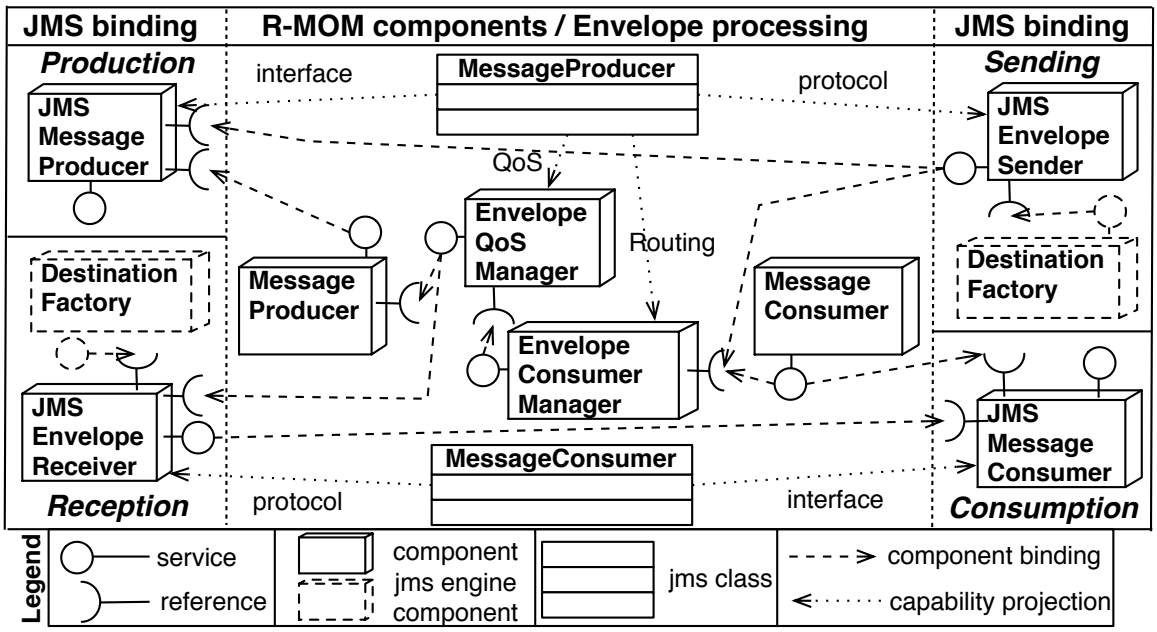


FIGURE 4.4 – **Architecture d'intégration de composants JMS dans l'architecture R-MOM** – projection des capacités

```
1. <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2. <sca:composite
    xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
    name="Producer">
3.  <sca:service name="SProducer" promote="producer/SProducer"/>
4.  <sca:component name="destinationFactory">
5.      <sca:implementation.java
            class="rmom.frascati.activemq.DestinationFactory"/>
6.      <sca:service name="SDestinationFactory"/>
7.  </sca:component>
8.  <sca:component name="producer">
9.      <sca:implementation.java
            class="rmom.frascati.jms.message.producer.Producer"/>
10.     <sca:service name="SProducer"/>
11.     <sca:reference name="REnvelopeProcessor"
            target="sender/SSender"/>
12.     <sca:reference name="RMessageToEnvelope"
            target="jmsMessageToEnvelope/SMessageToEnvelope"/>
13. </sca:component>
14. <sca:component name="jmsMessageToEnvelope">
15.     <sca:implementation.java
            class="rmom.frascati.jms.envelope.convertor.MessageToEnvelope"/>
16.     <sca:service name="SMessageToEnvelope"/>
17. </sca:component>
18. <sca:component name="sender">
19.     <sca:implementation.java
            class="rmom.frascati.jms.envelope.sender.EnvelopeSender"/>
20.     <sca:service name="SSender"/>
21.     <sca:reference name="RDestinationFactory"
            target="destinationFactory/SDestinationFactory"/>
22.     <sca:reference name="REnvelopeToMessage"
            target="envelopeToBuffer/SEnvelopeToBuffer"/>
23. </sca:component>
24. <sca:component name="envelopeToBuffer">
25.     <sca:implementation.java
            class="rmom.frascati.envelope.convertor.EnvelopeToBuffer"/>
26.     <sca:service name="SEnvelopeToBuffer"/>
27. </sca:component>
28.</sca:composite>
```

FIGURE 4.5 – **Configuration d'un assemblage de composants R-MOM pour la production de messages JMS** – fichier composite SCA

Le composite expose le service "SProducer" à la ligne 3, qui promeut l'interface de production de messages JMS du composant fils "producer", définit de la ligne 8 à la ligne 13. Il utilise le composant "jmsMessageToEnvelope" de transformation de messages JMS vers une *Envelope* (voir les lignes 14 à 17), et le composant de liaison protocolaire JMS de nom "sender" (voir les lignes 18 à 23) pour envoyer les messages sur le réseau. Ce dernier utilise le composant "destinationFactory" (lignes 4 à 7) pour pouvoir créer des objets de production de messages JMS avec l'aide de la technologie ActiveMQ, et le composant "envelopeToBuffer" (lignes 24 à 27) de sérialisation d'*Envelope*s.

Cette configuration ne considère pas l'exécution de qualités de services au niveau R-MOM, mais seulement celles prises en charge par ActiveMQ. Et cet assemblage n'autorise pas la possibilité d'éviter du temps d'indisponibilité du système dans le cas d'une modification de liaison protocolaire, car il n'y a pas de composants de type *EnvelopeConsumerManager*. Toutefois, il est nécessaire d'introduire deux composants de transformation pour convertir un message JMS en une *Envelope*, puis de la sérialiser pour l'introduire dans un *BytesMessage*. De ce fait, la configuration minimale pour produire des messages JMS au dessus de l'architecture R-MOM comprend 3 composants, soit un *JMSEnvelopeSender*, un *EnvelopeToMessage* et un *DestinationFactory*. Réciproquement, la consommation de messages JMS au dessus de l'architecture R-MOM comprend 3 composants, soit un *JMSEnvelopeReceiver*, un *MessageToEnvelope* et un *DestinationFactory*. Ces configurations minimales introduisent les capacités de configuration et de maintenance facilitées par R-MOM, et les autres composants sont optionnels, et peuvent facilement être liés à l'assemblage durant l'exécution du système.

Finalement, la configuration du protocole JMS pour R-MOM propose une URI respectant l'expression régulière suivante :

$''JMS :'' [\$user['':'' \$password]''@''](''topic''|''queue'')['':'' \$destination][[''.'']''/''\$configuration]$

Où \$user est le nom de l'utilisateur, \$password son mot de passe. \$destination est le nom du topic ou de la queue à utiliser. Et \$configuration est le fichier de configuration de la technologie JMS à utiliser.

Intégration d'AMQP

AMQP est une spécification d'un intergiciel orienté messages qui adresse l'interopérabilité entre technologies MOM. Contrairement à JMS, il n'y a pas d'API mais une architecture proposée au niveau protocolaire. De ce fait, la technologie RabbitMQ [48] a été utilisée pour promouvoir l'interopérabilité entre MOMs et la liaison AMQP.

Cette technologie offre l'objet *Channel* pour envoyer et recevoir des messages, et pas d'interface. De ce fait, chaque composant de liaison protocolaire de ce type utilise cet objet, et les composants de liaison d'interface implémentent une interface offrant toutes les méthodes du *Channel*.

RabbitMQ envoit et reçoit des données sous la forme de tableaux d'octets, et y ajoute le support de certaines qualités de services. Comme pour l'intégration de JMS, R-MOM utilise le protocole AMQP pour transporter des *Envelope*s sérialisées, et les QoS de l'*Envelope* pouvant être traitées par RabbitMQ sont copiées dans la procédure d'envoi.

Finalement, la configuration de ce protocole propose une URI respectant l'expression régulière suivante :

$''RabbitMQ :'' [user['':'' password]''@''](''exchange''|''queue'')['':'' \$route][[''.'']''/''\$configuration]$

Où \$user est le nom de l'utilisateur, \$password son mot de passe. \$route est le nom du routage des messages. Et \$configuration est le chemin du fichier de configuration à utiliser.

Intégration de DDS

DDS est une spécification d'un modèle de distribution où les objets d'envoi et de réception de données sont crées à partir de trois concepts, et trois ensembles de qualités de services. Par exemple, l'objet *DataReader* qui permet de lire des données a besoin des objets suivants : un *DomainParticipant* pour participer à un domaine, un *Subscriber* et un *Topic* pour souscrire à la réception d'un type de donnée.

Les composants de liaison d'interface implémentent les interfaces de type *DataReader* et *DataWriter*. Les composants protocolaires fonctionnent comme les liaisons JMS, c'est à dire qu'ils sont génériques, mais requièrent la participation d'un moteur DDS pour envoyer ou recevoir des données.

48. `http://www.rabbitmq.com/`

Toute donnée DDS doit être décrite depuis un fichier IDL. R-MOM réutilise ce fichier pour décrire l'*Envelope* en respectant la structure de l'API de la sous-section 4.2.6. L'*Envelope* DDS contient par défaut un message de type *any* de l'IDL. L'utilisateur pourra éventuellement spécifier un autre type de message s'il n'est pas intéressé par tous les types de messages pouvant transiter sur le réseau.

La configuration du protocole DDS/R-MOM suit le formalisme suivant :

$''DDS :'' [user[''\!:'' password]''@''][\$type][''\!:'' \$topic][[''.'']''/''\$configuration]$

Où \$user est le nom de l'utilisateur, \$password son mot de passe. \$topic est le nom du topic, \$type est le nom du type de topic. Et \$configuration est le chemin du fichier de configuration à utiliser.

4.4 Conclusion et Résultat

R-MOM est une architecture de mise en œuvre du paradigme MOM dans un système distribué. Il facilite la conception et la maintenance d'un tel système (sous-section 4.3.1) en considérant des besoins spécifiques au domaine d'application aussi bien au niveau de l'API (sous-section 4.2.4) que des données transitant sur le réseau (sous-section 4.2.6). R-MOM assure également des fonctionnalités attendues par des besoins d'adaptation du système avec un assemblage fonctionnel reconfigurable à l'exécution (sous-section 4.2.3), joint à la préservation de contexte d'exécution, et une interopérabilité entre protocoles de communication respectant le paradigme MOM (sous-section 4.2.5).

R-MOM est la meilleure solution de personnalisation de fonctions MOM reconfigurables à l'exécution et interopérable avec tous les intergiciels basé sur le paradigme MOM. Finalement, il propose la méthode de sérialisation la plus flexible et la plus légère par rapport à tous les travaux existants, et le flux d'information est maximisé pendant un changement de technologie de communication.

Ce travail a résulté dans la soumission du papier scientifique [Labéjof et al., 2012b] à la conférence internationale de rang B EDOC2012 [49] qui réunit des industriels et des académiques autour des problématiques sur le domaine des systèmes distribués. Le retour fur l'acceptation au premier workshop SCDI [50], avec une présentation faite le 10 septembre 2012.

L'expérimentation de cette contribution dans un système de systèmes est reportée au chapitre 6, page 109.

49. http ://166.111.71.230/edoc/
50. Service and Cloud based Data Integration : `http://166.111.71.230/edoc/edoc-SCDI-Workshop-CfP-Tentative.html`

Chapitre 5

R-EMS – Gestion de l'environnement pour un système distribué

Sommaire

Cette dernière contribution s'intéresse à un système de gestion d'un environnement d'un système de systèmes (SoS), où il devient possible de manipuler des modèles réflexifs et distribués pour faciliter le contrôle des exigences d'un SoS, en supervisant toute activité extérieur ou intérieur à l'environnement.

5.1 Pourquoi un système d'environnement ?

Tout d'abord, voici une définition d'un environnement de système au sens naturel du terme (non uniquement spécifique à l'informatique) :

Environnement de système :

Ensemble des éléments qui entourent un système et qui contribuent à subvenir à ses besoins.

Les besoins sont de différentes nature. Par exemple, il peut s'agir de la spécification du système, et donc d'une partie de ses exigences. Autrement, il peut s'agir de la nature des opérations faites par le système à haut ou bas niveau.

Les éléments sont propres au domaine d'activité. Par exemple, un système bancaire s'exécute dans un environnement mélangeant des aspects économiques, avec les types de compte en banque, de financement, de crédit, d'assurance, etc. mais il faut aussi considérer des aspects politiques de la banque qui utilise le système. Ces aspects sont rarement modélisés par de tels systèmes puisqu'ils appartiennent à la sphère des activités privées et stratégiques de la banque, et qui par conséquent, n'ont pas pour but d'être partagés avec des systèmes appartenant à d'autres structures privées, et pourtant connectés à la banque. Cependant, ils n'en restent pas moins importants car ce sont eux qui vont orienter la spécification du système bancaire.

Il n'y a donc pas de type d'éléments précis dans un environnement. Tout dépend du domaine d'activité, et des décisions stratégiques.

Une fois la spécification écrite, elle est reprise par les développeurs et les architectes du système, et donc adaptée à des contraintes techniques, et à l'interprétation de ces mêmes personnes. Cette délégation fait qu'il y a une forte probabilité que les contraintes et les interprétations ne respectent pas les idées des personnes qui ont écrit la spécification. La conséquence est que lorsque le système est déployé, il peut y avoir des comportements non attendus par la spécification, et sujets à une évolution dans le meilleur des cas. Dans le pire des cas, il s'agit d'un échec de projet.

Il faut de toute évidence réussir à réduire l'incertitude qui existe entre les besoins exprimés par l'environnement et leur réalisation dans un système final.

Plaçons-nous à présent dans un contexte dynamique. Dans un tel contexte, l'environnement ou le système lui-même pourrait solliciter une modification des exigences et de la spécification par exemple. Or, les solutions actuelles nécessitent la présence de programmeurs en informatique pour réaliser les modifications nécessaires et satisfaire une évolution. Dans le cas d'une modification de l'environnement, il faudra passer par l'écriture d'un document de personnes ne possédant pas d'expérience technique de la vision informatique du système, puis par une interprétation d'experts techniques pour essayer d'appliquer la modification dans le système final.

Ces processus sont longs, rarement formalisés et donc sujets à des erreurs d'interprétation car ils multiplient les décisions et les actions de modification de systèmes.

Pour supporter une méta-description, une modélisation d'un système et lier les deux niveaux de description à leur réalisation dans un contexte dynamique, R-* propose R-EMS, le Système de Gestion d'Environnement, qui est schématisé dans la figure 5.1.

Cette figure est une vue logique d'un système de systèmes qui représente tout ce qui a été écrit précédemment. Cette figure contient une spécification, un système et un environnement contenant plusieurs systèmes. Cette vue est centrée sur un système, où tous les autres qui entretiennent une relation avec ce dernier font partie de son environnement. Ainsi, dans cette vue, le SoS englobe le tout, alors que R-EMS n'englobe que l'environnement, la spécification, les relations de dépendance et une partie du système, afin de mettre en évidence que la relation qu'entretient R-EMS avec un sous-système du SoS se constitue des interactions et de la modélisation de certaines parties du système.

Disposer d'un tel modèle d'environnement permet en théorie de faire du raisonnement sur les capacités du système de systèmes (identification des technologies capable de couvrir des exigences, aide à la décision sur les exigences à modifier pour se rapprocher d'un but, etc.), et de prédire des évolutions ou adaptations à venir avec des tests de simulation. L'ajout d'une synchronisation entre le modèle et les sous-systèmes ouvre la voix à de la supervision et du contrôle sur le système de systèmes.

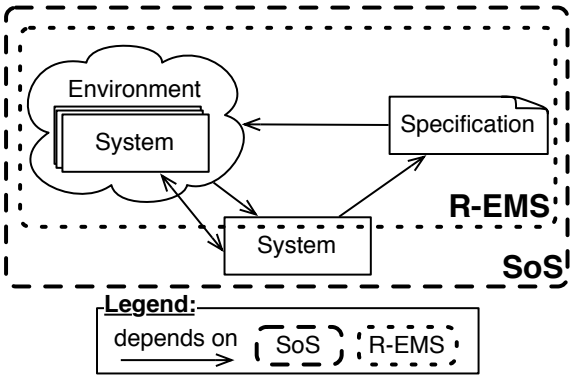


FIGURE 5.1 – **Vue logique de R-EMS dans un système de systèmes** – Environnement, spécification et systèmes

Ainsi, R-EMS souhaite couvrir différents besoins :
– modélisation réflexive et distribuée d'un environnement (concepts et éléments),
– synchronisation et interopérabilité de l'environnement avec les sous-systèmes.

L'utilisation d'un modèle réflexif sert à assurer une cohérence et un formalisme de modélisation de l'environnement, et a fortiori, des spécifications et des exigences systèmes qui doivent pouvoir être écrites par tout type de personne, afin de limiter les erreurs d'interprétation quelque soient les domaines dans lesquels va s'appliquer le système de systèmes.

La synchronisation et l'interopérabilité sont nécessaires dans le sens où l'environnement contribue aux besoins du système, mais l'activité du système peut être amenée à modifier l'environnement.

Dans l'état de l'art, des techniques de gestion de QoS (sous-section 2.7.7, page 50) sur le support de la dimension non-fonctionnelle ont été identifiées, et serviront de base au travail d'application dans un contexte dynamique. [Ortiz and Bordbar, 2008] propose une technique de configuration dynamique liant fortement l'implémentation de QoS avec leur modélisation en jouant avec un PIM et des PSMs propres à l'approche MDE (sous-section 2.3.5, page 23). L'AOP (sous-section 2.3.4, page 23) servant de lien fort entre ces deux types de modèle, il offre la possibilité de tisser des règles de transformation dans les plateformes d'exécution afin de convertir les contrats du PIM vers les PSMs.

L'idée ici est d'étendre ce travail dans un contexte dynamique, et d'assurer la formalisation de préoccupations issues de plusieurs domaines d'activité (économiques, politiques, etc...). Mais également de considérer que procéder de cette manière pourrait enrichir toutes les technologies de communication, voire même les composants fonctionnels, avec des capacités d'introspection. En utilisant cette approche, et en utilisant un modèle transverse pour y configurer l'intégralité du système et de son environnement, le modèle aurait ainsi une vue complète sur l'intégralité des éléments pouvant interagir avec le système.

R-EMS est un système de support d'exigences pour tout système informatique (y compris lui-même). Il entretient l'accès à un modèle d'exigences réflexif, interopérable et compatible avec tout moyen de communication informatique et tout type de configuration, afin d'accompagner tout système dans ses phases de conception, de (re-)configuration, de déploiement et de maintenance.

Si l'intégralité de la configuration est dans ce modèle, les points de coupe AOP pourront y être également disposés, sans avoir à modifier les fichiers de configuration initiaux, et donc d'une manière totalement non-intrusive avec les systèmes déjà existants. Ce PIM étant un modèle pivot, toutes les configurations parleraient le même langage, et des composants de (re-)configuration et de (re-)déploiement aidant, il deviendrait facile de tisser des aspects eux-aussi définis dans ce langage.

La figure 5.2 montre un exemple d'utilisation de R-EMS qui modélise la configuration d'un système à partir de fichiers de configuration écrits dans différents langages (à gauche), complété par des propriétés (*p1* et *p2*). L'exécution de la configuration est ensuite déléguée à des plateformes d'exécution (à droite).

La configuration comprend de haut en bas, un modèle à composant SCA, avec un composant nommé

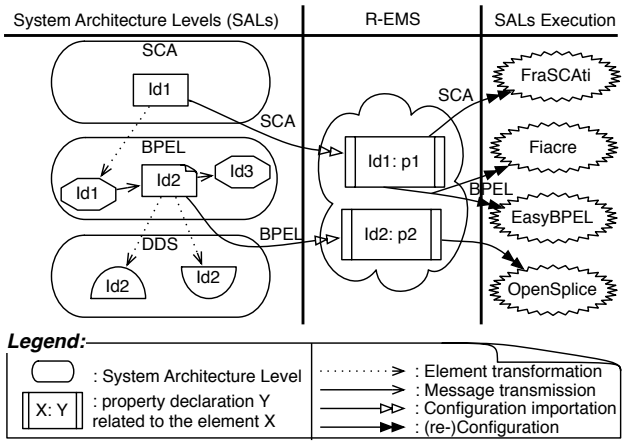


FIGURE 5.2 – **Utilisation de R-EMS pour (re-)configurer et exécuter un système à partir de fichiers de configuration de différentes nature, et d'une unification de leur sémantique dans un modèle commun enrichi de propriétés, vers différentes plateformes d'exécution** – Fichiers de configuration de type SCA, BPEL et DDS, et plateformes d'exécution FraSCAti, Fiacre, EasyBPEL et OpenSplice

Id1, dont le service est configuré par l'activité de même nom dans un processus métier écrit en BPEL. Cette même activité transmet un message *Id2* à une troisième activité, via la dernière configuration DDS, où le message est défini en tant qu'un couple de qualités de services définies sur le sujet et sur l'entité d'envoi ou de réception (voir la section 2.5.6).

L'architecture SCA est réalisée par FraSCAti, l'architecture BPEL est réalisée par EasyBPEL [Dar et al., 2011], les contraintes temporelles sont vérifiées par Fiacre [Fares et al., 2011], et finalement, l'architecture DDS est réalisée par OpenSplice.

Les fichiers sont transmis à R-EMS afin d'unifier la description des différents éléments du système, et de les compléter avec des préoccupations plus globales comme il peut y en avoir avec des SoS (nouvelles exigences politiques et économiques sur le fonctionnement d'un système déjà déployé par exemple). La configuration des plateformes d'exécution par tissage d'aspects est réalisée de manière totalement transparente et non-intrusive pour l'architecture initiale. Dans cet exemple, il est à noter que R-EMS résoud un conflit entre niveaux d'architecture, c'est à dire que le message *Id2* dans l'architecture BPEL est transformé en deux éléments dans l'architecture DDS. En effet, un producteur de message dans DDS a besoin de partager des QoS avec un sujet (voir sous-section 2.5.6, page 32), de ce fait, il convient de se mettre d'accord avec des qualités de services définies dans le BPEL, puis transformées et configurées dans deux éléments DDS. R-EMS résoud ce dilemme en introduisant une seule transformation et un seul élément décrit de manière sémantique. Il identifie l'élément dans l'architecture BPEL, pour modéliser ses propriétés, puis identifier le groupe d'éléments correspondant dans DDS, et mettre à jour les propriétés non-définies au niveau BPEL. Ces informations resteront pertinentes ensuite pour les deux niveaux d'architecture.

Une autre fonctionalité intéressante de l'approche sémantique dans R-EMS, et illustrée dans la figure 5.3, est la possibilité de choisir les technologies à utiliser pour réaliser les exigences d'un système.

L'approche SPL (sous-section 2.3.1, page 21) est fortement sollicitée dans ce cas. Après expression des besoins (1), un moteur de raisonnement peut aider R-EMS à choisir parmi un groupe de technologies ou d'architectures qui viennent avec leur jeux de capacités (mode RPC, support de QoS, etc.) et de caractéristiques (logiciel libre, souvent utilisé dans des projets, réputation, version bêta, etc.) qui trouvent une compatibilité sémantique avec celles du modèle d'environnement. Dans la figure, le choix est porté sur les modèles à composants CCM, OSGi et FraSCAti, qui connaissent chacun le type d'élément T qui est équivalent au type demandé par R-EMS. Puis, R-EMS peut utiliser un algorithme de type minimax

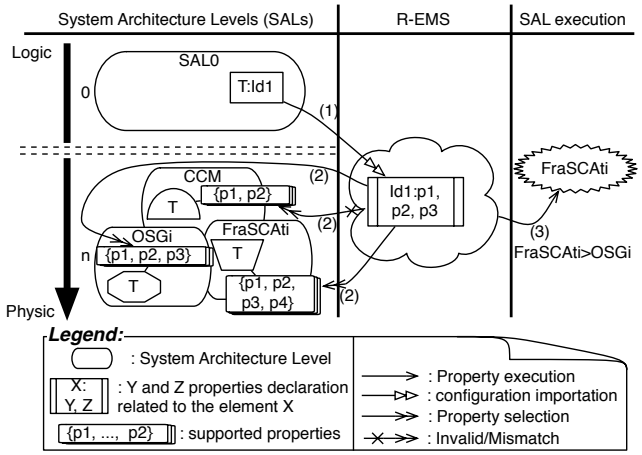


FIGURE 5.3 – **Choix d'un modèle à composnt à utiliser en fonction des fonctions attendues et celles offertes par des modèles à composant existants** – Raisonnement par identification sémentique des fonctions

[Willem, 1996] pour choisir selon des critères d'exigences la technologie qui saura le mieux répondre aux besoins système (2). Dans la figure, CCM est la seule architecture qui ne peut supporter la propriété de type *p3*, et OSGi supporte moins de propriétés que FraSCAti, donc il se présente comme moins apte à répondre à des besoins futurs. Alors par anticipation, le choix se portera plutôt sur le modèle à composant FraSCAti (3).

Finalement, la force de R-EMS est bien d'enrichir toute architecture de capacités de re-configuration, de traçabilité de tout élément du système, d'analyses poussées et de capacités de raisonnement sur le système, quelque soit les types de configuration d'entrée.

Idéalement, *R-EMS* doit répondre aux défis suivants :

1. être interopérable avec tout composant logiciel,
2. fournir un modèle exécutable, réflexif et extensible pour pouvoir s'auto-modifier en fonction des systèmes à accompagner, depuis sa conception jusqu'à sa mort, mais aussi garantir la modélisation de tout élément aussi bien structurel que conceptuel,
3. être distribué pour répondre aux problèmes techniques de distribution de données dans de très grands systèmes, et assurer des domaines de responsabilité logiques et physiques,
4. être transactionnel pour assurer un maximum de cohérence lors de chaque lecture, modification ou exécution du modèle,
5. être protégé des accès concurrents,
6. être versionné pour consolider la gestion d'évolution de systèmes et de dépendances inter-système,
7. être formalisé pour justifier soit sa propre cohérence, soit la cohérence de dépendance entre exigences systèmes,
8. supporter un modèle sémantique pour faciliter le choix des participants à l'environnement en fonction de leurs capacités (approche SPL de la sous-section 2.3.1, page 21).

L'aspect interopérable sera assuré par R-* à travers les paradigmes de communication RPC et MOM. Soient respectivement remplis par les solutions FraSCAti de l'équipe projet ADAM/Inria (sous-section 1.4.1, page 6) et R-MOM (chapitre 4, page 67).

La figure 5.4 représente l'architecture globale du système R-EMS. Au coeur de l'architecture se trouve le moteur d'exécution des modèles d'environnement (en haut à droite). Le moteur requiert des opérations de transformation de modèles (interface *Model transformation*) pour assurer des interprétations du modèle

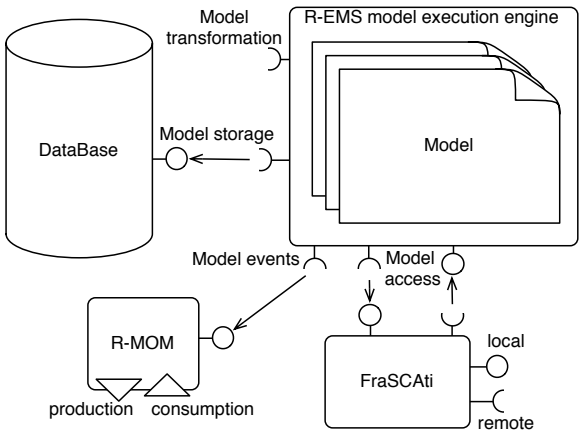


FIGURE 5.4 – **Architecture R-EMS** – Exécution, transformation, persistance et communication autour des modèles d'environnement

par différents langages. Il nécessite aussi un moyen d'accès à une base de données (interface *Model storage*) pour sauvegarder les différents modèles, mais aussi un moyen interopérable de produire et consommer des évènements (composant R-MOM et interface *Model events*). Finalement, il offre une interface d'accès au modèle (interface *local* du composant FraSCAti), et nécessite également une autre interface pour pouvoir communiquer avec d'autres moteur d'exécution de modèles d'environnement (interface *remote* du composant FraSCAti).

5.2 Un modèle d'environnement

L'objectif est ici de pouvoir modéliser un environnement englobant tout système. Un tel modèle doit permettre de définir tout type d'élément, afin de programmer du comportement et du raisonnement sur l'environnement visé.

Pour ce faire, il faut utiliser un modèle de programmation d'environnement qui soit très haut niveau, où la cohérence du modèle prend considérablement le pas sur la vitesse d'exécution des algorithmes qui peuvent y être écris, et qui seront de toute manière exécutés par des technologies tierces. Et la structure des éléments doit pouvoir être compatible avec une représentation pivot de tous les fichiers de configuration possibles.

L'approche MDE est une solution pour concevoir un tel modèle. Les architectures PIM/PSM permettent d'introduire une vision très haut niveau d'un environnement (PIM) dont des parties seront réalisées ou affinées par des ressources dédiées à leur exécution et à leur vérification et définies dans des PSMs.

Dans l'approche "environnementale" proposée ici, les systèmes distribués existants seront compris comme des PIMs ou PSMs fils par rapport au PIM environnemental. Afin de garantir une communication entre ces PIMs et PSMs fils, il convient de pouvoir définir une description sémantique au niveau du PIM environnemental. Ce dernier assurera de manière cohérente toute interaction entre les sous-systèmes. Ou facilitera l'acquisition d'informations depuis le PIM parent vers un PIM/PSM fils (tout sous-système n'aura pas besoin de connaitre l'intégralité du modèle environnemental pour contribuer dans sa réalisation).

La nature évolutive d'un environnement doit le préparer à se décrire et à se modifier. Mais à quel point doit-il pouvoir se décrire, et se modifier ?

La description du modèle d'environnement doit permettre de s'auto-décrire, c'est à dire, assurer l'accès à sa méta-description, pour pouvoir le parcourir de manière générique au niveau du méta-modèle, ou de manière spécifique au niveau du modèle.

Toutes les propriétés identifiées respectent le fait que l'on parle ici d'un modèle de programmation d'environnement qui vise une vitesse d'exécution suffisamment rapide pour être exploitable. Donc certaines propriétés introduisent volontairement des mécanismes complexes à résoudre. D'un point de vue langage, le modèle correspondra aux langages haut-niveau (sous-section 2.4.2, page 25) dans le sens où il ne doit pas gérer les contraintes mémoires au niveau PIM, mais cela ne l'empêche pas de les modéliser pour affecter les PSMs ciblés.

Pour assurer la capacité d'introspection, deux méta-modèles sont proposés, un statique et un dynamique. Le méta-modèle statique comprend tous les axiomes de modélisation, il n'est donc pas considéré comme modifiable, et ne doit servir qu'à créer un modèle. Le méta-modèle dynamique sera défini dans le modèle, et proposera tous les axiomes du méta-modèle statique, plus d'autres méta-informations qui seront spécifiques au modèle, comme les types d'objets de la programmation OOP, ou les types de relations sémantiques. Ainsi, il devient possible d'analyser le modèle avec l'aide des deux méta-modèles. L'intérêt d'avoir un méta-modèle dynamique, est de pouvoir se passer du méta-modèle statique pour le parcours et la méta-définition d'un modèle durant son exécution, et par la même occasion, enrichir le comportement du modèle durant son exécution.

Le modèle d'environnement doit pouvoir modifier tous ses éléments, et surtout sa méta-description. Tout doit être modifiable, mais pas nécessairement appliqué, c'est pourquoi la propriété de responsabilité doit être introduise. Cette responsabilité permet de déléguer à un système le soin de gérer les droits d'accès à un élément de l'environnement. Par ailleurs, en complément de cette capacité de modification, il est conseillé d'utiliser une notion de référentiel pour distinguer les modifications appliquées. Ce référentiel dans un environnement physique à l'échelle humaine pourrait être le couple de dimensions d'espace et de temps. En ingénierie logicielle, le référentiel le plus utilisé est une information de version. Le format le plus connu en occident est composé de trois entiers significatif qui représentent les numéros de version majeur, mineur et de révision. En langage Java ou C#, on parlera de librairies.

Ensuite, le modèle doit être extensible pour pouvoir y définir tous les concepts que l'on souhaite. Dans ce cas, le typage est une propriété appartenant à la méta-description d'un programme, mais doit rester modifiable. Chaque élément doit connaitre une description distribuée, à la manière des langages C++ ou C#.

Finalement, le modèle d'environnement doit pouvoir identifier quels sont les paradigmes de programmation employés pour pouvoir les isoler et les fournir à des tiers intéressés uniquement par un ensemble de paradigmes. Et dans le sens inverse, la prise en charge des paradigmes doit faciliter l'importation de fichier de code pour enrichir l'environnement. Par exemple, il doit pouvoir importer un fichier de configuration BPEL ou SCA (sous-section 2.6.5, page 45).

Dans le but de faciliter le suivi de la modification du modèle, il convient d'adopter la programmation orientée évènement...

5.3 Méta-modèle d'environnement statique – R-EM3

R-EM3 [51] est le nom du Méta-Modèle de Gestion d'Environnement qui va servir à définir les concepts génériques permettant de programmer l'environnement. Les Modèles de Gestion d'environnement seront appelés R-EM2 [52].

Contrairement à des approches comme ALF (sous-section 2.4.3, page 27), le méta-modèle et les modèles doivent être agnostiques d'une quelconque représentation visuelle (Textuelle, graphique ou toute autre spécifique à un domaine), afin de laisser libre cours à la manière dont les utilisateurs vont interagir avec eux, mais en respectant un minimum les règles de cohérence de modélisation pour satisfaire un formalisme, et minimiser les défauts d'interprétation.

Tout d'abord R-EM2 comprend un élément racine qui comprend une séquence finie d'éléments : *Root*.

Le but est qu'une fois une racine donnée à un moteur d'exécution d'environnement, elle se retrouve exécutée, à la manière d'un bloc d'instruction.

51. Reflective-Environment Management Meta-Model
52. Reflective-Environment Management Model

5.3.1 Élément : ensemble distribué

Concept de base du méta-modèle, l'élément ou *Element*, est perçu comme un ensemble d'éléments afin de permettre de modifier sa structure interne durant l'exécution du système R-EMS.

De plus, l'aspect programmable du modèle comprend un élément comme une expression, c'est à dire que tout élément peut-être évalué. Dans ce cas, tout élément possède la méthode *evaluate() : Element*. Par exemple, l'évaluation d'une valeur booléenne sera un booléen, l'évaluation d'un type sera son enregistrement dans le modèle, alors que l'évaluation d'une instruction sera son exécution, avec une valeur de retour possible.

Un élément comprend une description structurelle et une description sémantique qui peuvent être distribuées dans tout le système, à la manière du C# qui offre la possibilité de définir une classe dans plusieurs fichiers pour une même librairie, sauf qu'ici, la librairie est l'intégralité de l'environnement. Cette spécificité oblige l'élément à maintenir une table des autres parties de l'environnement qui contiennent la description, et à ne surtout pas rapatrier toutes les informations car si elles ont été co-modélisées, c'est pour rester spécifique à un moteur d'exécution.

Description structurelle, notion d'héritage et de compatibilité d'ensembles

La description structurelle entretient une relation de référence/composition avec d'autres éléments, qui peuvent être tout élément de l'environnement. Dans le sens inverse de cette relation, tout élément peut-être typé (relation de composition) ou référencé dans un ensemble, de manière à y participer.

Une relation de type permet de lier fortement un élément à une ou plusieurs méta-descriptions par le biais du contenu. La relation de compatibilité enrichit la relation d'héritage utilisée dans les OOPs connus par respect pour la description structurelle, et même le principe de composition de classe mixée de Scala [Odersky et al., 2004] ou des traits SmallTalk (sous-section 2.4.5).

Compatibilité d'éléments :

Soit A et B deux ensembles. A est dit compatible avec B si et seulement si $B \subseteq A$.

En plus de cette équivalence remarquable, la relation de compatibilité voit B comme un ensemble de méta-données, et non comme un type identifié.

Dans un modèle d'environnement, il faut voir la relation de compatibilité de la description structurelle avec une cohérence sur le contenu, et non sur le nom d'un ensemble. Par exemple, l'opération $(\mathbb{R} \setminus 0)$ est la spécialisation de l'ensemble des réels sans l'élément 0. Un autre exemple avec des interfaces Java : soit les interfaces *java.lang.Runnable* et *java.lang.Iterable* comprenant respectivement les méthodes *run()* et *iterator()*, alors on a :

$java.lang.Runnable \bigcup java.lang.Iterator = \{run(), iterator()\} = T$.

T correspond à un ensemble comprenant les méthodes *run()* et *iterator()* des interfaces. Ce nouvel ensemble est vu comme compatible avec les deux interfaces du fait que les méthodes sont les mêmes, alors que l'ensemble $T \setminus \{run\}$ n'est plus compatible avec *java.lang.Runnable*. Dans cette logique, tout élément doit être capable de spécifier précisément la provenance locale ou distante de son contenu. Au final, un élément est un ensemble qui se compose de spécificités qui sont propres à l'élément sur la base de l'union de ses relations de compatibilité et de ses relations d'héritage.

Cet exemple avec les interfaces Java met en évidence que la logique de la relation d'héritage OOP est respectée du point de vue d'un développeur Java, mais qu'elle peut-être également étendue à la notion de correspondance. Un développeur Java pourra enrichir le modèle d'environnement avec sa logique, sans nécessairement devoir apprendre une nouvelle logique de programmation.

En ingénierie logicielle, cette correspondance d'ensemble couplée à l'héritage est très intéressante en terme de re-factorisation de code [Fowler and Beck, 1999] car il permet par exemple de partager la description d'un élément à différents endroits pour limiter le nombre ou la complexité de classes, ou d'éviter les ambiguïtés liées par exemple au problème du diamant [53] bien connu du langage C++, et qui

53. Lors d'un héritage de classe proposant des signatures de méthodes compatibles mais pas les mêmes implémentation, c'est le compilateur qui choisit la méthode à utiliser, et donc il y a un risque d'incohérence de la méthode choisie.

est résolu ici par l'utilisateur avec la spécification explicite des éléments à conserver. Dans le cas d'un conflit, par défaut (comportement personalisable), le conflit doit être résolu, car le moteur est incapable de savoir par lui-même quel est l'élément à utiliser.

Le langage Scala [Odersky et al., 2004] a proposé une alternative à ces relations de compatibilité avec les *traits*, mais les possibilités restent moins riches que dans notre cas, où nous pouvons soustraire des éléments de description structurelle pour constituer un ensemble final de valeur dans un type d'élément. De plus, un conflit est résolut par l'ordre de d'utilisation des traits, le dernier devenant plus important que les précédants.

Les traits du langage SmallTalk (sous-section 2.4.5) proposent une spécification aussi rîche et basé sur la même résolution explicite de conflit. Mais la spécification ne se base pas sur des opérations d'ensemble. Et il faut spécifier un à un les éléments à intégrer qui proviennent nécessairement d'un type d'élément. Il ne peut s'agir d'une opération définie indépendemment d'un type.

Description sémantique et raisonnement

La description sémantique choisie répond au nom d'ontologie.

L'ontologie permet de monter encore plus en abstraction par rapport à la programmation structurelle de l'OOP. L'objectif est d'ajouter des informations de sens, ou sémantiques, autour d'éléments d'un programme. Un exemple simple est la possibilité de donner le même sens à deux objets qui n'ont pas la même valeur structurelle. Par exemple le mot "tree" en anglais a le même sens que le mot "arbre" en français, alors que l'orthographe n'est pas la même. Ainsi, une ontologie permet de caractériser des relations non structurelles entre classes, propriétés et individuels qui correspondent respectivement aux types, champs et instances du paradigme OOP. Certaines relations de base sont proposées pour pouvoir identifier des équivalences ou des participations par exemple. Et il est également possible d'enrichir leur comportement avec des propriétés telles que l'inversion, la symétrie, ou encore la transitivité.

Pour des raisons de respect d'une standardisation, la logique de conception d'ontologie utilisée par le système R-EMS s'inspire du travail de spécification d'un langage d'ontologie pour le Web OWL proposée par le W3C [McGuiness et al., 2004].

Cette spécification est explicite sur les relations qu'entretiennent les concepts entre eux, mais ne garantit en rien que les moteurs de raisonnement respecteront les définitions données, à cause d'algorithmes pour raisonner sur de tels modèles qui ne sont pas spécifiés, car la plupart appartiennent à la classe des problèmes NPC. Dans le cas de tels problèmes, il convient à chaque moteur d'apporter des algorithmes d'approximation [Pan and Thomas, 2007].

La description sémantique comprend par défaut deux types de relations. Soient l'équivalence, et la participation. Les autres descriptions sémantiques sont définies dans le méta-modèle dynamique, en sous-section 5.4.6.

L'équivalence et la participation peuvent être couplées à des instructions de transformation, qui permettent ainsi de convertir des éléments en d'autres éléments à l'aide de règle de transformations sur leurs propriétés structurelles.

Ce patron de programmation est appelé transtypage sémantique (ou *semantic cast* en anglais). Il étendra l'opération de transtypage structurel (*cast* en anglais) bien connu des langages OOP. Les raisonnements attendus par un tel patron sont typiquement sources de mécanismes de résolution d'algorithmes de classe NPC discutés dans la section 5.2, page 88, et donc à utiliser avec beaucoup de précautions, en dehors de besoins de raisonnement.

5.3.2 Éléments primitifs

Trois éléments primitifs sont proposés afin de pouvoir modéliser toute structure d'ensemble, soient la logique booléenne, la notion de quantité et le mot. Ces éléments respectent des domaines philosophiques ou linguistiques propres au modèle de pensée de l'homme pour décrire une idée.

La logique booléenne est introduite avec le type *Boolean*, qui est accompagné des opérations de conjonction, de disjonction et de négation.

La notion de quantité est gérée avec le type *Number*, représenté par un couple de nombres réels, correspondant respectivement aux parties réelle et imaginaire de la quantité.

Le mot est l'élément qui ouvre la voie au langage. Sa représentation informatique sera une chaîne de caractère, ou *String* en anglais.

5.3.3 Domaine et importation

Le domaine et l'importation permettent de définir les zones logiques de définition des éléments de l'environnement en s'abstrayant de l'endroit physique où ils sont définis. Ils ont pour but de faciliter les liaisons entre plateformes d'environnement, et de configuration de systèmes.

Domaine

Le domaine, ou *Domain* permet de spécifier des zones logiques de description d'éléments. Ainsi, tout élément peut être présenté dans une arborescence de domaines logiques, en plus de pouvoir appartenir à un autre élément par relation de composition.

Le domaine est constitué d'un nom, et accessoirement d'une URI pour spécifier le moyen d'y accéder depuis toute plateforme d'exécution. Dans le contexte de l'environnement distribué, un domaine pourra ainsi connaître plusieurs URI, en fonction des parties de description du domaine, et de la plateforme de leur réalisation.

Un domaine définit aura pour effet de charger une description de tous les éléments distribués appartenant au même domaine, mais leur exécution reste sous la responsabilité des plateformes hébergeant les dits éléments.

Importation

L'importation ou *Import* permet de spécifier à l'aide d'une URI, ou d'un chemin vers un domaine, une ou plusieurs ressources à inclure dans le langage. Ces ressources peuvent-être des domaines, ou tout fichier de configuration système. Il peut-être associé à un nom pour en faciliter sa référence dans le modèle.

5.3.4 Élement d'instantiation et invocation

Un élément d'instantiation, ou *InstantiableElement*, est un élément qui étend le principe d'accès statique/dynamique à une ressource en gérant une méta-description et une ou plusieurs instances de cette méta-description, afin d'offrir un patron d'exécution complexe concernant le partage de ressource. De plus, l'accès partagé ou exclusif est également géré pour fournir des instances déjà utilisées, ou attendre d'avoir une instance non utilisée, ou encore d'en créer volontairement une dans le cas d'un accès exclusif.

Pour ce faire, il prend en paramètre le nombre maximum d'instances possibles, et est capable de savoir si une ressource est utilisée ou non, afin de fournir des instances partagées ou exclusives au besoin. L'accès statique à une ressource correspond à une seule instantiation possible de cet élément. Alors que le mode dynamique correspond à un nombre indéterminé d'instantiations.

L'invocation de cet élément peut-être paramétrée par l'instantiation visée (si elle existe ou peut-être créée), et/ou une instance non-utilisée. Ce patron est essentiel pour gérer au mieux des instructions partagées, qui à la manière d'un groupe de fils d'exécution (*thread pool* en anglais), peut organiser les instances en fonction du nombre de ressources intéressées.

Si l'élément d'instantiation comprend des paramètres d'entrée, et que l'invocation ne propose pas une valeur pour chacun d'eux, alors le résultat de l'invocation est l'invocation avec les paramètres non initialisés, qui pourront être complétés par la suite.

La forte liaison entre l'élément d'instantiation et les instances garantit une relation de composition entre les deux, tant que l'élément existe. S'il venait à être supprimé, alors le traitement par défaut (personnalisable) est de répéter cette suppression sur toutes les instances, qui accepteront ou non d'être supprimées.

D'autres spécificités pourront compléter le comportement de cet élément à l'aide des décorations du modèle. Par exemple, une décoration peut spécifier une fonction de priorité sur les threads.

5.3.5 Variable, Opération et Type

Les concepts de variable opération et type étendent le concept d'élément d'instantiation, mais en proposant plus ou moins de méta-données ou paramètres liés aux instantiations.

D'un point de vue conceptuel, la variable offre l'accès à un élément. L'opération spécialise l'accès à l'élément avec du traitement intermédiaire. Et le type spécialise l'accès et le traitement en ajoutant du comportement à l'élément retourné.

Variable

Une variable, ou *Var*, est un ensemble qui ne contient qu'un seul élément qu'on appel valeur, ou *Value* en anglais, et utilise optionnellement une valeur d'initialisation. La variable utilise une méta-description qui doit être compatible avec celle de la *Value*. Une variable est partagée par défaut entre toutes les ressources qui ont accès à l'élément parent, c'est pourquoi le nombre d'instantiations possible est fixé à 1 par défaut, et l'accès est partagé.

Opération

L'opération, ou *Operation*, hérite de la variable en considérant des paramètres d'entrée qui vont permettre d'enrichir le contexte d'exécution. Ces paramètres héritent des variables et offrent en plus une cardinalité.

Accessoirement, une méthode peut utiliser des types paramétrés (similaires aux types paramétrés Java) pour spécialiser les types des paramètres et du résultat en bornant les possibilités de typage, et il est également possible d'expliciter une liste d'exceptions pouvant être lancées lors de l'exécution de la méthode.

Type

Un type, ou *Type*, hérite de l'opération (tout comme le langage Scala [Odersky et al., 2004]) mais la valeur est utilisée pour affiner la description des méta-données de cette même valeur. La valeur finale se traduit par l'union des méta-données, et de la valeur. La valeur par défaut permet d'affecter une valeur à une variable par défaut, sinon, l'initialisation vaut *Nil*. Par exemple, la valeur par défaut d'un nombre est 0. Si on définit une variable de type nombre, et qu'elle n'est pas initialisée, alors la valeur par défaut sera prise en compte, c'est à dire que cette valeur vaudra 0, et non *Nil*.

Tout comme le paradigme OOP, un type propose le concept de constructeur afin de proposer plusieurs moyens d'instancier des valeurs.

5.3.6 Évènements du modèle

Le modèle étant exécutable et observé, il doit pouvoir produire des évènements après chaque action effectuée sur ses éléments.

Un évènement est accompagné d'un contexte d'exécution qui est partagé entre toutes les entités qui l'écoutent, et comporte des paramètres d'émission de l'évènement, comme l'élément source de l'émission.

Même si ce mécanisme est surtout défini dans le méta-modèle dynamique, neuf types d'évènements font partie du méta-modèle statique. Le modèle d'évènement est extensible dans le sens où tous les évènements sont extensibles en fonction des besoins applicatifs désirés, ou de manière personnalisée, il est toujours possible de créer un nouveau type d'évènement héritant du type *Event*.

L'élément observable peut spécifier s'il souhaite que les évènements qu'il produit sont bloquants ou non, c'est à dire s'il doit attendre que tous les observateurs aient traité l'évènement ou non. L'émission bloquante facilite la tâche des observateurs qui peuvent ainsi arrêter le traitement de l'élément observé, en appelant la méthode inverse de l'émission de l'évènement, avec toujours l'évènement passé en paramètre. Ainsi, l'observé pourra identifier l'évènement en question, et arrêter son traitement, puis détruire l'évènement.

TABLE 5.1 – Évènements par défaut pris en charge par le méta-modèle statique. Tous prennent au moins en paramètre l'élément source de l'émission de l'évènement s'il existe dans le modèle.

Type	Paramètres	Déclenchement
INITIALIZED	-	initialisation d'un élément
MODIFYING	ancienne/nouvelle valeur	modification d'un élément
MODIFIED	ancienne/nouvelle valeur	élément modifié
MOVING	anciens/nouveaux ensemble et indexes	changement d'ensemble
MOVED	anciens/nouveaux ensemble et indexes	élément changé d'ensemble
EXECUTING	invocation	invocation d'une instruction
EXECUTED	invocation et résultat	instruction invoquée
DELETING	-	suppression d'un élément
DELETED	-	élément supprimé

5.3.7 Instructions

En dehors des nombreuses possibilités de définition du modèle, il convient de profiter d'instructions, ou *Statement*s afin de pouvoir programmer l'environnement. Ces instructions inscrites dans le modèle sont réflexives, c'est à dire qu'elles s'auto-décrivent et peuvent s'auto-modifier.

Cette capacité réflexive du code ouvre de nombreuses perspectives, notamment concernant une analyse fine et dynamique du code, ou une adaptation des algorithmes en fonction des capacités de différentes ressources matérielles de moteurs d'exécution qui restent indépendantes des modèles. Sinon, le simple fait de pouvoir analyser le code permet d'assurer avant la génération de code pour les plateformes d'exécution, que les algorithmes utilisés respectent bien les spécifications système ou des normes de sécurité (notamment pour les systèmes critiques), qui sont actuellement applicables uniquement pour les langages C et Ada.

Toute instruction est vue comme une fonction, et peut donc potentiellement retourner un résultat après exécution.

Les instructions servent principalement dans les implémentations de méthode, mais rien n'empêche de les écrire dans la valeur par défaut d'un paramètre, ou dans la spécification d'un type de résultat ou de paramètre.

Lorsqu'une instruction est lue par le modèle, un responsable d'exécution est désigné pour l'exécuter.

Douze types d'instruction sont proposées, soient le *Block*, le *Lock*, l'*Assignment*, l'*Instance*, la *Reference*, la *Description*, le *Jump*, le *Leave*, le *BackTo*, le quadruplet *Throw/Try/Catch/Finally*, le duo *fireEvent/catchEvent*, et le *Return*.

Block

Ensemble d'instructions et de variables. L'ordre d'exécution des sous-instructions est séquentiel par défaut, le modèle peut apporter des méta-types pour traiter chaque instruction d'un *Block* d'une manière asynchrone comme c'est le cas dans le langage ALF (sous-section 2.4.3, page 27) ou Go [54].

Lock

Offre un accès exclusif à une instruction. Il est également possible de spécialiser la portée de cet accès exclusif à un contexte d'exécution particulier avec l'aide d'un élément issu de ce contexte, et passé en paramètre du *Lock*.

Assignment

Change la valeur d'une variable. Prend en paramètre la variable et la nouvelle valeur.

54. `http://golang.org/`

Instance

Extension de l'instruction *this* Java, le contexte permet d'accéder au premier élément instancié parent de cette instruction qui peut être une opération.

Reference

Accès à un élément. Il prend en paramètre un chemin, soit relatif à l'endroit où est utilisée cette instruction, soit absolu et résolu par une URI.

Description

Permet à un élément d'accéder à sa méta-description. Équivalent de la méthode *getClass() : Class*" d'un objet Java.

Cast

Extension du transtypage OOP, avec le support des transformations entre objets liés sémantiquement par les relations d'équivalence, ou de participation.

Jump

Similaire à l'instruction *JUMP* d'un processeur.

Leave

Extension de l'instruction *break* Java. Permet de sortir d'une séquences d'instructions. Les paramètres optionnels sont l'instruction à terminer, et un résultat.

BackTo

Extension de l'instruction *continue* Java. Interrompt une séquence d'instruction, et ré-exécute une instruction passée.

Throw/Try/Catch/Finally

Même logique que la gestion des exceptions en C#, sauf que le *Catch* prend en paramètre un test sur des possibles exceptions à capturer.

fireEvent/catchEvent

Instructions qui permettent respectivement de produire ou de consommer un évènement.

Return

Interrompt l'exécution d'une opération, avec éventuellement une valeur.

5.4 Modèle, méta-modèle dynamique et langage textuel – R-EM2 et R-EML

Le modèle R-EM2 doit définir par défaut l'intégralité du méta-modèle statique avec les éléments de type *Type* et *Operation*. Par exemple, les types *Element* et *Type* du modèle sont définis à l'aide du *Type* du méta-modèle statique.

La suite de cette sous-section spécifie le langage R-EML et d'autres éléments du méta-modèle dynamique jugés nécessaires dans la volonté de disposer d'un modèle multi-domaine d'activités.

5.4.1 DSL textuel pour la gestion d'un environnement – R-EML

R-EML est le langage dédié, ou DSL textuel (sous-section 2.3.5, page 23), et proposé par cette thèse pour la gestion de l'environnement d'un système.

R-EML sert à simplifier un grand nombre de patrons de programmation, tout en réduisant leur écriture. La tâche de construction de ces patrons pourra être déléguée aux moteurs d'exécution d'environnements.

La taille et la complexité du code sont des éléments importants à réduire dans un contexte où il est plus ingénieux de laisser les différents plateformes d'exécution d'environnement manipuler les informations, et raisonner sur les transformations de modèles, plutôt que de solliciter les programmeurs à réfléchir aux conflits entre modèles ou ne pas réduire un maximum la taille des flux de données sur le réseau.

Afin de respecter la philosophie de l'interopérabilité (sous-section 2.2.3, page 18), il est nécessaire de s'appuyer sur des langages de programmation connus pour faciliter le travail d'acquisition de connaissance de ce nouveau langage, et voir comment ajouter les nouveaux paradigmes de programmation et fonctionnalités de manière transparente pour un développeur débutant avec ce langage. Le temps d'apprentissage est l'un des plus coûteux en entreprise, donc il ne faut pas le négliger. Ainsi, un développeur aguerri dans un autre langage devrait pouvoir facilement développer dans son propre langage sans devoir trop retoucher à son code pour l'intégrer dans le langage de gestion d'environnement.

Pour ce faire, le langage propose un ensemble d'éléments syntaxiques qui sont inspirés des langages OOP les plus utilisés, tels que le Java, le C# ou encore le Scala.

Le formalisme utilisé pour les décrire respecte la grammaire BNF étendue [Scowen, 1998], avec les règles terminales suivantes :

Basic rules :

$letter = "a" .. "z" \mid "A" .. "Z";$
$digit = "0" .. "9";$
$integer = digit+$
$decimal = ["-"]\ integer\ ["."\ integer];$
$character = (letter \mid "+" \mid "-" \mid "/" \mid "*" \mid "?" \mid "!" \mid "=" \mid "\%" \mid "\wedge" \mid "'" \mid "\&" \mid "<" \mid ">" \mid "|" \mid "\_" \mid "@" \mid "\sim" \mid "\$" \mid "£");$
$identifier = character\ (character \mid digit)*;$
$absolute\_identifier = identifier\ ("."\ absolute\_identifier)*;$
$cardinality = "*"\ (*\ 0..\infty\ *) \mid "+"\ (*\ 1..\infty\ *) \mid element\ [".."\ [element]]\ (*\ custom\ bounds\ *);$
$comments = ("/*"\ .*\ "*/") \mid ("//"\ .*\ "\backslash n");$

Le langage utilise le retour à la ligne comme séparateur d'instructions dans des blocs d'instructions, ou *Block*, rendant le point virgule optionnel, mais n'introduisant pas d'ambiguïté supplémentaire. Ce type de séparation a été formellement accepté pour être communément utilisés dans des langages comme OCaml [Rémy, 2002], Scala [Odersky et al., 2004], ou encore Go [55].

Les parenthèses, ou *parentheses*, ouvrantes et fermantes servent à exécuter une instruction (bloc d'instruction, ou méthode par exemple), avec possibilité d'y insérer un ensemble de paramètres. Ou à forcer l'ordre d'exécution de plusieurs instructions pour éviter toute ambiguité.

Block and parenthesis rules :

$parentheses = "("\ element\ ")";$
$block = "\{"\ (element\ ("\backslash n" \mid ";")+)*\ "\}";$

55. `http://golang.org/`

Block and parenthesis code examples :

```
/* code example with one block and two empty blocks separated with semi-colon and new line */
// one line comment
{ (2) };{}
({})
```

La nature extensible du modèle permet d'interpréter les méthodes comme des nouveaux mots du langage. L'idée est de reprendre le principe des opérateurs Scala ou OCaml pour ne pas avoir à introduire nécessairement un caractère de séparation [56] entre une instance et l'une de ses méthodes par exemple. Par ailleurs, cette extensibilité oblige le langage à considérer un ordonnancement préfixé ou postfixé (par défaut) par opérateur :

Soit un triplet d'instructions (a, b, c), et un triplet d'opérateurs $(., *, \alpha)$ respectivement d'ordres préfixé, postfixé et quelconque, alors on a : $a\ .\ b\ \alpha\ c \Leftrightarrow (a\ .\ b)\ \alpha\ c$ et $a * b\ \alpha\ c \Leftrightarrow a * (b\ \alpha\ c)$.

Par exemple, en supposant que les opérateurs d'addition et de multiplication sont tous les deux postfixés (ordre par défaut), alors l'instruction $2 * 2 + 2 \Leftrightarrow 2 * (2 + 2) = 8$ est vrai dans le modèle, alors que dans la logique mathématique on a l'équivalence suivante : $2 * 2 + 2 = (2 * 2) + 2 = 6$. Cela n'a pas de sens dans cette représentation, mais ce modèle n'a pas vocation à garantir une cohérence dans une vue particulière. Pour lui, les parenthèses implicites se trouvent bien autour de l'addition car c'est la dernière méthode qui apparaît dans un ordre postfixé, et donc prioritaire sur la multiplication.

Les niveaux de description structurelle et sémantique d'un élément sont respectivement accessibles avec les symboles " : " et " :: ". " : " représente l'accès à la description structurelle, alors que " :: " correspond à la relation d'équivalence sémantique. Dans la même logique, la relation de sous-classe sera traduite par " <: ", la relation de super classe par " :< ", la relation de participation par " <:: ", et le moulage sémantique par " ::= ".

5.4.2 Types primitifs

Six types primitifs sont introduits dans le langage pour des raisons de facilité d'écriture, soit les booléens, les nombres, les textes, les intervalles, les valeurs *Nil* et *Undefined*.

Le booléen, ou *Boolean*, respecte la logique binaire associée. Les valeurs possibles sont 'true' et 'false'.

Un nombre, ou *Number*, est soit composé d'une partie réelle et d'une partie imaginaire, soit l'infinie.

Un texte, ou *Text*, est un ensemble ordonné de caractères. La valeur dans le langage est entouré par des simples ou double apostrophes.

Un intervalle, ou *Interval*, est un ensemble de valeurs bornées ou non bornées. Il est utilisé pour spécifier un ensemble ou la cardinalité d'un paramètre de méthode.

Le *Nil* et le *Undefined* correspondent respectivement à une valeur d'élément inexistant (non alloué/instancié au sens objet), et à une non définition d'un élément. Le *Undefined* sert notamment à spécifier qu'une méthode ne comprend pas d'implémentation dans le modèle.

Primitive element rules :

$primitive = boolean \mid text \mid number \mid interval \mid nil \mid "\_";$
$boolean = "true" \mid "false";$
$number = decimal \mid ([decimal]\ i) \mid ([" - "]\ "infinite");$
$string = ('"' (. - '"' \mid '\"') * '"');$
$nil = "Nil";$
$undefined = "Undef";$
$interval = (["[" \mid "]"]\ element\ ";"\ element\ ["[" \mid "]"]);$

56. Qui est lui même une méthode du langage qui résout la relation d'appartenance entre un ensemble et son contenu.

Primitive element code examples :

```
true ; false // true and false boolean value
2 ;-2i ;Nil ;Undef // real, imaginary, Nil and undefined values
]2 ;2+3] // interval from 2 excluded to 5 included
```

5.4.3 Domaines et importations

Les domaines et importations du méta-modèle dynamique respectent la sémantique du méta-modèle statique, mais le langage facilite leur définition.

Les définitions de domaine et de sous-domaine sont confondus, et facilitées par une écriture possible dans une seule expression, où le chemin depuis un domaine parent et un sous-domaine est résolu à l'aide d'un charactère d'accession au contenu de l'ensemble. De plus, il est possible de décrire un domaine de manière structurelle, ou absolue, à l'image des *packages* Scala.

La définition de plusieurs importations est facilitée en spécifiant un import et plusieurs valeurs possibles, avec la possibilité de nommer les domaines ou importations pour éviter les ambiguïtés entre éléments de même nom, mais appartenant à des espaces de définition différents.

Domain and import rules :

$domain = \text{"}domain\text{"}\ identifier\ (\text{"."}\ identifier) *\ [\text{"}uri = \text{"}\ string]\ [block];$
$import = \text{"}import\text{"}\ [identifier\ \text{" : "}]\ (string\ (* configuration\ file\ *) \mid absolute\_identifier)$
$(\text{","}\ [identifier\ \text{" : "}]\ (string \mid absolute\_identifier)) *\ (* multi\ import\ definition\ *);$

Domain and import code examples :

```
// global import of a "zero" domain
import zero
// sub-domain "two", son of "one"
domain one.two{
/* importation of composite and bpel files, named respectively "comp" and "proc" in scope of domain
one.two */
import comp :"http ://www.w3c.sca/comp.composite", proc :"http ://www.w3c.bpel/proc.bpel"
// sub-domain "three", grand-son of "one"
domain three{
// sub-domain one.two.three.four
domain four
// domain one.two.three.five
domain five
} // end of one.two.three domain
} // end of one.two domain
```

5.4.4 Élément instanciables : variables, opérations et types

Les variables, les opérations et les types du langage sont définis à l'aide d'un minimum de caractères, ou de mots clefs déjà utilisés par les langages orientés objet comme Java, ou C#, ou encore Scala.

Si plusieurs variables sont spécifiées en une fois, alors soit elles partagent toutes la même valeur, soit il faut définir au plus autant de valeurs qu'il y a de variables. Les variables sans valeur seront initialisées avec la valeur par défaut du type d'objet attendu (*Nil* par défaut). Le procédé de méta-description suit cette logique, soit une description est donnée, et donc appliquée à toutes les variables, soit il y a au plus

autant de descriptions qu'il y a de variables. Dans ce cas, les variables non associées à une description seront du type *Element*.

Instanciable element rules :

$instanciable\_element = [\text{"}'\text{"}\ [visibility]\ [access]\ [cardinality]]\ (*\ instance\ count\ *)$
$(variable \mid ([\text{" < "} \mid \text{" > "}]\ (*\ prefix \mid post\ operator\ order\ *)\ (operation \mid type))\,;$

$visibility = (\text{"}public\text{"} \mid \text{" + "}) \mid (\text{"}private\text{"} \mid \text{" − "}) \mid (\text{"}protected \mid \text{" } \sim \text{ "});$
$access = (\text{"}shared\text{"} \mid \text{"\&"}) \mid (\text{"}exclusive\text{"} \mid \text{"}X\text{"}) \mid \text{"}static\text{"}\ (*\ equivalent\ to\ one\ shared\ instance\ *);$

Variable rule :

$variable = \text{"}var\text{"}\ var\_identifiers\ [\text{" : "}\ [element\ (\text{","}\ element)*]\ (*\ value\ descriptions\ *)$
$[semantic]\ [\text{" = "}\ element\ (\text{","}\ element)*]]\ (*\ default\ values\ *);$

$absolute\_identifiers = absolute\_identifier\ (\text{","}\ absolute\_identifier)*;$
$var\_identifiers = identifier\ [\text{" = "}element]\ (*\ default\ value\ *)\ (\text{","}\ var\_identifier)*;$
$semantic = \text{" :: "}\ absolute\_identifiers\ (*\ equivalences\ *)$
$\text{" <:: "}\ absolute\_identifiers\ (*\ partOf\ *)$
$\text{" ::= "}\ element\ (*\ sementic\ cast\ *);$

Operation rule :

$operation = \text{"}op\text{"}\ [parameterized\_types]\ [identifier]\ (*\ identifier,\ generated\ if\ anonymous\ *)$
$[cardinality]\ (*\ instance\ count\ *)\ parameters\ [\text{" : "}\ element]\ (*\ result\ description\ *)$
$[exceptions]\ [semantic]\ [(\text{"}='\ element) \mid block]\ (*\ implementation\ *)$
$\text{" ::= "}\ element\ (*\ sementic\ cast\ *);$

$parameterized\_types = \text{" < "}\ parameterized\_type\ (\text{";"}\ parameterized\_type) *\ \text{" > ";}$
$parameterized\_type = [element\ \text{" :< "}]\ identifier\ [\text{" <: "}\ element\ (\text{","}\ element)*];$
$parameters = \text{"("}\ [parameter\ (\text{","}\ parameter)*]\ \text{")"}$
$parameter = [identifier]\ (\text{" : "}\ element)\ [\text{"}='\ element)$
$\text{" ::= "}\ element\ (*\ sementic\ cast\ *);$

$exceptions = \text{"}throws\text{"}\ absolute\_identifiers\ (*\ thrown\ exceptions\ *);$

Type rule :

$type = (\text{"}let\text{"} \mid \text{"}class\text{"})\ [parameterized\_types]\ identifier\ [parameters]\ (*\ header\ *)$
$[\text{" : "}\ element\ (\text{","}\ element)*]\ (*\ meta - description\ *)$
$[exceptions]\ [semantic]\ [block]\ (*\ body\ *)\ [\text{" = "}\ element]\ (*\ default\ value\ *);$

$constructor = \text{"}this\text{"}\ parameters\ block;$

Instanciable element code example :

$let\ Number$ { // default empty constructor
// constructor with two arguments which match with variables
$this(real, imaginary = 0)$
// native addition operation, implementation is specific to environment execution engine
$[native]op + (n : Number)$
$' \sim \&\ var\ real : Number$ // shared and protected variable real=0
$' \sim \&\ var\ imaginary : Number$ // shared and protected variable imaginary=0
$\} = 0$ // 0 *is the default value for not initialized Number elements*

$'X\ var\ x$ // *public and exclusive variable.* $x = Nil$
$' -\ \ var\ x, y, z\ :\ Number = 2, 2 * x$ // *private and shared variables.* $x = 2, y = 2 * x = 4, z = 0$
$' \sim\ \ var\ x, y = 2, z := 1$ // *protected and shared variables.*$x = z = 1, y = 2$

Invocation and meta-access rules :

$invocation = absolute\_identifier$
["#" $element$["!"]] (∗ *instanciation index and exclusive access* ∗)
("(" [$parameter\_value$ "," $parameter\_value$)∗] ")") (∗ $default$ *invocation* ∗)
|
(" " [$parameter\_value$ (" " $parameter\_value$)∗] " "]) (∗ *new word invocation* ∗);

$parameter\_value = [identifier$ " = "] $element$;

Invocation code examples :

element#2 // try to access to the second instance
element#2() // similar access to previous instantiation
element#2 !(2) // access to the second instance when available with one parameter

5.4.5 Décorations

Les décorations, ou *Decoration*s, permettent d'implanter le patron de conception du même nom [Gamma, 1995]. La puissance de ce patron respecte celui des propriétés C#, en intégrant la possibilité de paramétrer une annotation avec n'importe quel élément. De plus, il devient possible de décorer une décoration, afin de compléter le comportement d'une instance de décoration.

Un exemple d'utilisation est la possibilité de spécifier la durée de vie d'une décoration, ou d'associer une information de version. Passée une date, la décoration ciblée par la décoration de durée de vie est détruite, ou remplacée par une nouvelle décoration conforme à un nouvelle version.

Decoration rule :

$decoration$ ::= "[" $decoration * \ invocation$ "]" $element$; // decoration related to one element

5.4.6 Relations ontologiques

En plus des relations ontologiques par défaut du méta-modèle statique (sous-section 5.3.1), et en accord avec la spécification OWL [McGuiness et al., 2004], il est possible de spécifier de nouvelles relations ontologiques personnalisées, enrichies d'annotations pour ajouter des caractéristiques, telles que la

transitivité, la symétrie, et les propriétés fonctionnelles et inversement fonctionnelles.

Example of semantical code :

$let\ ontology(\_\ 2..)\{'1\ op\ ?(\_\ 2..) : Boolean\}$
$let\ forest\{var\ country\}$
$let\ forêt\ ::\ forest\{var\ pays\ ::\ forest.country\}\ //\ forêt \Leftrightarrow forest\ and\ forêt.pays \Leftrightarrow forest.country$
$let\ tree :: arbre <:: forest\ //\ \Rightarrow tree\ is\ arbre\ and\ part\ of\ forest$
$is?(arbre, tree)// \Rightarrow true\ because\ tree\ is\ arbre$
$partOf?(arbre, forêt)/* \Rightarrow true\ because\ arbre\ is\ tree, tree\ part\ of\ forest\ and\ forêt\ is\ forest */$
$//custom\ transitive\ semantical\ relationship$
$[transitive()]$
$let\ ancestor(\_\ 2..) : ontology$
$ancestor(grandfather, dad, son)//grandfather\ is\ ancestor\ of\ dad,\ dad\ is\ ancestor\ of\ son$
$ancestor?(grandfather, son)// \Rightarrow true\ because\ grandfather\ is\ ancestor\ of\ son\ by\ transitivity$

5.4.7 Instructions et évènements

Sept instructions sont proposées pour se rapprocher des instructions connues et utilisées par les langages de programmation habituels.

For, *While* et *DoWhile*

Ces instructions servent à simuler le comportement des boucles conditionnelles, à l'aide des instructions *Block*, *BackTo* et *Leave*.

If, *IfNil*, *Switch*, et *Match*

Ces instructions respectent les instructions de contrôle de même nom dans le langage Scala. Les trois servent à exécuter une instruction en fonction d'un test conditionnel plus ou moins complexe. Le *If* prend une condition et une ou deux instructions en entrée. Si la condition est vérifiée, alors la première instruction est exécutée, sinon, c'est la seconde. Le *IfNil* correspond à l'instruction "??" du C# qui retourne un élément parmi deux passés en paramètres. Si le premier paramètre vaut *Nil*, alors le second est retourné, c'est le premier qui est retourné. Le *Switch* prend un élément en entrée, et va chercher dans une séquence de valeurs si l'une est égale. Si l'une d'elles vérifie l'égalité, alors la première instruction qui correspond à cette valeur, ou aux suivantes est exécutée, ainsi que toutes celles qui suivent. Si aucune valeur ne correspond, il est possible de spécifier une instruction par défaut à exécuter. Finalement, le *Match* est similaire au *Switch* excepté que le test n'est pas une égalité, mais une compatibilité avec la structure d'entrée.

Alias

Cet élément permet d'identifier de manière unique tout élément du langage à l'aide d'un nom, plutôt que l'identifiant numérique utilisé par le modèle. Il devient alors possible de faire référence à un élément marqué à l'aide de ce nom. Par exemple, le parcours d'un fichier importé pourra être facilité en réduisant le chemin du fichier importé en un diminutif. Ou bien d'annoter un bloc d'instruction pour s'en servir de label.

Les instructions et les évènements définis dans le méta-modèle statique existent ici sous différentes formes. Soit ils utilisent une syntaxe simplifiée, soit ils utilisent une syntaxe propre aux langages existants.

Instruction rules (1/2) :

$lock = "\&!" \mid "lock"\ [element]\ (*\ execution\ context\ *)\ element\ (*\ instruction\ *)$
$assignment = target\ "="\ element;$
$instance = "this" \mid "self";$
$alias = identifier\ ":"\ element\ (*\ alias'\ name\ and\ target\ element\ *);$
$description = element\ ":";$
$cast = "("absolute\_identifier")"\ element;$
$jump = "jmp"\ target;$
$leave = "break" \mid "->"\ [target]\ ["="\ element];$

Instruction rules (2/2) :

$back\_to = "continue" \mid "<-"\ [target]$
$throw = "throw" \mid "!!"\ [element];$
$try = "try" \mid "!!?"\ element\ ("catch" \mid ">!!"\ element\ element)*\ ["finally" \mid "!!>"\ element];$
$fire = "fire" \mid "!"\ element\ (*\ event\ *);$
$listen = ">!"\ element\ (*\ event\ filter\ *)\ element\ (*\ listener\ *);$
$target = ("@", ["/" ,] (".." \mid "." \mid identifier))+;$
$if ::= ("if" \mid "?")\ element\ element\ ["else" \mid ":"]\ element;$
$if\_nil = element\ "??"\ element;$
$switch ::= ("switch"\ element) \mid (element\ "?="\ )\ "\{"\ ("case" \mid "is" \mid "="\ )\ element\ ":"\ (element\ ("\backslash n \mid ";"))*) * [("default" \mid "\_")\ ":"\ element]"\}";$
$match ::= ("match"\ element) \mid (element\ "?\sim"\ )\ "\{"\ ("case" \mid "as" \mid "\sim"\ )\ element\ ":"\ (element\ ("\backslash n \mid ";"))*) * [("default" \mid "\_")\ ":"\ element]"\}";$
$return = "return" \mid "=>"\ [element];$

5.4.8 Énumerateurs

Un dernier élément propre à ce langage est la possibilité de définir des énumérateurs, qui sont des types d'éléments où le constructeur est privé.

Enumerator rules :

$enum = "enum"\ enum\_header\ enum\_body;$
$enum\_header = [parameterized\_types]\ identifier\ [parameters]\ (*\ header\ *)\ [":"\ element\ ("," element)*]\ (*\ meta-description\ *)\ [exceptions]\ [semantic]$
$enum\_body = "\{"\ identifier\ ("," identifier)*\ (*\ values\ *)\ [";"\ (element\ ("\backslash n" \mid ";")+)*\ (*\ body\ *)\ "\}"\ ["="\ element]\ (*\ default\ value,\ first\ value\ if\ not\ defined\ *);$

Enumerator code example :

```
// access of instanciable elements
enum INSTANCIABLE_ELEMENT_VISIBILITY { PUBLIC, PROTECTED, PRIVATE; }
enum INSTANCIABLE_ELEMENT_VISIBILITY : : JAVA_VISIBILITY{
PUBLIC : :JAVA_VISIBILITY.PUBLIC
PROTECTED : :JAVA_VISIBILITY.PROTECTED
PRIVATE : :JAVA_VISIBILITY.PRIVATE
}
```

5.5 Dualité de communication inter-modèles/systèmes

La dualité de communication vise à synchroniser R-EMS avec les sous-systèmes ou avec d'autres parties de l'environnement.

Toute opération du modèle d'environnement permet d'informer automatiquement les sous-systèmes intéressés d'effectuer les traitements associés (notion de domaine de responsabilité), et réciproquement, le modèle d'environnement peut être automatiquement mis à jour en fonction de l'activité des sous-systèmes.

5.5.1 Accessibilité de R-EMS

R-EMS est accessible, soit par l'intermédiaire du paradigme de communication RPC (sous-section 2.5.1, page 30) pour l'invocation d'opérations (opération des sous-systèmes ou opération de lecture, écriture ou exécution du modèle d'environnement), soit par l'intermédiaire du paradigme de communication MOM (sous-section 2.5.4, page 31) pour la publication et la souscription d'évènements.

Pour couvrir ces deux paradigmes dans un milieu hétérogène et dynamique, R-* soutient FraSCAti (sous-section 1.4.1, page 6) et R-MOM (sous-section 4, page 67). Ainsi, c'est R-EMS qui s'adapte aux types de communication des sous-systèmes, et non l'inverse, préservant ainsi un maximum leur autonomie.

L'avantage de modéliser un environnement permet d'y inscrire n'importe quel élément d'un système, et ainsi réaliser un réel couplage entre un modèle d'environnements, et les technologies de communication inter-modèles par exemple. Les acteurs du système pourront spécifier dans le modèle distribué, le moyen de les contacter par voix RPC ou MOM via une simple URI, et les modèles pourront facilement interopérer avec n'importe quel utilisateur, ou faire l'intermédiaire entre utilisateurs, ou encore permettre aux différents utilisateurs de s'échanger mutuellement et directement des informations, pour peu qu'ils respectent le même protocole de communication associé à la configuration attendue, mais ce cas de figure ne concerne pas R-EMS.

De manière à faciliter les requêtes lourdes, un modèle d'environnement peut attendre en entrée un texte au format du langage de description d'environnement, plutôt que le format XML qui est devenu un standard très verbeux dans toutes les applications web, et communément utilisé par les approches de type MDE (sous-section 2.3.5, page 23).

Le modèle d'environnement devient accessible à toute plateforme, que ce soit un système, ou une plateforme d'intégration, ou même un autre environnement, et répond donc à la principale demande de contrôle qu'on pourrait espérer d'un système de systèmes (SoS).

5.5.2 Synchronisation des sous-systèmes avec R-EMS

Dans un objectif de synchronisation des sous-systèmes avec le modèle d'environnement de manière la moins intrusive possible, il convient de déployer des intercepteurs sur les opérations des sous-systèmes pour mettre à jour le modèle d'environnement au besoin.

Ce point fait d'ores et déjà parti des perspectives puisqu'on espère qu'il peut être automatisé en fonction des domaines de systèmes visés, mais en l'état, la mise en place de cette solution demandera l'utilisation de proxy afin de ne pas avoir à modifier le comportement des sous-systèmes existants.

Avec cette synchronisation, le modèle devient un réel observateur des sous-systèmes, et complété de l'étape d'accessibilité précédente, les capacités de supervision et de contrôle sont supportées par R-EMS.

5.6 Accès concurrents et Transactions

Le modèle d'environnement étant distribué, plusieurs systèmes peuvent souhaiter accéder en même temps à une même partie du modèle qui soit physiquement localisé à un seul endroit.

En accord avec Jim Gray et Andreas Reuter, "Un système transactionnel est la clef pour la gestion cohérente et l'exploitation sûre des ressources d'un système d'information" [Gray and Reuter, 1993]. Il faut donc que R-EMS puisse gérer les accès concurrents.

Dans le contexte d'accès concurrents, les transactions doivent également être couplées à un système de verrouillage des ressources du système, en fonction des parties lues, modifiées ou exécutées (si l'exécution introduit des verrous).

Finalement, ces transactions doivent être partagées entre plusieurs plateformes, puisqu'il est question de résoudre des opérations d'adaptation sur des éléments utilisés par différentes plateformes, et pouvant être bloquantes. Les mécanismes de communication introduisent de grandes possibilités de travail collaboratif, mais en même temps, une grande difficulté dans leur utilisation. En effet, les transactions concurrentes peut devenir encore plus complexe à résoudre que les adaptations une fois les transactions validées. C'est pourquoi il faut pouvoir superviser ce qu'il se passe, et le système d'environnement est encore une fois certainement une solution pour y arriver. Ainsi, les transactions agissant sur le modèle seraient elles-aussi modélisées dans le modèle, permettant à quelqu'un ayant les droits de les observer, de les analyser, de les tracer, ou même de les modifier/stopper.

Malheureusement, l'étude ne trouva pas le temps de s'intéresser à la question, et cet objectif fait d'ores et déjà parti des perspectives.

5.7 Réflexivité comportementale

La réflexion comportementale est un impératif dans la logique du tout réflexif de cette thèse (sous-section 1.3, page 5). Il est tout à fait possible de vouloir observer les comportements des fils d'exécution, afin de déboguer l'exécution de l'environnement par exemple, de manière distribuée et collaborative. Pour ce faire, il faut modéliser le mécanisme de gestion des contextes liés aux fils d'exécution dans le modèle, et de lier fortement le comportement des moteurs d'exécution avec ces éléments du modèle. À la vue de l'extensibilité du modèle, il pourrait même être possible de modéliser la logique d'exécution, afin de faire du déboguage distribué de la logique d'exécution.

La logique d'exécution d'un moteur ne fait pas partie de cette étude car elle reste spécifique aux besoins d'un environnement qui dépendent des plateformes matérielles et des technologies pour les réaliser.

5.8 Support des qualités de service

Cette dernière sous-section correspond au problème de fond lorsqu'on s'intéresse aux propriétés non-fonctionnelles d'un système distribué. Cependant, l'approche R-* a montré qu'il était nécessaire de s'équiper lourdement pour pouvoir convenablement affronter ce thème de recherche.

À présent que les modèles d'environnement sont décrits, il convient d'y introduire des exigences systèmes. Ici, il n'est point idée de s'intéresser à leur exécution, mais d'abord à leur définition.

[Kritikos and Plexousakis, 2006] fournit l'ontologie de qualités de service *OWL-Q* pour un système profitant exclusivement des WS (sous-section 2.5.3, page 30).

En complément de cette ontologie, ce document fournit la taxonomie 5.5 des qualités de service par type de besoins techniques de système, aussi bien haut-niveau, que bas-niveau.

Cette taxonomie reste extensible, et incomplète pour des utilisations futures (elles le sont toutes dans un environnement dynamique).

Pour résumer succinctement cette carte, on trouve :

- les mesures de temps *(T)*, que ce soit le temps de départ *(Ts)*, de fin *(Tf)*, de délai *(Td)* ou de fluctuation de temps *(Tj)*,
- la définition des flots *(F)* avec gestion de la priorité *(Fp)* et de l'ordre *(Fo)*,
- les capacités matérielles *(H)*, en rapport avec le processeur (fréquence, architecture *(Hpf)*, fil d'exécution *(Hpet)*), l'énergie *(He)* ou la mémoire (morte *(Hms*, vive *(Hmr)*, virtuelle *(Hmv)*, durée *(Hmad)*, historique *(Hmah)*, partagée *(Hmf)*),
- les types d'accès *(A)* (lecture, écriture, exécution),
- la notion de participation *(C)* par type *(Ct)* ou avec une notion d'importance *(Cn)*,
- la cohérence *(U)* qui se décline en fiabilité *(Ur)*,
- la localisation logique *(Li)* ou physique *(Lp)* qui se découpe sous la forme de différentes unités physiques (angle *(Lpan)*, vitesse *(Lpv)*, accélération *(Lpac)*, etc...).

Cette taxonomie mérite énormément de critiques car sa réalisation est loin d'être la spécialité de cette thèse. Mais le but est de partir de quelque chose de concret pour ensuite consolider les bases de R-EMS.

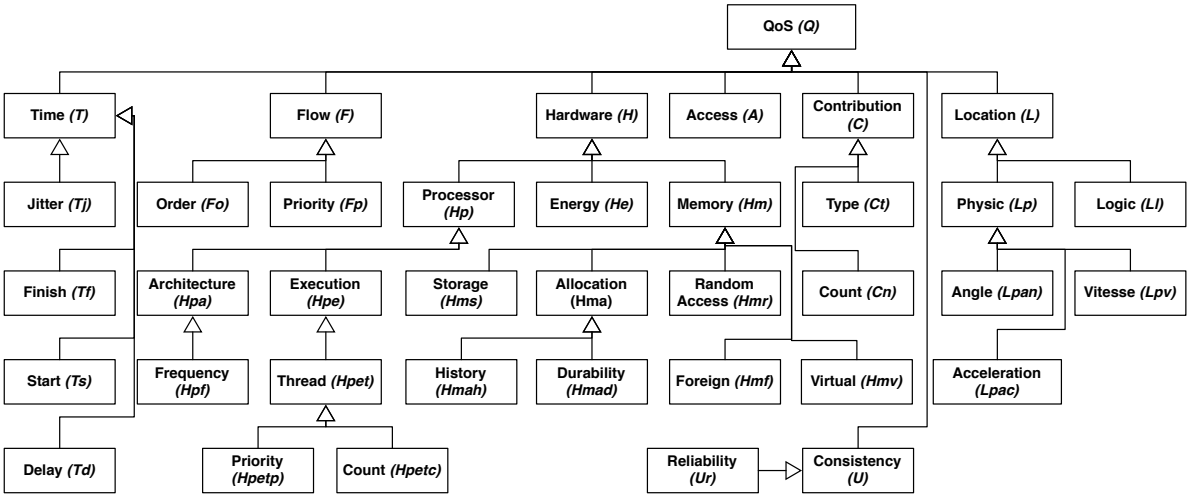


FIGURE 5.5 – **Taxonomie non-exhaustive des Qualités de services techniques à prendre en compte dans un système de systèmes** – Tableau de spécialisation de QoS

5.9 Mise en œuvre de R-EMS

L'implémentation réalisée s'appuie sur des technologies Java, et des travaux de recherche connexes, afin de garantir un maximum sa portabilité dans un milieu hétérogène. Cependant, du fait de son arrivée tardive dans l'étude, elle témoigne d'une immaturité importante pour ne supporter qu'un méta-modèle dynamique restreint, et suffisant pour assurer l'évaluation du chapitre 6, page 109.

Le tout est disponible à l'adresse suivante sous la forme d'un projet svn :

`http://websvn.ow2.org/listing.php?repname=frascati&path=/sandbox/jlabejof/R-EMS/`

Le diagramme de R-EM3 et le fichier XText du langage R-EML sont également disponibles dans l'annexe B, page 141.

5.9.1 Méta-modèle et modèles exécutable – R-EM3 et R-EM2

Le méta-modèle et les modèles sont développés à l'aide du canevas EMF [Steinberg et al., 2008].

L'outil permet de vérifier des règles de cohérence autour des constructions génériques et spécifiques des méta-modèles et des modèles.

EMF permet de générer à partir du méta-modèle une librairie Java de modèle exécutable. Ce modèle est étendu pour devenir une plateforme d'exécution d'environnement, en garantissant un couplage entre importations de modèles ou fichiers de configuration sur le réseau, et en liant les accès en mode RPC ou MOM avec les technologies FraSCati et R-MOM existantes.

La section B.1, page 141 offre plus de détails concernant le méta-modèle R-EM3 dans l'environnement EMF.

5.9.2 Moteur d'exécution - R-EME

La distribution du modèle s'appuie sur le canevas CDO et le travail de distribution de modèle [Jurack and Taentzer, 2009].

Le comportement des moteurs d'environnement complète le travail de distribution en liant fortement les descriptions co-localisées de type et de domaine. De plus le canevas CDO assure une utilisation partagée d'un même modèle, segmenté en domaine.

CDO offre aussi la possibilité de sauvegarder le modèle dans une base de données. Ce faisant, la gestion des versions des éléments est facilitée. Il gère également les accès concurrents à l'aide de transactions exclusives.

5.9.3 Langage d'environnement – R-EML

Le méta-modèle proposé s'appuie sur la méthode Xtext [Eysholdt and Behrens, 2010] pour concevoir un DSL (sous-section 2.3.5, page 23) à l'aide d'une grammaire textuelle, et d'importations de méta-modèles EMF.

Une fois la grammaire écrite, et le choix d'extension des fichiers source "*reml*" choisi, la méthode Xtext est capable de vérifier par preuves formelles qu'il n'y a pas d'ambiguïté dans la grammaire proposée, et le cas échéant, un méta-modèle et des extensions Eclipse sont générés.

Les extensions Eclipse sont enrichies de règles de cohérence dynamiques (recherche de la description d'un type à partir d'importations de ressources) afin d'accompagner le développeur dans l'écriture des fichiers "*reml*".

L'implantation proposée est spécifique à l'évaluation du prochain chapitre (6, page 109), de ce fait le fichier proposé est simplifié par rapport à la grammaire du langage donnée dans la sous-section 5.4.1.

La sous-section B.2, page 146 montre le contenu du fichier XText spécifique à l'évaluation.

5.10 Conclusion et résultats

Le but du système de gestion d'environnement étudié ici est d'abstraire toute la complexité de la définition exigences de leur réalisation. Ainsi, des systèmes distribués profitant de technologies adéquates peuvent collaborer ensembles à des exigences gérées à très haut niveau par un modèle distribué et transverse à leur réalisation, de manière à harmoniser les préoccupations bas et haut niveau.

Techniquement parlant, cette vision est primordiale pour déléguer des traitements complexes et pas nécessairement automatisés, à l'opposé du niveau composant des systèmes (sous-section 2.2.1, page 12).

La solution proposée supporte un modèle de programmation d'environnement partagé.

Les modèles de programmation introduisent de nouveaux patrons de programmation ou d'exécution, tels que le transtypage sémantique qui facilite la conversion de classe d'un modèle à l'autre, ou une méta-description structurelle plus riche que les approches existantes avec choix explicite des méthodes appartenant à un type, et non toutes les méthodes d'un type. Mais aussi le patron d'exécution concurrent d'instantiation, qui permet de gérer une politique sur le nombre et l'accès partagé/exclusif des instances d'une variable, d'une opération ou d'un type, et certaines instructions algorithmiques qui facilitent l'écriture de boucles (instructions *BackTo* et *Leave*).

À notre connaissance, le seul travail qui se rapproche le plus du méta-modèle R-EMS est le langage ALF [Alf, 2010]. ALF comprend environ deux cent cinquante neuf concepts (dont la plupart sont spécifiques à Java, OCL et opérateurs spécifiques), contre trente pour le méta-modèle de R-EMS (il n'y a pas d'opérateurs spécifiques, hormis ceux qui sont dédiés à la description distribuée de l'environnement et à la gestion des exceptions/évènements), soit environ un nombre de concepts neuf fois moins inférieur à apprendre pour une solution plus générique, plus riche (support des descriptions sémantiques et autres extension du langage OOP) et non contrainte à une seule représentation textuelle, orientée Java et OCL (sous-section 2.7.1, page 48).

L'approche DSL est réutilisée pour offrir un langage textuel et extensible, qui facilite l'utilisation du modèle de par sa nature peu verbeuse et dédiée au domaine de l'environnement, tout en préservant le réseau d'un surplus d'échanges d'information (sous-section 5.4) comme peut l'être un langage verbeux comme l'XML.

Le tout reste accessible à toute plateforme d'exécution par la mise en œuvre de l'interopérabilité à l'aide de R-MOM et FraSCAti (sous-section 5.5).

R-EMS se nourrit de configurations de systèmes existants ou à déployer, puis couplé à des contrôleurs de composants systèmes, les modifications apportées à l'environnement permettent de tisser dynamiquement des aspects de manière totalement non-intrusive avec les données d'entrée, enrichissant les configurations système avec des capacités réflexives.

Toutefois, tous les challenges identifiés dans l'introduction (section 5.1) n'ont pas trouvé de réponses (par exemple, la problématique des accès concurrents et transactionnels (section 5.6) n'a pu être étudié pour satisfaire une faisabilité de la solution), et d'autres liés au contexte dynamique de code partagé et distribué méritent qu'on s'y intéresse pour compléter la solution R-EMS, et faciliter le travail des utilisateurs.

Ce travail n'a pas été soumis à un journal ou à une conférence car il a démarré lors des derniers mois de la rédaction de ce manuscrit.

Chapitre 6

Évaluation de R-* dans le système de systèmes TACTICOS

Sommaire

Ce chapitre présente les différentes évaluations menées sur les contributions de ce manuscrit, afin de consolider la faisabilité des solutions. Le cas d'étude est le SoS TACTICOS.

6.1 Le système TACTICOS

Le cas d'étude pour évaluer les contributions s'appel TACTICOS [TAC, 2006]. Ce système de systèmes appartient à la famille des systèmes de gestion de combat CMS. Il est utilisé par 15 divisions navales dans le monde, et est en service depuis 1993.

Son objectif est de contrôler un ensemble de bateaux (actuellement, 22 classes de bateaux, depuis ceux de patrouille jusqu'à des contre-torpilleurs) et de matériels (tourelles, radars, etc...) depuis des postes de contrôle situés soit sur ces bateaux, soit dans des bases terrestres dédiées. Sa spécificité technique est d'utiliser le service de distribution de données DDS, pour assurer le flux maximal de 4 000 publications par seconde de données ($< 32ko$) d'information et de commandes entre tous les acteurs du système. Mais également pour permettre de gérer tous les composants autonomes du système.

L'architecture est dite centrée sur l'information, permettant de respecter les exigences opérationnelles suivantes :

- haut niveau d'automatisation,
- support de la montée en charge,

- tolérance aux fautes,
- fusion de données provenant des détecteurs,
- évaluation automatique des menaces,
- couplage automatique entre détecteurs et armes,
- support de commande extensible,
- facilité d'utilisation et de maintenance,
- flexibilité dans les missions et armements,
- simulation et entraînement intégré,
- support de qualités de services comme la latence, la fiabilité et la durée des informations à l'aide d'une base de données.

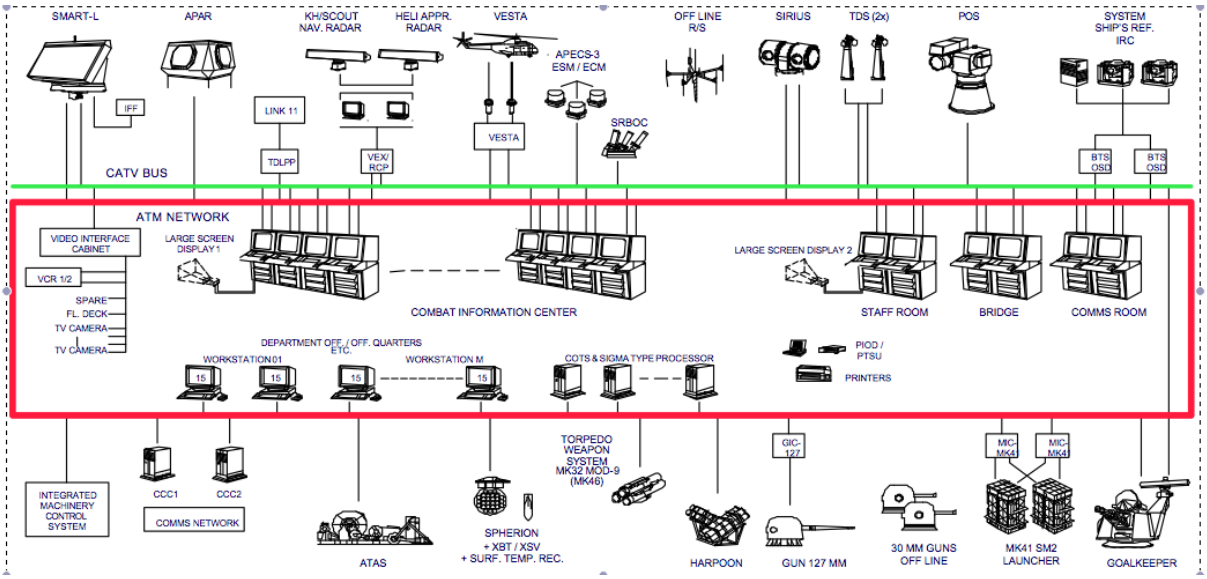


FIGURE 6.1 – **Exemple de communication entre acteurs du système TACTICOS** – transmissions de données (vert) et d'informations (rouge)

La figure 6.1 montre l'architecture avec deux niveaux de communication (traits vert et rouge) réalisant des échanges d'information entre trois types d'acteurs du système. Disposés de haut en bas sur la figure, il y a des détecteurs, des entités de contrôle, et des éléments exécutifs (réseaux de communications vers l'extérieur du système, armes, et machinerie).

Le niveau de communication vert réunit les détecteurs et les contrôleurs, permettant à ces derniers de traiter intelligemment les données enregistrées (fusion, corrélation). Une fois les données traitées, elles sont transmises sous forme d'informations aux exécutifs via le niveau de communication rouge, qui relie ainsi tous les acteurs.

La figure 6.2 représente les mêmes informations mais dans un vue centrée sur les acteurs par systèmes RT et non-RT.

Ainsi, l'ensemble des acteurs de type détection et exécutif répondent à des exigences temps-réel (et embarqués pour certains). Et les acteurs de contrôle répondent à des besoins de systèmes IT.

Un sous-modèle respectant la logique d'architecture de la figure 6.2 va être utilisé afin d'évaluer les contributions de cette thèse.

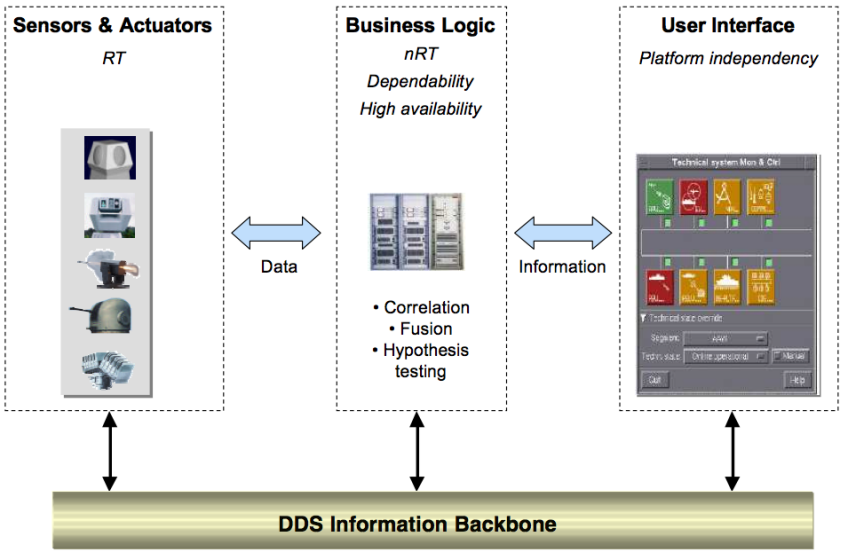


FIGURE 6.2 – **Échange de données et d'informations entre systèmes RT et non-RT de TACTICOS** – via un bus d'information DDS

6.2 Évaluation de R-* par comparaison d'implantation d'un sous-modèle de TACTICOS

L'évaluation des contributions va se dérouler sous la forme d'une implantation d'un sous modèle de TACTICOS, en utilisant ***(I)*** l'approche TACTICOS, ou ***(II)*** l'approche R-*.

6.2.1 Architecture d'évaluation

L'architecture respecte la communication d'informations et de données de la figure 6.2, mais le bus entre les contrôleurs et les interfaces utilisateur respectera une technologie MOM spécialisée pour les systèmes d'information, c'est à dire JMS ou AMQP.

De plus, TACTICOS utilise des senseurs qui n'utilisent pas le paradigme DDS pour communiquer avec les systèmes temps-réel. La solution adoptée est d'utiliser le protocole UDP pour transmettre les informations d'état ou celles qui sont enregistrées par son activité, vers une plateforme temps-réel qui va transformer les données reçues pour les retransmettre via le bus DDS.

Le système R-EMS est déployé pour permettre de modéliser l'architecture d'évaluation de R-*, avec les différents composants système, et quelques propriétés qui seront modifiées automatiquement durant l'exécution. R-EMS écoute chacun des composants métier via les mêmes bus de messages déjà utilisés. En plus des bus de messages de ***(I)***, ***(II)*** va utiliser des composants de synchronisation avec l'environnement (***Sync***) rattachés aux acteurs afin de supporter des communications de type RPC avec le système R-EMS, et mettre à jour leur configuration dès demande du système d'environnement par envoi de messages de modification de propriété.

L'architecture du système ***(II)*** va utiliser R-DDS et R-MOM pour communiquer avec les bus de données. Les acteurs du système RT vont utiliser R-DDS et R-MOM+DDS. Et les acteurs du système non-RT vont utiliser R-MOM+(UDP, JMS, AMQP).

De ce fait, l'architecture ***(II)*** sera évaluée de deux manières, une première fois avec R-DDS, puis une seconde fois avec R-MOM.

Les figures 6.3 et 6.4 représentent respectivement les architectures liées à l'évaluation normale et avec l'approche R-*.

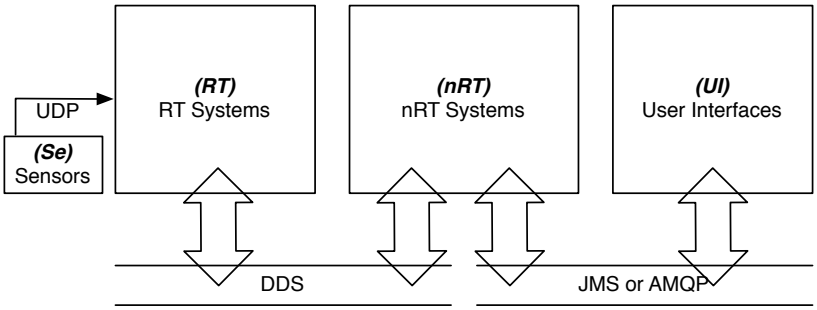


FIGURE 6.3 – **Architecture *(I)* du sous-système TACTICOS** – via les bus d'information DDS, JMS et AMQP

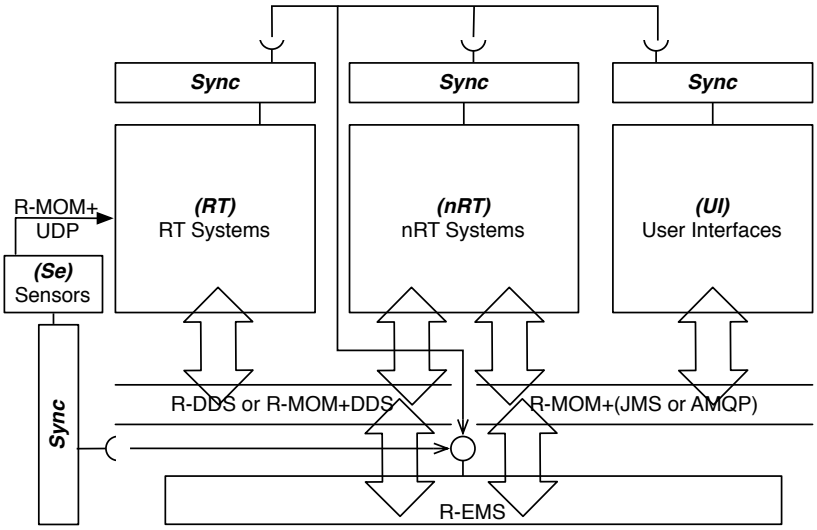


FIGURE 6.4 – **Architecture *(II)* du sous-système TACTICOS** – via les bus d'information R–DDS, R–MOM, et le support du système R-EMS

6.2.2 Scénario d'utilisation et de re-configuration

Le scénario discuté dans cette partie s'intéresse à deux grandes phases, c'est à dire, l'utilisation et la reconfiguration du système.

L'utilisation du système consiste à simplement prendre l'hypothèse que le système utilise un modèle composé de composants en place. Dans ce cas, nous utiliserons simplement les bus de communication pour assurer des échanges d'information.

Les phases de reconfiguration peuvent être multiples dans TACTICOS, mais nous nous intéressons principalement à celles centrées sur les moyens de communication, c'est à dire :

(D) modification d'une propriété dynamique des moyens de communication utilisés (applicable dans le cas d'une montée en charge par exemple),

(S) modification d'une propriété statique des moyens de communication utilisés (applicable dans le cas d'un changement de base de données, et nécessite un redéploiement de la ressource ciblée),

(P) changement du protocole JMS par AMQP (applicable pour des besoins de couverture de qualités de service assurées uniquement par le moteur AMQP, de performances, et également de futurs besoins d'interopérabilité entre TACTICOS et d'autres systèmes répartis).

L'évaluation de l'exécution du système est observée durant une minute qui est suffisante pour analyser un jeux de résultat en accord avec les exigences de TACTICOS.

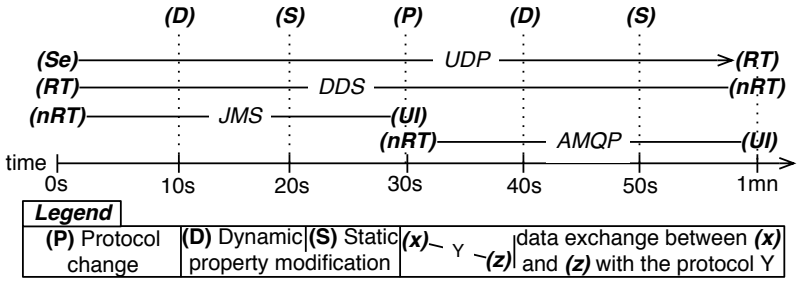


FIGURE 6.5 – **Échange continu de données dans le système de surveillance durant une minute** – Données UDP, DDS, JMS et AMQP

La figure 6.5 représente les transmissions de données durant cette même minute. La figure montre que les phases d'adaptation sont exécutées successivement à dix secondes d'intervalle, et en conséquence, JMS est utilisé pendant les 30 premières secondes, puis AMQP le reste du temps.

Autour de ces quatre phases, nous comparerons les qualités et les défauts des implantations ***(I)*** et ***(II)***.

6.3 Choix de configuration

L'architecture des contributions R-DDS et R-MOM vise la portabilité des paradigmes de communication en exposant des interfaces compatibles avec ces mêmes paradigmes.

Par exemple, R-DDS offre des interfaces compatibles avec la spécification tout en s'abstrayant des moteurs d'exécution DDS utilisés (sous-section 3.2.3). Et R-MOM utilise les composants de liaison d'interface pour assurer une portabilité avec des technologies intégrées (sous-section 4.2.5). Donc les configurations avec et sans R-* sont similaires du point de vue des interfaces de communication.

La seule différence est la mise en œuvre de R-DDS et de R-MOM qui profite des fichiers de configuration SCA d'extension "composite" pour configurer finement et simplement les échanges de données entre acteurs du système.

Par ailleurs, la configuration des assemblages R-MOM va utiliser un composant de type *EnvelopeConsumerManager* (sous-section 4.2.3) pour les composants d'envoi de données dans le but d'éviter d'interrompre le flux d'émission lors d'un changement de protocole de communication.

Le système R-EMS est déployé avant même la conception des trois acteurs. Durant la phase de conception du système, les acteurs sont modélisés, et les fichiers de configuration SCA sont utilisés de manière à aider R-EMS à modéliser les configurations possibles des différents acteurs, et leurs moyens de communication. Une fois la modélisation réalisée, il devient possible d'associer aux différents acteurs d'autres propriétés que celles déjà configurées dans les fichiers SCA, comme l'écriture des algorithmes d'adaptation ou la configuration des protocoles utilisés par R-MOM (voir la sous-section 6.3.1).

Ainsi, on simplifie la configuration du système distribué, en unifiant la logique de configuration, et en respectant les langages et les technologies utilisés par les sous-systèmes qui sont liés à des contraintes propres à leur domaine d'exécution.

Les fichiers de configuration SCA sont utilisés dans l'implantation de l'approche R-* pour la réalisation des contributions R-DDS et R-MOM. L'état de l'art a positionné cette configuration comme la meilleure pour configurer un modèle à composant qui se concentre sur les principes de composition et de communication entre composants, tout en laissant une totale liberté d'implantation ou de choix de communication des composants (sous-section 2.6.5). Les moyens de configurer TACTICOS ne sont pas connus, mais de ce fait, ils peuvent être, dans le meilleur des cas, proches de la configuration SCA en terme de précision de configuration, et donc du choix fait pour configurer les composants R-*.

Toutefois, l'incrément majeur de R-* est dans la gestion et les possibilités offertes autour des tâches d'adaptation du système, durant son exécution.

6.3.1 Implémentation du scénario d'adaptation

L'instance ***(I)*** du système comprend une solution spécifique à l'architecture de départ, et des traitements d'adaptation développés pour chaque technologie, c'est à dire, ***(D)*** et ***(S)*** écrits dans quatre différents modèles de programmation, et ***(P)***, qui se traduit par l'arrêt de la couche JMS et la configuration et le démarrage de la couche AMQP.

Ce qui montre que pour des tâches d'adaptation qui se révèlent communes à différentes solutions, il faut pouvoir maîtriser autant de modèles ou de langages de programmation qu'il y a de technologies différentes utilisées dans un système hétérogène.

Dans l'instance ***(II)*** du système, les traitements d'adaptation sont définis dans un même modèle de programmation et coordonnés par R-EMS avant d'être assurés par les acteurs. Ainsi, toutes les dix secondes, le modèle d'environnement géré par R-EMS va être modifié, puis les acteurs vont être notifiés de la modification, et vont devoir en conséquence la gérer en exécutant la tâche adéquate pour réaliser l'adaptation. Finalement, les acteurs vont prévenir le modèle que l'adaptation a bien eu lieu ou non.

Les figures 6.6, 6.7, 6.8, 6.9 et 6.10 sont des éléments du code dans le langage R-EML utilisé pour configurer R-EMS vis à vis du scénario prévu.

```
// domain TACTICOS
domain TACTICOS
```

FIGURE 6.6 – **Définition du domaine TACTICOS dans R-EMS** – Utilisation du langage R-EML

La figure 6.6 permet dans un premier temps de spécifier le nom du domaine, en l'occurrence, il s'agit du domaine TACTICOS.

La figure 6.7 montre comment d'autres partie du système et des fichiers de configuration SCA sont importés sous la forme de librairies. Le but est de pouvoir manipuler le contenu des fichiers une fois importés. Ainsi, le fichier de nom *System.rems* contient tous les éléments pour réaliser des traitements de base, comme par exemple, pouvoir définir des variables. Ensuite, le fichier *RMOM.rems* contient les éléments permettant de manipuler l'instance de R-MOM utilisé par R-EMS. Finalement, le fichier *SCA.rems* permet d'importer les éléments dédiés à manipuler des fichiers de configuration SCA, qui sont importés et associés à des diminutifs, relativement aux acteurs auxquels ils correspondent.

```
// importations of system, rmom and sca libraries, and SCA configuration files
import System.rems, RMOM.rems, SCA.rems,
    "http://tacticos.sos/Se.composite" as se,
    "http://tacticos.sos/RT.composite" as rt,
    "http://tacticos.sos/nRT.composite" as nrt,
    "http://tacticos.sos/UI.composite" as ui
```

FIGURE 6.7 – **Importation des ressources pour le système TACTICOS dans R-EMS** – Utilisation du langage R-EML

La figure 6.8 montre comment les types d'acteur, de communication et de reconfiguration sont définis, respectivement à l'aide des énumérateurs de nom *ACTOR*, *COM* et *RECONF*. Puis la variable globale de nom *reconf* est définie, à laquelle des composant d'écoute (composants fils de nom "listener" pour chaque acteur) définis dans les fichiers SCA vont venir écouter toute modification de la variable. Ce qui permettra à l'exécution de notifier les composants d'écoute dès le changement de valeur de la variable de nom *reconf*.

```
// actors
enum ACTOR {SE, RT, nRT, UI}
// communication
enum COM {UDP, DDS, JMS, AMQP}
// reconfigurations
enum RECONF {DYNAMIC, STATIC, CHANGE}
// global reconfiguration step, every actor should listen to value modification
var reconf = DYNAMIC
// catch reconf modification event
>! reconf ~ {se.listener.listen rt.listener.listen nrt.listener.listen ui.listener.listen}
// protocol configurations
var UDPProtocol = "UDP:TACTICOS:8080"
var DDSProtocol = "DDS:TACTICOS:DATA"
var JMSProtocol = "JMS:QUEUE:INFORMATION"
var AMQPProtocol = "AMQP:QUEUE:INFORMATION"
```

FIGURE 6.8 – **Définition des types d'acteur, de communication, de reconfiguration, de la configuration des protocoles de communication et d'une écoute de notification d'évènement d'effet de reconfiguration du système TACTICOS dans R-EMS** – Utilisation du langage R-EML

Pour rappel, le mécanisme de notification est assuré par R-MOM afin de laisser la possibilité aux éléments extérieurs à R-EMS d'utiliser le moyen de communication asynchrone qu'ils souhaitent utiliser.

Finalement, des champs texte correspondant aux configurations des protocoles gérés par R-MOM sont définis dans quatre variables.

La figure 6.9 définit l'opération principale de notre scénario, qui commence par configurer les différents acteurs à l'aide des configurations destinées aux couches R-MOM. Puis à dérouler le scénario d'adaptation en modifiant toutes les dix secondes la variable *reconf*.

La dernière figure (6.10) de cette section est l'algorithme d'adaptation de la couche JMS pour les acteurs ***(nRT)*** et ***(UI)***, paramétré par le type d'acteur et l'interface liée à la couche JMS.

Dans cette évaluation, l'algorithme a été réécrit manuellement, mais dans une optique d'utilisation optimale de R-EMS, il apparaît nécessaire d'utiliser le principe des modèles PIM/PSM (voir la sous-section 5.2, page 88) pour que le code cible spécifique soit généré à partir de l'algorithme modélisé indépendamment des plateformes d'exécution dans le modèle R-EMS.

Les autres algorithmes ont également été écrits, mais celui-ci suffit à comprendre la logique de déroulement du scénario dans l'implémentation ***(II)***.

```
// main execution
op run {
    // system configuration
    se.publisher.addProtocol(UDPProtocol)
    rt.subscriber.addProtocol(UDPProtocol)
    rt.publisher.addProtocol(DDSProtocol)
    nrt.subscriber.addProtocol(DDSProtocol)
    nrt.publisher.addProtocol(JMSProtocol)
    ui.subscriber.addProtocol(JMSProtocol)
    // every 10 seconds, do an adaptation task
    while(true) {
        System.sleep(10s) // sleep run operation during 10 seconds
        reconf=reconf.next // update value with next one modulo RECONF values count
    }
}
```

FIGURE 6.9 – **Définition de l'opération d'exécution du système TACTICOS dans R-EMS** – Utilisation du langage R-EML

```
[native("JMS")] // // let the "JMS" resource run this method
op reconf_JMS(actor, itf) { // reconfiguration parameterized by an actor and an interface
    reconf?={ // reconf switch
        RECONF.DYNAMIC: // if reconf equals to DYNAMIC
            actor?={ // actor switch
                ACTOR.nRT: // if actor equals to nRT then reset priority dynamic property
                    var priority=itf.getPriority(); itf.setPriority(priority)
                    => // and leave this operation
                ACTOR.UI:   // if actor equals to UI, then reset listener dynamic property
                    var listener=itf.getMessageListener(); itf.setMessageListener(Nil)
                    itf.setMessageListener(listener)}
            -> // leave reconf switch
        RECONF.STATIC: // if reconf equals to STATIC
            actor?={ // actor switch
                ACTOR.nRT: // if actor equals to nRT, then remove and add the publisher component
                    var publisher=it.publisher
                    it.remove(publisher) ; it.add(publisher)
                    => // leave this operation
                ACTOR.UI: // if actor equals to UI, then remove and add the subscriber component
                    var subscriber=ui.subscriber
                    ui.remove(subscriber)
                    ui.add(subscriber)}
            -> // leave reconf switch
        RECONF.CHANGE: // if reconf equals to CHANGE
            actor?={ // switch on actor
                ACTOR.nRT: // if actor equals to nRT, add AMQP protocol then remove JMS protocol
                    nrt.publisher.addProtocol(AMQPProtocol)
                    nrt.publisher.removeProtocol(JMSProtocol)
                    => // leave this operation
                ACTOR.UI: // if actor equals to UI, add AMQP protocol, then remove JMS protocol
                    ui.subscriber.addProtocol(AMQPProtocol)
                    ui.subscriber.removeProtocol(JMSProtocol)}
    }
}
```

FIGURE 6.10 – **Définition de l'algorithme d'adaptation de la couche JMS du système TACTICOS dans R-EMS** – Utilisation du langage R-EML

6.3.2 Plateformes d'exécution de l'évaluation

Nous considérons un acteur par colonne de la figure 6.3 et 6.4. Chaque acteur va être exécuté sur une plateforme indépendante. ***(Se)***, ***(RT)***, ***(nRT)*** et ***(UI)*** vont être respectivement exécutés sur des machines virtuelles et utilisant Ubuntu 10. R-EMS va être exécuté sur la plateforme d'exécution native. Toutes les machines virtuelles sont exécutées sur un MacBookPro 2.8 GHz Intel Core 2 Duo, 4Go 1067 MHz DDR3, utilisant Mac OS X 10.7.

Même si cet environnement d'exécution n'est pas représentatif du système TACTICOS, il est toute fois suffisant pour comparer les choix d'architecture et les résultats d'exécution entre ***(I)*** et ***(II)***. Il permet entre autre de se baser sur la même horloge machine quelque soit les acteurs du système.

6.4 Exécution du système

À l'exécution, le système va devoir assurer la transmission de données entre acteurs, tout en respectant les exigences non-fonctionnelles décrites dans R-EMS, comme la disponibilité, et d'autres non étudiées dans ce manuscrit (sécurité, fiabilité, etc...). Respecter cette exécution c'est devoir évoluer dans un environnement dynamique, en effectuant les tâches d'adaptation de type ***(D)***, ***(S)*** et ***(P)***.

En plus des tâches d'adaptation pré-citées, les tâches d'envoi et de réception de données sont définies respectivement par les acronymes ***E*** et ***R***.

La suite de cette section va ainsi étudier les différents comportements issus des choix d'architecture ***(I)*** et ***(II)***, mesurer les temps d'activité de différentes fonctions du système, comme l'envoi et la réception de données, mais aussi les opérations de reconfiguration.

L'objectif est de quantifier le surcoût induit par le traitement des couches de composant réflexif et des architectures adaptatives au dessus des technologies de communication utilisées.

Toutefois, l'exigence forte du système TACTICOS qui souhaite atteindre 4 000 publications par seconde de données de taille inférieur à $32ko$ (sous-section 6.1) va servir d'indicateur de validité pour l'étude.

Afin d'étendre les mesures à d'autres systèmes, les premières 10 000 données envoyées et reçues serviront de compteur de mesure, auxquelles nous en déduiront une valeur de comparaison avec l'objectif des 4 000 publications. Ainsi, les opérations d'écriture ou de lecture de données comprennent le traitement de 10 000 données.

Les données échangées seront de simples tableaux d'octets que nous remplirons aléatoirement, mais dont la taille va varier de manière à pouvoir faire une analyse plus large sur les comparaisons entre solutions ***(I)*** et ***(II)*** vis à vis de différentes tailles de données.

6.4.1 Évaluation de R-EMS

Durant le choix de configuration, R-EMS s'est montré décisif en terme d'intérêt par rapport aux solutions existantes.

L'évaluation de R-EMS va consister à observer les effets escomptés sur l'échelle de temps des reconfigurations attendues par le scénario. La configuration des fichiers SCA a bien été prise en compte, et le processus de demande d'adaptation auprès du modèle d'évènement, puis la notification auprès des acteurs, et finalement la demande des acteurs à propos de la nouvelle configuration ont également montré une modification de la configuration des composants métier des acteurs.

R-EMS vérifie qu'un système d'environnement est utile pour ordonner à des composants de réaliser des opérations d'adaptation dans un milieu hétérogène, en unifiant une configuration distribuée, tout en respectant les contraintes liées aux domaines des différents sous-systèmes.

Contrairement aux contributions R-DDS et R-MOM, des mesures de temps de traitement de R-EMS sont inutiles dans un contexte où R-EMS n'est pas exécuté sur un serveur dédié, et, à notre connaissance, son utilisation n'est pas comparable avec des systèmes existants. De plus, une minute d'exécution de l'évaluation a suffit à dérouler complètement le scénario, et mesurer les temps d'envoi et de réception de 10 000 données comme vous le verrez par la suite.

6.4.2 Formalisation des mesures

Cette sous-section propose de s'attarder sur une formalisation de ce qu'il y a à mesurer dans le temps de traitement des contributions R-MOM et R-DDS, pour en déduire une pré-analyse des résultats à observer.

Pour l'ensemble des mesures, on considère les ensembles suivants :

Soit $\mathbb{S} = \{\boldsymbol{I}, \boldsymbol{II}\}$ l'ensemble des implantations du système d'évaluation.

Soit $\mathbb{T} = \{8o, 512o, 1ko, 32ko\}$ l'ensemble des tailles de données à échanger entre acteurs du système.

Soit $\mathbb{P} = \{UDP, DDS, JMS, AMQP\}$ l'ensemble des protocoles utilisés au cours de l'évaluation.

Soit $\mathbb{O} = \{\boldsymbol{E}, \boldsymbol{R}, \boldsymbol{D}, \boldsymbol{S}, \boldsymbol{P}\}$ l'ensemble des opérations évaluées dans cette section.

Soit $\mathbb{I} = \{\textit{R-DDS}, \textit{R-MOM}\}$ l'ensemble des différentes contributions R-* évaluées.

Soit ϵ la variable dite d'imprécision de calcul. Cette variable introduit une variation dans la mesure des différents temps de traitement associés aux opérations de $\mathbb{O}$ dans les systèmes de $\mathbb{S}$ qui peut varier en fonction de l'imprédictabilité des plateformes d'exécution, dû à une architecture plus ou moins capricieuse (vitesse d'accès à la mémoire vive, programmes démons des machines virtuelles, exécution du ramasse-miettes java, etc...).

Afin de se rapprocher un maximum d'un contexte d'exécution réel, ϵ ne doit pas être minimisé car il existe toujours du fait qu'aucune plateforme n'a été développée spécifiquement pour exécuter notre système d'évaluation. Afin de considérer un ϵ possible, chaque mesure sera répétée plusieurs de fois, et une moyenne finale sera calculée. Après une première analyse, il s'avère que cinquante répétitions sont suffisantes pour obtenir au moins 90% de mesures dans un intervalle de temps réduit à l'échelle de la seconde, laissant de côté 5 mesures minimales ou considérées comme du bruit. Cette moyenne permet de s'abstraire un maximum du bruit induit par ϵ.

Voyons à présent de manière formelle les résultats qui seront observés tout au long des différentes évaluations.

Soit la fonction $\tau : \mathbb{R}$ le temps d'accès entre une interface d'un composant FraSCAti et le métier associé. τ dépend du code généré par la plateforme FraSCAti. Or la logique de génération de code est spécifique à la signature de la méthode à adapter avec l'interface et le code métier visé [Bruneton et al., 2004]. De ce fait, la mesure de τ est la même quelque soit l'opération, le protocole ou la taille des données. Donc, la fonction ne dépend d'aucun paramètre. Après mesure, il apparaît que $\tau \in [5\mu s; 15\mu s]$.

Soit $\psi : \frac{\mathbb{O},\mathbb{P},\mathbb{T}\longmapsto\mathbb{R}}{o,p,t\to r}$ le temps de traitement de l'opération o, du protocole p, pour des données de taille t.

Soit $\theta_{\boldsymbol{I}} : \frac{\mathbb{O},\mathbb{P},\mathbb{T}\longmapsto\mathbb{R}}{o,p,t\to r}$ le temps de traitement de l'opération o, du protocole p, pour des données de taille t, dans le système ***(I)***.

Soit $\theta_{\boldsymbol{II}} : \frac{\mathbb{O},\mathbb{P},\mathbb{I},\mathbb{T}\longmapsto\mathbb{R}}{o,p,i,t\to r}$ le temps de traitement de l'opération o, du protocole p, dans l'intergiciel i, pour des données de taille t, dans le système ***(II)***.

Soit $\phi : \frac{\mathbb{O},\mathbb{P},\mathbb{I}\longmapsto\mathbb{R}}{o,p,i\to r}$ le temps de traitements métier d'un composant pour le traitement de o moins le temps de traitement du protocole p associé dans le système ***(II)*** utilisant l'intergiciel i.

En considérant les fonctions ψ, θ et ϕ, on a :

$\forall o \in \mathbb{O}, \forall p \in \mathbb{P}, \forall t \in \mathbb{T}, \forall i \in \mathbb{I}, \theta_{\boldsymbol{II}}(o,p,i,t) = \tau + \phi(o,p,i) + \psi(o,p,t)$.

Or le système ***(I)*** considère une solution spécifique aux protocoles utilisés, c'est à dire que :

$\theta_{\boldsymbol{I}}(o,p,t) = \psi(o,p,t)$

$\Rightarrow \theta_{\boldsymbol{II}}(o,p,i,t) = \tau + \phi(o,p,i) + \psi(o,p,t) = \tau + \phi(o,p,i) + \theta_{\boldsymbol{I}}(o,p,t)$

$\Leftrightarrow \theta_{\boldsymbol{II}}(o,p,i,t) - \theta_{\boldsymbol{I}}(o,p,t) = \tau + \phi(o,p,i)$.

Donc la différence des temps de traitement entre les systèmes ***(I)*** et ***(II)*** dépend de la somme $\tau + \phi(o,p,i)$. La taille de la donnée n'est pas un facteur influant sur la différence de la mesure des temps de traitement entre ***(I)*** et ***(II)***, alors qu'elle influe sur les opérations ***(E)*** et ***(R)*** via la fonction de (dé)sérialisation des données et de taille des packets réseau à allouer.

À présent, voyons combien vaut la somme $\tau + \phi(o,p,i)$ par rapport à $\theta_{\boldsymbol{I}}(o,p,t)$.

6.4.3 Bus entre systèmes RT

Les systèmes ***(I)*** et ***(II)*** utilisent tous deux le moteur d'exécution OpenSplice v4.

Par ailleurs, le système ***(II)*** utilise tour à tour R-DDS et la liaison DDS dans R-MOM pour pouvoir comparer l'efficacité de chacun vis à vis d'un même protocole de communication.

Transmission de données avec R-DDS

R-DDS n'utilisant pas plus de traitement intermédiaire entre les appels au moteur d'exécution pour la lecture par "tirage" (sous-section 2.5.4) et l'écriture de données, il vaut mieux utiliser la lecture par "poussée" de R-DDS, qui elle fait appel à une transformation des évènements du moteur d'exécution DDS en évènements génériques qui sont re-transmis entre composants, et vers les acteurs intéressés (voir le contrôleur d'évènements de la sous-section 3.2.3).

L'utilisation du mode par "poussée" permettra de voir dans le meilleur des cas (écriture de donnée) et dans le pire des cas (lecture de donnée par "poussée"), comment R-DDS se défend vis à vis des exigences du système TACTICOS, et de son concurrent directe qui n'utilise pas autant de traitements intermédiaires.

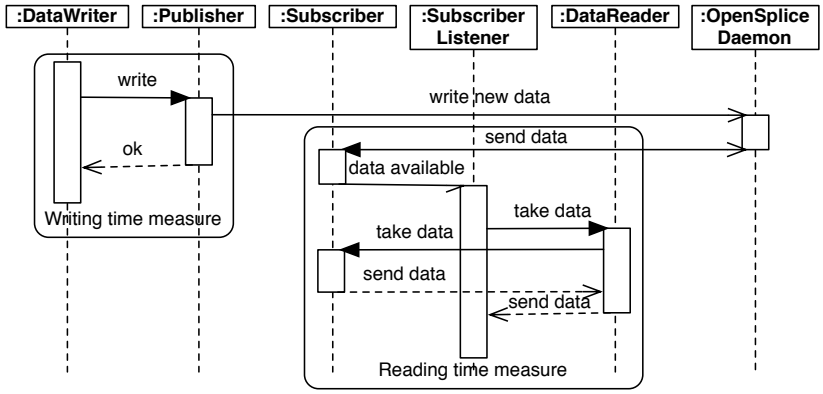


FIGURE 6.11 – **Diagramme de séquence d'opérations d'écriture et lecture d'une donnée DDS** – Avec de mesure de temps des opérations

Le diagramme de séquence 6.11 illustre les phases d'écriture et de lecture en mode "poussée" d'une donnée avec le moteur OpenSplice. Il met également en évidence ce qui est mesuré, c'est à dire le temps d'appel des opérations d'écriture et de lecture.

La taille des boîtes de mesure de temps du diagramme met en évidence la complexité du métier des composants, et donc l'influence sur le temps de traitement de la fonction ϕ. $\phi(\boldsymbol{E}, DDS,\ R\text{-}DDS)$ sera proche de 0 puisqu'il s'agît d'un simple appel direct à une méthode d'envoi de donnée. Alors que $\phi(\boldsymbol{R}, DDS,\ R\text{-}DDS)$ est beaucoup plus complexe, en sollicitant trois acteurs et trois appels de méthode synchrones.

La diagramme à barre 6.12 représente les résultats mesurés pour les opérations d'écriture, de lecture, pour les solutions ***(I)*** et ***(II)***. En abscisse, il y a quatre parties correspondant aux données de taille 8, 512, 1k et 32k octets. En ordonnée, l'échelle est de 0 à 450 milli-secondes. Les mesures concernant les données de taille 32k octets sont le double des autres mesures car la taille des paquets réseau utilisés pour transporter les données est deux fois plus grande que pour les autres mesures.

Ce diagramme met en évidence que les temps de traitement des opérations d'écriture de DDS et R-DDS sont de même durée sur l'échelle de la milli-seconde. La différence est négligeable au point que pour les données de taille 64 octets, le temps de traitement d'écriture de R-DDS est inférieur à celui de DDS.

Concernant les opérations de lecture, R-DDS dure 5% (pour les données de taille 64 octets) à 20% (pour les données de taille 1 024 octets) plus longtemps que DDS. Ceci s'explique par la complexité induite du mécanisme étendu et complexe des évènements et du mode par "poussée" de R-DDS.

Conjointement à notre calcul de mesure (sous-section 6.4.2), plus la taille des données augmente, et plus la différence de temps de traitement de transmission des données diminue :

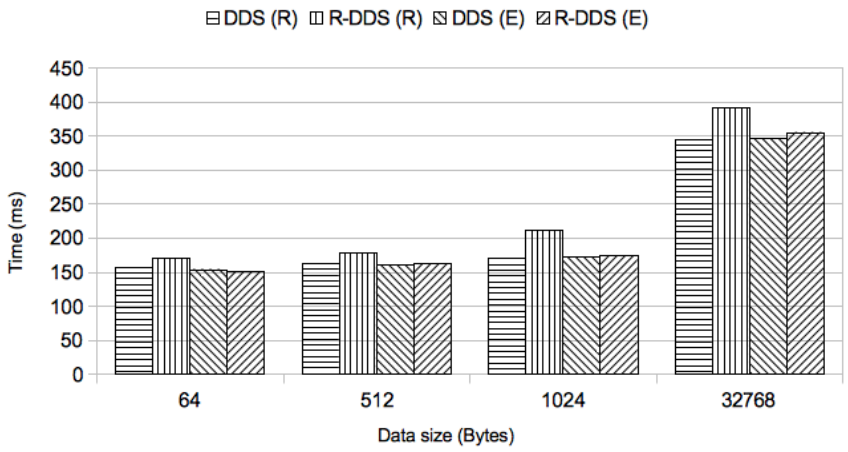


FIGURE 6.12 – **Temps de traitement de 10 000 lectures (R) et écritures (E), pour les solutions DDS et R-DDS** – Moyenne sur 50 séries pour des tailles de données de 8, 512, $1k$ et $32k$ octets

Par ailleurs, l'objectif est d'assurer un maximum de 4 000 publications de données par seconde. Même si R-DDS induit un surcoût de 20%, le résultat final respecte bien les exigences du système TACTICOS, puisqu'il permet d'envoyer et recevoir 10 000 données de taille $32ko$ sous les $400ms$.

Transmission de données de la liaison DDS avec R-MOM

Le métier des composants d'envoi et de réception de données de R-MOM est spécifique au protocole lié, et comprend comme traitement complexe supplémentaire la sérialisation des *Envelope*s avec le protocole visé. Or, cette sérialisation est négligeable à l'échelle choisie pour observer l'utilisation du bus DDS. Puisque ϕ est encore plus petite, il devient encore plus difficile d'observer une différence de temps de traitement entre les deux implantations du système évalué.

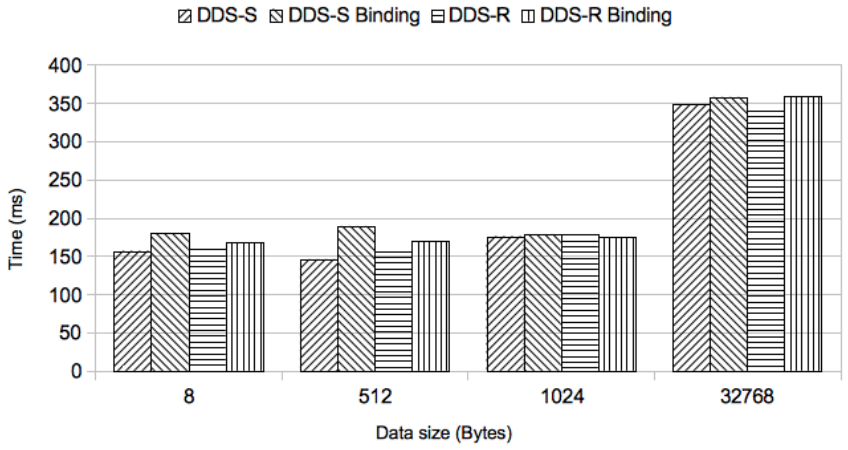


FIGURE 6.13 – **Temps d'envoi (-S) et de réception (-R) de 10 000 données via le protocole DDS et liaison R-MOM DDS** – Moyenne sur 50 sessions

La diagramme à barre 6.13 expose les résultats de temps de transmission de données entre DDS et R-MOM+DDS pour chaque taille de $\mathbb{T}$, sur une échelle de 0 à $400ms$.

Les trois premières taille de données utilisent le même traitement, avec la même taille de paquets réseau (seules les valeurs changent car il s'agît d'une moyenne). Donc seuls seront comparés les résultats

observés pour les tailles de donnée $1ko$ et $32ko$. La différence de temps de traitement existe mais reste inférieur à 1%, voire est même négative pour la réception de données de taille $1ko$.

Même si cette fois, la différence est encore plus discrète que pour les mesures faites avec R-DDS, le résultat est que l'exigence de TACTICOS concernant les 4 000 publications est toujours respectée. On admet donc que R-DDS et R-MOM+DDS induisent un sur-coût de temps d'envoi ou de réception de données négligeable quelque soit la taille des données.

6.4.4 Bus pour systèmes non-RT

Cette fois, le processus de mesure est un peu différent que pour celui du bus RT.

R-MOM est utilisé dans le système ***(II)*** pour envoyer des données via les protocoles UDP, JMS et AMQP.

Utilisation du protocole UDP

UDP est un protocole qui fait partie de la couche de transport. Il est donc souvent utilisé, au même titre que TCP par d'autres protocoles plus intelligents [57], comme les MOMs (sous-section 2.5.4).

OpenSplice utilise UDP comme couche de transport, de ce fait, les résultats observables entre temps de traitement de ***(E)*** et ***(R)*** pour les deux systèmes ***(I)*** et ***(II)*** seront plus importants que pour le bus DDS.

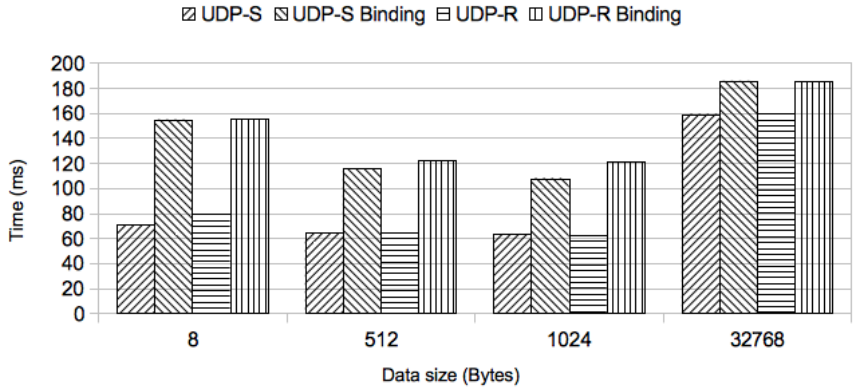


FIGURE 6.14 – **Temps d'envoi (-S) et de réception (-R) de 10 000 données via le protocole UDP et liaison R-MOM UDP** – Moyenne sur 50 sessions

La figure 6.14 met en évidence cette différence, sur une échelle de 0 à $200ms$. Tout comme pour le bus DDS, il est possible d'identifier l'utilisation de deux tailles de paquets. Une première qui permet de transporter des données de taille $8o$, $512o$ et $1ko$, et une seconde deux fois plus grande que la première pour transporter des données de taille $32ko$.

La première observation *(i)* est que pour la première famille de paquets (inférieurs à $32ko$), le protocole UDP met deux fois moins de temps à s'exécuter que son homologue avec R-MOM. La dernière taille *(ii)* est par contre beaucoup moins rapide à s'exécuter, et la différence avec la version R-MOM n'est plus que de 12%.

$\forall o \in \{\boldsymbol{E}, \boldsymbol{R}\}, t \leq 1ko, \theta_{\boldsymbol{II}}(o, UDP,\ R\text{-}MOM, t) = 2 * \theta_{\boldsymbol{I}}(o, UDP, t)$.
Or $\forall o' \in \mathbb{O}, \forall p \in \mathbb{P}, \forall t' \in \mathbb{T}, \forall i \in \mathbb{I}, \theta_{\boldsymbol{II}}(o', p, i, t') - \theta_{\boldsymbol{I}}(o', p, t') = \tau + \phi(o', p, i)$
$\Rightarrow \theta_{\boldsymbol{I}}(o, UDP, t) = \tau + \phi(o, UDP,\ R\text{-}MOM)$
$\Rightarrow (\tau + \phi(o, UDP,\ R\text{-}MOM)) * 10\ 000 \leq 160ms$
De plus, la sous-section 6.4.2 a montré que $\tau \in [5\mu s; 15\mu s]$.
$\Rightarrow (\tau * 10\ 000) \in [50ms; 150ms]$
$\Rightarrow \phi(o, UDP,\ R\text{-}MOM) * 10\ 000 \in [10ms; 110ms]$

57. La couche métier est accompagnée de traitements logiques et spécifiques à un paradigme de communication particulier.

$\Leftrightarrow \phi(o, UDP,\ R\text{-}MOM) \in [1\mu s; 11\mu s]$.

Dans notre contexte d'évaluation, le code métier des composants est beaucoup plus rapide à s'exécuter que le code d'accès au code métier, qui lui est aussi long à s'exécuter que l'envoi ou la réception de données UDP de taille inférieur ou égale à $1ko$. C'est pourquoi, lorsque la taille des paquets et le temps d'exécution pour l'envoi ou la réception de données sont doublés, la valeur de la somme $\tau + \phi(o, UDP, R-MOM)$ ne change pas, et la différence des mesures entre ***(I)*** et ***(II)*** diminue jusquà devenir négligeable comme ce fut le cas lors de l'analyse du bus DDS avec R-MOM (voir figure 6.13) sur une échelle de temps deux fois plus grande.

Utilisation des bus JMS et AMQP

Les technologies de communication JMS et AMQP répondent au paradigme MOM. Les implantations de JMS et AMQP évaluées utilisent respectivement comme couche de transport les protocoles TCP et UDP. Cependant, le métier de chacun est dynamique et par conséquent beaucoup moins rapides qu'un intergiciel aussi spécifique au domaine de l'embarqué qu'est OpenSplice.

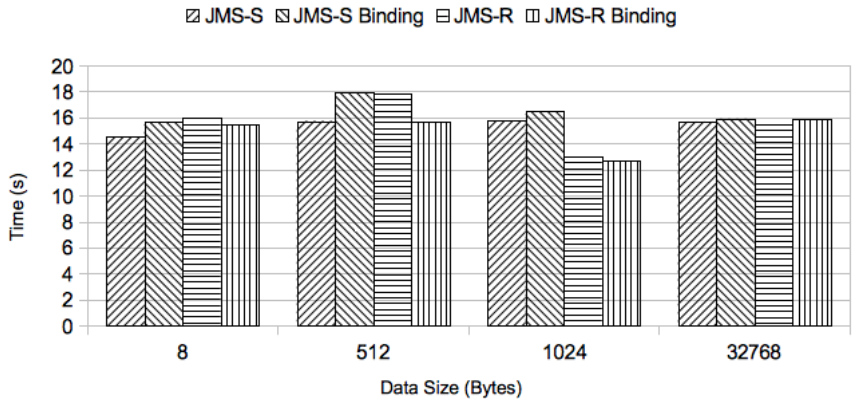


FIGURE 6.15 – **Temps d'envoi (-S) et de réception (-R) de 10 000 données via le protocole JMS et liaison R-MOM JMS** – Moyenne sur 50 sessions

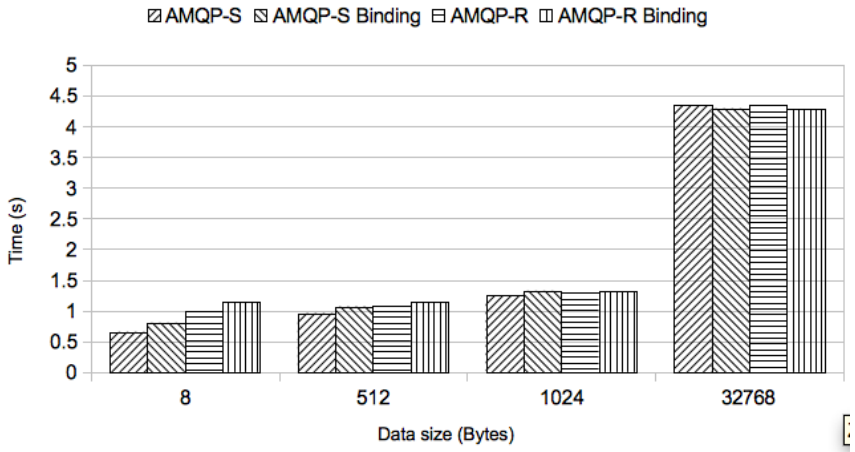


FIGURE 6.16 – **Temps d'envoi (-S) et de réception (-R) de 10 000 données via le protocole AMQP et liaison R-MOM AMQP** – Moyenne sur 50 sessions

Les figures 6.15 et 6.16 montrent respectivement les résultats des mesures, cette fois sur une échelle

de temps beaucoup plus grande que précédemment. Soit de 0 à $20s$ pour JMS et de 0 à $5s$ pour AMQP.

Tout comme les précédentes analyses, plus le traitement du protocole est long à s'exécuter, et plus la différence de temps de traitement entre ***(I)*** et ***(II)*** est faible. Sur une échelle de temps entre 10 et 50 fois supérieure que pour celle de DDS, il devient inutile d'analyser en détail les résultats observés. La seule grande remarque que l'on peut faire avec les résultats précédents est la présence du protocole TCP utilisé par JMS qui provoque une constance dans les temps de traitement quelque soit les tailles de données utilisées, même si on observe une variation des résultats de $5s$ entre la plus grande valeur (envoi de données de taille 512 octets avec R-MOM) et la plus petite (réception de données de taille $1ko$ avec R-MOM), qui est dû à un ϵ invisible sur une échelle de temps à la seconde près.

Conclusion

$\forall o \in \{\boldsymbol{E}, \boldsymbol{R}\}, \forall p \in \mathbb{P}, \forall i \in \mathbb{I}, \lim_{t \to \infty} \theta_{\boldsymbol{I}}(o,p,t) - \theta_{\boldsymbol{II}}(o,p,t,i) = 0.$

Le surcoût induit par l'utilisation de R-* est négligeable dans le contexte d'envoi et de réception de 10 000 données par rapport à l'utilisation directe des technologies embarquées.

6.4.5 Temps d'exécution des opérations d'adaptation

Cette sous-section s'intéresse au temps de traitement induit du déroulement d'opérations d'adaptation appliquées au niveau de l'intergiciel. Ainsi, on trouve la modifications de propriétés dynamiques ***(D)***, statiques ***(S)*** et aussi le changement de protocole d'envoi ou de réception ***(P)***.

Cette évaluation respecte la figure 6.5 et un calcul de moyenne sur cinquante exécutions du scénario. Cependant, la taille des données ne rentre pas en compte dans les opérations d'adaptation, donc les mesures ont profité de toutes les exécutions du scénario avec variation de taille pour augmenter le jeu de mesures. C'est à dire que les moyennes sont calculées sur les différentes tailles de données testées, multipliées par le nombre de sessions, soit $|\mathbb{T}| * 50 = 4 * 50 = 200$.

La moyenne calculée révèle la formule de la sous-section 6.4.2 :

(i) : $\forall o \in \mathbb{O}, \forall p \in \mathbb{P}, \forall t \in \mathbb{T}, \forall i \in \mathbb{I}, \theta_{\boldsymbol{II}}(o,p,i,t) - \theta_{\boldsymbol{I}}(o,p,t) = \tau + \phi(o,p,i)$

Table 6.1 – Temps d'exécution moyen (200 sessions) en μs pour des opérations de déploiement et de modifications de propriétés dynamiques et statiques sur des éléments d'envoi et de réception de données par technologie et par protocole

Protocol	Entity	Technology	DEPLOYMENT	DYNAMIC	STATIC
UDP	Sender	–	110	5	15
		R-MOM	140	20	30
	Receiver	–	220	5	20
		R-MOM	260	10	25
JMS	Sender	–	2198	942	960
		R-MOM	2212	970	969
	Receiver	–	3148	1049	994
		R-MOM	3347	1054	1004
AMQP	Sender	–	524	16	29
		R-MOM	719	17	33
	Receiver	–	2117	1	12
		R-MOM	2331	20	30
DDS	Sender	–	50	2	45
		R-MOM	76	20	62
		R-DDS	103	28	102
	Receiver	–	123	5	101
		R-MOM	155	17	135
		R-DDS	198	35	230

Vérifions cette dernière à l'aide des différents résultats notés dans le tableau 6.1.

Ce tableau présente les temps de traitement de déploiement et de modification de propriété dynamique et statique par entités d'envoi/de réception de données et par protocole de communication. Ainsi, ***(D)*** et ***(S)*** sont mesurées, mais le temps de déploiement remplace celui du changement de protocole ***(P)*** car le système ***(II)*** n'interrompt pas le flux de données. Dans ce cas, la mesure intéressante est la durée nécessaire avant de pouvoir profiter du nouveau protocole de communication puisque le système reste disponible.

D'une manière générale (même s'il s'agît d'une moyenne sur 200 mesures dans un environnement non déterministe), on retrouve la formule *(i)*. Les traitements métier de R-MOM sont plus simple que ceux de R-DDS qui fait intervenir un ensemble de 4 composants de contrôle par entité du système. C'est pourquoi, les mesures observées sur les entités provenant du modèle R-MOM sont plus rapides que celles du modèle R-DDS.

(D) et ***(S)*** montrent un surcoût maximal de $28\mu s$ pour R-MOM (modification d'une propriété dynamique sur une entité d'envoi de données JMS) et de $129\mu s$ pour R-DDS (modification d'une propriété statique sur une entité de réception de données DDS).

Par ailleurs, les temps de déploiement des entités de R-MOM et R-DDS sont plus importants que pour celles des protocoles sources. Ceci s'explique par le fait que cette phase comprend l'initialisation et le déploiement des composants FraSCAti, avec toute la complexité que l'on associe aux capacités réflexives (voir la sous-section 2.6.6). Il faudra compter entre 50 et 200 μs supplémentaire par rapport à la version non-adaptative.

De ce fait,

$\forall o \in \{\boldsymbol{D}, \boldsymbol{S}\}, \forall p \in \mathbb{P}, \forall t \in \mathbb{T}, \psi(o,p) \leq \phi(o,p,\ \textit{R-MOM}) \leq \phi(o,p,\ \textit{R-DDS})$

$\Rightarrow \theta_{\boldsymbol{I}}(o,p,t) < \theta_{\boldsymbol{II}}(o,p,\ \textit{R-MOM},t) < \theta_{\boldsymbol{II}}(o,p,\ \textit{R-DDS},t) < \psi(o,p,t) + 129\mu s$

et

$\theta_{\boldsymbol{I}}(\boldsymbol{P},p,t) > \theta_{\boldsymbol{II}}(\boldsymbol{P},p,t) = 0.$

Pour conclure sur l'exécution du système évalué, R-DDS et R-MOM ont révélé des performances tout à fait satisfaisantes par rapport à leurs homologues non-adaptatifs ou non-interopérables.

L'approche R-* concernant l'ajout de capacité d'adaptation ou d'interopérabilité ne nuit pas aux performances exigées par le système de systèmes TACTICOS, et permet même une disponibilité continue du système durant le changement de protocole.

6.5 Conclusion

L'évaluation réalisée dans ce chapitre a permis de mettre en évidence que dans le contexte du système de systèmes TACTICOS, l'approche R-* ajoute de manière transparente des capacités d'adaptation et d'interopérabilité pour les intergiciels orientés message, et ce, dès sa phase de configuration, jusqu'à son exécution.

La phase de configuration (section 6.3) est facilitée par un modèle à composant réflexif qui respecte les interfaces promues par les intergiciels visés, et un système d'environnement qui permet de définir dans une même logique, la configuration et les algorithmes d'adaptation d'un système distribué hétérogène.

La phase d'exécution du système (section 6.4) assure une flexibilité et une adaptation fine autour des fonctions du paradigme visé sans stopper le flux de données, en garantissant des performances à l'utilisation qui respectent les exigences du SoS, et une vue de la configuration du système cohérente avec son état.

Par ailleurs, FraSCAti ne proposant pas d'implantation pour l'embarqué ou pour le temps réel, nous n'avons pas poussé l'expérience dans ces domaines en mesurant par exemple l'empreinte mémoire utilisée, ou en utilisant un système d'exploitation temps-réel. Il y a des chances qu'il ne soit donc pas utilisable dans des conditions réelles, mais cela reste du domaine des perspectives de test.

L'expérience a montré d'excellents résultats dans un contexte hétérogène et non-déterministe (sous-section 6.3.2), pour les deux défis adressés par cette thèse (sous-section 1.3).

Troisième partie

Conclusion et Perspectives

Chapitre 7

Conclusion et Perspectives

Sommaire

7.1 Résolutions de défis d'évolution des systèmes de systèmes dans un contexte industrielo-académique

Le sujet "Réflexivité au service de l'Évolution des Systèmes de Systèmes" a trouvé une solution dans l'approche R-* qui s'intéresse tout particulièrement à l'ajout de la réflexivité pour le support de l'évolution dans un système distribué et hétérogène.

Pour ce faire, il a fallut dans un premier temps identifier les deux principaux défis (section 1.3, page 5) qui rendent l'approche R-* difficile au premier abord. Soient où (sous-section 1.3.1, page 5) et comment (sous-section 1.3.2, page 5) appliquer la réflexivité, qui par hypothèse, peuvent trouver une réponse générique dans l'identification de points de variabilité disponibles sous la forme de fonctions, et transformation de ces fonctions dans des modèles réflexifs en sur-couche de technologies existantes.

La réponse s'est battie dans un premier temps grâce à un encadrement fort d'expérience et d'expertise dans des domaines connexes à l'évolution des systèmes distribués.

Le contexte industriel de cette thèse a invité à considérer le cadre d'étude des systèmes de systèmes (SoS), en profitant d'une expertise THALES dans le domaine du temps réel et de l'embarqué avec notamment une technologie CCM (sous-section 2.6.1, page 40), utilisée dans des systèmes ferroviaires ou navals par exemple.

Le cadre académique a proposé de démarrer sur les travaux de l'équipe projet ADAM/Inria, liés à l'évolution et l'adaptation logicielle, dont le canevas FraSCAti (sous-section 2.6.6, page 45) garantit un environnement de développement et d'expérimentation exhaustif dans ces thématique de recherche.

L'axe de recherche poussé fut la faisabilité de l'hypothèse de réponse aux défis de R-* dans des systèmes de systèmes aussi bien à bas-niveau d'un système avec l'ajout de la réflexivité dans des technologies de

communication avec les contributions R-DDS (chapitre 3, page 55) et R-MOM (chapitre 3, page 55), qu'à haut niveau avec l'ajout de la réflexivité pour la configuration et la spécification d'un système distribué et hétérogène avec la contribution R-EMS (chapitre 5, page 83). Finalement, une évaluation (chapitre 6, page 109) a servi d'élément de preuve supplémentaire pour la faisabilité de l'approche R-*.

Voici en détail le bilan de l'approche R-* pour chaque contribution.

7.1.1 R-DDS

R-DDS (chapitre 3, page 55) est une architecture qui a étendu la spécification DDS en ajoutant des capacités de reconfiguration à chaud. Lui permettant de spécifier précisément le comportement attendu, et en ajoutant une dynamique de prise en charge des qualités de services, tout en respectant l'API originelle.

La séparation du fonctionnel et du non-fonctionnel permet également de personnaliser le moteur DDS utilisé dans les composants de l'architecture. Se faisant, des règles de cohérence viennent justifier l'implémentation qui ne respecte jamais totalement la spécification.

Complétée d'un travail d'intégration conséquent d'un moteur d'exécution dans l'architecture R-DDS, le respect de la spécification et l'apport de capacités réflexives apportent une touche technique conséquente vis à vis des possibilités d'auto-adaptation à chaud de la solution.

Tout en apportant un surcout d'exécution négligeable par rapport à l'utilisation d'un moteur DDS embarqué.

7.1.2 R-MOM

La solution haut-niveau de préoccupations du paradigme MOM est l'architecture R-MOM (chapitre 4, page 67). Cette solution, à l'instar de R-DDS, propose de personnaliser toute entité de production ou de consommation de données, et d'utiliser n'importe quelle technologie orientée message. L'interopérabilité entre intergiciels est assurée, ainsi qu'un effort tout particulier sur l'intégration de cette architecture dans les systèmes existants, mais surtout la possibilité de personnaliser et de reconfigurer très finement les besoins techniques en terme de fonctionnalités MOM.

Il se montre tout à fait pertinent et novateur sur tous les points adressés et nécessaire pour un intergiciel orienté message face à un environnement dynamique, hormis une évaluation non-réalisée dans un contexte de domaine RT-E.

7.1.3 R-EMS

R-EMS (chapitre 5, page 83) est la contribution qui adresse la réflexivité au niveau d'un système distribué, avec les conséquences que la gestion d'un environnement réflexif peut offrir, comme la formalisation d'une spécification, l'unification d'une configuration et du raisonnement sur des besoins d'adaptations. Il offre un modèle exécutable et extensible d'unification des exigences pour les systèmes de systèmes, et un support dynamique d'exigences, avec des propriétés de reconfiguration, de raisonnement, d'aide à la décision, et de traçabilité des activités. La solution est non-intrusive pour les systèmes existants, accessible par toute ressource capable d'utiliser les paradigmes MOM et RPC et complètement distribuée pour parfaire des zones physiques et logiques de responsabilités du modèles, rendant service à la fois à la conception du système, mais aussi à la sécurité de systèmes partitionnés.

Le langage de programmation d'environnement R-EML est également disponible et met en avant des patrons de conception ou d'exécution nécessaires dans des problématiques de programmation de langages très haut niveau, distribués et à accès concurrents.

7.1.4 Difficultés rencontrées et non résolues

L'évaluation réalisée ne s'est pas faite sur un environnement d'exécution temps-réel ou embarqué. Il en résulte que de tels domaines n'ont pas encore prouvé la faisabilité de R-* dans des environnements fortement contraints. De ce fait, le cadre SoS proposé par THALES fut attaqué en supportant des préoccupations de personnalisation fine des fonctions des intergiciels (solutions à la carte et la moins gourmande possible en espace mémoire).

La validation ne comprend que des environnement faiblement contraints tels que les systèmes d'information.

Par ailleurs, R-EMS a démarré depuis le début, mais son prototypage n'est pas aussi exhaustif que le potentiel montré par les différentes architectures des contributions. Le développement furt spécifique et limité au scénario d'évaluation comprenant seulement quatre acteurs, et quelques opérations d'adaptation pré-programmées. L'évaluation de R-EMS n'a consisté qu'à vérifier que le peu d'éléments modélisés réagissaient comme il le fallait durant la conception et l'exécution des acteurs (configuration et exécution des adaptations des intergiciels).

7.1.5 Bilan

Hormis le manque de preuve de la faisabilité de R-* dans les domaines RT-E, cette thèse a réussi à démontrer qu'un système hétérogène et distribué, comme le sont les systèmes de systèmes, avait nécessairement besoin d'interopérabilité, de réflexivité et d'un système d'environnement transverse pour pouvoir s'adapter le plus efficacement et le plus rapidement possible à de nouvelles exigences.

Cependant, il reste encore des aspects précis à résoudre dans ces propriétés, et c'est pourquoi un grand nombre de perspectives apparaissent et sont discutées dans la section suivante.

7.2 Perspectives

Les perspectives proposées sont toutes relatives à des outils utilisés au cours de l'étude et appropriées à R-*, ou spécifiques aux réponses apportées.

7.2.1 FraSCAti

Temps-réel et embarqué

FraSCAti fut le modèle à composant utilisé comme brique de base de tout travail d'implémentation de l'approche R-*. Cependant, même s'il s'est démarqué des autres modèles à composants (sous-section 2.6.7, page 46), la thèse ne connaît pas de travaux prouvant la faisabilité de son utilisation dans le monde de l'embarqué ou du temps réel, pourtant fortement sollicité par les systèmes de systèmes.

C'est pourquoi la première perspective à court terme est de trouver soit un modèle à composant réflexif au moins aussi agile que FraSCAti pour le domaine de l'embarqué et le temps-réel, soit de prouver la faisabilité de FraSCAti dans les domaines RT-E.

Et vérifier à nouveau qu'avec un tel modèle à composant, et dans un tel domaine de système, R-* peut toujours s'appliquer, et dans quelles limites de contraintes.

Langage de description d'architecture Fractal+SCA

Une faible perspective serait d'ajouter une propriété paramétrée par le nom d'un composant, au niveau d'une configuration de composite SCA, de manière à résoudre la relation de partage de composant chère à Fractal (sous-section 2.6.4, page 43), et nécessaire à l'architecture R-DDS. De plus, inviter l'organisme de standardisation OASIS à permettre de définir plus d'un niveau de profondeur dans l'ADL des composites SCA, et ne pas imposer le format un fichier par composite.

7.2.2 R-DDS

Interoperabilité entre moteurs d'exécution DDS

L'interopérabité est nécessaire dans la vision R-* pour les systèmes auto-adaptatifs et hétérogènes. Cependant, elle est proposée par le biais de la spécification DDS-I [OMG, 2009c].

Comme toute spécification, elle engage un travail de la part des moteurs d'exécution de la respecter, sous peine de ne pas voir se réaliser une telle fonctionnalité devenue nécessaire.

Des tests ont montré une volonté à la respecter, et ont conduit à un succès d'échange d'informations enter moteurs d'exécution. Cependant, comme toutes les phases d'évolution logicielle, rien n'assure que

les prochaines versions supporteront toujours cette capacité d'interopérabilité, assurée par une ancienne version, mais incompatible avec une nouvelle.

Il pourrait être intéressant de voir s'il est possible d'héberger derrière une entité composite, un ensemble d'entités techniques provenant de divers moteurs d'exécution. Assurant ainsi, malgré les choix d'implantation des moteurs d'exécution, qu'une interopérabilité deviendrait possible en se servant de l'architecture R-DDS comme d'un modèle pivot. Cette solution demanderait à prendre en compte des traitements complexes apportés par cette pluralité des technologies DDS derrière l'interface R-DDS.

Évolution des types de données

Une autre perspective qui semble plus importante est l'évolution des types de données, ou *TypeSupport*. En effet, la spécification stipule qu'il est possible d'enregistrer au moins un *TypeSupport* dans le système distribué afin de pouvoir créer des sujets relatifs au type défini. Mais une fois créé, il n'y a pas de possibilités spécifiées pour supprimer un *TypeSupport*, ni en modifier un existant. Cette notion partagée sur tout le système mérite de pouvoir être modifiée car sinon, il incombe aux utilisateurs de gérer les versions de type de données de manière distribuée. Deux solutions se proposent actuellement. Soit déléguer à R-EMS la gestion des versions des *TypeSupport*, ce qui aura pour effet d'appliquer des reconfigurations importantes sur tous les nœuds du système. Soit gérer localement un *TypeSupport* dynamique qui à la manière d'un adaptateur, fera le lien entre les types visés dont le nom sera la concaténation du type et de la version à utiliser, et donc qui contiendra un tableau de *TypeSupport*s à offrir en fonction de la version à utiliser. Ainsi, ce *TypeSupport* dynamique offrira de manière transparente le même type, relatif à une version qui intéresse les préoccupations d'évolution du système.

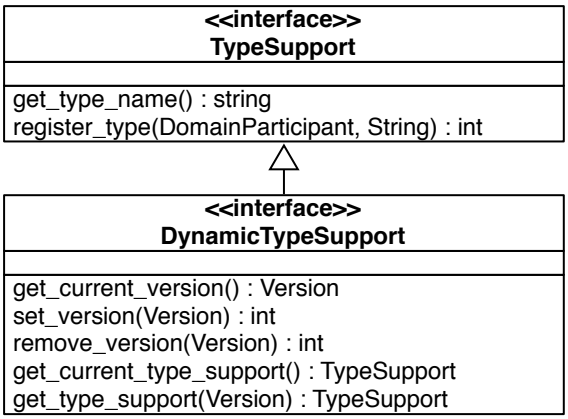


FIGURE 7.1 – **Diagramme de classe du DynamicTypeSupport** – Héritage avec le TypeSupport

La figure 7.1 représente ce *TypeSupport* dynamique comme une interface héritant de l'interface *TypeSupport* de manière à ce que l'extension soit transparente pour d'anciens systèmes ne connaissant que le moyen de la spécification.

7.2.3 R-MOM

Couche d'évènement locale en support au traitement des envelopes

Une perspective à long terme pourrait être de développer un système d'évènements local à un assemblage de composants pour automatiser des réactions entre composants suite aux traitements des *Envelope*s.

Interopérabilité avec d'autres protocoles de communication asynchrones

Une autre perspective serait plus dans une vision d'interopérabilité avec d'autres protocoles que des protocoles de type MOM. Par exemple, voir s'il est possible d'utiliser R-MOM pour concevoir des applications de type protocole rumeur (*Gossip* en anglais) [Lin et al., 2011] qui offre une excellente alternative pour le partage de modèles distribués dans de très grands systèmes évolutifs, et qui pourrait devenir nécessaire dans des configurations particulières de systèmes de systèmes.

7.2.4 R-EMS

Même si R-EMS a été réfléchi dès le début de la thèse, il avait besoin des autres travaux et de tout un état de l'art pour voir le jour ne serait-ce que sur papier, vis à vis d'une possible concrétisation de l'idée.

Aujourd'hui, nombre de portes ont été enfoncées pour offrir la gestion d'un environnement, telles que le modèle exécutable, et un langage très haut niveau qui facilite l'utilisation d'un tel modèle. Mais il reste encore beaucoup à faire.

Évaluation et formalisation des règles de cohérence du modèle

Premièrement, réaliser des évaluations poussées sur le langage d'exécution d'environnement. Ainsi que formaliser les règles de cohérence de modélisation du modèle vis à vis du mélange entre relations d'héritage, de compatibilité, d'équivalence et de participation.

Modèle transactionnel et réflexivité comportementale

Un autre exemple est le support du mode transactionnel et l'exécution exclusive qui sont deux pans qu'il faut résoudre le plus rapidement possible pour assurer des travaux collaboratifs et automatiser les tâches d'adaptation concurrentes dans le modèle.

Raisonnement de la spécification avec les technologies existantes

Ensuite, offrir la possibilité au système de choisir les technologies qui seront capables de répondre au mieux à des exigences attendues et propres aux approches SPL (sous-section 2.3.1, page 21 et le travail de Russel Nzekwa [Nzekwa, 2010] pour avoir une plateforme intégralement réflexive). Manquant de connaissance dans ce domaine, cette étude ne peut proposer de premier axe de recherche, mais il s'agît là encore d'une fonction nécessaire pour aider à la décision des choix technologiques à utiliser, et peut-être par la suite, automatiser cette tâche.

Vision globale d'une gestion de modèle dynamique et distribué

À long terme, il faudra réfléchir à ce qu'entraîne la gestion d'un modèle dynamique et distribué sur sa cohérence globale, pour pouvoir prévenir de manière automatique d'erreurs éventuelles induites d'une certaine modification. Cette prévention doit être réalisée sur de simples patrons de conception pour pouvoir garantir une certaine dynamique dans leur exécution.

Prise en compte de l'utilisateur final

Même si le modèle s'est voulu agnostique d'une quelconque représentation, s'intéresser aux problématiques de l'utilisateur final est essentiel dans un système qui souhaite mélanger des préoccupations de différents domaines d'activité (techniques, politiques, économiques, etc.). Ainsi, il faut pouvoir développer d'autres langages dédiés que R-EML qui ne s'adresse qu'aux développeurs en informatique, des points de vue du modèle et des interfaces utilisateur dans différents domaines pour vérifier qu'un tel système est exploitable par une hétérogénérité de domaines d'activité.

Quatrième partie

Annexes

Annexe A

Liaison DDS pour SCA

Sommaire

A.1 Introduction

La liaison de communication DDS pour SCA vise à profiter du paradigme DDS pour la gestion des évènements SCA [Beisiegel et al., 2009].

La sous-section 2.6.5, page 45 explique brièvement la gestion des évènements dans une architecture SCA. Pour résumer, il est possible de configurer l'émission d'évènements en s'abstrayant de comment ils sont réalisés. Le protocole comprend cinq grand concepts, c'est à dire les consommateurs et les producteurs d'évènements, les puits ou chaînes d'évènements, et les liens d'interaction et de promotion.

La figure A.1 montre un exemple d'architecture SCA utilisant des évènements. Dans cette figure, il est à noter qu'un lien de promotion relie deux même types d'éléments, et le lien d'intéraction se trace depuis un producteur vers une chaîne, ou depuis une chaîne vers un consommateur. Le modèle résultant mériterait d'être simplifié en permettant par exemple de lier directement un producteur vers un consommateur, mais cela n'a pas été proposé dans la dernière spécification, et ce, sans argument justificatif.

La principale difficulté rencontrée est que les deux paradigmes sont différents par nature. En effet, DDS (sous-section 2.5.6, page 32) est orienté donné, et offre un modèle objet, et non composant pour envoyer et recevoir des données. Le modèle SCA profite de la relation de composition entre composants pour isoler les flux d'évènements, il utilise les relations d'intéraction et de promotion pour bien définir les parcours de transmission et de plus, il agrège le comportement de plusieurs producteurs ou consommateurs avec la relation de promotion de capture ou d'émission d'évènements.

L'objectif est donc d'établir une correspondance fonctionnelle entre les concepts SCA et DDS, de résoudre l'isolation des flux de données dans un composant avec l'extérieur, et permettre la configuration du parcours des données dans un modèle DDS.

A.2 Correspondance fonctionnelle entre concepts SCA et DDS

Il est possible d'effectuer une correspondance limitée des fonctions entre les concepts DDS et les concepts SCA.

Considérons les concepts suivants internes à un composite : les consommateurs et les producteurs d'évènements SCA sont par fonction respectivement similaires à des entités de lecture et d'écriture de

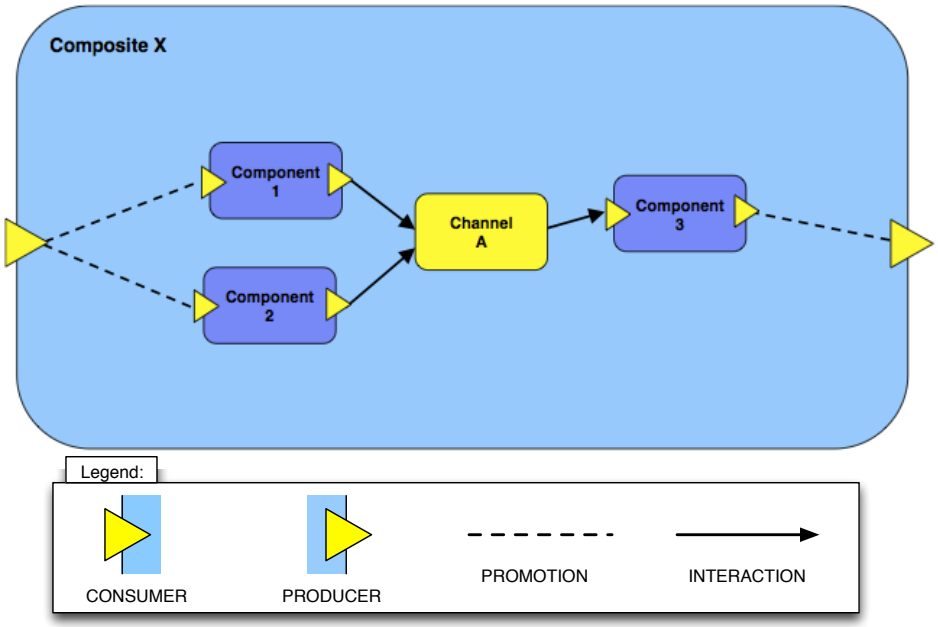


FIGURE A.1 – **Exemple d'architecture d'évènements SCA** – Promotion de consommateurs de de producteurs (source : [Beisiegel et al., 2009])

données DDS. Une chaîne d'évènements SCA représente un point d'accès à un flux de données, c'est à dire dans le modèle DDS, à un domaine ou à une partition. Le filtre d'évènements SCA correspond à l'objet DDS *ContentFilteredTopic*. Finalement, même si DDS n'est pas spécifique à l'échange d'évènements, la spécification DDS4LwCCM de l'OMG [OMG, 2009b] autour de l'intégration de DDS dans le modèle à composant LwCCM a définit le profile DDS qui propose un ensemble de valeurs de QoS pour que les données DDS se comportent comme des évènements.

Le tableau A.1 montre les correspondances fonctionnelles entre concepts des modèles SCA et DDS.

TABLE A.1 – Tableau de correspondance entre concepts d'évènements pour SCA et DDS

SCA	DDS
Consommateur	Lecteur de donnée
Producteur	Écrivain de donnée
Chaîne	Domaine ou Partition
Filtre	Sujet filtré de contenu
Évènement	Profile d'évènement [OMG, 2009b]
Type d'évènement	Support de type

A.3 Isolation des flux de données dans le modèle DDS

L'isolation entre niveaux d'imbrication de composants SCA doit utiliser l'isolation de flux de données entre entités DDS. Pour se faire, deux solutions sont offertes, c'est à dire l'utilisation de domaines ou de partitions. Dans les deux cas, il faudra générer un nom unique par espace d'isolation de données DDS. La solution de résolution de nom unique par relation de composition entre composants est infaisable avec la

relation de composant partagé comme le propose la plateforme FraSCAti, car dans ce cas précis, il peut y avoir plusieurs composants parents par composant, et donc plusieurs noms possibles.

Cependant, le fait que la partition DDS ne soit qu'un simple champ texte facilite sa (re)configuration. Malheureusement, DDS soutient que la partition est une isolation publique, c'est à dire que tout le monde peut envoyer ou recevoir des données sur la partition, une fois celle-ci connue. Ce soucis doit être résolu dans l'implantation des liaisons DDS pour SCA, en vérifiant qu'une donnée provient bien d'un élément interne du composite qui héberge la liaison, et non extérieur au composite.

A.4 Mise en œuvre du routage des évènements dans la transformation SCA vers DDS

Une fois la correspondance fonctionnelle entre concepts SCA et DDS et l'isolation des flux d'évènements dans un composite SCA étudié, il faut résoudre les moyens techniques pour mettre en œuvre les relations de promotion et d'interaction entre producteurs, chaînes et consommateurs d'évènements SCA.

Le routage des données n'est pas un comportement attendu par DDS. La plupart des systèmes l'utilisant se réfèrent à un bus pensé orthogonal aux acteurs [58]. Alors que cette section s'intéresse à des points d'entrée et de sortie d'évènements via les chaînes, les producteurs et les consommateurs de composite, l'objectif va être de transformer le mode de distribution par bus de données DDS en un graphe de flux de données.

Au moins deux solutions ont été pensées et étudiées pour réaliser la projection attendue :

(i) les éléments portés par un composite (chaînes, producteurs et consommateurs d'un composite) sont considérés comme des éléments logiques de l'architecture, et dans ce cas, seuls les éléments propres à un composant métier sont transformés dans le modèle DDS pour être exécutés.

(ii) chaque élément porté par un composite doit assurer une fonction bien précise à l'exécution, c'est à dire, assurer une isolation des comportements et des données transitant à l'extérieur et à l'intérieur d'un composite.

La solution ***(i)*** s'appuie sur de l'agrégation de comportement des éléments portés par un composant (et non un composite), issue d'arborescence d'informations déduites des relations de promotion et d'intéraction partant d'un même élément. Dans ce cas précis, il faut analyser les informations à agréger, et assurer la cohérence de l'aggrégation à tout instant. Les informations SCA sont les flux de données, et les filtres pour les consommateurs d'évènements. L'objet *ConsistencyChecker* sera utilisé sur chacune des transformations des éléments SCA dans le modèle DDS pour maintenir une cohérence des exigences du système.

La solution ***(ii)*** est plus simple à comprendre puisqu'il s'agît de transformer tout élément de l'architecture SCA dans le modèle DDS. Solution plus facile à écrire et à maintenir à l'exécution du système dynamique, mais beaucoup plus gourmande en mémoire que la solution **(i)**.

Dans les cas ***(i)*** et ***(ii)***, chaque transformation comprendra au minimum le déploiement d'objets DDS, soit d'un *DomainParticipant* et un *Subscriber*, ou un *Publisher* par chaîne de routage d'évènement, si l'élément à transformer est respectivement un consommateur ou un producteur d'évènements. De plus, nous suivons la logique de parcours par ordre d'identification de cible, c'est à dire que l'isolation des flux de données va considérer des partitions DDS définies sur les éléments de consommation de données. Ainsi, chaque élément de consommation va devoir créer une unique partition qui sera à utiliser par chaque élément de production. Pour rappel, une partition est portée par un *Subscriber* ou un *Publisher* (voir sous-section 2.5.6, page 32). Finalement, pour chaque *Subscriber* nécessaire, il faudra déployer autant de *DataReader*s qu'il y a de types d'évènements pouvant être lus sur la chaine de routage associée. Le même procédé est appliqué pour un *Publisher* et des *DataWriter*s nécessaires par type d'évènement pouvant être produits.

La solution ***(i)*** va devoir parcourir le modèle SCA pour identifier depuis chaque élément à transformer, des arbres où les nœuds sont les éléments de composite, les liens sont les relations de promotion et d'interaction, et les feuilles seront des consommateurs et des producteurs portés par des composants non

58. Voir la figure 6.2 qui superpose les types d'acteur du système TACTICOS au dessus d'un bus DDS

composite. Les producteurs devront utiliser un couple (*Publisher*, *DataWriter*) par consommateur, et y identifieront les partitions à adresser. Les consommateurs déploieront un couple (*Subscriber*, *DataReader*) par producteur de l'arbre, dont la chaine de parcours définira le filtre à utiliser comme "et" logique de tous les filtres parcourus. Les modes de réception par "poussée" et par "tirage" (voir sous-section 2.5.4, page 31) seront respectivement implantés par un seul *DataReaderListener* utilisé par tous les *DataReader*s, et un seul *WaitSet* associé aux *ConditionListener*s de tous les *DataReader*s (voir la sous-section 2.5.6, page 32 pour les modes de "poussée" et de "tirage" de données dans DDS).

La solution ***(ii)*** va faciliter les opérations de parcours du modèle à composant SCA en considérant des arbres de hauteur 1 depuis tout élément à transformer, c'est à dire depuis tous les consommateurs, producteurs et chaînes d'évènements. Donc il n'y aura pas de composition de filtre autre que ceux qui sont définis dans un modèle SCA, et tous les éléments portés par un composite comprendront des *DataReader*s et des *DataWriter*s pour bien isoler les données entrantes et sortantes (d'un composite ou d'une chaîne). Ainsi chaque donnée reçue par un *DataReader* sera automatiquement envoyé à tous les *DataWriter*s.

Pour résumer, la solution ***(i)*** nécessite moins de mémoire, et est plus rapide que la solution ***(ii)***, puisqu'elle déploie moins d'objets DDS, et ne fait pas de réception/écriture de données intermédiaires pour chaque chaîne ou chaque relation de promotion. La maintenance de la solution ***(i)*** est plus compliquée mais pas impossible (soit par parcours du modèle SCA, soit par sauvegarde d'informations contextuelles pour identifier quelles sont les partitions à utiliser, ou les parties d'un filtre à modifier). Même si un micro-benchmark n'a pu être réalisé pour comparer les deux solutions, il apparaît que l'étude met en avant la solution ***(i)***.

A.5 Conclusion

La liaison de communication DDS pour SCA étudiée dans ce chapitre est possible même si le modèle DDS n'est pas orienté composant. Toutefois, la seule difficulté non résolue est l'assurance de l'isolation des flux d'évènements dans un composant, dont la fonction n'est pas assurée par les partitions DDS, mais deux réponses dans ce chapitre considèrent la propriété d'isolation respectivement comme une fonction logique ou physique.

La voix logique déploie un minimum d'objets DDS, pour un résultat plus rapide des échanges entre composants (pas de traitements intermédiaires), mais une maintenance du système plus compliquée si le modèle à composant doit être modifié. Par ailleurs, la solution ***(i)*** semble en théorie plus efficace en terme de performance et de maintenance, mais hélas, il n'y a pas eu d'implémentation des solutions, et donc pas de comparaison en terme de vitesse d'utilisation et de reconfiguration des deux solutions.

Annexe B

Détail d'implémentation de R-EMS

Sommaire

B.1 Diagramme du méta-modèle R-EM3

Le diagramme du méta-modèle R-EM3 a été modélisé à partir d'extensions (*plugins* en anglais) EMF de l'environnement eclipse. Afin de s'adapter à l'outil, certains concepts ont été ajouté, afin notamment de faciliter la conception de règles de cohérence du méta-modèle (exemple d'héritage entre relation).

Afin de rendre le tout plus facile d'accès, le méta-modèle a a été découpé en *paquets* Ecore, qui correspondent aux modules de la solution R-EMS.

B.1.1 Module racine

Le module racine représenté par la figure B.1 contient le concept d'élément ou *Element*. Et ajoute les concepts de modèle ou *Model* et d'élément nommé, ou *NamedElement* pour faciliter la modélisation Ecore de modèles R-EM2.

L'élément propose un ensemble de relations qui permettent de faire le lien entre sa description structurelle et sémantique.

Le modèle correspond à l'élément racine (*Root*) de l'instantiation Ecore d'un modèle R-EM2.

Finalement, l'élément nommé est un raccourci d'écriture dans la modélisation d'éléments uniques dans un ensemble identifiés par leur nom.

Les autres paquets de la figure sont décrits ci-dessous.

B.1.2 Module de domaine

Le module de domaine représenté dans la figure B.2 permet de modéliser les concepts de Domaine ou *Domain* et d'Importations de ressources, ou *Import*.

La différence avec la solution proposée est l'héritage entre un domaine et l'élément nommé. Ainsi, qu'un champ texte dans l'importation.

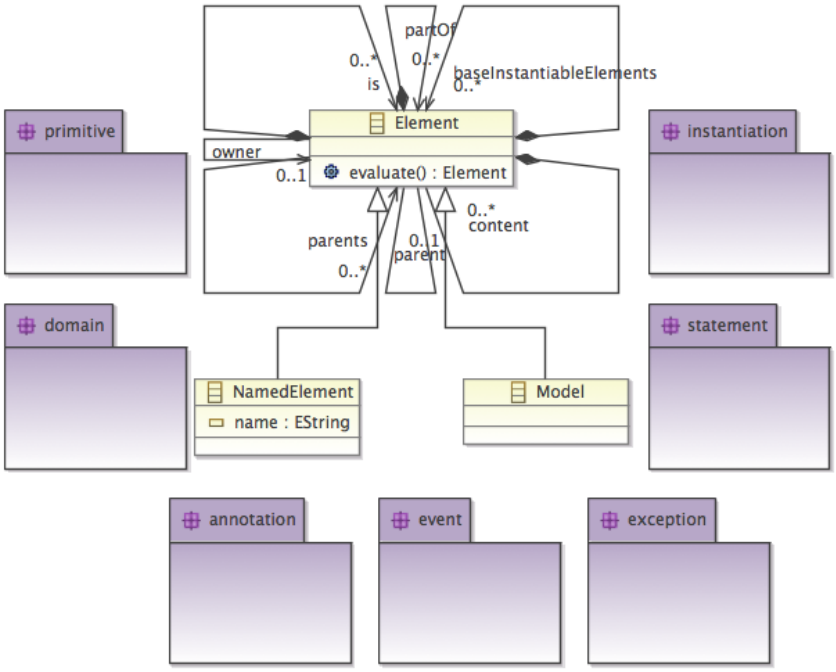


FIGURE B.1 – **Module racine** – *Element*, *Model*, *NamedElement* et *Packages*

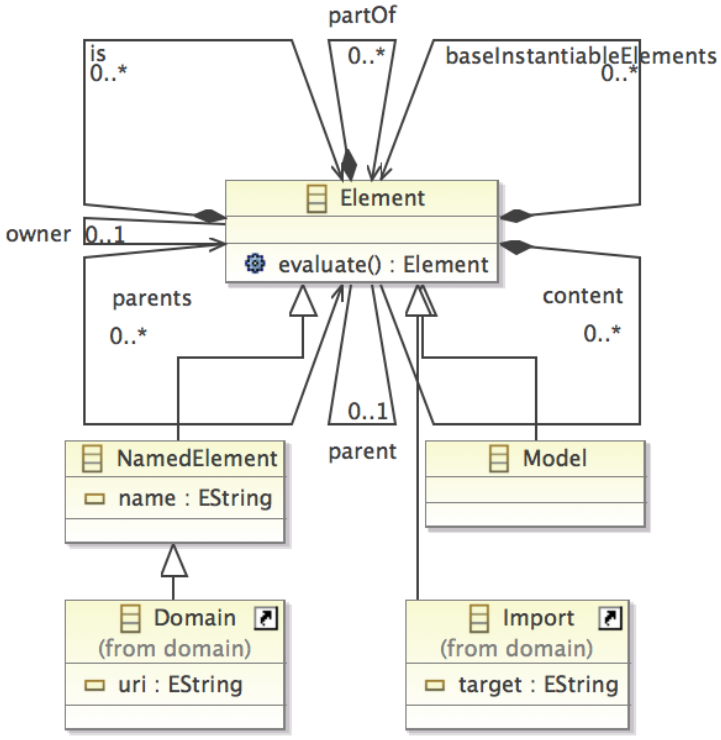


FIGURE B.2 – **Module de domaine** – *Domain* et *Import*

B.1.3 Module d'annotation

Le module d'annotation représenté par la figure B.3 montre l'héritage entre une annotation, ou *Annotation* et une invocation, ou *Invocation*, provenant du module d'instantiation (sous-section B.1.6).

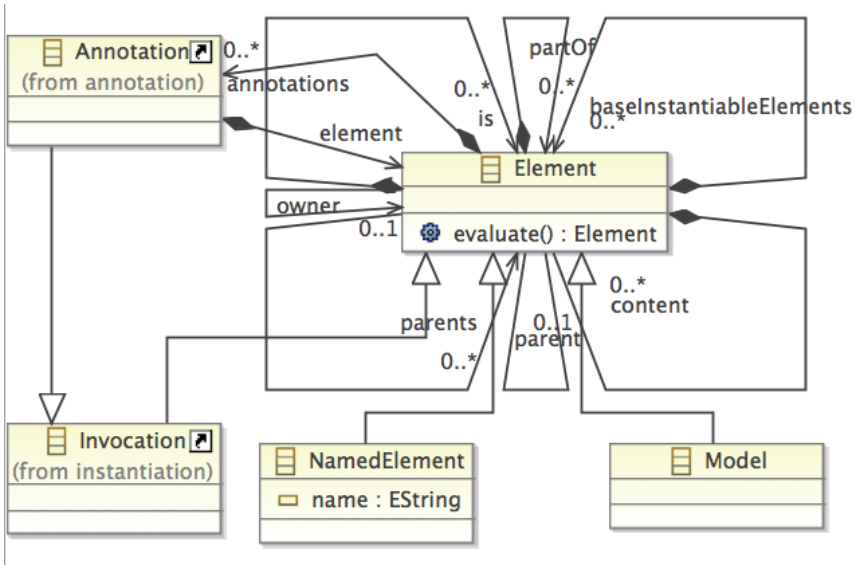


FIGURE B.3 – **Module d'annotation** – *Annotation* inherits from *Invocation*

B.1.4 Module des éléments primitifs

Le module des éléments primitifs, ou *Primitive*, représenté dans la figure B.4, montre un héritage entre le concept *Primitive* et le concept *Statement* issu du module d'instruction (sous-section B.1.5).

L'élément *Primitive* conserve un champ texte qui correspond à sa valeur uniformisée quelque soit le type d'élément primitif visé par la future modélisation.

De plus, le concept d'intervalle, ou *Interval*, est introduit en considérant une borne inférieure et supérieure de type *Statement*.

B.1.5 Module des instructions

Le module d'instructions, ou *Statement*, représenté dans la figure B.5, n'apporte pas d'instruction supplémentaire par rapport à R-EM3.

B.1.6 Module des éléments d'instantiation

Le module d'instantiation, représenté dans la figure B.6, utilise le concept de *NamedElement* pour introduire le concept d'élément instantiable, ou *InstantiableElement* et de paramètre d'opération, ou *Parameter*.

B.1.7 Module des évènements et des exceptions

Les modules d'évènements et des exceptions sont représentés dans la figure B.7. Tous les concepts liés à ces modules héritent du concept de *Statement*.

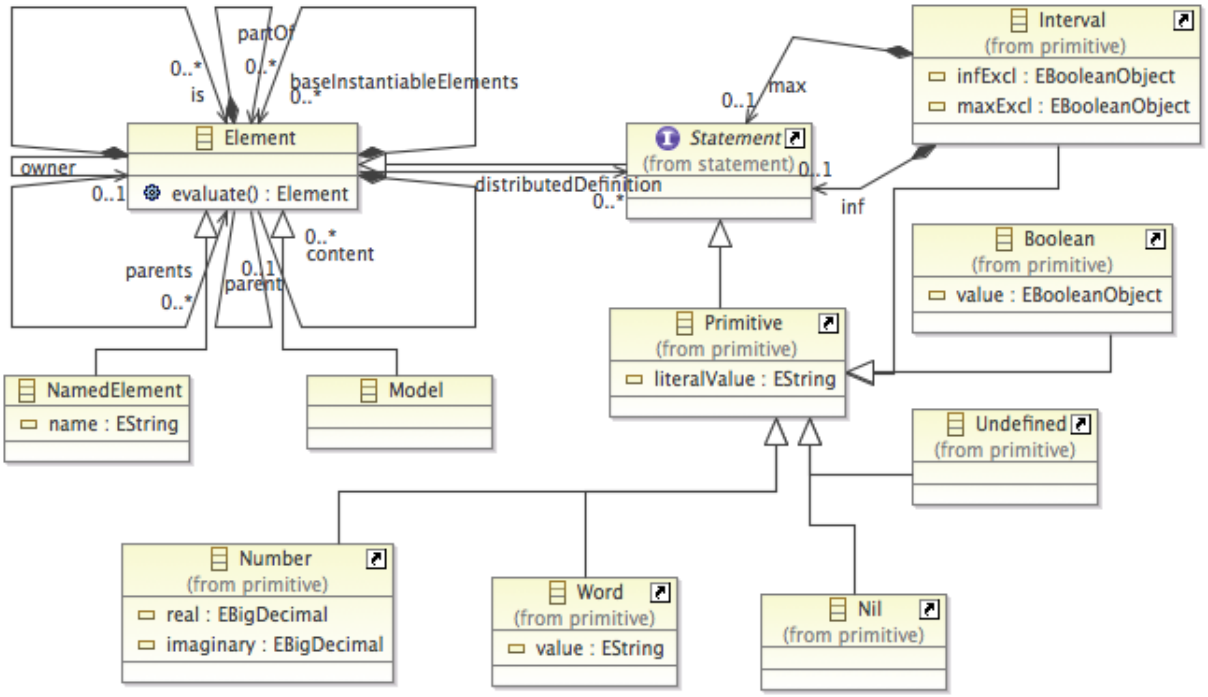


FIGURE B.4 – **Module des éléments primitifs** – *Primitive* inherits from *Statement*

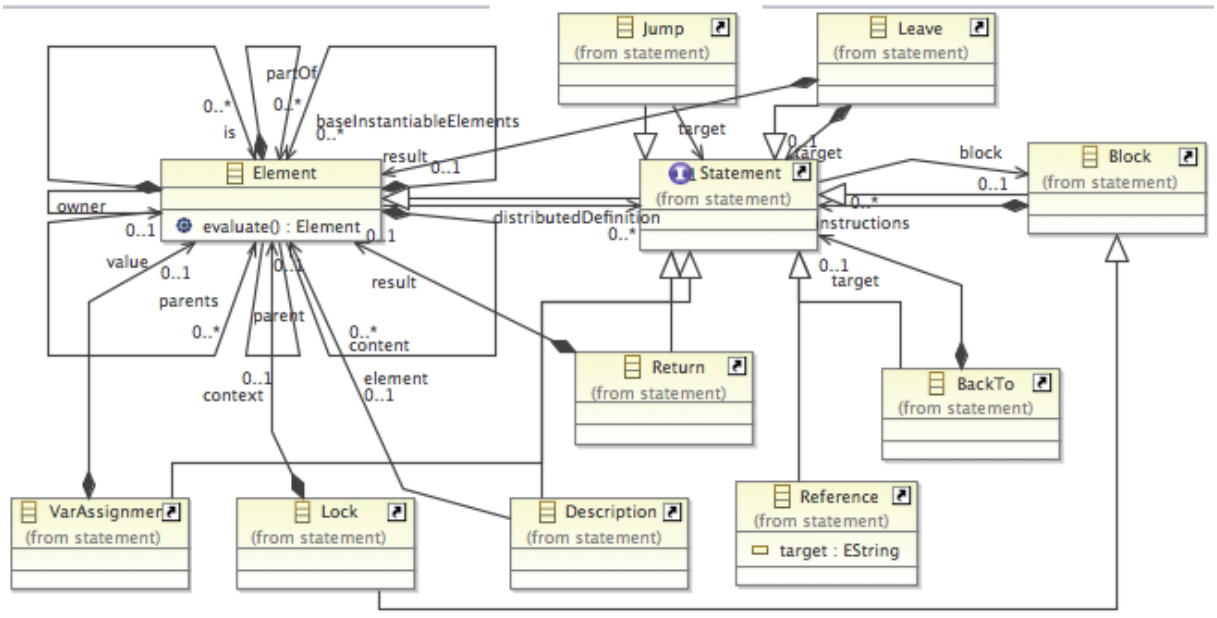


FIGURE B.5 – **Module des instructions** – *Statement* inherits from *Element*

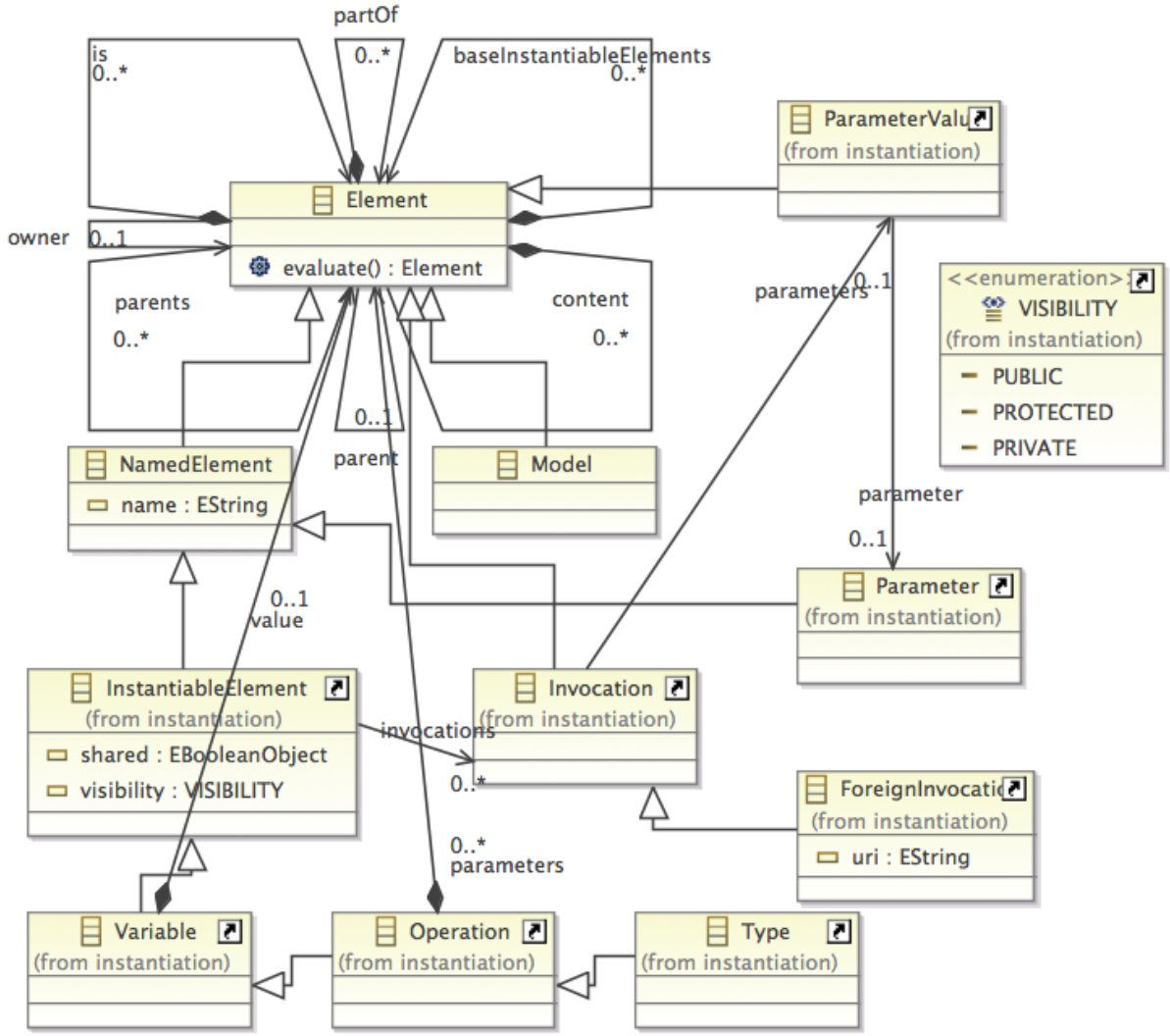


FIGURE B.6 – **Module des éléments d'instantiation** – *InstantiatedElement* inherits from *NamedElement*

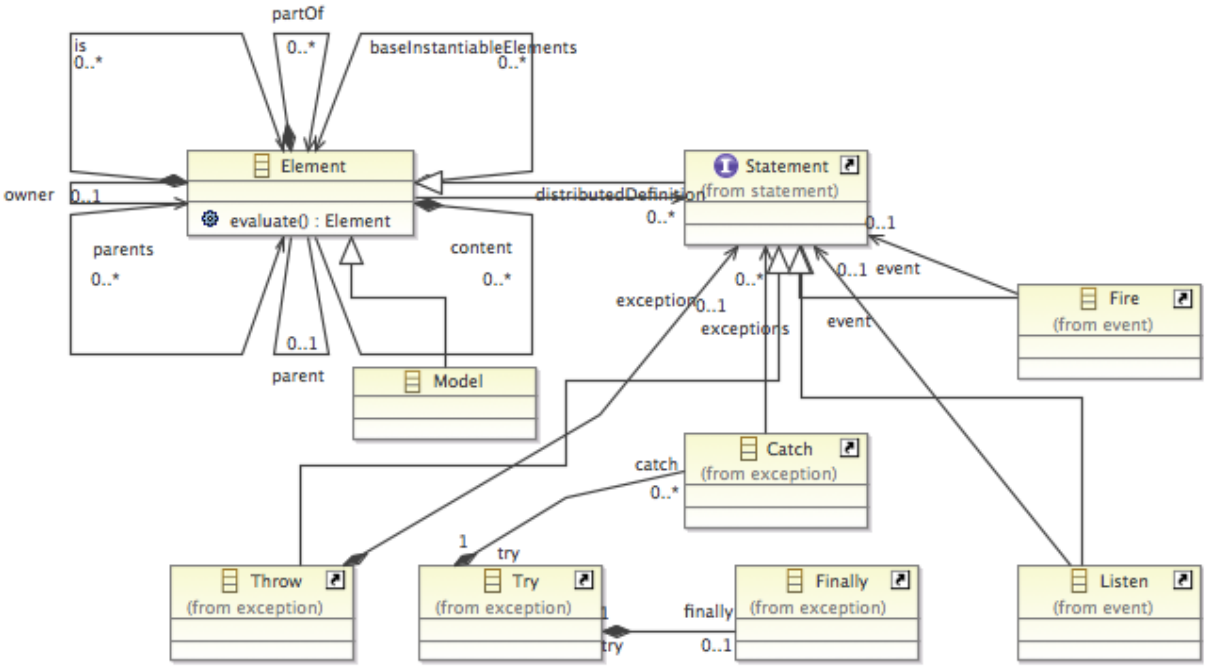


FIGURE B.7 – **Module des évènements et des exceptions** – *Exception* and *Event* operations inherits from *Statement*

B.2 Fichier des règles syntaxique du langage R-EML avec XText

La technologie XText a été utilisée pour concevoir le langage R-EML (sous-section 5.4.1, page 96) et vérifier la grammaire de manière formelle.

Par soucis de temps de concrétisation de l'idée, le fichier ne comporte que des éléments proposés pour écrire le code dévaluation.

Ainsi, il manque le code des conditions, mais surtout une flexibilité dans les types d'éléments et une association aux éléments du méta-modèle R-EM3

Le concept de *Value* (voir figure B.9) permet de caractériser les ensembles, ou *Set*, les éléments primitifs, ou *Primitive*, et les invocations, ou *Invocation*.

B.3 Bilan de la réalisation de R-EMS dans les outils d'eclipse

L'utilisation des outils d'eclipse oblige l'implémentation de R-EMS a être soumise à l'interprétation des développeurs des différents outils concernant la modélisation d'un méta-modèle, et la conception d'un langage dédié. Puis à l'interprétation de l'utilisateur qui doit transformer une architecture dans une technologie avec des forces et des limitations souvent identifiables lors du développement.

Il en résulte que les trente concepts du méta-modèle R-EM3, se sont transformés en trente cinq concepts dans le monde Ecore. Par contre, XText offre beaucoup plus de libertés concernant le développement du langage.

Même s'il y a une différence certaine entre la conception de l'architecture et leur implémentation, la phase de développement permet d'enrichir la phase de conception, surtout si cette dernière est riche en fonctionnalités, et mélange plusieurs domaines d'expertise.

```
grammar org.xtext.example.Tacticos hidden(WS, ML_COMMENT, SL_COMMENT)

generate tacticos "http://www.xtext.org/example/Tacticos"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Model:
    (elements+=Element END_STATEMENT+)+;

Element:
    Imports | Domain
    | Type | Variables | Operation | Enum
    | AnnotatedElement
    | Statement;

Imports:
    'import' imports+=Import (',' imports+=Import)*;

Import:
    name=(STRING | GUARD_ID) ('as' alias=ID)?;

Domain:
    'domain' name=GUARD_ID;

Enum:
    'enum' name=GUARD_ID '{' values+=GUARD_ID (',' values+=GUARD_ID)* '}';

EventType:
    INITIALIZED='*' | INITIALIZED='INITIALIZED'
    | MODIFYING='>~' | MODIFYING='MODIFYING'
    | MODIFIED='~' | MODIFIED='MODIFIED'
    | DELETING='>-' | DELETING='DELETING'
    | DELETED='-' | DELETED='DELETED'
    | MOVING='>>' | MOVING='MOVING'
    | MOVED='>' | MOVED='MOVED'
    | EXECUTING='>%' | EXECUTING='EXECUTING'
    | EXECUTED='%' | EXECUTED='EXECUTED'
    | CUSTOM=GUARD_ID | CUSTOM=STRING;

Variables:
    'var' variables+=Variable (',' variables+=Variable)*
    (':' types+=GUARD_ID (',' types+=GUARD_ID)* ('=' values+=Value (',' values+=Value)*)?)?;

Variable:
    name=ID ('=' value=Value)?;

Type:
    ('let' | 'class') name=GUARD_ID
    ('(' (parameters+=Parameter (',' parameters+=Parameter)*)? ')')?
    value=Set
    => ('=' defaultValue=Element)?;
```

FIGURE B.8 – **Grammaire du langage R-EML sous XText** – Partie 1/4

```
Operation:
    ('op' | 'def') name=GUARD_ID
    ('(' (parameters+=Parameter (',' parameters+=Parameter)*)? ')')?
    (value=Set | ('=' value=Element));

Parameter:
    name=GUARD_ID;

AnnotatedElement:
    annotations+=Annotation+
    => element=Element;

Annotation:
    '[' annotations+=Annotation* value=Invocation ']';

Statement:
    Value
    | For | ForEnum | DoWhile | Switch | Match
    | Return | BackTo | Leave
    | Fire | Listen
    | Throw | Try
;

Value:
    Invocation | Set | Primitive;

Set:
    {Set}
    '{' (content+=Element (END_STATEMENT content+=Element)*)? '}';

Invocation:
    name=GUARD_ID '(' parameterValues+=ParameterValue (',' parameterValues+=ParameterValue)* ')';

ParameterValue:
    name=GUARD_ID '=' value=Element;

Primitive:
    literalValue=STRING | Number | Bool | BaseNumber;

Number:
    value=DOUBLE;

Bool:
    value=BOOLEAN;

StringValue:
    STRING ('.' (STRING | ID))*;

BaseNumber:
    base=('b' | 'O' | '0x') value=INT;
```

FIGURE B.9 – **Grammaire du langage R-EML sous XText** – Partie 2/4

```
For:
    'for' '(' init=Element ';' condition=Element ';' incrementation=Element ')' impl=Element;

ForEnum:
    'for' '(' variable=GUARD_ID ':' values=GUARD_ID ')' impl=Element;

DoWhile:
    'do' statement=Element 'while' condition=Element;

Switch:
    ('switch' | '?=') expression=Element '{' (cases+=Case END_STATEMENT)* (default=Default)? '}';

Case:
    ('case' | '=' | '~') expression=Element ':' (statement=Element)?;

Default:
    {Default}
    ('default' | ':') (statement=Element)?;

Match:
    ('match' | '?~') expression=Element '{' (cases+=Case END_STATEMENT)* (default=Default)? '}';

Return:
    {Return}
    ('return' | '=>') => (value=Element)?;

BackTo:
    {BackTo}
    ('<-' | 'continue') (statement=GUARD_ID)?;

Leave:
    {Leave}
    ('->' | 'break') statement=GUARD_ID? (('with') value=Element);

Fire:
    ('fire' | '!') event=Element;

Listen:
    ('>!' | 'on') notifier=GUARD_ID event=EventType listener=Statement;

Throw:
    ('throw' | '!!') exception=Statement;

Try:
    ('try' | '!!?') expression=Statement (catches+=Catch)+ (finally=Finally)?;

Catch:
    ('catch' | '>!!') condition=Statement expression=Statement;
```

FIGURE B.10 – **Grammaire du langage R-EML sous XText** – Partie 3/4

```
Finally:
    ('finally' | '!!>') expression=Statement;

terminal END_STATEMENT:
    ';' | '\n';

terminal DOUBLE returns ecore::EBigDecimal:
    ('-')? ('0'..'9')+ ('.' ('0'..'9')+)?;

terminal INT returns ecore::EIntegerObject:
    ('0'..'9')+;

terminal BOOLEAN returns ecore::EBooleanObject:
    'true' | 'false';

terminal GUARD_ID:
    ID ('.' ID)*;

terminal ID:
    ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*;

terminal STRING:
    '"' ('\\' ('b' | 't' | 'n' | 'f' | 'r' | 'u' | '"' | "'" | '\\') | !('\\' | '"'))* '"';

terminal WS:
    (' ' | '\t' | '\r' | '\n')+;

terminal ML_COMMENT:
    '/*'->'*/';

terminal SL_COMMENT:
    '//' !('\n' | '\r')* ('\r'? '\n')?;
```

FIGURE B.11 – **Grammaire du langage R-EML sous XText** – Partie 4/4

Index

Bibliographie

[0MQ,] 0MQ web page - http ://www.zeromq.org/.

[Act,] ActiveMQ web page - http ://activemq.apache.org/.

[AOP,] Aop alliance - http ://aopalliance.sourceforge.net.

[afn,] Association française de normalisation (afnor) web page - http ://www.boutique.afnor.org/.

[Fra,] Frascati fscript home page - http ://frascati.ow2.org/doc/1.1.1/ch08.html.

[ite,] Itemis web page - http ://www.itemis-anr.org/.

[JBo,] JBossMQ web page - http ://www.jboss.org/.

[JGr,] JGroups web page - http ://www.jgroups.org/.

[JOR,] JORAM web page - http ://joram.ow2.org/.

[Ope, a] OpenDDS web page – http ://www.opendds.org.

[Ope, b] OpenJMS web page - http ://openjms.sourceforge.net/.

[OSG,] Osgi web page - http ://www.osgi.org/.

[OSO,] Osoa home page - http ://www.osoa.org/display/main/home.

[Rab,] RabbitMQ web page - http ://www.rabbitmq.com/.

[RTI,] RTI web page - http ://www.rti.com.

[SCA,] Sca - oasis - home page - http ://www.oasis-opensca.org/sca.

[SLA,] Service level agreement, ibm, http ://www.service-level-agreement.net/.

[Spr,] Spring web page - http ://www.springframework.org/.

[MAR,] Uml profile for marte, http ://www.omgmarte.org/.

[DNC, 2002] (2002). *Deployment and Configuration of Component-based Distributed Applications Specification*. OMG.

[TAC, 2006] (2006). Tacticos architecture.

[MAR, 2009] (2009). *UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded Systems*. OMG.

[AMQ, 2010] (2010). *AMQP Recommendation*. AMQP Working Group.

[amq, 2010] (2010). Amqp web page - http ://www.amqp.org/.

[Alf, 2010] (2010). *Concrete Syntax for a UML Action Language*. OMG.

[Pro, 2011] (2011). Google Inc. ProtoBuf - Protocol Buffers - Google's data interchange format - http ://code.google.com/p/protobuf/. WebSite.

[Aagedal, 2001] Aagedal, J. (2001). *Quality of Service Support in Development of Distributed Systems*. PhD thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo.

[Abrial et al., 1991] Abrial, J., Lee, M., Neilson, D., Scharbach, P., and Sørensen, I. (1991). The b-method. In Prehn, S. and Toetenel, H., editors, *VDM '91 Formal Software Development Methods*, volume 552 of *Lecture Notes in Computer Science*, pages 398–405. Springer Berlin / Heidelberg. 10.1007/BFb0020001.

[ADAM, 2012] ADAM (2012). Inria/FraSCAti home page - http ://frascati.ow2.org/.

[Alonistioti et al., 2006] Alonistioti, N., Patouni, E., and Gazis, V. (2006). Generic architecture and mechanisms for protocol reconfiguration. *Mobile Networks and Applications*, 11 :18.

[Ameziani, 2009] Ameziani, H. (2009). Modèles de coordination avancés pour intégration des systèmes d'information et embarqués. Master report, INRIA-Lip6.

[Anerousis, 1999] Anerousis, N. (1999). An architecture for building scalable, web-based management services. *Journal of Network and Systems Management*, 7(1) :32.

[anter et al., 2010] anter, E., Moret, P., Binder, W., and Ansaloni, D. (2010). Composition of dynamic analysis aspects. In *GPCE '10 : Proceedings of the ninth international conference on Generative programming and component engineering*, pages 113–122, New York, NY, USA. ACM.

[Apel and Batory, 2006] Apel, S. and Batory, D. (2006). When to use features and aspects ? : a case study. In *GPCE '06 : Proceedings of the 5th international conference on Generative programming and component engineering*, pages 59–68, New York, NY, USA. ACM.

[Avizienis et al., 2004] Avizienis, A., Laprie, J., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1) :11–33.

[Backus et al., 1963] Backus, J., Bauer, F., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A., Rutishauser, H., Samelson, K., Vauquois, B., et al. (1963). Revised report on the algorithmic language algol 60. *The Computer Journal*, 5(4) :349–367.

[Baker, 2001] Baker, S. (2001). A2A, B2B-Now We Need M2M (Middleware to Middleware) Technology. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA'01). IONA Technologies*.

[Baligand and Monfort, 2004] Baligand, F. and Monfort, V. (2004). A concrete solution for web services adaptability using policies and aspects. In *Proceedings of the 2nd international conference on Service oriented computing*, ICSOC '04, pages 134–142, New York, NY, USA. ACM.

[Baligand et al., 2008] Baligand, F., Rivierre, N., and Ledoux, T. (2008). Qos policies for business processes in service oriented architectures.

[Barack et al., 2008] Barack, R., Beisiegel, M., Blohm, H., Booz, D., Edwards, M., Karmarkar, A., Keith, M., Malhotra, A., Patil, S., Peddada, P., Peshev, P., Peters, M., and Rowley, M. (2008). *SCA - Java EE Integration Specification*. OSOA Community.

[Becht et al., 1999] Becht, M., Gurzki, T., Klarmann, J., and Muscholl, M. (1999). Rope : Role oriented programming environment for multi-agent systems. In *Cooperative Information Systems, 1999. CoopIS'99. Proceedings. 1999 IFCIS International Conference on*, pages 325–333. IEEE.

[Beck, 2000] Beck, K. (2000). Extreme programming explained. 2000.

[Becker, 2008] Becker, S. (2008). Quality of Service Modeling Language. In Eusgeld, I., Freiling, F., and Reussner, R., editors, *Dependability Metrics*, volume 4909 of *Lecture Notes in Computer Science*, pages 43–47. Springer Berlin / Heidelberg.

[Beisiegel et al., 2009] Beisiegel, M., Bezrukhov, V., Booz, D., Chapman, M., Edwards, M., Karmarkar, A., Malhotra, A., Niblett, P., Patil, S., and Vorthmann, S. (2009). *SCA - Assembly Model Specification Extensions for Event Processing and Pub/Sub*. OSOA Community.

[Beisiegel et al., 2007a] Beisiegel, M., Blohm, H., Booz, D., Edwards, M., Hurley, O., Ielceanu, S., Miller, A., Karmarkar, A., Malhotra, A., Marino, J., Nally, M., Newcomer, E., Patil, S., Pavlik, G., Raepple, M., Rowley, M., Tam, K., Vorthmann, S., Walker, P., and Waterman, L. (2007a). *SCA - Assembly Model Specification*. OSOA Community.

[Beisiegel et al., 2007b] Beisiegel, M., Booz, D., Chao, C.-Y., Edwards, M., Ielceanu, S., Karmarkar, A., Malhotra, A., Newcomer, E., Patil, S., Rowley, M., Sharp, C., and Yalçinalp, Ü. (2007b). *SCA - Policy Framework*. OSOA Community.

[Benguria et al., 2007] Benguria, G., Larrucea, X., Elvesæter, B., Neple, T., Beardsmore, A., and Friess, M. (2007). A platform independent model for service oriented architectures. In Doumeingts, G.,

Müller, J., Morel, G., and Vallespir, B., editors, *Enterprise Interoperability*, pages 23–32. Springer London. 10.1007/978-1-84628-714-5_3.

[Bertolino et al., 2010] Bertolino, A., Chiaradonna, S., Costa, G., Di Giandomenico, F., Di Marco, A., Grace, P., Issarny, V., Kwiatkowska, M., Martinelli, F., Masci, P., Matteucci, I., Qu, H., Rouncefield, M., Saadi, R., Sabetta, A., Spalazzese, R., and Taiani, F. (2010). Conceptual models for assessment & assurance of dependability, security and privacy in the eternal connected world. Rapport technique.

[Beugnard et al., 1999] Beugnard, A., Jézéquel, J., Plouzeau, N., and Watkins, D. (1999). Making components contract aware. *Computer*, 32(7) :38–45.

[Bézivin et al., 2005] Bézivin, J., Hillairet, G., Jouault, F., Kurtev, I., and Piers, W. (2005). Bridging the ms/dsl tools and the eclipse modeling framework. page 7.

[Bicer et al., 2008] Bicer, M., Kulp, J., and Pilhofer, F. (2008). Increasing the cots content of the sca.

[Billings et al., 2006] Billings, J., Sewell, P., Shinwell, M., and Strniša, R. (2006). Type-safe distributed programming for OCaml. In *Proceedings of the 2006 workshop on ML*, ML '06, pages 20–31, New York, NY, USA. ACM.

[Booch, 1986] Booch, G. (1986). Object-oriented development. In *IEEE Trans. Software Eng.*, volume 12, pages 211–221.

[Booz et al., 2007] Booz, D., Edwards, M., Kanaley, M., Little, M., Malhotra, A., Newcomer, E., Patil, S., Robinson, I., and Rowley, M. (2007). *ACID Transaction Policy in SCA*. OSOA Community.

[Bosch, 2000] Bosch, J. (2000). *Design and use of software architectures : adopting and evolving a product-line approach*. Addison-Wesley Professional.

[Bracha and Cook, 1990] Bracha, G. and Cook, W. (1990). Mixin-based inheritance. *SIGPLAN Not.*, 25(10) :303–311.

[Braga et al., 2009] Braga, C., Chalub, F., and Sztajnberg, A. (2009). A Formal Semantics for a Quality of Service Contract Language. *Electronic Notes in Theoretical Computer Science*, 203(7) :103–120.

[Brown et al., 1998] Brown, W. H., Malveau, R. C., and Mowbray, T. J. (1998). *AntiPatterns : Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1 edition.

[Bruneton et al., 2004] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2004). An open component model and its support in java. In [Crnkovic et al., 2004], pages 7–22.

[Bruneton et al., 2002] Bruneton, E., Coupaye, T., and Stefani, J.-B. (2002). Recursive and dynamic software composition with sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOPi02)*. Citeseer.

[Cansado et al., 2010] Cansado, A., Canal, C., Salaün, G., and Cubo, J. (2010). A Formal Framework for Structural Reconfiguration of Components under Behavioural Adaptation. *Electronic Notes in Theoretical Computer Science*, 263(0) :95–110.

[Cassou, 2011] Cassou, D. (2011). *Développement logiciel orienté paradigme de conception : la programmation dirigée par la spécification*. PhD thesis, Université de Bordeaux 1.

[CENELEC, 2012] CENELEC (2012). European committee for electrotechnical standardization (cenelec) web page - http ://www.cenelec.eu/cenelec/.

[Cerqueira et al., 2001] Cerqueira, R., Hess, C. K., Romàn, M., and Campbell, R. H. (2001). Gaia : A development infrastructure for active spaces. In *Workshop on Application Models and Programming Tools for Ubiquitous Computing (held in conjunction with the UBICOMP 2001)*, volume 2, page 11.

[Chang and Collet, 2007] Chang, H. and Collet, P. (2007). Patterns for integrating and exploiting some non-functional properties in hierarchical software components. In *Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops.*, pages 83–92. IEEE.

[CHAPPELL, 2007] CHAPPELL, D. (2007). Introducing sca.

[Charfi and Mezini, 2007] Charfi, A. and Mezini, M. (2007). AO4BPEL : An Aspect-oriented Extension to BPEL. *World Wide Web*, 10 :309–344.

[Cheng et al., 2009] Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., et al. (2009). Software engineering for self-adaptive systems : A research roadmap. *Software Engineering for Self-Adaptive Systems*, pages 1–26.

[Chiba and Ishikawa, 2005] Chiba, S. and Ishikawa, R. (2005). Aspect-Oriented Programming Beyond Dependency Injection. In Black, A., editor, *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 121–143. Springer Berlin / Heidelberg.

[Cleland-Huang et al., 2010] Cleland-Huang, J., Czauderna, A., Gibiec, M., and Emenecker, J. (2010). A machine learning approach for tracing regulatory codes to product specific requirements. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 155–164. ACM.

[Clements and Northrop, 2001] Clements, P. and Northrop, L. (2001). *Software Product Lines : Practices and Patterns*, volume 67. Addison-Wesley Longman Publishing Co., Boston, MA, USA.

[Collaboration, 2007] Collaboration, O. (2007). Power combination : Sca, osgi and spring. page 11.

[Collet et al., 2005] Collet, P., Rousseau, R., Coupaye, T., and Rivierre, N. (2005). A Contracting System for Hierarchical Components. In Heineman, G., Crnkovic, I., Schmidt, H., Stafford, J., Szyperski, C., and Wallnau, K., editors, *Component-Based Software Engineering*, volume 3489 of *Lecture Notes in Computer Science*, pages 75–90. Springer Berlin / Heidelberg.

[Computing et al., 2006] Computing, A. et al. (2006). An architectural blueprint for autonomic computing. *IBM White Paper*.

[Comuzzi and Pernici, 2009] Comuzzi, M. and Pernici, B. (2009). A framework for QoS-based Web service contracting. *ACM Trans. Web*, 3(3) :10 :1–10 :52.

[Cook et al., 2007] Cook, S., Jones, G., Kent, S., and Wills, A. (2007). *Domain-specific development with visual studio dsl tools*. Addison-Wesley Professional, first edition.

[Costa et al., 2007] Costa, P., Coulson, G., Mascolo, C., Mottola, L., Picco, G. P., and Zachariadis, S. (2007). Reconfigurable component-based middleware for networked embedded systems. *International Journal of Wireless Information Networks*, 14(2).

[Coulson et al., 2008] Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., and Sivaharan, T. (2008). A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1) :1 :1–1 :42.

[Crnkovic et al., 2004] Crnkovic, I., Schmidt, H. W., and Wallnau, K. C., editors (2004). *Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings*, volume 3054 of *Lecture Notes in Computer Science*. Springer.

[Crnkovic et al., 2009] Crnkovic, I., Sentilles, S., Vulgarakis, A., and Chaudron, M. R. (2009). A classification framework for software component models. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, pages 1–25.

[Şavga and Rudolf, 2007] Şavga, I. and Rudolf, M. (2007). Refactoring-based support for binary compatibility in evolving frameworks. In *PCE '07 : Proceedings of the 6th international conference on Generative programming and component engineering*, pages 175–184, New York, NY, USA. ACM.

[Dar et al., 2011] Dar, K., Taherkordi, A., Rouvoy, R., and Eliassen, F. (2011). Adaptable service composition for very-large-scale internet of things systems. In *Proceedings of the 8th Middleware Doctoral Symposium*, MDS '11, pages 2 :1–2 :6, New York, NY, USA. ACM, ACM.

[David et al., 2006] David, P., Ledoux, T., et al. (2006). Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceeding of Fractal CBSE Workshop, ECOOP*, volume 6.

[David et al., 2008] David, P.-C., Ledoux, T., Léger, M., and Coupaye, T. (2008). Fpath and fscript : Language support for navigation and reliable reconfiguration of fractal architectures.

[Dearle, Alan, 2007] Dearle, Alan (2007). Software Deployment, Past, Present and Future. In *2007 Future of Software Engineering*, FOSE '07, pages 269–284, Washington, DC, USA. IEEE Computer Society.

[Deif and ElMaraghy, 2006] Deif, A. M. and ElMaraghy, W. (2006). Effect of reconfiguration costs on planning for capacity scalability in reconfigurable manufacturing systems. *Int J Flex Manuf Syst*, 18 :14.

[Dijkstra, 1974] Dijkstra, E. W. (1974). Dijkstra, e. w. (1974). on the role of scientific thought. selected writings on computing : A personal perspective, pages 60–66.

[Dimitrakopoulos et al., 2006] Dimitrakopoulos, G., Moessner, K., Kloeck, C., Grandblaise, D., Gault, S., Sallent, O., Tsagkaris, K., and Demestichas, P. (2006). Adaptive resource management platform for reconfigurable networks. *Mobile Networks and Applications*, 11 :13.

[Dobson et al., 2006] Dobson, S., Denazis, S., Fernández, A., Gaiti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., and Zambolli, F. (2006). A survey of autonomic communications. In *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, volume 1, pages 223–259.

[Douence et al., 2006] Douence, R., Le Botlan, D., Noyé, J., and Südholt, M. (2006). Concurrent aspects. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, pages 79–88, New York, NY, USA. ACM.

[Ducasse et al., 2006] Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P. (2006). Traits : A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2) :331–388.

[Duke et al., 2005] Duke, A., Davies, J., and Richardson, M. (2005). Enabling a scalable service-oriented architecture with semantic web services. *BT Technology Journal*, 23(3) :11.

[Duran-Limon et al., 2003] Duran-Limon, H., Blair, G., Friday, A., Sivaharan, T., and Samartzidis, G. (2003). A resource and QoS management framework for a real-time event system in mobile ad hoc environments. In *Object-Oriented Real-Time Dependable Systems, 2003. WORDS 2003 Fall. Proceedings. Ninth IEEE International Workshop on*, pages 217–224, Dept. of Comput., Lancaster Univ., UK. IEEE.

[En-Nouaary, 2007] En-Nouaary, A. (2007). A scalable method for testing real-time systems. *Software Qual J*, 16(3) :20.

[Espert et al., 2005] Espert, I. B., García, V. H., and Quilis, J. D. S. (2005). An ogsa middleware for managing medical images using ontologies. *Journal of Clinical Monitoring and Computing*, 19.

[Eysholdt and Behrens, 2010] Eysholdt, M. and Behrens, H. (2010). Xtext : implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 307–309, New York, NY, USA. ACM, ACM.

[Fabro and Valduriez, 2007] Fabro, M. D. D. and Valduriez, P. (2007). Towards the efficient development of model transformations using model weaving and matching transformations. *Softw Syst Model*, 8 :20.

[Fares et al., 2011] Fares, E., Bodeveix, J., and Filali, M. (2011). Verification of Timed BPEL 2.0 Models. *Enterprise, Business-Process and Information Systems Modeling*, pages 261–275.

[Fiadeiro et al., 2005] Fiadeiro, J. L., Lopes, A., and Bocchi, L. (2005). A formal approach to service component architecture. page 21.

[Fowler, 2004] Fowler, M. (2004). Inversion of Control Containers and the Dependency Injection pattern.

[Fowler and Beck, 1999] Fowler, M. and Beck, K. (1999). *Refactoring : improving the design of existing code*. Addison-Wesley Professional.

[Fraenkel et al., 1973] Fraenkel, A., Bar-Hillel, Y., and Levy, A. (1973). *Foundations of set theory*, volume 67. North Holland.

[Friedenthal et al., 2011] Friedenthal, S., Moore, A., and Steiner, R. (2011). *A practical guide to SysML : the systems modeling language*. Morgan Kaufmann.

[Frølund and Koistinen, 1998a] Frølund, S. and Koistinen, J. (1998a). Qml : A language for quality of service specification. *HP LABORATORIES TECHNICAL REPORT HPL*.

[Frølund and Koistinen, 1998b] Frølund, S. and Koistinen, J. (1998b). Quality-of-service specification in distributed object systems. *Distributed Systems Engineering*, 5(4) :179.

[G. Outhred, 1998] G. Outhred, J. P. (1998). A model for component composition with sharing. In *Component Oriented Programming (WCOP)*, page 9.

[Gamma, 1995] Gamma, E. (1995). *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Professional.

[Gibiec et al., 2010] Gibiec, M., Czauderna, A., and Cleland-Huang, J. (2010). Towards mining replacement queries for hard-to-retrieve traces. pages 245–254.

[Gogolla et al., 2008] Gogolla, M., Kuhmann, M., and Büttner, F. (2008). A Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency. In Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., and Völter, M., editors, *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 446–459. Springler Berlin / Heidelberg.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80 : the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Gray and Reuter, 1993] Gray, J. and Reuter, A. (1993). *Transaction processing : concepts and techniques*. Morgan Kaufmann.

[Grossman, 2009] Grossman, R. L. (2009). The case for cloud computing. *IT Professional*, 11(2) :23–27.

[Group, 2007] Group, O. M. (2007). *Unified Modeling Language (UML) : Superstructure, v2.1.1*.

[Hapner et al., 2002] Hapner, M., Burridge, R., Sharma, R., Fialli, J., and Stout, K. (2002). *Java Message Service Specification*. Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303 U.S.A.

[Harrison and Ossher, 1993] Harrison, W. and Ossher, H. (1993). Subject-oriented programming : a critique of pure objects. *ACM Sigplan Notices*, 28(10) :411–428.

[Hassan et al., 2010] Hassan, M., Song, B., and Huh, E.-N. (2010). A dynamic and fast event matching algorithm for a content-based publish/subscribe information dissemination system in Sensor-Grid. *The Journal of Supercomputing*, 54 :330–365. 10.1007/s11227-009-0327-0.

[Haugen et al., 2010] Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G. K., and Svendsen, A. (2010). Adding standardized variability to domain specific languages. *12th International Software Product Line Conference*, page 10.

[Hoffert et al., 2010] Hoffert, J., Schmidt, D. C., and Gokhale, A. (2010). Adapting distributed real-time and embedded pub/sub middleware for cloud computing environments. In *ACM*.

[Hoffert et al., 2009] Hoffert, J., Schmidt, D. C., and Gokhale, A. S. (2009). Evaluating transport protocols for real-time event stream processing middleware and applications. In [Meersman et al., 2009], pages 614–633.

[Huang et al., 2006] Huang, G., Mei, H., and Yang, F.-Q. (2006). Runtime recovery and manipulation of software architecture of component-based systems. *Autom Software Engineering*, 13(2) :257–281. 10.1007/s10515-006-7738-4.

[Hudak, 1989] Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3) :359–411.

[Huebscher and McCann, 2008] Huebscher, M. and McCann, J. (2008). A survey of autonomic computing-degrees, models, and applications. *ACM Comput. Surv.*, 40(3) :1–28.

[Hursch and Lopes, 1995] Hursch, W. L. and Lopes, C. V. (1995). Separation of concerns.

[Inc., 2006] Inc., E. (2006). Esper web page - http ://esper.codehaus.org/.

[Inc, 2006] Inc, G. (2006). Google guice - http ://code.google.com/p/google-guice/.

[Inc., 2009] Inc., G. (2009). Kryo web page - http ://code.google.com/p/kryo/.

[Inc, 2009] Inc, G. (2009). KryoNet web page - http ://code.google.com/p/kryonet/.

[Inc., 2010] Inc., G. (2010). Protostuff web page - http ://code.google.com/p/protostuff/.

[Inc, 2011] Inc, G. (2011). Cicles in gmail, https ://plus.google.com/101560853443212199687/posts/7ifktty4uap.

[Jamshidi, 2005] Jamshidi, M. (2005). System-of-Systems Engineering-a Definition. *IEEE SMC (October 2005)*.

[Jie et al., 2006] Jie, L., Ru-chuan, W., and Zheng-ai, B. (2006). Mobile intelligent agent entity model towards qos guarantee. *Front. Electr. Electron. Eng. China*, 4 :7.

[Jim Marino, 2009] Jim Marino, M. R. (2009). *Understanding SCA*. Addison-Wesley Professional.

[J.L.J.Laredo et al., 2008] J.L.J.Laredo, P.A.Castillo, A.M.Mora, and Merelo, J. J. (2008). Evolvable agents, a fine grained approach for distributed evolutionary computing : walking towards the peer-to-peer computing frontiers. *Soft Comput*, 12 :12.

[Jouault, 2008] Jouault, F. (2008). Fscript home page - http ://fractal.ow2.org/fscript/.

[Jung and Hatcliff, 2007] Jung, G. and Hatcliff, J. (2007). A correlation framework for the corba component model. *Int J Softw Tools Technol Transfer*, 9(417) :11.

[Jurack and Taentzer, 2009] Jurack, S. and Taentzer, G. (2009). Towards Composite Model Transformations Using Distributed Graph Transformation Concepts. In Schürr, A. and Selic, B., editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 226–240. Springer Berlin / Heidelberg.

[Kavimandan and Gokhale, 2007] Kavimandan, A. and Gokhale, A. (2007). Automated Middleware QoS Configuration Techniques using Model Transformations. In *EDOC Conference Workshop, 2007. EDOC'07. Eleventh International IEEE*, pages 20–27. IEEE.

[Keller and Ludwig, 2003a] Keller, A. and Ludwig, H. (2003a). The WSLA Framework : Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11 :57–81.

[Keller and Ludwig, 2003b] Keller, A. and Ludwig, H. (2003b). The wsla framework : Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11 :57–81.

[Kephart and Chess, 2003] Kephart, J. and Chess, D. (2003). The vision of autonomic computing. *Computer*, 36(1) :41–50.

[Keskes et al., 2011] Keskes, N., Lehireche, A., and Rahmoun, A. (2011). Web services selection based on mixed context and quality of service ontology. *Computer and Information Science*, 4(3) :p138.

[Kiczales et al., 1991] Kiczales, G., Des Rivieres, J., and Bobrow, D. (1991). *The art of the metaobject protocol*. The MIT press.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). An Overview of AspectJ. In *ECOOP 2001 – Object-Oriented Programming*, number 2072, pages 327–354. Springer Berlin / Heidelberg.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Aksit, M. and Matsuoka, S., editors, *ECOOP'97 – Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg.

[Klas et al., 2009] Klas, M., Heidrich, J., Munch, J., and Trendowicz, A. (2009). CQML Scheme : A Classification Scheme for Comprehensive Quality Model Landscapes. In *Software Engineering and Advanced Applications, 2009. SEAA'09. 35th Euromicro Conference on*, pages 243–250. IEEE.

[Kleppe et al., 2003] Kleppe, A. G., Warmer, J., and Bast, W. (2003). *MDA Explained : The Model Driven Architecture : Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Kolsi et al., 2009] Kolsi, N., Abdellatif, A., and Ghedira, K. (2009). Data warehouse access using multi-agent system. *Distrib Parallel Databases*, 25 :17.

[Krämer, 2007] Krämer, B. J. (2007). Component meets service : what does the mongrel look like ? *Innovations Syst Softw Eng*, 4 :10.

[Kritikos and Plexousakis, 2006] Kritikos, K. and Plexousakis, D. (2006). Semantic qos metric matching. In *Web Services, 2006. ECOWS'06. 4th European Conference on*, pages 265–274. IEEE.

[Kritikos and Plexousakis, 2009] Kritikos, K. and Plexousakis, D. (2009). Requirements for qos-based web service description and discovery. *Services Computing, IEEE Transactions on*, 2(4) :320–337.

[Labéjof, 2010] Labéjof, J. (2010). *WS4LwCCM*. THALES/INRIA.

[Labéjof et al., 2012a] Labéjof, J., Léger, A., Merle, P., Seinturier, L., and Vincent, H. (2012a). A reflective component model for OMG DDS.

[Labéjof et al., 2012b] Labéjof, J., Léger, A., Merle, P., Seinturier, L., and Vincent, H. (2012b). R-MOM : A Component-Based Framework for Interoperable and Adaptive Asynchronous Middleware Systems. In *SCDI - The First International Workshop on Service and Cloud Based Data Integration - 2012*, Beijing, Chine. Sixteenth IEEE International EDOC Conference, Springer.

[Labreuche and Le Huédé, 2005] Labreuche, C. and Le Huédé, F. (2005). MYRIAD : a tool suite for MCDA. *EUSFLAT'05*, pages 204–209.

[Lau and Wang, 2007] Lau, K.-K. and Wang, Z. (2007). Software component models. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, pages 709–724.

[Lavinal et al., 2008] Lavinal, E., Simoni, N., Song, M., and Mathieu, B. (2008). A next-generation service overlay architecture. *Ann. Telecommun*, 64 :11.

[Leclercq et al., 2004] Leclercq, M., Quéma, V., and Stefani, J.-B. (2004). DREAM : a component framework for the construction of resource-aware, reconfigurable MOMs. In *Proceedings of the 3rd workshop on Adaptive and reflective middleware*, ARM'04, pages 250–255, New York, NY, USA. ACM.

[Lee et al., 2006] Lee, E. K., Easton, T., and Gupta, K. (2006). Novel evolutionary models and applications to sequence alignment problems. *Ann Oper Res*, 148 :21.

[Lee et al., 2009] Lee, J., Lee, J., Kim, S., et al. (2009). A quality model for evaluating software-as-a-service in cloud computing. In *Software Engineering Research, Management and Applications, 2009. SERA'09. 7th ACIS International Conference on Information Systems*, pages 261–266. IEEE.

[Léger et al., 2010] Léger, M., Ledoux, T., and Coupaye, T. (2010). Reliable dynamic reconfigurations in a reflective component model. In Grunske, L., Reussner, R., and Plasil, F., editors, *Component-Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 74–92. Springer Berlin / Heidelberg. 10.1007/978-3-642-13238-4_5.

[Lehman, 1980] Lehman, M. M. (1980). Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9) :1060–1076.

[Li, 2009] Li, K. (2009). Fast and highly scalable parallel computations for fundamental matrix problems on distributed memory systems. *J Supercomput*, page 27.

[Li and Gui, 2009] Li, X.-Y. and Gui, X.-L. (2009). A comprehensive and adaptive trust model for large-scale p2p networks. *JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY*, 24(5) :15.

[Lim et al., 2007] Lim, S. B., Lee, H., Carpenter, B., and Fox, G. (2007). Runtime support for scalable programming in java. *J Supercomput*, page 18.

[Lin et al., 2011] Lin, S., Taïani, F., Bertier, M., Blair, G., and Kermarrec, A.-M. (2011). Transparent Componentisation : High-Level (Re)configurable Programming for Evolving Distributed Systems. In *SAC '11 Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 203–208. ACM New York, NY, USA ©2011.

[Lin et al., 2007] Lin, S., Taïani, F., and Blair, G. (2007). Gossipkit : A framework of gossip protocol family. In *5th MiNEMA Workshop (Middleware for Network Eccentric and Mobile Applications)*, pages 26–30.

[Lin et al., 2008] Lin, S., Taïani, F., and Blair, G. S. (2008). Facilitating Gossip Programming with the Gossipkit Framework. In Meier, R. and Terzis, S., editors, *Distributed Applications and Interoperable Systems 8th IFIP WG 6.1 International Conference, DAIS 2008, Oslo, Norway, June 4-6, 2008.*, volume 5053/2008. Springer Berlin / Heidelberg.

[Linthicum, 2000] Linthicum, D. S. (2000). *Enterprise application integration*. Addison-Wesley Longman Ltd., Essex, UK, UK.

[Liu, 2000] Liu, J. W. S. W. (2000). *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition.

[Liua et al., 2006] Liua, Q., Zhou, S., and Giannakis, G. B. (2006). Cross-layer modeling of adaptive wireless links for qos support in heterogeneous wired-wireless networks. *Wireless Netw*, 12 :11.

[Loring et al., 2008] Loring, P. A., III, F. S. C., and Gerlach, S. C. (2008). The services-oriented architecture : Ecosystem services as a framework for diagnosing change in social ecological systems. *Ecosystems*, 11 :12.

[Louis M.Rose, 2009] Louis M.Rose, Richard F.Paige, D. S. F. A. (2009). An analysis of approaches to model migration. page 10.

[Ludwig et al., 2003] Ludwig, H., Keller, A., Dan, A., P.King, R., and Franck, R. (2003). *Web Service Level Agreement (WSLA) Language Specification*. IBM.

[Lynch, 2008] Lynch, C. (2008). Big data : How do your data grow ? *Nature*, 455(7209) :28–29.

[Maier, 1998] Maier, M. W. (1998). Architecting Principles for Systems-of-Systems, http ://www.infoed.com/open/papers/systems.htm.

[McGuiness et al., 2004] McGuiness, D., Van Harmelen, F., et al. (2004). OWL web ontology language overview. *W3C recommendation*, 10.

[Medvidovic and Taylor, 2000] Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 26 :24.

[Meersman et al., 2009] Meersman, R., Dillon, T. S., and Herrero, P., editors (2009). *On the Move to Meaningful Internet Systems : OTM 2009, Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009, Vilamoura, Portugal, November 1-6, 2009, Proceedings, Part I*, volume 5870 of *Lecture Notes in Computer Science*. Springer.

[Mehta et al., 2000] Mehta, N. R., Medvidovic, N., and Phadke, S. (2000). Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 178–187, New York, NY, USA. ACM.

[Mélisson et al., 2010] Mélisson, R., Romero, D., Rouvoy, R., , and Seinturier, L. (2010). Supporting Pervasive and Social Communications with FRASCATI. In of the EASST, E. C., editor, *Third International DisCoTec Workshop on Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services*, volume 28, Amsterdam, Netherlands. June.

[Mercadal, 2011] Mercadal, J. (2011). *Approche langage au développement logiciel : application au domaine des systèmes d'informatique ubiquitaire*. PhD thesis, Université de Bordeaux.

[Microsoft©, 2008] Microsoft© (2008). Windows Communication Fundation© (WCF©) Web page - http ://msdn.microsoft.com/en-us/netframework/wcf.

[Mietzner et al., 2008] Mietzner, R., Leymann, F., and Papazoglou, M. (2008). Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-tenancy Patterns. In *Internet and Web Applications and Services, 2008. ICIW '08. Third International Conference on*, pages 156–161.

[Moore et al., 2004] Moore, B., Dean, D., Gerber, A., Wagenknecht, G., and Vanderheyden, P. (2004). *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM.

[Mosser, 2010] Mosser, S. (2010). *Behavorial composition in service-oriented architecture*. Computer science, École Doctorale de Sciences et Technologies de l'Information et de la Communication (STIC).

[Na and Lee, 2002] Na, G. S. and Lee, S. H. (2002). Interoperability of Event Service in Java ORB Environment. In *Proceedings of Tenth International Workshop on Database and Expert Systems Applications*, pages 29–33. IEEE.

[Navas et al., 2010] Navas, J. F., Babau, J.-P., and Pulou, J. (2010). Applications of dynamic code evolution for java in gui development and dynamic aspect-oriented programming. In *GPCE '10 : Proceedings of the ninth international conference on Generative programming and component engineering*, pages 73–82, New York, NY, USA. ACM.

[Nikolov et al., 2008] Nikolov, V., Kapitza, R., and Hauck, F. J. (2008). Recoverable class loaders for a fast restart of java applications. *Mobile Netw Appl*, 14 :12.

[Nzekwa, 2010] Nzekwa, R. (2010). Modeling Feedback Control Loops for Self-Adaptive Systems.

[OASIS, 2007] OASIS (2007). Simple object access protocol (soap) web page - http ://www.w3.org/tr/soap/.

[OASIS, 2010a] OASIS (2010a). *SCA Policy Framework Version 1.1*. OASIS.

[OASIS, 2010b] OASIS (2010b). *Service Component Architecture Assembly Model Specification*. OASIS.

[O'Brien and Marakas, 2007] O'Brien, J. A. and Marakas, G. (2007). Introduction to Information Systems. New York, NY, USA. McGraw-Hill, Inc.

[Odersky et al., 2004] Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. (2004). An overview of the Scala programming language. Technical report, Technical Report IC/2004/64, EPFL Lausanne, Switzerland.

[OMG, 1991] OMG (1991). OMG's IDL – interface description language – http ://www.omg.org/gettingstarted/omg_idl.htm.

[OMG, 2001] OMG (2001). *CCM – Corba Component Model Specification*. OMG, 4 edition.

[OMG, 2004] OMG (2004). *Event Service Specification*. OMG.

[OMG, 2007] OMG (2007). *Data Distribution Service for Real-time Systems*. OMG.

[OMG, 2009a] OMG (2009a). Application management and system monitoring for cms systems.

[OMG, 2009b] OMG (2009b). *DDS for Lightweight CCM*. OMG.

[OMG, 2009c] OMG (2009c). *The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification*. OMG.

[OMG, 2010] OMG (2010). *OMG Systems Modeling Language (OMG SysML)*. OMG.

[Oram, 2001] Oram, A., editor (2001). *Peer-to-Peer : Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[Ortiz and Bordbar, 2008] Ortiz, G. and Bordbar, B. (2008). Model-driven quality of service for web services : an aspect-oriented approach. In *Web Services, 2008. ICWS'08. IEEE International Conference on Web Services*, pages 748–751. IEEE.

[OSGi, 2009] OSGi (2009). The OSGi Alliance. OSGi service platform core specification, release 4.2.

[Pan and Thomas, 2007] Pan, J. and Thomas, E. (2007). Approximating owl-dl ontologies. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 1434. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999.

[Papazoglou and van den Heuvel, 2007] Papazoglou, M. and van den Heuvel, W.-J. (2007). Service oriented architectures : approaches, technologies and research issues. *The VLDB Journal*, 16 :389–415. 10.1007/s00778-007-0044-3.

[Pautet, 2001] Pautet, L. (2001). *Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition*. PhD thesis, Université Pierre et Marie-Curie Paris VI.

[Pawlak et al., 2004] Pawlak, R., Retaillé, J.-P., and Seinturier, L. (2004). *La programmation orientée aspect pour Java/J2EE*. Eyrolles.

[Peltz, 2003] Peltz, C. (2003). Web services orchestration and choreography. *Computer*, 36(10) :46–52.

[Pessemier, 2007] Pessemier, N. (2007). *Unification des approches par aspects et 'a composants*. Computer science, Université Lille 1.

[Pfenning and Elliot, 1988] Pfenning, F. and Elliot, C. (1988). Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA. ACM.

[PLSEK, 2009] PLSEK, A. (2009). *SOLEIL : An Integrated Approach for Designing and Developing Component-based Real-time Java Systems*. Computer science, Université Lille 1.

[Popovici et al., 2002] Popovici, A., Gross, T., and Alonso, G. (2002). Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, AOSD'02, pages 141–147, New York, NY, USA. ACM.

[Prehofer, 1997] Prehofer, C. (1997). Feature-oriented programming : A fresh look at objects. In Aksit, M. and Matsuoka, S., editors, *ECOOP'97 – Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer Berlin / Heidelberg.

[PrismTech, 2012] PrismTech (2012). Opensplice home page - http ://www.prismtech.com/opensplice.

[Project, 2007] Project, S. (2007). SCA Platform Specifications - Version 1.0. http : //www.scorware.org/projects/en/deliverables.

[Quinot, 2003] Quinot, T. (2003). *Conception et réalisation d'un intergiciel schizophrène pour la mise en œuvre de systèmes répartis interopérables*. PhD thesis, Docteur de l'Université Paris VI — Pierre-et-Marie-Curie.

[Reed, 2006] Reed, W. (2006). Information, power, and war. In Trappl, R., Shakun, M. F., Bui, T., Faure, G. O., Kersten, G., Kilgour, D. M., and Faratin, P., editors, *Programming for Peace*, volume 2 of *Advances in Group Decision and Negotiation*, pages 335–354. Springer Netherlands. 10.1007/1-4020-4390-2_13.

[Rémy, 2002] Rémy, D. (2002). Using, Understanding, and Unraveling the OCaml Language From Practice to Theory and Vice Versa. *Applied Semantics*, pages 115–137.

[Riederer et al., 2011] Riederer, C., Erramilli, V., Chaintreau, A., Krishnamurthy, B., and Rodriguez, P. (2011). For sale : your data : by : you. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 13 :1–13 :6, New York, NY, USA. ACM.

[Romero, 2011] Romero, D. (2011). *Context as a Resource : A Service-Oriented Approach for Context-Awareness*. Computer science, Université Lille 1.

[Romero et al., 2010] Romero, D., Rouvoy, R., Seinturier, L., and Loiret, F. (2010). Integration of heterogeneous context resources in ubiquitous environments. In *EUROMICRO International Conference on Software Engineering and Advanced Applications (SEAA'10)*, volume 36, Lille : France.

[Röttger and Zschaler, 2003] Röttger, S. and Zschaler, S. (2003). CQML+ : Enhancements to CQML. In *In Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering*, pages 43–56. Cépaduès-Éditions.

[Rouvoy and Merle, 2012] Rouvoy, R. and Merle, P. (2012). Rapid Prototyping of Domain-Specific Architecture Languages. In Larsson, M. and Medvidovic, N., editors, *International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE'12)*, Bertinoro, Italie. ACM.

[Rushkoff, 2010] Rushkoff, D. (2010). *Program or be programmed : Ten commands for a digital age*. Or Books.

[Sacha et al., 2009] Sacha, J., Biskupski, B., Dahlem, D., Cunningham, R., Meier, R., Dowling, J., and Haahr, M. (2009). Decentralising a service-oriented architecture. *Peer-to-Peer Netw Appl*, page 28.

[Salus and Vinton, 1995] Salus, P. and Vinton, G. (1995). *Casting the Net : From ARPANET to Internet and Beyond...* Addison-Wesley Longman Publishing Co., Inc.

[Schärli et al., 2003] Schärli, N., Ducasse, S., Nierstracsz, O., and Black, A. (2003). Traits : Composable Units of Behaviour. In Cardelli, L., editor, *ECOOP 2003 - Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 327–339. Springer Berlin / Heidelberg. 10.1007/978-3-540-45070-2_12.

[Schmidt, 2006] Schmidt, D. C. (2006). Guest editor's introduction : Model-driven engineering. *Computer*, 39(2) :25–31.

[Scowen, 1998] Scowen, R. (1998). Extended bnf-a generic base standard. Technical report, Technical report, ISO/IEC 14977. http ://www. cl. cam. ac. uk/mgk25/iso-14977. pdf.

[Seborg et al., 1989] Seborg, S., Edgard, T., Mellichamp, D., and III, F. D. (1989). Process Dynamics and Control. *John Wiley & sons*, 3 edition.

[Seinturier et al., 2009] Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., and Stefani, J. B. (2009). Reconfigurable sca applications with the frascati platform. HAL - CCSD.

[Seinturier et al., 2012] Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., and Stefani, J. B. (2012). A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software : Practice and Experience*, 42.

[Seinturier et al., 2007] Seinturier, L., Pessemier, N., Duchien, L., and Coupaye, T. (2007). A component model engineered with components and aspects. page 15.

[Shannon, 2001] Shannon, C. (2001). A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1) :3–55.

[Shaw., 1995] Shaw., M. (1995). Beyond objects. In *ACM SIGSOFT Software Engineering Notes (SEN)*, volume 20, pages 27–38.

[Shih et al., 2002] Shih, E., Bahl, P., and Sinclair, M. J. (2002). Wake on wireless : an event driven energy saving strategy for battery operated devices. In *Proceedings of the 8th annual international conference on Mobile computing and networking*, MobiCom '02, pages 160–171, New York, NY, USA. ACM.

[Smaragdakis et al., 2008] Smaragdakis, Y., Csallner, C., and Subramanian, R. (2008). Scalable satisfiability checking and test data generation from modeling diagrams. *Autom Softw Eng*, 16 :27.

[Smith, 1982] Smith, B. C. (1982). *Procedural reflection in programming languages*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science.

[Snyder, 1986] Snyder, A. (1986). Encapsulation and inheritance in object-oriented programming languages. *SIGPLAN Not.*, 21(11) :38–45.

[Sommerville et al., 2012] Sommerville, I., Cliff, D., Calinescu, R., Keen, J., Kelly, T., Kwiatkowska, M., Mcdermid, J., and Paige, R. (2012). Large-scale complex IT systems. *Commun. ACM*, 55(7) :71–77.

[Stankovic and Ramamritham, 1998] Stankovic, J. A. and Ramamritham, K. (1998). *Tutorial on Hard Real-Time Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA.

[Steinberg et al., 2008] Steinberg, D., Bidinsky, F., Merks, E., and Paternostro, M. (2008). *EMF : Eclipse Modeling Framework*. Addison-Wesley Professional.

[Steve, 1997] Steve, V. (1997). CORBA : integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2) :46–55.

[Storey, 1996] Storey, N. R. (1996). *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Suleman et al., 2008] Suleman, H., Parker, C., and Omar, M. (2008). Lightweight component-based scalability. *Int J Digit Libr*, 9 :10.

[Szyperski et al., 2002] Szyperski, C., Gruntz, D., and Murer, S. (2002). *Component software : beyond object-oriented programming*. Addison-Wesley Professional.

[Taherkordi et al., 2010] Taherkordi, A., Loiret, F., Abdolrazaghi, A., Rouvoy, R., Le-Trung, Q., and Eliassen, F. (2010). Programming Sensor Networks Using REMORA Component Model. In (Springer), L. ., editor, *6th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS'10)*, volume 6, pages 45–62, Santa Barbara, California, USA.

[Taïani, 2002] Taïani, F. (2002). Adaptabilité et tolérance aux fautes : intérêt des intergiciels réflexifs face à l'évolutivité des systèmes informatiques. 3ème Congrès des Doctorants de l'Ecole Doctorale Systèmes 02192, LAAS-CNRS, Toulouse (France).

[Talpin et al., 2005] Talpin, J.-P., Guernic, P. L., Shukla, S. K., and Gupta, R. (2005). A compositional behavioral modeling framework for embedded system design and conformance checking. *International Journal of Parallel Programming*, 33(6) :31.

[Tambe et al., 2009] Tambe, S., Dabholkar, A., and Gokhale, A. (2009). Cqml : Aspect-oriented modeling for modularizing and weaving qos concerns in component-based systems. In *ECBS 2009. 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2009*, pages 11–20. IEEE.

[Tamura, 2012] Tamura, G. (2012). *QoS-CARE : A Reliable System for Preserving QoS Contracts through Dynamic Reconfiguration*. Computer science, Université des Sciences et Technologie de Lille - Lille I Universidad de Los Andes.

[THALES, 2007] THALES (2007). TACTICOS Combat Management System - home page - http ://www.thalesgroup.com/tacticos/ ?pid=1568.

[Thomas and vom Brocke, 2008] Thomas, O. and vom Brocke, J. (2008). A value-driven approach to the design of service-oriented information systems—making use of conceptual models. *Inf Syst E-Bus Manage*, 8 :31.

[Tripathy and Patra, 2011] Tripathy, A. and Patra, M. (2011). Modeling and monitoring sla for service based systems. In *Proceedings of the 2011 International Conference on Intelligent Semantic Web-Services and Applications*, page 10. ACM.

[TYAN and MAHMOUD, 2005] TYAN, J. and MAHMOUD, Q. H. (2005). A comprehensive service discovery solution for mobile ad hoc networks. *Mobile Networks and Applications*, 10(423) :12.

[Vergnaud et al., 2004] Vergnaud, T., Hugues, J., Pautet, L., and Kordon, F. (2004). Polyorb : a schizophrenic middleware to build versatile reliable distributed applications. *Reliable Software Technologies-Ada-Europe 2004*, pages 106–119.

[Verheecke et al., 2003] Verheecke, B., Cibrán, M. A., and Jonckers, V. (2003). Aop for dynamic configuration and management of web services. *Web Services-ICWS-Europe 2003*, pages 55–85.

[Vincent et al., 2010] Vincent, S., Olivier, H., Bruno, C., and Hugues, B. (2010). (wo/2010/060926) method and system for convering ccm software components into components that can be deployed in an sca standard-compatible environment.

[W3C, 2004] W3C (2004). Web services activity, w3c, http ://www.w3.org/tr/ws-arch/.

[Waignier, 2010] Waignier, G. (2010). *Canevas de développement agile pour l'évolution fiable de systèmes logiciels à composants et orientés services*. PhD thesis, LIFL.

[Wang et al., 2006] Wang, J., Zhang, L.-Y., and Han, Y.-B. (2006). Client-centric adaptative scheduling of service-oriented applications. *J. Comput. Sci. & Technol.*, 21(4) :10.

[Weilkiens, 2007] Weilkiens, T. (2007). *Systems engineering with SysML/UML : modeling, analysis, design*. Morgan Kaufmann.

[White et al., 2011] White, J., Dougherty, B., Schantz, R., Schmidt, D., Porter, A., and Corsaro, A. (2011). R&D challenges and solutions for highly complex distributed systems : a middleware perspective. *Journal of Internet Services and Applications*, pages 1–9.

[Willem, 1996] Willem, M. (1996). *Minimax theorems*, volume 24. Birkhauser.

[W.M.P. and van der Aalst, 1999] W.M.P. and van der Aalst (1999). Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10) :639–650.

[Wolfgang, 1994] Wolfgang, P. (1994). *Design patterns for object-oriented software development*. Reading, Mass. : Addison-Wesley.

[Wu et al., 2006] Wu, E., Diao, Y., and Rizvi, S. (2006). High-Performance Complex Event Processing over Streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, number 12 in SIGMOD '06, pages 407–418, New York, NY, USA. ACM.

[Xiao et al., 2008] Xiao, G., Ruan, R., Lu, J., Gao, F., and Zhan, Y. (2008). Research of Reflective Component Model Based on the Separation of Multidimensional Concerns. *E-Business Engineering, IEEE International Conference on*, 0 :759–764.

[Xiong et al., 2007] Xiong, M., Parsons, J., Edmondson, J., Nguyen, H., and Schmidt, D. (2007). Evaluating technologies for tactical information management in net-centric systems. In *Proceedings of the Defense Transformation and Net-Centric Systems conference*. Citeseer.

[Yang et al., 2009] Yang, T.-H., Sun, Y. S., and Lai, F. (2009). A Scalable Healthcare Information System Based on a Service-oriented Architecture. *J Med Syst*, page 17.

[Zeng et al., 2004] Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J., and Chang, H. (2004). Qos-aware middleware for web services composition. *Software Engineering, IEEE Transactions on*, 30(5) :311–327.