

N° d'ordre: 3000

THÈSE

Présentée devant

devant l'université de Rennes 1

pour obtenir

le grade de : DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention INFORMATIQUE

par

Karine MACEDO DE AMORIM

Équipe d'accueil : Triskell

École doctorale : MATISSE

Composante universitaire : IFSIC/IRISA

Titre de la thèse :

*Modélisation d'aspects qualité de service en UML :
application aux composants logiciels*

soutenue le 17 mai 2004 devant la commission d'examen

M. :	Françoise	ANDRÉ	Président
MM. :	Laurence	DUCHIEN	Rapporteurs
	François	TERRIER	
MM. :	Noël	PLOUZEAU	Examineurs
	Jean-Marc	JÉZÉQUEL	

A Mathilde,

Remerciements

Je remercie Françoise ANDRÉ, Professeur à l'Université de Rennes1, qui me fait l'honneur de présider ce jury.

Je remercie Laurence DUCHIEN, Professeur à l'Université des Sciences et Technologies de Lille, et François TERRIER, Professeur à l'INSTN (Institut national des sciences et techniques nucléaires) , d'avoir bien voulu accepter la charge de rapporteur.

Je remercie Noël PLOUZEAU, Maître de conférence à l'Université de Rennes1, d'avoir bien voulu juger ce travail.

Je remercie enfin Jean-Marc JÉZÉQUEL, Professeur à l'Université de Rennes1, qui a dirigé ma thèse.

Je tiens également à remercier tous mes amis et ma famille pour leur soutien et plus particulièrement mes parents, Ludwig, Sandrine, Mary, Thomas, Matthieu et Catherine.

Enfin merci à Nicolas, compagnon de route qui a donné un sens à cette aventure...

Table des matières

I	Introduction	7
II	État de l'art	15
1	Component Based Development	17
1.1	Pourquoi des composants ?	17
1.1.1	Naissance du tout-objet	17
1.1.2	Les limites du tout-objet	18
1.1.3	Après la technologie objet	19
1.1.4	Qu'est-ce qu'un composant ?	19
1.1.5	Les interfaces du composant	20
1.1.6	L'implantation du composant	21
1.1.7	Les différentes abstractions d'un composant	22
1.2	Les différents cycles de vie dans le développement par composants	22
1.2.1	Le cycle de vie du système à base de composants	23
1.2.2	Le cycle de vie d'un composant	26
1.3	Les plateformes d'intégration	27
1.3.1	MetaH	27
1.3.2	PtolemyII	27
1.3.3	Metropolis	28
1.3.4	Conclusion	28
1.4	Les différentes plate-formes d'exécution	28
1.4.1	CCM : CORBA Component Model	28
1.4.2	J2EE : Java 2 Enterprise Edition	30
1.4.3	La plate-forme .NET	32
1.4.4	Quelle architecture choisir ?	34
1.5	Conclusion	36
2	Les contrats de qualité de service	37
2.1	La notion de contrat	38
2.1.1	Définitions	38
2.1.2	Les niveaux de contrats	38
2.2	La qualité de service	40

2.2.1	Définitions	40
2.2.2	Les travaux existants	42
2.2.3	Evaluation des différents travaux	44
2.3	QML : Quality of service Modeling Language	46
2.3.1	Présentation	46
2.4	Problématique	48
2.4.1	Description des contrats	48
2.4.2	Expressions QML d'un contrat de fiabilité	49
2.4.3	Les différents aspects de spécification	50
2.5	Conclusion	55
3	Notation et méthode utilisée	57
3.1	UML : Unified Modeling Language	57
3.1.1	UML et les composants	60
3.1.2	Modélisation de l'exemple du GPS	62
3.2	MDA : Model Driven Architecture	64
3.2.1	Le PIM : Platform Independent Model	65
3.2.2	Le PSM : Platform Specific Model	66
3.2.3	Les différentes transformations	68
3.2.4	Passage d'un PIM vers un PSM	70
3.2.5	Présentation de l'outil Kase	70
3.3	Conclusion	74
III	Contribution	75
4	Les composants et la qualité de service	77
4.1	Introduction	77
4.2	Extension de QML	78
4.3	Le modèle de contrat QoSCL (Quality of Service Constraint Language)	81
4.3.1	Principes	81
4.3.2	Les principaux termes	81
4.3.3	Comparatif QML/QoSCL	84
4.4	Les différents aspects de spécification	85
4.4.1	Délégation de contrats	85
4.4.2	Dépendance de contrats	85
4.4.3	Comportement adaptatif	87
4.4.4	Raffinage de contrat	87
4.5	Exemple du GPS	89
4.6	La qualité de service de bout en bout	91
4.7	Calcul de la qualité de service de bout en bout	92
4.8	Conclusion	95

5	Du modèle à l'implantation	97
5.1	Introduction	97
5.2	Principe	98
5.3	Tissage de l'aspect contractuel	99
5.3.1	Les préconditions OCL	100
5.3.2	Le contrat modélisé en UML	104
5.3.3	Aspect contractuel et surveillance de contrats	104
5.3.4	Le tissage de l'aspect contractuel et de la surveillance de contrats	107
5.3.5	Génération de code	116
5.4	Validation de l'approche	117
5.4.1	MobiForo	117
5.4.2	REGI	118
5.4.3	Le système de télé-médecine	120
5.4.4	Résultat de l'évaluation	120
IV	Conclusion	123
V	Annexes	131
A	Grammaire étendue de QML	133
B	Partie du code de tissage du contrat <i>TimeOutC</i> en MTL	137
C	Code du tissage du contrat <i>TimeOutC</i> en python	145
D	code généré par <i>Kase</i> pour l'exemple du contrat <i>TimeOutC</i>	153
E	Transformation en PrologIV	157

Première partie

Introduction

Les composants logiciels

Les systèmes aujourd’hui sont de plus en plus complexes, et créer un logiciel de haute qualité demande énormément de temps et de moyens humains.

La réutilisation de code est souvent suggérée pour améliorer la productivité et la qualité d’un développement logiciel [49], [12]. Selon l’étude de Wayne C. Lim dans [49], la réutilisation de code permet une meilleure compréhension de l’application et diminue le nombre de bugs par rapport au développement d’un nouveau code.

La principale technique dans la réutilisation de code réside dans l’utilisation des composants logiciels (*software components*) dont la définition a beaucoup évolué en trente cinq ans :

- En 1968, lors de la conférence de NATO sur l’ingénierie logicielle, McIlroy introduit la notion de réutilisation du logiciel [54] afin de répondre à la crise du logicielle. L’idée est de permettre aux développeur de créer une application à partir de composants logiciels existants sur étagères.
- En 1987, Grady Booch, dans son livre *Software Components with Ada : Structures, Tools, and Subsystems* [13] explique qu’un composant logiciel réutilisable est un module cohésif et faiblement couplé qui indique une abstraction simple : cette définition ignore les dépendances environnementales.

- En 1996, dans leur ouvrage *The Essential Distributed Objects : Survival Guide* [58], les auteurs Robert Orfali, Dan Harkey et Jeri Edward, citent Jed Harris, président à l’époque du CI Labs (*Component Integration Laboratories*)[17], pour sa définition de composant :

un composant est un morceau de logiciel assez petit à créer et à maintenir, et assez grand pour se déployer. Un composant a des interfaces standards pour l’interopérabilité.

Dans cette définition, il n’y a pas les notions de services requis ni de dépendances.

- En 1999, Desmond D’Souza et Alan Wills définissent dans leur livre *Objects, Components and Frameworks with UML : The Catalysis Approach* [24] un composant comme un package d’*artifacts* logiciels qui peuvent être développés indépendamment les uns des autres et livrés comme des unités. Ils ajoutent qu’un composant a des interfaces explicites et où sont spécifiés les services qu’il fournit ainsi que les services qu’il requiert. Un composant peut être construit à partir d’autres composants, sans modification de ceux-ci.
- En 2002, Clemens Szyperski dans la deuxième édition de son ouvrage *Component Software Beyond Object-Oriented Programming* [64] définit un composant logiciel comme une unité de composition ayant des interfaces contractualisées ainsi que des dépendances contextuelles. Ces dépendances explicitent les interfaces exigées ainsi que les plate-formes d’exécutions possibles.

Un composant peut être déployé indépendamment, et être sujet à la composition. De ce point de vue, le composant est une unité exécutable. Dans le cadre de cette thèse, nous nous basons sur cette définition.

La réutilisation de composants impliquent deux approches distinctes et complémentaires [28] :

- l’ingénierie des composants réutilisables (*Design for reuse*) pour la conception des composants proprement dit qui correspond à l’aspect «infrastructure»de l’application,
- l’ingénierie des applications composites (*Design by reuse*) pour la réutilisation des composants, qui correspond à l’aspect «opérationnel»de l’application.

Les contrats de qualité de service

Aujourd’hui, le service rendu n’est plus le seul critère de choix : la qualité du service compte de plus en plus. Prenons l’exemple de la téléphonie mobile. Sur le marché, un utilisateur va avoir le choix entre plusieurs opérateurs qui vont lui fournir le même service : téléphoner à partir d’un téléphone portable. Le choix de l’opérateur va donc devoir s’effectuer sur d’autres critères qui vont être le taux de couverture, la saturation des lignes, la fiabilité ou encore la sécurité de ces services.

Les utilisateurs veulent donc non seulement un service, mais ils veulent l’obtenir dans les meilleures conditions. Toutes ces conditions sont des propriétés extra-fonctionnelles du service et définissent in-fine sa qualité. Ces propriétés sont directement dépendantes des propriétés extra-fonctionnelles des composants en interaction qui constituent ce service.

Les propriétés fonctionnelles correspondent aux spécifications du composant, c’est-à-dire ce qu’il est capable de faire : les services qu’il fournit. Mais les composants ont également des propriétés additionnelles telles que la fiabilité, la disponibilité, le temps d’exécution, la sécurité ou encore la synchronisation. Ces aspects extra-fonctionnels peuvent être vus comme des propriétés de qualité de service (QdS) [26].

Ces propriétés influent sur le développeur dans le choix d’un composant. Svend Frolund et Jari Koistinen dans [26] montrent qu’il est essentiel de prendre en compte cette qualité de service dès la phase de conception.

L’évaluation de la QdS doit être effectuée de bout en bout tout au long de l’exécution de l’application. Les besoins de QdS attachés à l’application vont être transférés aux composants qui la constituent. Un procédé de négociations permet de déterminer si les composants sont capables de satisfaire le niveau de QdS demandé par l’application.

Le concepteur doit spécifier la qualité de service souhaitée mais aussi mettre en

place un système de surveillance pour s'assurer de la qualité effectivement rendue ainsi qu'un mécanisme de renégociation des contrats. Le concept de contrat de qualité de service va constituer le concept fondamental autour duquel va s'articuler la contribution de cette thèse.

Les contrats définissent clairement les contraintes entre les différentes ressources [42]. Ils spécifient les droits et les obligations entre un client et un fournisseur de service. Ils déterminent également les relations entre les différents acteurs. La notion de contrat va permettre à l'application d'avoir un retour d'information et de pouvoir ainsi réagir à son environnement par le mécanisme de renégociation. Le contrat s'exprime sous la forme de contraintes à respecter en temps réel.

Actuellement, les services offerts par un composant sont le plus souvent décrits de manière fonctionnelle alors que les propriétés extra-fonctionnelles sont absentes de la spécification. Le projet européen QCCS (Quality Controlled Component-based Software development) a mis en place une méthodologie utilisant des contrats pour spécifier et garantir la qualité des composants logiciels.

Le projet européen QCCS (Quality Controlled Component-based Software development)

Les travaux présentés dans ce document ont été effectués dans le cadre du projet européen QCCS [62]. Ce projet a développé une méthodologie et des outils pour la conception et la mise en œuvre de composants distribués et contractualisés, en s'appuyant sur la conception et la réalisation par aspects (aspect-oriented programming ou aop). Dans le cadre de ce projet, il a fallu :

- développer une méthodologie de conception de composants à contrats,
- étudier plus précisément les propriétés de qualité de service (QdS),
- concevoir une infrastructure de développement et d'outils,
- valider l'approche au moyen de trois applications.

Les partenaires du projet QCCS sont :

- *Technische Universität Berlin (TUB)*, pour l'infrastructure globale de la plateforme,
- *l'IRISA* pour la méthodologie et le système de composition d'aspects,
- la société *Schlumberger Sema Espagne*, fournisseur de la première application,
- la société *kd software*, PME tchèque, fournisseur de la seconde application,
- *l'Université de Chypre*, fournisseur de la troisième application.

Contribution de la thèse

La contribution de cette thèse est de proposer, dans le cadre du projet européen QCCS, une méthodologie pour la spécification des contrats de qualité de service de composants logiciels, et leur intégration dans un environnement de développement par composants. Dans un premier temps, la notation UML a été étendue pour offrir aux concepteurs de logiciels à objets un outil de spécification et de conception traitant la qualité de service. Dans le modèle de composant obtenu, les aspects fonctionnels et non fonctionnels sont séparés. Dans un deuxième temps, un outil permettant la génération automatique des composants a été mis en place. Pour ce faire, nous avons utilisé la méthode du tissage d'aspect inspirée des travaux sur la programmation par aspects [43]. Cette méthode nous a permis au niveau conceptuel, de lever la séparation des différents aspects et d'introduire les mécanismes de surveillance et de renégociation des contrats.

Plan de la thèse

La première partie présente les différentes notions introduites ci-dessus, avec tout d'abord les notions de composant logiciel et de systèmes à base de composants, suivies d'un aperçu des principales approches sur le marché des composants, dont les leaders sont l'OMG (Corba et CCM), Sun (J2EE et EJB) et Microsoft (.Net). Ensuite, nous parlerons des contrats de qualité de service et nous étudierons un langage de modélisation de qualité de service particulier : QML (Quality of service Modeling Language) dont nous nous sommes inspirés pour notre modèle de contrat. Puis nous terminerons ce chapitre consacré aux contrats de qualité de service par une présentation de la problématique. Pour clore cet état de l'art, nous aborderons la notation UML et la méthode MDA, toutes deux proposées par l'OMG.

La deuxième partie porte sur la contribution de cette thèse. Le premier chapitre de cette contribution porte sur notre modèle de contrat. Avec tout d'abord la présentation de l'extension de QML qui nous a permis de *capturer* les notions importantes pour notre méthodologie. Puis nous présentons notre modèle de contrat appelé QoSCL (Quality of Service Constraint Language) avec l'extension faite dans UML2.0. Nous terminerons ce chapitre par les différents aspects de spécification de QoS offerts par notre modèle de contrat.

Le dernier chapitre porte sur les transformations de modèles c'est-à-dire comment passer d'un modèle de composant sans contrat de qualité de service à un modèle de composant avec contrat de qualité de service grâce au tissage d'aspect. Nous présenterons la transformation de modèle utilisée dans QCCS avec l'outil *Kase*. Ensuite, nous verrons une transformation de modèle décrite en MTL (Modeling Transformation Language). MTL est un langage qui permet l'écriture de transformation de modèles indépendamment des métamodèles utilisés. Enfin, nous terminerons par l'évaluation de l'approche par les différents partenaires de QCCS.

Pour illustrer cette méthodologie, nous utiliserons tout au long de cette thèse l'exemple d'un GPS (*Global Positioning System*) simplifié qui permet de calculer la position courante de l'utilisateur, de donner le cap, la vitesse ainsi que le chemin parcouru à partir des positions successives de l'utilisateur.

Deuxième partie

État de l'art

Chapitre 1

Component Based Development

Contents

1.1	Pourquoi des composants ?	17
1.1.1	Naissance du tout-objet	17
1.1.2	Les limites du tout-objet	18
1.1.3	Après la technologie objet	19
1.1.4	Qu'est-ce qu'un composant ?	19
1.1.5	Les interfaces du composant	20
1.1.6	L'implantation du composant	21
1.1.7	Les différentes abstractions d'un composant	22
1.2	Les différents cycles de vie dans le développement par composants	22
1.2.1	Le cycle de vie du système à base de composants	23
1.2.2	Le cycle de vie d'un composant	26
1.3	Les plateformes d'intégration	27
1.3.1	MetaH	27
1.3.2	PtolemyII	27
1.3.3	Metropolis	28
1.3.4	Conclusion	28
1.4	Les différentes plate-formes d'exécution	28
1.4.1	CCM : CORBA Component Model	28
1.4.2	J2EE : Java 2 Enterprise Edition	30
1.4.3	La plate-forme .NET	32
1.4.4	Quelle architecture choisir ?	34
1.5	Conclusion	36

1.1 Pourquoi des composants ?

1.1.1 Naissance du tout-objet

Face à une croissance constante de la taille et de la complexité des systèmes informatiques, leur conception est devenue modulaire : le développement de ces systèmes est

réparti entre différentes équipes [21]. Une première approche de la modularité porte sur la notion de procédure. La programmation procédurale consiste à analyser un problème par décomposition fonctionnelle descendante. Les types de données manipulées sont définis par ailleurs. Cette séparation dans l'analyse entre les données structurelles et fonctions rend très difficile la maintenance et la réutilisation du code. La programmation objet dont l'origine remonte à *Simula* en 1966 [22], va rapprocher et regrouper les fonctions et les types de données qu'elles manipulent au sein d'une même entité logique : la classe. Les données, appelés attributs, sont les caractéristiques structurelles statiques d'une classe, et les fonctions, appelées méthodes, en sont les caractéristiques dynamiques comportementales. Un objet est une instantiation particulière d'une classe, c'est-à-dire un ensemble de valeurs connues pour ses attributs. Un problème est alors analysé comme un ensemble d'objets en interaction. La maintenance et la réutilisation d'un programme objet est assurée par les mécanismes d'héritage (spécialisation de classes, ajout d'attributs) et de polymorphisme (redéfinition de méthodes).

Dans les années 1980, la technologie objet a progressivement pris le pas sur la technologie procédurale dans l'industrie.

«En raison des formidables capacités unificatrices du paradigme objet, le passage de la technologie procédurale à la technologie des objets apportera une énorme simplification conceptuelle pour l'ingénieur logiciel. Comme tout sera considéré comme objet (principe du tout objet), on assistera à une formidable réduction du nombre de concepts nécessaires.»(Selon Jean Bézivin, Circa 1980)

Pourtant, malgré l'enthousiasme général, la communauté objet s'aperçoit que cette technologie ne tient pas toutes ses promesses.

1.1.2 Les limites du tout-objet

Même si la technologie objet a apporté de grands bouleversements dans le domaine informatique dans les années 1990, celle-ci n'est pas suffisante pour répondre aux besoins devenus de plus en plus complexes. La réutilisabilité des objets est limitée à cause de certains problèmes liés au paradigme objet, comme l'anomalie de l'héritage dans les langages concurrents [52] ou encore le problème de l'encapsulation des aspects transversaux [43] comme la gestion de la répartition, la persistance, la tolérance aux défaillances, etc. De plus, il est difficile d'intégrer des objets hétérogènes conçus pour différents contextes applicatifs, en particulier si ils sont dépendants de leur contexte d'exécution. Ceci est dû aux interactions qui sont insérées dans les différentes parties du code fonctionnel, et donc occultées dans le code des objets [9].

Il fallait donc trouver un mécanisme d'appel de méthodes qui ne connaisse pas explicitement l'objet appelé. Ce mécanisme est proposé dans le paradigme à composants qui va permettre une meilleure réutilisation que ne le permet le modèle à objets. En effet, ce paradigme définit des composants totalement indépendants de l'environnement logiciel

dans lequel ils vont être utilisés. Un composant définit une interface de communication qui contient des points d'entrée et de sortie appelés *ports*. Un point de sortie va être connecté à un ou plusieurs points d'entrée. Un composant va donc exporter l'ensemble des services qu'il propose mais aussi l'ensemble des services qu'il requiert (voir partie II §1.1.5 page 20).

1.1.3 Après la technologie objet

Aujourd'hui, le développement par composants est perçu comme un atout majeur pour le développement des systèmes complexes. C'est une avancée considérable dans le domaine de la méthodologie du développement logiciel. Cette avancée est comparable à la transition entre la programmation procédurale et les langages à objets dans les années 1990.

La technologie des composants permet la structuration des systèmes complexes dans le processus de développement. Un système composite, c'est-à-dire composé de composants, permet une structuration claire et donc une maintenance facilitée. Grâce à la spécification des composants et à la définition d'architectures de lignes de produits, les composants peuvent être réutilisés pour différents produits ou familles de produits, même s'ils sont développés par des organisations complètement différentes avec des technologies elles aussi différentes.

1.1.4 Qu'est-ce qu'un composant ?

L'objectif premier du développement par composants est de structurer un système logiciel en composants vus comme des unités de déploiement.

Clemens Szyperski définit trois principales propriétés pour un composant [64] :

- un composant est une unité de déploiement indépendante ;
- un composant est une unité de composition ;
- un composant n'a pas d'état observable au niveau de l'environnement qui l'entoure.

Ces propriétés vont donc avoir plusieurs implications. Premièrement, le fait qu'un composant soit une unité de déploiement indépendante implique que celui-ci doit être entièrement séparé de l'environnement et des autres composants. D'autre part, celui-ci ne pourra jamais être déployé partiellement. Un composant pouvant être composable avec d'autres composants, il doit avoir une spécification précise de ses besoins mais aussi de ce qu'il peut fournir. En d'autres termes, un composant encapsule son implantation et interagit avec l'environnement qui l'entoure par des interfaces clairement définies. Finalement, le fait qu'un composant n'ait pas d'état observable au niveau de l'environnement qui l'entoure implique qu'on ne pourra pas le distinguer des autres composants qui offrent des services identiques.

Le contrat de déploiement du composant spécifie ses dépendances, ses interfaces, ses modes de déploiement et d'instantiation ainsi que le comportement de ces instances à travers les interfaces offertes.

La définition de composant que nous retiendrons dans ce mémoire est celle donnée par Clemens Szyperski dans la deuxième édition de son ouvrage *Component Software Beyond Object-Oriented Programming* [64] :

Définition 1 *Un composant logiciel est une unité de composition ayant des interfaces contractualisées ainsi que des dépendances contextuelles. Ces dépendances explicitent les interfaces exigées ainsi que les plate-formes d'exécutions possibles. Un composant peut être déployé indépendamment, et être sujet à la composition. De ce point de vue, le composant est une unité exécutable.* \square

1.1.5 Les interfaces du composant

L'interface d'un composant résume les propriétés qui sont visibles depuis son environnement extérieur. Elle va lister les différentes signatures des opérations fournies par le composant. Cette liste va permettre d'éviter les erreurs de typage au niveau des connexions du composant puisque les composants se connectent *via* leurs interfaces (*via* leurs ports). Les informations contenues dans les interfaces d'un composant facilitent donc la vérification de l'interopérabilité entre composants, et permettent à certaines propriétés d'être vérifiées plus tôt dans le cycle de conception (par exemple la vérification statique des erreurs de typage).

Techniquement, une interface est un ensemble d'opérations nommées qui vont être invoquées par le client. La sémantique de chaque opération est spécifiée. Cette spécification joue un double rôle car elle permet :

- au fournisseur d'implanter le composant ;
- au client de pouvoir utiliser l'opération.

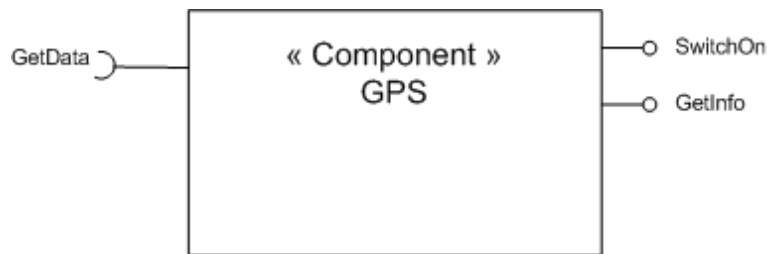
Le fournisseur et le client ne se connaissent pas, la spécification de l'interface devient donc le médiateur qui permet aux deux parties de travailler ensemble. Un composant peut :

- soit fournir directement des interfaces qui correspondent aux interfaces procédurales des bibliothèques classiques ;
- soit implanter des objets qui, si ils sont accessibles par les clients, fournissent des interfaces. Ces interfaces implantées indirectement correspondent aux interfaces des objets contenus dans le composant.

Une spécification de composant nomme toutes les interfaces auxquelles pourraient adhérer un composant et ajoute toutes les propriétés spécifiques au composant.

Les interfaces fournies par le composant sont représentés par un cercle et les interfaces requises par un demi-cercle.

Prenons l'exemple d'un GPS (Global Positioning System) simplifié, voir figure 1.1 :

FIG. 1.1 – Le composant *gps*.

Une fois allumé, le GPS demande à son utilisateur (que ce soit un système ou un être humain) sa position approximative et avec quelle précision il veut cette donnée ; il essaye alors de déterminer quels satellites couvrent la zone de positionnement donnée, et choisit des fréquences de récepteurs pour écouter ces satellites.

Les services offerts par ce GPS sont les suivants :

- *SwitchOn* : qui permet d’allumer (ou éteindre) le GPS et de l’initialiser. Dans la phase d’initialisation, l’utilisateur doit donner sa position approximative (au moins dans quelle partie du monde l’utilisateur est censé être) ainsi que la précision souhaitée des résultats ;
- *GetInfo* : qui renvoie les informations tel que la position courante, le cap, la vitesse ;

Au moins trois satellites sont nécessaires pour calculer la position de l’utilisateur dans l’espace ; plus il y a de satellites, meilleure sera la précision de la localisation. Pour renvoyer la position courante, le GPS requiert les informations des différents satellites qu’il « écoute » : ce besoin s’exprime par l’interface requise *GetData*.

La spécification du composant *GPS* doit spécifier que l’interface fournie *GetPosition* est liée à l’interface requise *GetData* ; pour calculer la position courante, on a besoin des informations données par les différents satellites. Toutes les implantations de ce composant doivent être conformes à cette spécification.

1.1.6 L’implantation du composant

L’implantation du composant est la réalisation exécutable du composant, obéissant aux règles du modèle de composant. Dépendant du modèle de composant, l’implantation du composant peut être fournie en code compilable C, en forme binaire etc...

Il n’est pas nécessaire qu’un composant contienne seulement des classes, ou même contienne de classes du tout. Il peut :

- contenir simplement des procédures classiques et des variables globales,
- être écrit entièrement dans une approche de programmation par fonctions,
- utiliser un langage assembleur,
- ...

1.1.7 Les différentes abstractions d'un composant

L'abstraction d'un composant correspond à la visibilité de son implantation. On trouve trois sortes d'abstractions :

Blackbox (Boîte noire) : Le client ne connaît aucun détail au-delà des interfaces et de leurs spécifications. La réutilisation d'une boîte noire se réfère au concept de réutilisation de l'implantation en reliant les interfaces entre elles.

Whitebox (Boîte blanche) : L'implantation d'une boîte blanche est entièrement disponible et peut donc être étudiée afin d'augmenter sa compréhension. La réutilisation d'une boîte blanche s'apparente à l'utilisation d'un fragment de logiciel à travers ses interfaces, tout en tenant compte de la compréhension acquise par la connaissance de l'implantation. On peut trouver dans la littérature, le terme de *Glassbox* ou *Boîte transparente*. Quand la distinction est faite cela signifie que la boîte blanche permet la manipulation de l'implantation alors que la boîte transparente permet simplement l'étude de l'implantation.

Graybox (Boîte grise) : Seule une partie contrôlée de l'implantation est visible.

1.2 Les différents cycles de vie dans le développement par composants

Le CBSE (*Component Based Software Engineering*) [36] utilise les méthodes, outils et principes de l'ingénierie logicielle classique. Cependant, la CBSE distingue deux cycles de vie : un pour le développement du composant (*Design for reuse*) et un autre pour le développement d'un système à base de composants (*Design by reuse*). On va avoir une approche différente pour chacun des cas.

- Dans la phase de développement du composant, la réutilisabilité est l'enjeu principal : les composants sont créés pour être utilisés mais surtout réutilisés dans différentes applications, dans la plupart des cas, celles-ci n'existant pas encore. Un composant doit être spécifié précisément et formellement, facile à comprendre, assez générique, facile à adapter, à délivrer, à déployer et à replacer. La spécification d'un composant est donnée par ses interfaces sous forme de méthodes externes, séparées de l'implantation du composant. Dans la plupart des cas, une interface spécifie uniquement les propriétés syntaxiques. Une interface qui spécifie aussi les propriétés comportementales ou non fonctionnelles est appelée une *interface enrichie*.
- Le développement d'un système à base de composants se focalise sur l'identification des entités réutilisables et des relations qui les relient entre-elles. L'implantation du système n'est pas la partie la plus longue. Le plus laborieux se situe au

niveau du choix des composants : les localiser, sélectionner le plus approprié, les tester, les vérifier etc...

On peut donc distinguer deux cycles de vie : le cycle de vie du composant logiciel en lui même et celui du système à base de composants.

1.2.1 Le cycle de vie du système à base de composants

Le développement d'un système à base de composants diffère d'un système traditionnel : il se focalise sur l'identification des entités réutilisables et leurs relations. L'identification initiale des besoins est déterminée comme pour un développement traditionnel avec une attention particulière pour la cohérence entre le système et les exigences des composants. La réutilisabilité étant l'enjeu principal dans le développement de système à composants, lors de la phase d'identification des besoins du système, les besoins des composants devront être identifiés.

Le cycle de vie d'un système à base de composants comporte cinq phases :

La phase de conception de l'application qui comporte deux étapes essentielles :

- La vue logicielle du système qui spécifie les architectures du système en terme de composants et leurs interactions. Dans cette vue, les composants sont représentés par la spécification de leurs interfaces, on pourra inclure la spécification des propriétés non fonctionnelles. Dans les systèmes temps réel, on pourra inclure les propriétés temporelles.
- La vue structurelle spécifie les architectures du système constituées des implantations des différents composants. L'implantation d'un composant doit être conforme à un modèle de composant particulier décrit pour une plate-forme d'exécution particulière, à un *framework* de composant et aux différents services qui sont spécifiques à la technologie employée. Le modèle de composant et le *framework* ont un impact significatif dans la solution de conception : ils doivent donc être pris en compte au plus tôt dans la phase de conception.

Le processus de spécification de l'architecture est combiné par la recherche, l'évaluation, la sélection et l'adaptation des composants qui correspondent aux rôles définis par l'architecture du système. Les besoins du système vont être reformulés afin de faciliter l'adaptation des composants disponibles pour ce système : la conception du système doit tenir compte des besoins des différents composants et du système. Un modèle de composant contenant des interfaces enrichies est capable de vérifier les besoins du système et les prédictions de propriétés du système à partir des propriétés des composants.

La phase d'implantation inclut l'adaptation, la composition et le déploiement des composants grâce à un *framework* de composant.

La phase de vérification ou test vérifie le système en le testant. Un modèle de composant enrichi permet à une partie significative du système d'être vérifiée dès la phase de conception, ce qui permet une économie considérable dans la phase de test.

La phase de maintenance permet le remplacement ou la mise à jour des composants du système.

Nous pouvons maintenant dresser un bilan des activités spécifiques au développement des systèmes à composants.

- *Spécifier l'architecture fonctionnelle et structurelle du système.* Cette architecture est le résultat de la conception, fondée sur les besoins du système pour lesquels les méthodes de conception sont utilisées. Le processus de spécification de l'architecture doit prendre en compte les besoins du système qui doivent être compatibles avec les besoins des composants disponibles. De plus, la sélection d'une technologie particulière doit être prise en considération : une technologie de composant peut requérir une implantation particulière et inclure des services spécifiques tels que la communication inter-composants.
- *Trouver et sélectionner les composants qui vont être utilisés dans le système.* Pour réussir cette étape, il faut un nombre significatif de candidats possibles ainsi que des outils pour les trouver. On pourra utiliser un dépôt (*repository*) pour stocker ces composants.

Un processus de sélection de composants possibles, par approximations successives, est présenté par la figure 1.2 [25]. Dans un premier temps, on crée une liste des candidats potentiels. Une fois cette liste complète, on analyse chaque composant et on décide si on l'inclut ou non dans notre système. La phase d'analyse va déterminer :

- l'adéquation du composant, c'est-à-dire déterminer les ajustements nécessaires pour intégrer ce composant dans le système ;
- les dépendances du composant ;
- le coût de l'adaptation de ce composant dans le système ;
- les risques associés à ce composant, c'est-à-dire sa qualité, sa maintenance, ses performances, etc ;
- l'impact sur le système, c'est-à-dire déterminer si l'inclusion de ce composant dans le système modifie les composants déjà sélectionnés et enfin raffiner le système si nécessaire.

Le processus de sélection se termine lorsque tous les composants ont été sélectionnés et qu'il n'y a plus de changement à effectuer dans l'architecture.

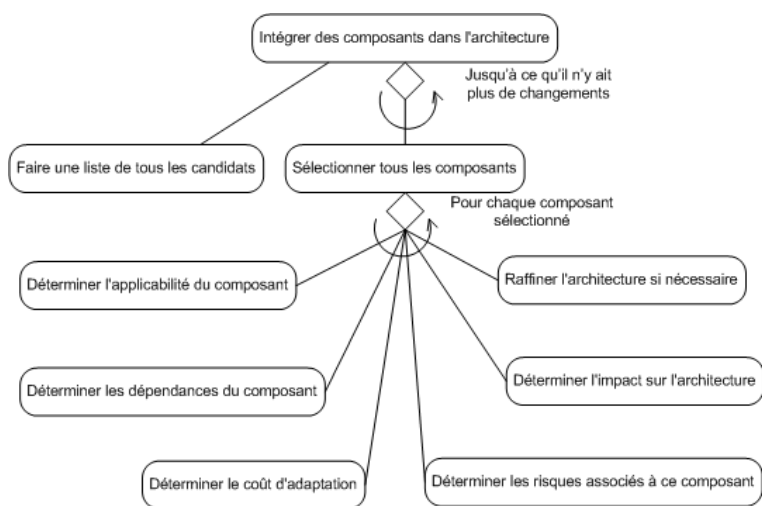


FIG. 1.2 – Processus de sélection de composants.

Trouver des composants, les tester dans un environnement particulier et les stocker dans une base de données sont des activités qui peuvent être séparés du développement du système. Par contre, la manière dont les composants vont être stockés dans la base, le choix des *catégories* de composants ainsi que les critères de recherches disponibles pour une base particulière vont influencer l'usage du composant. Souvent, les besoins ne sont pas complètement formulés et une analyse *a posteriori* est nécessaire pour ajuster l'architecture au système et reformuler les besoins afin de rendre possible l'utilisation des composants existants.

- *Associer les besoins du composant avec les besoins du système et vérifier les propriétés du système à partir des propriétés des composants.* L'un des objectifs de la recherche aujourd'hui est de pouvoir prédire les propriétés d'un système à partir des propriétés des composants qui le constituent. Dans le cas des propriétés non fonctionnelles, les interfaces enrichies sont essentielles.
- *Créer un composant spécifique aux besoins du système.* Dans de nombreux cas, il est impossible de définir entièrement un système à partir de composants existants. On va, en particulier, vouloir créer certains composants *noyaux* du système qui pourront alors fournir un avantage concurrentiel au produit final. Les parties créées de cette façon devront être conçues comme des composants avec des interfaces bien définies, afin de permettre leur réutilisation dans les applications avenir et de faciliter leur maintenance. Cette alternative demande en générale plus d'effort et de temps que l'adaptation des composants existants.
- *Adapter les composants sélectionnés afin de suivre le modèle de composant existant ou la spécification des besoins.* Certains composants peuvent être intégrés directement dans le système, d'autres ont besoin de modification, soit par un pro-

cessus de paramétrage ou encore par ajout d'un code qui permet une meilleure adaptation. Dans certains cas, il n'est pas possible de réutiliser le composant lui-même, mais seulement son interface, qui doit être raffinée et implantée à nouveau.

- *Composer et déployer les composants en utilisant un framework pour composants.* Typiquement, les *frameworks* pour composants ne fournissent que les services. Pour obtenir une fonction particulière, les composants doivent souvent se composer dans une *assembly*. En introduisant les *assemblies* dans le système, des conflits entre composants élémentaires peuvent apparaître. Il se peut, par exemple, que plusieurs *assemblies* soit composées d'un même composant basique mais dans des versions différentes. Pour régler de tels conflits, il existe des mécanismes de reconfiguration des *assemblies*, soit supportés par le framework pour composants, soit utilisés manuellement.
- *Mettre à jour au plus tôt les composants avec leurs dernières versions.* Ceci correspond à la phase de maintenance du système. Les implantations des composants, et donc le système entier, peuvent évoluer avec le temps. Certains bugs peuvent être éliminés et de nouvelles fonctionnalités peuvent apparaître.
- L'élimination des bugs dans les implantations de composants, lesquels n'affectent pas les interfaces, doit être complètement indépendante du comportement du système. Dans le meilleur des cas, ceci exige tout au plus une validation de la nouvelle implantation au niveau de l'interface.
- Toute évolution du système qui affecte ses interfaces requiert une validation additionnelle au niveau du système. Si une fonctionnaire est ajoutée, la validation minimale consiste à vérifier que cette nouvelle fonctionnalité n'est pas utilisée de manière indésirable par d'autres composants.

Pour rendre la réutilisation des composants efficace, il est nécessaire de s'occuper de la maintenance et de l'utilisation des bibliothèques de composants logiciels (*artifact repositories*). La réutilisation de composant logiciel englobe trois tâches de base :

1. construire une bibliothèque de composants logiciels,
2. indexer les composants,
3. récupérer le composant logiciel sélectionné grâce à un processus d'identification.

1.2.2 Le cycle de vie d'un composant

Le processus de développement d'un composant est, par plusieurs aspects, similaire au développement d'un système ; les besoins doivent être capturés, analysés et définis, le composant doit être conçu, implanté, vérifié, validé et livré. Lors de la construction d'un nouveau composant, les développeurs peuvent réutiliser d'autres composants et utiliser des procédures d'évaluation du composant semblables à celles utilisées pour le développement d'un système. Cependant, il y a quelques différences significatives : les

composants sont construits pour faire partie de quelque chose. Ils vont être réutilisés dans différents produits. Ceci a pour conséquence :

- Plus de difficultés dans la gestion des besoins, causées par l'interaction entre le composant et le système.
- Plus de précisions dans la spécification du composant.
- De plus gros efforts sont nécessaires pour créer des unités réutilisables.
- Plus de rigueur et de documentation dans la vérification de la spécification des composants surtout lors des transferts de composants entre organisations.

Une fois que le composant a été testé, spécifié, stocké dans une librairie de composants, la prochaine étape dans le cycle de vie du composant est la phase de déploiement dans le système. Le déploiement du composant consiste à son enregistrement dans le système ainsi qu'à établir la communication avec le reste du système. Cette communication est obtenue par lien dynamique entre les interfaces du composant et le système. Le déploiement doit se faire de manière automatique et sans provoquer de changement dans le reste du système. Un modèle de composant fournit le support pour lier le composant dans le système par un ensemble de fonctions ou de composants qui sont spécifiques à la plate-forme.

1.3 Les plateformes d'intégration

Dans cette section, nous présentons trois plates-formes qui permettent de modéliser les systèmes composés de composants hétérogènes. Un système est représenté comme une architecture de composants interconnectés. Ces composants ont souvent des langages et des formalismes différents. Lors de la composition, le code «glue» est généré automatiquement.

1.3.1 MetaH

MetaH est un langage et un ensemble d'outils pour développer des architectures fiables de systèmes avioniques temps réels et multiprocesseurs. Le langage MetaH décrit les interfaces et les propriétés du logiciel afin de les combiner dans un système intégré global. MetaH permet l'automatisation des procédés d'intégration de système, l'orchestration de la conformité et des liens nécessaires afin de combiner les morceaux de logiciel selon la configuration de matériel indiquée.

1.3.2 PtolemyII

PtolemyII est un ensemble hétérogène de support de packages Java permettant la modélisation et la conception. Le package *kernel* supporte les graphes hiérarchiques, qui sont des collections d'entités et de relations entre ces entités. Le package *actor* étend le package *kernel* afin que les entités aient des fonctionnalités et puissent communiquer

via des relations. L'extension du package *actor* se fait grâce à la notion de domaines avec l'ajout de modèles permettant le calcul des interactions entre les entités.

1.3.3 Metropolis

Metropolis [7] est un environnement pour la conception des systèmes hétérogènes embarqués. Le *framework* est fondé sur une représentation générale du système appelée le méta-modèle Metropolis. Ce modèle forme le squelette du système logiciel et est employé pour permettre l'intégration d'outils d'analyse et de synthèse. La modélisation compositionnelle est utilisée pour assembler des composants en satisfaisant un ensemble de propriétés.

1.3.4 Conclusion

Ces outils de conception permettent de concevoir des systèmes par assemblage de composants hétérogènes. L'avantage de ces outils est qu'ils soutiennent une variété de notations de conception. Cependant, les composants ne peuvent être assemblés que dans l'outil de support, signifiant que les différentes phases de développement doivent nécessairement être développées dans le même environnement.

1.4 Les différentes plate-formes d'exécution

Aujourd'hui, on distingue trois principales approches sur le marché des composants. Tout d'abord, nous parlerons de l'approche fondée sur le standard CORBA qui émerge principalement du monde de l'informatique d'entreprise et de l'intégration d'application. Ensuite nous verrons l'approche SUN dont les origines proviennent surtout d'Internet. Enfin, nous terminerons par l'approche Microsoft qui évolue dans le secteur des systèmes d'exploitation et de la bureautique.

1.4.1 CCM : CORBA Component Model

Le CCM (*CORBA Component Model*) a été définie par l'OMG [34]. Cette organisation, fondée en 1989 par onze compagnies, est devenu le plus grand consortium dans le domaine de l'industrie informatique avec environ 800 membres. Cet organisme de normalisation international a pour but de standardiser les technologies objets pour une meilleure interopérabilité des solutions logicielles. La norme CORBA (Common Object Request Broker Architecture) [32], sur lequel repose CCM, a été définie à partir de 1991 par l'OMG.

CCM [33] propose toute une structure pour définir un composant, son comportement, son intégration dans un container (application), et son déploiement dans l'environnement distribué CORBA. openCCM est une plate-forme ouverte qui permet de concevoir, implanter, compiler, déployer et exécuter des applications réparties conformes au CCM [51]. La structure de CCM est composée de quatre modèles :

Le modèle abstrait qui spécifie en IDL (Interface Definition Language) le composant en terme de ports et d'interfaces. IDL est un langage normalisé par l'OMG qui permet de définir les interfaces d'accès à un objet. Le composant CORBA est constitué de (voir figure 1.3) :

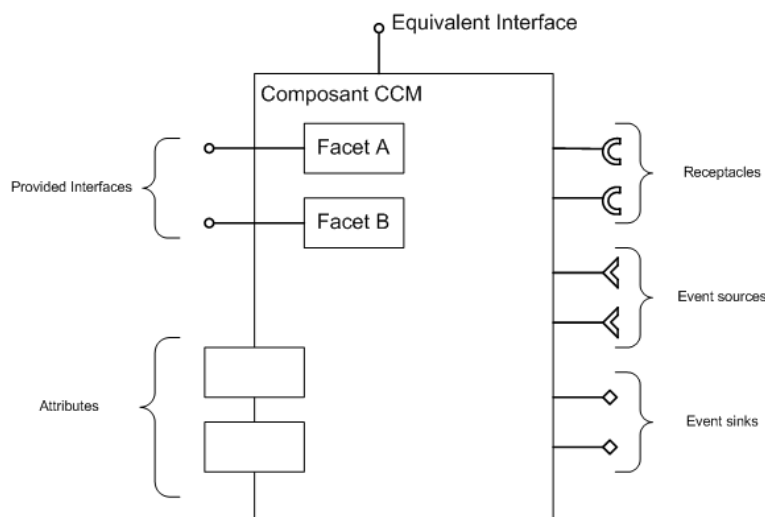


FIG. 1.3 – Le composant CCM.

- d'une référence,
- de quatre types de ports :
 1. **Facet** : c'est une interface fournie par le composant.
 2. **Receptacle** : c'est une interface requise qui permet au composant de recevoir et d'exploiter des données fournies par d'autres composants via un lien dynamique.
 3. **Event source** : permet de diffuser un type d'événement vers des *Event sinks*.
 4. **Event sink** : permet de recevoir des événements.
- des attributs,
- des fabriques qui contrôlent le comportement du composant en fonction de son cycle de vie et qui gère les instances du composant (création, gestion de références, recherche, exploration,...)

Dans les **facets** du composant, une interface particulière nommée *Equivalent Interface*, permet la navigation entre les différents **facets** du composant.

Le modèle d'implantation qui décrit le comportement du composant. Ce modèle est écrit en CIDL (Component Implementation Description Language). La compilation du code CIDL permet de tisser le framework du composant appelé CIF (Component Implementation Framework).

A partir de ce framework, le développeur va implanter les différentes méthodes du composant.

Le modèle de container Chaque instance de composant est placée dans un container (voir figure 1.4).

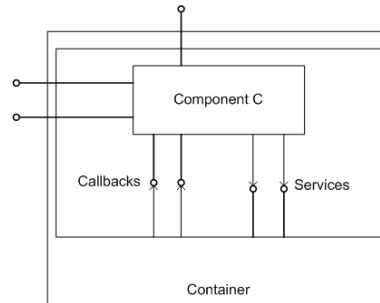


FIG. 1.4 – Le container CCM.

Ce composant interagit avec un POA (Portable Object Adapter) pour les transactions, la sécurité, la persistance et la notification des services via les interfaces du container.

Le modèle de déploiement facilite le développement du composant dans une application. Il est constitué de l'implantation du composant mais aussi de quatre descripteurs écrit en XML(eXtensible Markup Language) :

- le Software Package Descriptor fournit diverses informations sur le logiciel tel que : le nom, la version, les bibliothèques utilisées.
- le CORBA Component Descriptor désigne la catégorie du composant (session, service, entity, process component), le modèle de thread, la gestion des transactions, la qualité de service sur les gestions d'événements.
- le Component File Property Descriptor décrit les propriétés particulières du composant.
- le Component Assembly Descriptor décrit comment les composants vont s'assembler pour former une application composite. Les interactions entre composants y sont détaillées.

1.4.2 J2EE : Java 2 Enterprise Edition

J2EE [39] est une norme qui spécifie une infrastructure particulière de gestion des applications réparties ainsi qu'un ensemble de services accessibles via des APIs (Application Programming Interface) pour concevoir ces applications. Cette norme fait partie de la nouvelle plate-forme JAVA2 introduite en 1998 par Sun qui contient :

J2ME (Java2 Micro Edition) qui cible les terminaux portables.

J2SE (Java2 Standard Edition) qui cible le client. Le J2SE est un sous-ensemble de J2EE.

J2EE (Java2 Enterprise Edition) qui définit le cadre d'un serveur d'applications et d'intégrations.

La figure 1.5 présente l'architecture de la plate-forme J2EE, et dont voici une rapide description des principales APIs offertes :

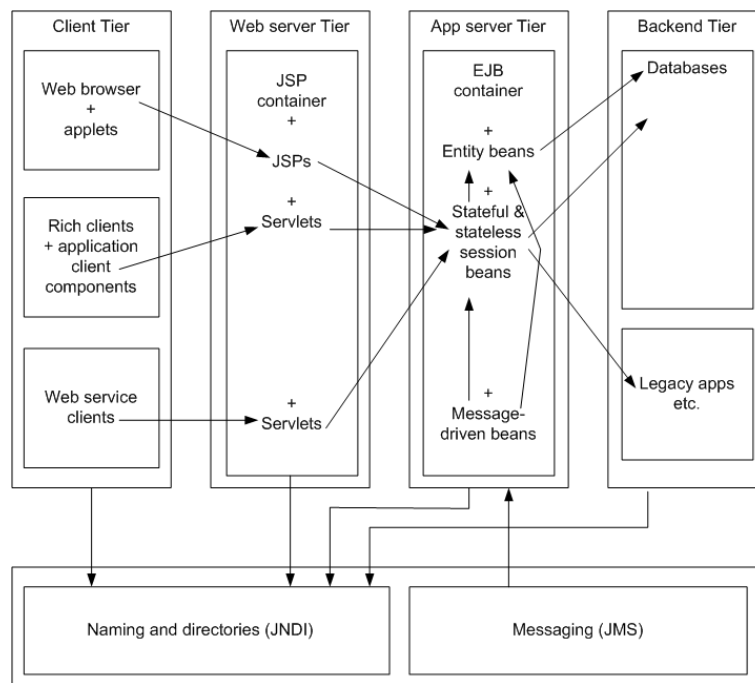


FIG. 1.5 – La plate-forme J2EE.

- *JDBC (Java DataBase Connectivity)* : cette API permet de travailler avec des bases de données relationnelles. Elle permet l'envoi des requêtes SQL (Structured Query Language) à une base, la récupération et l'exploitation des résultats ainsi que l'obtention d'informations concernant la base elle-même et de tables.
- *RMI (Remote Method Invocation)* : elle permet l'invocation des méthodes d'un objet distant de la même manière que pour l'appel d'une méthode d'un objet local. Pour que cette API fonctionne, il faut que le serveur (dans lequel résident les objets distribués) et les clients soient écrits en Java.
- *Java IDL* : si l'utilisation du RMI n'est pas possible (client ou serveur écrits dans un autre langage que java), la plate-forme Java2 inclut un ORB (Object Re-

quest Broker) qui permet à un programme Java de communiquer avec des objets CORBA. Nous avons vu dans le paragraphe 1.4.1 (page 28) que l'interface d'un objet CORBA est décrite dans un langage indépendant de la plate-forme et du langage d'implantation appelé IDL. Sun fournit un compilateur IDL qui permet de générer les classes nécessaires à un objet Java pour communiquer avec un ORB.

- *JNDI (Java Naming and Directory Interface)* : cette API permet de communiquer avec les services de nommage et d'annuaire en réseau. Les objets Java sont accessibles via des chemins ou des valeurs d'attributs.
- *EJB (Enterprise Java Beans)* : Les EJB (Enterprise Java Beans) sont des composants Java portables, réutilisables et déployables [38].
- *Servlets* : les servlets sont des objets qui tournent sur un serveur pour répondre aux requêtes du client de manière dynamique.
- *JSP (Java Server Pages)* : ce sont des pages HTML comportant du code imbriqué. Les JSPs, tout comme les servlets, répondent aux requêtes du client de manière dynamique.
- *JMS (Java Message Service)* : cette API permet l'échange asynchrone de messages ou d'événements critiques entre applications.
- *JTA (Java Transaction API)* : La JTA gère les transactions distribuées via un service de gestion des transactions distribuées avec lequel elle communique au travers de l'API XA (standard défini par l'Open Group)

1.4.3 La plate-forme .NET

En 1998, lors du lancement par Sun de Java2 (§1.4.2 page 30), Microsoft fait le point sur ses solutions d'architectures [65]. De ce bilan, on constate, tout d'abord, que Microsoft est à l'origine de certains concepts clé des serveurs d'application qui ont été repris par Sun lors de la publication des spécifications originales de J2EE (§1.4.2 30). On y trouve par exemple, la notion de transactions automatiques gérées par un *container*. Mais ses solutions d'architectures n'étaient pas sans défaut :

- Visual Basic restait limité et n'était pas réellement objet et C++ était, quant à lui, trop complexe techniquement pour le contexte de l'informatique de gestion.
- L'architecture 3 tiers à base de composants (architecture DNA : Digital Network Architecture) reposait très fortement sur la technologie COM qui entraînait d'importants problèmes de maintenance et de déploiement des composants.
- Il n'y avait pas de standard, et l'ensemble restait une solution propriétaire.

Face à la concurrence de Java2 Enterprise Edition et aux difficultés rencontrées par les composants COM, Microsoft met en place une nouvelle plate-forme : la plate-forme

.NET [20] Son but est de développer simplement des applications Web inter opérables, reposant sur une architecture nouvelle.

Les applications dans .NET ne s'exécutent plus en code machine natif : elles abandonnent le code Intel x86 au profit d'un langage intermédiaire, le MSIL (Microsoft Intermediate Language), s'exécutant sur une machine virtuelle, la CLR (Common Language Runtime). Ce nouvel environnement d'exécution prend en charge les tâches de gestion de ressources et fournit l'abstraction nécessaire entre l'application et le système d'exploitation sous-jacent. Tous les langages de .NET sont compilés sous la forme d'un même code intermédiaire (MSIL) : on va pouvoir dériver dans un langage une classe écrite dans un autre langage, ou encore instancier dans un langage un objet d'une classe écrite dans un autre langage.

Le framework .NET est constitué des éléments suivant (voir fig 1.6) :

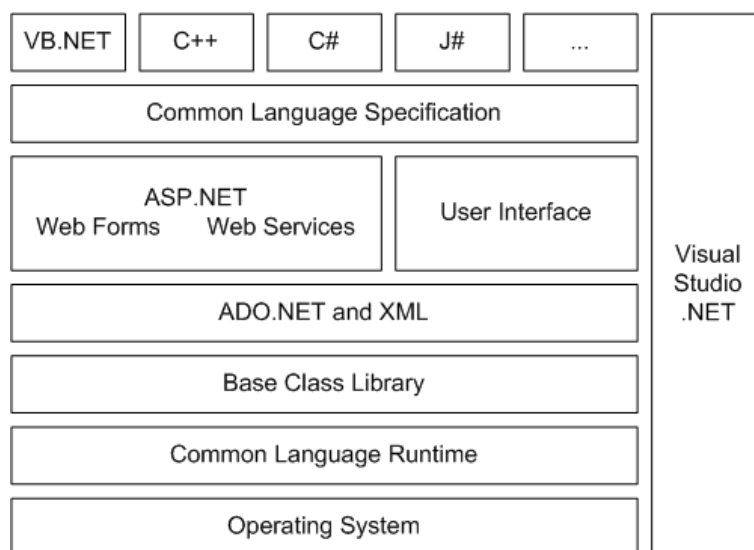


FIG. 1.6 – Le framework .NET.

le CLR (Common Language Runtime) : est l'environnement commun d'exécution qui exécute un *bytecode* écrit en MSIL.

la bibliothèque de composants d'objets de base : le framework met à disposition via les assemblages un ensemble très complet de classes, services et types pour accéder au système d'exploitation (accès aux données, aux fichiers, création de fenêtres...) tout en permettant une gestion précise des différentes versions grâce aux fichiers de configuration et aux assemblages partagés.

ASP.NET (Active Server Pages .NET) : qui est la nouvelle version d'ASP [19] et qui supporte la compilation en MSIL (ASP était interprété).

ADO.NET (Active Data Objects) : qui est la nouvelle génération de composants d'accès aux bases de données ADO, et qui utilise XML et SOAP (Simple Object Access Protocol) pour l'échange des données.

le CLS (Common Language Specification) : permet de créer un compilateur .NET pour n'importe quel langage, à condition de respecter ses spécifications.

le CTS (Common Type System) : ce sont les types gérés par le framework : types de valeurs (aussi appelés types exacts) et types de références (objets, interfaces et pointeurs). Indirectement, le framework s'appuie de façon transparente sur les services Windows et COM+ (architecture DNA), avec lequel il est possible d'interopérer. Le CLS et CTS assurent l'interopérabilité des langages et la bonne exécution dans le CLR.

Pour qu'un langage soit éligible au rang de langage supporté par la plate-forme .NET, il faut qu'il fournisse un ensemble de possibilités, de constructions qui sont recensées dans le CLS. Pour ajouter un langage à .NET, il suffit donc a priori qu'il satisfasse aux exigences du CLS, et qu'un compilateur soit développé depuis ce langage vers MSIL.

.Net fournit des mécanismes d'interopérabilité avec les objets au standard COM, ce qui permet la réutilisation de l'existant utilisant cette technologie.

Les composants .NET s'auto-décrivent grâce à ce qui est appelé un manifeste d'assemblage (méta-données) contenant une description de l'assemblage, des types et des attributs personnalisés. Ce sont ces méta-données qui permettent à des classes écrites dans des langages différents de cohabiter sans problèmes. Ils peuvent coexister en plusieurs versions. De plus, ils bénéficient du ramasse miette (Garbage Collector ou GC) : le comptage de références ou la libération explicite ne sont pas nécessaires.

1.4.4 Quelle architecture choisir ?

On trouve dans la littérature de nombreux comparatifs sur ces différentes plates-formes, parmi ceux-ci [64] et [41]. Le choix de l'architecture va dépendre de plusieurs paramètres comme par exemple les besoins du clients ou les normes disponibles. Il n'y a pas de normes meilleures que d'autres mais plutôt de meilleurs choix en fonction des différents concepts pris en compte. Nous allons détailler certains de ces concepts : la performance, la mise en œuvre et le temps de développement, le déploiement et le contexte d'exécution, les services requis, et nous terminerons par les coûts occasionnés.

- Pour ce qui est de la *performance*, le serveur d'application de Microsoft est le plus rapide à machine équivalente. Ce serveur permet un réglage fin sur les composants. Par contre, pour l'exploiter au mieux, il faut une solide connaissance de l'environnement COM+.
- Le *temps de développement* avec CCM est le plus onéreux car il n'existe pas d'implantation professionnelle ce qui nécessite des réglages et des astuces de programmation sur le code. La *mise en œuvre*, quant à elle, est assez simple lorsqu'il s'agit de J2EE ou COM+ avec des langages de programmation de haut niveau tels que *Visual Basic* ou *Java*.
- La *plate-forme d'exécution* va tenir un rôle important dans le choix de l'environnement. Si la plate-forme est à dominante Microsoft, il est préférable de choisir l'environnement .NET : les produits Microsoft sont parfaitement intégrés et interagissent entre eux. Mais si les compétences sont J2EE ou CORBA, il est tout de même possible de développer dans ces environnements.
Si, nous sommes sur une plate-forme Unix ou Linux, ou si plusieurs serveurs de type différents sont utilisés, on choisira un environnement J2EE ou CORBA.
- Le choix va aussi être guidé par les *services* nécessaires à la mise en œuvre de l'application. Une fois, tous les services déterminés, il faudra pondérer ces services en fonction de leur utilité. Pour ceux à qui la notion de déploiement est essentielle, les EJBs seront à privilégier. Pour le transactionnel, le choix se portera sur CCM, etc.
- Le choix de la plate-forme peut aussi être déterminé par rapport aux différents *coûts* que cela implique.
Microsoft propose ses produits gratuitement dès l'achat de son système d'exploitation contrairement à ses concurrents : le coût d'un serveur EJB ou CORBA varient selon les services et le middleware choisis. Ils existent des implantations EJBs et CORBA en Open Source mais elles sont peu utilisées.
En coût de maintenance, les applications .NET et CORBA sont plus onéreuses que celles de Java.

Fondamentalement, les trois plate-formes présentées dans le paragraphe 1.4 (page 28) sont des candidates potentielles pour le projet européen QCCS. Cependant, la plate-forme .Net répond au mieux aux besoins du projet. Tout d'abord, tous les partenaires du projet avaient déjà utilisé les produits Microsoft en général et .NET en particulier. De plus, le framework .NET a un support pour les méta-données (information décrivant les données, ce qui nous est utile pour stocker les contrats associés aux composants) [56] et il n'exige pas une solution basée sur les conteneurs.

En effet chaque composant peut avoir sa propre manière de manipuler des invocations à distance. Il est important que les concepteurs d'application puissent se servir des propriétés de QdS du composant pendant la phase de conception. En utilisant les méta-données il est possible de coder les propriétés de QdS dans le composant. Un outil

de conception pourra extraire cette information plus tard.

C'est donc la plate-forme .NET avec l'environnement de développement Visual Studio .NET qui sont utilisés dans le projet QCCS.

1.5 Conclusion

Dans ce premier chapitre, nous avons présenté les besoins face aux limites du paradigme objet : le paradigme à composant répond à ces besoins. Nous avons vu qu'un composant est :

une unité de composition ayant des interfaces contractualisées et des dépendances contextuelles. Un composant peut être déployé indépendamment et être sujet à la composition [64].

Puis nous avons étudié les deux cycles de vie dans le développement par composants :

- le cycle de vie du système à composants,
- le cycle de vie du composant.

Nous avons présenté trois plates-formes de recherche permettant de modéliser des systèmes de composants hétérogènes :

- MetaH,
- PtolemyII,
- Metropolis.

Et enfin, nous avons évoqué les trois principales approches sur le marché à savoir :

- CORBA
- J2EE
- .NET

Nous avons vu que la plate-forme choisie pour ce projet est la plate-forme .NET. Dans le chapitre suivant, nous présentons la problématique de l'association de contrats de qualité de service à des composants afin de garantir la qualité des services proposés. Nous rappellerons tout d'abord la notion de contrat. Ensuite, nous reviendrons sur ce qu'est la qualité de service. Nous passerons en revue les travaux existants et en particulier QML. Puis nous terminerons ce chapitre par une présentation de la problématique abordée dans cette thèse.

Chapitre 2

Les contrats de qualité de service

Contents

2.1	La notion de contrat	38
2.1.1	Définitions	38
2.1.2	Les niveaux de contrats	38
2.2	La qualité de service	40
2.2.1	Définitions	40
2.2.2	Les travaux existants	42
2.2.2.1	Les méthodes formelles	43
2.2.2.2	SMIL : Synchronized Multimedia Integration Language	43
2.2.2.3	TINA ODL	43
2.2.2.4	MAQS : Management for Adaptative QoS-enabled Services	43
2.2.2.5	QuO : Quality of service for Objects	44
2.2.2.6	QML : Quality of service Modeling Language	44
2.2.2.7	QDL : Quality of service Definition Language	44
2.2.3	Evaluation des différents travaux	44
2.3	QML : Quality of service Modeling Language	46
2.3.1	Présentation	46
2.4	Problématique	48
2.4.1	Description des contrats	48
2.4.2	Expressions QML d'un contrat de fiabilité	49
2.4.3	Les différents aspects de spécification	50
2.4.3.1	La délégation de contrats	51
2.4.3.2	La dépendance de contrats	52
2.4.3.3	Le comportement adaptatif des contrats	53
2.4.3.4	La qualité de service de bout en bout	54
2.5	Conclusion	55

2.1 La notion de contrat

Dans le cadre de cette thèse, nous nous intéressons aux contrats de qualité de service associés aux composants. Mais les contrats ne sont pas spécifiques aux composants et ces concepts sont généralisables [55].

2.1.1 Définitions

Avant d'introduire la notion de contrat, nous allons tout d'abord définir les notions importantes dans les contrats.

- **Un service** est une opération qui va être appelée par un composant et exécutée par un autre.
- **Un client** est un composant souhaitant utiliser un service.
- **Un fournisseur** est le composant qui va exécuter ce service.

Définition 2 *Un contrat spécifie les droits et les obligations entre un client et un fournisseur de service. Les contrats déterminent les relations entre les différents acteurs.*

On va distinguer deux opérations importantes dans les contrats qui sont deux façons distinctes d'exprimer un comportement complexe en terme de comportements plus simples [29] :

- le raffinement qui permet la spécialisation des obligations contractuelles et des invariants de contrats, et
- l'inclusion qui décompose un contrat en sous-contrats plus simples.

2.1.2 Les niveaux de contrats

Le contrat vise à expliciter et contrôler la collaboration et le comportement entre les objets [29]. Il va définir les invariants du service et l'ensemble des participants communicants. On peut distinguer deux formes d'obligations :

Les obligations contractuelles sont définies par le contrat. Elles permettent d'inclure les contraintes entre objets. Il peut y avoir incompatibilité entre l'objet et certaines variables.

Les obligations causales forcent les participants à effectuer une séquence ordonnée d'actions. Grâce à cette deuxième famille d'obligations, les contrats capturent les dépendances comportementales entre les objets.

Les contrats peuvent être classés en quatre niveaux où chaque niveau contient les propriétés des contrats de niveau inférieur[10].

Le contrat syntaxique comporte les opérations que le composant peut exécuter, les paramètres d'entrées et de sortie qu'il doit avoir et les exceptions qui pourraient survenir lors de l'opération. Il permet de vérifier la conformité entre les interfaces fournies et requises du composant et facilite la vérification statique des composants.

Le contrat comportemental spécifie le comportement des opérations en utilisant les pré- et post-conditions, pour chaque service offert et les classes d'invariants sur une déclaration d'opération [55] :

La pré-condition doit être satisfaite à la demande du service par le client.

La post-condition quant à elle doit être satisfaite par le fournisseur après la réalisation du service.

La violation d'une pré-condition va indiquer une erreur au niveau du client c'est-à-dire que le demandeur n'a pas observé les conditions imposées pour des appels corrects alors que la violation d'une post-condition implique une erreur du côté du fournisseur, le service demandé a échoué. Plus on a de pré-conditions, plus la difficulté pour le client sera grande et plus la tâche sera facile pour le fournisseur [55].

Ces conditions peuvent être décrites en OCL (Object Constraint Language)[44] [3].

- Une pré-condition sera de la forme *pré : prédicat*,
- une post-condition sera de la forme *post : prédicat*.

Le contrat de synchronisation. Le contrat comportemental considère les services comme atomiques. Le contrat de synchronisation, quant à lui, spécifie le comportement global des objets en terme de synchronisation entre les appels de méthodes. Le but de ce contrat est de décrire les dépendances entre les services d'un composant tels que la séquence ou le parallélisme. Ce contrat garantit, quel que soit le client qui demande le service, que ce service sera correctement exécuté.

Il s'exprime sous forme de contraintes à respecter en temps réel et peut offrir de nombreux services, par exemple :

- l'accès à un objet peut être subordonné à l'obtention préalable d'un contrat (sécurité),
- son exclusivité garantit l'exclusion mutuelle (verrou),
- la liaison d'un ensemble d'objets, qu'ils soient locaux ou distants (groupes de communication, transparence à la localisation),
- l'intégration de la notion de contraintes de temps (*deadline*).

Le contrat de qualité de service. Une fois toutes les propriétés comportementales spécifiées, on peut essayer de quantifier le comportement voulu, c'est-à-dire la

qualité de service attendue (voir partie II §2.2 page 40) en énumérant les caractéristiques que doit respecter le serveur comme par exemple :

- le délai de réponse maximal,
- le temps de réponse moyen,
- la qualité du résultat, que l'on peut exprimer sous forme de précision.

2.2 La qualité de service

Les systèmes sont élaborés à partir de besoins fonctionnels c'est-à-dire en fonction de ce que l'on a besoin qu'il fasse. Prenons l'exemple d'une machine à café, sa spécification fonctionnelle est de faire du café. Cependant, si la machine à café met trop de temps à faire le café, le client ne sera pas satisfait et la prochaine fois, il utilisera une autre machine qui prendra moins de temps. Cette spécification de temps sur le comportement fonctionnel de la machine est une spécification non fonctionnelle ou encore appelée qualité de service (QdS).

2.2.1 Définitions

Dans [68], la QdS dans les systèmes multimédias est définie comme suit :

La QdS représente l'ensemble des caractéristiques quantitatives et qualitatives des systèmes multimédia distribués qui sont nécessaires pour réaliser les fonctionnalités exigées par l'application.

La QdS, dans les réseaux, est vue comme [37] :

Le moyen d'allouer des ressources dans les commutateurs et les routeurs afin que les données arrivent à destination rapidement, de manière uniforme et fiable.

De manière plus générale, la QdS est l'effet combiné des performances d'un service qui déterminent le degré de satisfaction d'un utilisateur de ce service [2].

La QdS d'une application est définie par l'ensemble de ses propriétés non fonctionnelles. Elle peut varier tout au long de l'exécution de l'application, et évoluer avec les besoins des applications et des utilisateurs.

Reprenons l'exemple du GPS vu dans le §1.1.5 de la page 20. Nous avons déterminé deux propriétés non fonctionnelles sur notre GPS :

- la première est une propriété sur le temps de réponse qui va être représentée par une valeur numérique ;
- la deuxième est proposée sous la forme d'un mode de fonctionnement. Trois types sont proposés :

1. *Power Save* : ce mode correspond à un niveau bas d'énergie du GPS. Pour économiser l'énergie, le GPS va *écouter* seulement trois satellites (le minimum requis) avec la plus longue période de rafraîchissement des données possibles. Si la réception des données est mauvaise sur un des satellites ou si la liaison satellite est interrompue, le GPS va alors se *connecter* à un autre satellite.
2. *Best Track* : le GPS doit maintenir une liaison avec cinq satellites. Si l'un d'entre eux est mal reçu, un processus de recherche de satellites pouvant émettre correctement les données est appliqué. On choisira alors le meilleur satellite (celui dont le taux d'erreur sera le plus faible).
3. *Best Effort* : le GPS écoute le maximum de satellites (avec un minimum de cinq satellites) afin de calculer la position avec la meilleure précision possible quelque soit le coût en terme d'énergie.

La QdS du GPS varie tout au long de l'exécution, cette variation est due :

- au niveau de la batterie : si le GPS n'a plus assez d'énergie, il y aura reconfiguration du contrat de QdS qui basculera en mode *Power Save*.
- à l'utilisateur qui décide avec quelle précision et quels moyens techniques (nombre de satellites, dépenses d'énergie...) il veut obtenir sa position.
- à la liaison avec les différents satellites. Si le GPS est en mode *Best Track* ou *Best Effort* et qu'il n'y a plus que 4 liaisons satellites satisfaisantes, le mode passera en *power save*.

La QdS peut s'exprimer sous forme de caractéristiques [5]. Une caractéristique de QdS représente un aspect de la QdS du système qui peut être identifié et quantifié. Il ne faut pas confondre une caractéristique de QdS et un paramètre de QdS. Un paramètre de QdS est une valeur liée à la QdS qui est véhiculée entre les différentes entités. Les caractéristiques de QdS peuvent être groupées en catégories de QdS qui représentent les besoins extra-fonctionnels de l'application. Selon J. Aagedal, il y a cinq états possibles pour les caractéristiques de la QdS :

1. Les besoins de QdS qui expriment les contraintes que les utilisateurs du système (ou de ses composants) ont sur le système (ou sur les composants).
2. Les capacités de QdS qui expriment les possibilités actuelles en terme de QdS fournies par les composants dans le système.
3. Les offres de QdS qui décrivent la QdS proposée.

4. Les contrats de QdS qui sont le résultat des négociations.
5. Les observations de QdS qui expriment les valeurs des différentes caractéristiques de QdS qui ont été évaluées.

Selon les différents points de vue de l'ODP (Open Distributed Processing) décrits dans [48], le niveau de QdS sera différent :

- du point de vue de l'entreprise, la QdS est subjective et non formelle : elle est orientée vers la spécification des utilisateurs par exemple : «le son doit être clair et net quelles que soient les conditions.» ;
- du point de vue de l'information, la QdS est objective. Les besoins de QdS sont dérivés de la spécification subjective de la qualité de service. Cette spécification est souvent indépendante de l'application, par exemple, une connection réseau à 100 Mb/s ;
- du point de vue calculatoire , les objets sont identifiés et les caractéristiques de QdS de l'application sont souvent décrites en terme de qualité des média et de relations entre ceux-ci, par exemple le rafraîchissement des images peut être exprimé en images/s ;
- du point de vue ingénierie, la QdS peut être basée soit sur le système, soit sur le réseau :
 - sur le système, c'est la description des besoins de QdS du système d'opérations, par exemple la taille du buffer ou la mémoire en Mb ;
 - sur le réseau, c'est la description des besoins de QdS du réseau, par exemple le débit en Mbit/s ;
- du point de vue technologique, les mécanismes de QdS utilisés pour implanter le système sont spécifiés, par exemple le format vidéo pour l'application sera le format PAL.

2.2.2 Les travaux existants

De nombreux travaux de recherche axés sur la spécification de la QdS ont donné le jour à des langages supportant la QdS. Certains sont spécifiques à une catégorie de QdS comme la fiabilité ou la QdS relative au temps. D'autres sont plus génériques et couvrent tous les domaines de QdS. Dans cette section, nous donnons un aperçu des principaux travaux relatifs à la QdS.

2.2.2.1 Les méthodes formelles

Les méthodes formelles permettent de spécifier le comportement du système. On peut donc naturellement se demander si ces méthodes peuvent être utilisées pour spécifier la QdS. De nombreuses approches se focalisent sur les aspects temps réel comme QTL (Quality of service Temporal Logic) [11] qui permet de spécifier d'une part les besoins temporels et d'autre part les performances supposées du système. TLA (Temporal Logic of Actions) [47] a beaucoup influencé la spécification de la QdS. C'est une logique temporelle basée sur les états dans laquelle la garantie des spécifications peut être faite pour modéliser comment le système agit si l'environnement fait ce qu'il est supposé faire.

2.2.2.2 SMIL : Synchronized Multimedia Integration Language

SMIL [57] permet la description du comportement temporel d'une présentation en fournissant une ligne temporelle pour coordonner l'affichage des objets multimédias. Ces objets peuvent être présentés soit séquentiellement soit en parallèle et leur synchronisation peut être spécifiée soit en temps absolu soit grâce à des événements. L'adaptation est supportée par l'élément *switch* qui évalue les propriétés du système hôte et affiche les objets multimédias en conséquence. SMIL n'est pas un langage de spécification de QdS générique puisqu'il se restreint à la spécification des présentations multimédia.

2.2.2.3 TINA ODL

TINA ODL [18] est un surensemble de CORBA IDL (voir partie II §1.4.1 page 28) qui permet la spécification des objets via leurs interfaces. TINA ODL supporte la spécification de la QdS. Elle est spécifiée en utilisant une paire *nom-valeur* directement liée à une opération ou à un flot de données. Il n'est donc pas possible d'associer différentes spécifications de QdS avec la même interface : différentes implantations de la même interface avec différentes QdS n'est pas permis. Cependant, il suffit d'hériter de cette interface et d'ajouter à chaque nouvelle interface héritée des spécifications de QdS différents.

2.2.2.4 MAQS : Management for Adaptative QoS-enabled Services

MAQS [8] inclut QIDL une extension d'IDL (voir partie II §1.4.1 page 28) qui supporte la spécification de la QdS en fournissant la possibilité de spécifier des interfaces de QdS et en leurs assignant des interfaces fonctionnelles. Dans les interfaces de QdS, les différentes caractéristiques de QdS sont définies en utilisant des types IDL et des opérations relatives à la QdS vont être spécifiées afin d'être fournies aux clients. QIDL est une extension de CORBA IDL, ce qui disqualifie tout autre langage de définitions d'interfaces. Enfin MAQS supporte l'adaptation pour la renégociation.

2.2.2.5 QuO : Quality of service for Objects

QuO [50] est une architecture qui supporte la QdS d'un objet CORBA et établit un rapprochement conceptuel entre les garanties du réseau et les besoins de l'application. QuO se focalise sur les connections entre les objets distribués. Ces connections ont un comportement qui adapte les applications aux conditions considérées. Chaque propriété du système est une dimension de QdS et QuO divise cet espace multidimensionnel en régions définies par des prédicats provenant des propriétés du système. Deux niveaux de régions sont définis :

- le niveau de négociation qui définit les conditions du système avec lesquelles le client et le serveur essaient de travailler ;
- le niveau *réel* : dans les régions de négociation, il peut y avoir plusieurs régions *réelles* qui sont les conditions mesurées du systèmes.

QuO se focalise sur les connections entre clients et serveurs et supporte seulement les dimensions numériques et mesurables de QdS. Cette contrainte restreint les types de QdS qui peuvent être définis. La spécification des caractéristiques subjectives telles que la sémantique des pannes (*halt, initial state, rolled back*), la productivité ou la satisfaction ne peuvent pas être décrite naturellement.

2.2.2.6 QML : Quality of service Modeling Language

QML[27] a trois principaux mécanismes d'abstraction : *contract type, contract* et *profile*. C'est un langage de spécification de QdS qui sépare la spécification de la QdS de la spécification des aspects fonctionnels (en IDL). Bien que les profils définissent les QdS offertes par le service fournisseur, il n'est pas possible de spécifier ce que le service pourra réellement fournir en regard des conditions environnementales.

2.2.2.7 QDL : Quality of service Definition Language

QDL [59] définit les relations de QdS en définissant des objets de QdS. Un objet de QdS contient une attente et une obligation de QdS (QdS offerte). Une obligation de QdS contient un nombre de propriétés classées soit simples soit complexes. Une propriété simple est une paire *nom-valeur* alors qu'une propriété complexe dépend d'autres propriétés provenant d'autres objets de QdS. Un besoin de QdS est spécifié comme une contrainte sur les propriétés des autres objets de QdS. QDL utilise OCL pour spécifier les relations de QdS.

2.2.3 Evaluation des différents travaux

Tous ces langages ont été étudiés dans [5]. Dans ce document, Jan Oyvind Aagedal évalue ces différents langages selon 25 critères tels que :

- La généralité : Est-il générique ou spécifique à une certaine catégorie de QdS ?
- Le support au niveau cycle de vie : Supporte-il la spécification de tous les aspects de QdS tout au long du processus de développement du logiciel ?

- L'indépendance au niveau de la plate-forme.
- L'objet : Est-il un langage à objets ?
- La séparation : Sépare-t-il les propriétés fonctionnelles des propriétés extra-fonctionnelles ?
- La négociation : Y-a-t-il possibilité de négociation entre le client et le serveur ?
- La composition : Y-a-t-il possibilité de composition entre propriétés extra-fonctionnelles ?

Voici un aperçu des résultats de cette évaluation :

- De manière générale, tous ces langages ont une approche à objets, avec un bémol pour les méthodes formelles : un sous-ensemble de ces méthodes ont cette approche mais pas toutes.
- Seul SMIL se restreint à une seule catégorie de QdS, ne supporte pas la spécification des aspects de QdS tout au long du processus de développement, n'est pas indépendant de la plate-forme et n'est pas conforme à l'ODP.
- De toutes ces approches, il n'y a que TINA ODL qui n'est pas typé.
- Seules les méthodes formelles donnent une définition précise de la QdS spécifiée ce qui évite les incompréhensions entre développeurs comme les caractéristiques subjectives de la QdS.
- QuO et QML séparent la spécification de la QdS des propriétés fonctionnelles du composant. Ceci permet à une interface d'être implémenté par plusieurs composants avec différentes QdS.
- QML supporte l'intégration des spécifications décrites en IDL et permet la négociation entre client et serveur.

Jan Oyvind Agedal conclut de son évaluation sur les langages supportant la QdS que QML est le plus complet et le plus satisfaisant : il est générique, supporte tous les aspects de QdS durant le processus complet du logiciel, ainsi que l'intégration des spécifications décrites en IDL. Il est indépendant d'une plate-forme d'exécution, conforme au RM-ODP (Reference Model of Open Distributed Processing), à objet. De plus, il sépare la spécification de la QdS des propriétés fonctionnelles et permet la négociation entre le client et le serveur pour établir des contrats de QdS ainsi que les raffinements de spécifications. QML est typé ce qui facilite la vérification de conformité des contrats négociés. Enfin, une extension de UML existe afin d'intégrer les notions de QML dans UML [26]. Nous présentons plus en détails QML dans la section suivante (voir partie II § 2.3 page 46).

2.3 QML : Quality of service Modeling Language

2.3.1 Présentation

QML [27] a été créé par Svend Frolund et Jari Koistinen. Ce langage permet de décrire les propriétés extra-fonctionnelles de QdS des composants logiciels et de les spécifier à différents niveaux : interfaces, attributs, opérations, résultats et paramètres d'opérations. On distingue quatre concepts fondamentaux dans QML :

Contract type : Un contrat type représente la catégorie de la QdS comme par exemple la performance. Il décrit toutes les dimensions possibles de cette catégorie qui vont être utilisées pour caractériser un aspect particulier de la qualité de service. Par exemple, si l'on définit un contrat type Performance, on va pouvoir décrire comme dimension *latency* ou *throughput*. Un contrat type représente «une famille de contrat». C'est juste un modèle que l'on va en quelque sorte instancier afin de créer nos contrats.

Dimension : Une notion très importante de QML est la notion de dimension. Une dimension caractérise un aspect particulier de la QdS pour une catégorie donnée. Dans le contrat type, une dimension est définie par son nom et le domaine. Le domaine peut être ordonné et avoir une unité. Il y a trois types de domaines différents : ensemble, énumération ou nombre. L'ensemble et l'énumération sont des domaines définis par l'utilisateur.

Contract : C'est une instance de contract Type qui représente une spécification particulière de la QdS. Les contrats capturent les spécifications de QdS pour les éléments d'interfaces tels que les opérations ou les attributs. Un élément d'interface peut être associé à plusieurs contrats à la fois : un contrat pour chacune des catégories qui nous intéressent. Par exemple, la même opération peut avoir un contrat qui spécifie la performance et un autre spécifiant les propriétés de sécurité.

Profile : Il décrit les propriétés de QdS d'un service. Il est défini pour une interface particulière et spécifie les contrats de QdS pour les attributs et les opérations de cette interface. Un *profile* décrit les associations entre les contrats et les éléments d'interfaces pour une interface particulière. Un profile peut spécifier un contrat par défaut qui sera appliqué à tous les éléments de l'interface. En plus, on va pouvoir associer des contrats particuliers à certains éléments d'interfaces. Par exemple, on va associer un contrat de performance par défaut à toutes les opérations d'une interface et ajouter des contrats avec des contraintes plus fortes pour les opérations qui nécessitent une meilleure performance que celle imposée par le contrat par défaut.

Parmi les contrats types généralement employés, on peut citer :

1. reliability : TTR (Time to repair), server failure, failureMasking ...
2. availability
3. performance : delay, throughput, latency ...
4. security
5. timing

Dans le paragraphe §2.2.1 page 40, nous avons déterminé deux contrats de QdS pour notre GPS. Nous allons maintenant spécifier le premier contrat de QdS en QML. Ce contrat représente une contrainte sur le temps de réponse maximum du GPS. Nous voulons spécifier que ce temps de réponse, associé à l'opération *getPosition()* qui renvoie la position courante, sera inférieur à un certain délai. Tout d'abord, nous allons créer un *contractType* que nous appelons *TimeOut* qui représente la catégorie de QdS que nous voulons exprimer. Ce *contractType* a une dimension appelée *delay* qui est une valeur numérique (numeric) exprimée en millisecondes (msec). Le mot-clé *decreasing* signifie que plus la valeur sera petite mieux c'est. A l'inverse, le mot-clé *increasing* implique que plus la valeur est grande mieux c'est. Ces mot-clés sont utiles lors de la phase de négociation ou de la vérification de conformité entre deux contrats.

Voici la spécification QML de notre *contractType TimeOut* :

```
type TimeOut= contract {
    delay : decreasing numeric msec ;
};
```

Le *contractType* spécifié, nous pouvons maintenant créer ses instances, c'est-à-dire des contrats qui le réalise et leurs affecter des valeurs pour les différentes dimensions déclarées.

Pour le *contractType TimeOut*, nous avons créé une instance nommée *TimeOutC* dont la valeur numérique *delay* doit être inférieure à 3 msec :

```
TimeOutC= TimeOut Contract {
    delay < 3 msec
};
```

L'affectation des valeurs peut s'exprimer de deux façons :

- par une valeur simple précédée d'un opérateur (<, >, ==, <=, >=) comme dans l'exemple ci-dessus ;
- par une fonction prédéterminée par QML : *percentile*, *mean*, *variance*, *frequency* comme le montre l'exemple suivant ;

Le contrat *TimeOutCbis* est une instance du *contractType TimeOut* pour laquelle la contrainte sur la dimension *delay* s'exprime sous forme d'une moyenne et d'un pourcentage. La moyenne du temps de réponse doit être inférieure ou égale à 5 msec et 100% des valeurs doivent être inférieur à 10 msec :

```
TimeOutCbis= TimeOut Contract {
    delay {
        mean <= 5 msec;
        percentile 100 < 10 msec
    }
};
```

Une fois les contrats instanciés, il reste à les associer aux interfaces choisies. Pour le GPS, nous voulons spécifier un temps de réponse sur l'opération *GetPosition()*. Cette opération est offerte par l'interface *ComputerI* qui fournit aussi les services *SwicthOn()* et *SwitchOff()*. Ce qui nous donne en QML :

```
TimeOutProfile for ComputerI= Profile {
    from GetPosition require TimeOutC;
};
```

Il est possible de spécifier le contrat pour toute l'interface ou seulement une opération de cette interface (comme dans l'exemple ci-dessus). Supposons maintenant que nous voulions attribuer à toute l'interface *ComputerI* notre contrat *TimeOutC* et que sur l'opération *GetPosition*, nous voulions un délai moyen de temps de réponse inférieur à 2 msec. Dans ce cas, la spécification en QML de ce *profile* s'écrit sous la forme :

```
TimeOutProfileBis for ComputerI= Profile {
    require TimeOutC;
    from GetPosition require TimeOut contract {
        delay mean < 2msec
    }
};
```

2.4 Problématique

2.4.1 Description des contrats

L'expression de la QdS dans un composant est facilement exprimable lorsqu'il s'agit d'expression simple comme nous l'avons vu avec le contrat *TimeOut* du GPS (voir partie

II §2.3 page 46). Outre cet exemple, on trouve de nombreuses descriptions de contrat de QoS pour la fiabilité, la disponibilité, la performance, la sécurité ou encore sur les aspects temporels. Dans [45], Jari Koistinen présente la spécification d'un contrat de type fiabilité. Il énumère un certain nombre de dimensions liées à ce contrat :

- *MTTR* : *Mean Time To Repair* donne le temps moyen nécessaire pour la réparation d'un service après son échec.
On peut apporter des variantes à cette dimension comme $V(MTTR)$ qui est la variance de MTTR ou encore $Max(MTTR)$ qui représente le temps maximum autorisé pour la réparation d'un service.
- *MTTF* : *Mean Time To Failure* représente le temps moyen entre deux échecs. Tout comme MTTR, cette dimension est modifiable selon les besoins.
- *Continuous Availability/T* est la probabilité (comprise entre 0 et 1) que le service soit disponible sur l'intervalle de temps T.
- *Availability/t* est la probabilité (comprise entre 0 et 1) que le service sera disponible à l'instant t.
- *Failure Masking* détermine les types d'échecs masqués par le service. Les échecs non masqués doivent être gérés par le client.
- *Server Failure* détermine l'état du serveur, une fois rétabli, après l'échec du service.
- *Rebinding Policy* détermine si la référence faite à un service avant son échec est encore valide après le rétablissement du service.
- *Expected Number of Service Failures* représente le nombre prévu N d'échecs sur une certaine période T.

2.4.2 Expressions QML d'un contrat de fiabilité

Toutes ces dimensions peuvent s'exprimer en QML dans un *ContractType* :

```

type QMLReliability = contract {
    MTTR : decreasing numeric sec ;
    MTTF : increasing numeric day ;
    availability : increasing numeric ;
    contAvailability : increasing numeric ;
    failureMasking : decreasing set {omission, lostResponse, noExecution, response, responseValue, stateTransition} ;
    serverFailure : enum {halt, initialState, rolledBack} ;
    rebindingPolicy : decreasing enum {rebind, noRebind} with order {noRebind < rebind} ;
};

```

Une instance de ce *ContractType* s'exprime alors aisément :

```

ServiceReliability = QMLReliability contract {

```

```

    MTTR < 20 sec;
    MTTF
    {
        percentile 100 > 0.05 days;
        percentile 80 > 20 days;
        mean > 24 days;
    }
    availability >= 0.99999;
    contAvailability > 0.999999;
    failureMasking == { omission };
    serverFailure == initialState;
    rebindingPolicy == rebind;
};

```

2.4.3 Les différents aspects de spécification

La spécification des contrats de QoS dans les composants logiciels n'est possible avec QML que jusqu'à un certain niveau de description. Reprenons l'exemple du GPS spécifié dans la partie II §2.2.1 page 40, nous avons déterminé deux contrats de QoS pour notre application. La spécification QML du premier contrat qui porte sur les propriétés temporelles est donnée dans la partie II §2.3 page 46.

Avant de donner la description QML du deuxième contrat qui porte sur le mode de fonctionnement du GPS, nous allons décrire son comportement :

La sélection du mode de fonctionnement est laissée à l'utilisateur lors de la mise en route du GPS via la fonction *SwitchOn()*. Ce mode va s'adapter aux différents changements de l'environnement du GPS comme par exemple un niveau de batterie faible ou une mauvaise réception des données. Le contrat est vérifié lors de l'appel de la fonction *GetPosition()*. Si les conditions requises par le contrat ne sont pas valides, il y a renégociation du contrat dans un nouveau mode. Voici la description QML de ce contrat :

```

type Mode = contract {
    modeLevel : increasing enum {PowerSave, BestTrack, BestEffort} with order {PowerSave < BestTrack; BestTrack < BestEffort;}
};

ModeGPS = Mode contract {
    modeLevel == BestEffort;
}

```

La description en QML est succincte. Elle indique qu'il y a 3 modes de fonctionnement : *PowerSave*, *BestTrack* et *BestEffort* et que ces modes sont ordonnés. Le mode de fonctionnement est de type *énumération* ce qui signifie qu'un seul mode n'est valide à la fois .

Mais cette spécification ignore les aspects de dépendances, de comportement ainsi que la délégation de contrats. Nous allons détailler ces différents aspects pour notre exemple du GPS.

La figure 2.1 représente le modèle à base de composants du GPS et les différents contrats associés. On y retrouve les aspects cités ci-dessus. Les contrats sont représentés graphiquement comme un manuscrit dans lequel on va noter le nom du contrat par exemple *TimeOut* ainsi que l'opération sur laquelle va porter ce contrat. Dans le cas du contrat *TimeOut*, c'est l'opération *getPosition()* qui sera contractualisée.

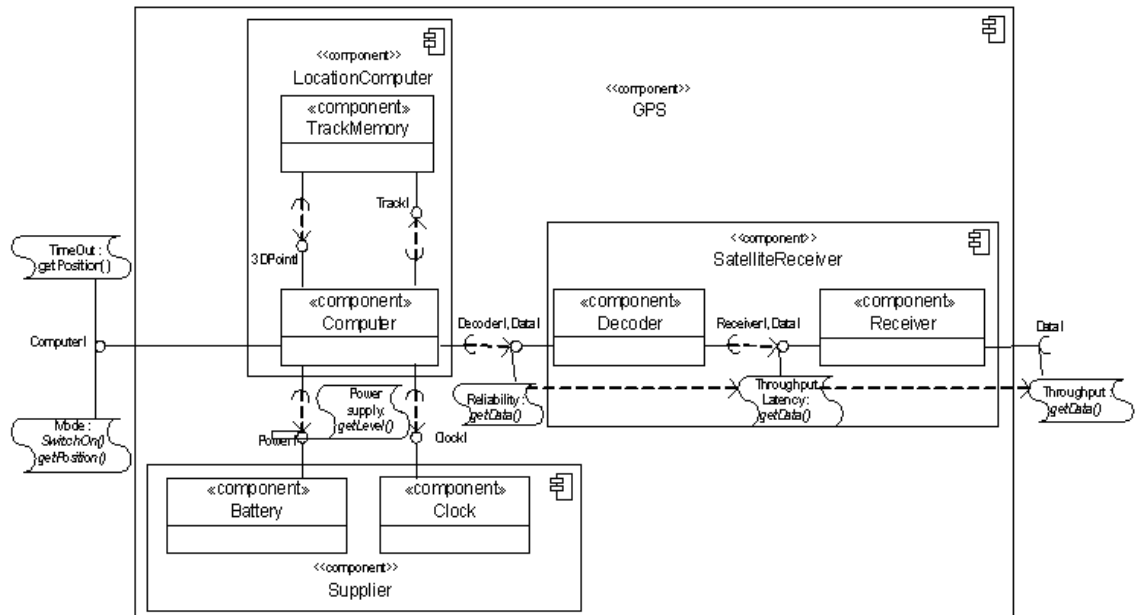


FIG. 2.1 – Représentation interne du GPS.

2.4.3.1 La délégation de contrats

Le GPS est constitué de 4 composants (voir partie II §3.2.1 page 65). Tout comme il est possible de structurer un composant en sous-composants, les contrats de QdS sont décomposables.

Le contrat *ModeGPS* se décompose comme suit :

- Un contrat de type *PowerSupply* qui vérifie que le niveau de batterie est suffisant. Ce contrat s'applique au service *getLevel()* du composant *Manager*. Il se compose d'une dimension nommée *PowerLevel*. La description QML du *Contract*

Type *PowerSupply* est :

```
type PowerSupply = contract {
    powerLevel : increasing enum {none, low, high} with order {none <
    low; low < high; }
};
```

- Des contrats de type *Reliability* qui permettent de vérifier la fiabilité des données émises par chaque satellite. Ces contrats s’appliquent aux services *getData()* des composants *Decoder* : un contrat par *Decoder*. Une description d’un *ContractType Reliability* a été donné dans la partie II §2.4.2 page 49. Pour l’exemple du GPS, nous allons déterminer un *ContractType OurReliability* plus simple dont voici la description QML :

```
type Reliability = contract {
    confidence : increasing enum {none, low, medium, high} with order
    {none < low; low < medium; medium < high; }
};
```

Si la valeur de la dimension *confidence* est *none* ou *low* pour tous les contrats associés aux *Decoder* actifs cela signifie que le GPS est situé dans une zone où la liaison satellite est mauvaise.

QML ne permet pas d’exprimer la délégation de contrats.

2.4.3.2 La dépendance de contrats

Pour calculer les valeurs de ses différentes propriétés extra-fonctionnelles, un contrat peut avoir besoin des valeurs des propriétés extra-fonctionnelles d’autres contrats. Dans ce cas, le contrat est dépendant d’autres contrats.

Plus concrètement dans le cas de notre GPS, le récepteur reçoit les données d’un satellite. Ces données sont envoyées au décodeur associé. Si la fiabilité des données reçues par le récepteur est mauvaise, la fiabilité des données au niveau du décodeur sera elle aussi mauvaise.

La fiabilité des données côté décodeur est donc dépendante de la fiabilité des données émises par le récepteur. La spécification QML du contrat de fiabilité côté récepteur (*Receiver*) sera la même que pour le contrat associé au décodeur. Pourtant le *calcul* de la fiabilité est différent dans les deux cas :

- côté *Decoder*, le niveau de fiabilité est une combinaison de la fiabilité de la liaison entre le *Decoder* et le *Receiver* et de la fiabilité des données émises par le *Receiver* :

$$\text{Reliability}(\text{Decoder}) = f(R, \text{Reliability}(\text{Receiver}))$$

avec R= fiabilité de la liaison entre le *Decoder* et le *Receiver*

- côté *Receiver*, le niveau de fiabilité est calculé périodiquement selon la valeur de la latence et le débit effectif par rapport aux valeurs contractualisées :

$$\text{Reliability}(\text{Receiver}) = f(\text{Latency}, \text{Throughput})$$

QML ne permet la spécification de ces dépendances.

2.4.3.3 Le comportement adaptatif des contrats

Lors de l'exécution d'un service si la QoS requise n'est pas satisfaite, le client peut demander la mise en place d'un autre service. Selon les caractéristiques du GPS, le contrat *ModeGPS* aura un comportement différent. Or l'environnement du GPS est variable : il dépend de l'endroit où le GPS est situé, de la zone de couverture des satellites, mais aussi de l'énergie dont dispose le GPS pour fonctionner ainsi que du choix de mode de fonctionnement fait par l'utilisateur (*PowerSave*, *BestTrack* ou *BestEffort*).

Supposons que l'utilisateur choisisse le mode *BestEffort* : ce mode implique qu'au moins 5 satellites doivent être *écouter*. Si le GPS est dans une zone où la couverture est mauvaise, le contrat *ModeGPS* va devoir s'adapter à ce nouvel environnement :

Le contrat va reconfigurer le GPS afin de trouver une bonne réception pour au moins 5 satellites. Si la reconfiguration a échoué mais que le GPS reçoit les données d'au moins 3 satellites alors le contrat *ModeGPS* basculera en mode *PowerSave*. Sinon, un message avertira l'utilisateur que le GPS ne peut assurer son contrat. Il en va de même pour le mode *BestTrack* qui impose une écoute de 5 satellites et pour le mode *PowerSave* dont l'écoute de 3 satellites est nécessaire.

Les modes *BestEffort* et *BestTrack* sont valides si et seulement si le niveau d'énergie est suffisant c'est-à-dire que le contrat de type *powerSupply* indique un niveau *high*. Lors du passage du niveau de la batterie de *High* à *Low*, le contrat de mode de fonctionnement doit s'adapter : il bascule en mode *PowerSave*, reconfigure les liaisons satellites en n'*écoutant* que 3 satellites et impose une période de rafraîchissement des données plus longues afin d'économiser de l'énergie.

Si le contrat *PowerLevel* prend la valeur *none*, alors le contrat *modeGPS* n'est plus valide car il n'y a plus assez de d'énergie.

La renégociation de ces contrats ne peut pas être explicités à l'aide de QML.

2.4.3.4 La qualité de service de bout en bout

La QdS offerte par un composant dépend de son environnement. Comme nous l'avons vu ci-dessus, les contrats de QdS peuvent être :

- décomposables (délégations),
- dépendants d'autres contrats,
- ou encore, adaptatifs.

Pour calculer la qualité de service de bout en bout d'un composant, ces différents aspects doivent être pris en compte. QML ne permet donc pas de calculer la QdS de bout en bout dans un système.

Voici une spécification envisageable pour le contrat *ModeGPS* :

```

Specification contract ModeC : GPSPMode {
  // Déclaration des dépendances
  Required: {
    integer nbDecoders;
    PowerC powerC;
    ReliabilityC[] reliability;
  }

  //Déclaration des états possibles du contrat
  States: {
    StartRunning
      ExclusivesStates : {'PowerSave', 'BestTrack', 'BestEffort'};
    StateDependencies:
      powerC.power= 'low' => PowerSave;
  }

  //Déclaration des dimensions
  Dimensions: {
    mode = f1(powerC, nbDecoders);
    BestEffort => precision = f2(nbDecoders, reliability);
    BestTrack => precision = f3(reliability);
    PowerSave => precision = f4(reliability);
    StartRunning => precision = undefined;
  }
}

// Déclaration des valeurs à l'implantation
Implementation:
  BestEffort: if nbDecoders > 5;
  BestTrack:  if nbDecoders = 5 and not StartRunning;
  PowerSave:  if nbDecoders = 3;

```

Grâce à toutes ses informations, le contrat *ModeGPS* est bien défini, et la QdS de bout en bout peut être déterminée. QML ne permet pas d'exprimer les aspects de dépendances, de délégation et d'adaptation, il n'est donc pas en mesure d'exprimer la QdS de bout en bout.

2.5 Conclusion

Dans ce deuxième chapitre consacré aux contrats de qualité de service, une définition de la notion de contrat a été présentée 2 :

Un contrat spécifie les droits et les obligations entre un client et un fournisseur de services. Il détermine les relations entre ces deux acteurs.

Nous avons vu les différents niveaux de contrats dont celui qui nous intéresse plus particulièrement : le contrat de QdS. La QdS d'une application est définie par l'ensemble de ses propriétés extra-fonctionnelles. Nous avons abordé les différentes approches existantes pour la spécification de la QdS, dont QML qui apparaît comme étant la plus complète.

Pourtant, nous avons vu que certains aspects ne sont pas traités dans QML comme la délégation, les dépendances ou le comportement adaptatif de contrats et qu'il ne permettait pas de déterminer la QdS de bout en bout d'un système. Dans le cadre du projet européen QCCS, nous avons besoin de ces aspects de spécification de contrats qui permettent d'obtenir la QdS de bout en bout. Le langage QML répond en partie aux besoins de QCCS, nous nous sommes donc inspirés de QML pour créer notre propre modèle de contrat qui est détaillé dans la partie III.

Après avoir présenté les composants et les contrats de QdS, nous allons donner un aperçu, dans ce dernier chapitre de *l'état de l'art*, des notation et méthode qui sont la base de la méthodologie de QCCS : UML (Unified Modeling Language) et MDA (Model Driven Architecture).

Chapitre 3

Notation et méthode utilisée

Contents

3.1 UML : Unified Modeling Language	57
3.1.1 UML et les composants	60
3.1.2 Modélisation de l'exemple du GPS	62
3.2 MDA : Model Driven Architecture	64
3.2.1 Le PIM : Platform Independent Model	65
3.2.2 Le PSM : Platform Specific Model	66
3.2.3 Les différentes transformations	68
3.2.4 Passage d'un PIM vers un PSM	70
3.2.5 Présentation de l'outil Kase	70
3.2.5.1 La conception du PIM	71
3.2.5.2 Passage du PIM au PSM	71
3.3 Conclusion	74

3.1 UML : Unified Modeling Language

UML [3] permet de modéliser les systèmes quels que soient les langages et les plateformes utilisées : il s'intègre parfaitement dans la philosophie de l'OMG qui veut favoriser l'essor industriel des technologies objet, en offrant un ensemble de solutions technologiques non propriétaires. UML a donc été adopté par l'OMG et intégré à l'OMA comme notation standard en novembre 1997.

- UML est utile dans les différentes activités du développement, en particulier pour :
- l'analyse,
 - la conception,
 - l'implantation,
 - le test.

UML inclut OCL (Object Constraint Language) qui est un langage de représentation des contraintes en approche objet. Nous avons donc un langage visuel des diagrammes

mais aussi une syntaxe pour l'écriture des contraintes. Ce langage de contraintes n'apparaît qu'à partir de la version 1.1 d'UML, avant les contraintes ne pouvaient s'exprimer que sous forme de texte libre. UML intègre des concepts de haut niveau tels que les *collaboration*, les *frameworks*, les *patterns*, et les *composants*.

UML propose une dizaine de diagrammes adaptés aux différentes activités du développement.

Les diagrammes vont être raffinés, complétés au cours du développement.

Notre modèle de contrat se fonde sur le modèle de composant UML2.0 [4]. La figure 3.1 représente la partie du métamodèle d'UML 2.0 v0.671 autour duquel nous avons construit notre modèle :

Définition 3 *Un composant représente une partie du système remplaçable, déployable, et modulable qui encapsule son contenu. Un composant se conforme aux interfaces qu'il expose. Elles représentent les services fournis ou requis par les éléments contenus dans le composant.* □

Dans le métamodèle, un composant *Component* est un sous-type de *Classifier*. Un composant ne peut pas avoir ses propres attribut et opération, mais au lieu de cela ses caractéristiques sont représentées par ses *Interfaces*, *Classifiers* et *Connectors*. Un composant *Component* est composé de *Ports*. Un port est une interface nommée du composant. L'interface définit l'ensemble des opérations et des événements qui sont fournis ou requis par le composant. Les ports sont les points d'attache du composant dans son environnement.

Une *Operation* fournit un service. La spécification d'une opération définit ce que le service fournit et non comment il le fournit. La classe *Operation* hérite de la classe *BehavioralFeature* c'est-à-dire qu'une opération va avoir des paramètres (*Parameter*) qui sont des éléments typés (*ElementTyped*). Du fait qu'une opération est un *BehavioralFeature*, on va pouvoir lui associer un comportement *Behavior*.

Un *Behavior* est la réalisation ou l'implantation d'un *BehavioralFeature*. La spécification d'un *Behavior* peut être donnée sous la forme :

- d'interactions ;
- de machines à états ;
- d'activités ;
- ou encore, de séquences d'actions.

Le comportement d'une opération est lié à l'opération elle-même (comportement intrinsèque) mais peut également être lié à son implantation (comportement extrinsèque). Reprenons l'exemple du GPS : l'interface *ComputerI* permet, entre autres, la gestion des *decoders*. Lors d'une mauvaise réception des données, le *computer* peut reconfigurer les *decoders* grâce à l'opération *ConfigureDecoders()*.

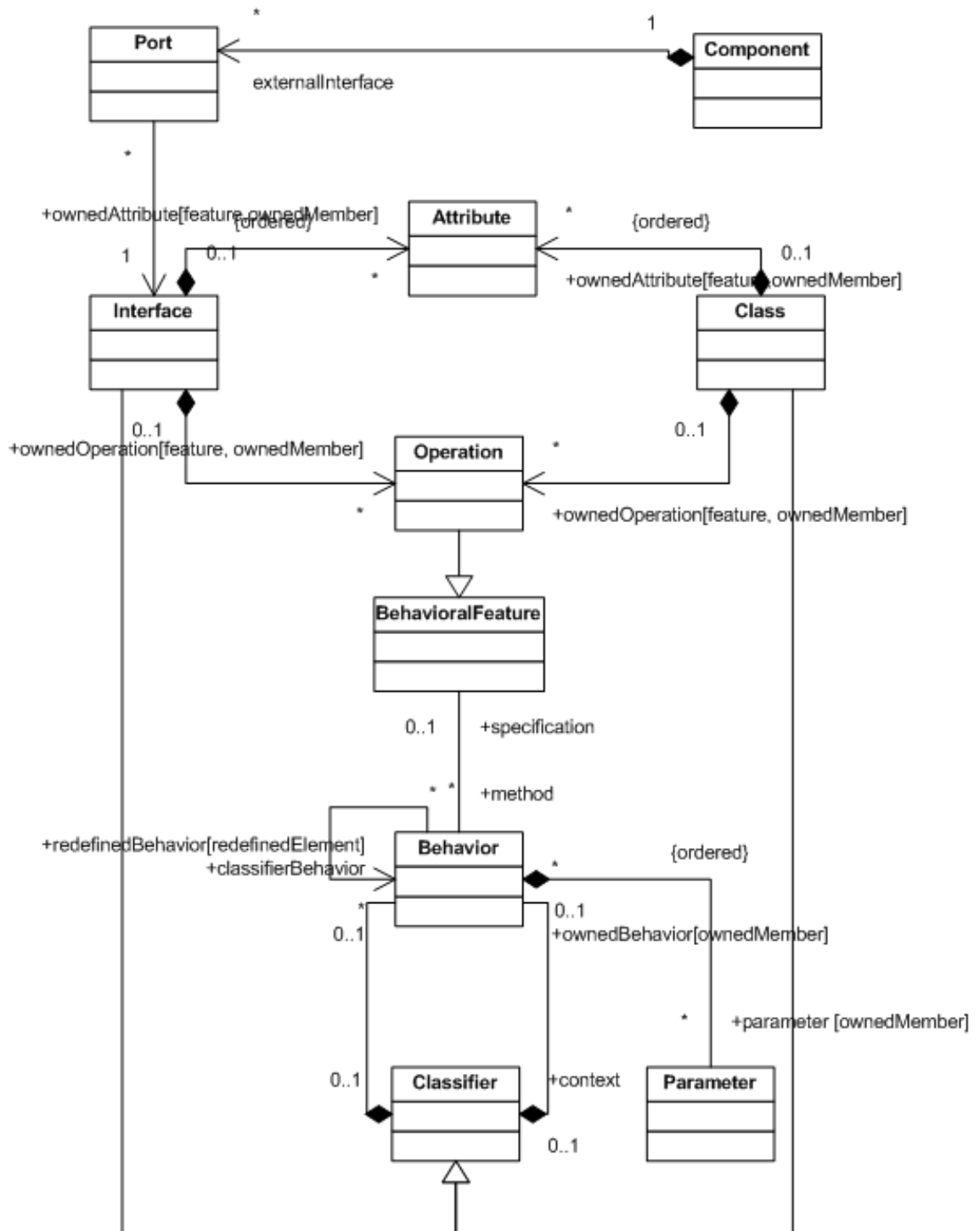


FIG. 3.1 – Le modèle de composant UML 2.0 v0.671.

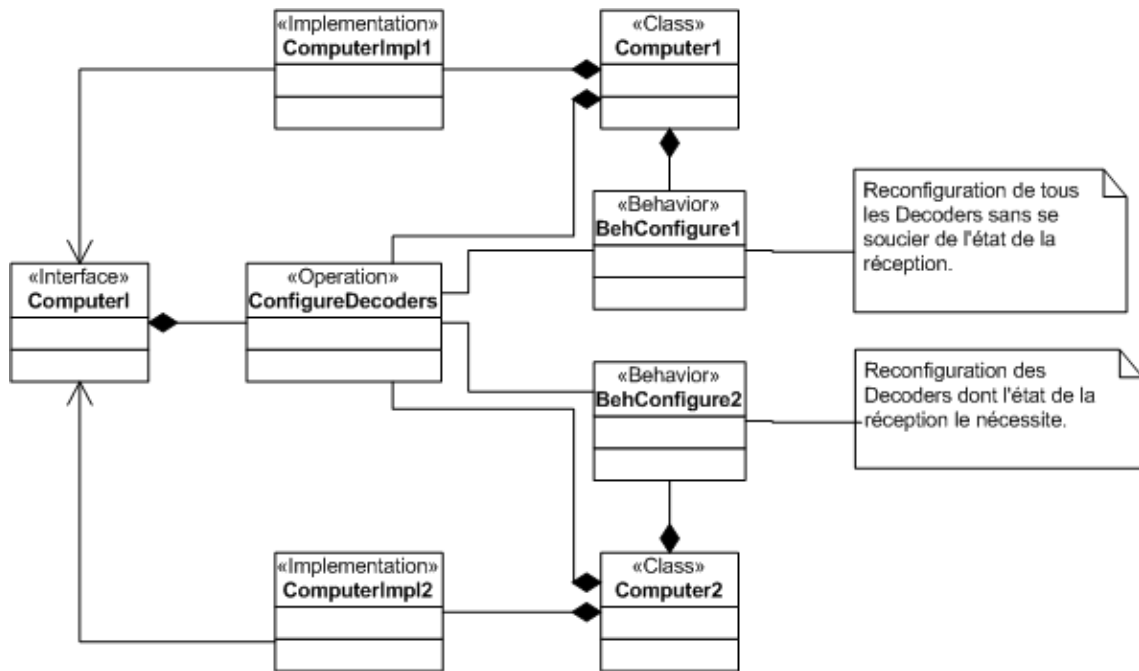


FIG. 3.2 – Deux comportements différents pour l’opération *ConfigureDecoders*.

Dans la figure 3.2, les classes *Computer1* et *Computer2* sont des réalisations de l’interface *ComputerI*, celle-ci est composée d’une opération nommée *ConfigureDecoders*. Selon l’implantation de l’interface *ComputerI*, l’opération *ConfigureDecoders* aura un comportement différent :

- Dans le cas de la classe *Computer1*, l’opération *ConfigureDecoders* aura le comportement *BehConfigure1* : l’opération reconfigure tous les *Decoders* sans se soucier de l’état de la réception des données.
- Dans le cas de la classe *Computer2*, l’opération *ConfigureDecoders* aura le comportement *BehConfigure2* : l’opération reconfigure seulement les *Decoders* dont l’état de la réception le nécessite.

3.1.1 UML et les composants

Le standard UML2.0 v0.671[4] propose un modèle de composants plus élaboré que dans la version UML1. Un composant est une entité instanciable qui interagit avec son environnement à travers des ports. La description du comportement d’un composant est décrit par des *states machines*.

Un port se comporte comme un point d’interaction avec l’environnement du composant. Un port est typé comme une interface c’est-à-dire qu’il peut être fourni ou requis.

Un port requis signifie qu'une instance du composant doit se connecter à une instance de composant fournissant le service demandé via un port fourni. Un connecteur est une entité qui relie les ports des instances de composants.

La figure 3.3 représente les composants nécessaires pour notre GPS. Les composants y sont décrits par les services qu'ils fournissent (*provided interfaces*) et les services qu'ils requièrent (*required interfaces*).

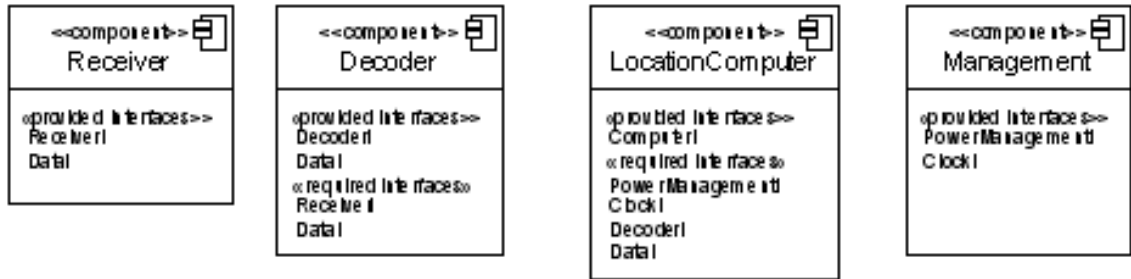


FIG. 3.3 – Les composants du GPS.

Les instances de ces composants vont donc se connecter via leurs ports. Un port fourni est représenté par un cercle (comme pour une interface fournie) alors qu'un port requis est représenté par un demi-cercle. La figure 3.4 montre les instances du GPS et leurs interactions. Les 12 décodeurs-récepteurs étant trop importants à modéliser ici, nous n'en représentons que 3 mais le principe reste le même pour les composants restants.

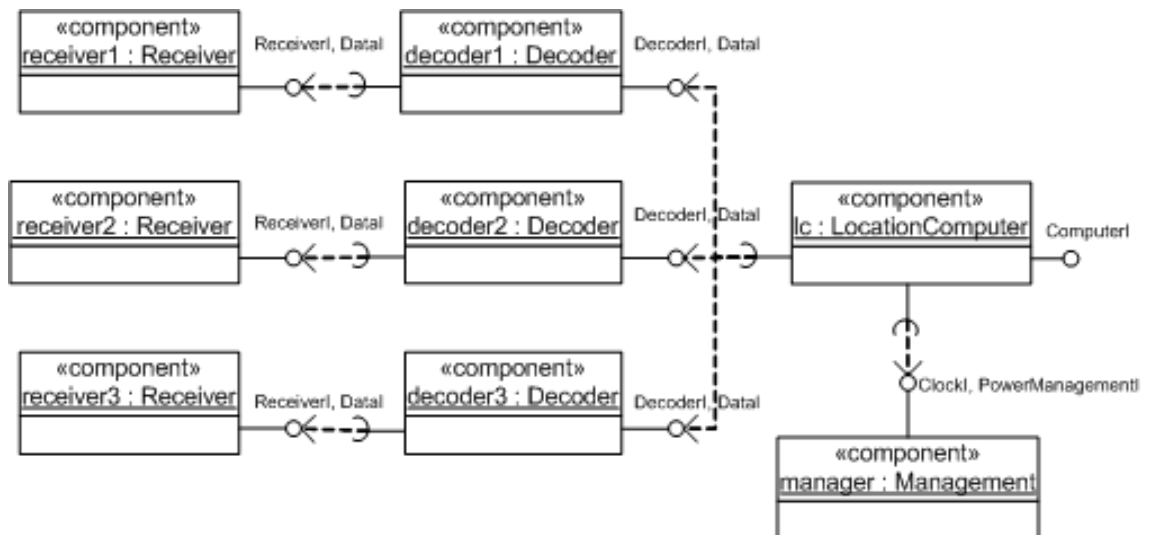


FIG. 3.4 – Les instances des composants du GPS et leurs interactions.

3.1.2 Modélisation de l'exemple du GPS

Nous présentons ici, brièvement, la modélisation de notre GPS à travers quelques diagrammes UML.

- Le diagramme de classe du GPS est représenté dans la figure 3.5.

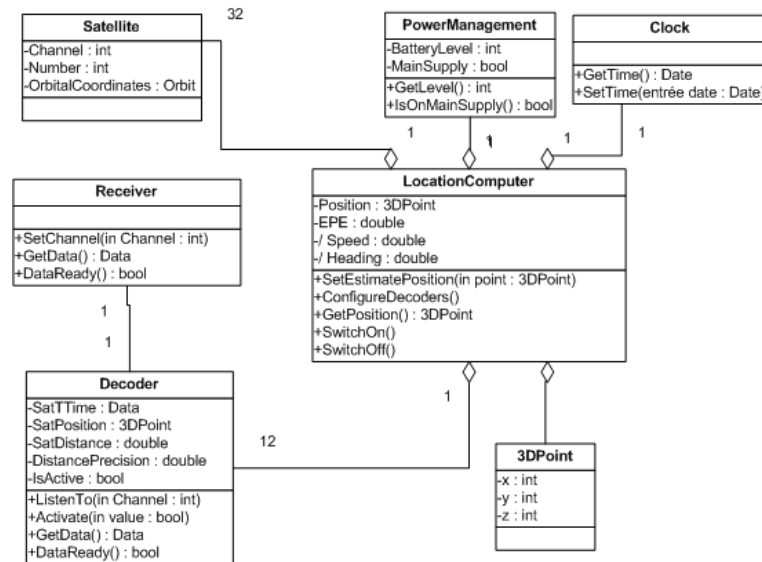


FIG. 3.5 – Diagramme de classe simplifié du GPS.

Ce diagramme se compose de 6 classes :

- La classe *Location computer* est le *moteur* du GPS. Elle peut être lié à une interface graphique ou intégré un système plus complexe. Elle contient les opérations *SwitchOn*, *SwitchOff*, *SetPosition* et *GetPosition*.
- La classe *Power Management* qui gère l'alimentation électrique du GPS.
- La classe *Clock* qui représente l'horloge interne du GPS.
- La classe *Satellite* qui stocke les données d' un satellite telles que ses coordonnées orbitales ou encore son canal d'émission.
- La classe *Decoder* qui permet les décodages des données reçues par le *Receiver*.
- La classe *Receiver* qui reçoit les données du satellite.

Location Computer est composé d'une horloge interne *Clock*, un contrôleur d'énergie *Power Management*, de 12 décodeurs *Decoder* qui servent à décoder les données reçues par les récepteurs *Receiver* (un récepteur par décodeur). De plus , le *Location Computer* possèdent une table qui contient les informations sur les 32 satellites auxquels le GPS peut se "connecter".

- Le diagramme des cas d'utilisations : *uses cases* permet la description du comportement d'un système du point de vue de l'utilisateur sous forme d'actions et de réactions. Un acteur représente une rôle joué par un objet (personne ou chose) qui interagit avec le système. Un objet peut jouer plusieurs rôles et donc peut

apparaître dans divers «uses cases ». La figure 3.6 est un exemple de «use cases »de notre GPS vu dans la partie II §1.1.5 page 20.

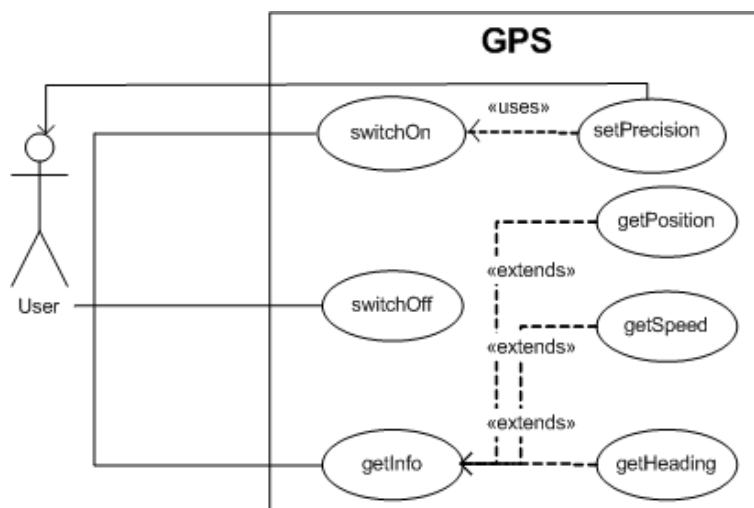


FIG. 3.6 – Exemple de *use cases* pour le gps.

L'utilisateur peut :

- allumer le GPS grâce à l'appel de *switchOn()* qui utilise l'action *setPrecision()* qui va demander à l'utilisateur sa position approximative ;
 - éteindre le GPS grâce à l'appel de *switchOff()* ;
 - récupérer les informations qui l'intéressent avec *getInfo()* : sa position courante avec *getPosition()*, sa vitesse avec *getSpeed()* ou son cap avec *getHeading()*.
- Le diagramme de collaboration montre les interactions entre objets (voir figure 3.7).



FIG. 3.7 – Diagramme de collaboration simplifié du GPS.

C'est une extension du diagramme d'objets. Une interaction est réalisée par un groupe d'objets qui collaborent en échangeant des messages. Une collaboration doit être reliée à une opération ou à un cas d'utilisation et doit le décrire. Un diagramme de collaboration est composé d'objets dans une situation donnée, des liens qui relient les objets qui se connaissent, et des messages échangés par les objets. Le temps n'est pas représenté de manière explicite : les messages sont numérotés pour indiquer l'ordre d'envoi.

- Le diagramme de séquence présente les interactions entre les objets selon un point de vue temporel. Ce diagramme s'avère utile pour :

- documenter des « uses case » : description de l'interaction en termes proches de ceux de l'utilisateur, identification des événements.
- représenter précisément les interactions ; identification et envoi de messages.

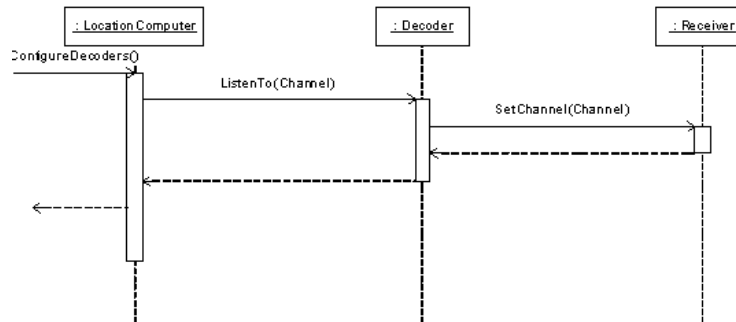


FIG. 3.8 – Diagramme de séquence simplifié du GPS.

La figure 3.8 un diagramme de séquence simplifié du GPS. Lorsque le *Location Computer* doit reconfigurer ses décodeurs *Decoder* (à cause d'une mauvaise réception satellite par exemple), il détermine quels autres satellites couvrent la zone de positionnement du GPS, envoie les canaux d'émission des satellites trouvés aux décodeurs par la fonction *ListenTo(Channel)*. Les décodeurs répercutent l'information aux récepteurs avec la fonction *SetChannel(channel)*.

3.2 MDA : Model Driven Architecture

L'OMG a mis à contribution son expérience dans les middlewares avec CORBA (voir partie II §1.4.1 page 28) et dans la modélisation avec UML (voir partie II §3.1 page 57) afin de proposer une démarche de développement qui permet de séparer les spécifications fonctionnelles d'un système des spécifications de son implantation sur une plate-forme donnée : le Model Driven Architecture. Poursuivant la même politique que CORBA et UML, le MDA [35] [15] est une approche orientée modèle dans laquelle les composants vont pouvoir être spécifiés sans prendre en compte leur aspect technique (c'est-à-dire leur plate-forme d'exécution). L'OMG veut avec le MDA, définir une représentation abstraite et indépendante de toute architecture technique et comportant une multitude de services métiers. Le MDA permet de créer une représentation UML de la logique métier et de lui associer des caractéristiques MDA. Une fois qu'on a cette représentation abstraite appelée PIM (Platform Independent Model), on choisit la plate-forme, et on génère le nouveau modèle appelé PSM (Platform Specific Model).

Le noyau de l'approche MDA est basé sur les technologies de l'OMG [34] : MOF (Meta Object Facility), UML (Unified Modeling Language) (voir partie II §3.1 page

57). Ces technologies sont utilisés pour décrire les PIMs. Un PIM est raffiné autant de fois qu'il est nécessaire pour obtenir le niveau de description souhaité. Ensuite, l'infrastructure est prise en compte pour transformer le PIM obtenu en PSM. De la même manière, plusieurs raffinages de ce PSM vont permettre d'atteindre le niveau de description voulue.

3.2.1 Le PIM : Platform Independent Model

Lors de la conception du système, la première chose à faire sera d'isoler la *logique métier*. Dans cette étape, on va pouvoir déterminer tout ce qui est spécifique à l'application mais qui est totalement indépendant de la technique.

La figure 3.9 représente le PIM du GPS.

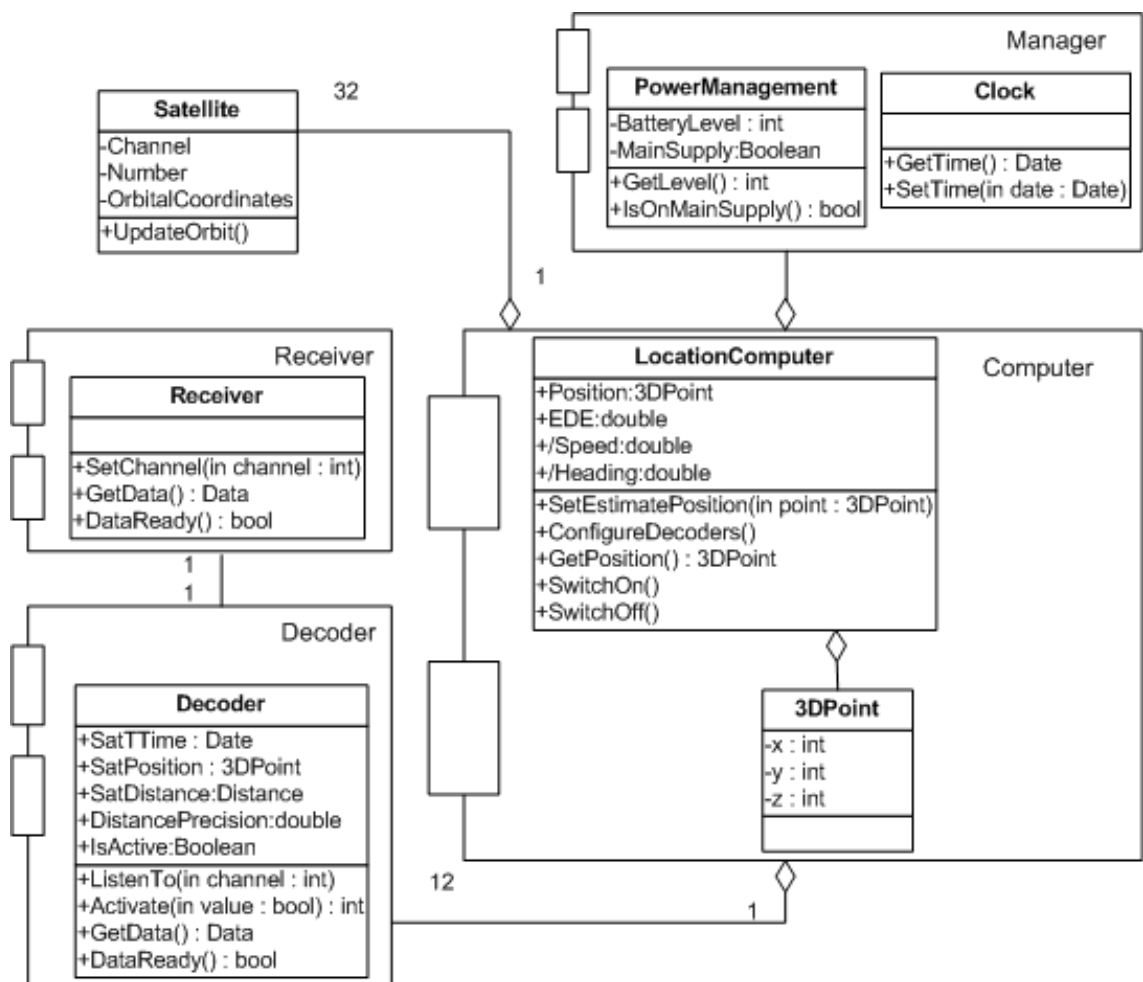


FIG. 3.9 – Le PIM du GPS.

Dans le §3.1.2 page 62, nous avons établi un diagramme de classe simplifié de notre

GPS (voir figure 3.5). Dans ce PIM, nous prenons en compte les composants qui constituent notre GPS. Notre architecture repose sur 4 composants :

- Le premier composant appelé *Computer* est noyau du GPS : il centralise les données émises par les différents satellites via les *Decoders* et les exploite afin de répondre au mieux aux requêtes qui lui parviennent de l'extérieur.
- Le second appelé *Decoder* est spécifique au décodage des données émis par le *Receiver* afin qu'elles soient exploitables par le *Computer*.
- Le troisième appelé *Receiver* permet la liaison avec un satellite, reçoit les données de ce satellite et les envoie au *Decoder* qui lui est associé.
- Le dernier appelé *Manager* s'occupe de la gestion de l'énergie du GPS ainsi que tous les aspects lié au temps.

Ici, aucun aspect technique n'est pris en compte. A partir de ce PIM, nous allons pouvoir générer plusieurs PSMs : un par plate-forme choisie.

3.2.2 Le PSM : Platform Specific Model

Une fois le PIM établi et la plate-forme choisie, on va pouvoir passer à l'élaboration du PSM, c'est-à-dire générer le modèle décrit dans des termes métiers vers une plate-forme spécifique de middleware.

Le PSM est exprimé en UML, or nous avons vu dans la partie II §3.1 page 57 que cette notation était indépendante de toutes plate-formes. Pour pallier ce problème, une solution est de créer un *profile* spécifique à la technologie choisie. Pour la norme Corba (voir partie II §1.4.1 page 28), un profile spécifique a été adopté en 2000 qui définit les stéréotypes propres à la technologie Corba. La figure 3.10 est une représentation UML d'une interface Corba.

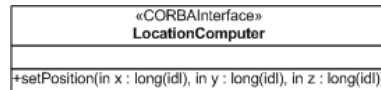


FIG. 3.10 – La représentation UML d'une interface Corba.

Comme nous l'avons vu dans la partie II §1.4.4 page 34, la plate-forme d'exécution choisie dans le projet européen QCCS est la plate-forme .NET . Nous avons donc établi le PSM du GPS (voir figure 3.11) à partir du PIM décrit dans la partie II §3.2.1 page 65.

Ce diagramme représente le PSM .NET du GPS : on y retrouve des informations spécifiques à la technologie .NET comme l'utilisation des *delegates*. Une déclaration de *delegate* définit un type de référence qui est employé pour encapsuler une méthode avec une signature spécifique. Une instance de *delegate* encapsule une instance de méthode. Les *delegates* sont semblables aux pointeurs de fonctions *C++* ; cependant les *delegates* sont fortement typés et sécurisés.

La représentation d'un composant dans le PSM indique sa structure pour la technolo-

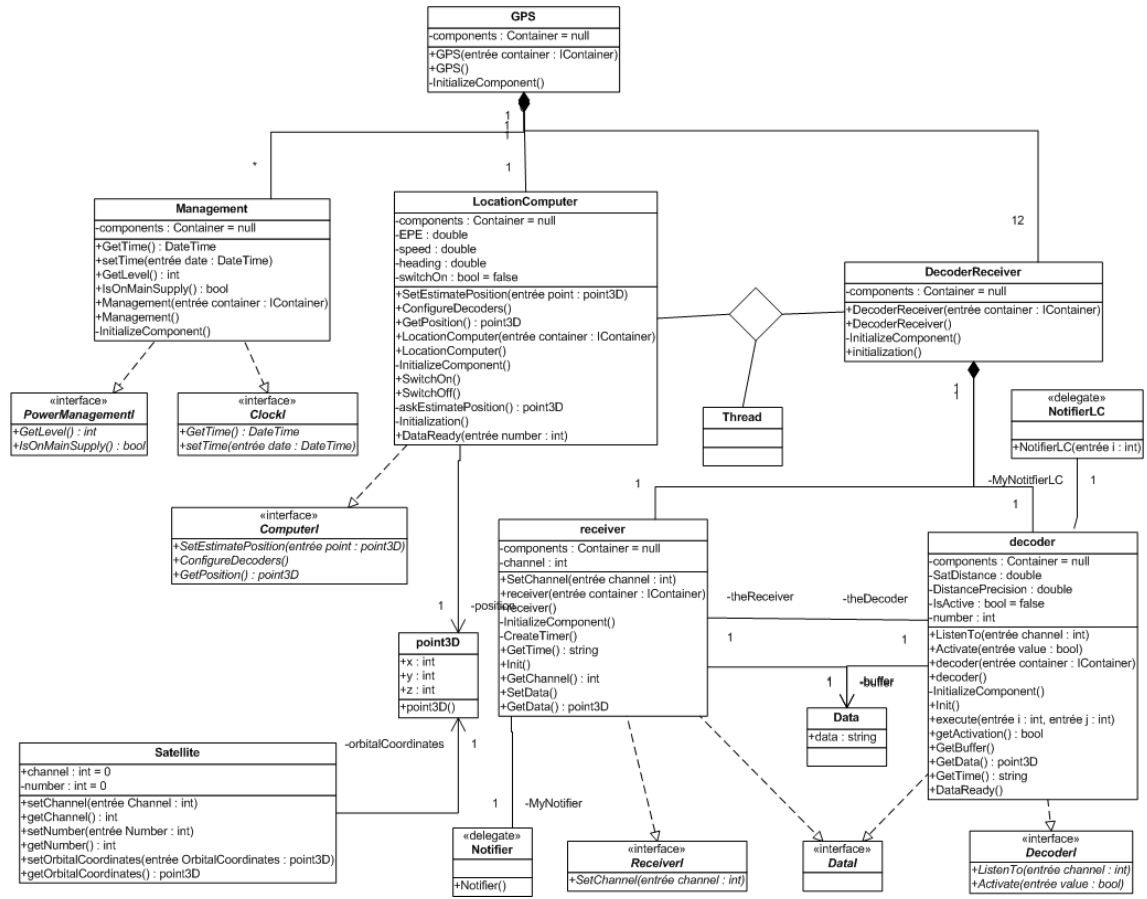


FIG. 3.11 – Le PSM du GPS.

gie .NET. Un composant .NET est une classe qui hérite de *System : :ComponentModel : :Component*. Les types doivent être spécifiés, par exemple, un type *int* dans un PIM devient un type *Csharp : :int* dans un PSM .NET ou encore la classe *Data* qui devient est fourni par le package *System : :IO : :Stream*. Voici une description succincte du code pour la déclaration du composant *GPS*.

```
public class GPS : System.ComponentModel.Component
{
    private System.ComponentModel.Container components = null;

    public const int nbDec = 12;
    public const int nbsat = 32;

    static public DecoderReceiver[] listDecoderReceivers=
        new DecoderReceiver[nbDec];
    static public Management manager = new Management();
    static public LocationComputer lc = new LocationComputer();

    public GPS(System.ComponentModel.IContainer container)
    {
        container.Add(this);
        InitializeComponent();
    }
    .
    .
    .
}
```

3.2.3 Les différentes transformations

La figure 3.12 présente le métamodèle de description du MDA. Les PIMs, les PSMs et les techniques de transformations sont décrits à l'aide de métamodèles généralement exprimés avec des technologies de l'OMG telles que : MOF ou UML.

Un des éléments clés dans l'approche MDA est la notion de *mapping*. Le *mapping* ou transformation de modèle, est un ensemble de règles et de techniques utilisées pour transformer un modèle en un autre. Les *mappings* sont utilisés pour transformer :

- Un PIM vers un PIM : cette transformation est utilisée lors d'amélioration, de filtrage ou de spécialisation des modèles durant le cycle de vie du développement tant que ces changements sont indépendants d'une quelconque technologie. Les transformations PIM-PIM sont généralement employées lors des raffinages d'un modèle.

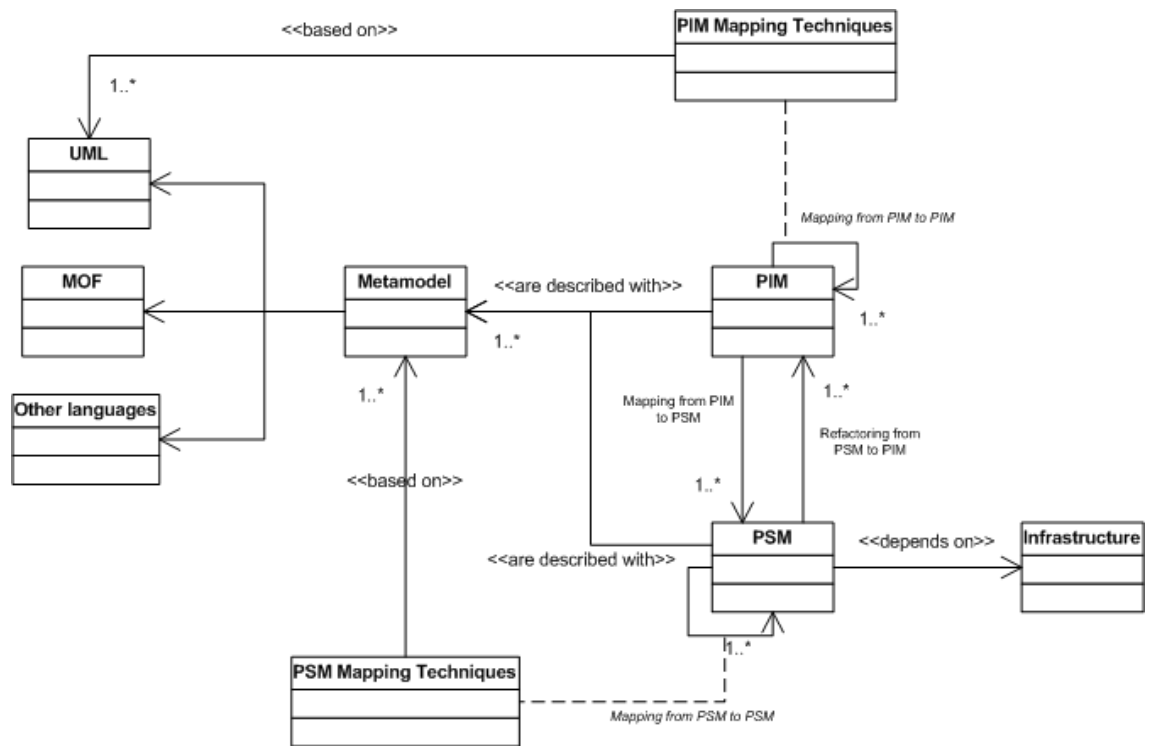


FIG. 3.12 – le métamodèle de description du MDA.

- Un PIM vers un PSM : cette transformation est utilisée quand le PIM est suffisamment raffiné pour être projeté dans une infrastructure d'exécution. La projection est fondée sur les caractéristiques de la plate-forme. Ces caractéristiques vont être décrites en utilisant la notation UML et éventuellement à l'aide d'un *profile* qui donne la description des concepts communs à la plate-forme.
- Un PSM vers un PSM : tout comme la transformation du PIM vers le PIM, le passage d'un PSM à un PSM est généralement employé lors des raffinages de modèles.
- Un PSM vers un PIM : cette transformation est requise lors de l'abstraction des modèles déjà implantés pour une technologie particulière. Cette procédure ressemble souvent à une exploitation du processus qui est difficilement automatisable.

3.2.4 Passage d'un PIM vers un PSM

Il y a quatre façons de transformer un PIM exprimé en UML en un PSM :

De manière totalement manuelle : l'étude du PIM et la construction du PSM se font manuellement avec des étapes de raffinages successifs.

De manière manuelle avec utilisation de modèles : l'étude du PIM et la construction du PSM peuvent se faire manuellement avec l'utilisation de modèles qui vont alléger la construction et les étapes de raffinements.

De manière semi-automatique : un algorithme est appliqué au PIM et renvoie un squelette du PSM qui sera complété manuellement grâce aux mêmes modèles que ceux utilisés ci-dessus dans la *manière manuelle avec utilisation de modèles*.

De manière automatique : un algorithme crée un PSM complet à partir du PIM. Dans le projet européen QCCS, le passage du PIM vers le PSM se fait automatiquement et est intégré dans l'outil de modélisation *Kase* (voir partie II §3.2.5 page 70).

3.2.5 Présentation de l'outil Kase

L'université de Berlin, partenaire du projet QCCS, a développé un outil, appelé *Kase*, qui permet les transformations de modèle. Le but du modèle de transformation employé par l'outil *Kase* est de faciliter la mise en oeuvre et la compréhension des transformations de modèles. Au lieu d'employer un langage de programmation traditionnel pour décrire les transformations, *Kase* emploie UML pour décrire le processus de transformation. Ce processus se décompose de deux parties :

- Les transformations complexes peuvent se décomposer en transformations plus simples. De ce fait, le développeur de la transformation va pouvoir indiquer l'ordre du traitement des différentes transformations par un diagramme d'activité.
- Dans le cas des transformations simples, tous les éléments qui vont être transformés sont décrits ainsi que leurs aspects après la transformation. Le développeur de la transformation crée un modèle à l'aide d'un outil UML tel que *Kase*. Tous les éléments du modèle de transformation vont être recherchés dans le modèle à transformer. Une fois les éléments trouvés, ils sont transformés en d'autres éléments ou simplement modifiés.

L'un des avantages dans l'utilisation de *Kase* est que le développeur de transformations n'a pas à se préoccuper de la recherche des éléments dans le modèle à transformer. Il s'intéresse uniquement aux types des éléments à transformer et du résultat obtenu après la transformation. En outre il est facile de comprendre la structure de la transformation, puisque la transformation entière est exprimée en un ensemble de diagrammes d'UML.

Kase est conforme à UML1.x étendu avec quelques notions d'UML2.0 tels que, par exemple, le concept de composant UML2.0. *Kase* permet la conception des diagrammes de classes, de séquences, de cas d'utilisation, d'activités, d'états, de composants ou de déploiement. La figure 3.13 donne un aperçu de l'outil *Kase*.

3.2.5.1 La conception du PIM

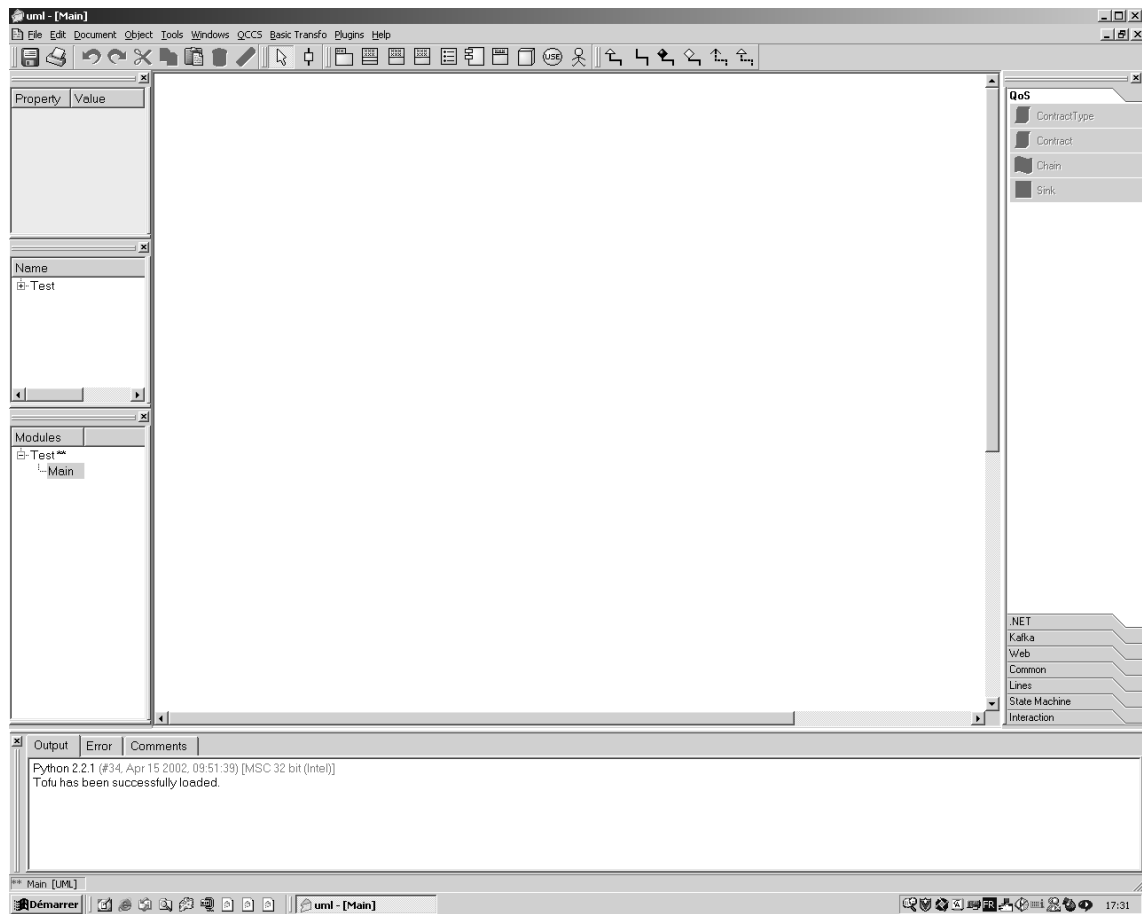
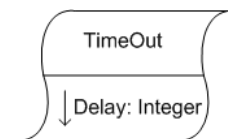
Dans cette phase, le concepteur utilise *Kase* comme un outil de modélisation en utilisant les profils communs pour les éléments UML.

La figure 3.15 montre un *ContractType* nommé *TimeOut* qui a une dimension *delay*. Chaque dimension est décrite par un nom, une unité, une valeur par défaut ainsi que d'un opérateur (↑ ou ↓) permettant la vérification de conformité entre contrats. Cet opérateur s'apparente aux mots-clés *increasing* et *decreasing* utilisés dans QML (voir partie II §2.3 page 46). Le contrat *TimeOutC* est une implantation du *ContractType TimeOut* pour lequel on a spécifié la valeur pour la dimension *Delay*

Dans la figure ??, le composant *Component1* fournit un service *AI* réalisé par la classe *A*. Ce service est contractualisé : le temps de réponse maximal est de 10 secondes.

3.2.5.2 Passage du PIM au PSM

Kase permet le passage automatique d'un PIM vers un PSM.Net : Un composant PIM est transformé au niveau PSM .NET en un package constitué d'une classe ayant le même nom. Cette classe hérite de la classe *System : :ComponentModel : :Component* et implante les différentes interfaces décrites dans le PIM du composant. Les ports et les connections sont remplacés par des références aux interfaces. La figure 3.16 représente le PSM du GPS dans *Kase*.

FIG. 3.13 – L'outil *Kase*.FIG. 3.14 – Le *contractType TimeOut* avec *Kase*.

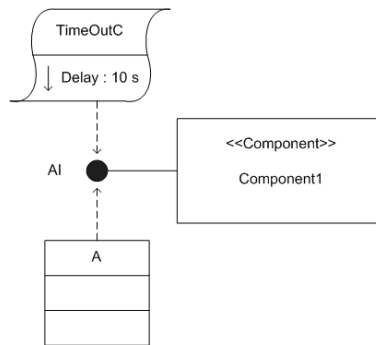


FIG. 3.15 – Le composant *Component1* contractualisé.

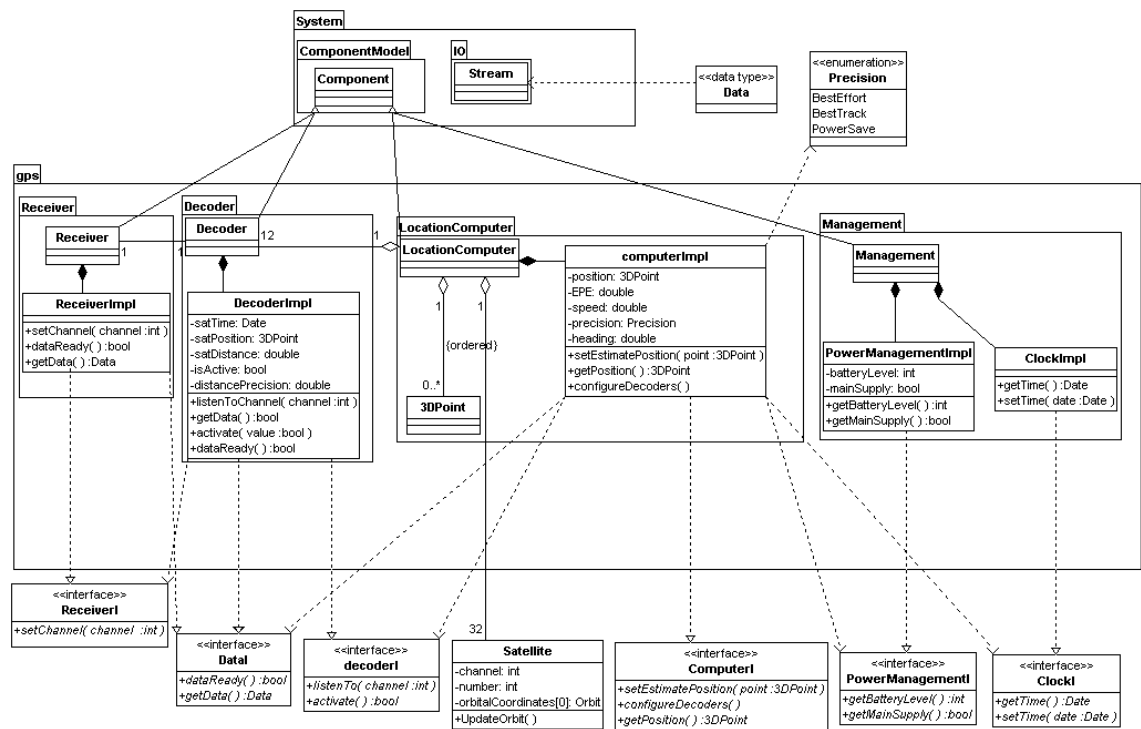


FIG. 3.16 – Le PSM du GPS dans *Kase*

3.3 Conclusion

Dans ce dernier chapitre de l'état de l'art, nous avons donné un aperçu de la notation UML et de la méthode MDA qui sont toutes deux utilisées dans le projet QCCS et proposées par l'OMG. Nous avons appliqué ces notation et méthode sur notre GPS.

Tout au long de cette première partie, nous avons vu, les éléments qui sont à la base de notre méthodologie :

- les composants logiciels ;
- les contrats de qualité de service ;
- UML et MDA ;

Le travail que j'ai effectué s'inscrit dans le cadre du projet européen QCCS. Le but de ce projet est de développer une méthodologie et des outils pour la conception et la mise en œuvre de composants distribués et contractualisés.

Dans le chapitre « *Les contrats de qualité de service* », nous avons dégagé la problématique liée à ce projet. Dans la deuxième partie, nous allons présenter la contribution de cette thèse. Tout d'abord, nous verrons le modèle de contrat QoSCL qui répond aux besoins de spécifications des aspects non traités avec QML. Ensuite, nous présentons les transformations de modèles et leurs outils. Et nous terminerons par l'évaluation de cette approche par les différents utilisateurs du projet QCCS.

Troisième partie
Contribution

Chapitre 4

Les composants et la qualité de service

Contents

4.1	Introduction	77
4.2	Extension de QML	78
4.3	Le modèle de contrat QoSCL (Quality of Service Constraint Language)	81
4.3.1	Principes	81
4.3.2	Les principaux termes	81
4.3.3	Comparatif QML/QoSCL	84
4.4	Les différents aspects de spécification	85
4.4.1	Délégation de contrats	85
4.4.2	Dépendance de contrats	85
4.4.3	Comportement adaptatif	87
4.4.4	Raffinage de contrat	87
4.5	Exemple du GPS	89
4.6	La qualité de service de bout en bout	91
4.7	Calcul de la qualité de service de bout en bout	92
4.8	Conclusion	95

4.1 Introduction

Dans la première partie de ce mémoire, nous avons vu que le paradigme à composants permet une meilleure réutilisation que le modèle à objets. Un composant logiciel spécifie clairement les services offerts et requis. Cette spécification facilite l'intégration du composant dans un logiciel.

Aujourd'hui, le service rendu par le logiciel n'est plus le seul critère de choix. Un critère prépondérant pour le choix d'un logiciel est la qualité de service qu'il offre. Une

technique pour la gestion de la QdS est l'utilisation de contrats de QdS qui déterminent les droits et les obligations entre un fournisseur et un client de service.

QML apparaît comme étant l'approche la plus complète pour la spécification de la QdS. Pourtant, certains aspects ne sont pas traités dans QML : la délégation, la description du comportement adaptatif ainsi que les dépendances entre contrats.

La première étape de cette thèse a été d'étendre QML afin d'y intégrer la notion de dépendance de contrats.

4.2 Extension de QML

Reprenons l'exemple du contrat de type *Reliability* sur le *Decoder*, la description de ce contrat peut s'exprimer sous la forme :

```
ReliabilityC (var : Reliability) = Reliability contract {
  confidence == high;
}
```

Ce qui signifie que le contrat de fiabilité *ReliabilityC* dépend d'un contrat de type *Reliability*.

Pour permettre l'ajout de paramètres dans une déclaration de contrat, nous avons ajouté à la grammaire de QML, le terme *contFunc* qui correspond à un contrat paramétré :

```
contFunc ::= xc( var1: yt1, ..., varn: ytn)
```

xc correspond au nom du contrat, les termes *var1* à *varn* déterminent les *n* variables passées en paramètres qui sont des *ContractType* de type *yt*.

Une fois cette dépendance décrite, il faut pouvoir l'utiliser dans la description des dimensions numériques. Nous allons ajouter une dimension *NOF* qui signifie *NumberOfFailures* au *Contract Type OurReliability*. *NOF* détermine le nombre d'échecs par an du service :

```
type Reliability = contract {
  confidence : increasing enum {none, low, medium, high} with order {none
    < low; low < medium; medium < high; }
  NOF : decreasing numeric no/year;
};
```

Nous voulons que la dimension *NOF* s'exprime par le biais d'une autre dimension. Plus précisément, dans notre exemple, le nombre d'échecs doit être inférieur à deux fois

le nombre d'échecs du contrat passé en paramètre (*var*) :

```
ReliabilityC (var : Reliability) = Reliability contract {
    NOF < var.NOF * 2;
    confidence == high;
}
```

Pour permettre cette spécification, nous avons ajouté à la grammaire de QML la notion de *function* dont voici la déclaration :

```
function ::= var.dimName
          | function operator function
          | number
```

Une fonction peut :

- être égale à la valeur de la dimension passée en paramètre :
`function ::= var.dimName`
- s'exprimer sous forme de combinaisons de fonctions :
`function ::= function operator function`
- ou correspondre à une valeur numérique :
`function ::= number`

Grâce à cet ajout, on peut donc avoir des expressions telles que :

```
NOF < (var1.NOF+var2.NOF+var3.NOF)/3
```

Ces fonctions peuvent être combinées avec les différents aspects déterminés dans QML (*percentile*, *mean*, *variance*, *frequency*) :

```
aspectFunc ::= percentile percentNum constraintOp function
           | mean constraintOp function
           | variance constraintOp function
           | frequency freqRangeFunc constraintOp number %
```

On peut donc exprimer des contraintes de la forme :

```
NOF {
    percentile 90 < var.NOF;
    mean < var.NOF * 2;
}
```

L'annexe A présente la grammaire QML étendue pour l'expression de ces paramètres ainsi que l'ajout de fonctions simples portant sur les contrats. Les lignes précédées d'une (*) correspondent aux lignes modifiées ou ajoutées.

La figure 4.1 représente l'arbre syntaxique de la spécification QML du contrat *ReliabilityC* :

ReliabilityC (*var* : *Reliability*) = *Reliability* contract {

NOF < *var.NOF* * 2;
confidence == *high*;

}

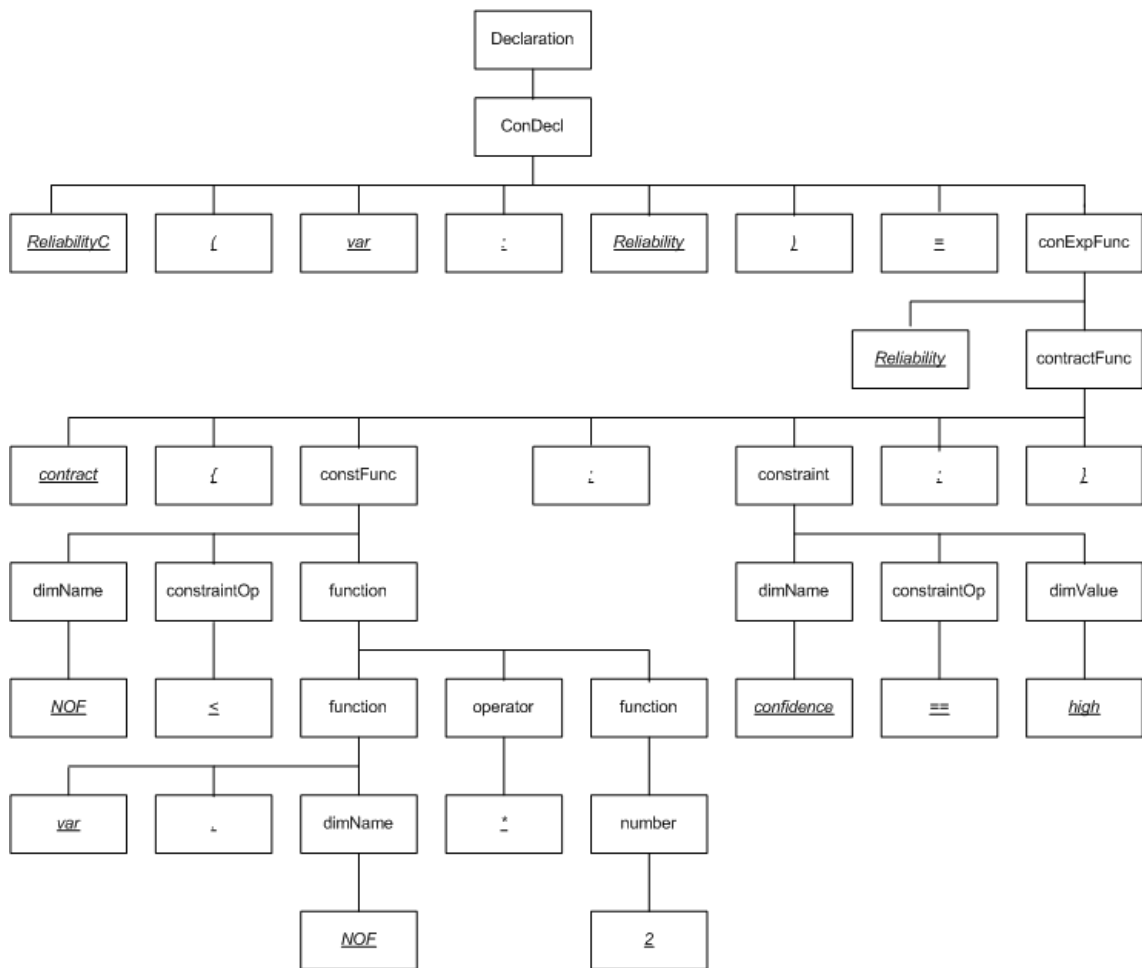


FIG. 4.1 – Arbre syntaxique du contrat *ReliabilityC*.

4.3 Le modèle de contrat QoSCL (Quality of Service Constraint Language)

4.3.1 Principes

Notre modèle de contrat est fortement inspiré de QML, et étend la partie d'UML2.0 qui contient les composants. Nos principales motivations sont principalement de concevoir des contrats à partir d'un outil de modélisation en UML et deuxièmement d'hériter naturellement des mécanismes existants tels que par exemple la conformance des types. L'extension de QML a permis de *capturer* les notions importantes qui nous ont inspirés pour notre méthodologie.

Nous avons conservé de QML l'idée de *famille de contrats* avec la notion de *ContractType* qui représente la catégorie de la QdS. Dans le modèle QoSCL, un *ContractType* est vu comme un type de comportement et a donc été modélisé comme une interface.

Dans notre modèle, un *ContractType* représente la signature d'un contrat et non plus un modèle : le contrat n'est donc plus une instance de *ContractType* mais devient une implantation d'un *ContractType*.

Le contrat (appelé *contractQoSCL*) est donc une classe spécialisée qui implante une interface spécialisée nommée *contractType*.

Une dimension en QML est une contrainte statique alors que dans le modèle QoSCL, les dimensions deviennent des opérations paramétrables qui vont permettre de définir l'interdépendance des contrats.

Nous avons donc étendu la partie du métamodèle présenté dans la partie II §3.1 page 59 afin d'y introduire les contrats de QdS.

4.3.2 Les principaux termes

La figure 4.2 représente la modélisation de notre modèle. Les classes grisées représentent les classes ajoutées au métamodèle d'UML2.0 afin d'inclure la notion de contrat de QdS.

Nous avons ajouté 5 éléments au modèle de composant d'UML2.0 :

1. **ComponentQoSCL** : Un composant de QdS appelé *ComponentQoSCL*, est un composant, au sens UML 2.0, que l'on a spécialisé. Un composant fournit et requiert des ports typés par des interfaces fonctionnelles. Ces interfaces exhibent les services fonctionnels offerts ou requis par le composant. Un *ComponentQoSCL* hérite de *Component* auquel on a ajouté le concept de port contractuel : *PortQoSCL*.

```
self.generalization.parent->includes(Component)
self.externalInterface->forall (p|
```

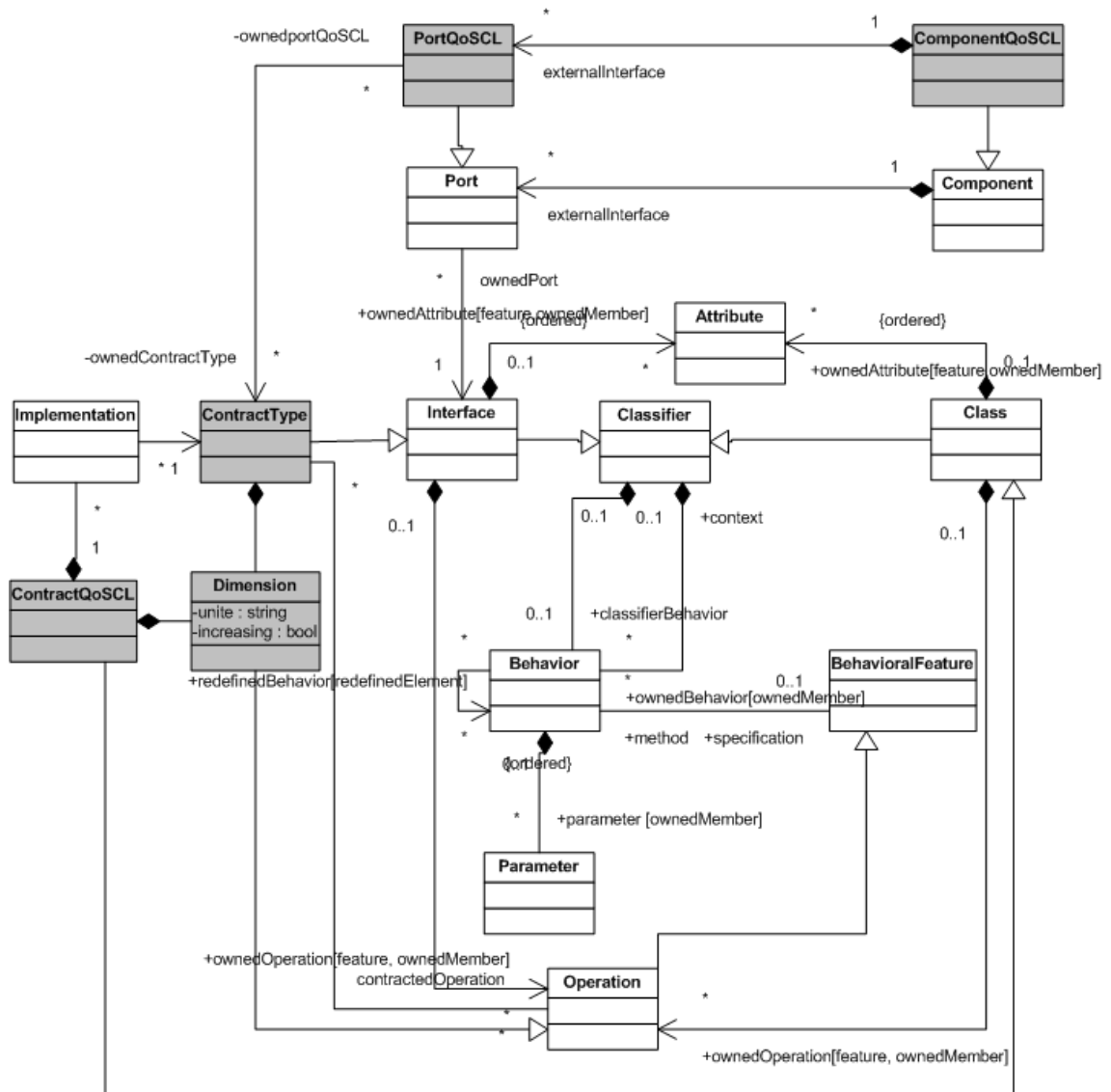


FIG. 4.2 – Notre modèle de composant QoSCL.

```

    p.oclIsKindOf(Port      ) or
    p.oclIsKindOf(PortQoSCL )
  )

```

2. **PortQoSCL** : Un *portQoSCL* est un port spécialisé : c'est donc un point d'attache du composant. Il est lié à une interface contractualisée ou non. Dans le cas d'une interface contractualisée, on spécifiera le type de QdS par un *ContractType*. On a donc ajouté une association entre la classe *PortQoSCL* et la classe *ContractType* avec une multiplicité $0..*$. Si la multiplicité est différente de zéro alors l'interface est contractualisée.

```

self.generalization.parent->includes(Port)
self.ownedContractType->forAll (ct |
  ct.oclIsKindOf(ContractType)
)

```

3. **ContractType** : Un *ContractType* est une interface spécialisée. On va ajouter aux caractéristiques de cette interface des *Dimensions*.

La spécification UML2.0 précise que si 2 ports sont connectés, l'interface fournie doit être du même type que l'interface requise ou un sous-type de celle-ci. Étant donné qu'un *ContractType* est une interface spécialisée et qu'un *PortQoSCL* hérite de la classe *Port*, lors de la connection de 2 *PortQoSCL*, il doit y avoir conformité entre les interfaces mais aussi entre les *ContractType*.

Un *ContractType* peut porter sur un sous-ensemble (c'est-à-dire sur certaines opérations) de l'interface que l'on veut contractualiser. L'association entre le *ContractType* et la classe *Operation* permet de distinguer sur quel élément de l'interface se porte le *ContractType*. La multiplicité au niveau des opérations correspond au nombre d'opérations contractualisées. Si la multiplicité a pour valeur 0, nous prendrons comme convention que tous les éléments de l'interface sont contractualisés. Un invariant OCL a été ajouté au modèle :

- les opérations associés au *ContractType CT* doivent être spécifiées dans une *Interface I*,
- le *ContractType CT* et l'*Interface I* doivent être associés au même *PortQoSCL*.

```

self.generalization.parent->includes(Interface)

```

```

self.allFeatures->forAll(f |
  f.oclIsKindOf(Operation)or
  f.oclIsKindOf(Reception)or
  f.oclIsKindOf(Dimension)
)

```

```

self.contractedOperation->forAll(op |

```

```

op.oclIsKindOf(Operation) and
op.owner.ownerPort->intersection(self.ownedPortQoSCL)->size() >=1
)

```

4. **ContractQoSCL** : Un *ContractQoSCL* est l'implantation d'un *ContractType*. Un *ContractQoSCL* est une classe spécialisée dans laquelle on va implanter les différentes dimensions décrites dans le *ContractType*.

```

self.generalization.parent->includes(Class)

self.feature->forall->(f |
    f.oclIsKindOf(Operation )or
    f.oclIsKindOf(Attribut  )or
    f.oclIsKindOf(Dimension )
)

```

5. **Dimension** : Une dimension est une opération qui peut avoir une unité et une relation sémantique (pour la conformité des contrats). Si l'attribut *increasing* prend la valeur *true* cela signifie que lors du choix du contrat, on privilégiera le contrat dont la valeur de sa dimension sera la plus grande. Une dimension pourra avoir comme paramètre un contrat puisqu'un paramètre est un élément typé *TypedElement* : un *TypedElement* référence un *Classifier* ou un *ContractQoSCL* est (par héritage) un *Classifier*.

```

self.generalization.parent->includes(Operation)

```

Grâce à ce paramétrage, on va pouvoir spécifier la dépendance de contrats (voir partie III §4.4 page 85) :

- Si aucune dimension d'un contrat n'est paramétrée alors on peut dire que le contrat est indépendant. C'est le cas pour un service qui ne détermine sa QoS qu'avec des mesures fonctionnelles, comme par exemple le contrat *TimeOut* défini dans le GPS (voir partie II §2.2.1 page 40). La *Dimension Delay* se calcule par une fonction telle que *getTime*. La validité du contrat passe par une mesure fonctionnelle.
- Si non, le contrat est dépendant d'autres contrats, c'est-à-dire que pour obtenir la validité du contrat, il faudra d'abord vérifier la validité des contrats dont il dépend (voir partie III §4.10 page 92).

4.3.3 Comparatif QML/QoSCL

Le tableau 4.1 présente un récapitulatif des principaux termes usités dans QML et QoSCL avec leurs descriptions.

	QML	QoSCL
Dimension	Domaine de valeurs	Opération
ContractType	Groupe de dimensions	Interface composée de dimensions
Contract	Instance de ContractType : affectation des valeurs aux dimensions	Implantation du ContractType : mise en œuvre des dimensions (fonctions dépendantes de services)
Profile	Lien entre interface et contrat	Notion de service. Modèle de composant

TAB. 4.1 – Comparatif des termes usités dans QML et QoSCL.

4.4 Les différents aspects de spécification

Le modèle de contrat QoSCL permet la description des trois aspects présentés dans la partie II §2.4.3 page 50 : la délégation, la dépendance et le comportement adaptatif des contrats. Nous présentons, dans cette section, ces différents aspects dans le modèle QoSCL auxquels nous ajoutons le raffinement de contrats qui, à la différence des autres, est exprimable en QML

4.4.1 Délégation de contrats

La délégation de contrats est facilement exprimable dans notre modèle grâce à l'utilisation d'UML2.0. Au niveau du diagramme de classe, une délégation va pouvoir se représenter comme une composition. Un contrat étant une classe spécialisée, on va spécifier les contrats composites grâce à un lien de composition. La figure 4.3 donne la modélisation de cette délégation. Soient $C1$ un contrat composite, $C2$ et $C3$ ses composantes. Sémantiquement, la délégation du contrat $C1$ correspond à :

$$C1 = C2 \wedge C3 \quad (4.1)$$

La figure 4.4 représente la délégation au niveau des composants.

4.4.2 Dépendance de contrats

De la même façon que pour la délégation de contrats, l'expression de la dépendance entre contrats se fait naturellement dans le modèle QoSCL. Elle est représentée par une association entre les deux contrats. De plus dans la partie III §4.3.2 81, nous avons vu qu'une dimension permet de spécifier la dépendance grâce à ses paramètres. L'association et la déclaration de paramètres vont donc exprimer la dépendance de contrats. Dans la figure 4.5, le contrat $C1$ dépend du contrat $C2$, cette dépendance s'exprime au

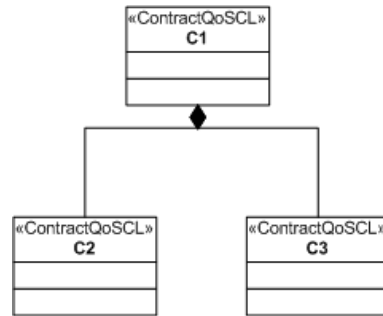


FIG. 4.3 – La délégation de contrats dans un diagramme de classe.

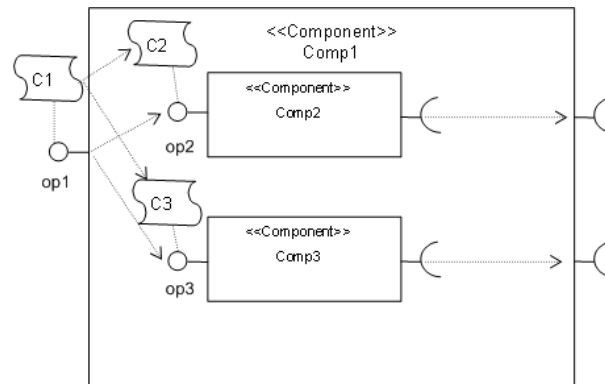


FIG. 4.4 – La délégation de contrats au niveau du composant.

niveau de la dimension *Dim1*. Sémantiquement, la dépendance du contrat C1 correspond à :

$$C1 = f(C2.Dim1) \quad (4.2)$$

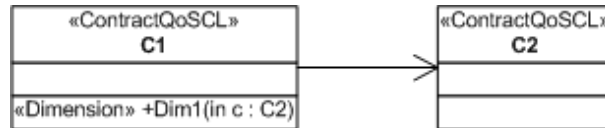


FIG. 4.5 – La dépendance de contrats.

4.4.3 Comportement adaptatif

Le comportement d'une classe s'exprime en UML2.0 par la classe *Behavior* à l'aide de diagramme de séquence, d'activités, d'interactions ou encore d'une machine à état. Le comportement adaptatif d'un contrat permet d'exprimer la négociation du contrat : si le contrat devient invalide quelles sont les actions à effectuer ? Qui prévenir ? Comment se déroule la négociation ? Quels sont les acteurs de cette négociation ? etc... Toutes les réponses à ces questions vont pouvoir être exprimées dans le comportement du contrat et dans le comportement des dimensions concernées.

Dans le modèle QoSCL, nous exprimons le comportement adaptatif des contrats principalement à partir des machines à états. Le diagramme 4.6 présente le comportement (via une machine à états) du contrat *ModeGPS* :

Une fois le choix du mode effectué par l'utilisateur, le contrat passe dans l'état *Mode BestEffort*, le GPS est configuré pour le mode choisi, le contrat est dans l'état *Décodeurs actifs*. Si le niveau de la batterie est faible, le contrat passe dans l'état *Mode PowerSave* afin de préserver son énergie et il y a reconfiguration du système pour ce nouveau mode et le contrat repasse dans l'état *Décodeurs actifs*. Mais si le niveau de batterie est vraiment trop faible, le GPS est arrêté. Enfin, si la liaison est mauvaise et qu'un décodeur devient inactif, le GPS passe dans l'état *Reconfiguration*.

4.4.4 Raffinage de contrat

Tout comme dans QML, QoSCL permet le raffinement de contrat. Le raffinement s'exprime dans QoSCL par simple lien d'héritage. Si nous prenons l'exemple de la figure 4.7, le contrat *TimeOut1* est un *contractType* qui assure que le temps d'exécution de l'opération associée au contrat sera compris entre 5 et 10 secondes. Mais le concepteur a besoin d'une contrainte plus forte pour une autre opération du modèle. Dans ce cas, il suffit de raffiner *TimeOut1* en un *contractType TimeOut2* avec une contrainte

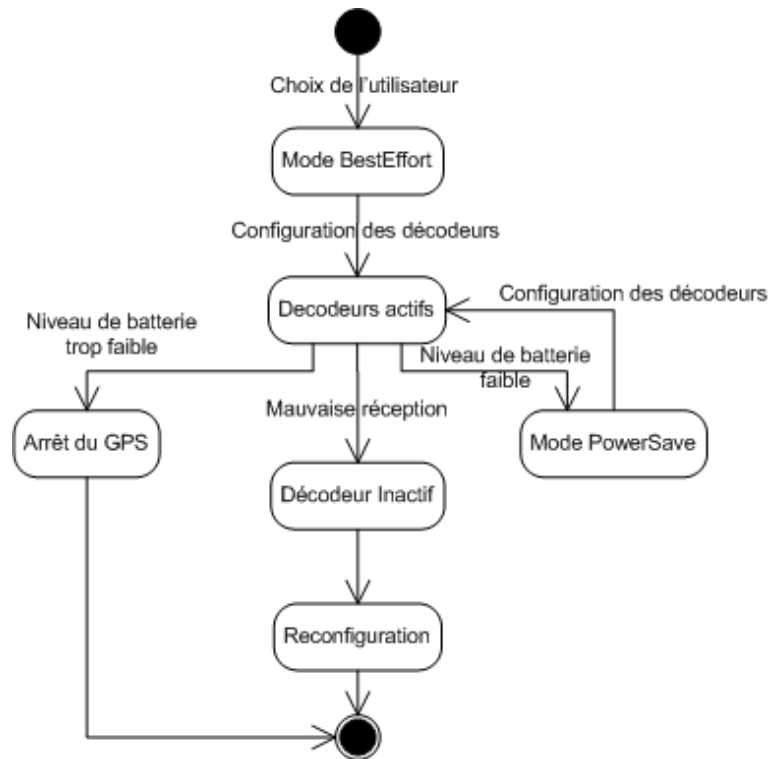


FIG. 4.6 – Exemple de comportement adaptatif pour le contrat *ModeGPS*.

plus forte ($7s \leq time \leq 9s$). Sémantiquement, le raffinement du contrat $C1$ correspond à :

$$C2 \Rightarrow C1 \quad (4.3)$$

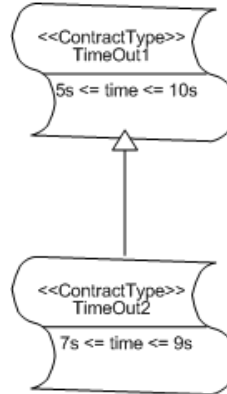


FIG. 4.7 – Exemple de raffinement de contrat.

4.5 Exemple du GPS

Le diagramme 4.8 représente le modèle QoSCL appliqué au composant *Computer* du GPS.

Ce composant offre des services via l'interface *ComputerI*. Il est donc composé d'un port de QdS *PortComputer* qui est lui-même associé à une interface *ComputerI* et à deux *ContractType* : *TimeOut* et *Mode*.

L'interface *ComputerI* est donc contractualisée. Elle est composée de deux opérations *SwitchOn* et *getPosition*. Le *ContractType TimeOut* est associé à l'opération *getPosition* alors que *Mode* se porte sur l'opération *SwitchOn*.

Le contrat *ModeC* est une implantation du *ContractType Mode*. Ce contrat est composé de deux contrats *ReliabilityC* et *PowerSupplyC*.

Les contrats *ReliabilityC* et *PowerC* sont les implantations respectives des *ContractType Reliability* et *PowerSupply*.

La figure 4.9 présente les dépendances de composants et de contrats au niveau du composant *Computer* :

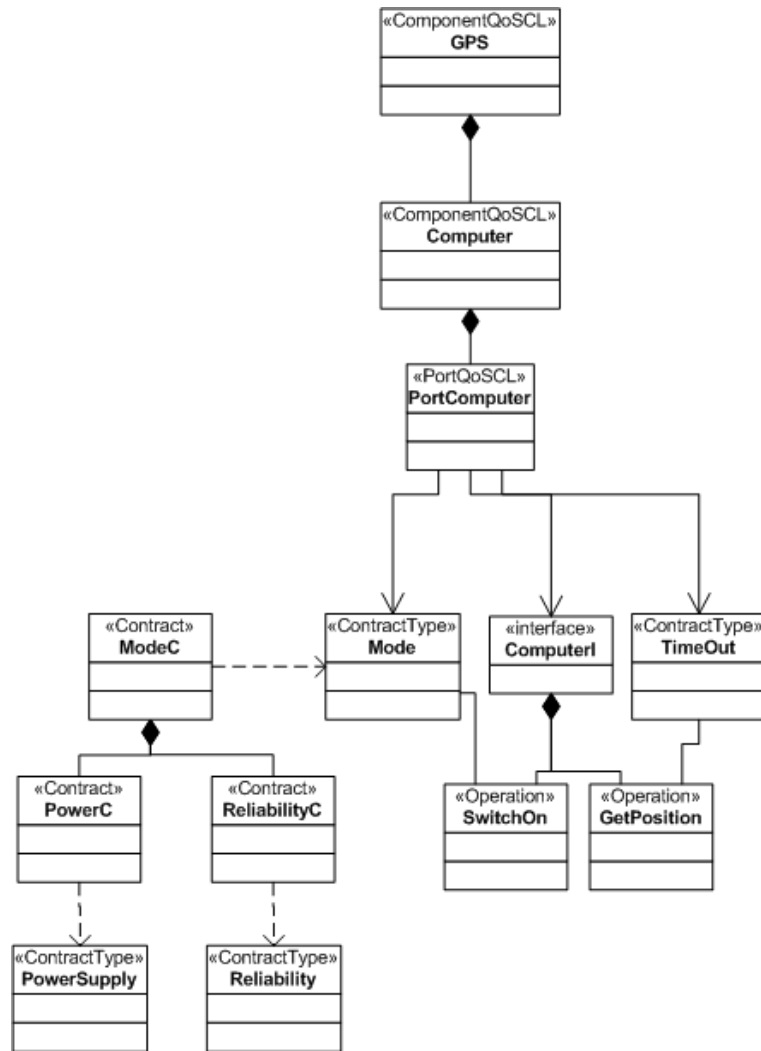


FIG. 4.8 – Le modèle QoSCL appliqué au composant *Computer* du GPS.

Le composant *Computer* pour fournir son service *SwitchOn* avec le contrat *ModeC* a besoin de deux services contractualisés qui sont fournis par les composant *Manager* et *Decoder*. *Manager* fournit le service *getLevel* avec le contrat *PowerC* et *Decoder* fournit le service *getData* avec le contrat *ReliabilityC*.

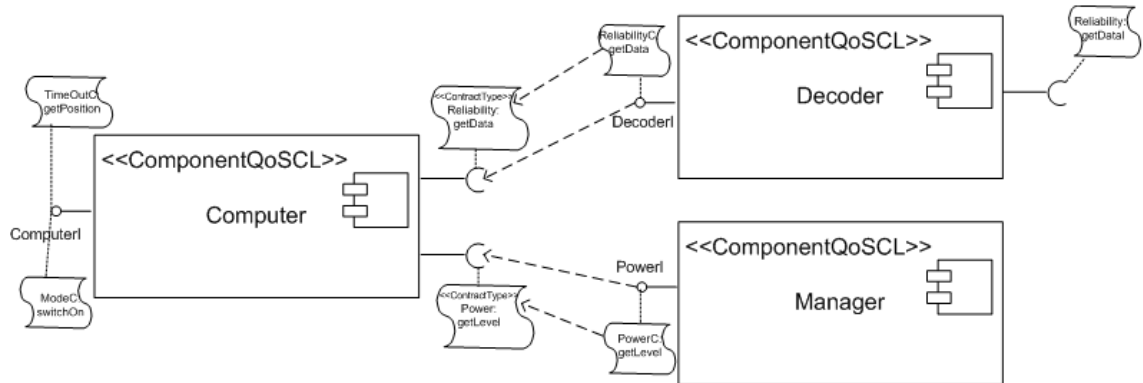


FIG. 4.9 – Les dépendances de contrat au niveau du composant *Computer*

4.6 La qualité de service de bout en bout

Le langage QoSCL permet de définir des propriétés extra-fonctionnelles sur les composants. Le concepteur d'application, une fois ces composants créés, doit les assembler pour construire son application. Il doit donc pouvoir estimer, au moment de la conception, les propriétés extra-fonctionnelles de chaque composant afin de respecter leur spécification. La partie la plus délicate dans la conception d'une application est de sélectionner et de configurer les composants afin d'obtenir le comportement fonctionnel et qualitatif global voulu par le concepteur. Réciproquement, les applications étant construites à partir de composants préconfigurés, le concepteur d'application a besoin de fournir une spécification raisonnable du comportement de l'application globale. Aujourd'hui, dans le modèle de contrat QoSCL, les valeurs des propriétés non fonctionnelles requises ou fournies sont données sous forme de contraintes associées au contrat ou plus précisément à la dimension. Une représentation de cette valeur, au niveau du composant, est donnée dans la figure 4.10.

Pour connaître, *a priori* le niveau de la qualité globale de son application, le concepteur doit propager (voir partie III §4.7 page 92), *à la main*, les différentes valeurs des propriétés non fonctionnelles. Cette étape est faisable, si :

- il y a peu de propriétés extra-fonctionnelles,
- leur niveaux de propagation est faible,
- et leur relation est décrite simplement.

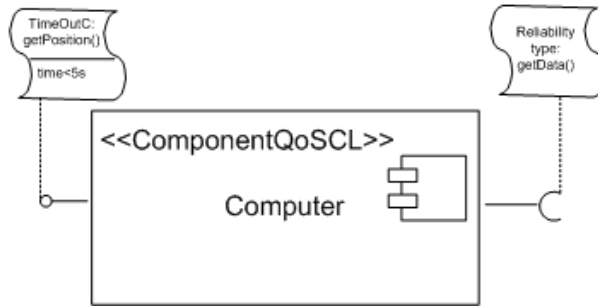


FIG. 4.10 – Représentation de la valeur du timer.

Sinon ce calcul devient vite impossible à gérer. La prochaine version de QoSCL porte donc sur la création d'un outil permettant la prédiction automatique du niveau de qualité global d'une application composite. La section IV (partie III page 128) présente l'approche qui sera utilisée afin de répondre au mieux à ces besoins.

4.7 Calcul de la qualité de service de bout en bout

QoSCL permet la spécification des contrats de QoS et de leurs dépendances. Ces modèles vont être utilisés par des *tisseurs d'aspect* (voir partie III § 5.3 page 99) afin d'intégrer les évaluations contractuelles ainsi que leurs renégociations éventuelles dans des composants. Cependant, au moment de la conception, il est possible de prévoir le niveau de la qualité globale du logiciel composé. Les objets manipulés par l'outil de prédiction sont les propriétés non fonctionnelles. Les dépendances qui lient ces propriétés sont souvent décrites sous forme de formules ou de règles. Pour prédire la qualité de service d'une application composite, il faut propager ses propriétés non fonctionnelles à travers les dépendances.

Reprenons l'exemple du contrat *TimeOutC* :

Chaque récepteur actif reçoit un signal émit par un satellite. La durée totale du signal est de 15 secondes. Le temps pris pour recevoir les données, noté Θ_S au niveau d'un récepteur, se situe donc entre 15 et 30 secondes :

$$\Theta_S \in [15; 30] \quad (4.4)$$

Chaque récepteur va démultiplexer le signal, pour en extraire les données. Cette opération dure environ 2 secondes. De plus, les signaux démultiplexés vont être transformés en vecteur de données simples. Cette opération dure 3 secondes. Soit Θ_R , le temps écoulé pour le service *getData()* et Θ_D , le temps passé pour acquérir les données au niveau du décodeur. Nous obtenons les équations suivantes :

$$\Theta_R = \Theta_S + 2, \quad (4.5)$$

$$\Theta_D = \Theta_R + 3. \quad (4.6)$$

A partir de ces 3 formules, on peut déduire la validité du domaine de Θ_R ainsi que de Θ_D :

$$\Theta_R \in [17, 32] \quad (4.7)$$

$$\Theta_D \in [20, 35] \quad (4.8)$$

Notons **nbr** le nombre de récepteurs actifs c'est-à-dire le nombre de récepteurs recevant des données d'un satellite. A priori, ce nombre n'est pas connu, car il dépend de l'environnement extérieur. A partir de **nbr** et du temps consommé par chaque décodeur pour l'extraction des données Θ_D , on va pouvoir déterminer le temps consommé pour le service *getPosition()*, noté Θ_C :

$$\Theta_C = \max(\Theta_D) + \mathbf{nbr} * \log(\mathbf{nbr}) \quad (4.9)$$

L'expression $\mathbf{nbr} * \log(\mathbf{nbr})$ correspond au temps utilisé par le composant *Computer* pour interpoler la position à partir des différentes données reçues.

Dans notre GPS, le nombre de récepteurs actifs est soit 3, 5 ou 12. Ce nombre influe sur le mode de fonctionnement du GPS :

$$\mathbf{nbr} \in \{3; 5; 12\} \quad (4.10)$$

$$\mathbf{nbr} = 3 \Leftrightarrow mode = PowerSave$$

$$\mathbf{nbr} = 5 \Leftrightarrow mode = BestTrack$$

$$\mathbf{nbr} = 12 \Leftrightarrow mode = BestEffort$$

Le composant *Battery* fournit l'énergie nécessaire au fonctionnement du GPS *GetPower()*. L'énergie active, notée **P**, correspond à la moyenne de la puissance fournie sur une période. Le calcul de la position courante a un coût en énergie qui dépend du nombre de récepteurs actifs **nbr** :

$$\mathbf{P} = \mathbf{nbr} * 3 \quad (4.11)$$

Les valeurs possibles de **nbr** sont connus 4.10, on peut donc aisément en déduire le domaine de validité de **P** :

$$\mathbf{P} \in \{9; 15; 36\} \quad (4.12)$$

Et enfin, nous pouvons déterminer le domaine de validité de Θ_C 4.9 puisque nous avons déterminé les domaines de validité de Θ_D 4.8 et de **nbr** 4.10. Ce qui nous donne :

$$\Theta_C \in [21, 5; 48] \quad (4.13)$$

La qualité du service *getPosition()* peut aussi se décliner sous la forme de la précision estimée, notée **epe**, de la position courante. Cette précision peut prendre 3 valeurs :

- high \sim 1m
- medium \sim 5m
- low \sim 30m

Cette estimation de la précision dépend du nombre de récepteurs actifs **nbr** ainsi que de la durée totale du service Θ_C :

- pour **nbr**=3 :
 - si $\Theta_C < 25$ s alors **epe**= high ;
 - si $\Theta_C > 25$ s alors **epe**= low ;
- pour **nbr**=5 :
 - si $\Theta_C < 24$ s alors **epe**= high ;
 - si $24 \text{ s} < \Theta_C < 30$ s alors **epe**= medium ;
 - si $\Theta_C > 30$ s alors **epe**= low ;
- pour **nbr**=12 :
 - si $\Theta_C < 32$ s alors **epe**= high ;
 - si $32 \text{ s} < \Theta_C < 45$ s alors **epe**= medium ;
 - si $\Theta_C > 45$ s alors **epe**= low ;

La règle permettant le calcul de cette estimation, notée **R**, est représentée dans le diagramme 4.11.

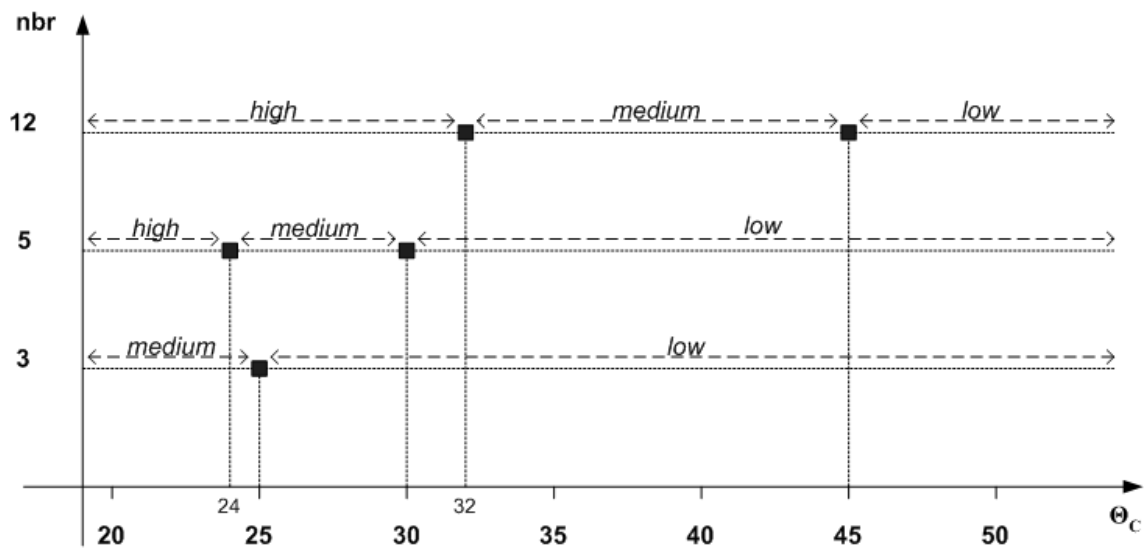


FIG. 4.11 – La règle de calcul de **epe**.

Ce diagramme montre que quelque soit la valeur de **nbr**, **epe** a le même compor-

tement face au temps : la précision augmente quand le temps écoulé diminue. En effet, un temps court pour le calcul de la position courante signifie que toutes les données ont été reçues dans le même intervalle de temps, et donc que l'interpolation espace-temps est plus précise. Au contraire, si les signaux ont été émis dans un intervalle de temps plus long, l'interpolation sera moins précise.

Supposons que l'utilisateur impose que le service doit être réalisé avec une moyenne pour la puissance fournie \mathbf{P} de moins de 150mW et que la précision doit avoir comme valeur *high*, alors la propagation des intervalles induit les résultats suivants pour les propriétés extra-fonctionnelles :

- $\mathbf{nbr}=5$ (d'après la figure 4.11 et l'équation 4.11).
- $\Theta_C \in [23, 5; 24]$ (d'après la figure 4.11 et l'équation 4.9).
- $\Theta_D \in [20; 20, 5]$ (d'après l'équation 4.9).
- $\Theta_R \in [17; 17, 5]$ (d'après l'équation 4.6).
- $\Theta_S \in [15; 15, 5]$ (d'après l'équation 4.5).

Pour satisfaire les demandes du client, à savoir une puissance moyenne inférieure ou égale à 15W et une bonne précision c'est-à-dire *high*, il faut 5 récepteurs actifs et que les signaux soient reçus en moins de 15,5 secondes.

4.8 Conclusion

Dans ce chapitre, nous avons exposé le modèle de contrat QoSCL qui permet de spécifier le raffinement, la dépendance, la délégation et le comportement adaptatif des contrats. La propagation de la qualité de service s'effectue *à la main* si l'application le permet c'est-à-dire si il y a peu de propriétés extra-fonctionnelles avec un niveau de propagation faible et les relations entre ces propriétés sont décrites simplement. Dès que l'application devient conséquente, la propagation *à la main* s'avère être impossible. La création d'un outil permettant la prédiction automatique du niveau de qualité global d'une application composite fait l'objet de la prochaine version de QoSCL.

Lors de l'assemblage de composants, le concepteur d'application doit s'assurer de la conformité entre les services fournis et requis que ce soit au niveau fonctionnel ou extra-fonctionnel.

Le chapitre suivant porte sur les transformations de modèles. Dans le cadre du projet européen QCCS, des outils sont mis en place pour passer de la spécification du composant indépendant d'une technologie à la réalisation de ce composant dans la plateforme d'exécution visée. Nous allons, dans un premier temps, présenter la méthodologie QoSCL puis nous nous intéresserons au tissage d'un aspect contractuel dans un composant. Puis nous présenterons la transformation de modèle utilisée dans QCCS avec l'outil *Kase* et cette même transformation écrite en MTL (Modeling Transformation Language). Enfin nous terminerons par une description des différentes applications qui ont utilisé la méthodologie QoSCL.

Chapitre 5

Du modèle à l'implantation

Contents

5.1	Introduction	97
5.2	Principe	98
5.3	Tissage de l'aspect contractuel	99
5.3.1	Les préconditions OCL	100
5.3.2	Le contrat modélisé en UML	104
5.3.3	Aspect contractuel et surveillance de contrats	104
5.3.4	Le tissage de l'aspect contractuel et de la surveillance de contrats	107
5.3.4.1	Transformation de modèles pour l'ajout d'un contrat dans <i>Kase</i>	109
5.3.4.2	Transformation de modèle en MTL (Modeling Transformation Language)	113
5.3.5	Génération de code	116
5.4	Validation de l'approche	117
5.4.1	MobiForo	117
5.4.2	REGI	118
5.4.3	Le système de télé-médecine	120
5.4.4	Résultat de l'évaluation	120

5.1 Introduction

Le modèle de contrat QoSCL permet la spécification des composants avec contrats de QoS dans un modèle indépendant de toute technologie. D'un point de vue industriel, ce modèle n'est utilisable que si des outils sont mis en place permettant d'utiliser ces contrats de QoS avec des technologies particulières. Ces outils permettent de passer de la spécification du composant indépendant d'une technologie à la réalisation de ce composant dans la plate-forme d'exécution visée [35] [46].

5.2 Principe

La figure 5.1 représente les différentes étapes de développement d'un composant tenant compte de la QoS.

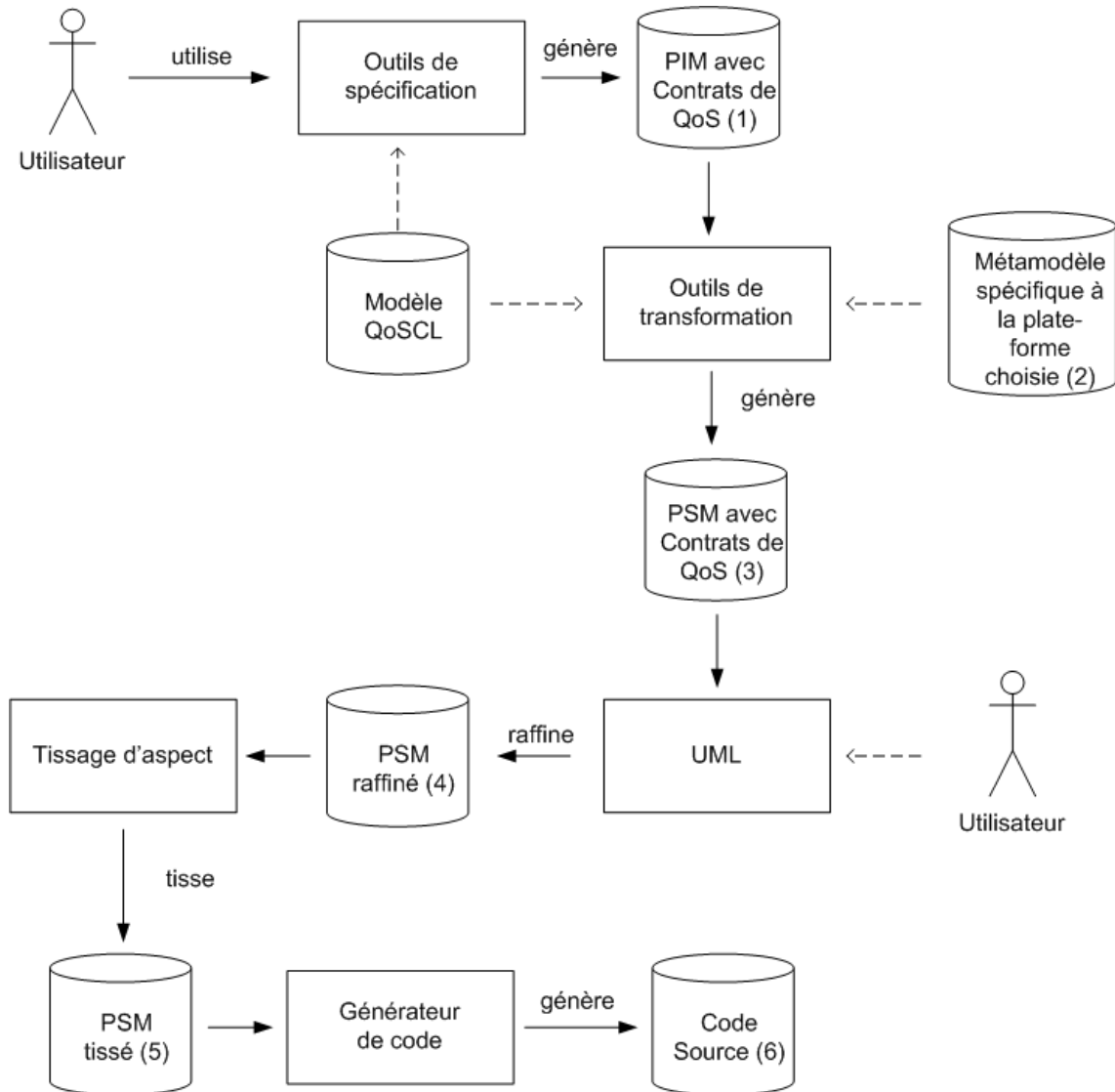


FIG. 5.1 – Les différentes étapes de développement d'un composant de QoS.

Dans un premier temps, l'utilisateur va spécifier le composant de QoS grâce au modèle de contrat QoSCL (présenté partie III §4.3 page 81), il va donc obtenir un PIM (voir partie II §3.2 page 64) avec contrats de QoS (1). Une fois la plate-forme d'exécution choisie, le PIM va être transformé en PSM grâce à des outils de transformations fondés sur le métamodèle de la plate-forme (2) en question. L'outil permettant la transfor-

mation reçoit en entrée un modèle complet selon un métamodèle indépendant de la plate-forme. On a, à la sortie de ce *transformateur*, un modèle conforme au métamodèle spécifique de la plate-forme choisie (3). Ce modèle pourra être raffiné par l'utilisateur si nécessaire (4).

Les aspects fonctionnels et extra-fonctionnels sont séparés dans le modèle QoSCL grâce aux notions de *ContractQoSCL*, *ContractType*, *Dimensions...* Lors de la transformation du PIM vers le PSM, il faut donc transformer les contrats du PIM en classes, méthodes, diagrammes de séquence, etc dans le PSM cible. Or l'outil de transformation de modèle n'a pas à connaître et comprendre la sémantique de chaque contrat. Cela signifierait que pour chaque création de contrat, il faudrait étendre l'outil de transformation de modèle afin d'y intégrer sa sémantique spécifique. De cette manière, l'outil deviendrait vite très lourd et impossible à gérer.

La solution que nous proposons (et qui à été validée dans le projet QCCS) est d'utiliser la méthode du tissage d'aspects : l'outil de transformation de modèle connaît les différents concepts relatifs aux contrats ainsi que la manière de transformer un aspect PIM en un aspect PSM. Cette information est stockée dans un dépôt qui contient les informations pour chaque contrat : son nom et la référence du modèle UML qui contient l'aspect PSM correspondant.

La prochaine étape est donc de transformer le PSM où les aspects sont maintenus séparés en un PSM qui contient tous les éléments nécessaires à la création de l'exécutable (5). Cette étape est décrite dans le paragraphe intitulé *Tissage de l'aspect contractuel* (partie III §5.3 page 99).

Le PSM, ainsi contractualisé, va pouvoir être donné en entrée à un générateur de code pour obtenir son exécutable.

5.3 Tissage de l'aspect contractuel

Dans cette section, nous allons détailler le mécanisme de tissage de l'aspect contractuel dans un PSM, utilisé dans le projet européen QCCS. Le contrat se présente sous forme de *Package* dans lequel nous allons trouver toutes les informations nécessaires à son déploiement. Pour illustrer cette section, nous prenons comme exemple le contrat *TimeOut* du GPS présenté dans la partie II §2.3 page 46.

Le *package* d'un contrat est composé des éléments suivants :

- un fichier contenant les préconditions nécessaires au modèle pour l'intégration du contrat,
- un package contenant les éléments modélisés en UML qui vont être injectés au modèle de composant à surveiller,

- un *aspect* qui permet de spécifier le comportement du contrat,
- et le code de la transformation qui intègre le contrat au modèle initial.

5.3.1 Les préconditions OCL

Pour contractualiser un composant, celui doit respecter certaines préconditions. Ces préconditions décrites déterminent les contraintes du modèle. Dans le cas du contrat *TimeOutC*, l'utilisateur doit sélectionner l'opération sur laquelle il désire ajouter un contrat.

La figure 5.2 représente un composant décrit avec l'outil *Kase* sur lequel on peut appliquer la transformation.

Les préconditions pour l'intégration du contrat *TimeOutC* dans le modèle sont :

- La sélection est bien une opération *op* ;
- *op* appartient à une classe *cl* ;
- *cl* implante une interface *I* ;
- *op* est définie dans *I* ;
- *cl* doit être une composition d'un composant *comp* ;
- *comp* hérite de `system.ComponentModel.component` (SCMC en abrégé) de la librairie *Csharp* ;
- Nous travaillons avec l'outil de modélisation *Kase* or dans *Kase* un composant au niveau PSM .NET est représenté par un package constitué d'une classe ayant le même nom. On a donc comme précondition supplémentaire : *comp* doit être inclus dans un package du même nom (voir figure 5.3).

Dans *Kase*, l'extension *Csharp* n'est pas visible sur le modèle mais est spécifiée dans les propriétés de l'attribut.

Voici ces préconditions exprimées en OCL pour le contrat *TimeOutC* :

listSelection est la sélection (sous forme de séquences) d'éléments sur lequel on veut appliquer le contrat.

```
Let listSelection : sequence=getSelection()
```

La sélection ne doit comporter qu'un seul élément.

```
listSelection->size()==1
```

Cet élément doit être de type *behavior* :

en UML2.0, le *behavior* est le comportement d'une opération c'est-à-dire la représentation de l'implantation.

```
beh=listSelection->first()
beh.type.oclIsKindOf(Behavior)
```

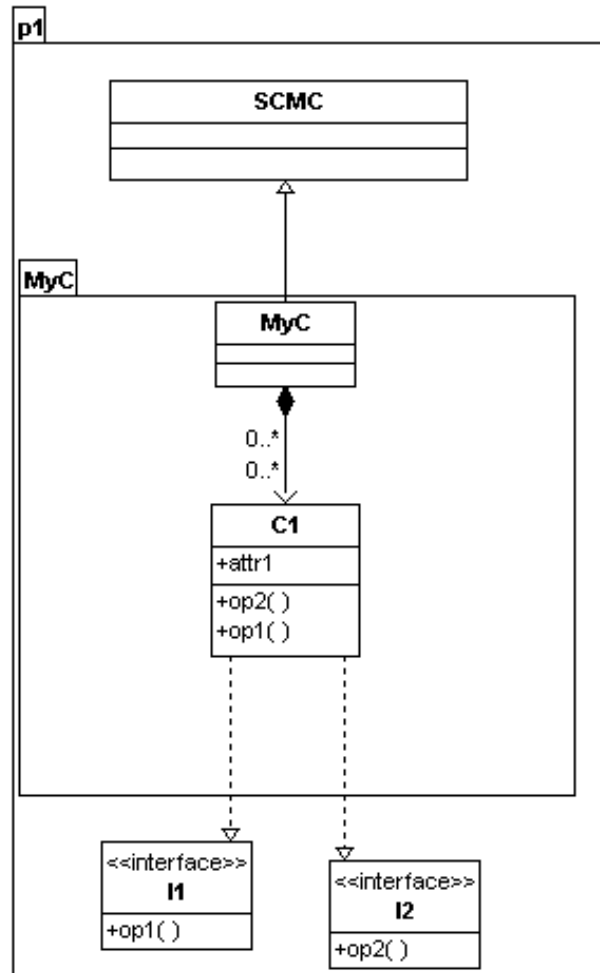


FIG. 5.2 – Un composant pour lequel l'intégration du contrat *TimeOutC* est possible.

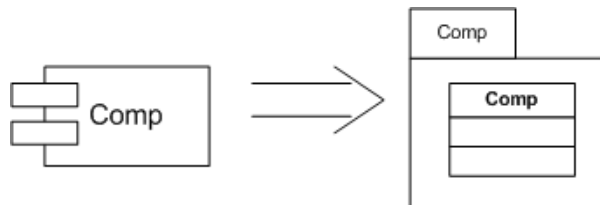


FIG. 5.3 – Notre représentation de composant .Net dans Kase.

Classif est le *classifier* qui contient *beh* :

```
Classif=beh.context
```

classif doit être de type *class* :

```
classif.type.ocIsKindOf(Class)
```

op est l'opération qui correspond à *beh* :

```
op=beh.specification
```

op doit être de type *Operation* :

```
op.type.ocIsKindOf(Operation)
```

listClassifier est la liste des classifieurs qui contiennent *op* :

```
listClassifier=op.featuringClassifier()
```

Cette liste doit avoir au moins 2 éléments : la classe implantée et l'interface.

```
listClassifier->size()>2
```

listInterface est la sous-liste ne contenant que les interfaces :

```
listInterface=listClassifier
-> select( c~: c.type.ocIsKindOf(Interface))
```

listInterface ne doit contenir qu'un seul élément :

```
listinterface->size()=1
```

Une fois l'environnement de l'opération vérifié, on passe aux contraintes au niveau composant. Dans un PSM .NET, un composant est représenté par un package avec le nom du composant. Dans ce package on a une classe qui a le même nom que le package et qui hérite de la classe *System.ComponentModel.component*.

listAssoEnd est la liste des *associationEnd* de la classe *classif* :

```
listAssoEnd=classif.reference
```

Pour chaque *associationEnd*, on regarde si l'association qui lui est associé est une agrégation si c'est le cas on l'ajoute a *listAggreg* :

```
listAggreg=listAssoEnd
    ->select(a| a.association.oclIsKindOf(Aggregation))
```

On n'autorise dans ce modèle qu'une seule agrégation :

```
listAggreg.size()=1
```

class1 est le classifieur associé à *classif* par une agrégation :

```
aggreg=listAggreg.first()
class1=aggreg.aggregEnd.referencingClassifier
```

class1 doit être de type *class* :

```
class1.oclIsKindOf(Class)
```

pack est le namespace qui englobe *class1* :

```
pack=class1.namespace
```

pack doit être un *package* :

```
pack.oclIsKindOf(Package)
```

Le nom du *package* doit être le même que le nom de *class1* :

```
class1.name=pack.name
```

Et enfin, *class1* hérite de la class *system.ComponentModel.Component* :

```
c=class.generalization.general
c.name="system.ComponentModel.Component"
```

5.3.2 Le contrat modélisé en UML

Le contrat modélisé en UML est décrit dans un package qui regroupe les différents éléments à ajouter aux modèles c'est-à-dire les classes nécessaires à la mise en place du contrat.

Le contrat *TimeOutC*, par exemple, a besoin de la classe *Timer* pour s'exécuter correctement. La figure 5.4 représente le package à ajouter au modèle pour l'intégration du contrat *TimeOutC*.

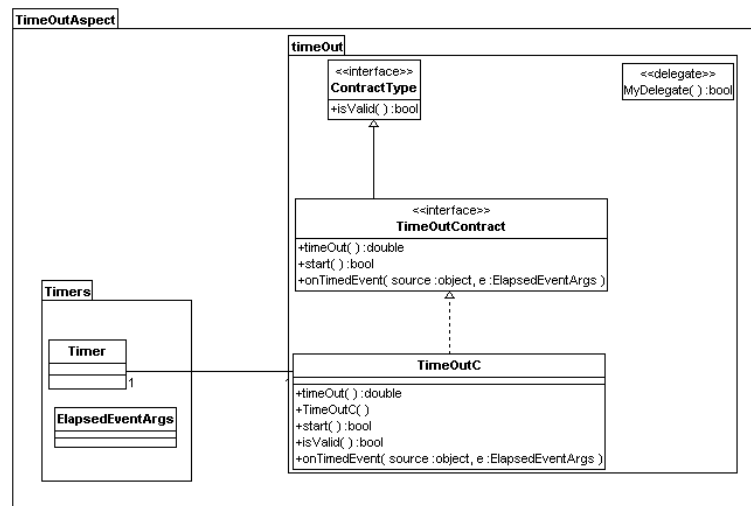


FIG. 5.4 – Le package contenant les différents éléments nécessaires au contrat *TimeOutC*.

5.3.3 Aspect contractuel et surveillance de contrats

Notre technique sépare la conception des modèles d'application implantant les contrats extra-fonctionnels de l'implantation des moniteurs de contrat. La conception par contrat exige la surveillance de ce contrat. Il faut donc ajouter cette partie lors de la construction de l'application. Cette intégration peut dans la théorie être effectuée à plusieurs endroits :

- au niveau du binaire (*bytecode*),
- à la compilation c'est-à-dire au niveau du source,
- ou alors, au niveau du modèle.

Dans tous les cas, la partie application et la partie moniteur doivent se coordonner le plus efficacement possible. Au niveau du binaire, l'adaptabilité dynamique offre une performance honorable mais en deçà des performances obtenues pour l'intégration au

niveau du source [61]. Si l'intégration est effectuée au niveau du source, alors la technique la plus appropriée est la programmation par aspects (AOP [43]). Les langages de programmation par aspects (par exemple AspectJ [67]) permettent la composition sophistiquée des pièces du source, en utilisant des règles de synchronisation. Cependant, si le tissage est effectué au niveau de code source seulement, il est difficile de l'intégrer au niveau de la modélisation UML. Puisque la surveillance des contrats extra-fonctionnels a un impact sur le comportement de l'application (par exemple pour la renégociation ou pour la reconfiguration), le mécanisme de surveillance doit être visible au niveau de la conception de l'application et donc dans les modèles de conception UML. En effet, nous défendons l'idée que le contrôle de surveillance des contrats doit être laissé au concepteur d'applications, pour plusieurs raisons :

- le coût de surveillance doit s'équilibrer avec la sévérité de la violation de contrat. Le choix de surveiller la précision doit donc être laissé au concepteur d'applications. Il devra configurer le contrat et manipuler les événements de détection de violation ;
Reprenons l'exemple du GPS, la gestion de l'énergie peut être envisagée de différentes manières. Le GPS peut être relié à une batterie ou être sur secteur. Selon les cas, la surveillance du niveau de l'énergie sera différente. La précision sera plus importante dans le cas d'une batterie : on pourra avoir plusieurs niveaux d'énergie (*high*, *low*, *none*) alors que dans le cas du secteur, il faudra «juste »vérifier qu'il y a de l'énergie.
Pour notre GPS, nous utilisons 3 degrés de précision pour le niveau d'énergie (*high*, *low*, *none*). Dans certains cas, ces degrés de précision ne sont pas suffisants et l'ajout d'un quatrième degré (*medium*) s'avère nécessaire.
- selon le type d'architecture de déploiement, on pourra utiliser différents algorithmes.

Les contraintes de transformation incluent des propriétés spécifiques du composant qui doivent être surveillées. Ces propriétés incluent les contraintes de typage, l'existence d'opérations spécifiques sur des types spécifiques, ou encore l'existence d'associations spécifiques entre les éléments du composant. Le concepteur peut exprimer ces contraintes en utilisant une extension du metamodèle d'UML. Cette extension est basée sur le concept de *template* qui existe déjà dans la notation. Nous l'avons étendu afin d'exprimer les propriétés comportementales du moniteur, c'est-à-dire comment il doit agir avec les entités du composant à surveiller. L'aspect pour l'insertion du contrat *TimeOutC* est présenté figure 5.5. Un aspect est composé de plusieurs éléments :

- Un nom (avec le stéréotype UML *aspect*).
- Une liste de paramètres formels, lesquels seront liés aux objets existants du modèle lors du tissage de l'aspect.
- Un ensemble d'éléments qui représentent les contraintes exprimées graphique-

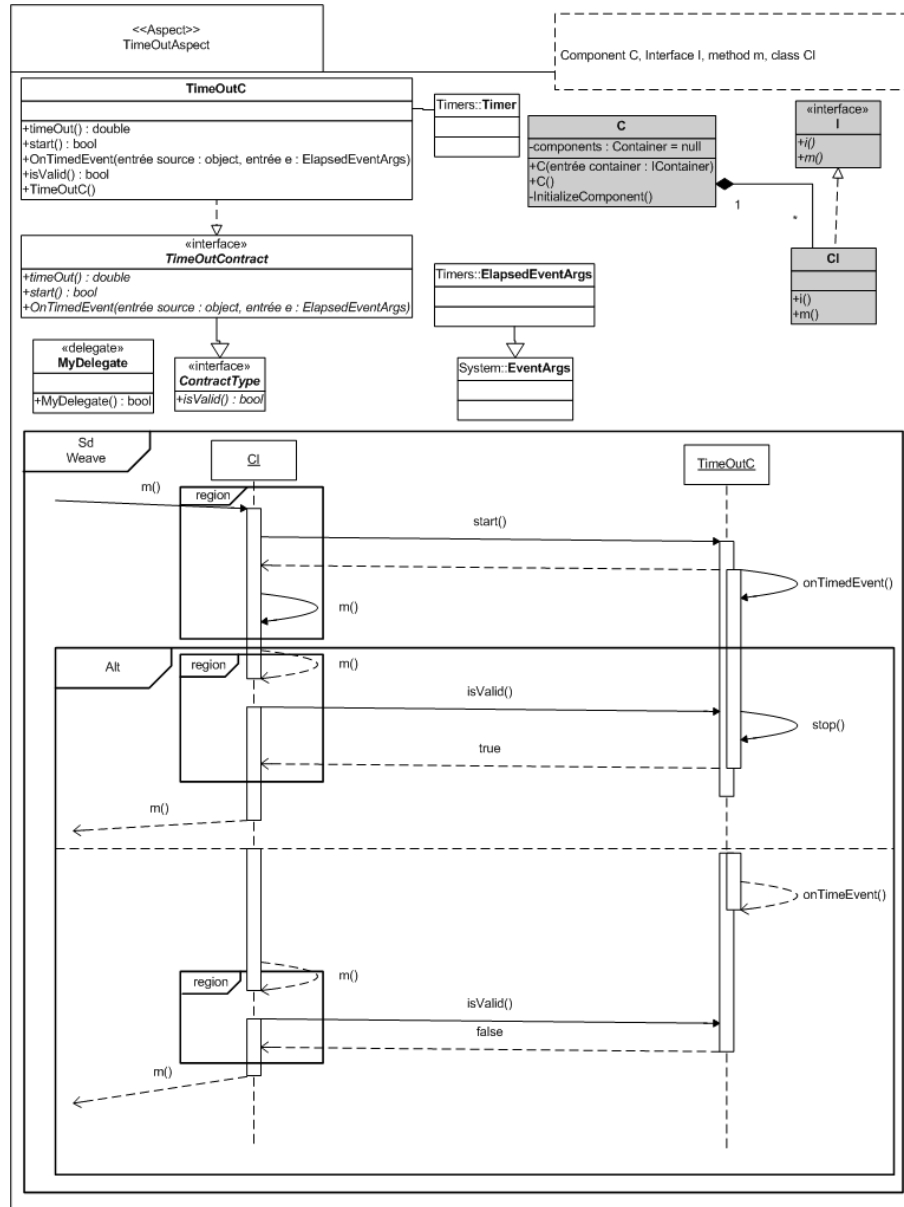


FIG. 5.5 – L'aspect pour l'insertion du contrat *TimeOutC*

ment. Dans l'exemple du contrat *TimeOutC*, les contraintes représentées sont :

- la classe *c1* doit être une implantation de l'interface *i1*,
 - l'opération *op* doit être implémenté par la classe *c1*,
 - *c1* doit faire partie du composant à surveiller
 - et le package du modèle référencé doit inclure la classe de *TimeOutC*.
- un *high-level message sequence chart* ou *HMSC* [30] [53] qui est la nouvelle forme des diagrammes de séquences d'UML2 adaptée à la description de scénarios. Il établit comment les éléments ajoutés interagissent avec les éléments du modèle environnant. Cette forme de spécifications graphiques est définie par une norme de l'ITU (International Telecommunications Union). Un MSC est un diagramme fini qui décrit les relations de cause à effet entre émissions et réceptions de messages constitutives d'un scénario. Les MSCs sont munis d'une opération de concaténation par mise bout à bout de diagrammes, compatible avec la concaténation des séquences sur chaque site. Un HMSC est un automate fini sur un alphabet dont les lettres sont des MSCs. Le langage d'un HMSC est l'ensemble des extensions linéaires des MSCs obtenus en interprétant les chemins de cet automate par concaténation des lettres.

L'utilisation d'un HMSC permet d'améliorer la compréhension des différentes interactions entre les éléments ajoutés par le tisseur et les éléments existants dans le modèle. Il détermine les contraintes de synchronisation entre le modèle surveillé et les éléments de surveillance et fournit les moyens de déclarer les comportements alternatifs et itératifs.

Le HMSC de la figure 5.5 déclare que le message demandant le début de la surveillance *start* doit être envoyé au moniteur au début de l'exécution de *m()*. Le contrat lance le *timer*. Si le *timer* s'arrête avant la méthode *m()* alors le contrat sera valide sinon il sera obsolète. Ces deux possibilités sont représentées dans la partie *Alt* du diagramme. La région correspond à un ordre total des événements le long de chaque ligne de vie des objets contenues dans cette région [31].

5.3.4 Le tissage de l'aspect contractuel et de la surveillance de contrats

Chaque modèle de moniteur de contrat est lié à un script qui effectue le tissage dont l'exécution change la structure du modèle UML en entrée. Et avant de s'exécuter, le script vérifie que les contraintes du modèle en entrée sont satisfaites.

Chaque moniteur de contrat a son propre code de transformation de modèle. Dans le cas de notre exemple du contrat *TimeOutC*, le code de transformation vérifie d'abord les préconditions mentionnées graphiquement dans l'aspect ou encore exprimé en OCL

(voir §5.3.1 page 100). Ensuite le code de la transformation insère les éléments appropriés du modèle de moniteur (présenté dans le §5.3.2 page 104 et dans la figure 5.4) dans le modèle du composant (prolongation du modèle statique). Puis dans la dernière phase, le code de tissage change le comportement de l'application surveillée pour synchroniser le moniteur de contrat (présenté dans le HMSC de la figure 5.5) (prolongation du modèle dynamique). Ce changement est effectué en éditant les diagrammes d'état du composant. La figure 5.6 représente un exemple de modèle après l'ajout d'un contrat *TimeOutC* sur l'opération *op1*.

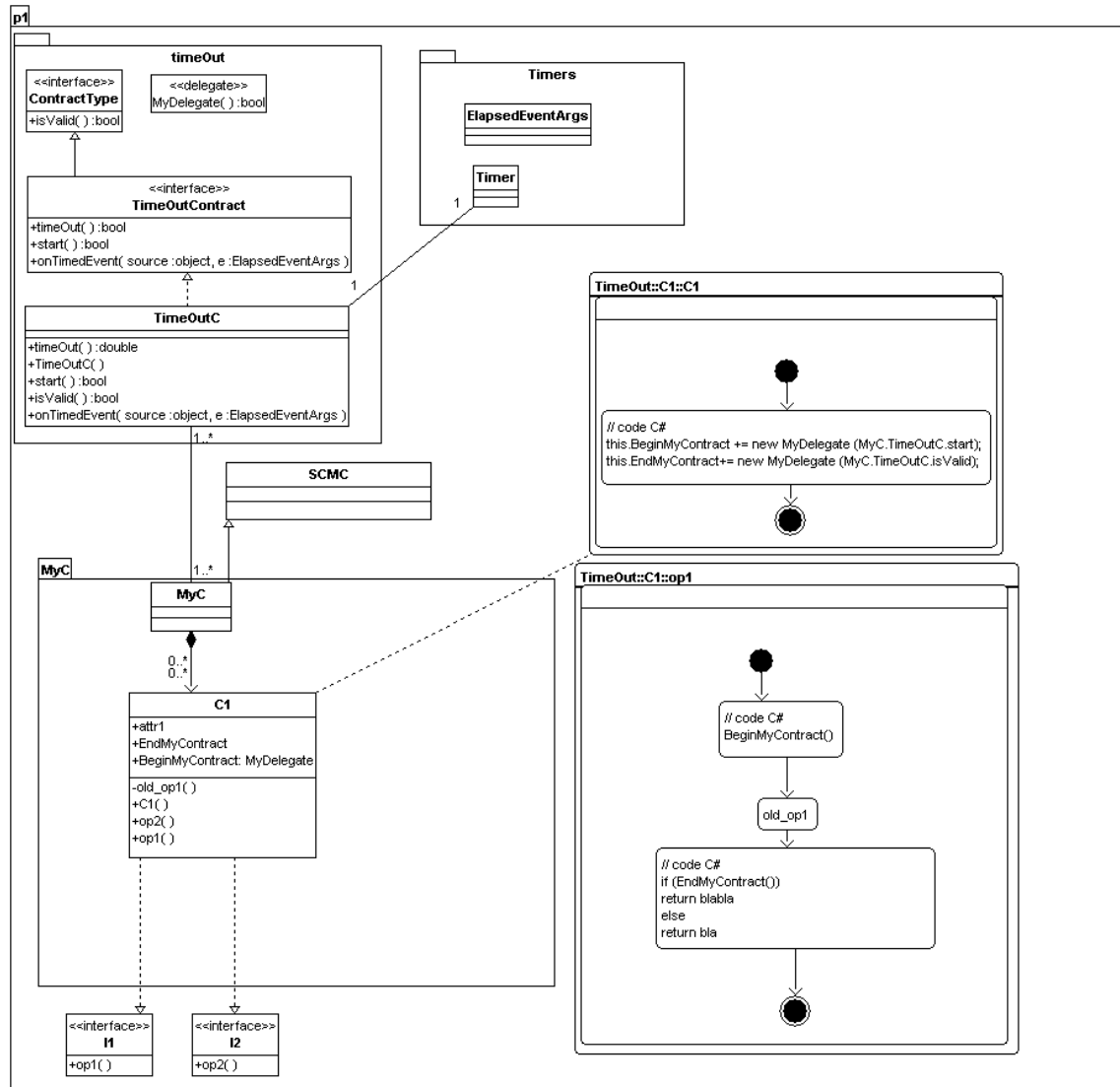


FIG. 5.6 – Application du contrat *TimeOutC* sur l'opération *op1*

Cet exemple est une vue simplifiée du résultat de la transformation qui ajoute le

contrat *TimeOutC* à un modèle dans le cadre du projet QCCS. Ce qui signifie que ce diagramme correspond à un PSM .Net qui a été réalisé avec l'outil *Kase*. Cette transformation a été écrite avec le langage Python. Dans le chapitre suivant (voir partie III §5.3.4.1 page 109), nous détaillons cet exemple avec la mise en place de cette transformation en Python dans *Kase*.

5.3.4.1 Transformation de modèles pour l'ajout d'un contrat dans *Kase*

En général, un script de transformation consiste à modifier un modèle UML de manière automatique. Le transformateur prend donc en entrée un modèle UML, et renvoie en sortie un autre modèle UML. *Kase* manipule des modèles UML, mais la transformation elle-même est définie en python [14]. Ce qui rend *Kase* plus flexible, car il est possible de définir de nouveaux transformateurs sans recompiler *Kase*. Dans cette section, nous décrivons les différents étapes de la transformation pour l'exemple du GPS. Cette transformation est exécutée dans le modèle du GPS décrit à l'aide de l'outil *Kase*. Le script de la transformation est donné en annexe C.

La première étape est de vérifier les préconditions pour le tissage du contrat *TimeOutC* sur le composant. Ces contraintes ont été listées dans le §5.3.1 page 100. Pour notre exemple, nous allons appliquer le contrat *TimeOutC* sur l'opération *getPosition()* fourni par le composant *LocationComputer*. La figure 5.7 représente le composant *LocationComputer* décrit à l'aide de l'outil *Kase*.

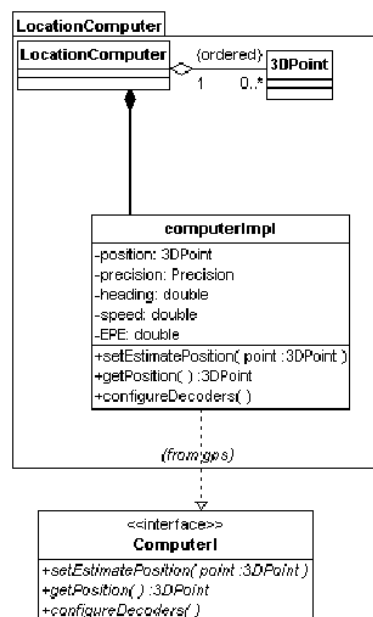


FIG. 5.7 – Composant *LocationComputer* sur lequel on va ajouter le contrat *TimeOutC*.

Une fois toutes ces préconditions vérifiées, on peut appliquer le contrat *TimeOutC* sur l'opération *getPosition()* :

- Le package *timeOut* peut avoir été importé lors de l'ajout du contrat *TimeOutC* sur un autre composant du modèle ou sur le même composant mais sur un service différent. Il faut donc tout d'abord tester si celui-ci est déjà intégré au modèle, sinon il faut l'importer :

```
# verify if the package timeOutAspect is still in the package
# if not, add it
listPack=parentPack.getAllModelElements("timeOut", "Package")
if len(listPack)==0 :
    timeOut.TimeOutAspect().transform(parentPack)
```

La fonction *transform* ajoute le package *timeOut* dans le modèle *parentPack*. La figure 5.8 montre le composant à transformer auquel on a ajouté le package *TimeOut*.

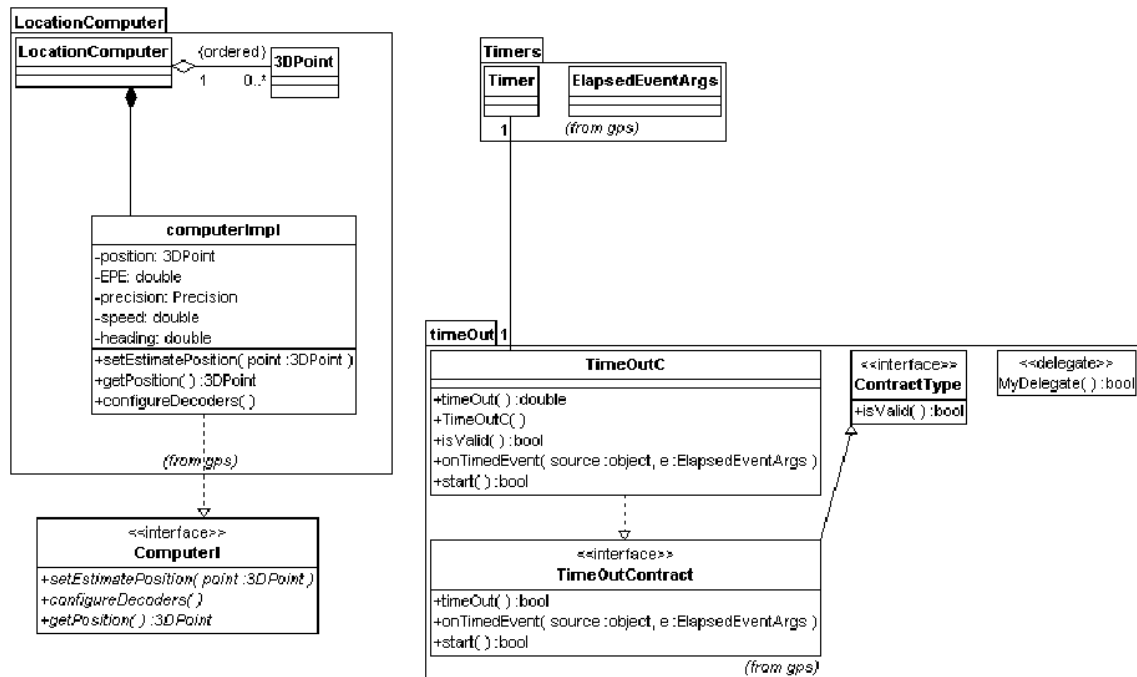


FIG. 5.8 – Ajout des packages *Timers* et *TimeOut*.

- Ensuite, il faut créer une association entre le composant à transformer et le contrat :

```
#en python if not (comp.isAssociate(contract))~:
    comp.addAssociation(contract,
        multiplicity1=[(1,"*")],multiplicity2=[(1,"*")])
```

Le contrat va être géré par deux *delegates* : l'un active le contrat, et l'autre

l'arrête. Il faut donc les déclarer en tant qu'attributs dans la classe qui implante le service à contractualiser.

```
# search for an Attribute named 'BeginMyContract'

#and if not found add it listAttribute=class1.getAllModelElements
    ("BeginMyContract", "Attribute")
alist = [] for f in listAttribute:
    if f.toAttribute().type()==MyDelegate:
        alist.append(f)
        long =len(alist)
if long > 1 :
    showMessage( "Error : multiple MyDelegate
        BeginMyContract!",["Dismiss"],
        "QCCS TimeOutContract", 0, -1, Warning )
    return
elif long == 0 :
    # add BeginMyContract
    class1.addAttribute(name="BeginMyContract",type=MyDelegate)
```

La figure 5.9 est la représentation *Kase* du composant *LocationComputer* auquel on a associé le contrat *TimeOutC* et ajouté deux *delegates* qui vont permettre la supervision dû dit-contrat.

- Une fois le composant lié au contrat *TimeOutC* et les deux *delegates* créés, il faut ajouter au *delegate beginMyContract* l'activation du contrat (*contract.start()*) et au *delegate endMyContract* l'arrêt du contrat (*contract.isValid()*). Cette association va être faite au moment de la création de la classe c'est-à-dire dans le (ou les) constructeurs de la classe. Ce qui donne en *Csharp* :

```
this.BeginMyContract += new MyDelegate(contract.start);
this.EndMyContract += new MyDelegate(contract.isValid);
```

Cet ajout de code est représenté sous forme de *composite state* dans la transformation. Si le constructeur existe déjà on le renomme en *old* suivi du nom du constructeur et on met sa visibilité en privé. On crée un nouveau constructeur auquel on associe un *compositeState* contenant les informations et le code à ajouter.

- La dernière étape consiste à armer et à désarmer le *timer* à l'aide des *delegate* dans l'opération en question *ConfigureDecoders*. il faut donc renommer l'opération en *old* suivi du nom de l'opération puis mettre sa visibilité en privé. On crée une nouvelle opération à laquelle on associe un *compositeState* contenant les informations et le code à ajouter :

```
#rename op and creation of new operation
    oldName="old_"+method.name()
    listOp=class1.getModelElements(oldName,"Operation")
```

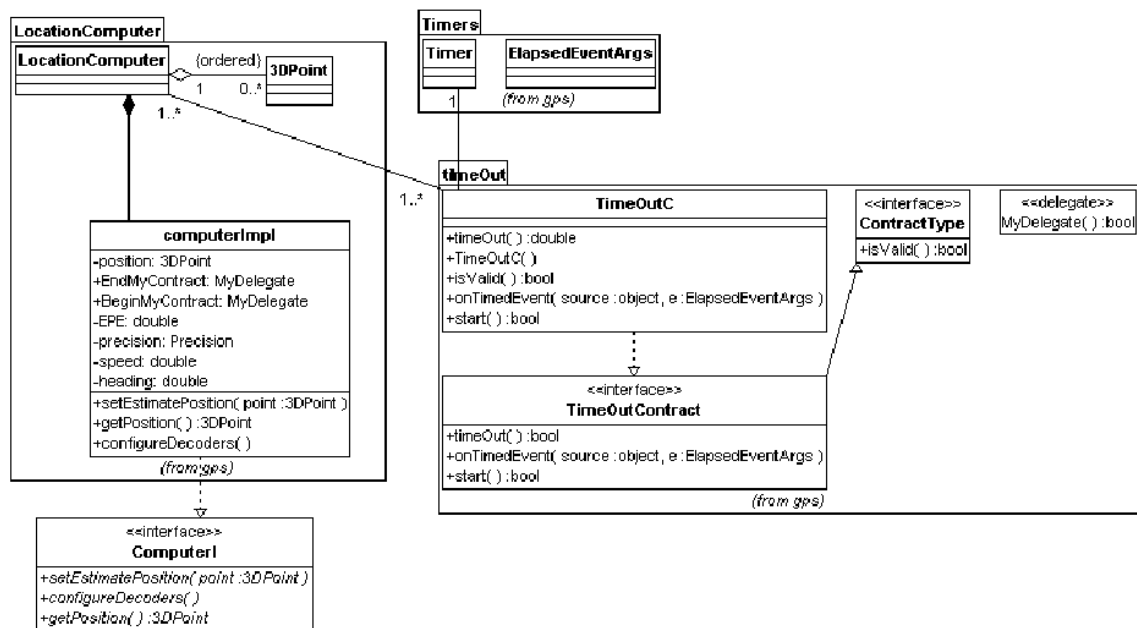


FIG. 5.9 – Ajout d’une association entre le composant et le contrat et des différents *delegates*.

```

alist=[]
for op in listOp :
    op=op.toOperation()
    if not(op.isNull()):
        if op.hasSameParameters(method):
            alist.append(op)
if len(alist)==0:
    method.setVisibility("private")
    opName=method.name()
    method.setName(oldName)
    newOp=method
    method=class1.addNewOperation(name=opName)
else :
    newOp=alist[0]
    listbeh=op.behavior()
    for b in listbeh:
        listOp[0].appendBehavior(b)

```

La transformation d'un aspect contractuel peut, évidemment, s'écrire dans d'autres langages à partir du moment où ces langages de transformation connaissent le métamodèle utilisé. L'outil *Kase* est fondé sur UML1.x auquel on a ajouté les notions de composants d'UML2.0 afin de pouvoir appliquer la méthodologie QoSCL. Les transformations de modèle dans *Kase* sont donc limités à des modèles basés sur le métamodèle connu par l'outil. MTL (Modeling Transformation Language)[66] [16] a été créé afin de pallier ce problème de transformation de modèles. Nous avons donc écrit nos différentes transformations en MTL (voir section suivante §5.3.4.2 page 113). Au moment de l'écriture de ces transformations en *python*, MTL n'était pas encore exploitable.

5.3.4.2 Transformation de modèle en MTL (Modeling Transformation Language)

MTL [66] est un langage créé par l'équipe *Triskell* de l'*IRISA* (Institut de Recherche en Informatique et systèmes aléatoires) permettant l'écriture de programmes de transformations de modèles. Grâce à ce langage, les transformations peuvent être décrites dans un métamodèle pivot. Une fois écrite, cette transformation sera mappée dans le métamodèle désiré : les transformations sont donc portables. Les objets MTL et les éléments du modèle sont manipulés de la même façon. De plus, MTL permet les transformations de transformations de modèles.

MTL se connecte à un depositaire de modèles afin de naviguer dans un modèle présent dans ce depositaire et éventuellement le modifier. Il s'utilise quelque soit le depositaire grâce à une API générique.

MTL vérifie la cohérence d'un modèle vis-à-vis de son métamodèle et permet de transformer un modèle dans un autre modèle même si ceux-ci sont basés sur des métamodèles différents.

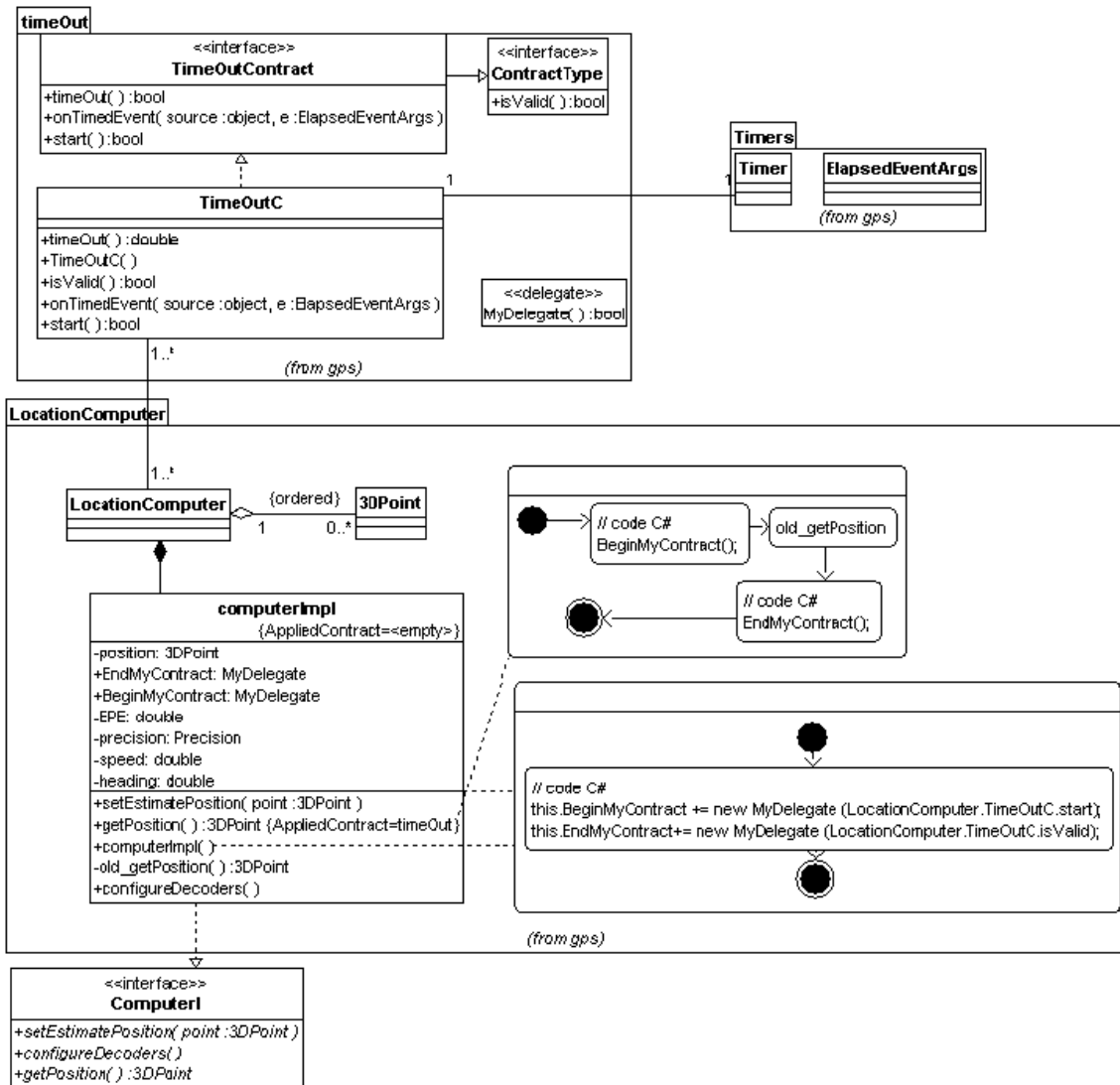


FIG. 5.10 – Création et renommage des opérations et mise en place des *CompositeState*.

Une partie du code de la transformation en MTL pour l'ajout du contrat *TimeOutC* est donné en annexe B.

Il faut tout d'abord spécifier les bibliothèques utilisées et préciser dans quels métamodèles sont décrits les différents modèles manipulés :

```
library AspectTimeOutC;

model source_model : RepositoryModel; // a Repository model
(UML1.4) model contract_model: RepositoryModel; // a Repository
model (UML1.4) model uml          : UML_Utils;
```

Une fois que MTL est connecté au dépositaire de modèle adéquat, la transformation peut commencer. On va tout d'abord rechercher les opérations sur lesquelles le contrat va être appliqué. Ces opérations sont marquées d'une *TaggedValue* appelé *selected*.

Le code ci-dessous recherche dans l'ensemble des opérations du modèle toutes les opérations ayant une *TaggedValue selected*. Ces opérations sont regroupées dans la variable *myOperationSet* qui est déclaré comme un ensemble.

```
operationSet := !source_model::Core::Operation!.allInstances();
JavaCode [BMTL_myOperationSet=new BMTLSet()];
myIterator := operationSet.getNewIterator();
while myIterator.isOn()
{

    myOperation := myIterator.item().oclAsType
        (!source_model::Core::Operation!);
    if uml.IsModelElementTaggedValue (myOperation, 'selected').[=] (true)
    {
        myOperationSet := myOperationSet.including(myOperation);
    }
    myIterator.next();
}
```

Ensuite, on va appliquer sur chaque élément de cet ensemble, la méthode *verification* qui va vérifier les préconditions sur l'opération :

```
myIterator := myOperationSet.getNewIterator();
while myIterator.isOn()
{
    myOperation := myIterator.item().oclAsType
        (!source_model::Core::Operation!);
    verification(myOperation);
    myIterator.next();
}
}
```

5.3.5 Génération de code

Une fois le PSM contractualisé, le code peut être généré.

Kase permet la génération automatique de code à partir d'un modèle UML. Après la transformation d'un modèle, on peut générer le code *Csharp* associé. Pour chaque élément du modèle, le générateur détermine le code associé. L'annexe D présente le code généré automatiquement par *Kase* à partir du modèle présenté dans la figure 5.10.

Ce générateur a dû être modifié pour permettre la génération de code sur un modèle ayant subi un ajout de contrat puisqu'il ne gérait pas les éléments tels que les diagrammes d'états. Voici une partie du code qui permet la génération d'un *CompositeState* :

```
def codeForCompositeState( cs, indent, tab ) :
    if ( not cs.isCompositeState() ):
        showMessage( "Code generation is only possible for
            composite state", ["Dismiss"], "C# code generator",
            0, -1, Warning )
        raise "dontshow"

    buffer = "// code of composite state \n"
    for s in cs.subvertex():
        s=s.toPseudoState()
        if not(s.isNull()):
            if s.kind()=="Initial":
                initState=s
    nonstop=0

    while nonstop==0:
        listTr=initState.outgoing()
        if len(listTr)>1:
            showMessage( "Code generation is only possible
                for simple composite state", ["Dismiss"],
                "C# code generator", 0, -1, Warning )
            raise "dontshow"
        elif len(listTr)==1:
            state=listTr[0].target()
            if state.isFinalState():
                buffer+=indent + "// end of code of
                    composite state"+" \n"
                nonstop=1
            elif state.isCompositeState():
                buffer+=indent + codeForCompositeState
                    (state.toCompositeState(), indent, tab)
```

```
        initState=state
        listTr=[]
    elif state.isActionState():
        buffer+=codeForActionState
            (state.toActionState(),indent,tab)
        initState=state
        listTr=[]
    else:
        initState=state
        listTr=[]

    elif len(listTr)==0: nonstop=1
return buffer
```

5.4 Validation de l'approche

La méthodologie QoSCL s'inscrit dans le projet européen QCCS dont l'approche a été validée par trois cas d'études :

1. La conception d'un composant nommé *MobiForo* qui permet d'offrir plus de mobilité aux utilisateurs du système de *workflow FORO* de SchlumbergerSema [6].
2. Une application internet *REGI*[23], créé par *KD Software* qui fournit les informations sur les services qui sont offerts dans une certaine région.
3. Un système de télé-médecine [60] de l'université de Chypre, qui permet de surveiller la prise de tous les médicaments au moment opportun par les patients.

5.4.1 MobiForo

Le système de *workflow FORO* a une architecture client-serveur strict. Le client n'a pas les moyens d'accomplir sa tâche correctement si la liaison réseau est indisponible. Cette indisponibilité peut se produire en raison d'une perturbation soudaine dans le réseau ou le serveur, ou parce que l'utilisateur travaille dans un endroit où le raccordement au serveur est mauvais voire inexistant. Avec une architecture différente et plus de fonctionnalités du côté du client, les participants du *workflow* pourraient continuer à travailler (*off-line*) quelque soit l'endroit et la disponibilité du serveur et se synchroniser au serveur quand le réseau est disponible.

SchlumbergerSema voulait réutiliser au maximum le code de FORO qui est écrit en Java et C++ avec CORBA. Or pour des raisons spécifiques à QCCS, les outils utilisés fonctionnent seulement dans le cadre de .NET. La solution choisie par SchlumbergerSema est de créer des nouveaux composants pour le système de *workflow FORO*. C'est sur ces composants que sera appliquée la méthodologie QCCS. Nous allons maintenant décrire trois contrats qui ont été mis en place avec la méthodologie QCCS :

1. Le premier est spécifique au *workflow* : toutes les tâches doivent être terminées avant leur deadline.

```
type WorkitemDeadline = contract {
    deadline: increasing numeric sec; // time to the deadline
}
```

2. Les mises à jour entre le client et le serveur dépendent de l'environnement de travail du client. Si le serveur reçoit un appel d'un poste de travail relié en permanence au serveur et un appel d'un téléphone mobile : il va d'abord exécuter les requêtes du téléphone mobile. Il est également important de connaître les protocoles utilisés pour la transmission de données ou le type de base de données que le client utilise afin d'optimiser la manière les réponses à toute la file d'attente des demandes.

```
type ClientHardware = contract {
    kindOfClient : increasing enum
        {server, workstation, laptop, palm, mobilphone} with
        order {server > workstation, workstation > laptop,
        laptop > palm, palm > mobilphone}
    protocolOfClient: increasing numeric Kbps;
    // the interesting feature of the protocol
    // is the throughput , so we decide to specify directly.
    BBDDofClient: increasing numeric queries/s;
    // as in the previous dimension the
    // interesting feature is the amount of queries the
    //Database can process per second.
} // contract to select the response preference
```

3. Le contrat spécifie les préférences et les besoins du système *MobiForo*. Dans ce contrat, on s'intéresse au nombre de requêtes par seconde au niveau du serveur, le temps moyen entre les échecs acceptables, et au temps de réponse de la requête.

```
type ServerPerf = contract {
    // number of requests serviced per second
    throughput: increasing numeric req/s;
    // typical mean time between failures
    mtbf: increasing numeric hours;
    // execution time of a service request
    serviceDuration: decreasing numeric msecs;
}
```

5.4.2 REGI

REGI est une application Internet qui fournit des informations sur les services offerts par une région. L'objectif de *KD SOFTWARE*, est de faire un comparatif entre les nouvelles méthodes de travail (méthodologie QCCS et ces outils) et les méthodes

utilisées avant QCCS (c'est-à-dire sans modélisation logicielle). La finalité n'étant pas de créer une application complète mais juste une partie de l'application qui servira de support pour cette étude.

Le but du projet *REGI* est de développer une plate-forme logicielle internet qui permet aux fournisseurs de service régionaux d'offrir leurs services par l'intermédiaire du portail *REGI*. Ce portail fournit à ses utilisateurs un accès vers différents services de la plate-forme *REGI*.

REGI doit fournir à ses utilisateurs les informations et les services les plus appropriés à leurs besoins. Le système doit connaître la région par laquelle l'utilisateur accède à l'application. REGI doit alors présenter les informations utiles à l'utilisateur et supprimer les services qui n'ont pas d'intérêt pour sa requête.

Il y a trois modes d'utilisation du système :

1. Définition et contrôle des services,
2. exploration du catalogue de services, recherche de services.
3. achat en ligne de services.

Pour cette application, deux contrats ont été mis en place : un contrat portant sur la persistance et un autre portant sur la disponibilité.

1. Un contrat de type persistance spécifie des propriétés pour la gestion des données persistantes. Il permet de spécifier :
 - Le lieu de stockage et le stockage : toutes les informations, telles que le profil et le compte de l'utilisateur, tous les produits et services offerts, sont stockées dans une base de données.
 - Le comportement transactionnel : la persistance des données est sécurisée par transaction sur des opérations atomiques.

Ce contrat possède un attribut *IsMandatory* qui prend pour valeur *True* ce qui signifie que le contrat ne peut être négocié et qu'il ne peut être stoppé. La persistance ne peut être effective si le temps de réponse pour une action est expiré. Il faut donc surveiller le temps de réponse avec la dimension *duration* du contrat de persistance.

2. Le deuxième contrat porte sur la disponibilité c'est-à-dire le nombre de fois où l'application est disponible pour l'utilisation. Plus spécifiquement la disponibilité est le pourcentage de fois où l'application est réellement disponible aux requêtes de services comparé au temps d'exécution disponible prévu. Le calcul de cette disponibilité inclut le temps de réparation de l'application puisque quand une application est en cours de réparation, elle ne peut être utilisée :

$$\text{Availability} = (\text{MTBF} / (\text{MTBF} + \text{MTTR})) \times 100$$

avec MTBF^{\sim} : Temps moyen d'exécution de l'application avant échec.

MTTR^{\sim} : Temps moyen nécessaire pour réparer et restaurer le service après échec.

5.4.3 Le système de télé-médecine

Le système de télé-médecine [60] permet de surveiller la prise de tous les médicaments au moment opportun par les patient. Toutes les données sont stockées au centre de contrôle, ces données sont accessibles via le réseau par des unités mobiles. Le personnel médical pourra ainsi changer les ordonnances et vérifier la prise de médicament de chaque patient à partir des postes fixes ou mobiles mais ce système offre aussi la possibilité d'assister à des téléconférence. C'est un système réparti et quasi temps réel, il comprend donc des unités autonomes qui se connectent via le réseau. Toutes ces unités travaillent ensemble, peuvent coordonner leur activités et échanger des messages. Ces unités peuvent être regroupées en quatre catégories :

- les unités distantes ont la capacité de communiquer avec certains dispositifs.
- Le centre de contrôle est le coeur du système dans lequel les unités distantes et les unités administratives sont connectées. Ces unités peuvent envoyer et recevoir des messages au travers de ces connections.
- Les unités administratives : les messages de ces unités correspondent à un changement de requêtes et permettent la mise à jour des unités distantes et des copies qui sont stockées dans d'autres unités.
- Les administrateurs mobiles.

Toutes les composants de l'application communiquent entre eux. La QdS ajoutée à ces composants permet de calculer entre autres :

- Le niveau de sécurité c'est-à-dire la capacité à résister aux intrusions.
- Le niveau d'adaptabilité mesure la capacité du composant à tolérer les changements au niveau des ressources et des requêtes de l'utilisateur.
- Le niveau de portabilité qui correspond à sa capacité à migrer vers un nouvel environnement.
- Le nombre d'images par seconde.
- Le délai d'exécution maximal sur les opérations.
- Ou encore, la facilité d'un composant a évolué sur une période donnée.

Une quinzaine de «mesures»ont été ainsi déterminées. Combinées entre elles, ces dimensions déterminent les différents contrats utilisés par le système de télé-médecine.

5.4.4 Résultat de l'évaluation

L'évaluation de l'approche a été effectuée par trois utilisateurs dont le domaine et les connaissances étaient totalement différentes. Les utilisateurs de SchlumbergerSema pour le contrat QCCS n'avait pas d'expérience en modélisation et en conception de systèmes à composants. L'acquisition de la notion de propriétés extra-fonctionnelles a été difficile à maîtriser. De plus, le code initial de l'application devait rester au maximum inchangé. La solution choisie par SchlumbergerSema (créer des nouveaux composants pour le système de *workflow FORO*) a permis de garder le code intact mais a limité le choix des contrats de QdS. Chez KD Software, les développeurs ont l'expérience des systèmes à base de composants, mais la modélisation et la notion de propriétés extra-fonctionnelles leur étaient inconnus. Leur cahier des charges étaient précis et leurs

contrats ont été rapidement définis. Malgré le temps consacré à l'apprentissage des nouvelles techniques et méthodes, leur objectif pour le projet QCCS a été atteint à savoir faire un comparatif entre les nouvelles méthodes de travail (méthodologie QCCS et ces outils) et les méthodes utilisées avant QCCS (c'est-à-dire sans modélisation logicielle). Pour l'université de Chypre, les outils et méthodes utilisés leur étaient familiers avant le projet QCCS, l'ajout de contrats de QdS dans leur application s'est fait aisément. Globalement la principale difficultés rencontrée par les utilisateurs a été la compréhension des nouveaux concepts tels que les notions de PIM et de PSM, les transformations de modèles, le tissage d'aspects etc.

Les résultats de cette évaluation pour chacun des participants est donnée dans [63] :

- Les conclusions de SchlumbergerSema sont positives, la méthodologie QCCS leur a apporté de nouvelles techniques et de nouveaux outils puisque les utilisateurs de SchlumbergerSema pour le contrat QCCS n'avait pas d'expérience en modélisation et en conception de systèmes à composants. L'utilisation d'OCL pour les pré et post conditions a été particulièrement appréciée ce qui leur permettait de rendre leurs cas d'utilisation plus formels. La notion de contrat est simple à comprendre mais difficile à assigner aux bons ports. L'approche MDA et l'idée d'inclure les modèles de conceptions dans les scripts de transformations leur a plu puisqu'ils déclarent être très heureux des résultats obtenus avec QCCS et vont continuer à employer *Kase* et l'approche MDA.
- Les concepteurs de logiciels de KD Software sont habitués à coder directement leurs applications et n'avaient pas d'expérience en modélisation. La phase de conception était fait manuellement avec des outils inadaptés tel que *Microsoft Word*. Ils leur fallait donc des outils faciles à utiliser et qui permettent de décrire de façon claire et précise la phase de conception afin d'améliorer la qualité logicielle de leurs produits. Ils ont constaté que la méthodologie de QCCS est une bonne manière d'atteindre ces objectifs. La méthodologie de QCCS rend le procédé de développement plus rapide et efficace, puisque le code de base est directement lié aux diagrammes. Les avantages de la méthodologie de QCCS sont :
 - L'adoption d'un processus de développement logiciel avancé qui permet de réaliser une bonne conception orientée objet avec des facteurs de qualité.
 - Amélioration de la communication et collaboration entre les développeurs.
 - Le temps économisé par développeur. Le bénéfice au niveau de la productivité pour la production globale de code est entre 18 et 20%
 En conclusion, la méthodologie QCCS est robuste et répond à presque tous leurs besoins.
- L'université de Chypre avait une application qui était fondée sur les composants, totalement opérationnelle et avec beaucoup de contrôles concernant la validité de la connectivité. Cependant, cette application manquait de contrôles importants au niveau de la QdS. La méthodologie QCCS a permis d'ajouter de la QdS à cette application. En outre, en appliquant la méthodologie, le code de QdS est

réutilisable pour d'autres applications. Il est facile de définir et d'identifier des contrats dans le code et de les exécuter. La méthodologie QCCS est robuste. Elle est facile à employer et est accessible. En conclusion, les concepteurs du système de télé-médecine pensent que la méthodologie peut être facilement utilisée dans le secteur commercial, avec des résultats remarquables.

Les résultats de cette évaluation sont plutôt positives. Néanmoins, les utilisateurs se sont heurtés à des difficultés avec tout d'abord l'apprentissage de nouveaux concepts qui selon le cas étaient plus ou moins nombreux. Le tissage d'aspects a été la notion la plus compliquée à assimiler avec la séparation des aspects fonctionnels et extra-fonctionnels. L'apprentissage de la méthodologie a donc pris plus de temps que prévu. Le choix des contrats a du être adapté afin de faciliter la compréhension de ses nouvelles notions. Dans ces conditions, la partie test n'est pas complète notamment sur les aspects de dépendances de contrats. Cette évaluation est une première étape, il faudrait l'approfondir avec des études de cas plus conséquentes pour évaluer la complexité globale de la méthodologie.

Quatrième partie

Conclusion

Rappel de la problématique

Devant la complexité grandissante des systèmes, les développeurs ont recours aux composants logiciels pour améliorer la productivité et la qualité d'un développement logiciel. Ces composants facilitent la réutilisation de code. Dans le cadre de cette thèse, nous nous sommes appuyés sur la définition d'un composant logiciel donnée par Clémens Szypersky dans [64] :

Un composant logiciel est une unité de composition ayant des interfaces contractualisées ainsi que des dépendances contextuelles. Ces dépendances explicitent les interfaces exigées ainsi que les plate-formes d'exécutions possibles. Un composant peut être déployé indépendamment, et être sujet à la composition. De ce point de vue, le composant est une unité exécutable.

Un composant logiciel fournit et requiert des services. Les utilisateurs de logiciel, aujourd'hui, veulent non seulement un service, mais ils veulent l'obtenir dans les meilleures conditions. Le service rendu n'est plus le seul critère de choix, la qualité de service offerte devient un enjeu important. Cette qualité de service est déterminée par les propriétés extra-fonctionnelles du service et ces propriétés sont dépendantes des propriétés extra-fonctionnelles des composants qui constituent le service en question.

Afin d'évaluer la qualité de service de bout en bout de l'application, il faut pouvoir déterminer si les composants constituant le service satisfont le niveau de QdS offert par l'application. Les contrats de QdS spécifient les droits et les obligations entre un client et un fournisseur de service et déterminent les relations entre les acteurs. L'application va pouvoir réagir à son environnement par un mécanisme de renégociation. Le concept de contrat de QdS constitue le concept fondamental de cette thèse.

On constate, aujourd'hui, que les propriétés extra-fonctionnelles sont souvent absentes de la spécification alors que ces propriétés influent le développeur dans le choix d'un composant. Il faut donc prendre en compte ses propriétés extra-fonctionnelles dès la phase de conception. Les travaux qui ont été présentés dans cette thèse ont été effectués dans le cadre du projet européen QCCS. Ce projet a développé une méthodologie et des outils pour la conception et la mise en œuvre de composants distribués et contractualisés. La contribution de cette thèse est de proposer une méthodologie pour la spécification des contrats de QdS de composants logiciels, et leur intégration dans un environnement de développement par composants.

Démarche utilisée

La première partie de ce document introduit les différentes notions qui sont à la base de notre méthodologie : les composants logiciels et les systèmes à base de composants,

les contrats de qualité de service, la notation UML et la méthode MDA.

Dans un premier temps, il a fallu choisir les outils et méthodes utilisés. Au niveau plate-forme d'exécution, la plate-forme .Net répond au mieux aux besoins du projet QCCS grâce à son support de méta-données qui vont permettre de coder les propriétés de QdS dans le composant.

Pour ce qui est de la spécification de la QdS, Jan Oyvind Aagedal a évalué les principales approches existantes. QML en sort comme le langage le plus complet et le plus satisfaisant. Il répond en partie aux besoins de QCCS. Pourtant certains aspects de spécifications ne sont pas traités par QML comme la délégation, la dépendance et le comportement adaptatif. De ce fait, le calcul de la QdS de bout en bout d'un système est impossible en QML.

Dans un premier temps, QML a été étendu afin d'y intégrer la notion de dépendance de contrats. Mais nos principales motivations dans l'élaboration de notre méthodologie étaient de concevoir des contrats à partir d'un outil de modélisation UML et d'hériter naturellement des mécanismes existants tels que par exemple la conformance des types. L'extension de QML a permis de *capturer* les notions importantes sur lesquelles est fondée notre méthodologie.

Notre modèle de contrat repose donc sur UML2.0 et est inspiré du langage QML. Il permet de spécifier des composants avec contrats de QdS dans un modèle indépendant de toute technologie. D'un point de vue industriel, ce modèle est utilisable si les contrats de QdS sont manipulables avec des technologies particulières. La mise en place d'outil permettant passer de la spécification du composant indépendant d'une technologie à la réalisation de ce composant dans la plate-forme d'exécution visée fait l'objet de la deuxième partie de cette thèse. Pour créer un outil permettant la génération automatique des composants contractualisés, la solution proposée est d'utiliser la méthode du tissage d'aspects qui nous a permis au niveau conceptuel, de lever la séparation entre les aspects fonctionnels et non fonctionnels et d'introduire les mécanismes de surveillance et de renégociation.

Apports de la thèse

Notre modèle de contrat permet de concevoir des composants contractualisés au niveau conceptuel à partir d'un outil de modélisation UML. Les aspects de spécification tels que la dépendance, la délégation, le raffinage et le comportement adaptatif sont facilement exprimables et permettent le calcul de la QdS de bout en bout d'un système. Grâce au comportement adaptatif des contrats, la renégociation s'exprime aisément. Tous les éléments de notre modèle héritent naturellement des mécanismes existants. On peut citer, par exemple, la conformance des types entre un *ContractType* et le *Contract* qui lui est associé.

À partir de ce modèle de contrat, une méthodologie a été élaborée afin de permettre au concepteur d'applications de spécifier des composants contractualisés technologiquement indépendant et ensuite de réaliser ce composant dans la plate-forme d'exécution visée. C'est à ce niveau que la surveillance des contrats est mise en place car la surveillance des contrats extra-fonctionnels a un impact sur le comportement de l'application (par exemple pour la renégociation ou pour la reconfiguration). Le mécanisme de surveillance doit être visible au niveau de la conception de l'application et donc dans les modèles de conception UML. La méthode utilisée pour la mise en place de cette surveillance au niveau conceptuel est le tissage d'aspects. Le tissage d'aspects, au niveau PSM, permet de :

- passer d'un modèle QoSCL où les aspects extra-fonctionnels et fonctionnels sont séparés à un modèle sans séparation de ces aspects ;
- mettre en place les mécanismes de surveillance et de renégociation.

Dans le cadre du projet européen, cette méthodologie est utilisée avec l'outil de modélisation *Kase* (donc des scripts de transformation en python) et la plate-forme .Net mais elle peut s'appliquer avec d'autres outils et d'autres plate-formes. Des tests ont été réalisés avec l'outil de modélisation *Poséidon* et les transformations en MTL et on pourrait très envisager de changer de plate-forme d'exécution.

Perspectives

Notre modèle de contrat a été élaboré à partir de la version UML 2.0 v0.671. Depuis UML2.0 a évolué et des changements ont été apporté sur la partie composant. Une prochaine étape sera donc la prise en compte de ces changements dans notre modèle. Celui-ci risque d'ailleurs d'être simplifié puisque dans la version actuelle d'UML2.0 (qui n'est pas encore finalisée), les notions de ports ont été enlevées. Les *ContractType* seront alors liés directement au composant QoSCL sans passer par la notion de *PortQoSCL*.

L'approche a été validée par les utilisateurs du projet européen QCCS. Les résultats dans l'ensemble sont positives mais les utilisateurs ont eu des difficultés : beaucoup de notions et de concepts nouveaux tels que le MDA, les transformations de modèles, le tissage d'aspects etc. Face à ses difficultés, les tests prévus initialement n'ont pu être mené à terme. Et, pour certains utilisateurs, le choix des contrats a dû être restreint. De ce fait, la partie test n'est pas complète notamment sur les aspects de dépendances de contrats. De notre côté, l'exemple du GPS que nous avons utilisé est assez complet et intègre les différents aspects de spécification possibles du modèle QoSCL mais cet exemple n'est pas assez consistant pour évaluer la complexité globale de la méthodologie.

Sur la partie tissage d'aspects, l'écriture des scripts pour des contrats simples est relativement facile mais peut devenir vite complexe pour des contrats composés. Des

efforts sont à mener pour aider à l'écriture de ces scripts.

Vers un outil de prédiction

Les propriétés extra-fonctionnelles sont des quantités valables avec des contraintes numériques au niveau de l'interface qui sont liées entre elles au niveau composant. Cette interprétation des propriétés extra-fonctionnelles est commune à la programmation logique de contraintes (CLP) [40].

La programmation logique permet d'induire et de déduire des propriétés à partir de règles et de faits. La CLP est une forme évoluée de la programmation logique auquel ont été ajoutés les domaines de valeurs. Un outil CLP intégré dans un outil de conception permet au concepteur de lier les propriétés extra-fonctionnelles et de propager les informations numériques à travers le diagramme de composant. Le concepteur a ainsi des connaissances *a priori* sur la qualité de service de son composant. Cette information peut par ailleurs être encore enrichie par l'ajout de nouvelles connexions ou de nouveaux contrats ajoutés par le concepteur ce qui permet de visualiser directement l'influence de ses actions sur la qualité globale de son composant.

Les spécifications contractuelles sont écrites en QoSCL. Bien que possédant toutes les caractéristiques pour la CLP, QoSCL n'est pas adapté à ce type d'utilisation. Pour résoudre ce problème, une solution est de transformer les spécifications QoSCL extra-fonctionnelles dans un langage CLP existant, en utilisant un modèle de transformation conforme au MOF tel que MTL [16] (voir partie III chapitre 5.3.4.2 page 113).

En reprenant l'exemple du GPS vu dans le paragraphe *Calcul de la qualité de service de bout en bout* (voir partie III §4.7 page 92), nous donnons en annexe (voir annexe E) un résultat possible de cette transformation dans un programme *PrologIV* [1].

Les prédicats *receiver* (ligne 01), *decoder* (ligne 04), *computer* (ligne 10) déterminent les dépendances des propriétés extra-fonctionnelles définies dans chaque composant. Le prédicat *rule* est utilisé par le prédicat *computer* pour associer les variables *Epe*, *P* et *Nbre*. Les lignes 25 à 27 représentent la requête qui se décompose en deux parties :

- La ligne 25 détermine les contraintes numériques soit induites par l'environnement et la spécification, soit par le concepteur qui veut connaître l'impact en terme de qualité d'une restriction spécifique sur une ou plusieurs propriétés. Ici, les contraintes sont :
 1. $\mathbf{P} \leq 15$,
 2. $\mathbf{epe} = \{high\}$,
 3. $\Theta_S \in [15; 30]$,
 4. $\mathbf{nbr} \in \{3; 5; 12\}$.
- Les lignes 26 et 27 représentent la connexion entre les différents composants, issue du modèle de composant. La variable Θ_D est une propriété partagée entre le *decoder* et le *computer*.

Le résultat de ce programme est conforme au résultat déterminé dans la section *Calcul de la qualité de service de bout en bout* (voir III §4.7 page 92) :

```
ThetaS ~ cc(15, 15.5),  
Nbr = 5,  
ThetaD ~ cc(20, 20.5),  
ThetaC ~ cc(23.5, 24),  
Epe = high,  
P = 15.
```

Grâce à cet outil de prédiction, le développeur pourra estimer la QdS offert pour son application et choisir les composants contractualisés les mieux adaptés pour son application en terme de QdS.

Cinquième partie

Annexes

Annexe A

Grammaire étendue de QML

```
declaration ::= conTypeDecl;
             | conDecl;
             | profileDecl;

profileDecl ::= xp for intName = profileExp

profileExp  ::= profile
             | xp refined by {req1;...; reqn;}

intName    ::= identifier

profile    ::= profile {req1;...; reqn;}

req        ::= require contractList
             | from entityList require contractList

*contractList ::= \textbf{xc1,..., xcn}, contFunc1,...,
                 contFuncn, conExp1,..., conExpn, conExpFunc1,..., conExpFuncn

*contFunc   ::= xc(var1 : yt1,..., varn : ytn )

entityList  ::= entity1,..., entityn

entity      ::= opName
             | attrname
             | opname.parName
             | result of opName

opName     ::= identifier
```

```

attrName      ::= identifieur

parName       ::= identifieur

conTypeDecl   ::= type y = conType

*conDecl      ::= xc = conExp
                |   contFunc = conExpFunc

*conExpFunc   ::= y contractFunc

conExp        ::= y contract
                |   xc refined by {constraint1 ; ... ; constraintk ;}

conType       ::= contract {dimName1 : dimType1 ; ... ; dimNamek
: dimTypek ;}

dimName       ::= n

dimType       ::= dimSort
                |   dimSort unit

dimSort       ::= enum {n1, ..., nk}
                |   relSem enum {n1,..., nk} with order
                |   set {n1,...,nk}
                |   relSem set {n1,...,nk}
                |   relSem set {n1,...,nk} with order
                |   relsem numeric

order         ::= order {ni<nj,..., nk<nm}

unit          ::= unit/unit
                |   %
                |   msec
                |   ...

relSem        ::= decreasing
                |   increasing

contract      ::= contract {constraint1; ...; constraintk ;}

*contractFunc ::= contract {constraint1; ... ; constraintk ;
constFunc1 ;... ;consttFunc ;}

```

```

*constFun      ::= dimName constraintOp function
                |   dimname {aspectFunc1;...;aspectFuncn;aspect1;...;aspectn;}

constraint     ::= dimName constraintOp dimValue
                |   dimName {aspect1 ; ...;aspectn ;}

dimValue      ::= literal unit
                |   literal

literal        ::= n
                |   {n1,...,nk}
                |   number

aspect         ::= percentile percentNum constraintOp dimValue
                |   mean constraintOp dimValue
                |   variance constraintOp dimValue
                |   frequency freqRange constraintOp number %

*aspectFunc    ::= percentile percentNum constraintOp function
                |   mean constraintOp function
                |   variance constraintOp function
                |   frequency freqRangeFunc constraintOp number %

*freqRageFunc  ::= function
                |   lRangeLimit function, dimvalue rRangeLimit
                |   lRangeLimit function, function rRangeLimit
                |   lRangeLimit dimvalue, function rRangeLimit

freqRange      ::= dimValue
                |   lRangeLimit dimValue, dimValue rRangeLimit

*function      ::= var.dimName
                |   function operator function
                |   number

*operator      ::= + | - | * | /

lRangeLimit    ::= ( | [

rRangeLimit    ::= ) | ]

constraintOp   ::= == | >= | <= | < | >

percentNum     ::= 0 | 1 | ... | 99| 100

```


Annexe B

Partie du code de tissage du contrat *TimeOutC* en MTL

```
library AspectTimeOutC;

model source_model : RepositoryModel; // a Repository model
(UML1.4) model contract_model: RepositoryModel; // a Repository
model (UML1.4) model uml          : UML_Utils;

main() : Standard::Void {

    // local variables
    mdrdriver          : MDRDriver::MDRModelManager;
    metamodelFilename : Standard::String;
    inputFile         : Standard::String;
    contractFilename  : Standard::String;
    contractOutputFilename : Standard::String;
    outputFile        : Standard::String;
    thePreconditions  : Precondition;

    // we define some filenames
    metamodelFilename := '../..../UML1.4/MetaModel/xmi_1.2/01-02-15.xml';
    inputFile         := '../..../Models/modelComp.xmi';
    outputFile        := '../..../Models/modelComp_modified.xmi';
    contractFilename  := '../..../Models/TimeOutC.xmi';
    contractOutputFilename := '../..../Models/TimeOutC_modified.xmi';

    // we initialize the MDR driver
    mdrdriver := new MDRDriver::MDRModelManager();
```

```

mdrdriver.init();
'driver ok'.toOut();
// intanciate the model, save result in another file
source_model := mdrdriver.getModelFromXMI (
metamodelFilename,
'UML',          // name of the root package in the Uml1.4 metamodel
'UML1.4_model',
inputFilename,
outputFilename
);
'sourcemodel ok'.toOut();
// intanciate the model, save result in another file
contract_model := mdrdriver.getModelFromXMI (
metamodelFilename,
'UML',          // name of the root package in the Uml1.4 metamodel
'contract_UML1.4_model',
contractFilename,
contractOutputFilename
);

// we initialize the uml utils
uml := new UML_Utils ();
uml.init (source_model);

'contract model connected'.toOut();

thePreconditions := new Precondition();
thePreconditions.run();
'Preconditions!'.toOut();

}

/*****
/*****

class Precondition {
////////////////////
//verification de la structure du composant
////////////////////

run ()
{
    operationSet      : Standard::Set;
    myOperationSet    : Standard::Set;

```

```

myOperation      : source_model::Core::Operation;
myIterator       : Standard::Iterator;

//recherche de l'ensemble des opérations sur
//lesquelles on va appliquer le contrat
operationSet := !source_model::Core::Operation!.allInstances();
JavaCode [BMTL_myOperationSet=new BMTLSet()];
myIterator := operationSet.getNewIterator();
while myIterator.isOn()
{
    myOperation := myIterator.item().oclAsType
        (!source_model::Core::Operation!);
    if uml.IsModelElementTaggedValue (myOperation, 'selected').[=] (true)
    {
        myOperationSet := myOperationSet.including(myOperation);
    }
    myIterator.next();
}
myIterator := myOperationSet.getNewIterator();
while myIterator.isOn()
{
    myOperation := myIterator.item().oclAsType
        (!source_model::Core::Operation!);
    verification(myOperation);
    myIterator.next();
}
}

verification(op:source_model::Core::Operation)
{
    myClass          : source_model::Core::Class;
    myInterface      : source_model::Core::Interface;
    myAbs            : source_model::Package::Abstraction;
    myAbstractionSet : Standard::Set;
    myDependencySet  : Standard::Set;
    myIteratorDep    : Standard::Iterator;
    myIterator       : Standard::Iterator;
    name            : Standard::String;
    result          : Standard::Boolean;

    //op ne doit pas contenir une taggedValue dont la valeur est timeOutC

```

```

if uml.IsModelElementTaggedValue (op, 'timeOutC').[=] (false)
{
  //verification de la structure du composant
  myClass := op.owner;
  if myClass.getType.[=] (!source_model::Core::Class!)
  {
    myAbstractionSet:= myClass.clientDependency;
    myIterator := myAbstractionSet.getNewIterator();
    result:=false;
    while myIterator.isOn()
    {

      myAbs:= myIterator.item();
      myDependencySet := myAbs.supplier;
      myIteratorDep := myDependencySet.getNewIterator();
      while result.[=](false).[and] (myIteratorDep.isOn())
      {
        myInterface:=myIteratorDep.item();
        name:=op.name.oclAsType(!Standard::String!);
        result:=foundOperation(myInterface,name);
        myIteratorDep.next();
      }

      myIterator.next();
    }

    if result.[=](true)
    {
      //search good component
      result:= foundGoodComponent(myClass);
    }
    else
    {
      'error: don t find the interface
        with selected operation'.toOut();
    }
  }
  else
  {
    'Error: the operation s owner is not a class'.toOut();
  }
}

```

```

else
    {
        'Error: operation has yet this contract'.toOut();
    }
}

foundOperation(
    myInterface : source_model::Core::Interface;
    name        : Standard::String
): Standard::Boolean

{
    myIterator  : Standard::Iterator;
    result      : Standard::Boolean;
    myFeature   : source_model::Core::Operation;

    // check the signature of operation
    myIterator := myInterface.feature.getNewIterator();
    result:=false;
    while result.[=](false).[and] (myIterator.isOn())
    {
        //'verif1'.toOut();
        myFeature:= myIterator.item();

        if isNull(myFeature.name).not()

        {
            if myFeature.name.[=] (name.oclAsType(!Standard::String!))
            {
                'operation of interface ok!'.toOut();
                result:= true;
            }
        }
        myIterator.next();
    }
    return result;
}

foundGoodComponent(
    myClass: source_model::Core::Class

```

```

): Standard::Boolean

{
    result                : Standard::Boolean;
    compFound             : Standard::Integer;
    myAssoSet             : Standard::Set;
    myConnectionSet      : Standard::OrderedSet;
    myIterator            : Standard::Iterator;
    myConnectionIterator : Standard::Iterator;

    myAsso                : source_model::Core::Association;
    myConnection          : source_model::Core::Connection;
    myAssoEnd1            : source_model::Core::AssociationEnd;
    myAssoEnd2            : source_model::Core::AssociationEnd;
    myComp                : source_model::Core::Class;
    myPackage             : source_model::Core::Package;

    result:= false;
    compFound:=0;
    myAssoSet:=!source_model::Core::Association!.allInstances();
    myIterator:= myAssoSet.getNewIterator();
    while myIterator.isOn()
    {
        myAsso := myIterator.item();
        myConnectionSet:= myAsso.connection;
        //on ne garde que les asso a deux elements
        if myConnectionSet.size().[=] (2)
        {

            myAssoEnd1:=myConnectionSet.at(1);
            myAssoEnd2:=myConnectionSet.at(2);
            if myAssoEnd1.participant.[=] (myClass)
            {
                //verifier l'aggregation
                myComp:= myAssoEnd2.participant;
                if myComp.oclIsTypeOf(!source_model::Core::Class!)
                {
                    compFound:= compFound.[+](1);
                }
            }
        }
    }
    else
    {

```

```

        if myAssoEnd2.participant.[=] (myClass)
            {
                //verifier l'aggregation
                myComp:= myAssoEnd1.participant;
                if myComp.oclIsTypeOf
                    (!source_model::Core::Class!)
                    {
                        compFound:= compFound.[+](1);
                    }
            }
        }
        myIterator.next();
    }
    if compFound.[=] (0)
        {
            'erreur: composant non trouve'.toOut();
        }
    else
        {
            if compFound.[>] (1)
                {
                    'erreur: une seule composition possible'.toOut();
                }
            else
                {
                    //verification: nom du package = nom du composant
                    myPackage:=comp.container;
                }
        }
    return (result);
}
}
}

```


Annexe C

Code du tissage du contrat *TimeOutC* en python

```
""" This transformation add a contract 'TimeOutC' to the selected
operation "op" of the class "c1" which implements the interface
"i1" of the component "comp"
```

Precondition of the tranformation

In OCL read file "oclConstraint.txt"

```
- the selected behavior of operation "op" is included in a Class
(let C1 this class) - c1 implements an interface (let I1 this
interface) - verify that op is also defined in the interface I1 -
Check if c1 is an aggregation of a Component (let "comp" this
component)
- Verify that "comp" inherit from system.ComponentModel.Component
of the C# Lib
```

```
- Check the existance of the contract's Package and import it in
the model
```

Then it's possible to apply the transformation

```
"""
```

```
from pyKase import * from xuml import * from pythonkase.umlhelper
import * #import pythonkase.umlmerge as umlmerge import timeOut
#import the package timeOut
```

```

# Adding TimeOut contract to the operation named 'method' from the
'class1' class # which realizes the 'interface' interface of the
'component' component.

class TimeOutAspectdef:
    def __init__(self):
        print "adding TimeOutContract ..."
        self.doc=Document()

    def run(self):
        """ This transformation attach a contract to a method of a class which
        implement an interface of a component
        """

        ##### PRECONDITIONS #####

        #Test of preconditions
        # - selection not empty
        selection=self.doc.getSelection()
        if len(selection) == 0 or len(selection) > 1:
            showMessage( "You must select an and only one operation in a class !",
                ["Dismiss"], "QCCS TimeOutContract", 0, -1, Warning )
            return

        # - only one element is selected
        method = selection[0].toOperation()
        if method.isNull() :
            showMessage( "The selected element isn't an operation", ["Dismiss"],
                "QCCS TimeOutContract", 0, -1, Warning )
            return

        # find component, interface and class
        class1=method.owner().toClass()
        if class1.isNull():
            showMessage( "The operation's owner is not a class", ["Dismiss"],
                "QCCS TimeOutContract", 0, -1, Warning )
            return

        # check whether there is a dependency between class1 and an interface
        selDep=class1.clientDependency()
        if len(selDep)==0 :

```

```

        showMessage( "Your class must be a realization of an interface",
["Dismiss"], "QCCS TimeOutContract", 0, -1, Warning )
        return

# search the good interface
# search for the interface which has the operation with the same
# signature as 'method'
interf = None
for d in selDep:
    interfa = d.supplier().toInterface()
    if not interfa.isNull():
        for m in interfa.feature():
            if m.isOperation() and
                m.toOperation().isSameSignature(method):
                    interf=interfa
if interf == None:
    showMessage( "Don't find the interface with selected operation",
["Dismiss"], "QCCS TimeOutContract", 0, -1, Warning )
    return

#search the good component
#get list of AssociationEnd
comp=class1.getTHEComposite()

# check that PackageName == ComponentName
compnamespace=class1.namespace()
pack=compnamespace.toPackage()
if pack.isNull():
    showMessage( "don't find the package", ["Dismiss"],
"QCCS TimeOutContract", 0, -1, Warning )
    return

if comp.name() != pack.name():
    showMessage("don't find the package", ["Dismiss"],
"QCCS TimeOutContract", 0, -1, Warning )
    return

# Check that comp is a SCMC's class: verify the inheritance
# ie. inherits from system.ComponentModel.Component of Csharp
listG=comp.generalization()
if len(listG)!= 1:
    showMessage( "Error in PSM: you must have one and only one
inheritance", ["Dismiss"], "QCCS TimeOutContract",
0, -1, Warning )

```

```

    return
if listG[0].parent().qualifiedName() !=
    "System::ComponentModel::Component":
    showMessage( "Error : bad inheritance", ["Dismiss"],
        "QCCS TimeOutContract", 0, -1, Warning )
    return

#####      END OF PRECONDITIONS      #####

#verify if operation has this contract
if method.hasContractTag("timeOut"):
    showMessage( "operation has yet this contract", ["Dismiss"],
        "QCCS TimeOutContract", 0, -1, Warning )
    return

parentPack=pack.namespace()
if parentPack.isNull():
    showMessage( "don't find the namespace", ["Dismiss"],
        "QCCS TimeOutContract", 0, -1, Warning )
    return

# verify if the package timeOutAspect is still in the package pack
# if not, add it
listPack=parentPack.getAllModelElements("timeOut", "Package")
if len(listPack)==0 : timeOut.TimeOutAspect().transform(parentPack)

# search for the contract in the package named 'TimeOutC'
listcontract=parentPack.getAllModelElements("TimeOutC", "Class")
if len(listcontract)!= 1:
    showMessage( "Error : class TimeOutC not import", ["Dismiss"],
        "QCCS TimeOutContract", 0, -1, Warning )
    return
contract = listcontract[0].toClass()

# Add association between component and contract
if not (comp.isAssociate(contract)):
    comp.addAssociation(contract,
        multiplicity1=["(1,*")],multiplicity2=["(1,*")])

# search Mydelegate
listDelegate=parentPack.getAllModelElements("MyDelegate", "Class")
if len(listDelegate)== 0:

```

```

        showMessage( "Error : class MyDelegate not import",
                    ["Dismiss"], "QCCS TimeOutContract", 0, -1, Warning )
        return
MyDelegate = listDelegate[0].toClass()

# search for an Attribute named 'BeginMyContract',
# and add it if not found

listAttribute=class1.getAllModelElements("BeginMyContract", "Attribute")
alist = []
for f in listAttribute :
    if f.toAttribute().type()==MyDelegate:
        alist.append(f)
longB = len(alist)
if longB > 1 :
    showMessage( "Error : multiple MyDelegate BeginMyContract !",
                ["Dismiss"], "QCCS TimeOutContract", 0, -1, Warning )
    return
elif longB == 0 : # add BeginMyContract
    class1.addAttribute(name="BeginMyContract",type=MyDelegate)

listAttribute=class1.getModelElements("EndMyContract","Attribute")
alist = []
for f in listAttribute :
    if f.toAttribute().type()==MyDelegate:
        alist.append(f)
longE = len(alist)
if longE > 1 :
    showMessage( "Error : multiple MyDelegate EndMyContract !",
                ["Dismiss"], "QCCS TimeOutContract", 0, -1, Warning )
    return
elif longE == 0 : # add BeginMyContract
    class1.addAttribute(name="EndMyContract",type=MyDelegate)

if not class1.hasContractTag("timeOut"):
    #add a constructor of the class
    # check existence of constructors and create if not
    #if constructor exists rename it:
    listCons=class1.getModelElements(class1.name(),"Operation")
    listOldCons=[]
    listNewCons=[]
    if len(listCons)==0 : # there isn't constructor
        # So we create one (cons)
        cons=class1.addNewOperation(name=class1.name())

```

```

        # and had it to the constructor list
        listNewCons.append(cons)
    else: #there is any cons
        oldName="old_"+class1.name()
        # get the list of old constructors
        listOldCons=class1.getModelElements(oldName,"Operation")
        cons1=[]
        for op in listCons :
            op=op.toOperation()
            for oldOp in listOldCons:
                oldOp=oldOp.toOperation()
                if op.hasSameParameters(oldOp):
                    cons1.append(op)
                    listNewCons.append(op)
        if cons1 != []:
            for c in cons1:
                listCons.remove(c)

    for op in listCons:
        # il ne reste dans la liste que des constructeurs sans old
        listParam=[]
        op=op.toOperation()
        op.setName(oldName)
        op.setVisibility("private")
        cons=class1.addNewOperation(name=class1.name())
        listParam=op.parameter()
        for p in listParam:
            cons.appendParameter(p)
        listOldCons.append(op)
        listNewCons.append(cons)

#rename op and creation of new operation
oldName="old_"+method.name()
listOp=class1.getModelElements(oldName,"Operation")
alist=[]
for op in listOp :
    op=op.toOperation()
    if not(op.isNull()):
        if op.hasSameParameters(method):
            alist.append(op)
if len(alist)==0:
    method.setVisibility("private")

```

```

        opName=method.name()
        method.setName(oldName)
        newOp=method
        method=class1.addNewOperation(name=opName)
        for p in newOp.parameter():
            method.appendParameter(p)

else :
    newOp=alist[0]
method.addContractTag("timeOut")

# add the c# code to link methods of contracts and events

# test existence of a composite state associated with cons...
# if this composite state exists:
# rename it and add it in a new composite state

if not class1.hasContractTag("timeOut"):
    class1.addContractTag(contractName="timeOut")
    li=class1.getContractTagList()
    for cons in listNewCons:
        #creation of composite state
        cs, beh=cons.encapsCompositeState()
        pack.appendOwnedElement(cs)
        script="// code C#\nthis.BeginMyContract +=
            new MyDelegate ("+"pack.name()+"."+contract.name()+".start);
            \nthis.EndMyContract+= new MyDelegate
            ("+"pack.name()+"."+contract.name()+".isValid);"
        cs.appendUninterpretedAction(script)
        if beh!=None:
            cs.appendStateVertex(beh)
        else:
            listOldCons=[]
            oldName="old_"+class1.name()
            listOldCons=class1.getModelElements(oldName,"Operation")
            for oldOp in listOldCons:
                oldOp=oldOp.toOperation()
                if oldOp.hasSameParameters(cons):
                    cs.appendCallAction(oldOp)
                    break

    cs.appendFinalState()

```

```
#add the c# code in operation to activate automatically the events
#creation of an compositeState
csOp, oldBeh=method.encapsCompositeState()
pack.appendOwnedElement(csOp)
if longB==0:
    scriptOp1="// code C#\nBeginMyContract();"
    csOp.appendUninterpretedAction(scriptOp1)
    if oldBeh==None:
        csOp.appendCallAction(newOp)
    else:
        csOp.appendStateVertex(oldBeh)
else:
    if oldBeh==None:
        csOp.appendCallAction(newOp)
    else:
        csOp.appendStateVertex(oldBeh)
scriptOp2="// code C#\nEndMyContract();"
csOp.appendUninterpretedAction(scriptOp2)
csOp.appendFinalState()

return
```

Annexe D

code généré par *Kase* pour l'exemple du contrat *TimeOutC*

```
using System.ComponentModel; using gps.timeOut;

namespace gps.LocationComputer {
    //447
    public class LocationComputer : System.ComponentModel.Component
    {
        public System.Collections.ArrayList m_timeOutC =
            new System.Collections.ArrayList();

        public void AddTimeOutC( gps.timeOut.TimeOutC arg ) {
            if ( arg == null ) return;
            if ( m_timeOutC.Contains( arg ) ) return;
            m_timeOutC.Add( arg );
            arg.AddLocationComputer( this );
        }

        public void RemoveTimeOutC( gps.timeOut.TimeOutC arg ) {
            if ( arg == null ) return;
            if ( !m_timeOutC.Contains( arg ) ) return;
            m_timeOutC.Remove( arg );
            arg.RemoveLocationComputer( this );
        }

    }

    public class 3DPoint
    {
    }
}
```

```
public class computerImpl : ComputerI, decoderI, DataI,
    PowerManagementI, ClockI
{

    private 3DPoint position;
    private double EPE;
    private double speed;
    private double heading;
    private Precision precision;
    public gps.timeOut.MyDelegate BeginMyContract;
    public gps.timeOut.MyDelegate EndMyContract;

    public virtual void setEstimatePosition(3DPoint point)
    {
        //+C User defined code starts here
        //-C End of user defined code
    }

    public virtual void configureDecoders()
    {
        //+C User defined code starts here
        //-C End of user defined code
    }

    private virtual 3DPoint old_getPosition()
    {
        //+C User defined code starts here
        //-C End of user defined code
    }

    public computerImpl()
    {
        //+C User defined code starts here
        // code of composite state
        // code C#
        this.BeginMyContract +=
            new MyDelegate (LocationComputer.TimeOutC.start);
        this.EndMyContract+=
            new MyDelegate (LocationComputer.TimeOutC.isValid);
        // end of code of composite state
        //-C End of user defined code
    }
}
```

```
public virtual 3DPoint getPosition()
{
    //+C User defined code starts here
    // code of composite state
    // code C#
    BeginMyContract();
    old_getPosition();
    // code C#
    EndMyContract();
    // end of code of composite state
    //-C End of user defined code
}
}
}
```


Annexe E

Transformation en PrologIV

```
00- %% CONTRACTUAL DEPENDENCIES
01- >> receiver( ThetaR, ThetaS) :-
02-     ThetaR ~ ThetaS + 2.
03-
04- >> decoder( ThetaD, ThetaS) :-
05-     receiver( ThetaR1,ThetaS),
06-     receiver( ThetaR2, ThetaS),...,
07-     max( X, [ThetaR1, ThetaR2,...] ),
08-     ThetaD ~ X + 3.
09-
10- >> computer( ThetaC, Epe, P, ThetaD, Nbr) :-
11-     ThetaC ~ ThetaD + Nbr * log( Nbr ),
12-     P ~ Nbr * 3,
13-     rule( Epe, P, Nbr).
14-
15- >> rule( medium, P, 3) :- P =< 25.
16- >> rule( low, P, 3) :- P > 25.
17- >> rule( high, P, 5) :- P =< 24.
18- >> rule( medium, P, 5) :- P ~ oc(24,30).
19- >> rule( low, P, 5) :- P > 30.
20- >> rule( high, P, 12) :- P =< 32.
21- >> rule( medium, P, 12) :- P ~ oc(32,45).
22- >> rule( low, P, 12) :- P > 45.
23-
24- %% PROPAGATION OF VALIDITY DOMAINS
25- >> P =< 15, Epe = high, ThetaS ~ cc(15,30), in( Nbr, [3,5,12]),
26-     decoder( ThetaD, ThetaS),
27-     computer( ThetaC, Epe, P,ThetaD, Nbr).
```


Bibliographie

- [1] PrologIV : reference manual and user's guide. Technical report, PrologIA, 1994.
- [2] Terms and definitions related to quality of service and network performance including dependability. Technical Report Recommendation E.800(08/94), CCITT/ITU, 1998.
- [3] OMG unified modeling language specification, v 1.3. Technical report, OMG Organization, 1999.
- [4] OMG unified modeling language, v 2.0. Technical Report v0.671, OMG Organization, 2002.
- [5] J Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.
- [6] G. Amorós, W. Paredes, and A-M. Sassen. D5.1.1 : Documentation on application design of workflow system. Technical Report D511, The QCCS consortium, 2002.
- [7] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, Y. Watanabe, and G. Yang. Concurrent execution semantics and sequential simulation algorithms for the metropolis meta-model. In *the Tenth International Symposium on Hardware/Software Codesign*, Colorado, April 2002.
- [8] C. Becker and K. Geihs. MAQS : management for adaptive QoS-enabled services. In *Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, 1997.
- [9] L. Berger. Interactions et modèles de programmation : Support des interactions par les modèles à objets et à composants. *L'Objet*, 8(3) :9–38, 2002.
- [10] A. Beugnard, J.M. Jezequel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, pages 38–45, July 1999.
- [11] G. Blair, L. Blair, and J-B. Stefani. A specification architecture for multimedia systems in open distributed processing. *Computer Networks and ISDN Systems*, 29 :473–500, 1997.
- [12] B. Boehm. Managing software productivity and reuse. *IEEE Computer*, 32(9) :111–113, September 1999.
- [13] G. Booch. *Software Components with Ada : Structures, Tools, and Subsystems*. Benjamin-Cummins, 1987.
- [14] M.C. Brown. *Python : The Complete Reference*. Osborne/McGraw-Hill, 2001.

- [15] J. Bézivin. From object composition to model transformation with the MDA. In *TOOLS'USA*, volume 39, Santa Barbara, August 2001.
- [16] J. Bézivin, Jézéquel J.M Facet, N., B. Langlois, and D. Pollet. Reflective model driven engineering. In G. Booch P. Stevens, J. Whittle, editor, *Proceedings of UML 2003*, volume 2863 of *LNCS*, pages 175–189, San Francisco, October 2003. Springer.
- [17] Cilabs. Component integration laboratories. Technical report, <http://www.cilabs.org>.
- [18] Telecommunications Information Networking Architecture Consortium. Telecommunications information networking architecture. Technical report, <http://www.tinac.com/index.htm>.
- [19] Microsoft Corporation. Microsoft active server pages .net. Technical report, <http://www.asp.net>.
- [20] Microsoft Corporation. Microsoft .net. Technical report, <http://www.microsoft.com/net>.
- [21] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11) :1268–1287, November 1988.
- [22] O.J. Dahl and K. Nygaard. SIMULA : an ALGOL-based simulation language. *Communications of the ACM*, 9(9) :671–678, September 1966.
- [23] P. Donth. D5.1.2 : Documentation on application design of e-commerce system. Technical Report D512, The QCCS consortium, 2002.
- [24] D. D'Souza and A.C. Wills. *Objects, Components, and Frameworks : The Catalysis Approach*. Addison-Wesley, 1999.
- [25] T. Faison. *Component-Based Development with Visual C#*. John Wiley & Sons, INC., 2002.
- [26] S. Frolund and J. Koistinen. Quality of service specification in distributed object systems. Technical Report HPL-98-159, Software Technology Laboratory, Hewlett-Packard Company, 1997.
- [27] S. Frolund and J. Koistinen. Qml : A language for quality of service specification. Technical Report HPL-98-10, Software Technology Laboratory, Hewlett-Packard Company, 1998.
- [28] A. Gacemi and D. Seriai. La réutilisation : concepts et techniques. Technical Report 2003-4-2.
- [29] D. Gangopadhyay, R. Helm, and I.M. Holland. Contracts : Specifying behavioral compositions in object-oriented systems. In *ECOOP/OPSLA'90 Proceedings*, 1990.
- [30] J. Grabowski, P. Graubmann, and E. Rudolph. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12) :1629–1641, 1996.
- [31] J. Grabowski, P. Graubmann, and E. Rudolph. Towards a harmonization of UML-Sequence Diagrams and MSC. In R. Dsoulli, G. von Bochmann, and Y. Lahav,

- editors, *SDL'99 : The Next Millennium, Proceedings of the 9th SDL Forum*, Montreal, Canada, jun 1999. Elsevier Science Publishers.
- [32] Object Management Group. Common object request broker architecture. Technical report, <http://www.corba.org>.
- [33] Object Management Group. Corba component model. Technical report, <http://www.omg.org/technology/documents/formal/components.htm>.
- [34] Object Management Group. Object management group. Technical report, <http://www.omg.org>.
- [35] Object Management Group. Omg model driven architecture. Technical report, <http://www.omg.org/mda>.
- [36] G. Heineman and W. Councill. *Component-based software engineering : putting the pieces together*. Addison Wesley, 2001.
- [37] E. Hindin. Say what? QoS in english. *Network World*, <http://www.nwfusion.com/netresources/0817qos.html>, 1998.
- [38] Sun Microsystems Inc. Sun microsystems enterprise java beans. Technical report, <http://java.sun.com/products/ejb>.
- [39] Sun Microsystems Inc. Sun microsystems java 2 enterprise edition. Technical report, <http://java.sun.com/j2ee>.
- [40] J. Jaffar and J.L Lassez. Constraint logic programming. In ACM, editor, *14th ACM symposium on principles of programming languages(POPL'87)*, pages 111–119, 1987.
- [41] F. Jean-francois. Architectures distribuées et serveurs d'application. *WEBATRIUM*, <http://www.webatrium.com>, 2001.
- [42] J.M. Jezequel, S. Lorcy, and N. Plouzeau. Un patron pour la gestion de la qualite de services d'applications reparties. *L'Objet*, 3(6), 1998.
- [43] G. Kizales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In Springer Verlag, editor, *Proceeding of ECOOP'97*, number 1241 in LNCS, pages 220–242, Jyväskylä (Finlande), 1997.
- [44] A.G. Kleppe and J. Warmer. *The Object Constraint Language : Precise Modeling With UML*. october 1998.
- [45] J. Koistinen. Dimensions for reliability contracts in distributed object systems. Technical Report HPL-97-119, Software Technology Laboratory, Hewlett-Packard Company, 1997.
- [46] I. Kurtev and K. van den Berg. Model driven architecture based xml processing. In *Proceedings of the 2003 ACM symposium on Document engineering*, pages 246–248. ACM Press, 2003.
- [47] L. Lamport. The temporal logic of actions. In *ACM Transactions on Programming Languages and systems*, volume 16, pages 872–923, 1994.
- [48] P. Leydekkers and V. Gay. ODP view on quality of service for open distributed multimedia environments. In *Proceedings International Workshop on QoS (IW-QoS)*, Paris (France), March 1995.

- [49] W .C Lim. Effects of reuse on quality, productivity, and economics. In *IEEE Software*, volume 11, pages 23–30, September 1994.
- [50] JP. Loyall, RE. Schantz, JA. Zinky, and DE Bakken. Specifying and measuring quality of service in distributed object systems. In *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*, Kyoto, Japon, 1998.
- [51] R. Marvie, P. Merle, and Geib J-M. Towards a dynamic corba component platform. In *the Second International Symposium on Distributed Object Applications (DOA'00)*, pages 305–314, Belgium, September 2000.
- [52] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
- [53] S. Mauw and M. A. Reniers. High-level message sequence charts. In *Proceedings of the Eighth SDL Forum (SDL'97)*, pages 291–306, 1997.
- [54] M. D. McIlroy. Mass-produced software components. In P. Naur J.M. Buxton and B. Randell, editors, *Software Engineering Concepts and techniques : NATO conference of Software Engineering*, pages 138–155, 1968.
- [55] B. Meyer. Applying "design by contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10) :40–52, october 1992.
- [56] B. Meyer. .NET is coming. *IEEE Computer*, pages 92–97, August 2001.
- [57] T. Michel. Synchronized multimedia integration language. Technical report, W3C, <http://www.w3.org/AudioVideo>.
- [58] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects : Survival Guide*. John Wiley & Sons, INC., 1996.
- [59] PP. Pal, JP. Loyall, RE. Schantz, JA. Zinky, R. Shapiro, and J. Megquier. Using QDL to specify QoS aware distributed (QuO) application configuration. In Proceedings of ISORC 2000, editor, *The Third IEEE International Symposium on Object-Oriented Real-time Distributed Computing*, Newport Beach, CA, March 2000.
- [60] G. Papadopoulos, A. Chimaris, O. Papapetrou, A. Zographos, C. Tsaggaridou, D. Vogiatzis, and M. Kounnapi. D5.1.3 : Documentation on application design of the tele-medecine system. Technical Report D513, The QCCS consortium, 2003.
- [61] R. Pawlak. *La Programmation Orientée Aspect Interactionnelle Pour La Construction d'Applications à Préoccupations Multiples*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, 2003.
- [62] Project QCCS. Quality controlled component-based software development. Technical report, <http://www.qccs.org>.
- [63] A-M. Sassen, G. Amorós, T. Weis, K. Geihs, N. Plouzeau, K. Macedo de Amorim, J-M. Jézéquel, P. Donth, O. Papapetrou, and G. Papadopoulos. Qccs quality controlled component-based software development : Final report. Technical report, The QCCS consortium, 2003.

- [64] C. Szyperski. *Component Software : Beyond Object-Oriented Programming, Second Edition*. Addison-Wesley, 2002.
- [65] Octo Technology. Architecture d'application : la solution .NET. http://www.dotnetguru.org/downloads/wp_dotnet.pdf, 2003.
- [66] Equipe Triskell-IRISA. Model transformation. Technical report, http://modelware.inria.fr/article.php3?id_article=15.
- [67] J. Viega and J. Voas. Can aspect-oriented programming lead to more reliable software? In Jeffrey Voas, editor, *IEEE Software*, volume 17, pages 19–21, 2000.
- [68] A. Vogel, B. Kerhervé, G. Bochmann, and J. Gecsei. Distributed multimedia and QoS - A survey. In *IEEE Multimedia*, volume 2, pages 10–19, May 1995.

Liste des acronymes

ADO.NET : Active Data Objects
AOP : aspect-oriented programming
API : Application Programming Interface
ASP.NET : Active Server Pages .NET
CBSE : Component Based Software Engineering
CCM : CORBA Component Model
CIDL : Component Implementation description Language
CIF : Component Implementation Framework
CI Labs : Component Integration Laboratories
CLP : Constraint Logic Programming
CLR : Common Language Runtime
CLS : Common Language Specification
COM : Component Object Model
COM+ : Component Object Model +
CORBA : Common Object Request Broker Architecture
CTS : Common Type System
CWM : Common Warehouse Metamodel
DCOM : Distributed Component Object Model
DNA : Digital Network Architecture
EJB : Enterprise Java Bean
GC : Garbage Collector
GPS : Global Positioning System
HMSC : High-level Message Sequence Charts
IDL : Interface Description Language
ITU : International Telecommunications Union
J2EE : Java2 Enterprise Edition
J2ME : Java2 Micro Edition
J2SE : Java2 Standard Edition
JDBC :Java DataBase Connectivity
JMS : Java Message Service
JNDI : Java Naming and Directory Interface
JSP : Java Server Pages
JTA : Java Transaction API
JVM : Java Virtual Machine
MAQS : Management for Adaptative QoS-enabled Services

MDA : Model Driven Architecture
MOF : Meta Object Facility
MSIL : Microsoft Intermediate Language
MTL : Modeling Transformation Language
MTS : Microsoft Transaction Services
MTTF : Mean Time To Failure
MTTR : Mean Time To Repair
NOF : Number Of Failures
OCL : Object Constraint Language
ODP : Open Distributed Processing
OMA : Object Management Architecture
OMG : Object Management Group
ORB : Object Request Broker
PIM : Platform Independent Model
POA : Portable Object Adaptater
PSM : Platform Specific Model
QCCS : Quality Controlled Component-based Software development
QDL : QoS Definition Language
QML : Quality of service Modeling Language
QdS : Qualité de Service
QoSCL : Quality of Service Constraint Language
QTL : Quality of service Temporal Logic
QuO : Quality of service for Objects
RMI : Remote Methods Invocation
RM-ODP : Reference Model of Open Distributed Processing
SMIL : Synchronized Multimedia Integration Language
SOAP : Simple Object Access Protocol
SQL : Structured Query Language
TLA : Temporal Logic of Actions
TTR : Time to repair
UML : Unified Modeling Language
XML : eXtensible Markup Language

Résumé

La réutilisation de composants logiciels permet de produire un logiciel en optimisant son coût global de production. Un composant logiciel est défini comme une unité de composition qui comporte des interfaces ainsi que des dépendances contextuelles. Aujourd'hui, le choix d'un composant se porte sur les services qu'ils offrent mais aussi et surtout pour sa qualité de réalisation qui est définie par les propriétés extra-fonctionnelles du composant.

Ces propriétés appelées aussi propriétés de qualité de service (QdS) vont guider le choix du développeur : il faut donc les prendre en compte dès la phase de conception. Le concepteur doit spécifier la qualité de service souhaitée, et concevoir la gestion de cette qualité c'est-à-dire surveiller la qualité effectivement réalisée et permettre la négociation de contrats.

Le concept de contrat de qualité de service va constituer le concept fondamental autour duquel vont s'articuler les solutions proposées. Les contrats spécifient les droits et les obligations entre un client et un fournisseur de service. La notion de contrat va permettre à l'application d'avoir un retour d'informations et de pouvoir ainsi réagir à son environnement par le mécanisme de renégociation.

La contribution de cette thèse est de proposer, dans le cadre du projet européen QCCS, une méthodologie pour la spécification des contrats de qualité de service de composants logiciels, et leur intégration dans un environnement de développement par composants, en s'appuyant sur la conception et la réalisation par aspects (*aspect-oriented programming* ou *aop*). Dans un premier temps, la notation UML a été étendue pour offrir aux concepteurs de logiciels à base de composants un outil de spécification et de conception traitant la qualité de service. Une fois le modèle de composants contractualisés créé, le concepteur a la possibilité de générer automatiquement ces composants avec une mise en place d'un système de surveillance pour s'assurer de la qualité effectivement réalisée, ainsi qu'un mécanisme de renégociation des contrats.

abstract

Software components reuse reduces the global development costs. A software component can be defined as a composition unit equipped with interfaces and contextual dependencies. Nowadays, the choice of a peculiar software component depends on the services provided, but also on the quality of its realisation, which is defined by non-functionnal properties.

These properties are also called quality of service (QoS) properties. As they will guide the choice of developpers, they must be considered from the beginning of the conception phase. Developers must define a desired quality of service, and design quality management policies, ie control the quality effectively provided by a component, and allow contract negotiations. The concept of "quality of service contract" will be the fundamental concept for the solutions proposed in this thesis.

Contracts specify rights and obligations for a client and a service provider. The notion of contract will provide feedback to an application hence allowing it to react to its environment through renegotiation mechanisms.

This thesis was realized within the framework of the european project QCCS. Its main contribution is a QoS contracts specification methodology for components, and its integration within a component-based developement environment relying on aspect-oriented conception and realisation.

First, UML notation has been extended to provide component-based software designers with a conception and specification tool dealing with quality of services. Then, when a contractualized components model is created, designers can generate automatically these components equipped with a monitoring systems that checks if a quality of service is effectively reached, and with a contract renegotiation mechanism.